I still know it's you! On Challenges in Anonymizing Source Code

Micha Horlboge
Technische Universität Berlin

Erwin Quiring
ICSI & Ruhr Universität Bochum

Roland Meyer
Technische Universität Braunschweig

Konrad Rieck
BIFOLD & Technische Universität Berlin

ABSTRACT

The source code of a program not only defines its semantics but also contains subtle clues that can identify its author. Several studies have shown that these clues can be automatically extracted using machine learning and allow for determining a program's author among hundreds of programmers. This attribution poses a significant threat to developers of anti-censorship and privacy-enhancing technologies, as they become identifiable and may be prosecuted. An ideal protection from this threat would be the *anonymization of source code*. However, neither theoretical nor practical principles of such an anonymization have been explored so far. In this paper, we tackle this problem and develop a framework for reasoning about code anonymization. We prove that the task of generating a *k-anonymous program*—a program that cannot be attributed to one of k authors—is not computable in the general case. As a remedy, we introduce a relaxed concept called *k-uncertainty*, which enables us to measure the protection of developers. Based on this concept, we empirically study candidate techniques for anonymization, such as code normalization, coding style imitation, and code obfuscation. We find that none of the techniques provides sufficient protection when the attacker is aware of the anonymization. While we observe a notable reduction in attribution performance on real-world code, a reliable protection is not achieved for all developers. We conclude that code anonymization is a hard problem that requires further attention from the research community.

KEYWORDS

Authorship Attribution, Code Obfuscation, Machine Learning

1 INTRODUCTION

The source code of a program provides a wealth of information for analysis. It not only defines syntax and semantics, but also contains clues suitable for identifying its author. These clues result from the personal *coding style* and range from obvious programming habits, such as the naming of variables and functions, to subtle preferences in the usage of data types, control structures, and API [50]. Thus, similar to writing style in literature, a source code unnoticeably carries a fingerprint of its developer. Several studies have shown that this coding style can be automatically extracted using machine learning and allows for identifying the author of a program among hundreds of other developers [e.g., 5, 8, 11, 30]. As an example, Abuhamad et al. [2] report a detection accuracy of 96% on a dataset of source code from 1,600 developers participating in a coding competition. Although these methods still struggle under realistic conditions, some approaches reach up to 61% on only fragments of real-world code from 104 programmers [18].

While authorship attribution of source code resembles a valuable tool for digital forensics, it also poses a threat to developers of anti-censorship and privacy-enhancing technologies. Anonymous contributors to open-source projects, such as Tor [20] and I2P [45], become identifiable through learning-based attribution and might be prosecuted for their work in repressive countries. Unfortunately, defenses against this threat have received little attention so far. Even worse, prior work has shown that strong obfuscation of source code is still not sufficient to prevent an attribution [see 2, 11, 12], indicating the challenge of protecting developers.

In this paper, we tackle this problem and study the *anonymization of source code* from a theoretical and practical perspective. To this end, we propose a framework for reasoning about code anonymization and attribution. Based on this framework, we introduce the concept of a *k-anonymous* program, that is, a program that cannot be attributed to one of k authors and hence is protected by an anonymity set. We prove that changing a given source code, so that it becomes k-anonymous in the general case is unfortunately not computable and resembles an undecidable problem. Consequently, a universal method for code anonymization cannot exist and so the search of practical protection is a challenge for research.

As a remedy, we derive a relaxed concept that we denote as *k-uncertainty*. Instead of a program being perfectly indistinguishable between authors, we require that it is attributed to k authors with similar confidence. While this concept cannot overcome the undecidability of k-anonymity, it provides a novel means for *measuring* the protection of developers. By inspecting the confidence range of the k most similar authors in an attribution, we can evaluate how well a developer is hidden in an anonymity set. Based on this concept, we introduce a numerical measure called *uncertainty score* that ranges from 0 (no protection) to 1 (k-anonymity) and can be used to empirically assess how well a source code is protected. As a result, it becomes possible to empirically compare techniques for protecting the identity of developers.

Based on this numerical measure, we conduct a series of experiments to analyze candidate techniques for code anonymization. In particular, we consider *code normalization*, *coding style imitation* [47] and *code obfuscation* [16, 53] as defenses against popular attribution methods [2, 11]. For our experiments, we work with two datasets, one from a programming competition with 30 developers and the other from open-source projects of 81 developers. At first, all techniques hinder an attribution and lead to high uncertainty scores. However, their performances diminishes once the attacker becomes aware of the protection. For the strongest technique, the

Proceedings on Privacy Enhancing Technologies 2024(3), 744–760

https://doi.org/10.56553/popets-2024-0102

popular obfuscator Tigress [16], the attribution still reaches an accuracy up to 24% and 8%, respectively. To understand this result, we develop a method for explaining the attributions and uncover clues in the source code that remain after anonymization. This explanation method complements our uncertainty score by indicating weak spots in the realized protection.

When iteratively removing clues with our method from the competition dataset, we eventually bring the source code to an uncertainty score close to 1. However, this result should not be interpreted as a defeat of the attribution methods. Rather, it shows that anonymization can be achieved in a limited and controlled setup. The systematic removal of clues, however, cannot simply be transferred to the real world, where neither the attribution method nor the learning data is known to a developer. We thus argue that there is a need for novel anonymization concepts and consider our work as the first step towards formalizing and evaluating these approaches. In summary, we make the following contributions:

- *Theoretical view on code anonymization.* We propose a framework for reasoning about code anonymization. This enables us to prove that universal k-anonymity cannot be reached.
- *Practical view on code anonymization.* We introduce a concept for measuring anonymization. Based on this, we empirically compare protection techniques under different adversaries.
- *Insights on obstacles of code anonymization.* Finally, we develop an approach for explaining attribution methods and identifying clues remaining after an anonymization attempt.

Roadmap. We review authorship attribution of source code in Section 2 and discuss our threat model in Section 3. Our framework for analyzing code anonymity is introduced in Section 4 and we empirically evaluate different techniques with it in Section 5. We analyze the deficits of the techniques in Section 6. Limitations and related work are presented in Section 7 and Section 8, respectively. Section 9 concludes the paper.

2 SOURCE CODE ATTRIBUTION

We start with a short primer on authorship attribution of source code. The objective of this task is to automatically attribute a given source code to its author based on individual properties of coding style [30]. As these properties are hard to formally characterize, this objective is typically achieved by extracting *features* from source code and constructing an attribution method using *machine learning*. Existing approaches are hence best described based on the considered features and the employed learning algorithms.

2.1 Features of Source Code

The features currently used in authorship attribution roughly fall into three types: layout, lexical, and syntactic. Each of them relies on a different representation and thus provides different types of stylistic patterns. We briefly review these features in the following.

2.1.1 Layout Features. The first type of features are derived from the layout of the source code. Developers often have specific preferences to format their code, such as different forms of indentation. Figure 1 shows a function, where different features are highlighted. Even in this short snippet a variety of layout features is visible,

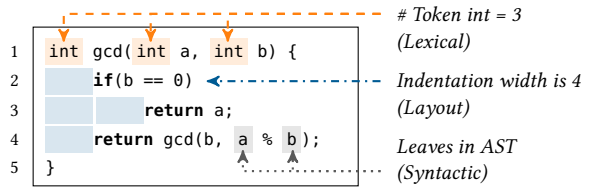


Figure 1: Code snippet in C with highlighted feature types.

such as the indentation width of 4. Consequently, any attempt to anonymize code needs to start by removing individual formatting.

2.1.2 Lexical Features. The second type of features is derived from the lexical analysis of source code. The resulting features comprise identifiers, keywords, literals, operators, and other terminal symbols of the underlying grammar [4]. These features implicitly encode the syntax and semantics of the source code. For example, Figure 1 shows a lexical feature that counts the occurrence of the token `int`. This reflects a developer's preference for this type in relation to others, such as `long` or `int32_t`. Compared to the layout, lexical features cannot be unified easily, as they are implicitly linked to syntax and semantics.

2.1.3 Syntactic Features. Finally, the syntax of source code provides further features for characterizing the programming habits of developers. In particular, the *abstract syntax tree* (AST) is a common representation that allows extracting patterns in the types, arithmetics, logic, and control flow used by developers [6, 8, 11]. These features range from single language constructs to syntactic fragments, such as tree-like structures in the AST. Figure 1 highlights two code locations that correspond to leaves in the AST. Syntactic features are hard to modify. Replacing a single keyword in the source code may lead to several modifications in the AST. Similarly, adapting one node of the tree may require multiple code modifications. The removal of coding style thus becomes challenging, as we demonstrate in Section 5.

2.2 Attribution Using Machine Learning

The described features provide a complex view on source code that is difficult to interpret by a human analyst. State-of-the-art attribution methods for source code therefore rely on machine learning to recognize stylistic patterns for a particular author [e.g., 2, 6, 11, 46]. To this end, a supervised learning algorithm is applied to infer characteristics for each author. The result is a *multiclass classifier* that returns confidences for all authors from the training data. The highest-ranked author is typically selected for attribution.

Previous work has studied several learning algorithms for this attribution, such as support vector machines [46], random forests [11], and deep neural networks [2, 6]. As we find in our evaluation, the algorithm can have a considerable impact on anonymization. For example, deep neural networks tend to overfit to particular authors, while other algorithms provide a more generalized prediction.

3 MOTIVATION AND THREAT MODEL

Code authorship attribution is typically considered an instrument of forensics, similar to stylometry [e.g., 3, 19, 56]. For example, Caliskan et al. [12] show how binary code can be attributed to

malware authors, aiding the prosecution of cybercrime. Unfortunately, however, authorship attribution can become a malicious tool itself when used by repressive countries to pursue the developers of regime-critical software, such as anti-censorship and anonymization tools. While there are no reports of automated attribution yet, a recent incident in China underscores the pressure on developers: On November 2, 2023, over 20 developers, some of which operate under a pseudonym, simultaneously removed their anti-censorship software from public repositories, likely indicating a concerted action against them [44, 51]. Similar incidents have already occurred in the past [9, 48], albeit not to this extent.

Any programmer contributing code to open-source code under their real name runs the risk that software developed later under a pseudonym will be linked to their identity. This risk increases if the prosecutor can narrow down a group of individuals and thus simplify the attribution task. In our evaluation in Section 5, we show that this risk is fortunately lower than in the common benchmarks for authorship attribution. Nevertheless, every third developer is correctly identified in our main experiment, raising the question of how such de-anonymization can be prevented, what techniques are available, and whether they offer sufficient protection.

3.1 Defending against Attribution

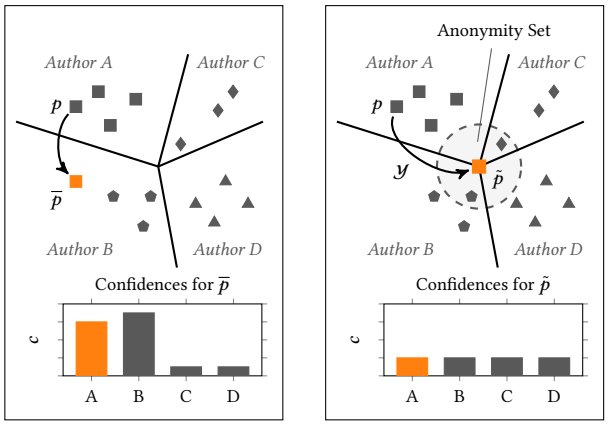
At a first glance, the anonymization of software may seem like a straightforward task: The developer needs to manipulate their code such that the attribution method is tricked into predicting the wrong author, similar to an *adversarial example* [40, 47, 50]. However, there is no reason for a repressive regime to focus only on the most likely person indicated by an attribution method; other highly ranked individuals can also be prosecuted. If we do not know where the prosecutor stops their investigation of the attribution ranking, defending becomes hard. In this case, protection is not about deceiving the attribution but ensuring that one person's coding style is indistinguishable from that of others.

The two plots in Figure 2 illustrate this setting. While an adversarial example can easily cause a misclassification by crossing the decision boundary of an attribution method, it does not provide reliable protection. The true author is still identifiable due to the large differences in the attribution confidences. By contrast, in the right plot, the software is moved near to the intersection of the decision boundary so that several of the developers become similarly likely. The author is protected from identification by an anonymity set. In this work, we set out to investigate this form of *code anonymity* and analyze its technical feasibility.

3.2 Threat Model

To provide a basis for investigating code anonymity, we define a threat model consisting of an *attacker* and a *defender*.

3.2.1 Attacker. The attacker analyzes software with unknown authorship. Their goal is to determine whether some of the code has been developed by the defender. The attacker knows a group of suspected developers and thus operates in a closed-world setup. In addition, the attacker has access to training data, that is, software developed by the suspects under their real identity, for instance, in open-source or work projects. This enables the attacker to apply machine learning and compute a ranking of likely suspects based on the confidence of the learning model.



(a) Misclassification (b) Anonymization

Figure 2: Schematic comparison of misclassification and anonymization in code authorship attribution.

3.2.2 Defender. The defender aims to avoid being identified when publishing specific software, such as anti-censorship tools. The defender has no knowledge of the attribution method employed by attacker. Similarly, the defender does not know the ranking or number of likely suspects. However, they can arbitrarily change their code as long as its semantics are preserved. When modifying their software, the defender prefers changes that preserve readability, so strong obfuscation is only considered as a last resort.

4 CODE ANONYMITY

To the best of our knowledge, there exists no previous work exploring code anonymization. Hence, we first introduce a unified notation for describing programs (software) and their semantics.

4.1 Unified Notation

We denote a *program* by $p \in P$ where P is the set of all valid programs. We differentiate between the *representation* and *semantics* of a program p, where the former defines its code, such as the source code, while the latter describes its behavior [4]. If two programs p_a and p_b have the same representation, that is, the source code is identical, we write $p_a = p_b$. If two programs implement the same semantics, that is, their behavior and output is identical for all inputs, we write $p_a \equiv p_b$.

This differentiation enables us to investigate the relation of representation and semantics: If we have $p_a = p_b$, it directly follows that $p_a \equiv p_b$. The opposite, however, does not hold. Rich programming languages, like C and C++, enable implementing the same behavior in infinite many ways. For example, identifiers can be changed and API functions can be substituted. Given a program p_a, there typically exist many $p_b \in P$, such that $p_a \equiv p_b$ but $p_a \neq p_b$. As an example, Figure 10 in the appendix shows four programs that are semantically equivalent yet make use of different identifiers, types, control flow and API functions. This asymmetry between

representation and semantics fuels the hope that anonymizing code might be a relatively simple task.

4.1.1 Anonymization and Attribution. We continue to introduce notation for attribution and anonymization methods. In particular, to identify the author of a given program, we define a generic *attribution method*

$$\mathcal{A} : P \to (0,1)^n, \quad p \mapsto c = (c_1, ..., c_n) \tag{1}$$

that maps a program p to a vector c of n values $c_1, ..., c_n$, each associated with the confidence for one of n possible authors. Without loss of generality, we assume that $\mathcal{A}$ is deterministic and attains a performance at least as good as random guessing.

In practice, attribution methods typically return the author associated with maximum confidence, that is, $\operatorname{argmax} \mathcal{A}(p)$. However, all current approaches for learning-based attribution provide a measure of confidence, such as the class probabilities returned by a random forest or a neural network [2, 6, 11]. Consequently, they all fit our generic definition of $\mathcal{A}$. As an example, in Figure 2(a) the attribution method $\mathcal{A}$ returns the confidence vector $\mathcal{A}(\tilde{p}) = (0.6, 0.7, 0.1, 0.1)$.

As antagonist to the attribution method in our analysis, we introduce a generic *anonymization method*

$$\mathcal{Y} : P \to P, \quad p \mapsto \tilde{p} \text{ with } p \equiv \tilde{p} \tag{2}$$

where the anonymized program $\tilde{p}$ is semantically equivalent to p but possess properties that obstruct the attribution. Without loss of generality, we assume that the anonymization remains in the set P of valid programs. For example, P could be defined as all programs that solve a particular task, and thus any semantic-preserving transformation remains within this set. In Figure 2(b), the anonymization method $\mathcal{Y}$ changes the attribution, so that we have $\mathcal{A}(\mathcal{Y}(p)) = (0.25, 0.25, 0.25, 0.25)$.

Note that we do not explicitly model the feature types and learning algorithms within $\mathcal{A}$ or the code transformations performed by $\mathcal{Y}$ at this point. In a our threat model, the defender is not aware of the employed attribution method and thus an analysis of the underlying feature space and the impact of code transformations on the features cannot be anticipated.

4.2 Modeling Anonymity

Equipped with this notation, we are ready to formally model the anonymity of source code. For this purpose, we build on the classic concept of *k-anonymity* proposed by Sweeney [55] and expand it to authorship attribution. As we see in the following, even this simple concept with known weaknesses is hard to realize on programs.

Definition 1 (k-anonymity). *Given an attribution method $\mathcal{A}$, a program p is k-anonymous, if the attribution confidence c_t of the true author is identical to the confidence values of at least $k-1$ other authors. That is, for $\mathcal{A}(p) = c$ holds $c_t = c_i = \ldots = c_{i+k-1}$ and* $\operatorname{argmax} \mathcal{A}(p)$ *is not unique.*

For ease of presentation, we reference the $k-1$ authors in sequential order from i to $i+k-1$, although their indices may be arbitrary distributed in the vector c. This definition implies that for a k-anonymous program, the true author is indistinguishable from at least $k-1$ other authors and thus remains hidden in an anonymity set of size k. Hence, an anonymization method realizing k-anonymity transforms a given program, so that it resides at the exact intersection of the classification function for k authors in the feature space, as shown in Figure 2.

The above definition is not directly applicable in our threat model, as the defender has not knowledge of the attribution method $\mathcal{A}$. Hence, we introduce *universal k-anonymity*, an extended definition which aims to protect against any possible attribution method, thus compensating the missing knowledge of the defender.

Definition 2 (Universal k-anonymity). *A program p is universal k-anonymous, if it is k-anonymous for any possible attribution method $\mathcal{A}$.*

Although Definition 2 may seem like a good start for reasoning about attribution and designing methods for code anonymization, it already reaches the general limits of computability.

Theorem 1. *Given a program p, the problem of transforming p using an anonymization method $\mathcal{Y}$ so that $\mathcal{Y}(p)$ is universal k-anonymous is incomputable (undecidable).*

Proof. We reduce the problem of *program equivalence*, which is known to be undecidable [24], to the task of creating universal k-anonymity. Let p_a and p_b be two programs written by developers a and b with $p_a \neq p_b$. Furthermore, let $\mathcal{Y}$ be an anonymization method whose output is universal k-anonymous. Then, the programs are semantically equivalent if and only if their anonymization yields the same representation, that is,

$$\mathcal{Y}(p_a) = \mathcal{Y}(p_b) \iff p_a \equiv p_b. \tag{3}$$

To understand this reduction, let us suppose the programs are semantically equivalent. Then, as long as $\mathcal{Y}(p_a)$ and $\mathcal{Y}(p_b)$ are not identical, there always exists an attribution method $\mathcal{A}_\delta$ that can differentiate the developers. This $\mathcal{A}_\delta$ can be constructed as follows: We describe the difference between the anonymized programs as $\delta = \mathcal{Y}(p_a) \setminus \mathcal{Y}(p_b)$, where we assume that $\delta \neq \emptyset$. As the programs are semantically equivalent, the difference δ can only result from the coding style of the developers. Thus, we can define $\mathcal{A}_\delta$ as

$$\mathcal{A}_\delta(p) = \begin{cases} (1,0) & \text{if } \delta \text{ is in } p, \\ (0,1) & \text{otherwise.} \end{cases} \tag{4}$$

Since $\mathcal{A}_\delta$ can be constructed for any difference δ, the method $\mathcal{Y}$ is forced to normalize the programs to the same representation, such that we have $\mathcal{Y}(p_a) = \mathcal{Y}(p_b)$. If the programs are not semantically equivalent, their anonymized representation can never be identical and we always get $\mathcal{Y}(p_a) \neq \mathcal{Y}(p_b)$. As a result, a method $\mathcal{Y}$ creating universal k-anonymous programs would solve the undecidable problem of program equivalence and thus is incomputable. □

Theorem 1 fundamentally limits our ability to anonymize code. Although k-anonymity is a rather weak concept that suffers from well-known shortcomings [32, 36], we are not even able to establish it on source code when the attribution method is unknown. In view of the great flexibility of expressing semantics in code, this is a surprising, negative result that unveils the challenges of protecting developers from identification.

Takeaway message. The problem of creating universal k-anonymity on source code is incomputable. Although theoretically appealing, the development of approaches to solve this problem for Turing-complete programming languages is a dead end for research.

4.3 Relaxing Anonymity

As a consequence of this situation, we lift our requirements and seek a weaker definition of code anonymity. To this end, we propose a relaxed form of an anonymity set: Instead of requiring k authors to receive an identical attribution, we demand that their confidence values lie close to each other, that is, within an interval of a small value ϵ. An anonymization method now needs to transform a program so that it is close to the intersection of the classification function, yet it is not forced to create identical programs. This relaxation is illustrated in the right plot of Figure 2 where the vicinity of the intersection is indicated by a circle.

To model this concept, we consider the k-nearest neighbors of an author t in the confidence vector c. In particular, we define a permutation π of c that sorts the confidences according to their distance from c_t in ascending order. The k-nearest neighbors can then be defined as a sequence $N_{t,k}$ as follows

$$N_{t,k} = (c_{\pi[1]}, c_{\pi[2]}, \ldots, c_{\pi[k]}) \tag{5}$$

where the true author's confidence is the first element and we always have $|c_{\pi[i]} - c_t| \leq |c_{\pi[j]} - c_t|$ for any $i < j$. Based on this relaxed form of an anonymity set, we introduce a new concept for anonymity that we denote as *k-uncertainty*. This concept is a generalization of k-anonymity from Definition 1, where for $\epsilon = 0$, both concepts are equivalent.

Definition 3 (k-Uncertainty). *Given an attribution method $\mathcal{A}$, a program p is k-uncertain, if there exist at least $k-1$ other authors whose confidence values are ϵ-close to the true author. That is, for $\mathcal{A}(p) = c$ holds $\max(N_{t,k}) - \min(N_{t,k}) \leq \epsilon$.*

Since k-uncertainty is a generalization of k-anonymity, it inherits its undecidability and is also incomputable when the attribution method is unknown. However, instead of enforcing a binary notion of anonymization (k-anonymous or not), this concept introduces a continuous level of anonymity ($0 \leq \epsilon \leq 1$). As we see in the following, this representation enables us to construct a measure for assessing the protection of developers in practice.

4.4 Measuring k-Uncertainty

The concept of k-uncertainty gives rise to a *quantitative measure* of code anonymization. Instead of fixing ϵ, we can determine the size of the interval around an author's k-nearest neighbors and thus gauge how well a program can be attributed to that author. To achieve this goal, we define a corresponding *uncertainty score*

$$u_k(t, c) = 1 - (\max(N_{t,k}) - \min(N_{t,k})) \tag{6}$$

that takes a normalized confidence vector c as input and returns the attribution uncertainty for the author t based on Definition 3. If the author is clearly identifiable, this score returns 0, whereas if she is perfectly hidden in an anonymity set, we obtain 1. As an example, let us consider the confidence vector $c = (0.8, 0.1, 0.1, 0.0)$ with $k = 3$. We immediately see that the first author stands out from the rest. This exposure is also reflected in the uncertainty score $u_3(1, c) = 0.3$. The second author cannot be clearly separated from the nearest neighbors, yielding $u_3(2, c) = 0.9$.

Note that this use of a threshold ϵ deviates from typical privacy research, where ϵ is fixed in advance. By contrast, we fill the other variables in Definition 3 and use ϵ as the output. This unusual inversion allows for empirical analysis of existing protection and attribution methods. Since theoretical guarantees are currently not in reach, we argue that this is the next feasible step towards understanding and limiting the identification of developers.

4.5 Interpreting k-Uncertainty

The uncertainty score provides us with a numerical measure for anonymity, yet its interpretation is not straightforward. The score depends on the particular type of confidence values. If these values correspond to class probabilities, as in many learning algorithms, we have $\sum_{i=1}^{n} c_i = 1$ and can thus define a heuristic for determining a threshold t_ϵ on the value of ϵ.

For class probabilities, the maximum confidence of an author needs to be above $\frac{1}{n}$ to make a reliable attribution, as otherwise the method would not be better than random guessing. Consequently, we define $t_\epsilon = \frac{1}{n}$. This ensures that the k authors of the anonymity set lie within an interval that is smaller or equal to the confidence of random guessing. With this heuristic, we can also interpret the uncertainty score and reason that scores above $1 - t_\epsilon$ provide practical k-uncertainty on class probabilities. We must emphasize, however, that this heuristic is not generally applicable and must be carefully considered for each type of confidence values.

4.6 Alternative Measures

As with any new measure, it is essential to question its necessity and explore the use of existing measures instead. While we do not claim that the proposed uncertainty score is the only way to quantitatively assess anonymity, we argue that it offers advantages over other performance measures. To understand these differences, recall that we focus on an attacker who is not limited to identifying the first predicted author of a software. Instead, they can analyze confidence values and use all information therein to find suspects to pursue, such as the top-k predicted authors or developers with suspicious gaps in confidence.

As a result, classic measures for attribution performance, such as the *accuracy* and the *F-measure*, are not suitable. They only reflect whether a prediction is correct and ignore the remaining information in the ranking and confidence. Whether the actual author is right next to the top prediction or in the middle of the ranking does not make a difference. Consequently, these measures overestimate the protection and may flag an anonymization approach as secure, although the defender is identifiable with little extra effort.

Performance measures from information retrieval partially address this problem [38]. For example, measures such as *mean reciprocal rank* and *top-k-accuracy* are specifically designed to take into account the order of predictions and thus where the true author ranks. Nonetheless, examining the ranking alone does not provide

sufficient protection: First, the defender cannot anticipate the number of top k entries investigated by the attacker. Second, unusual gaps in the confidence values may still indicate suspicious authors, regardless of their ranking. Even worse, if the attacker knew the defender hides below rank k, they could adapt their strategy accordingly, weakening the protection again.

We argue that a quantitative measure of protection must include a notion of privacy and not only attribution performance. That is, the actual author should not only be ranked low but also protected within an anonymity set of developers with similar confidence, making identification much more difficult. It is precisely this idea that forms the concept of our uncertainty score. We demonstrate in Section 5 that this notion of privacy pays off and helps to evaluate protection techniques more accurately.

5 ANONYMIZATION UNDER TEST

Prepared with a practical definition of anonymity, we can now take a look at different approaches for protecting developers. Our goal is to put these approaches to the test and assess how well they can realize k-uncertainty in different scenarios. In particular, we study a *static scenario*, where the adversary is unaware of the anonymization, and an *adaptive scenario*, where she adapts the attribution to it. Before presenting these tests, we introduce the *candidate techniques* for anonymization and our *evaluation setup*.

5.1 Candidate Techniques

As there exist no approaches explicitly designed for anonymizing source code, we focus on techniques that reduce the presence of coding style. Specifically, we examine three candidate techniques: *code normalization*, *coding style imitation*, and *code obfuscation*. All three differ in the amount and precision of their modifications.

5.1.1 Code Normalization. The goal of normalization is to modify code so that it conforms to a given policy or style guide. Normalization is regularly employed in collaborative software development, and larger projects typically define detailed guidelines for the layout and structure of code [e.g., 14, 22, 33]. Inspired by the available style guidelines, we develop a strong code normalization and make it available to the research community. Our normalization builds on 13 transformation rules for C code that unify the code layout, replace the names of variables and functions, reduce the variety of data types, and simplify control structures. All rules preserve the program semantics, so that the normalization complies with our definition of an anonymization method. Table 4 in Appendix D provides a detailed listing of the implemented rules.

Note that code normalization can be applied without access to an attribution method and thus provides a generic approach for reducing the presence of stylistic patterns in source code.

5.1.2 Coding Style Imitation. As second candidate, we consider techniques that can imitate the coding style of developers. In particular, we focus on approaches for creating adversarial examples of source code [34, 47]. In contrast to normalization, these attacks require access to an attribution method and allow more target-oriented code modifications. Typically, adversarial examples of source code are realized in a two-stage procedure: First, a set of code transformations is defined, each imitating a stylistic pattern, such as adding or removing preferences for particular data types or control structures. Second, these transformations are chained together using a search strategy until a target classification is reached. This procedures also preserves the semantics of the code.

There exist different variants for creating adversarial examples on source code. For our tests, we focus on the method by Quiring et al. [47], as it does not only induce misclassifications of the attribution method, but also enables lowering its confidence. While the objective of the method technically remains misclassification, we conjecture that the low confidence better protects the author and thus might serve as a suitable anonymization approach.

5.1.3 Code Obfuscation. As third technique, we consider the obfuscation of source code. This candidate aims to make code incomprehensible to humans while preserving its semantics. Technically, this can be achieved by, for example, encrypting constants or obscuring control flow. We refer the reader to the book by Nagra and Collberg [42] for an introduction to this topic. Obfuscation is agnostic to the attribution method and can be employed to any available code. For our experiments, we make use of two common obfuscators, *Stunnix* [53] and *Tigress* [16]. Stunnix obfuscates identifiers, constants and literals. Still, the overall structure of the program remains unchanged. Tigress is a more sophisticated tool and considered state of the art in obfuscation. It supports several advanced obfuscation techniques, such as function merging and code virtualization [see 16, for details].

Note that obfuscation is intended to prevent an understanding of code and not its attribution. Hence, obfuscators only implicitly destroy the coding style of developers.

5.2 Evaluation Setup

Before testing the different candidate techniques, we introduce our evaluation setup, which follows the common design of experiments with code attribution [30].

5.2.1 Evaluation Datasets. As basis for our evaluation, we consider two datasets of source code in the language C. We restrict our dataset to plain C and do not consider C++, since the obfuscator Tigress only works with this language and several features of C++ hinder code transformations, such as dynamic bindings.

GCJ Dataset. The first dataset called *GCJ* contains code written by 30 authors as part of the *Google Code Jam* [26] competition between 2012 and 2014. All authors solved the same 8 tasks, so the differences in their solutions are caused by their individual coding style. In total, our dataset contains 240 source files. Similar datasets are commonly used to evaluate attribution methods [see 2, 11]. For our experiments, we use a *grouped k-fold* split to select seven of the eight problems for training and reserve the last one for testing. Since the source code of this dataset comes from a competition, it is not fully representative of real-world programming.

GH Dataset. The second dataset called *GH* has been crawled from GitHub. It contains source code in the C language written by 81 developers. The code has been collected according to the procedure described in Appendix B and contains 391 repositories with a total of 1,284 source files. To simulate the threat model described in Section 3.2, we split the data along the repositories into training and test sets. That is, one repository for each author is considered

unknown and used for testing, while the others serve as training data. In contrast to the GCJ dataset, the collected code comes from actual software projects and exhibits a wide variety of functionality, complicating the task of inferring coding style.

Before extracting features from both datasets, we expand all macros, remove comments and use *clang-format* [15] to eliminate trivial layout differences. While this process requires little effort for the GCJ dataset, we need to establish functional build environments for several of the GitHub repositories, which is a labor-intensive task, consuming about one person month of work.

5.2.2 Attribution Methods. We employ two state-of-the-art attribution methods to evaluate the effectiveness of the candidate techniques: the method by Caliskan et al. [11] based on a random forest and the method by Abuhamad et al. [2] which primarily uses a recurrent neural network. The approaches differ in the extracted features, where Caliskan et al. employ a mixture of lexical and syntactic features, while Abuhamad et al. use lexical tokens only. As a result, we gain insights on how the learning algorithms and the features impact an anonymization. Technically, we build on the framework of Quiring et al. [47] and the corresponding implementations. As a coherence check, we compare the performance of our setup with the original C++ dataset used by Quiring et al. The results are shown in Table 1 and differ only slightly, indicating a valid experimental setup.

Table 1: Attribution performance (accuracy) as reported by Quiring et al. [47] and reproduced by us on C++ source code.

Attribution	Quiring et al.	Our implementation
Caliskan et al.	0.904 ± 0.02	0.901 ± 0.02
Abuhamad et al.	0.884 ± 0.04	0.879 ± 0.05

When applying the methods of Caliskan et al. and Abuhamad et al. to the two datasets of C source code considered for our evaluation, the performance changes significantly, as shown in Table 2. The accuracy decreases by up to 22% points for the GCJ dataset and 62% points for the GH dataset. We identify three factors contributing to this drop: First, we remove all comments and layout features during pre-processing, which eliminates trivial clues for discriminating developers. Second, we focus only on C code, which is less diverse in comparison to C++. Third, the GH dataset consists of real-world code, which is characterized by a large variety of functionalities. Consequently, less information about the coding style is accessible for attribution. Nevertheless, roughly one out of three authors is correctly attributed by the two methods for both datasets, demonstrating the need for code anonymization.

In addition, Table 2 shows our new uncertainty score for the two attribution methods for $k = 5$. Despite their similar performance, the uncertainty scores differ considerably. The approach of Caliskan et al. yields 0.84 on the GCJ dataset, whereas the method of Abuhamad et al. reaches only 0.26, indicating large differences in confidence between the authors. We examine these disparities later in Sections 5.3 and 5.4. For different values of k, the uncertainty score changes only slightly and thus we keep $k = 5$ for the remaining experiments.

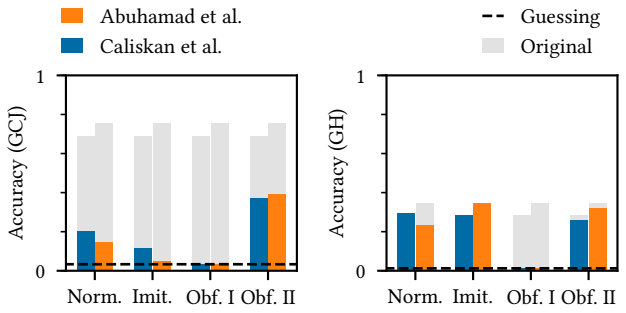


Figure 3: Attribution performance (accuracy) of candidate techniques in the *static* attribution scenario (regular training).

5.2.3 Candidate Techniques. We implement the code normalization using *LibTooling*, a versatile library of the LLVM infrastructure [15]. For the coding style imitation, we again build on the framework by Quiring et al. [47] and fit it to our setup. For the code obfuscation, we employ Stunnix in version 4.7 and Tigress in version 3.1. For Stunnix, we enable all options, while for Tigress we focus on advanced features, such as virtualizing functions, inserting random code, and obscuring API calls. Table 5 in Appendix E lists the used features. We ensure that both tools are given random seeds so that randomized elements are different in each run.

5.3 Static Attribution Scenario

In our first scenario, we consider a *static attribution*, where the adversary is unaware of the employed anonymization techniques and treats the modified code as regular programs. For this purpose, we apply the considered techniques for anonymization to the *test set only* and investigate their impact on the accuracy and uncertainty of the attribution methods. This setup reflects situations where the attacker overlooks the presence of manipulated code, for example, when the coding style is imitated.

5.3.1 Attribution Performance. Figure 3 shows the performance of the attribution methods when the four candidate techniques are employed. We observe a huge drop in accuracy compared to the original results in Table 2. Obfuscation I (Tigress) has the largest impact and changes the code so that the accuracy decreases dramatically on both datasets. The attained attribution of developers is no better than random guessing. In contrast, Obfuscation II (Stunnix) shows a weaker protection and reduces the accuracy by only a few percentage points on the GH dataset. The code normalization and coding style imitation reduce the accuracy on the GCJ dataset. For the GH dataset, however, only a minor reduction is observable for

Table 2: Attribution performance without any protection on the GCJ and GH datasets of C source code.

Data	Attribution	Accuracy	Uncertainty Score
GCJ	Caliskan et al.	0.688	0.840
GCJ	Abuhamad et al.	0.754	0.261
GH	Caliskan et al.	0.284	0.900
GH	Abuhamad et al.	0.346	0.686

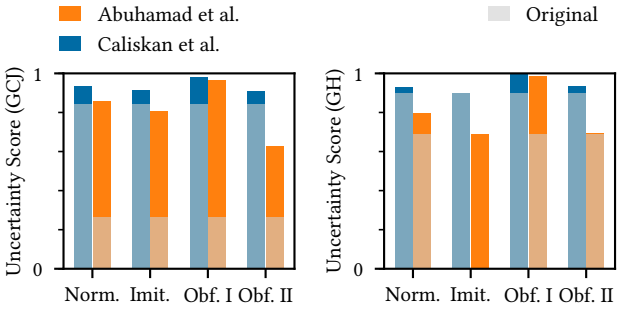


Figure 4: Anonymization performance (uncertainty score) in the *static* attribution scenario (regular training).

the normalization. The imitation provides no protection, since it has been tailored towards GCJ-style code [see 47] and lacks sufficient transformations to effectively manipulate the real-world code from the open-source projects of the GH dataset.

5.3.2 Anonymization Performance. We continue to investigate the anonymization performance of the candidate techniques. Figure 4 shows the average uncertainty score with $k = 5$ for both datasets, that is, an anonymity set of 5 authors. Compared to the original results, there is a significant increase of this measure, indicating better protection of the developers. For the method by Caliskan et al., all values are now above 0.91, while for the approach by Abuhamad et al. all but three scores reach over 0.79. The best performance is obtained for Tigress, reaching an uncertainty score of over 0.97 for both attribution methods on both datasets.

To interpret these values, we apply the heuristic proposed in Section 4.5. Since there are 30 authors in the GCJ dataset and 81 authors in the GH dataset, we can compute the thresholds $\frac{1}{30} \approx 0.03$ and $\frac{1}{81} \approx 0.01$, respectively. As a result, Tigress attains practical k-uncertainty in this experiment, since its uncertainty score reaches above 0.97 for the GCJ dataset and 0.99 for the GH dataset.

Another interesting result of this experiment is that even with a higher remaining accuracy, the uncertainty score for the code normalization is better than for the imitation. While the imitation of coding style more consistently causes misclassifications, the confidence values often remain indicative of the authors. In contrast, the normalization unifies the same stylistic patterns regardless of the original author, thus creating a tighter anonymity set. In view of the complex construction of adversarial examples for imitation, normalizing the source code is a reasonable defense that preserves a good level of readability and reduces stylistic patterns.

Takeaway message. In the static attribution scenario, all techniques reduce the accuracy of the attribution methods. The achieved protection, however, varies between the techniques, with the obfuscator Tigress providing the best performance and achieving practical k-uncertainty when the adversary does not adapt to the anonymization.

5.4 Adaptive Attribution Scenario

In our second scenario, we consider an *adaptive attribution*, in which the adversary is aware of the anonymization. As developing countermeasures for each of the considered techniques is tedious, we use a common trick from the area of adversarial machine learning: We employ two simple variants of *adversarial training* [25] that enable the learning algorithms to extract clues from the modified source code of any possible anonymization strategy.

For code normalization and coding style imitation, we simply augment the training data with modified samples. That is, we provide the original source code and a normalized or imitated version of it. Since both candidate techniques are easily overlooked by an attacker in practice, this augmentation ensures that the attribution methods can capture stylistic patterns from *both*, the original and the modified code. As a result, the methods are applicable regardless of whether the candidate techniques are used or not. To also account for this situation in the performance evaluation, we extend the test data by providing both versions of the source code.

For obfuscation, we pursue a different variant of adversarial training. In this case, the attacker can easily spot whether a source code has been modified and hence we train the attribution methods on obfuscated code only. This strategy forces the attribution method to hunt for subtle clues in the modified code, despite randomized names, virtualized functions, and obscured control flow.

5.4.1 Attribution Performance. Figure 5 presents the attribution performance for the different techniques in the adaptive scenario on both datasets. A notable drop in performance is not observable anymore. The accuracy of all techniques remains over 50% for the GCJ dataset and 30% for the GH dataset, except for Tigress. The obfuscator reduces the accuracy to at most 25% on the GCJ dataset and 8% on the GH dataset. Still, the remaining accuracy is significantly better than random guessing, which would correspond to 3% for the GCJ dataset and 1% for the GH dataset. Consequently, the attribution methods are capable of identifying some developers despite strong obfuscation. While the real-world code in the GH dataset increases the efficacy of Tigress, there is room for improving the protection of the 8% remaining developers.

Moreover, the impact of the adaptive attribution is particularly strong for normalization and imitations of code. While for the static scenario both techniques provide some protection, we now

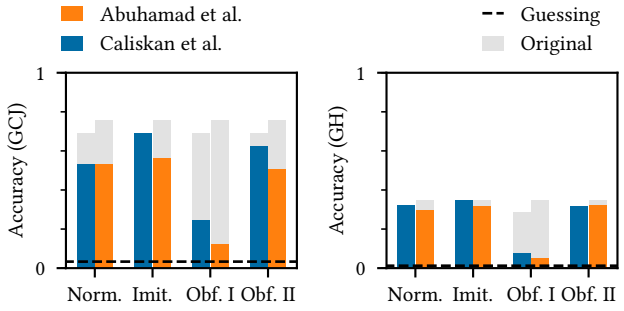


Figure 5: Attribution performance (accuracy) of candidate techniques in the *adaptive* attribution scenario (adversarial training).

observe no defense and in some cases even an improved performance. The weakness of the techniques is that they aim to modify specific aspects of the source code, but do not conduct broader transformations. These minor modifications are compensated by the adversarial training, so that the learning model can even better generalize the remaining coding style.

5.4.2 Anonymization Performance. The weak protection in the adaptive attribution is also reflected in the uncertainty scores shown in Figure 6 for both datasets. Compared to the static scenario, there is no clear tendency for the values to increase. In several cases, the scores even decrease, suggesting that the authors are now better identified than before anonymization. Based on our heuristic, none of the approaches provides adequate protection except for *one* configuration. The obfuscator Tigress achieves practical uncertainty for the method of Caliskan et al. on the GH dataset, as its uncertainty value is above 0.99. For Abuhamad et al.'s method, however, only a value of 0.97 is achieved. Consequently, our simple variants of adversarial training are already sufficient to largely remove the protection of the candidate methods.

We observe another phenomenon: In several cases, the uncertainty score increases for the method of Abuhamad et al. while it decreases for the approach of Caliskan et al. To investigate this, we analyze the distribution of uncertainty scores. The corresponding histograms for the GCJ dataset are shown in Figure 7. The method of Caliskan et al. leads to a one-sided distribution. Between 40% to 60% of the authors cannot be identified well. In contrast, the approach of Abuhamad et al. induces a two-sided distribution. Some authors are well protected while others are perfectly identifiable. We attribute this observation to the tendency of neural networks, as used by Abuhamad et al., to not generalize in all cases.

While adversarial training does not completely eliminate the effect of the four candidate techniques, it weakens the attained protection considerably. Given that this approach is a simple countermeasure and more advanced strategies can be conceived, we have to conclude that *none* of the techniques is a reliable solution for code anonymization if an adversary is aware of their application. Still, we note that strong obfuscation eliminates several clues from the code despite adversarial learning and thus closing the remaining gap is a promising direction for future research.

> *Takeaway message.* In the adaptive attribution scenario, the attribution methods are weakly affected by the considered techniques, and the majority of authors remains identifiable. The obfuscator Tigress provides the best protection, yet it fails to reach practical k-uncertainty in all cases, indicating a need for improved protection.

5.5 Alternative Measures

In Section 4.6, we argue that the proposed uncertainty score considers different information than existing performance measures and thus provides a better view on the anonymization of developers. To investigate this claim, we conduct an additional experiment in which we measure the *correlation* between the uncertainty score and alternative performance measures.

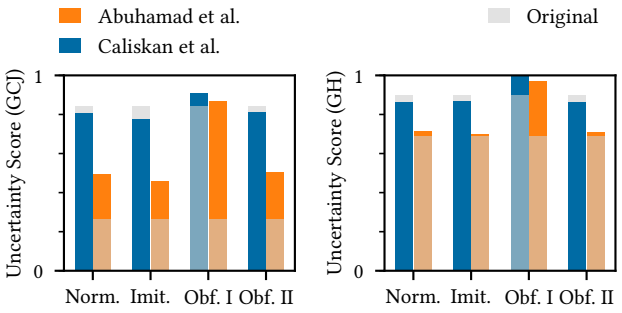


Figure 6: Anonymization performance (uncertainty score) in the *adaptive* attribution scenario (adversarial training).

For this experiment, we consider the *accuracy* as a traditional measure of performance. Furthermore, we consider the *top-5 accuracy*, the *mean reciprocal rank (MRR)* and the *normalized discounted cumulative gain (nDCG)* as three measures from the field of information retrieval suitable for evaluating rankings. We focus on the top 5 prediction, as our uncertainty score is also calculated over the 5 neighboring confidence values. To determine the correlation between these measures, we examine the attribution results on the GCJ dataset. Since we apply 2 attribution methods, 2 scenarios, 4 candidate techniques and a grouped cross-validation over 8 source files, as well as the results of the unmodified source code, we obtain a total of 144 performance values ($2 \times 2 \times 4 \times 8 + 2 \times 8$) for each measure. Based on these values, we calculate the correlation coefficient for each pair of the measures.

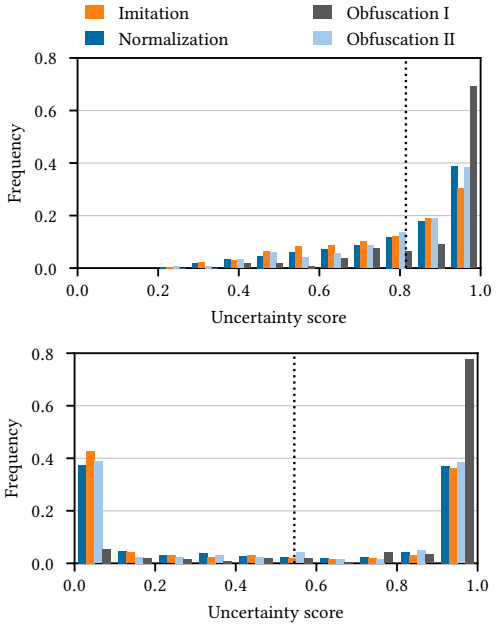


Figure 7: Distribution of uncertainty scores for the adaptive scenario on the GCJ dataset. Top: Caliskan et al. Bottom: Abuhamad et al.

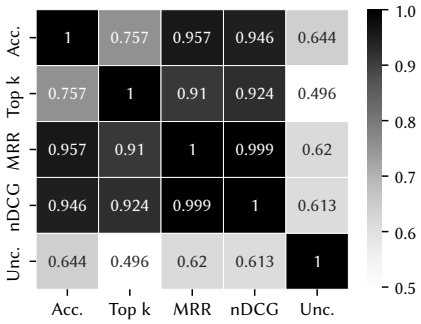


Figure 8: Correlation of alternative performance measures with the uncertainty score.

The results of this experiment are shown in Figure 8. A clear trend can be seen in the resulting correlation matrix. The ranked and unranked performance measures are more strongly correlated with each other than with the uncertainty score. That is, the correlation coefficients between the accuracy, the top-5 accuracy the, the MRR and the nDCG range between 0.757 and 0.999. In contrast, the uncertainty score correlates less with these measures. The highest values are observed for the accuracy at 0.644. The measures derived from information retrieval, which are attractive due to their ability to take rankings into account, only achieve correlation coefficients between 0.496 and 0.62.

These results are not sufficient to demonstrate that the proposed uncertainty score is the only suitable measure for anonymization, yet they show that existing approaches from machine learning and information retrieval rely on other information and cannot be used as a simple alternative. Following our reasoning from Section 4.6 on the role of an anonymity set in this measurement, we therefore argue that the proposed uncertainty score is better suited to describe how a developer is protected from identification than classical performance measures.

6 ANONYMIZATION DEFICITS

Our empirical analysis demonstrates that the four candidate techniques offer only limited protection in practice. In this section, we take a closer look on this problem and introduce two methods for explaining the decisions of attribution. Based on these explanations, we then uncover clues left by the techniques in the source code. This analysis enables us to finally improve Tigress, as the best approach in our experiments, and iteratively remove remaining clues.

6.1 Understanding Attribution

We introduce two strategies to understand why an attribution is still possible after a candidate technique has been applied to a program: *feature highlighting* and *occlusion analysis*.

6.1.1 Feature Highlighting. A simple yet effective way to explain an attribution is to trace back the decision of the learning algorithm to individual features of the code. To this end, we adjust the feature extractions to collect the *code regions* associated with each feature. For AST-based features, these regions can be easily determined using the Clang frontend. Only a few features, such as the depth of the AST, have no specific code region and are thus omitted.

Algorithm 1 Explaining attributions with occlusions

Require: Program p, attribution $\mathcal{A}$, anonymization $\mathcal{Y}$, author t
1: $c_t^* \leftarrow \mathcal{A}(\mathcal{Y}(p))$
2: $S \leftarrow \text{SEGMENTCODE}(p)$ ▷ Line splitting / program slicing
3: $R \leftarrow (0, \ldots, 0) \in \mathbb{R}^{|S|}$ ▷ Initialize relevance vector
4: **for all** $s \in S$ **do**
5: $\quad p^s \leftarrow \text{OCCLUDESEGMENT}(p, s)$
6: $\quad c^s \leftarrow \mathcal{A}(\mathcal{Y}(p^s))$ ▷ Attribution w/o segment s
7: $\quad R_s \leftarrow (c_t^* - c_t^s)$ ▷ Relevance of segment s
8: **end for**

Based on this mapping, we apply *explanation methods* to trace back the attributions to code regions [59]. For example, for the random forest classifier employed by Caliskan et al., we use the method *TreeInterpreter* [49], which returns the contribution of every tree node to the prediction. We then color the code regions based on this relevance. Figure 9 exemplifies the explanation for a code snippet after applying Stunnix. The includes, declarations, and API usages are shaded in darker color, indicating that they still provide clues for authorship attribution. In fact, these patterns consistently occur for the respective author in our evaluation.

6.1.2 Occlusion Analysis. Feature highlighting is particularly effective for explaining the attribution of mildly modified code. For strong obfuscation, however, it reaches its limits. While we can highlight areas in the obfuscated code generated by Tigress, these are incomprehensible by design and impede further analysis. To address this problem, we introduce *occlusion analysis*. Similarly to the field of computer vision, where classifications are often explained by occluding regions of an image [61], we occlude areas of the source code and observe the resulting attribution.

Algorithm 1 provides an overview of this approach. First, we partition the unobfuscated code into segments S (line 2). Then, we iteratively remove each segment $s \in S$ from the code, apply the obfuscation $\mathcal{Y}$ and perform the attribution $\mathcal{A}$ (lines 4–8). After this step, the relevance R_s of the segment s is given by the confidence difference to the original attribution (line 7). We repeat this process, so that we obtain a relevance map over all segments.

While the method can be applied to every anonymization techniques, we cannot remove arbitrary parts of a program without affecting its syntax. To address this problem, we introduce two strategies: As the first strategy, we split the source code along the textual lines. This approach naturally leads to incorrect syntax, yet often the remaining code is still valid and we can narrow down relevant code lines. As the second strategy, we employ *backward*

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 (...)
4 int  main() {
5     int  zf6b4214bfd, zfad4c462ea ;
6     (...)
7     scanf("\x25\x64", &zfad4c462ea);
```

Figure 9: Example of feature highlighting for explaining an attribution. Darker shading indicates more relevance.

program slicing. That is, we use the framework *Frama-C* [17] which enables creating syntactically correct program slices on C code. This strategy preserves the syntax, yet the segments often become large, making an identification of relevant regions difficult.

6.2 Identified Code Clues

With the help of the explanation methods, we investigate the deficits of the candidate techniques on the GCJ dataset. After manually analyzing the highlighted code regions with both strategies, we identify four recurring groups of patterns that remain in the code.

6.2.1 String Literals. The first group of patterns corresponds to string literals. Code normalization and coding style imitation do not modify these, Stunnix just replaces strings with their hexadecimal representations. Therefore, a learning algorithm can use them to find clues about the developers. In contrast, Tigress takes care to not reveal literals by dynamically generating strings at runtime. While the characters themselves are not present, the code necessary for their generation still leaves telltale signs, e. g. the length of the strings is implicitly reflected in the size of the generation routine. As a result, some subtle hints remain in the obfuscated code.

6.2.2 Include Directives. Another group is formed by `#include` directives, which reveal a developer's preferences for certain functions and libraries. Neither code normalization nor the obfuscator Stunnix touch these directives and thus expose these patterns to the attribution methods, as also highlighted in Figure 9. The coding style imitation by Quiring et al. [47] inserts and removes include directives to match other developers. Nevertheless, headers required for the implementation always remain in the code. Finally, Tigress "inlines" the headers by copying their content into the source code. While this makes the resulting code hard to understand for a human, the included content is no different from the directive for a learning algorithm and thus still serves as a valuable hint.

6.2.3 API Usage. API usage provides another set of patterns that remains after anonymization. Automatically changing it is a challenging task, since one must ensure that the replacement is equivalent in functionality. Although the coding style imitation contains some transformations to exchange equivalent C functions, the majority of API calls remains unchanged. The code normalization and Stunnix provide no transformations for this. Tigress calls the API functions by their addresses, so that it can hide function names. Nevertheless, an attribution method can use the types and number of call parameters to narrow down the particular API. In Appendix C, we provide a more detailed analysis of this remaining feature.

6.2.4 Code Structure. Finally, the program structure is often preserved. With the exception of Tigress, the other techniques retain the general organization of the source code. Although the coding style imitation is able to rearrange C statements locally, the overall structure stays unchanged. As a result, personal preferences to structure the program are available to the attribution methods.

6.3 Eliminating Code Clues

Equipped with knowledge of indicative patterns in the modified code, we are ready to refine the anonymization of source code. For this improvement, we focus on the obfuscator Tigress, as it provides the best protection in our experiments.

Table 3: Performance of Tigress with eliminated clues in the adaptive attribution scenario on the GCJ dataset. The numbers in brackets show the difference to the results on unmodified code.

Attribution	Accuracy	Uncertainty score
Caliskan et al.	0.071 (−0.617)	0.969 (+0.128)
Abuhamad et al.	0.058 (−0.696)	0.938 (+0.677)

6.3.1 Code Transformations. To eliminate the identified patterns, we design a set of code transformations that addresses the weak spots. We iteratively apply new transformations and then observe their impact on the attribution using the methods from Section 6.1. This feedback loop enables us to *systematically* identify and eliminate clues left in the code, increasing the attained k-uncertainty.

In particular, we devise the following transformations: To hide string literals, we remove empty function stubs inserted by Tigress and pad all strings to a minimum length. Furthermore, we include all headers from the C standard by default and add at least one call to every API function used in the dataset. For function pointers, we remove all information, except for necessary argument and return types. This makes it complicated to identify the called functions.

6.3.2 Results. Table 3 shows the attribution performance after applying these improvements and conducting another run of adversarial training on the GCJ dataset. We observe a significant decrease in accuracy compared to Table 2 and Figure 5. The values are close to guessing and the uncertainty scores are comparable to the static scenario (see Table 2). For the method of Caliskan et al., the score reaches the threshold of 0.96, so that we attain practical k-uncertainty also for the GCJ dataset. For the method of Abuhamad et al., we come close to this with an uncertainty value of 0.94, but a minor gap still remains.

This positive outcome may seem like the final defeat of the two attribution methods. Unfortunately, this is a misinterpretation of the conducted experiments. We only show that it is possible to achieve k-uncertainty in a controlled environment where the defender can systematically explore the attacker's capabilities for attribution. In practice, however, this is rarely the case, and so we demonstrate the technical feasibility of attaining k-uncertainty in an adaptive scenario but unfortunately not its general realization.

> *Takeaway message.* It is possible to attain k-uncertainty by systematically identifying and eliminating indicative clues in source code. However, this approach is only tractable if the defender operates in a controlled setup and has access to the attribution method, which is rarely the case in practice.

7 LIMITATIONS

With our theoretical and practical analysis, we shed light on challenges of anonymizing code. Naturally, our approach to tackle this problem comes with limitations that we discuss in the following.

7.1 Selection of Techniques

For our experiments, we select two attribution methods and four anonymization techniques. Consequently, our results are based on this particular choice. Nonetheless, we argue that the obtained results are unlikely to be completely different for other selections due to the following reasons:

Attribution Methods. We consider two state-of-the-art methods for authorship attribution. While other approaches would also be applicable [e.g., 8, 18, 19], none of these is fundamentally different in design. All methods extract layout, lexical, and syntactic features. Since the two considered methods already substantially weaken the anonymization, evaluating more attribution methods without new strategies would not provide further insights.

Anonymization Techniques. For code anonymization, we consider common approaches for reducing the impact of coding style. With Tigress, we employ one of the most powerful obfuscators available for C code [7, 16, 42]. Our analysis in Section 6 demonstrates how this obfuscation can be improved to realize k-uncertainty in a controlled setting. The four selected techniques thus provide a broad view on current defenses against authorship attribution. However, we concede that more advanced anonymization strategies are clearly conceivable and hope to encourage more work in this direction with our theoretical and practical analysis.

As a promising candidate for further improvements, we have started experimenting with large language models. These models allow the generation and transformation of source code in different contexts. Due to their probabilistic sampling process, they may serve as an alternative approach to manipulating and obfuscating coding style. In a first experiment, we therefore instructed the OpenAI model GPT-3.5 to change the coding style of the source files from our GCJ dataset. While the model returned modified code for all examples, we found that 11% of them were syntactically incorrect and another 37% unfortunately had different semantics or crashed. For this reason, we have not taken any further steps here and will leave this exploration to future work.

7.2 Size and Type of Code

Compared to prior work, we focus on two small datasets with only 30 and 81 authors, respectively. The reason for this limitation is that we restrict our experiments to plain C code, since advanced transformations on C++ are challenging and not supported by Tigress. Previous work has demonstrated that the performance of learning-based attributions methods decreases gradually with the number of considered authors [see 2, 11]. Hence, the lack of protection observed in our experiments may disappear once the attacker has to consider a large set of authors. However, this set is chosen by the attacker during training and cannot be controlled by the defender directly. A reliable protection should thus be also effective for a small number of authors.

Finally, we focus on C code because it is widely used in software development. Still, we note that interpreted languages, such as Python and JavaScript, offer further strategies for anonymization, including encrypting code and unpacking it at runtime. Although there exists a large series of research on unpacking malicious code [31, 57] that would reveal the original code, investigating other types of protecting code—compilation vs. interpretation—may provide further strategies for improving protection in practice.

7.3 Undecidability

The concept of k-uncertainty inherits the *undecidability* from k-anonymity. It is unfortunately impossible to create an anonymization method that can guarantee k-uncertainty for any possible attribution method and value of ϵ. We argue, however, that k-uncertainty provides an advantage over k-anonymity: It involves a tunable confidence range ϵ. By making this a measurable quantity, we create the uncertainty score that allows us to compare existing methods, which would not be possible with k-anonymity.

While guaranteed anonymity of source code would be preferable and might be attainable in controlled environments, our main result is negative. Nonetheless, we believe that this negative outcome is a central insight that advances research on protecting developers in practice by shaping directions for future research.

7.4 Confidence Values

The uncertainty score builds on the concept of confidence to assess how well a developer is protected. Unfortunately, not all learning algorithms implement this concept to the same extent. While some algorithms return proper confidence values, others only provide output normalized to a range between 0 and 1. In these cases, the uncertainty score only measures the relative proximity of the authors, without an appropriate interpretation of confidence. Similarly, our heuristic for determining practical uncertainty only provides a good estimate if suitable confidence values are provided.

In our experiments, we employ the confidence values returned by a random forest, which correspond to the mean predicted class probabilities of the trees in the forest. These values are based on a reasonable notion of confidence and therefore do not invalidate our analysis. However, when conducting experiments with other learning models, we recommend carefully examining the concept of confidence to avoid misinterpreting the uncertainty score.

8 RELATED WORK

Our work is the first to explore the problem of anonymizing source code. However, we naturally build on previous research from different areas, such as code authorship attribution and data anonymization. In the following, we briefly discuss these related branches.

8.1 Code Authorship Attribution

The starting point for our work has been the remarkable progress in code stylometry, that is, the authorship attribution of code.

8.1.1 Code Stylometry. Several methods have been developed that are able to almost perfectly attribute single-author code to developers using different concepts of machine learning [e.g., 2, 6, 11, 58]. These methods have been further extended to attribute code fragments written by multiple authors [e.g., 8, 18, 56]. This partial attribution, however, proves challenging and therefore leads to lower detection rates. For this reason, we focus our empirical analysis on methods analyzing single-author code. In this way, we evaluate techniques for protecting code under a stronger adversary model.

8.1.2 Coding Style Imitation. Another branch of research has explored the robustness of learning-based attribution methods. In the first study by Simko et al. [50], manual modifications have been used to mimic the style of developers. Following work has then developed concepts for automatically creating adversarial examples of source code [34, 40, 47]. These attacks differ in the employed code transformations and search strategy. For example, Quiring et al. [47] and Liu et al. [34] develop several code transformations that modify lexical and syntactic features, whereas Matyukhina et al. [40] apply rather simple modifications such as changing the layout or copying comments. For our evaluation, we focus on the attack by Quiring et al. [47], as it has a higher evasion rate than the method of Liu et al. [34] and allows changing various lexical and syntactic features in C code.

8.1.3 Text Stylometry. Finally, there is extensive work on attributing authorship of natural language texts and imitating the style of writing. Examples of this research include techniques for detecting patterns in writing style [e.g., 3, 19, 52] as well as approaches for misleading an attribution through writing style obfuscation [e.g., 10, 37, 41]. Our work shares inspiration from this. Due to the fundamentally different properties of natural language and source code, however, these approaches are not directly applicable in our setting.

8.2 Data Anonymization

Another related area is the anonymization and de-anonymization of data [e.g., 27, 36, 43, 55]. Early ideas of this area originate from general data processing and tackle the challenges of analyzing and exchanging privacy-sensitive data, such as medical records.

8.2.1 Anonymity Concepts. One of the first ideas from this area is the concept of *k-anonymity* by Sweeney [55]. In a dataset, every quasi-identifier needs to be hidden in a group of at least k persons with the same identifier, called the anonymity set. This set ensures that no individual can be isolated through personal properties.

The concept of k-anonymity, however, is insufficient when additional data is correlated with the anonymity set. This has led to the development of *ℓ-diversity* [36]. This concept requires for every group of equal quasi-identifiers that at least ℓ different sensitive attributes are also included. In this case, even if an individual can be assigned to a certain equivalence class, the attacker is not able to deduce further sensitive data. This concept was further improved by *t-closeness* [32], which tackles the problem of information disclosure through the different distributions of attributes in an equivalence class and the overall data. This concept requires a similar distribution in both. Hence, an attacker is not able to learn more about a specific individual than about the dataset.

Unfortunately, we conclude from our theoretical analysis that ℓ-diversity and t-closeness are not helpful for protecting code, since k-anonymity is incomputable for an unknown attribution method.

8.2.2 Differential Privacy. Finally, our work also relates to the powerful concept of *differential privacy* [21]. In this concept, privacy-sensitive data is not directly available to users but provided through an interface (or post-processing step). By adding carefully chosen noise to the answers of this interface, it becomes impossible to tell whether an individual is present in the data or not. This concept has recently gained popularity as a strategy for improving the privacy of data in learning models [e.g., 1, 13, 28, 29, 54] and also in the field of natural language processing [e.g., 23, 35, 39, 60].

In natural language processing, the noise is usually not added to the text itself, but to a vector representation of it [23, 35, 60]. While this is an elegant approach, in our setting, this requires knowledge and access to the feature representation used by the attacker. As this is typically unknown to the defender, these approaches are not directly applicable to protect developers from identification.

9 CONCLUSION

Methods for authorship attribution of source code have substantially improved in recent years. While first approaches have suffered from low accuracy, recent techniques can precisely pinpoint a single developer among hundreds of others. Defenses against this progress have received little attention so far and hence we provide the first analysis of code anonymization. Theoretically, we reveal a strong asymmetry between attackers and defenders, where the universal k-anonymity of programs is generally undecidable. Practically, however, we provide a framework for reasoning about and measuring anonymity using the concept of k-uncertainty.

Although we can generate k-uncertainty in a controlled setup, the main conclusion of our empirical analysis is negative: We find that effective techniques for protecting the identity of developers in practice are still lacking. Research on such techniques is challenging, as the defender is naturally not aware of all possible strategies for attribution, while the attacker can easily compensate new anonymization methods through adversarial training, as we demonstrate in our experiments.

In summary, we conclude that entirely new approaches to anonymization are needed, possibly starting already in program language design and software development. For example, new program languages and environments could be designed with anonymity in mind, so that stylistic patterns and telltale clues are reduced during development, potentially creating a unified mapping between semantically equivalent code and its representation. Our work is a first step in this direction and provides concepts for defining and measuring code anonymity in such future settings.

PUBLIC CODE AND DATA

To encourage further research on code anonymity, we make the implementation of our methods publicly available. In addition, we provide the collected source code so that all experiments can be reproduced and serve as basis for evaluating new approaches.

https://github.com/horlabs/anonymizer

ACKNOWLEDGMENTS

This work was funded by the German Federal Ministry of Education and Research in the project IVAN (16KIS1165K) and under the grant BIFOLD23B, the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA – (390781972), and the European Research Council (ERC) under the consolidator grant MALFOY (101043410). Moreover, this work was supported by the IFI program of the German Academic Exchange Service (DAAD) funded by the Federal Ministry of Education and Research (BMBF).

REFERENCES

[1] M. Abadi, A. Chu, I. J. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. Deep learning with differential privacy. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 308–318, 2016.

[2] M. Abuhamad, T. AbuHmed, A. Mohaisen, and D. Nyang. Large-scale and language-oblivious code authorship identification. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 101–114, 2018.

[3] S. Afroz, A. C. Islam, A. Stolerman, R. Greenstadt, and D. McCoy. Doppelgänger finder: Taking stylometry to the underground. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 212–226, 2014.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, second edition, 2006.

[5] S. Alrabaee, P. Shirani, L. Wang, M. Debbabi, and A. Hanna. On leveraging coding habits for effective binary authorship attribution. In *Proc. of the European Symposium on Research in Computer Security (ESORICS)*, pages 26–47, 2018.

[6] B. Alsulami, E. Dauber, R. E. Harang, S. Mancoridis, and R. Greenstadt. Source code authorship attribution using long short-term memory based networks. In *Proc. of the European Symposium on Research in Computer Security (ESORICS)*, pages 65–82, 2017.

[7] S. Banescu, C. S. Collberg, and A. Pretschner. Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning. In *Proc. of the USENIX Security Symposium*, pages 661–678, 2017.

[8] E. Bogomolov, V. Kovalenko, Y. Rebryk, A. Bacchelli, and T. Bryksin. Authorship attribution of source code: a language-agnostic approach and applicability in software engineering. In *Proc. of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 932–944, 8 2021.

[9] R. Brant. China's VPN developers face crackdown. BBC News Service, https://www.bbc.com/news/blogs-china-blog-40872486, accessed December 2023, 2017.

[10] M. Brennan, S. Afroz, and R. Greenstadt. Adversarial stylometry: Circumventing authorship recognition to preserve privacy and anonymity. *ACM Transactions on Information Systm Security*, 15(3):12:1–12:22, 2012.

[11] A. Caliskan, R. E. Harang, A. Liu, A. Narayanan, C. R. Voss, F. Yamaguchi, and R. Greenstadt. De-anonymizing programmers via code stylometry. In *Proc. of the USENIX Security Symposium*, pages 255–270, 2015.

[12] A. Caliskan, F. Yamaguchi, E. Dauber, R. E. Harang, K. Rieck, R. Greenstadt, and A. Narayanan. When coding style survives compilation: De-anonymizing programmers from executable binaries. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[13] K. Chaudhuri, C. Monteleoni, and A. D. Sarwate. Differentially private empirical risk minimization. *Journal of Machine Learning Research*, page 1069–1109, 2011.

[14] Chromium Project. Chromium coding style. https://www.chromium.org/developers/coding-style/, accessed April 2023, 2022.

[15] clang18. Clang: C language family frontend for LLVM. LLVM Project, https://clang.llvm.org, 2018.

[16] C. Collberg. The Tigress C Obfuscator. Project website: https://tigress.wtf, accessed April 2023, 2023.

[17] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: A software analysis perspective. In *Proc. of the International Conference on Software Engineering and Formal Methods*, 2012.

[18] E. Dauber, A. Caliskan, R. E. Harang, G. Shearer, M. J. Weisman, F. Free-Nelson, and R. Greenstadt. Git blame who?: Stylistic authorship attribution of small, incomplete source code fragments. *Proc. of the Privacy Enhancing Technologies Symposium (PETS)*, 2019(3):389–408, 2019.

[19] S. H. H. Ding, B. C. M. Fung, F. Iqbal, and W. K. Cheung. Learning stylometric representations for authorship analysis. *IEEE Trans. Cybern.*, 49(1):107–121, 2019.

[20] R. Dingledine, N. Mathewson, and P. F. Syverson. Tor: The second-generation onion router. In *Proc. of the USENIX Security Symposium*, pages 303–320, 2004.

[21] C. Dwork. Differential privacy. In *Automata, Languages and Programming*, pages 1–12, 2006.

[22] Firefox Project. Firefox coding style. https://firefox-source-docs.mozilla.org/code-quality/coding-style/index.html, accessed April 2023, 2023.

[23] S. Fletcher, A. Roegiest, and A. K. Hudek. Towards protecting sensitive text with differential privacy. In *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 468–475, 2021.

[24] R. Goldblatt and M. Jackson. Well-structured program equivalence is highly undecidable. *ACM Transactions on Computational Logic (TOCL)*, 13(3), 2012.

[25] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations (ICLR)*, 2015.

[26] Google. Google code jam. https://codingcompetitions.withgoogle.com/codejam, accessed April 2023, 2023.

[27] A. Haeberlen, B. C. Pierce, and A. Narayan. Differential privacy under fire. In *Proc. of the USENIX Security Symposium*, 2011.

[28] X. He, A. Machanavajjhala, C. J. Flynn, and D. Srivastava. Composing differential privacy and secure computation: A case study on scaling private record linkage. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 1389–1406, 2017.

[29] B. Jayaraman and D. Evans. Evaluating differentially private machine learning in practice. In *Proc. of the USENIX Security Symposium*, pages 1895–1912, 2019.

[30] V. Kalgutkar, R. Kaur, H. Gonzalez, N. Stakhanova, and A. Matyukhina. Code authorship attribution: Methods and challenges. *ACM Computing Surveys*, 52(1): 1–36, 2020.

[31] C. Kolbitsch, B. Livshits, B. G. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 443–457, 2012.

[32] N. Li, T. Li, and S. Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 106–115, 2007.

[33] Linux Project. Linux kernel coding style. https://www.kernel.org/doc/html/v4.10/process/coding-style.html, accessed April 2023, 2023.

[34] Q. Liu, S. Ji, C. Liu, and C. Wu. A practical black-box attack on source code authorship identification classifiers. *IEEE Transactions on Informations Forensics and Security*, 16:3620–3633, 2021.

[35] L. Lyu, X. He, and Y. Li. Differentially private representation for NLP: Formal guarantee and an empirical study on privacy and fairness. In *Findings of the Association for Computational Linguistics: EMNLP*, pages 2355–2365, 2020.

[36] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1):3–es, 2007.

[37] A. Mahmood, F. Ahmad, Z. Shafiq, P. Srinivasan, and F. Zaffar. A girl has no name: Automated authorship obfuscation using mutant-x. *Proc. of the Privacy Enhancing Technologies Symposium (PETS)*, 2019(4):54–71, 2019.

[38] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.

[39] J. Mattern, B. Weggenmann, and F. Kerschbaum. The limits of word level differential privacy. In *Findings of the Association for Computational Linguistics: NAACL*, pages 867–881, 2022.

[40] A. Matyukhina, N. Stakhanova, M. D. Preda, and C. Perley. Adversarial authorship attribution in open-source projects. In *Proc. of the ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 291–302, 2019.

[41] A. W. E. McDonald, S. Afroz, A. Caliskan, A. Stolerman, and R. Greenstadt. Use fewer instances of the letter "i": Toward writing style anonymization. *Proc. of the Privacy Enhancing Technologies Symposium (PETS)*, 7384:299–318, 2012.

[42] J. Nagra and C. Collberg. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley, 1 edition, 2009.

[43] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 111–125, 2008.

[44] Net4People Forum. Many popular censorship circumvention tools deleted or archived since November 2, 2023. Forum on Github, https://github.com/net4people/bbs/issues/303, accessed December 2023, 2023.

[45] P. H. Nguyen, P. Kintis, M. Antonakakis, and M. Polychronakis. An empirical study of the i2p anonymity network and its censorship resistance. In *Proc. of the Internet Measurement Conference (IMC)*, 2018.

[46] B. N. Pellin. Using classification techniques to determine source code authorship. Technical report, Department of Computer Science, University of Wisconsin, 2000.

[47] E. Quiring, A. Maier, and K. Rieck. Misleading authorship attribution of source code using adversarial learning. In *Proc. of the USENIX Security Symposium*, pages 479–496, 2019.

[48] Reporters Without Borders. Anti-censorship blogger sentenced to seven years for "subversion". RSF Blog, https://rsf.org/en/china-anti-censorship-blogger-sentenced-seven-years-subversion, accessed December 2023, 2023.

[49] A. Saabas. Treeinterpreter. Project repository: https://github.com/andosa/treeinterpreter, accessed April 2023, 2023.

[50] L. Simko, L. Zettlemoyer, and T. Kohno. Recognizing and imitating programmer style: Adversaries in program authorship attribution. *Proc. of the Privacy Enhancing Technologies Symposium (PETS)*, 2018(1):127–144, 2018.

[51] J. Singh. Anti-censorship tools are quietly disappearing into thin air in China. TechCrunch, https://techcrunch.com/2023/11/21/china-censorship-circumvention-tools-clash-disappear/?guccounter=1, accessed December 2023, 2023.

[52] A. Stolerman, R. Overdorf, S. Afroz, and R. Greenstadt. Breaking the closed-world assumption in stylometric authorship attribution. In *Proc. of IFIP International Conference on Digital Forensics*, pages 185–205, 2014.

[53] Stunnix. C/C++ Obfuscator. Project website: http://stunnix.com/prod/cxxo/, accessed April 2023, 2023.

[54] D. Su, J. Cao, N. Li, E. Bertino, and H. Jin. Differentially private k-means clustering. In *Proc. of the ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 26–37, 2016.

[55] L. Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):557–570, 2002.

[56] M. Tereszkowski-Kaminski, S. Pastrana, J. Blasco, and G. Suarez-Tangil. Towards improving code stylometry analysis in underground forums. *Proc. of the Privacy Enhancing Technologies Symposium (PETS)*, 2022(1):126–147, 2022.

[57] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas. Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 659–673, 2015.
[58] N. Wang, S. Ji, and T. Wang. Integration of static and dynamic code stylometry analysis for programmer de-anonymization. In *Proc. of the ACM Workshop on Artificial Intelligence and Security*, pages 74–84, 2018.
[59] A. Warnecke, D. Arp, C. Wressnegger, and K. Rieck. Evaluating explanation methods for deep learning in security. In *Proc. of the IEEE European Symposium on Security and Privacy*, pages 158–174, 2020.
[60] B. Weggenmann and F. Kerschbaum. Syntf: Synthetic and differentially private term frequency vectors for privacy-preserving text mining. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 305–314, 2018.
[61] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *Proc. of the European Conference on Computer Vision (ECCV)*, pages 818–833, 2014.

A EXAMPLES OF EQUIVALENT CODE

Figure 10 shows four implementations of the Euclidean algorithm in C. The programs are semantically equivalent but make use of different identifiers, types, arithmetics, and control flow. They serve as a simplified example for the variety of coding styles and possible program representations.

```c
/* Vanilla version */

int gcd(int a, int b) {

   while (b != 0) {
      int tmp = b;
      b = a % b;
      a = tmp;
   }

   return a;
}
```

(a) Variant 1

```c
typedef int num;

num gcd(num n, num m) {
 loop:
   if (n > m)
      n = n - m;
   else if (n < m)
      m = m - n;
   else
      return n;
   goto loop;
}
```

(b) Variant 2

```c
#include <stdint.h>

int32_t gcd(int32_t x,
            int32_t y) {
   if (y == 0)
      return x;

   int32_t z =
       x - x / y * y;

   return gcd(y, z);
}
```

(c) Variant 3

```c
#include <math.h>

int gcd(int a, int b) {
   long c;

   for(c = 0; b > 0;
       c = !(a = c)) {
       c = c + b;
       b = fmodl(a,b);
   }
   return a;
}
```

(d) Variant 4

Figure 10: Semantically equivalent programs implementing the Euclidean algorithm.

B GITHUB DATASET

For our experiments, we assemble a dataset of source code from GitHub. Our collection procedure consists of three steps, where we first crawl repositories with source code (Section B.1), then filter them to match our experimental setup (Section B.2), and finally configure proper build environments (Section B.3).

B.1 Crawling Step

To obtain a wide range of repositories, we implement a recursive crawl of Github. The crawl starts with the curated software list *awesome-c*[1], which contains a collection of popular software projects written in C and hosted on Github. From this list, we retrieve metadata from all repositories and extract developers who have contributed to them. We then recursively visit the repositories of these developers. To avoid an explosion of the recursion, we limit the crawling using the following criteria:

(1) We ignore all repositories not marked as C code.
(2) We ignore all forked repositories to avoid duplicates.
(3) We ignore large repositories with over 100 authors.
(4) We ignore large repositories with over 10 Mb codebase.

We terminate the crawl after a period of 24 hours. While this strategy results in a large list of potential source code, most of the retrieved repositories do not satisfy the requirements of our experimental setup. For instance, the majority of C projects are developed collaboratively, so source files can rarely be attributed to individual authors. This mix of authorship poses a problem for attribution, and we refer the reader to the work of Dauber et al. [18] for a corresponding discussion.

B.2 Filtering Step

Next, we employ a filtering step to remove repositories and source code unsuitable for our experimental setup. In particular, this step filters the repositories based on the following criteria:

(1) We remove all source files with less than 50 lines of code, as they are too short for inferring coding style.
(2) We remove all source files with more than 5,000 lines of code, as many of these contain large chunks of constant data or automatically generated code.
(3) We remove all source files whose commit messages contain the words “signed off” or “copied”, which indicates that the commit user may not be the author.
(4) We remove all source files with less than 5 commits. Several projects copy code from other repositories, which is indicated by a lack of active development.
(5) We remove all source files where less than 90% of the lines of code are not developed by a single author. We allow a gap of 10% to compensate for editorial changes.

After the data has been filtered at file level, we remove all empty repositories and the corresponding authors. Finally, to allow splitting the data into training and test partitions, as described in Section 5.2, we keep only those authors who contributed to at least 2 repositories, so that we can use one repository for testing and the others for training. We also require that the training repositories contain at least 7 files. This ensures that the split into training and testing is similar to the GCJ dataset and therefore the performance results are comparable.

B.3 Configuration Step

In the last step, we try to configure the build environment for each repository. For this purpose, we execute supplied configuration scripts, such as *cmake* and *configure*. To obtain a large number of correctly configured projects, we manually install dependencies and set configuration options where possible. We then test the resulting code against the anonymization techniques under test and

[1] https://github.com/oz123/awesome-c

remove those files that cannot be processed correctly, for example, because they require complicated dependencies or use unusual programming features. Overall, this step is the most time-consuming and takes about a whole person-month. As a result, the available source code for our experiments is drastically reduced. In the end, we have 81 authors, 391 repositories and 1,284 files of source code that successfully passes all stages of our experimental setup.

While we aim to provide a more realistic picture of source code than the common GCJ dataset, we have to acknowledge that our restrictive filtering limits its representativeness. However, we consider this filtering a necessary compromise: While we could select a larger proportion from the crawl for our evaluation, we would risk its ground truth being wrong due to mislabeled authors and corrupted source code. These defects could invalidate our experiments, which depend on accurate analysis of identified developers. Therefore, we strive for a balance between valid ground truth and representativeness by starting from a large crawl and then successively removing error sources in the underlying data.

C API HIDING OF TIGRESS

Tigress hides the usage of an API by determining the addresses of the API functions at runtime. Still, the types and number of parameters are specified, because the function pointers must be typed according to the passed parameters for every call. This is achieved by casting the pointers using function declarations from the header files. Some declarations include argument names and thus these are copied into the corresponding casts. This makes it possible to differentiate functions with the same types of parameters.

As an example, the functions `abs` and `close` require a single parameter of type `int` and return the same data type. This leads to a function pointer of type `int (*)(int)`. In the header files, however, the parameter of `abs` is named `__x`, while for `close` it is `filedes`. The corresponding casts in the obfuscated file are therefore (INT (*)(INT __X)) and (INT (*)(INT FILEDES)) and thus are easily distinguishable. As a result, even for Tigress, an attribution method can identify used library functions in this case.

D NORMALIZATION RULES

Table 4 provides a detailed listing of the implemented normalization rules for anonymization.

Table 4: Overview of implemented normalization rules

Rule name	Description
`Renaming`	All variables, functions, and structures are renamed to a generic version. For example, all variables are numbered as `var_x` with `x` being a number starting at 0.
`Types`	The used data types are mapped to a specified subset to eliminate redundant type names, such as `long` and `int_32`. Note that this transformation is platform-specific.
`Switch2If`	SWITCH statements are transformed to a chain of `if-else` statements.
`Comma`	Comma operators are largely eliminated and replaced with a sequence of statements containing the expressions.
`CompoundAssign`	Compound assignments are replaced with normal assignments and the specified binary operator, e. g. `a += 2` is transformed into `a = a + 2`.
`IfElse`	If the last statement inside the body of an `if` statement is for example a `return` or `break`, the following code is moved into an `else` to this `if`.
`MainParams`	This rule enforces the use of two parameters for the `main` function and a `return` statement at its end.
`Multidecl`	If multiple declarations are in a single statement, the statement is split into separate declaration statements.
`Braces`	Braces around every body are enforced, for example, for the bodies of all `if` and `for` statements.
`UnnecessaryReturn`	This rule removes `return` statements in `if` bodies if all following code is in the `else` clause and the function has no return value.
`VoidReturn`	This rule adds a `return` statement at the end of every `void` function.
`FlattenIf`	For nested `if` statements, this rule removes inner clauses by combining the conditions of the inner and outer `if`. It inserts an additional `if` for every `else`.
`Paren`	This rule removes unnecessary parentheses like in `a = (b + c)`, simplifying arithmetic expressions

E USED TRANSFORMATIONS FOR TIGRESS

Table 5 lists the used transformations and arguments for obfuscation with Tigress in detail.

Table 5: Overview of used transformations and arguments for source code obfuscation with Tigress

Transformation	Arguments
InitEncodeExternal	Functions=main InitEncodeExternalSymbols= `<ext. functions>`
InitEntropy	InitEntropyKinds=vars Functions=init_tigress
InitOpaque	Functions=init_tigress InitOpaqueStructs=env
RandomFuns	RandomFunsName=SECRET RandomFunsFunctionCount=3 RandomFunsCodeSize=20 RandomFunsLoopSize=5
EncodeLiterals	Functions= `<all in file>`,main_0, /SECRET.*/ EncodeLiteralsKinds=string EncodeLiteralsEncoderName= stringEncoder
Merge	MergeFlatten=false MergeName=MERGED Functions= `<w/o main>`,main_0, /SECRET.*/
Virtualize	VirtualizeDispatch=switch VirtualizeStackSize=48 VirtualizeOperands=mixed VirtualizeMaxDuplicateOps=2 VirtualizeSuperOpsRatio=0.1 VirtualizeMaxMergeLength=3 Functions=MERGED,stringEncoder
EncodeLiterals	Functions= main,MERGED,stringEncoder EncodeLiteralsKinds=integer
EncodeExternal	Functions=MERGED EncodeExternalSymbols= `<ext. functions>`
CleanUp	CleanUpKinds=*