SHIELD: Thwarting Code Authorship Attribution

Mohammed Abuhamad, *Senior Member, IEEE*, Changhun Jung, *Member, IEEE*, David Mohaisen, *Senior Member, IEEE*, and DaeHun Nyang, *Member, IEEE*

***Abstract*—Authorship attribution has become increasingly accurate, posing a serious privacy risk for programmers who wish to remain anonymous. In this article, we introduce SHIELD to examine the robustness of different code authorship attribution approaches against adversarial code examples. We define four attacks on attribution techniques, which include targeted and non-targeted attacks, and realize them using adversarial code perturbation. We experimented with a dataset of 200 programmers from the Google Code Jam competition to validate our methods. We target six state-of-the-art authorship attribution methods that adopt various techniques for extracting authorship traits from source code, including RNN, CNN, and code stylometry. Our experiments demonstrate the vulnerability of current authorship attribution methods against adversarial attacks. For the non-targeted attack, our experiments demonstrate the vulnerability of current authorship attribution methods against the attack with an attack success rate exceeding 98. 5% accompanied by a degradation of the identification confidence exceeding 13%. For targeted attacks, we show the possibility of impersonating a programmer using targeted adversarial perturbations with a success rate ranging from 66% to 88% for different authorship attribution techniques under several adversarial scenarios.**

***Index Terms*—Code authorship identification, machine learning, code transformation, identification evasion.**

I. Introduction

CODE authorship attribution is the process of recognizing programmers of a given software, and there have been several works on robust and scalable attribution [4], [6], [8], [13], [24]. These methods have shown that programmers can be accurately identified by their coding style, making this problem an easy task thanks to the rapid development of code analysis and machine learning techniques. Accurate attribution benefits software forensics and security, especially in identifying malicious code programmers, detecting plagiarism, and settling authorship disputes. However, the process also poses privacy risks for programmers who prefer to remain anonymous.

Received 25 September 2023; revised 10 March 2025; accepted 11 March 2025. Date of publication 21 March 2025; date of current version 4 September 2025. This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) under Grant RS-2023-00222385, and in part by the Korea Government (MSIT) under Grant 2023R1A2C2006373. *(Corresponding author: DaeHun Nyang.)*

Mohammed Abuhamad is with the Department of Computer Science, Loyola University Chicago, Chicago, IL 60611 USA.

Changhun Jung and DaeHun Nyang are with the Department of Computer Security, Ewha Womans University, Seoul 03760, South Korea (e-mail: nyang@ewha.ac.kr).

David Mohaisen is with the Department of Computer Science, University of Central Florida, Orlando, FL 32816 USA.

Digital Object Identifier 10.1109/TDSC.2025.3553753

Recent code authorship attribution techniques heavily use machine learning models. While effective, those techniques are prone to manipulations that force the identification models to generate specific desired outputs, e.g., misclassification. One line of work for general machine learning algorithms utilized small perturbations in the input domain, resulting in adversarial examples (AEs) that are similar to the original, making it difficult to distinguish them and posing a significant threat to machine learning models [16], [25], [31]. Examining AEs in the context of code authorship attribution has received limited attention despite their importance to assess robustness of these systems.

This work introduces SHIELD to generate AEs at the source code level to prevent attribution while preserving the functionality of the code. Such AEs will fool a classifier into misidentifying programmers and lead to targeted attacks, e.g., imitation or mimicking. Investigating such capabilities by examining how prone authorship attribution is to practical AEs allows for a finer understanding of the state-of-the-art methods and helps address their shortcomings, especially with the increasing reliance on them to identify the coding style of programmers for security applications [7], [8], [13]. As a byproduct, our attacks can serve as a building block to maintain privacy of programmers in the presence of attribution models.

SHIELD examines both the non-targeted and targeted attacks. In the non-targeted attacks, known as *confidence reduction* or *misclassification* attacks, SHIELD manipulates the input source code so that the identification model outputs any other author, i.e., *authorship dodging* [27], [29], where SHIELD can use this strategy to conceal the code author identity. In the targeted attack, SHIELD manipulates the input so that the identification model outputs a specific *target* author, i.e., *authorship imitation* or *evasion*, depending on the adversarial capability and objective. We apply those scenarios to various authorship attribution methods with SHIELD for an in-depth analysis of each method.

Although potentially extendable to binaries [14], SHIELD targets the attribution of the source code author for its prevalence. We note that source code-level attacks are challenging since the generated AEs should be syntactically correct, should preserve the code functionality, and should not be easily detected. Although these attacks can be conducted using code transformation [20], [22], [27], a process similar to author obfuscation in which authorship traits are hidden in the transformed code, code transformation techniques require analyzing and changing the code to the target features in various categories, including layout, lexical, syntactic, control flow and data flow features. However, a code perturbation approach provides an effective alternative for targeted and non-targeted attacks, by generating

AEs to meet a specific goal without targeting the features of different categories.

Conventionally, adversarial perturbations are applied directly to the input source and not to the feature space, e.g., perturbations in an image, not in the features extracted from that image. However, the code authorship attribution techniques are typically based on the authorship traits extracted from code, assuming a closed system where the feature extraction is not manipulated and requires perturbations at the code level. Therefore, attribution attacks should be designed explicitly using perturbations applied directly to the code. SHIELD injects carefully chosen code samples into the target source code and then obfuscates it using an off-the-shelf obfuscator. Unlike the prior work, where obfuscation is used to conceal the identity of the code author, we use obfuscation to make it difficult for the adversary to recognize or remove the injected code parts statically. For example, consider a simple code block:

```
if ((x*0 + 1) * sin(M_PI) +
    cos(2*M_PI) < -0.1){ // False
    condition
    int x = injected_fn1();
    injected_fn2(x);
    }
```

Static analysis tools face several challenges detecting this injected code. The condition uses complex expressions making *unreachability* determination difficult. The code follows valid syntax rules and scope boundaries. Static analyzers would need to evaluate trigonometric functions with constants to determine the condition is always false. Code obfuscation could further obscure unreachability.

Our threat model is more restrictive than without adopting obfuscation because code attribution must also be capable of identifying obfuscated source codes. However, it is well known that attribution models work even in the obfuscated domain [4], [13]. We note that, while our AEs are generated at the code level, the eventual effect of the injection will be perturbations in the feature space. For convenience, we use the term perturbation to refer to injection.

Contributions. Our key contributions are as follows. (1) We introduce SHIELD , a simple yet effective approach for generating AEs on code authorship attribution. The proposed approach does not change the original code, but adds carefully crafted code blocks (i.e., perturbations) to alter the authorship attributes of the code, leading to authorship dodging and imitation. (2) We provide a comprehensive evaluation of our technique against six authorship attribution methods: DL-CAIS [4], WE-C-CNN, WE-S-CNN, TF-IDF-C-CNN, and TF-IDF-S-CNN [7], and Code Stylometry [13]. Our evaluation features a large-scale analysis of code authorship robustness under various adversarial scenarios using a dataset of 200 programmers. Our approach achieved a misidentification rate exceeding 98.5% for non-targeted attacks while significantly reducing the confidence of the model output. We also demonstrate imitation attacks at a success rate of more than 66% for all targeted systems when using sufficient perturbations and at 88% when the adversary has access to samples of the targeted author.

Organization. In Section II, we provide an overview of the authorship attribution workflow along with details of the six targeted systems. In Section III, we introduce SHIELD and our attack strategy.In Section IV, we present non-targeted attacks and their experimental results, while Section V covers targeted attacks and results. Section VI provides discussion and limitations. In Section VII, we review the related work. We conclude our work in Section VIII.

II. Code Authorship Attribution

A. Authorship Attribution Workflow

The workflow of a typical code authorship attribution system includes four stages: pre-processing, feature extraction, feature selection, and authorship identification.

The *data pre-processing* entails removing undesired parts from the code, e.g., comments, links to external resources, *etc.*, and normalizing the layout features, e.g., white spaces and lines. The *feature extraction*, also known as authorship attributes extraction, requires defining distinctive coding traits. These traits are features that capture specific characteristics of different programming styles. Typically, the features may include some or all of the following: (i) syntactic features in the program structure and implementation choices, (ii) stylometric features in variables naming, documentation, language reserved keywords usage, *etc.*, (iii) layout features, e.g., whitespace characters and indentations, (iv) development environment features, e.g., platforms, editors, programming languages,*etc.* Feature extraction entails computing those features from the code.

The number of candidate features signifies the *feature selection* process. The feature selection includes evaluating authorship attributes based on, for example, the information gain or mutual information between attributes and authors to determine the prominent features. The final stage is the *authorship identification*, which is usually treated as a classification problem. Using the extracted (or selected) features, (supervised) classification models—such as the Support Vector Machine (SVM) [11], [26], Random Forest Classifier (RFC) [4], [13], or neural networks [11]—are trained to identify programmers.

The workflow of authorship attribution is shown in Fig. 1, highlighting the methods used by the six targeted approaches: Code Stylometry (CSFS) [13] Deep Learning-based Code Authorship Identification System (DL-CAIS) [4], TF-IDF-based concatenated CNN (TFIDF-C-CNN) [7], TF-IDF-based stacked CNN (TFIDF-S-CNN) [7], word embedding-based concatenated and stacked CNN (WE-C-CNN and WE-S-CNN, respectively) [7].

B. Our Baseline and Implementation Details

In the following, we review the baseline implementation of the six targeted authorship identification techniques.

DL-CAIS. The system [4] uses the Long-Short-term Memory (LSTM) technique to learn deep authorship feature representations. The supervised feature learning model is trained

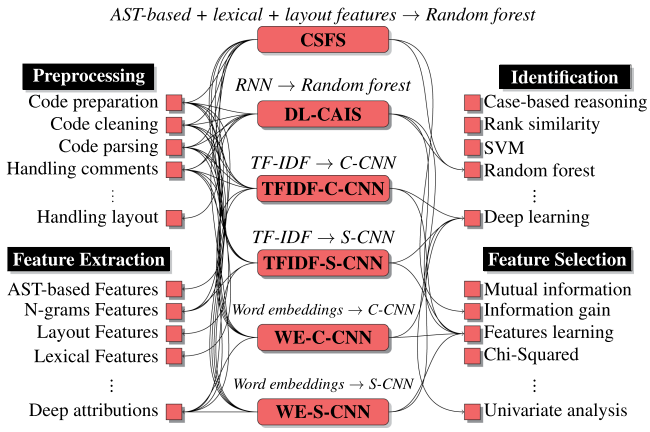


Fig. 1. The workflow of code authorship attribution process, including the pre-processing, feature extraction, feature selection, and identification.

to relate the input data to the output labels, where the input in DL-CAIS is the code files initially represented as TF-IDF vectors. That is, the input vector for a document d_j to the deep learning model is represented as $[\text{TF-IDF}(t_1, d_j, \mathcal{D}), \text{TF-IDF}(t_2, d_j, \mathcal{D}), \ldots, \text{TF-IDF}(t_n, d_j, \mathcal{D}))]$, where n is the total number of terms in the corpus $\mathcal{D}$.

For dimensionality reduction, DL-CAIS uses the order of term frequency to select features. For every term t_i and document d_j, DL-CAIS calculates $x_i = \bigcup \text{TF-IDF}(t_i, d_j, \mathcal{D}) \quad \forall\, j : 1 \leq j \leq \|\mathcal{D}\|$, where $\cup$ is a feature selection operator.

Following the implementation in the original work obtained from [4], we used the top 2,500 TF-IDF features as a single-step vector representation to the LSTM model to learn the deep authorship features. The LSTM architecture includes three layers of 128 LSTM units followed by three fully-connected layers with 1024 units connected to a softmax layer with the number of programmers (classes). In LSTM training, a model is created using input code samples and the corresponding authors (pairs) by minimizing the softmax cross-entropy loss. The LSTM model is then used to generate deep authorship feature representations of code samples as the output of the second-last layer. Those features are used to build an RFC with 150 trees grown to the maximum extent.

Code Stylometry. Caliskan-Islam et al. [13] introduced the Code Stylometry Feature Set (CSFS) for authorship attribution using three sets: lexical and layout features extracted directly from the source code, and syntactic features extracted from the Abstract Syntax Tree (AST) of the source code. Lexical features express the programmer's preferences for using functions, nesting depth, and specific expressions and keywords, while layout features are statistics about whitespace characters and indentation. Syntactic features represent properties of the program structure, e.g., the depth and frequency of an AST node, node unigrams and bigrams frequency *etc.* In total, CSFS generated +120 k features for 250 authors with nine files each. This requires a feature selection process, which is done using information gain by scoring features based on the difference in the Shannon entropy of classes' distribution and the conditional distribution of classes given a feature. The feature score is $\text{IG}(y, x_i) = E(Y) - E(y|x_i)$, where y is a class label, E is the Shannon entropy, and x_i is the i-th feature.

The resulting features with non-zero gain are denoted by IG-CSFS. The top-k features are used. Using the implementation of Caliskan-Islam et al. [13], the input code samples are represented with the top 1024 IG-CSFS authorship features to build RFC with 150 trees grown to the maximum extent.

CNN-based Systems. Abuhamad et al. [7] used word embedding and TF-IDF as input methods for CNN-based models with various architectures. For word embedding, a code sample $d_j \in \mathbb{R}^{1\times(n\times m)}$ is represented as a sequence of expressions, $d_j^{t_1:t_n} = t_1 \oplus t_2 \oplus \cdots \oplus t_n$, where $t_i \in \mathbb{R}^m$ is the representation of the i-th term in d_j and $\oplus$ is the concatenation operator. A predefined sequence length n was used for code samples where short sequences are padded and long sequences are truncated. For the TF-IDF representation, a similar approach was adopted as in DL-CAIS [4].

For identification, Abuhamad et al. [7] used two CNN architectures, concatenated and stacked. Concatenated CNN includes concatenating the feature maps of a number of convolutional layers, while stacked CNN follows the typical CNN model by applying convolutional layers on top of each other to generate feature maps based on features of the previous layer. For both architectures, a softmax classifier is used—the softmax function, defined as $\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j \forall\, j}}$, signifies the probability of classifying the class at the i-th position among all class labels—and the output is the class with the highest probability. The implementation details of targeted CNN-based systems are as follows.

- *WE-C-CNN:* We set the word embedding matrix to generate 128-word representations. The input code samples are then represented as a matrix of $n \times 128$ representations, where n is the number of terms in a code sample. We set $n = 256$ to fix the length representations to the convolutional layers as short samples are padded with zeros, and long samples are truncated. For the convolutional layers, three layers of 128 filters of different sizes (3, 4, and 5) are adopted to receive the input representation. The outputs of the three layers are concatenated and connected to a softmax layer of the same size as the number of considered classes.
- *WE-S-CNN:* Similar to the WE-C-CNN settings, the code samples are represented as $d_j \in \mathbb{R}^{256\times128}$ since the word embeddings are set to be 128 vector representations. The CNN model architecture follows the typical stacked CNN layers with three consecutive layers, each with 128 filters of different sizes (3, 4, and 5). For this architecture, we used max-pooling after each layer to reduce the size of feature maps produced by the convolutional layer. The last max-pooling layer is connected to the output layer.
- *TFIDF-C-CNN:* The input code samples are represented with the top 2500 TF-IDF features and fed to 1-dimensional convolutional layers. Similar to WE-C-CNN, we use three filter sizes (3, 4, and 5) and 128 filters per layer. The feature maps produced by the three convolutional layers are then concatenated and connected to a softmax layer of the same size as the number of classes.

- *TFIDF-S-CNN:* Similar to TFIDF-C-CNN, input code samples are represented with the top-2500 TF-IDF features and fed to three stacked 1-dimensional convolutional layers. All layers consist of 128 filters, and each layer has a different filter size. We used max-pooling for the stacked CNN architecture and the last max-pooling layer is connected to a softmax layer with the same size as that of the number of classes.

Deep Learning Training. Both RNN-based and CNN-based models are trained using the Adam optimizer with a static learning rate of 10^{-4} to minimize the softmax-cross-entropy loss since all models are trained in a supervised manner. In all cases, the training process is terminated after 1,000 iterations. To prevent overfitting, we used *dropout regularization* with keep-rate of 70% and *L2 regularization* with λ regularization strength of 10^{-3}. Moreover, we used a minibatch approach with a minibatch size of 64 observations in the training process of all deep learning-based architectures.

III. SHIELD: Methods

This section describes our proposed attack strategy and the methods employed for adversarial code perturbation.

A. The Adversarial Attack Strategy

Our robustness assessment of the authorship attribution systems includes exploring the characteristics of their behavior in adversarial settings. SHIELD introduces code perturbations to implement four threat models categorized by their target specificity and adversarial knowledge. All attacks are implemented in the *Closed box* threat model.

The adversary starts by defining the source of perturbation, i.e., code statements, expressions, variable names, *etc.*, to be used to generate the adversarial example. Based on the threat model, selecting the source of perturbations can be through random selection or targeted selection. This study investigates four threat models, namely the non-targeted attacks (Section IV) and the targeted attacks T1 through T3 (Section V). The non-targeted attacks and targeted attacks T1 use code perturbations that are generated through a random selection of code statements and expressions, while the targeted attacks T2 and T3 use a targeted selection of code perturbation. The code perturbations are then generated from a template library that includes code statements and blocks that follow the syntax rules of the programming language (e.g., C++ in our experiments).

We designed a library of code templates with a list of operational and control statements. Using the perturbation expressions, the code statements produced by the templates are injected directly into the original code or used as basic units for a larger code block, e.g., functions/methods. More details of the code perturbation are given in Section III-B.

After adding the code perturbation to the code sample to defeat identification, SHIELD aims to hide the added perturbation using *code-to-code obfuscation* [2], [3]. Assuming working with powerful authorship identification models that can identify and detect authorship traits despite obfuscation, SHIELD incorporates obfuscation to hinder the detectability of perturbation and easy-to-perform static analysis. Previous studies [4], [13] have shown that identifying authors of obfuscated code can be accurately achieved.

The AE is fed into the code authorship system for feature extraction and identification. Since SHIELD assumes a *Closed box* model, it optimizes the perturbation by repeatedly querying the system and observing the changes in the output distribution. Using the identification model's probability distribution, SHIELD optimizes the perturbation through changes in the code expressions and statements used to generate the attacks on the attribution system.

The number of queries that SHIELD performs in order to succeed depends on the threat model, SHIELD 's specific objectives, and level of knowledge. We describe the threat models in detail in Section IV-A (non-targeted attacks) and Section V-A (targeted attacks). We observed that launching successful non-targeted attacks, even with small random perturbations, requires fewer rounds of queries with the identification model compared to the targeted attacks.

For evaluation, we defined three threat models for the *targeted attacks.* T1: This model requires a random selection of perturbations for targeted attacks. T2: This model requires access to two samples of the target programmer that are not included in the training dataset used for the baseline model. T3: This model extends adversary knowledge to code samples from a suspect set of programmers. The adversarial objective of T3 is to generate code samples that imitate the style of the closest programmer.

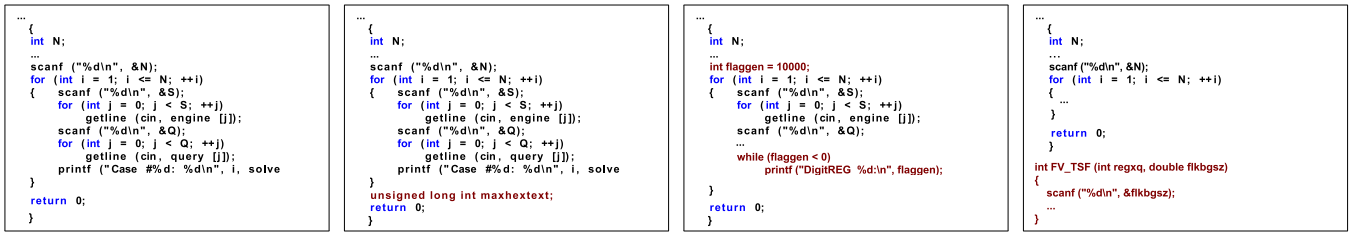
The adversary gains more capabilities by accessing samples from the target set using T2 and T3. Such capabilities include observing the most representative features of the target and improving the code perturbation to meet an adversarial goal. For T2, the adversary applies a targeted selection of perturbations (i.e., code statements, variable names, *etc.*) based on the target's samples. The adversary applies the targeted selection to samples from multiple programmers to launch T3.

Fig. 3 shows the adversarial capabilities and objectives of the different threat models explored by SHIELD categorized by the attack specificity (targeted and non-targeted attacks). Note that all scenarios are under the *Closed box* assumption. Therefore, the adversary's capabilities include only *oracle* and/or *samples*. For the oracle access, the adversary has no knowledge about the model, but the ability to access an oracle allows her to conduct queries to the model and infer the relation between inputs and outputs. For sample access, the adversary has access to several samples from the victim that can be used to enhance the adversarial perturbations.

B. Adversarial Code Perturbation

Introducing perturbations on the feature representations assumes a *white-box attack* where the adversary has full control and access to all stages of the system operation, limiting the approach's practicality. Assuming that the adversary has limited access to the system (i.e., *Closed box attack*), we investigate perturbations to the input code directly.

Code Perturbations. Code perturbations can be applied as variable declarations, loops, control statements, functions, *etc.*



(a) Original source code (b) Adversarial example (1) (c) Adversarial example (2) (d) Adversarial example (3)

Fig. 2. Adversarial source-code examples. The code perturbations can be variable declarations (Adversarial example 1), loops and control statements (Adversarial example 2), functions (Adversarial example 3). Note that the added perturbations (in red color) are not meant for execution.

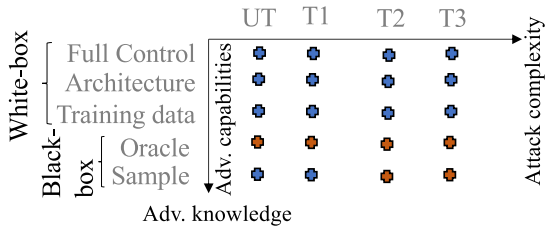


Fig. 3. A general categorization of the attacks from the perspective of the adversary's capabilities and goals. This study explores the orange-colored attacks as they are more relevant to the addressed problem space (UT is untargeted, while T1 through T3 are targeted attacks).

Fig. 2 shows adversarial examples. For an author who tries to evade identification, this approach is implemented by the author with full control over the source code. For code functionality preserving, the added perturbations are not meant for execution, and we achieve that by adding code perturbations as methods (functions) that never get called or executed in the main code, or as negative conditional statements. To enhance the perturbations, we: (1) limit the size of perturbation to the minimal size that enables the attack, (2) ensure that the perturbation code inherits the flow of the syntax rules of the language, (3) ensure that the statements of the adversarial code should maintain the same size as other statements in the original code, and (4) ensure that the variable names should follow the naming rules and convention of the language.

Targeted Programming Language. Our dataset includes code samples written in C++, and the code perturbations are presented as C++ functions that follow the structure of the C++ functions with a function body that includes multiple statements. The six major statement types that we considered in our implementation: (1) Definition and assignment of variables, (2) Arithmetic, relational, and logical operations, (3) *if* and *if-else* statements, (4) *switch* statement, (5) *for* loop statement, and (6) *do-while*, *while* loop statement.

Each statement includes at least one line of code, e.g., a declaration, an assignment, or an operation. The selection of statement types and the naming criteria for variables and functions in the code perturbation are based on the underlying assumptions of the attack. Assuming a Closed box scenario for non-targeted and targeted attacks, variable names are generated randomly with a predefined variable-naming template. However, assuming greater adversarial knowledge in imitation attacks, the considered names can be selected from previously observed used names by the targeted programmer. We note that generating random names might cause an out-of-vocabulary problem for approaches that use TF-IDF or word embeddings for representing the code sample.

IV. Non-Targeted Attack

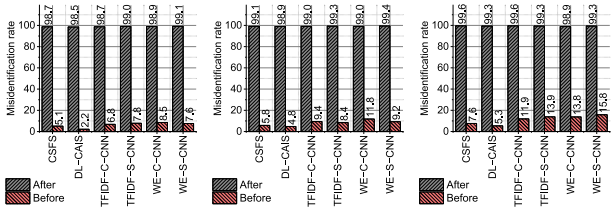
This section describes the non-targeted attacks and shows their impact on the performance of authorship attribution systems. We describe the threat model, explore the baseline performance of the six targeted attribution systems, and then evaluate the impact of the attack on their performance.

A. Threat Model 1: Non-Targeted Attack

For a non-targeted attack, we assume a *Closed box* scenario for accessing the code authorship attribution system. The adversary sends a source code file to the attribution system and receives the predicted output and confidence score. The adversary does not have access to or knowledge about the components of the system, including the training data, the feature extraction/selection methods, and the structure and design of the model. The adversarial objective of a non-targeted attack is to mislead the authorship identification model to predict a wrong programmer. The adversary tries to reduce the confidence of the model through the manipulation of authorship attributes using code perturbation. This attack is known as a *confidence reduction* and *dodging attack*.

Model Definition. The confidence of an identification model is defined as the probability of predicting a programmer given a test sample. Since we are targeting various systems with different classification techniques, the confidence of the targeted approaches is defined as follows. DL-CAIS and CSFS adopt the RFC where confidence is the number of decision trees voting for a given class. For an author $y_i \in \mathcal{Y}$, the confidence in identification is the percentage of trees that voted for y_i to be the programmer who wrote a given code sample $d_j \in \mathcal{D}$, where $\mathcal{Y} : \{y_0, y_1, \ldots, y_m\}$ is the suspect set (of size m) and $\mathcal{D} : \{d_0, d_1, \ldots, d_n\}$ is the collection of samples from a test set (of size n). The confidence is estimated as:

$$\text{conf}(y_i) = \sum_k vote_k(y_i) \times \|T\|^{-1},$$



(a) 50 programmers (b) 100 programmers (c) 200 programmers

Fig. 4. The misidentification rate of targeted systems before and after the attacks using different datasets. The baseline is obtained using 9-fold cross-validation.

where $vote_k(y_i)$ is the k-th tree voting for y_i and $\|T\|$ is the total number of trees in the RFC.

On the other hand, CNN-based models (WE-C-CNN, WE-S-CNN, TFIDF-C-CNN, and TFIDF-S-CNN) adopt the softmax classifier where the confidence/probability is the softmax score for a given class. For an author at the i-th position of $\mathcal{Y}$, the identification confidence to classify a sample d_j using a model f is estimated as: $\text{conf}(y_i) = \text{softmax}_i(f(d_j))$.

The Adversarial Objective. This attack aims to delude the identification model by manipulating the input code files so that the authorship attributes become ambiguous. More precisely, the goal is to decrease the confidence in the model's predictions to lead the model to misidentification or prediction rejection. This is done by adding code perturbation δ to the input d, such that the adversarial code generated $\bar{d} = d + \delta$ serves the purpose of minimizing the confidence with at least $\varepsilon > 0$, where ε the confidence reduction level. We note that δ is the code perturbation on the source-code input level.

The objective is to find a minimum δ disturbing more ε:

$$G_{\delta,\varepsilon}(\bar{d}) = \min_{\delta}\{|\delta| \quad s.t. \quad \{\text{conf}(y|d) - \text{conf}(y|\bar{d})\} \geq \varepsilon\}.$$

The $\text{conf}(y|d)$ indicates the probability of correctly assigning d to the *rightful programmer* y. Algorithm 1 shows the algorithm for threat model 1.

Adversarial Capabilities. For achieving the adversarial goal of non-targeted attacks under the *Closed box* scenario, the perturbation δ is generated randomly regardless of the code representation scheme adopted by the targeted system. The adversary only sends the adversarial example and receives the prediction output and score. The adversary iteratively enhances the adversarial perturbation until achieving the adversarial objective. We note that the adversary gains more advantage when accessing the training data since knowing the space of code expressions used in the dataset and their impact on the authorship attribution reduces the size of δ significantly. However, we only assume a *Closed box* scenario for this attack.

Algorithm 1: Non-Targeted Attack.

Require: Source code d, model f, iterations N, threshold ε
Ensure: Adversarial example $\bar{d}$ with reduced confidence
$\bar{d} \leftarrow d$ {Initialize adversarial example}
$\delta \leftarrow \emptyset$ {Initialize perturbation}
$\text{conf}_{orig} \leftarrow \text{conf}(f(d))$ {Get original confidence}
for $i = 1$ to N **do**
  $\delta_{new} \leftarrow$ GenerateRandomPerturbation() {Generate new block}
  $\bar{d}_{new} \leftarrow d + \delta_{new}$ {Add perturbation}
  $\text{conf}_{new} \leftarrow \text{conf}(f(\bar{d}_{new}))$ {Get new confidence}
  if $\text{conf}_{orig} - \text{conf}_{new} \geq \varepsilon$ **then**
    $\delta \leftarrow \delta_{new}$ {Accept perturbation}
    $\bar{d} \leftarrow \bar{d}_{new}$ {Update example}
    if $\text{conf}(y|\bar{d}) < \text{conf}(y^\star|\bar{d})$ AND $y^\star \neq y$ **then**
      break {Exit: Goal Achieved}
    end if
  end if
end for
return $\bar{d}$

B. Experiments: Baseline Evaluation

Our Dataset. The evaluation is conducted using the Google Code Jam (GCJ) competition [1]. GCJ is an international programming competition run by Google since 2008. At GCJ, programmers use several programming languages and development environments to solve programming problems over multiple rounds. The most common languages in GCJ are C++, Java, Python, and C, in order. For this work, we used a dataset of 200 C++ programmers with nine code samples that have 68.42 lines of code on average. Our choice of C++ is motivated by several key factors, such as, C++ is a widely-used general-purpose programming language that allows both high-level object-oriented and low-level systems programming, it has extensive syntax and language features that give programmers flexibility in coding style, making it ideal for studying authorship attribution, and it is statically typed and compiled, which helps ensure code correctness and functionality when applying adversarial perturbations. The dataset of C++ samples was collected from GCJ competition for all years from 2008 to 2016. The number of code samples, programmers, and years is consistent with previous work.

The Evaluation Metrics. We evaluate the success of non-targeted attacks based on the degradation in the model confidence and the misidentification rate (i.e., attack success rate). When the predicted label $\hat{y} = f(\bar{d})$, for the adversarial code sample $\bar{d}$ is not the same as the correct class label y of the original code sample d, this results in misidentification. The misidentification rate is simply calculated as: $\frac{1}{n}\sum_j^n I(f(\bar{d}_j) \neq f(d_j)$, where I is the count operator, $f(\bar{d}_j) = \hat{y}$ is the predicted label of AE, and $f(d_j) = y$ is the actual label of d_j.

Baseline Identification Results. Fig. 4 shows the identification accuracy achieved by the targeted systems using different sub-datasets with different numbers of programmers using 9-fold cross-validation evaluation. The results show that the RNN-based deep representations of DL-CAIS have enabled an identification accuracy ranging from 94.65% for identifying 200 programmers to 97.8% for 50 programmers.

Code stylometry features adopted in CSFS have achieved an identification accuracy between 92.4% and 94.9% for different datasets. Using word embeddings to represent code files

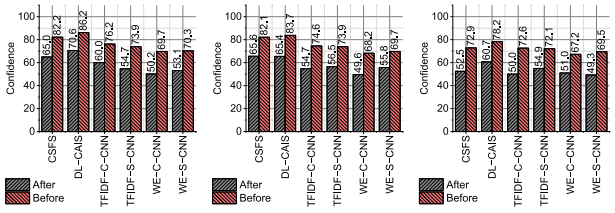


Fig. 5. The confidence of models for the predicted class before and after the attacks. The size of the dataset impacts the confidence of the models.

for CNN-based approaches, WE-C-CNN and WE-S-CNN have achieved an identification accuracy of 86.23% and 84.18% for identifying 200 programmers, respectively. TF-IDF representations with CNN increased the identification accuracy, reaching 88.1% and 86.12% for identifying 200 programmers using TFIDF-C-CNN and TFIDF-S-CNN, respectively.

Fig. 5 shows a high identification accuracy is achieved by most systems accompanied by high identification confidence, where the confidence exceeds 67% in all settings for all identification techniques. For systems that adopt RFC, such as DL-CAIS and CSFS, the model confidence is higher than in systems that utilize a softmax layer for classification, such as CNN-based systems (e.g., WE-C-CNN). This is because the confidence of RFC models is defined by the number of trees in the random forest that vote for a given class label, while the softmax layer captures the probability distribution of assigning the input data to all classes. In our experiments, the identification confidence decreases as the number of programmers increases, as shown in Fig. 5.

C. Experiments: Impact of Attacks

Fig. 4 shows the misidentification rate before and after launching the non-targeted attack. The attack produces an average misidentification rate above 98.8% for all targeted approaches using a dataset of 50 programmers. The rate increases as the dataset size increases, as shown in Fig. 4.

This misidentification rate is caused by the low confidence levels of identifying programmers when adding code perturbations. Fig. 5 shows the confidence score of the identification models before and after the attack using different datasets. The figure shows the clear impact of code perturbation on the confidence score of models: when using a dataset of 200 programmers, the confidence score reaches a high point of 60.69% and 52.46% for DL-CAIS and CSFS, respectively, and a low point of 49.27% for WE-S-CNN.

How the Number of Programmers Impacts Misclassification. Fig. 4 shows the average misidentification rate of different methods using different datasets of programmers ranging from 50 to 200. Even when the number of programmers is as low as 50, the attribution systems failed to generate the right output, since the average achieved misidentification rate was greater than 98.8%. The results show that increasing the dataset size increases the misidentification rate. A similar trend is observed in reporting the confidence score of models, as the larger the model's output size the more it is affected by the perturbation in terms of output confidence. For example, the confidence scores of DL-CAIS and CSFS after the attack are 70.61% and 64.98% when using the 50-programmer dataset, and 60.69% and 52.46% when using the 200-programmer dataset. These scores and the drop in confidence (i.e., before and after the attack) are shown in Fig. 5.

How the Number of Added Code Lines Impacts the Success Rate. To explore the impact of perturbation size on the attack success rate, we restricted the implementation of the attack to adding a specific number of code lines. The attack starts with adding two lines of code and iteratively probing the target system to a predefined number of access permissions (e.g., in this experiment 20 times). If the attack succeeds, the termination mechanism takes place. Otherwise, the attack generator increases the number of lines added by a step of two. When reaching the predefined limit without success, the attack generator terminates and marks the process as failed.

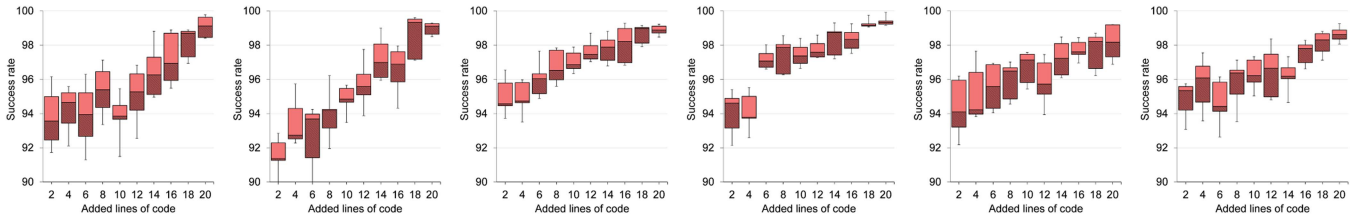
Fig. 6 shows the average misidentification rate achieved when launching the non-targeted attacks with a perturbation of 2 to 20 lines of code using a dataset of 200 programmers. Typically, increasing the number of lines affects the overall performance of the attack. For example, the difference in the success rate of the attack is $(99.16 - 94.3 = 4.86\%)$ when targeting CSFS by adding 20 and 2 lines of code, respectively. However, this difference becomes less obvious when using a larger perturbation than ten lines of code (e.g., $99.41 - 97.62 = 1.79\%$ when targeting TFIDF-S-CNN by adding 20 and 10 lines of code, respectively. These results confirm that using random code perturbation cripples the identification models' capabilities for identifying programmers. Furthermore, even when using the smallest perturbation size (i.e., two lines of code), the achieved misidentification rate exceeds 92% for all approaches when using a dataset of 200 programmers.

V. Targeted Attacks

A. Threat Model 2: Targeted Attack

SHIELD aims to fool the model to predict a specific *target* class (i.e., a programmer to be imitated/impersonated). Targeted attacks can serve multiple adversarial objectives. For example, a common adversarial objective is the *imitation attack*, also referred to as a *impersonation attack*. This kind of attack has serious implications, for example, a malicious programmer can manipulate the code to blame a benign programmer. Another adversarial objective is the *disguise attack* or *evasion attack*, where the SHIELD impersonates the programmer closest to their style as a disguise. Although SHIELD does not target a specific programmer for impersonation, the attack is considered a targeted attack.

Definition. The targeted attack aims to maximize the probability that an adversarial code sample is classified as the targeted class. If successful, this attack enables programmers to imitate the style of other programmers. Using the same definition of $\text{conf}(y|d)$, the aim is to minimize confidence in predicting the right programmer y and maximize the probability of the target $\hat{y}$, that is, $\text{conf}(\hat{y}|\bar{d}) > \text{conf}(y|d)$.



(a) CSFS (b) DL-CAIS (c) TFIDF-C-CNN (d) TFIDF-S-CNN (e) WE-C-CNN (f) WE-S-CNN

Fig. 6. The results of non-targeted attacks using a dataset of 200 programmers. The figure shows the average misidentification rate of different targeted approaches using added perturbations of code lines ranging from 2 to 20.

Adversarial Objective. The adversarial goal of the targeted attack is to maximize the confidence of the model prediction toward a target class $\hat{y}$.Similar to the previous threat model, this is carried out by code perturbation δ on d such that the adversarial code generated $\bar{d} = d + \delta$ serves the purpose of maximizing the confidence of a target programmer $\hat{y}$. Thus, the adversarial objective is formally defined as:

$$G_\delta(\bar{d}) = \min_{\delta}\{|\delta| \; s.t. \; \text{conf}(\hat{y}|\bar{d}) > \{\max_{i}(\text{conf}(y_i|\bar{d})) \; \forall y_i \neq \hat{y}\}\}$$

The attack achieves the objective when the model predicts that the target programmer is the author of $\bar{d}$.

Adversarial Capabilities. For the targeted attacks, we investigate different adversarial capabilities described as follows.

• *Targeted Attacks (T1):* A *Closed box* attack where SHIELD sends the AE and obtains the model output and scores. By iterative probing, SHIELD optimizes the randomly generated code perturbations to achieve her goal without any knowledge of the target's coding style. Algorithm 2 shows the algorithm for threat model T1. The adversary generates the perturbation δ based on a $mask$, a binary vector that represents the features of the used code in perturbation. The mask is updated based on the prediction of the model. In T1, the $mask$ is initialized as an empty feature vector representing all reserved words in the target programming language (e.g., C++ in our experiment). The GeneratePerturbation($mask$) function randomly samples features from the $mask$ to generate the perturbation. If the adversarial example fails, then the $mask$ is updated based on the prediction of the model. The UpdateMask(δ_{new}, y_{pred}) function updates the $mask$ such that $mask_{new} = mask_{old} \oplus [(1 - y_{pred}) \; \overline{\text{bin}}(\text{TF}(\delta_{new}))]$, where $\oplus$ is the XOR operator, $\overline{\text{bin}}(x) = 1$ if $x > 0.5$ else 0, and TF is the uni-gram vectorizer used to initialize $mask$.

• *Targeted Attacks (T2):* A *Closed box* attack, although SHIELD has access to two code samples of the target programmer. The perturbations are then generated and optimized based on the most representative features of the target programmer. Algorithm 3 summarizes the process for threat model T2. The algorithm applies similar steps to T1. However, in T2 and T3, the $mask$ is initialized with the representative features of the target (programmer in T2, programmers set in T3) using straightforward uni-grams TF representation, i.e., $mask = \text{bin}(\text{TF}(X))$, where bin is the binarization operator ($\text{bin}(x) =$ 1 if $x > 0$ else 0), TF is uni-gram token frequency, X are all samples from the target. Since the adversary obtain a target vector ($F_{\hat{y}}$) profile in T2 and T3, The UpdateMask($\delta_{new}, F_{\hat{y}}, y_{pred}$) function updates the $mask$ such that $mask_{new} = [mask_{old} \cup F_{\hat{y}}] \oplus [(1 - y_{pred}) \; \overline{\text{bin}}(\text{TF}(\delta_{new}))]$, where $\cup$ is the union operation, and $\oplus$, $\overline{\text{bin}}$, and TF are the same operation as in T1.

• *Targeted Attacks (T3):* This attack assumes that SHIELD has access to code samples of a group of programmers (e.g., ten programmers) and aims to disguise as the closest programmer to her coding style, thus, the attack is called *disguise* or *evasion attack*. Note that the adversarial objective here is not exactly the same as the general objective of the targeted attack, since SHIELD impersonates the closest programmer to her style to evade identification. Implementing this attack is easier compared to T1 and T2 due to the flexibility of the imitation options in assigning an adversarial class, especially when the target set is large.

For T3 (disguise/evasion attack), the attack goal is to maximize the second highest confidence in the model predictions so that $\text{conf}(\hat{y}|\bar{x}) > \text{conf}(y|\bar{x})$, where $\bar{y}$ is the programmer in the coding style closest to the adversary. This does not necessarily mean that the adversarial objective is to predict (specifically) the $\hat{y}$ target. For example, in softmax-based classifiers, the result of any $\text{argmax}(\text{conf}(y_i|\bar{x}) \forall i \neq k) \nRightarrow y$, where $y_k = y$, is acceptable. To select the target, the programmer closest to the adversary in terms of coding style, the adversary needs code samples from the target set.

Let the target set of m programmers be $\{y_1, y_2, \ldots, y_m\}$, each with n code samples. In our experiments, we assume that the adversary has access to two code samples for each programmer in the target set. For a programmer y_i, the programmer closest to $\hat{y}$ is the one with the code samples with the shortest euclidean distance to the samples of y_i. To estimate the euclidean distance, the adversary represents the accessible samples from the target set using a straightforward n-gram model that uses only the occurrences of unigrams. For each programmer, the adversary obtains the average representation of all samples such that a programmer y_i has the average vector representation $\text{avg}(d) = \frac{1}{n}\text{sum}(d_j) \quad \forall \, j \in \{1, 2 \ldots, n\})$, where n is the number of samples for a programmer y, $d_j \in \mathbb{R}^m$ is the vector representation of the j-th sample, m is the dimension of extracted unigrams, and the sum is the *point-wise vector summation operation*.

The closest programmer $\hat{y}$ to a programmer y with samples of $\text{avg}(d)$ is then selected using the shortest distance. Then, $\hat{y} = y_j$, for y_j that satisfies $\text{min}(\text{dis}(\text{avg}(d^{(y)}) - \text{avg}(d^{(y_j)})))\forall \, j \in$

Algorithm 2: Targeted Attack (T1).

Require: Source code d, target label $\hat{y}$, model f, iterations N
Ensure: Adversarial example $\bar{d}$ classified as $\hat{y}$

$\bar{d} \leftarrow d$ {Initialize adversarial example}
$\delta \leftarrow \emptyset$ {Initialize perturbation}
$mask \leftarrow \emptyset$ {Initialize feature mask}
for $i = 1$ to N **do**
  $\delta_{new} \leftarrow$ GeneratePerturbation($mask$) {Generate with mask}
  $\bar{d}_{new} \leftarrow d + \delta_{new}$ {Add perturbation}
  $y_{pred} \leftarrow f(\bar{d}_{new})$ {Get prediction}
  if $y_{pred} = \hat{y}$ **then**
    $\delta \leftarrow \delta_{new}$ {Accept perturbation}
    $\bar{d} \leftarrow \bar{d}_{new}$ {Update example}
    break {Exit: Goal Achieved}
  else
    $mask \leftarrow$ UpdateMask(δ_{new}, y_{pred}) {Update feature mask}
    $\delta \leftarrow$ OptimizePerturbation(δ_{new}, $mask$) {Optimize}
    $\bar{d} \leftarrow d + \delta$ {Update with optimized}
  end if
end for
return $\bar{d}$

Algorithm 3: Targeted Attack (T2).

Require: Source code d, target label $\hat{y}$, target samples $S_{\hat{y}}$, model f, iterations N
Ensure: Adversarial example $\bar{d}$ classified as $\hat{y}$

$\bar{d} \leftarrow d$ {Initialize adversarial example}
$\delta \leftarrow \emptyset$ {Initialize perturbation}
$F_{\hat{y}}, mask \leftarrow$ ExtractTargetFeatures($S_{\hat{y}}$) {Feature mask from target}
for $i = 1$ to N **do**
  $\delta_{new} \leftarrow$ GeneratePerturbation($mask$) {Generate with mask}
  $\bar{d}_{new} \leftarrow d + \delta_{new}$ {Add perturbation}
  $y_{pred} \leftarrow f(\bar{d}_{new})$ {Get prediction}
  if $y_{pred} = \hat{y}$ **then**
    $\delta \leftarrow \delta_{new}$ {Accept perturbation}
    $\bar{d} \leftarrow \bar{d}_{new}$ {Update example}
    break {Exit: Goal Achieved}
  else
    $mask \leftarrow$ UpdateMask(δ_{new}, $F_{\hat{y}}$, y_{pred}) {Update feature mask}
    $\delta \leftarrow$ OptimizePerturbation(δ_{new}, $mask$) {Optimize}
    $\bar{d} \leftarrow d + \delta$ {Update with optimized}
  end if
end for
return $\bar{d}$

$\{1, \ldots, m\}$, where m is the number of programmers in the target set, and dis is the euclidean distance between the two average vector representations.

After defining the nearest programmer $\hat{y}$, the adversary retrieves the most influential features using the order of the features to be the source of the perturbation. Similar to the imitation attack, this attack is conducted by adding code perturbation δ to the input code d so that the generated adversarial code $\bar{d} = d + \delta$ maximizes the confidence of predicting the programmer $\hat{y}$. Algorithm 4 summarizes the process for threat model T3.

B. Dataset and Evaluation Metrics

Our Dataset. We use a subset of the same dataset from the previous experiment. We randomly selected a subset of 100 programmers to simulate different experimental settings. Moreover, we marked each programmer in our subset dataset of 100 programmers to be subject to the imitation attack while using all other programmers to imitate the coding style of that target programmer. Since our dataset includes nine code samples per programmer, we generated $(9 \times 99 = 891)$ adversarial code samples $(9 \times 99 = 891)$ for each subject programmer, that is, we generated 89,100 adversarial code samples for each experimental setting considering the 100 programs we included.

One consideration we took into account when choosing the 100 random programmers is that each selected programmer should have more than 11 samples in the GCJ dataset. This constraint is to allow the implementation of the targeted attacks T2 using the same dataset for consistency. We emphasize that T2 assumes knowledge of two samples.

Attack Evaluation Metrics. We evaluate the targeted attack by its success rate as the proportion of correctly classified adversarial samples to the targeted programmer relative to the attempts. For a class $\hat{y}$, the attack succeeds when the model predicts $\hat{y}$. For example, in softmax-based classifiers, the model outputs $y_k = \hat{y}$ such that $\text{argmax} P(\mathcal{Y}|\bar{d}) = \hat{y}$. The success rate is: $\frac{1}{n \times m} \sum_j^m \sum_i^n \text{count}(d_i^{(y_j)} \Rightarrow \hat{y}_j)\ \forall i \in 1, 2, \ldots, n$ samples and $\forall j \in 1, 2, \ldots, m$ programmers, where count refers to the successful attempts of the attack.

For T3, i.e., the evasion attack, the attack success rate is defined by the rate of correctly misidentifying a specific programmer to all code samples presented by that programmer, i.e., similar to the misidentification rate. However, we adopted a targeted attack where an adversarial code sample $\bar{d}$ is attributed to the closest programmer $\hat{y}$ with respect to the original programmer y. Imitating the closest programmer is the reason for assuming this scenario as a targeted attack.

C. Experimental Results: The Impact of Attacks

The impact of the different targeted attacks on the performance of the authorship attribution systems is shown in Fig. 7, and in the following, we elaborate on those results.

Targeted Attack (T1). Fig. 7 shows that the success rate is considerably high with more than 66.55% success rate for all systems under the T1 attack. The average success rates are 66.55%, 70.26%, 73.51%, 71.16%, 78.45%, and 72.05% for CSFS, DL-CAIS, TFIDF-C-CNN, TFIDF-S-CNN, WE-C-CNN, and WE-S-CNN, respectively.

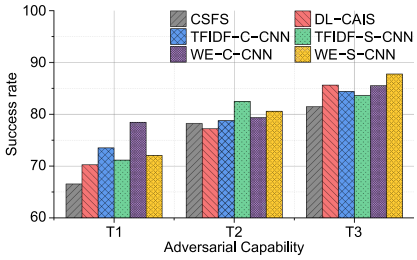


Fig. 7. The results of the targeted attack: the success rate of attacks on the targeted approaches categorized by different threat models. The results show the increase in the success rate is associated with adversarial knowledge.

Algorithm 4: Targeted Attack (T3).

Require: Source code d, suspect set $\mathcal{Y}$, samples $S_{\mathcal{Y}}$, model f, iterations N
Ensure: Adversarial example $\bar{d}$ classified as closest style $\hat{y}$
$\bar{d} \leftarrow d$ {Initialize adversarial example}
$\delta \leftarrow \emptyset$ {Initialize perturbation}
$\hat{y} \leftarrow$ FindClosestStyle(d, $S_{\mathcal{Y}}$) {Find closest programmer}
$F_{\hat{y}}, mask \leftarrow$ ExtractTargetFeatures($S_{\hat{y}}$) {Extract target features}
for $i = 1$ to N **do**
  $\delta_{new} \leftarrow$ GeneratePerturbation($mask$) {Generate with mask}
  $\bar{d}_{new} \leftarrow d + \delta_{new}$ {Add perturbation}
  $y_{pred} \leftarrow f(\bar{d}_{new})$ {Get prediction}
  if $y_{pred} = \hat{y}$ **then**
    $\delta \leftarrow \delta_{new}$ {Accept perturbation}
    $\bar{d} \leftarrow \bar{d}_{new}$ {Update example}
    break {Exit: Goal Achieved}
  else
    $mask \leftarrow$ UpdateMask(δ_{new}, $F_{\hat{y}}$, y_{pred}) {Update feature mask}
    $\delta \leftarrow$ OptimizePerturbation(δ_{new}, $mask$) {Optimize}
    $\bar{d} \leftarrow d + \delta$ {Update with optimized}
  end if
end for
return $\bar{d}$

Fig. 8 shows the results of ten random programmers as a matrix. Each cell represents the success rate of targeted attacks where each programmer attempts to mimic another programmer in implementing nine programming challenges. The figure shows that most programmers can be imitated and their coding style can be manipulated to match other programmers'. This is true for all targeted systems, with a higher success rate on CNN-based systems. The figure shows that some programmers are more difficult to imitate than others, for example, programmer **I** in Fig. 8(d) using TFIDF-S-CNN and programmer **B** in Fig. 8(e) using WE-C-CNN.

Targeted Attack (T2). Given more adversarial knowledge, i.e., the adversary has access to two samples from the target programmer, the results show an increased success rate for attacks. This scenario still assumes a *Closed box* attack, since the adversary has no knowledge or access to the system's internal components. The success rate of this attack exceeds 77% for all targeted systems. Fig. 7 shows that the average success rates are 78.23%, 77.22%, 78.79%, 82.49, 79.35%, and 80.58% for CSFS, DL-CAIS, TFIDF-C-CNN, TFIDF-S-CNN, WE-C-CNN, and WE-S-CNN, respectively.

These results show that the adversary's knowledge contributed significantly to the targeted attack's success. Accessing code samples from the target enabled the optimization of adversarial examples by considering the perturbations that benefited from the actual stylometry features of the target. This level of knowledge contributed to an increase of 11.68%, 6.96%, 7.63%, 11.33%, 0.9%, and 8.53% in the attack success rate in CSFS, DL-CAIS, TFIDF-C-CNN, TFIDF-S-CNN, WE-C-CNN, and WE-S-CNN, respectively.

Targeted Attack (T3). This model extends the adversary knowledge to include samples from a subset of programmers to imitate the closest among them. This capability allows for a higher success rate compared to T1 and T2. We note that the goal of the adversary in T3 is different from that of T1 or T2 (see Section V-A). Fig. 7 shows that the average success rates are 81.48%, 85.63%, 84.40%, 83.61, 85.52%, and 87.77% for CSFS, DL-CAIS, TFIDF-C-CNN, TFIDF-S-CNN, WE-C-CNN, and WE-S-CNN, respectively. The results show that the adversary has a higher success in imitating one programmer in a group of programmers with access to code samples from those programmers.

Impact of Added Code Lines. We implemented the attack with a specific number of code lines. Similar to the settings adopted for non-targeted attacks, the attack generator increases the number of added lines with a step of two and iteratively accessing the target system to a predefined number of access permissions each step (e.g., in this experiment 50 times). Fig. 9 shows the success rate of different attack scenarios using different lines of code as perturbations.

The results show that code perturbation can lead to a high success rate of targeted attacks when adding as few as two lines of code. For example, the average success rates achieved under T1, T2, and T3 are 50.37%, 55.37%, and 61.12%, respectively, for all targeted systems when using only two lines of code as a perturbation. When increasing the perturbation size to 20 lines of code, the success rate reaches 83.86%, 83.58%, and 92.03% for the threat models T1, T2, and T3, respectively. These results demonstrate that most authorship attribution systems are susceptible to adversarial perturbations and can be highly affected even with small changes in the input file.

VI. Discussion

We now explore the effects of adversarial perturbations on authorship attributes against different attacks. Second, we show the magnitude of perturbations when adding code blocks to the original code. We compare the performance of SHIELD with related work and, finally, list the limitations.

A. Adversarial Authorship Attributions

We show that code authorship attribution systems can be vulnerable to adversarial attacks. For all attacks, the authorship attributes are greatly affected by the smallest size of perturbations. This effect can be shown in the PCA visualization of

SR	A	B	C	D	E	F	G	H	I	J
A		0.33	0.56	0.89	0.78	1.00	1.00	0.89	1.00	1.00
B	0.89		0.44	0.56	0.78	1.00	0.44	0.89	0.89	1.00
C	0.78	1.00		1.00	0.89	0.89	0.78	0.78	0.00	0.22
D	0.89	0.89	0.33		0.89	1.00	0.89	0.67	0.22	0.89
E	0.78	1.00	1.00	0.67		1.00	1.00	1.00	0.33	1.00
F	1.00	0.44	0.33	1.00	0.33		0.89	1.00	0.22	1.00
G	0.89	1.00	0.78	0.56	0.00	1.00		0.78	0.89	1.00
H	1.00	0.67	0.67	0.44	1.00	0.56	0.78		1.00	1.00
I	0.67	0.89	0.44	0.11	0.22	0.89	1.00	0.44		0.78
J	1.00	0.22	1.00	1.00	0.56	0.56	0.89	0.78	0.33	

(a) CSFS [13]

SR	A	B	C	D	E	F	G	H	I	J
A		0.33	0.56	0.89	0.78	1.00	1.00	0.89	1.00	1.00
B	0.89		0.44	0.56	0.78	1.00	0.44	0.89	0.89	1.00
C	0.78	1.00		1.00	0.89	0.89	0.78	0.78	0.00	0.22
D	0.89	0.89	0.33		0.89	1.00	0.89	0.67	0.22	0.89
E	0.78	1.00	1.00	0.67		1.00	1.00	1.00	0.33	1.00
F	1.00	0.44	0.33	1.00	0.33		0.89	1.00	0.22	1.00
G	0.89	1.00	0.78	0.56	0.00	1.00		0.78	0.89	1.00
H	1.00	0.67	0.67	0.44	1.00	0.56	0.78		1.00	1.00
I	0.67	0.89	0.44	0.11	0.22	0.89	1.00	0.44		0.78
J	1.00	0.22	1.00	1.00	0.56	0.56	0.89	0.78	0.33	

(b) DL-CAIS [4]

SR	A	B	C	D	E	F	G	H	I	J
A		0.33	0.56	0.89	0.78	1.00	1.00	0.89	1.00	1.00
B	0.89		0.44	0.56	0.78	1.00	0.44	0.89	0.89	1.00
C	0.78	1.00		1.00	0.89	0.89	0.78	0.78	0.00	0.22
D	0.89	0.89	0.33		0.89	1.00	0.89	0.67	0.22	0.89
E	0.78	1.00	1.00	0.67		1.00	1.00	1.00	0.33	1.00
F	1.00	0.44	0.33	1.00	0.33		0.89	1.00	0.22	1.00
G	0.89	1.00	0.78	0.56	0.00	1.00		0.78	0.89	1.00
H	1.00	0.67	0.67	0.44	1.00	0.56	0.78		1.00	1.00
I	0.67	0.89	0.44	0.11	0.22	0.89	1.00	0.44		0.78
J	1.00	0.22	1.00	1.00	0.56	0.56	0.89	0.78	0.33	

(c) TFIDF-C-CNN [7]

SR	A	B	C	D	E	F	G	H	I	J
A		1.00	0.56	0.89	1.00	1.00	1.00	0.78	0.44	1.00
B	1.00		0.33	0.89	0.78	1.00	0.89	1.00	0.33	1.00
C	0.89	1.00		1.00	1.00	0.89	1.00	0.78	0.67	0.89
D	1.00	0.89	0.67		1.00	1.00	0.89	0.89	0.22	1.00
E	0.78	1.00	0.44	0.67		0.89	1.00	0.89	0.78	1.00
F	1.00	0.89	0.33	1.00	0.56		0.89	1.00	0.11	1.00
G	0.89	1.00	0.67	0.67	0.00	1.00		0.89	0.00	1.00
H	1.00	1.00	0.67	0.44	1.00	0.56	1.00		0.33	1.00
I	0.67	0.89	0.89	0.33	0.78	0.89	1.00	0.44		0.89
J	1.00	0.78	1.00	1.00	0.22	0.89	0.89	0.78	0.56	

(d) TFIDF-S-CNN [7]

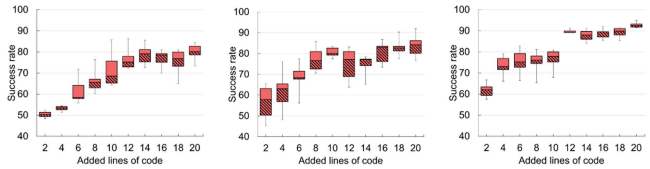
SR	A	B	C	D	E	F	G	H	I	J
A		0.33	0.78	0.89	1.00	1.00	1.00	0.89	0.44	1.00
B	1.00		0.44	1.00	0.89	1.00	0.89	1.00	0.56	1.00
C	0.89	0.22		1.00	0.78	1.00	1.00	0.78	0.44	0.89
D	0.78	0.56	0.67		0.89	1.00	1.00	1.00	0.78	1.00
E	1.00	0.33	0.56	0.78		0.89	1.00	1.00	0.67	1.00
F	1.00	0.22	0.78	0.89	0.56		1.00	1.00	0.67	1.00
G	0.78	0.78	0.89	0.56	0.00	1.00		0.89	0.33	1.00
H	1.00	0.67	0.67	1.00	1.00	0.78	1.00		0.44	1.00
I	0.67	0.11	0.89	0.33	0.78	0.89	1.00	0.44		0.89
J	1.00	0.00	1.00	1.00	0.22	1.00	1.00	0.89	0.89	

(e) WE-C-CNN [7]

SR	A	B	C	D	E	F	G	H	I	J
A		0.33	0.89	1.00	1.00	1.00	0.89	0.89	0.11	1.00
B	1.00		0.89	1.00	1.00	1.00	0.89	1.00	0.33	1.00
C	1.00	0.89		1.00	0.78	1.00	1.00	0.89	0.67	0.89
D	0.89	0.78	0.89		0.89	1.00	1.00	1.00	0.89	1.00
E	0.78	0.11	0.89	0.89		0.89	1.00	1.00	0.67	1.00
F	1.00	0.44	1.00	0.89	1.00		0.89	1.00	0.33	1.00
G	1.00	0.44	0.67	1.00	0.78	1.00		1.00	0.78	1.00
H	1.00	0.78	0.78	1.00	1.00	0.78	1.00		0.67	1.00
I	0.89	0.67	1.00	0.67	0.89	0.89	0.78	0.78		0.89
J	1.00	0.89	0.89	0.89	0.78	1.00	0.78	0.89	0.11	

(f) WE-S-CNN [7]

Fig. 8. Matrix representation of the targeted attack T1 for six authorship attribution systems. Each cell indicates the success rate of the attack for ten programmer pairs using nine code files per programmer.



(a) T1 (b) T2 (c) T3

Fig. 9. The results of different targeted attacks using a dataset of 100 programmers. The figure shows the average success rate of the targeted attacks using perturbations of code lines ranging from 2 to 20.

the authorship attributes of code samples generated by different adversarial settings, as shown visually in Fig. 10 for code samples of ten programmers with nine code samples each in various adversarial settings. For this visualization, we used the DL-CAIS system to demonstrate the effect of different attacks. Fig. 10(a) shows the original code representations of the ten programmers with nine code samples. It is clear that these authorship attributes are the reason for the accurate identification process. However, when introducing adversarial code examples as in Fig. 10(b) during the non-targeted attack, the authorship features become extremely scattered in the feature space, so it is difficult to establish decision boundaries, and hence the high misidentification rate. Fig. 10(c) shows a visualization of the targeted attack T1, where nine programmers attempt to imitate one label (i.e., *Label 10* in this figure), which leads to partial success since some programmers are still resilient to imitate other programmers. The figure also shows that most of the code samples are within the same proximity to the code samples of the targeted programmer. However, some programmers have very distinct coding, for example, *Label 1* has code samples that are not affected by the targeted perturbations. Other programmers are partially affected, such as (*Label 3* and *Label 7*). In Fig. 10(d), we show the effect of the targeted attack T3 (i.e., evasion attack), in which the adversary aims to imitate the programming style closest to the adversary's. In this figure, *Label 1* imitates the coding style of *Label 3* as their coding style is the closest among others, as can also be seen in Fig. 10(a). We explained this attack as if *Label 1* is using *Label 3* as a disguise to evade identification.

B. Comparison With Other Methods

We conducted a comparison with the work of Quiring et al. [27] using three different adversarial settings. The experiments included six targeted systems and a dataset of 20 programmers. We follow the implementation details of the original work of [27] and limit access to the identification model to 20 times. If the attack generator fails to achieve the adversarial goal by the 20th iteration, it is considered a failed attempt. Fig. 11 shows that the adversarial perturbation is as effective as code transformation methods, however considerably simpler. For the non-targeted attacks, both this work and the work of [27] achieved more than a 97% success rate, indicating the vulnerability of current code authorship attribution methods to adversarial attacks.

Under T1, SHIELD has a higher attack success rate due to the flexibility of the added perturbation that shifts attributes to the target output. However, the method by [27] performs better under T2, where the adversary has access to two samples from the target. This is due to the supported template transformations that help to achieve the adversarial objective.

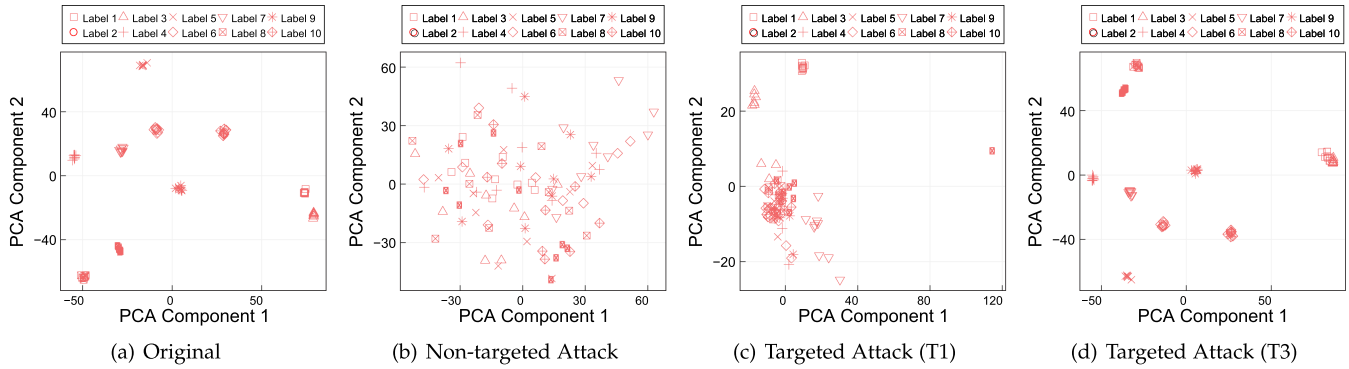


Fig. 10. The PCA visualization of authorship attributions of 10 programmers with nine code samples. The figure shows the original attributions along with the effects of different adversarial attacks on the generated attributions using DL-CAIS.

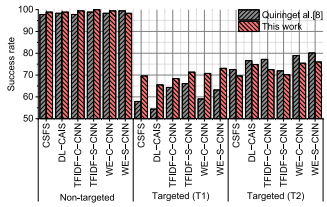


Fig. 11. Comparison with Quiring et al. [27] using three adversarial settings. The results are obtained using a dataset of 20 programmers and access to the model 20 times.

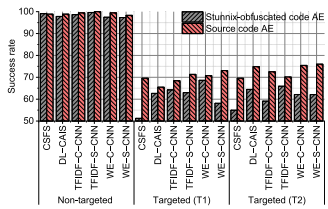


Fig. 12. The results of different attacks using the original and the obfuscated adversarial code examples. The results obtained using a dataset of 20 programmers.

C. Obfuscated Adversarial Code Examples

Code obfuscation is one way to thwart static analysis and elimination of the perturbation in the adversarial code examples by figuring out unreachable code blocks. Assuming a stronger authorship attribution pipeline that operates on an obfuscated input domain, we experimented with obfuscated examples. For this experiment, we used a dataset of 20 C++ programmers with nine files each (the same dataset in Section VI-B). We obfuscated the samples using Stunnix [2], a popular code-to-code C/C++ obfuscator that gives the code a cryptic appearance while preserving the functionality. Previous code authorship attribution studies have used Stunnix obfuscator to demonstrate the validity of working in an obfuscated domain and produced high identification accuracy [4], [13].

Assumptions. In the following experiment, we assume that the attribution system can operate in the obfuscated domain, and some models are trained and tested using obfuscated samples. The adversary aims to hide the code perturbations using the obfuscation tool, knowing that the authorship attribution system can recognize the tool and assign the sample to the models trained on the obfuscated domain.

Identification of Obfuscated Samples. Establishing baseline models that operate in an obfuscated domain, we evaluated the six targeted systems on a dataset of Stunnix-obfuscated code of 20 programmers. The 9-fold cross-validation accuracy was 97.64%, 95.43%, 94.79%, 98.19, 98.82%, and 96.18% for CSFS, DL-CAIS, TFIDF-C-CNN, TFIDF-S-CNN, WE-C-CNN, and WE-S-CNN, respectively.

Obfuscated Adversarial Code Examples. SHIELD follows the same method of generating adversarial code examples, as described in the different threat models (T1-T3), and obfuscates the adversarial code to produce the adversarial obfuscated example. The obfuscated code samples are then submitted to the authorship attribution system to identify the authors. We limit the adversary's access to the identification model to 20 times to achieve the adversarial goal.

Fig. 12 shows SHIELD 's success rate for different attacks in the obfuscated domain. Compared to the attacks on the source code domain, the obfuscated code samples achieved a similar success rate in the non-targeted attacks against all targeted systems. For targeted attacks using T1 and T2 threat models, the success rate degrades slightly due to the difficulty of optimizing the adversarial code example at the source code level using the feedback of the model on the obfuscated sample. Moreover, perturbations at the source level (before obfuscation) do not guarantee a shift in the feature space of the obfuscated sample (after obfuscation) toward a certain target. However, attacks exceed the success rate of 51.24% and 54.96% for targeted attacks T1 and T2, respectively.

D. The Impact of the Perturbation Size

We use ℓ_p to measure the magnitude of perturbation by p–$norm$ distance as: $\|\delta\|_p = (\sum_{i=1}^{n} \|\bar{x}_i - x_i\|^p)^{\frac{1}{p}}$. For p–$norm$, studying the ℓ_0, ℓ_2 and ℓ_∞ is very common [15]. The ℓ_0 is the count of changes in the adversarial example compared to the original sample, the ℓ_2 is the euclidean distance between the adversarial sample and the original sample, and ℓ_∞ is the maximum change with respect to all terms in the adversarial code examples. In this work, we explore the size of perturbation with respect to the number of code lines and show the magnitude

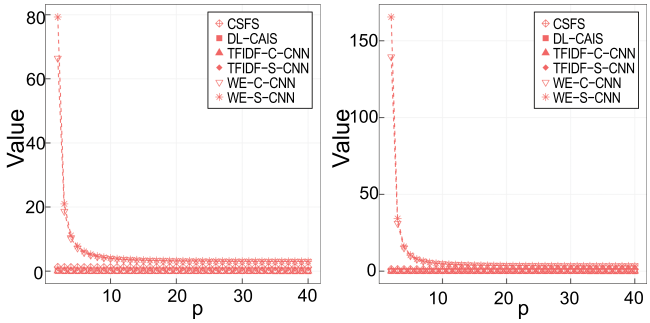


Fig. 13. The p–$norms$ of perturbations of different sizes with respect to the initial code representations. Small perturbations are generated by one statement, while large perturbations are generated by ten.

of such perturbation by p–$norm$ when considering non-targeted attacks.

Fig. 13 shows the values of p–$norm$, ranging from $p = 2$ to $p = 40$, for small code perturbations generated from one code statement and for large perturbations from ten code statements. The results shown in the figure are drawn from the average perturbation size using random perturbation generation (i.e., as used in the non-targeted attack) on a dataset of 100 programmers with nine files each. Generally, the effect of perturbations on the input representations varies based on the underlying method used to represent the code. The effects are highest when using word embeddings as initial representations, e.g., WE-C-CNN and WE-S-CNN, due to the high-dimensional and compact representations of code samples using word embeddings, while the lowest effects are exhibited by approaches using TF-IDF as initial representations, e.g., DL-CAIS and TFIDF-S-CNN.

Fig. 13(a) shows the ℓ_p of perturbations with one statement that includes only one line of code. The ℓ_2 values are: 0.07, 1.27, 0.08, 0.08, 66.43, 79.30 for DL-CAIS, CSFS, WE-C-CNN, WE-S-CNN, TFIDF-C-CNN, and TFIDF-S-CNN, respectively. Increasing the value of p slightly decreases the value of ℓ_p to reach $\ell_{40} \approx 3$ for WE-C-CNN, and WE-S-CNN. A similar observation is made using large perturbations regarding the effect on code representations. The ℓ_p values are shown to be slightly higher, which is due to the larger number of code lines added. Although these changes are not significant, they impact the outcome of models, as seen in Fig. 4.

VII. Related Work

Code Authorship Attribution. Authorship attribution is a relatively rich topic, with a large number of proposed systems. The earlier work in this space attempted to define a set of stylistic traits and features that characterize authors of program [18], [21], [26]. Such traits may include byte- or gram-level attributes [5], [17], control and data flow features [9], [24], [28], and AST features [13], [26].

Krsul and Spafford [18] proposed 60 authorship features for C programmers, achieving 73% accuracy in identifying 29 programmers. MacDonell et al. [21] introduced 26 stylistic features for C++, reaching 88% accuracy for seven programmers.

TABLE I
Comparison With the Related Adversarial Authorship Attribution

Related Work	Transformation Code	Transformation Feature	Obfuscation	○	◑	●	U	T
[30]	✓	·	·	·	·	✓	·	✓
[23]	✓	✓	·	✓	·	·	✓	✓
[22]	✓	·	·	·	✓	·	·	✓
[27]	✓	✓	·	✓	·	·	✓	✓
[20]	✓	·	·	·	·	✓	✓	✓
SHIELD	·	·	✓	·	·	✓	✓	✓+

SHIELD does not require transformations. ✓+ indicates multiple scenarios

Attack assumptions (○: white-box, ◑: gray-box, and ●: black-box attack) and attack objectives (**U**ntargeted and **T**argeted attack)

Frantzeskou et al. [17] leveraged byte-level n-grams, attaining 100% accuracy for eight programmers. Burrows et al. [12] combined n-grams with handpicked features, scaling attribution to 100 programmers with 80.37% accuracy. The first large-scale study by Caliskan-Islam et al. [13] used code stylometry to identify 1,600 C++ programmers with 92.83% accuracy. Abuhamad et al. [4] proposed DL-CAIS, an RNN-based model, identifying 8,903 programmers with 92.3% accuracy. Later, Abuhamad et al. [7] introduced a CNN-based approach, achieving 96.2% accuracy for 1,600 programmers. We evaluate the systems in [4], [7], [13] under our attacks, with operational details in Section II.

Adversarial Authorship Attribution. Brennan et al. [10] proposed adversarial stylometry to evade authorship attribution in textual documents. Their framework generates adversarial documents for two purposes: *obfuscation*, where an author hides their identity, and *imitation*, where an author mimics another's style. Both approaches involve human intervention. For code authorship attribution, Simko et al. [30] evaluated attribution under code forgeries by recruiting programmers to create forgeries and analysts to detect them. Meng et al. [23] introduced attacks on binary-based attribution, modifying feature vectors to achieve attack objectives. Matyukhina et al. [22] and Quiring et al. [27] developed transformation techniques to conceal or imitate coding styles. Liu et al. [20] proposed SCAD, which employs a substitute model to assess 37 code transformations that preserve functionality. Using a customized Jacobian-based saliency map, SCAD optimizes transformations before querying the target model, requiring exhaustive code modifications. This work generates adversarial code examples through perturbations that achieve adversarial objectives without requiring full code transformation. Another line of research explores defenses against such attacks. Li et al. [19] introduced RoPGen, which employs adversarial training and data augmentation to enhance model robustness against style imitation and obfuscation attacks.

This study investigates various adversarial scenarios and their impact on authorship attribution. Future research may focus on defenses, including evaluating RoPGen against adversarial code-based attacks. The comparison with related works is presented in Table I.

VIII. Conclusion

This work examined the robustness of six code authorship attribution systems under attacks using code-level perturbations.

We defined targeted and non-targeted attack objectives, leveraging perturbations to disrupt authorship recognition while preserving functionality. Our results demonstrated that increasing perturbation size amplifies its impact on attribution, though targeted techniques were fooled with minimal changes. All systems were compromised under adversarial conditions, significantly reducing model confidence compared to baseline performance.

REFERENCES

[1] Google Code Jam, (n.d.). May 2023. Accessed: Apr. 2023. [Online]. Available: https://github.com/google/coding-competitions-archive

[2] Stunnix, (n.d.). Accessed: Apr. 2023. [Online]. Available: http://stunnix.com

[3] The Tigress Diversifying C Virtualizer, 2017. Accessed: Apr. 2023. [Online]. Available: http://tigress.cs.arizona.edu

[4] M. Abuhamad, T. AbuHmed, A. Mohaisen, and D. Nyang, "Large-scale and language-oblivious code authorship identification," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 101–114.

[5] M. Abuhamad, T. Abuhmed, D. Mohaisen, and D. Nyang, "Large-scale and robust code authorship identification with deep feature learning," *ACM Trans. Privacy Secur.*, vol. 24, no. 4, pp. 1–35, 2021.

[6] M. Abuhamad, T. Abuhmed, D. Nyang, and D. Mohaisen, "Multi-χ: Identifying multiple authors from source code files," in *Proc. Privacy Enhancing Technol.*, vol. 2020, no. 3, pp. 25–41, 2020.

[7] M. Abuhamad, J.-S. Rhim, T. AbuHmed, S. Ullah, S. Kang, and D. Nyang, "Code authorship identification using convolutional neural networks," *Future Gener. Comput. Syst.*, vol. 95, pp. 104–115, 2019.

[8] S. Alrabaee, M. Debbabi, and L. Wang, "On the feasibility of binary authorship characterization," *Digit. Investigation*, vol. 28, pp. S3–S11, 2019.

[9] S. Alrabaee, N. Saleem, S. Preda, L. Wang, and M. Debbabi, "OBA2: An onion approach to binary code authorship attribution," *Digit. Investigation*, vol. 11, pp. S94–S103, 2014.

[10] M. Brennan, S. Afroz, and R. Greenstadt, "Adversarial stylometry: Circumventing authorship recognition to preserve privacy and anonymity," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 3, 2012, Art. no. 12.

[11] S. Burrows, A. Uitdenbogerd, and A. Turpin, "Comparing techniques for authorship attribution of source code," *Softw. Pract. Experience*, vol. 44, no. 1, pp. 1–32, 2014.

[12] S. Burrows, A. L. Uitdenbogerd, and A. Turpin, "Application of information retrieval techniques for source code authorship attribution," in *Proc. Int. Conf. Database Syst. Adv. Appl.*, 2009, pp. 699–713.

[13] A. Caliskan-Islam et al., "De-anonymizing programmers via code stylometry," in *Proc. 24th USENIX Conf. Secur. Symp.*, 2015, pp. 255–270.

[14] A. Caliskan-Islam et al., "When coding style survives compilation: De-anonymizing programmers from executable binaries," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–15.

[15] N. Carlini and D. A. Wagner, "Towards evaluating the robustness of neural networks," in *Proc. IEEE Symp. Secur. Privacy*, 2017, pp. 39–57.

[16] Y. Ding et al., "Interpreting universal adversarial example attacks on image classification models," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 4, pp. 3392–3407, Jul./Aug. 2023.

[17] G. Frantzeskou, E. Stamatatos, S. Gritzalis, and S. Katsikas, "Effective identification of source code authors using byte-level information," in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 893–896.

[18] I. Krsul and E. H. Spafford, "Refereed paper: Authorship analysis: Identifying the author of a program," *Comput. Secur.*, vol. 16, no. 3, pp. 233–257, Jan. 1997.

[19] Z. Li, G. Chen, C. Chen, Y. Zou, and S. Xu, "RoPGen: Towards robust code authorship attribution via automatic coding style transformation," in *Proc. 44th Int. Conf. Softw. Eng.*, 2022, pp. 1906–1918.

[20] Q. Liu, S. Ji, C. Liu, and C. Wu, "A practical black-box attack on source code authorship identification classifiers," *IEEE Trans. Inf. Forensics Security*, vol. 16, pp. 3620–3633, 2021.

[21] S. G. Macdonell, A. R. Gray, G. MacLennan, and P. J. Sallis, "Software forensics for discriminating between program authors using case-based reasoning, feedforward neural networks and multiple discriminant analysis," in *Proc. 6th Int. Conf. Neural Inf. Process.*, 1999, pp. 66–71.

[22] A. Matyukhina, N. Stakhanova, M. Dalla Preda, and C. Perley, "Adversarial authorship attribution in open-source projects," in *Proc. 9th ACM Conf. Data Appl. Secur. Privacy*, 2019, pp. 291–302.

[23] X. Meng, B. P. Miller, and S. Jha, "Adversarial binaries for authorship identification," 2018, *arXiv: 1809.08316*.

[24] X. Meng, B. P. Miller, and K.-S. Jun, "Identifying multiple authors in a binary program," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2017, pp. 286–304.

[25] N. Papernot, P. D. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *Proc. IEEE Eur. Symp. Secur. Privacy*, 2016, pp. 372–387.

[26] B. N. Pellin, "Using classification techniques to determine source code authorship," White Paper, Dept. Comput. Sci., Univ. Wisconsin, 2000.

[27] E. Quiring, A. Maier, and K. Rieck, "Misleading authorship attribution of source code using adversarial learning," in *Proc. 28th USENIX Conf. Secur. Symp.*, 2019, pp. 479–496.

[28] N. Rosenblum, X. Zhu, and B. Miller, "Who wrote this code? Identifying the authors of program binaries," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2011, pp. 172–189.

[29] M. Sharif, S. Bhagavatula, L. Bauer, and M. K. Reiter, "Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition," in *Proc. 2016 ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 1528–1540.

[30] L. Simko, L. Zettlemoyer, and T. Kohno, "Recognizing and imitating programmer style: Adversaries in program authorship attribution," in *Proc. Privacy Enhancing Technol.*, vol. 2018, no. 1, pp. 127–144, 2018.

[31] B. Wu, S. Wang, X. Yuan, C. Wang, C. Rudolph, and X. Yang, "Defeating misclassification attacks against transfer learning," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 2, pp. 886–901, Mar./Apr. 2023.

Mohammed Abuhamad (Senior Member, IEEE) received the PhD degree in computer science from the University of Central Florida in 2020, and the PhD degree in electrical and computer engineering from INHA University in 2020. He is currently an assistant professor of computer science with Loyola University Chicago. His research interests include deep learning-based applications in information security, software and mobile/IoT security, and adversarial machine learning.

Changhun Jung (Member, IEEE) received the MS and PhD degrees in computer science from Inha University, Korea, in 2017 and 2022, respectively, with a focus on information and cyber security. From 2022 to 2022, he was a postdoctoral researcher with Ewha Womans University, where he worked on machine learning-based network security, traffic monitoring, access control, programmable network switches, and medical image analysis using convolutional neural networks. Since 2022, he has been a senior research engineer with Hyundai Motor Company, Gyeonggi-do, Korea, where he is involved in cyber security standardization, including contributions to ITU-T SG17 Q13 on Intelligent Transport System (ITS) security. His research interests include network and system security, machine learning for cybersecurity, programmable networks, and secure intelligent transportation systems.

David Mohaisen (Senior Member, IEEE) received the MSc and PhD degrees in computer science from the University of Minnesota in 2012. He is currently a full professor with the University of Central Florida, where he directs the Security and Analytics Lab. From 2015 to 2017, he was an assistant professor with SUNY Buffalo, and from 2012 to 2015, he was a senior research scientist with Verisign Labs. His research spans networked systems security, online privacy, measurements, blockchain systems, and adversarial machine learning. He has served as an associate editor for *IEEE Transactions on Mobile Computing*, *IEEE Transactions on Cloud Computing*, *IEEE Transactions on Parallel and Distributed Systems*, and *IEEE Transactions on Dependable and Secure Computing*. He is a senior member of the ACM (2018), a distinguished speaker of the ACM, and a distinguished visitor of the IEEE Computer Society.

DaeHun Nyang (Member, IEEE) received the BEng degree with a double major in electronic engineering and computer science from the Korea Advanced Institute of Science and Technology (KAIST) in 1994, and the MS and PhD degrees in computer science from Yonsei University, Korea, in 1996 and 2000, respectively. From 2000 to 2003, he was a senior researcher with the Electronics and Telecommunications Research Institute (ETRI), Korea. He was a professor with the Department of Computer Science and Engineering, Inha University from 2003 to 2020. Since 2020, he has been a professor with the Department of Cyber Security, the College of Artificial Intelligence, Ewha Womans University, where he also serves as chief information officer (CIO) since 2025. He is the founding director of the Information Security Research Laboratory, established in 2003. He serves as an executive director of the Korean Institute of Information Security and Cryptology (KIISC), was editor-in-chief of the KIISC Journal from 2021 to 2024, and is currently a section editor for the ETRI Journal in the Information Security area. His research interests include network measurement, counting algorithms, fast hash tables, encrypted traffic analysis (including TOR/TLS), in-network security (INS) using P4-programmable routers, differential privacy in AI, AI/ML engineering, and compact tensor representations. He is also interested in zero-knowledge interactive proofs, digital signatures, usable security, and blockchain systems. He is a member of ACM and KIISC.