

I still know it's you!

On Challenges in Anonymizing Source Code

Micha Horlboge*, Erwin Quiring†, Roland Meyer*, Konrad Rieck*

* TU Braunschweig, Germany

† Ruhr University Bochum, Germany

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 and thus a dead end for research. 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 introduce an approach for removing remaining clues from the code, the main result of our work is negative: Anonymization of source code is a hard and open problem.

I. 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 [1]. 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., 2, 3, 4, 5, 6]. As an example, Abuhamad et al. [7] report a detection accuracy of 96% on a collection of source code from 1,600 developers.

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 [8] and I2P [9], 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 6, 7, 10], 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 general method for code anonymization cannot exist and thus the pursuit of strict *k-anonymity* is a dead end 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* [11] and *code obfuscation* [12, 13] as defenses against popular attribution methods [6, 7] on a dataset of 30 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 and conducts adversarial training. For the strongest technique, the popular obfuscator Tigris [12], the attribution still reaches an accuracy up to 24%. 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 indicative clues with our method, we eventually bring the source code in our experiments to a 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 controlled setup but stays an unsolved problem in the open world, where indicative clues remain unknown and cannot be eliminated. In summary, 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 thus make the follow contributions:

- *Theoretical view on code anonymization.* We propose a framework for reasoning about code anonymization. This framework enables us to prove that k -anonymity of code cannot be reached in the general case.
- *Practical view on code anonymization.* We introduce a concept for measuring anonymization in practice. 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 in source code remaining after an anonymization attempt.

Roadmap. We review authorship attribution of source code in Section II. Our framework for analyzing code anonymity is introduced in Section III and we empirically evaluate different techniques with it in Section IV. We analyze the deficits of the techniques in Section V. Limitations and related work are presented in Section VI and Section VII, respectively. Section VIII finally concludes the paper.

II. 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 [2]. As these properties are hard to formally characterize and vary widely among developers, this objective is typically achieved by extracting indicative *features* from source code and constructing an attribution method using *machine learning*. Consequently, existing approaches for authorship attribution are best described based on the considered features and the employed learning algorithms.

A. Features of Source Code

The features currently used in authorship attribution roughly fall into three categories: layout, lexical, and syntactic features. Each of these categories represents the source code from a different view and thus enables access to different types of stylistic patterns left by the authors in the code.

1) *Layout features:* The first type of features are derived from the layout of the source code. Developers often have specific preferences to format and layout their code, ranging from different forms of indentation to the placement of brackets and spacing between identifiers. Figure 1 shows a code snippet, where different types of features are highlighted. Even in this simple function a variety of layout features is visible, such as the indentation width of 4, comments in C++ style, and the placement of brackets. Consequently, any attempt

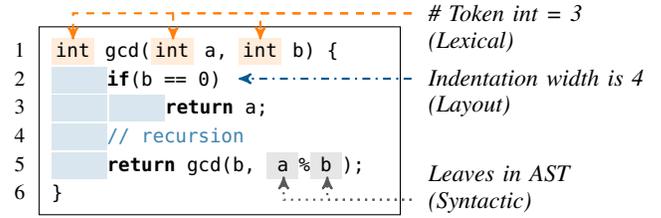


Figure 1: Code snippet in C with highlighted feature types.

to anonymize code needs to start at the layout and remove individual formatting. Fortunately, this can be easily achieved through common formatting tools, such as *GNU indent* and *clang-format*, that automatically unify most layout features.

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 [14]. 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 data type in relation to other types in C, such as `long` or `int32_t`. Moreover, it reveals that some form of integer processing takes place, giving clues about the program semantics. Compared to the layout of source code, lexical features cannot be unified easily, as they are implicitly linked to syntax and semantics. As a result, their anonymization is more challenging, as we show in Section IV.

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 individual patterns in types, arithmetics, logic, and control flow used by developers [6, 15, 3]. 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 manipulate or even unify. Replacing a single keyword in the source code may already 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 and often intractable for these features, as we also demonstrate in Section IV.

B. Attribution using Machine Learning

The described feature types provide a complex and diverse view on source code that is difficult to interpret by a human analyst. As a remedy, attribution methods for source code rely on machine learning for automatically spotting stylistic patterns for a particular author [e.g., 16, 7, 6, 15]. To this end, the extracted features are first post-processed, for example, with TF-IDF weighting and feature selection [7, 6]. Then, a supervised learning algorithm is applied to infer stylistic patterns characteristic for each author. The result is a *multiclass classifier* that returns a prediction along with 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 [16], random forests [6], and deep neural networks [15, 7]. As we find in our evaluation, the choice of learning algorithm can have a considerable impact on anonymization. For example, deep neural networks tend to overfit to particular authors during attribution, while other learning algorithms, such as random forests, provide a more generalized prediction.

III. CODE ANONYMITY

At a first glance, the anonymization of source code may seem like a straightforward task: The defender needs to manipulate her code such that an attribution method is tricked into predicting the wrong author. There exist different approaches for constructing *adversarial examples* of source code that can realize such misclassification for attribution methods [e.g., 1, 11, 17]. However, anonymization is fundamentally different from misclassification, as we illustrate in Figure 2 and demonstrate in the following.

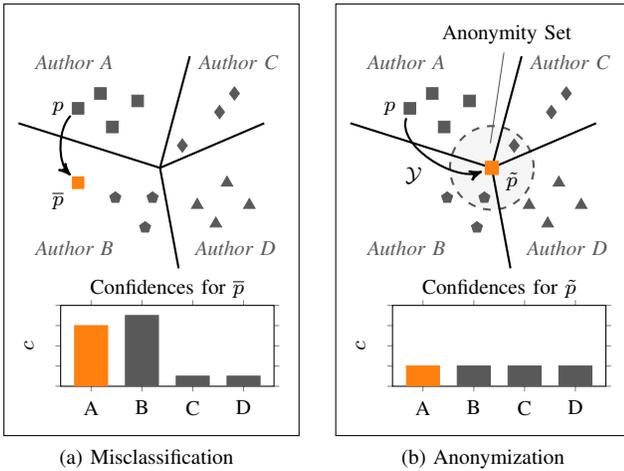


Figure 2: Schematic comparison of misclassification and anonymization in code authorship attribution.

The two plots schematically show a feature space that is partitioned by an attribution method into regions associated with authors. While an adversarial example of source code can easily cause a misclassification, 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 source code is moved near to the intersection of the classification function so that several of the authors become similarly likely. In this case, the author is protected from identification by an *anonymity set* of other programmers with similar attribution confidence.

A. Unified Notation

To the best of our knowledge, there exists no previous work exploring the feasibility of code anonymization. To lay the ground for this exploration, we thus first introduce a unified notation for describing programs and their semantics as well as methods for attribution and anonymization.

1) *Programs and their semantics*: 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 [14]. 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 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, expressions can be reformulated, 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 3 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 technically feasible.

2) *Anonymization and attribution*: To further explore the idea of code anonymity, we introduce notation for attribution and anonymization methods. In particular, to identify the author of a given program, we define a generic *attribution method*

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

that maps a program p to a vector c of n confidence values c_1, \dots, c_n , each associated with the confidence for one of n possible authors. Without loss of generality, we assume that \mathcal{A} is a deterministic function 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 [6, 7, 15]. 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 $\mathcal{A}(\bar{p}) = (0.6, 0.7, 0.1, 0.1)$.

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

$$\mathcal{Y} : P \rightarrow P, \quad p \mapsto \tilde{p} \text{ with } p \equiv \tilde{p} \quad (2)$$

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. The reason is that in a realistic scenario, we are typically 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 by the defender.

<pre> /* Vanilla version */ int gcd(int a, int b) { while (b != 0) { int tmp = b; b = a % b; a = tmp; } return a; } </pre>	<pre> 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; } </pre>
(a) Variant 1	(b) Variant 2
<pre> #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); } </pre>	<pre> #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; } </pre>
(c) Variant 3	(d) Variant 4

Figure 3: Programs implementing the Euclidean algorithm in C. The programs are semantically equivalent but make use of different identifiers, types, arithmetics, and control flow.

B. Modeling Anonymity

Equipped with a unified 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 [18] and expand it to authorship attribution. As we see in the following, even this simple concept with known weaknesses is already hard to realize on source code.

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 $c_t = c_i = \dots = c_{i+k-1}$ and $\text{argmax } \mathcal{A}(p)$ 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.

In practice, however, the defender can only speculate about the capabilities of the attacker and typically has no knowledge of the employed attribution method. To address this scenario, we introduce a more general type of k -anonymity that provides universal protection from authorship attribution.

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 reasonable start for designing robust 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 [19], 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,

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

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}_δ that can differentiate the developers. This \mathcal{A}_δ 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 δ is not empty. 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}_δ as

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

Since \mathcal{A}_δ 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 [20, 21], 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.

C. 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 near 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

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

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 $\max(N_{t,k}) - \min(N_{t,k}) \leq \epsilon$.*

As k -uncertainty is a generalization of k -anonymity, it naturally inherits the undecidability and also is incomputable when the attribution method is unknown. Still, this concept features an important difference when analyzing anonymization methods in practice: Instead of enforcing a binary notion of anonymization (k -anonymous or not), we obtain a continuous level of protection ($0 \leq \epsilon \leq 1$). As we see in the following, this continuous representation enables us to construct a measure for assessing the protection of developers in practice.

D. Measuring k -Uncertainty

The concept of k -uncertainty gives rise to a *quantitative measure* of code anonymization. Instead of fixing ϵ in advance, we can also 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*

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

that takes a 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 $u_3(1, c) = 0.3$. By contrast, the second author cannot be clearly separated from the nearest neighbors, yielding $u_3(2, c) = 0.9$. As a result, we obtain a simple numerical measure for assessing and comparing different approaches to code anonymization.

Clearly, guaranteed anonymization against any attribution method would be preferable, yet our results show that this is technically infeasible. Therefore, we argue that the empirical analysis of known anonymization and attribution methods using the uncertainty score is the next feasible step towards understanding and limiting the identification of developers.

E. Interpreting k -Uncertainty

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

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.

IV. ANONYMIZATION UNDER TEST

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

A. 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 the following three candidate techniques: *code normalization*, *coding style imitation*, and *code obfuscation*. All three techniques differ in the amount and precision of their modifications. While normalization and imitation modify targeted aspects of the source code to unify coding style, obfuscation rewrites the whole program, seemingly destroying all stylistic patterns.

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., 22, 23, 24]. Inspired by the available style guidelines, we develop an own code normalization that aims to unify as much of the coding style as possible. In particular, we implement 13 normalization rules for C source code. These rules unify the code layout, replace the names of variables and functions, reduce the variety of data types, and simplify control structures. All transformations preserve the program semantics, so that the normalization complies with our definition of an anonymization method in Section III-B. Table V in Appendix B 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.

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 [11, 25]. 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 procedure 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. [11], 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.

3) *Code obfuscation*: As third candidate technique, we consider the obfuscation of source code. Essentially, the goal of this technique is to make code incomprehensible to humans while preserving its semantics. Technically, this can be achieved by encrypting constants, obscuring control flow, and even virtualizing code execution. We refer the reader to the book by Nagra and Collberg [26] for further details. Similar to normalization, obfuscation is agnostic to the attribution method and can be employed directly to any available code. For our experiments, we make use of two common obfuscators, *Stunnix* [13] and *Tigress* [12]. *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.

Note, however, that obfuscation is intended to prevent an understanding of code and not its attribution. Hence, obfuscators only implicitly destroy the coding style of developers. As we demonstrate in Section IV-D, this different objective plays an key role when the adversary is aware of the obfuscation.

B. Evaluation Setup

Before testing the different candidate techniques, we introduce our evaluation setup, which follows the common design of experiments with code attribution [2].

1) *Evaluation dataset*: As basis for our evaluation, we collect a dataset of source code in the language C. The code has been written by 30 authors as part of the *Google Code Jam (GCJ)* [27] competition between 2012 and 2014. All authors solved the same 8 tasks in this competition, so that differences in their solutions are caused by their individual coding style. In total, our dataset contains 240 source code files (30 authors solving 8 tasks). Similar datasets are commonly used in previous

work for evaluating attribution methods [see 6, 7]. However, we restrict our dataset to plain C and do not consider C++, as several features of this language obstruct code transformations, such as dynamic binding¹. Moreover, the considered obfuscator *Tigress* also operates on plain C only.

Before extracting features from the collected source code, we expand all macros and remove comments. Furthermore, we use *clang-format* [28] to eliminate trivial layout differences to focus only on lexical and syntactic features. For our experiments, we use a *grouped k-fold* split to select seven of the eight problems for training and reserve the last for testing. This ensures that no characteristics of the tasks itself influence the attribution.

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. [6] based on a random forest and the method by Abuhamad et al. [7] which primarily builds on 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. The authors of the methods report an accuracy of 98.04% and 97.65%, respectively, on similar datasets of source code from the GCJ competition.

Table I: Attribution performance without any modifications.

Attribution method	Accuracy	Std. dev.	Unc. Score
Caliskan et al.	0.688	0.110	0.840
Abuhamad et al.	0.754	0.069	0.261

Table I shows the performance of the two attribution methods on our dataset. In contrast to the original publications, the performance drops by 20-30% in our evaluation setup. We carefully checked our implementation of the methods but could not find any defects. We thus credit this performance drop to two factors: 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++. Consequently, less information is available for the attribution and the classification of coding style. Nonetheless, two out of three authors in our dataset are still correctly attributed by the two methods, demonstrating the need for code anonymization.

In addition, Table I shows our new uncertainty score for the two attribution methods with $k = 5$. Although both methods attain a similar performance, their uncertainty score differs considerably. The approach of Caliskan et al. yields 0.84, whereas the method of Abuhamad et al. reaches only 0.26, indicating large differences in confidence between the authors and a better identification. We examine these differences later when evaluating the candidate techniques in Sections IV-C and IV-D. Note that the uncertainty score changes only slightly when we vary the neighborhood size k , and thus we keep $k = 5$ for the remaining experiments.

¹Some developers jokingly state that C++ is already obfuscated by design.

Table II: Performance of candidate techniques in the *static* attribution scenario (regular training). The numbers in brackets show the performance difference to the results on unmodified source code.

Candidate technique	Attribution method	Accuracy	Std. dev.	Uncertainty score
Code normalization	Caliskan et al.	0.204 (-0.483)	0.033 (-0.077)	0.933 (+0.093)
	Abuhamad et al.	0.146 (-0.608)	0.031 (-0.038)	0.858 (+0.597)
Coding style imitation	Caliskan et al.	0.117 (-0.571)	0.062 (-0.048)	0.913 (+0.073)
	Abuhamad et al.	0.050 (-0.704)	0.036 (-0.033)	0.807 (+0.547)
Obfuscation I (Tigress)	Caliskan et al.	0.033 (-0.654)	0.000 (-0.110)	0.979 (+0.139)
	Abuhamad et al.	0.033 (-0.721)	0.000 (-0.069)	0.966 (+0.704)
Obfuscation II (Stunnix)	Caliskan et al.	0.371 (-0.317)	0.065 (-0.044)	0.911 (+0.070)
	Abuhamad et al.	0.392 (-0.363)	0.092 (-0.023)	0.626 (+0.365)

3) *Implementation details*: We implement the code normalization using *LibTooling*, a versatile library of the clang frontend and LLVM compiler infrastructure [28]. For the coding style imitation, we adopt the open-source framework by Quiring et al. [11] and fit it to our evaluation 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 VI in Appendix C lists the used features in detail. Finally, we ensure that both tools are given proper random seeds so that randomized elements are different in each run.

Prior work [6, 7] has not specified the particular version of Stunnix. We use the publicly available evaluation edition 4.7 of Stunnix that is presumably also used by other work. This version does not provide manglers for identifiers, which may lead to an overestimation of the attribution performance. As a remedy, we re-implement an own MD5-mangler for identifiers according to the manual of the developer version of Stunnix that requires payment.

C. 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 modified code, for example, when the coding style is imitated, or simply fails to adapt to the anonymization.

1) *Attribution performance*: Figure 4 and Table II show 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 I. Obfuscation I (Tigress) has the largest impact on the attribution and transforms the code, so that the accuracy drops by 65% and 72%, respectively. The achieved attribution of the 30 developers is no better than guessing. By contrast, Obfuscation II (Stunnix) shows the weakest protection, yet the number of correctly attributed authors is still halved in comparison to Table I. As a result, code obfuscation leads to a notable number of misclassifications in this scenario, reducing the utility of the two attribution methods.

The code normalization and coding style imitation also reduce the accuracy of the attribution by 48% to 70%. As expected, the normalization has a lower impact than the imitation, since it does not account for the particular attribution method when unifying the coding style. For both techniques, we also see a more pronounced difference between the approaches by Caliskan et al. and Abuhamad et al. While the latter achieves significantly better results on the unmodified dataset, now the method by Caliskan et al. exhibits a higher accuracy. The method takes into account syntactic features that are not considered in the other approach. Consequently, it attains a broader view on the code and thus is more robust.

2) *Anonymization performance*: We continue with the investigation of the anonymization performance of the techniques under consideration. Table II shows the average uncertainty score with $k = 5$, 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 one score reach over 0.80. The best performance is obtained for Tigress, reaching an uncertainty scores of 0.96 for both attribution methods. To interpret these values, we apply the heuristic proposed in Section III-E. Since there are 30 authors in our dataset, we have $n = 30$ and get $t_\epsilon = \frac{1}{30}$. As a result, Tigress is the only approach that reaches k -uncertainty in this experiment, as we have $1 - t_\epsilon \approx 0.96$.

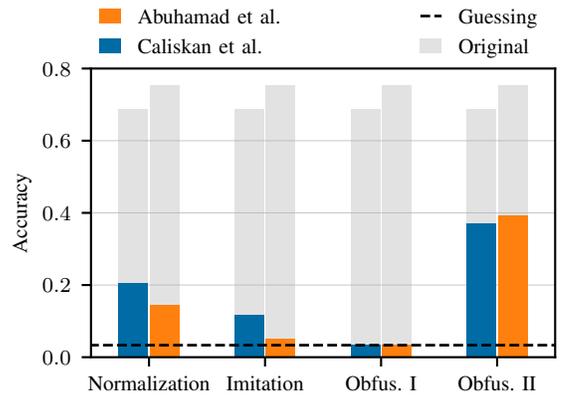


Figure 4: Attribution accuracy in static scenario.

Table III: Performance of anonymization techniques in the *adaptive* attribution scenario (adversarial training). The numbers in brackets show the performance difference to the results on unmodified source code.

Candidate technique	Attribution method	Accuracy	Std. dev.	Uncertainty score
Code normalization	Caliskan et al.	0.533 (-0.154)	0.090 (-0.020)	0.808 (-0.032)
	Abuhamad et al.	0.529 (-0.225)	0.071 (+0.002)	0.492 (+0.231)
Coding style imitation	Caliskan et al.	0.688 (± 0.000)	0.110 (± 0.000)	0.776 (-0.064)
	Abuhamad et al.	0.563 (-0.192)	0.079 (+0.010)	0.458 (+0.197)
Obfuscation I (Tigress)	Caliskan et al.	0.246 (-0.442)	0.087 (-0.023)	0.909 (+0.069)
	Abuhamad et al.	0.121 (-0.633)	0.040 (-0.029)	0.869 (+0.608)
Obfuscation II (Stunnix)	Caliskan et al.	0.625 (-0.063)	0.081 (-0.029)	0.811 (-0.029)
	Abuhamad et al.	0.504 (-0.250)	0.060 (-0.009)	0.502 (+0.240)

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 of the considered techniques significantly reduce the performance of the attribution methods. In particular, the obfuscator Tigress provides strong protection and achieves practical k -uncertainty when the adversary does not adapt to the anonymization.

D. Adaptive Attribution Scenario

In our second scenario, we consider an *adaptive attribution*. In this more realistic setup, the adversary is aware of the anonymization and takes steps to compensate for it during the attribution of source code. As developing countermeasures for each of the considered anonymization techniques is tedious, we use a common trick from the area of adversarial machine learning: We employ two simple variants of *adversarial training* [29] that enable the learning algorithms in the attribution methods 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 source code. That is, we provide the original source code and a normalized or imitated version of it for training, resulting in 14 code samples per author. 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 source code. As a result, the methods are applicable to any source code, 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 an original and a modified version of the source code.

For code 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. Compared to mixing original and modified code, this strategy forces the attribution method to hunt for subtle clues in the obfuscated code, despite randomized names, virtualized functions, and obscured control flow.

1) *Attribution performance:* Figure 5 and Table III present the attribution performance for the different techniques in the adaptive scenario. A notable drop in performance is not observable any more. Except for the obfuscator Tigress, the accuracy of all candidate techniques remains over 50%, so that every second author can be identified in the dataset. Tigress reduces the accuracy to 24% and 12% for the two attribution methods, respectively. However, the performance is notably better than random guessing and now exposes the identity of several developers in the dataset.

The impact of the adaptive attribution is particularly strong for normalization and imitations. While for the static scenario both techniques provide some protection, we now observe an attribution between 52% and 68%, corresponding to almost no defense. 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 remaining clues can still be uncovered.

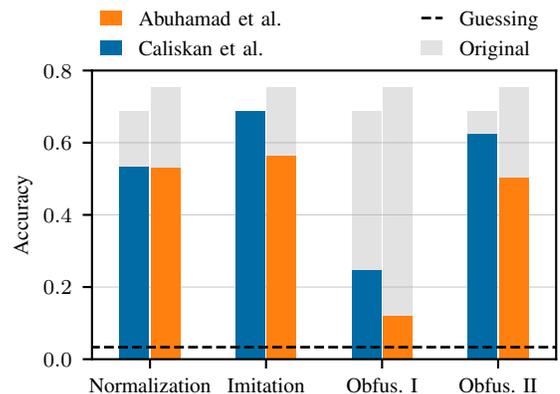


Figure 5: Attribution accuracy in the adaptive scenario.

2) *Anonymization performance*: The weak protection in the adaptive attribution is also reflected in the uncertainty scores listed in Table III. 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, which requires an uncertainty score of 0.96 for 30 authors, none of the techniques provides adequate protection. The obfuscator Tigress is again the best approach, but only achieves an uncertainty score of 0.87 and 0.91 for the two attribution methods, respectively. As a result, our simple variants of adversarial training are already sufficient to lift the protection provided by all candidate techniques.

We also 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 divergence, we analyze the distribution of uncertainty scores in detail. The corresponding histograms are shown in Figure 6. The method of Caliskan et al. leads to a one-sided distribution. Between 40% to 60% of the authors cannot be identified well, resulting in an average uncertainty score of 0.8. In contrast, the approach of Abuhamad et al. induces a two-sided distribution, where some authors are well protected at 1.0 and other are perfectly identifiable at 0.0. Thus, on average the protection is much weaker with a mean of roughly 0.5. We attribute this result to the tendency of the neural network in the approach of Abuhamad et al. to overfit and detect authors either with high confidence or not at all.

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 just a simple countermeasure and more advanced strategies can be conceived, such as extracting features invariant to certain transformations, we have to conclude that *none* of the techniques is a viable solution for code anonymization if an adversary is aware of their application. Unfortunately, we must assume that this is regularly the case in practice, so that our experiments close with another negative result.

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. Anonymity of source code cannot be attained in this scenario.

V. ANONYMIZATION DEFICITS

Our empirical analysis demonstrates that the four candidate techniques offer only limited protection in practice. So far, however, we do not understand the reason for this deficit. In this section, we take a closer look on this problem and introduce two methods for explaining the decisions of attribution methods (Section V-A). Based on these explanations, we then uncover clues left by the techniques in the source code (Section V-B). This analysis enables us to finally improve Tigress, as the best approach in our experiments, and iteratively remove remaining clues (Section V-C).

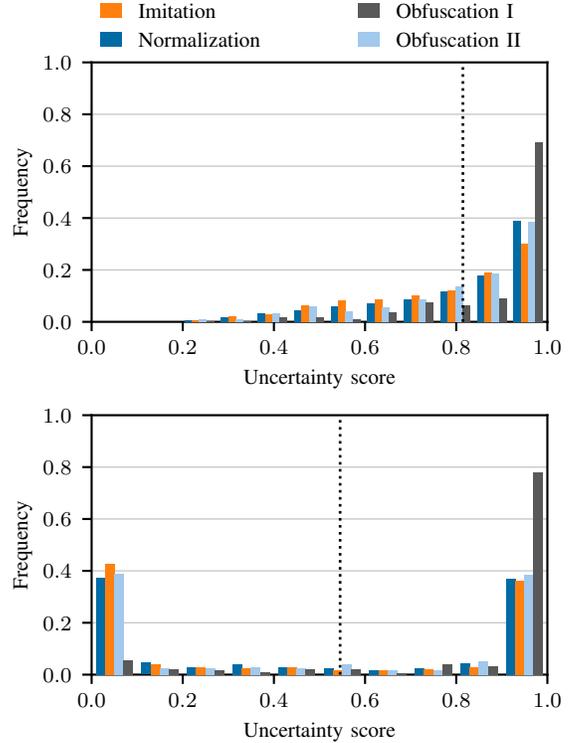


Figure 6: Distribution of uncertainty scores for the adaptive attribution. Top: Method of Caliskan et al. Bottom: Method of Abuhamad et al.

A. Understanding Attribution

We follow two strategies to understand why an attribution is still possible after a candidate technique has been applied to a program. The resulting explanation methods are referred to as *feature highlighting* and *occlusion analysis*.

1) *Feature highlighting*: A simple yet effective way to explain an attribution is to trace back the decision of the employed learning algorithm to individual features of the code. To this end, we adjust the feature extraction of the attribution methods and collect the *code regions* associated with each feature. For AST-based lexical and syntactic 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.

Based on this mapping from features to code regions, we apply *explanation methods* for machine learning to trace back the attributions to code regions [30]. For example, for the random forest classifier employed by Caliskan et al., we use the method *TreeInterpreter* [31], which traverses the trees of the forest, identifies active leaves and returns their contribution to the prediction. We then color the code regions based on this relevance. Figure 7 exemplifies the explanation for a code snippet from our evaluation after applying Stunnix. The header 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.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 (...)
5 int main() {
6     int zf6b4214bfd, zfad4c462ea;
7     int zcfadb12be1, z078a83f597;
8     int zad2aac68b7;
9     int max;
10    scanf("%x25%x64", &zfad4c462ea);

```

Figure 7: Example of feature highlighting for explaining an attribution. Darker shading indicates more relevance.

2) *Occlusion analysis*: Feature highlighting is particularly effective for explaining the attribution of normalized, imitated or mildly obfuscated 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 an analysis of the attribution. To address this problem, we introduce a second method that we denote as *occlusion analysis*. This method is inspired from computer vision, where classification decisions are often explained by occluding regions of an image [32]. Similarly, we occlude different areas of the source code and observe how the obfuscated code is then attributed.

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.

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: $R \leftarrow (0, \dots, 0) \in \mathbb{R}^{|S|}$ ▷ Initialize relevance vector
- 4: **for all** $s \in S$ **do**
- 5: $p^s \leftarrow \text{OCCLUDESEGMENT}(p, s)$
- 6: $c^s \leftarrow \mathcal{A}(\mathcal{Y}(p^s))$ ▷ Attribution w/o segment s
- 7: $R_s \leftarrow (c_t^* - c_t^s)$ ▷ Relevance of segment s
- 8: **end for**

While the method is simple and can be applied to arbitrary anonymization techniques, the crux is the definition of an appropriate segmentation. In contrast to the pixels of an image, we cannot simply remove arbitrary parts of a program without affecting its syntax. To address this problem, we introduce two complementary strategies: As the first strategy, we simply split the source code along the textual lines. While this trivial approach naturally leads to incorrect syntax, often the remaining code is still valid and we can narrow down relevant regions at the level of lines. As the second strategy, we employ *backward program slicing*. To this end, we use the framework *Frama-C* [33] which enables creating syntactically correct program slices on C code. This strategy preserves the syntax in all cases, yet the segments often become large, making an identification of relevant regions difficult.

B. Identified Code Clues

With the help of the explanation methods, we investigate the deficits of the candidate techniques. After manually analyzing the highlighted code regions with both segmentation strategies, we can identify four recurring groups of patterns that remain after the application of the techniques.

1) *String literals*: The first group of patterns corresponds to string literals. Code normalization and coding style imitation do not modify these literals, as they aim at preserving the code’s readability. Although Stunnix replaces strings with hexadecimal representations, the encoded characters still remain the same for all string occurrences. Therefore, a learning algorithm can identify these literals and 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. First, there is an empty function stub for every obfuscated string in the modified code, which signals the number of used strings to the attribution method. Second, the length of the strings is implicitly reflected in the size and operations of the generation routine. As a result, even for Tigress, some subtle hints about string literals remain accessible to the attribution methods.

2) *Include directives*: Another group of indicative patterns is formed by `#include` directives of C code. These directives reveal a developer’s preferences for certain functions and libraries. The code normalization and the obfuscator Stunnix do not touch these directives and thus expose these preferences to the attribution methods, as also highlighted by the example in Figure 7. The coding style imitation by Quiring et al. [11] inserts and removes include directives to match the programming habits of a developer. Nevertheless, headers required for the implementation always remain present in the code. Finally, Tigress “inlines” the headers by copying their content into the source code. While this copying 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.

3) *API Usage*: API usage provides another set of patterns that remains after anonymization. Automatically changing API usage 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 kind of feature. Tigress calls the API functions by their memory addresses, so that it can hide function names in the source code. Nevertheless, an attribution method can use the types and number of call parameters to narrow down the particular API. In Appendix A, we provide a more detailed analysis of this remaining feature.

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 of the code stays unchanged. As a result, personal preferences to structure the program are available to the attribution methods.

Table IV: Performance of Tigress with eliminated clues in the adaptive attribution scenario (adversarial training). The numbers in brackets show the performance difference to the results on unmodified code.

Candidate technique	Attribution method	Accuracy	Std. dev.	Uncertainty score
Obfuscation I (Tigress)	Caliskan et al.	0.071 (-0.617)	0.052 (-0.058)	0.969 (+0.128)
	Abuhamad et al.	0.058 (-0.696)	0.050 (-0.019)	0.938 (+0.677)

C. 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.

1) *Code transformations*: To eliminate the identified patterns, we design a set of code transformations that addresses the weak spots of Tigress. This development is repeated over multiple iterations where we first apply new transformations and then observe their impact on the attribution using the explanation methods from Section V-A. 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. In C, this can be easily realized by adding a NULL byte to each string followed by padding characters. 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. Finally, for function pointers, we remove all information, except for necessary argument and return types. This makes it complicated to differentiate the called functions.

2) *Results*: Table IV shows the attribution performance after applying these improvements and conducting another run of adversarial training. We observe a significant decrease in accuracy compared to Table I and Table III. The values are close to guessing and the uncertainty scores are comparable to the static scenario (see Table I). For the method of Caliskan et al., the score reaches the threshold of 0.96, so that we attain practical k -uncertainty according to the heuristic from Section III-E. For the method of Abuhamad et al., we come close to this threshold with an uncertainty value of 0.94.

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.

VI. LIMITATIONS

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

1) *Selection of techniques*: For our experiments, we select two attribution methods and four anonymization techniques. Consequently, our results are mainly based on this particular choice. However, a different selection likely would have only a minor impact on the reported performance and our conclusions due to the following reasons:

- a) We consider two state-of-the-art attribution methods. While other approaches would also be applicable [e.g., 3, 4, 34], none of these is fundamentally different in design. All methods extract layout, lexical, and syntactic features for training a classifier. Since the two considered methods already substantially weaken the anonymization, we conclude that adding more attribution methods to our evaluation would not provide new insights, unless they build on completely new attribution strategies.
- b) For code anonymization, we consider common approaches for reducing coding style and comprehension of source code. With Tigress, we employ one of the most powerful obfuscators available for C code [35, 12, 26]. Our analysis in Section V demonstrates how this obfuscation can be further improved through explanation methods to realize k -uncertainty in a controlled environment. The four selected techniques thus provide a broad view on current defenses against authorship attribution.

2) *Size and type of source code*: We base our empirical analysis on a dataset from the Google Code Jam competition. The underlying C code is ideally suited for studying attribution techniques, as several developers solve the exact same tasks, and differences in their solutions result from coding style. However, compared to prior work, we focus on a small dataset with only 30 authors. 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. Nonetheless, previous work has shown that learning-based attribution methods scale well with the number of authors [see 6, 7], so that stronger anonymity cannot be expected. Note also that the number of considered authors is defined by the attacker and not controllable by the defender.

Moreover, we focus on C code because it is widely used in software development of libraries and operating systems. Still, we note that interpreted languages, such as Python and JavaScript, offer further strategies for anonymization, including encrypting the entire code and unpacking it at runtime using the interpreter. Yet, there exists a large series of research on unpacking malicious code [36, 37] that is applicable and would reveal

the original code. Therefore, investigating other languages and types of execution—compilation vs. interpretation—unlikely changes the overall result of our work.

3) *Undecidability*: The proposed concept of k -uncertainty inherits a daunting property from k -anonymity: *undecidability*. It is 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 a notable advantage that k -anonymity cannot offer. The concept involves a tunable confidence range ϵ . By making this range a measurable quantity, we create the uncertainty score that allows us to compare existing methods—something that would not be possible with k -anonymity. Moreover, k -uncertainty converges to k -anonymity with decreasing values of ϵ , giving us a general intuition on the attained protection level.

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

VII. RELATED WORK

Our work is the first to explore the problem of anonymizing source code. For this investigation, 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 research branches.

A. Code Authorship Attribution

1) *Code stylometry*: The starting point for our work has been the remarkable progress in code stylometry, that is, the automatic authorship attribution of source code. In recent years, 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., 6, 7, 15, 38]. These methods have been further extended to allow for attributing code fragments developed by multiple authors [e.g., 4, 39, 3]. This partial attribution, however, proves challenging and therefore leads to lower detection rates. For this reason, we focus our empirical analysis on attribution methods analyzing single-author code. In this way, we evaluate techniques for protecting code under a stronger adversary model.

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. [1], manual modifications have been used to mimic the style of developers. Following work has then developed concepts for automatically create adversarial examples of source code [11, 25, 17]. These attacks differ in the employed code transformations and search strategy. For example, Quiring et al. [11] and Liu et al. [25] develop several code transformations that modify lexical and syntactic features, such as converting for-loops to while-loops, whereas the approach by Matyukhina et al. [17] targets only layout features. For our evaluation, we focus on the attack by Quiring et al. [11], as it has a higher evasion rate than the method of Liu et al. [25] and allows changing various lexical and syntactic features in C code.

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 modelling and detecting patterns in writing style [e.g., 40, 34, 41] as well as approaches for misleading an attribution through writing style obfuscation [e.g., 42, 43, 44]. Our work shares inspiration from this work. Due to the fundamentally different properties of natural language text and source code, however, these approaches are not directly applicable in our setting.

B. Data Anonymization

Another related area of research is the anonymization and de-anonymization of data [e.g., 45, 20, 46, 18]. Early ideas of this area originate from general data processing and tackle the challenges of storing, analyzing, and exchanging privacy-sensitive data, such as medical records.

1) *Anonymity concepts*: One of the first ideas from this area is the concept of k -anonymity by Sweeney [18]. In this concept, every quasi-identifier in a dataset 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 [20]. 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 about this particular person. This concept was further improved by t -closeness [21], which tackles the problem of information disclosure through the different distributions of attributes in an equivalence class and the overall data. This concept limits the possible gain of knowledge by requiring 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, as already k -anonymity is incomputable on source code if the attribution method is unknown.

2) *Differential privacy*: Finally, our work also relates to the powerful concept of *differential privacy* [47]. 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., 48, 49, 50, 51, 52] and also in the field of natural language processing [e.g., 53, 54, 55, 56].

In natural language processing, the noise is usually not added to the text itself, but to a vector representation of it [53, 54, 55]. While this is an elegant approach for realizing differential privacy, in our setting, this requires knowledge and access to the feature representation used by the attacker. As the concrete set of features is typically unknown to the defender and not accessible, these approaches are not directly applicable to protect developers from identification.

VIII. 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.

ACKNOWLEDGMENTS

The authors gratefully acknowledge funding from the German Federal Ministry of Education and Research (BMBF) under the projects BIFOLD (ref. 01IS18025A and ref 01IS18037A) and IVAN (ref. 16KIS1167), as well as funding from the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy EXC 2092 CASA-390781972.

REFERENCES

- [1] 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)*, vol. 2018, no. 1, pp. 127–144, 2018.
- [2] V. Kalgutkar, R. Kaur, H. Gonzalez, N. Stakhanova, and A. Matyukhina, "Code authorship attribution: Methods and challenges," *ACM Computing Surveys*, vol. 52, no. 1, pp. 1–36, 2020.
- [3] E. Bogomolov, V. Kovalenko, Y. Rebyrk, 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)*, 8 2021, pp. 932–944.
- [4] 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)*, vol. 2019, no. 3, pp. 389–408, 2019.
- [5] S. Alrabae, 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)*, 2018, pp. 26–47.
- [6] 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*, 2015, pp. 255–270.
- [7] 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)*, 2018, pp. 101–114.
- [8] R. Dingledine, N. Mathewson, and P. F. Syverson, "Tor: The second-generation onion router," in *Proc. of the USENIX Security Symposium*, 2004, pp. 303–320.
- [9] 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.
- [10] 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.
- [11] E. Quiring, A. Maier, and K. Rieck, "Misleading authorship attribution of source code using adversarial learning," in *Proc. of the USENIX Security Symposium*, 2019, pp. 479–496.
- [12] C. Collberg, "The Tigress C Obfuscator," Project website: <https://tigress.wtf>, accessed April, 2022.
- [13] "C/C++ Obfuscator," Project website: <http://stunnix.com/prod/cxxo/>, accessed April, 2022.
- [14] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, 2nd ed. Addison-Wesley, 2006.
- [15] 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)*, 2017, pp. 65–82.
- [16] B. N. Pellin, "Using classification techniques to determine source code authorship," Department of Computer Science, University of Wisconsin, Tech. Rep., 2000.
- [17] 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)*, 2019, pp. 291–302.
- [18] L. Sweeney, "k-anonymity: A model for protecting privacy," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 10, no. 5, pp. 557–570, 2002.
- [19] R. Goldblatt and M. Jackson, "Well-structured program equivalence is highly undecidable," *ACM Transactions on Computational Logic (TOCL)*, vol. 13, no. 3, 2012.
- [20] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkatasubramanian, "l-diversity: Privacy beyond k-anonymity," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 1, no. 1, pp. 3–es, 2007.
- [21] 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*, 2007, pp. 106–115.
- [22] "Linux kernel coding style," <https://www.kernel.org/doc/html/v4.10/process/coding-style.html>, accessed April, 2022.
- [23] "Chromium coding style," <https://www.chromium.org/developers/coding-style/>, accessed April, 2022.
- [24] "Firefox coding style," <https://firefox-source-docs.mozilla.org/code-quality/coding-style/index.html>, accessed April, 2022.
- [25] Q. Liu, S. Ji, C. Liu, and C. Wu, "A practical black-box attack on source code authorship identification classifiers," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 3620–3633, 2021.
- [26] J. Nagra and C. Collberg, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*, 1st ed. Addison-Wesley, 2009.
- [27] "Google code jam," <https://codingcompetitions.withgoogle.com/codejam>, accessed April, 2022.
- [28] "Clang: C language family frontend for LLVM," LLVM Project, <https://clang.llvm.org>, 2018.
- [29] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *International Conference on Learning Representations (ICLR)*, 2015.
- [30] 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*, 2020, pp. 158–174.
- [31] A. Saabas, "Treeinterpreter," Project repository: <https://github.com/andosa/treeinterpreter>, accessed April, 2022.

- [32] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *Proc. of the European Conference on Computer Vision (ECCV)*, 2014, pp. 818–833.
- [33] 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.
- [34] S. H. H. Ding, B. C. M. Fung, F. Iqbal, and W. K. Cheung, "Learning stylometric representations for authorship analysis," *IEEE Trans. Cybern.*, vol. 49, no. 1, pp. 107–121, 2019.
- [35] 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*, 2017, pp. 661–678.
- [36] 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*, 2015, pp. 659–673.
- [37] 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*, 2012, pp. 443–457.
- [38] 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*, 2018, pp. 74–84.
- [39] M. Tereszowski-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)*, vol. 2022, no. 1, pp. 126–147, 2022.
- [40] 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*, 2014, pp. 185–205.
- [41] 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*, 2014, pp. 212–226.
- [42] M. Brennan, S. Afroz, and R. Greenstadt, "Adversarial stylometry: Circumventing authorship recognition to preserve privacy and anonymity," *ACM Transactions on Information System Security*, vol. 15, no. 3, pp. 12:1–12:22, 2012.
- [43] 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)*, vol. 2019, no. 4, pp. 54–71, 2019.
- [44] 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)*, vol. 7384, pp. 299–318, 2012.
- [45] A. Narayanan and V. Shmatikov, "Robust de-anonymization of large sparse datasets," in *Proc. of the IEEE Symposium on Security and Privacy*, 2008, pp. 111–125.
- [46] A. Haeberlen, B. C. Pierce, and A. Narayan, "Differential privacy under fire," in *Proc. of the USENIX Security Symposium*, 2011.
- [47] C. Dwork, "Differential privacy," in *Automata, Languages and Programming*, 2006, pp. 1–12.
- [48] K. Chaudhuri, C. Monteleoni, and A. D. Sarwate, "Differentially private empirical risk minimization," *Journal of Machine Learning Research*, p. 1069–1109, 2011.
- [49] 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)*, 2016, pp. 26–37.
- [50] 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)*, 2016, pp. 308–318.
- [51] 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)*, 2017, pp. 1389–1406.
- [52] B. Jayaraman and D. Evans, "Evaluating differentially private machine learning in practice," in *Proc. of the USENIX Security Symposium*, 2019, pp. 1895–1912.
- [53] 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*, 2018, pp. 305–314.
- [54] 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*, 2020, pp. 2355–2365.
- [55] 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)*, 2021, pp. 468–475.
- [56] J. Mattern, B. Weggenmann, and F. Kerschbaum, "The limits of word level differential privacy," in *Findings of the Association for Computational Linguistics: NAACL*, 2022, pp. 867–881.

APPENDIX A 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 one 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))` which are easily distinguishable. As a result, even for Tigress, an attribution method can identify used library functions in this case.

APPENDIX B
NORMALIZATION RULES

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

Table V: 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 <code>var_x</code> with <code>x</code> being a number starting at 0.
Types	The used data types are mapped to a specified subset to eliminate redundant type names, such as <code>long</code> and <code>int_32</code> . Note that this transformation is platform-specific.
Switch2If	<code>switch</code> statements are transformed to a chain of <code>if-else</code> 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. <code>a += 2</code> is transformed into <code>a = a + 2</code> .
IfElse	If the last statement inside the body of an <code>if</code> statement is for example a <code>return</code> or <code>break</code> , the following code is moved into an <code>else</code> to this <code>if</code> .
MainParams	This rule enforces the use of two parameters for the <code>main</code> function and a <code>return</code> 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 <code>if</code> and <code>for</code> statements.
UnnecessaryReturn	This rule removes <code>return</code> statements in <code>if</code> bodies if all following code is in the <code>else</code> clause and the function has no return value.
VoidReturn	This rule adds a <code>return</code> statement at the end of every <code>void</code> function.
FlattenIf	For nested <code>if</code> statements, this rule removes inner clauses by combining the conditions of the inner and outer <code>if</code> . It inserts additional <code>ifs</code> for the <code>else</code> parts.
Paren	This rule removes unnecessary parentheses like in <code>a = (b + c)</code> , simplifying arithmetic expressions

APPENDIX C
USED TRANSFORMATIONS FOR TIGRESS

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

Table VI: 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=*