# RoPGen: Towards Robust Code Authorship Attribution via Automatic Coding Style Transformation

Zhen Li[*†], Guenevere (Qian) Chen[*], Chen Chen[♯], Yayi Zou[§], Shouhuai Xu[‡]
[*]University of Texas at San Antonio, USA
[†]Huazhong University of Science and Technology, China
[♯]Center for Research in Computer Vision, University of Central Florida, USA
[§]Northeastern University, China
[‡]University of Colorado Colorado Springs, USA
zh_li@hust.edu.cn,guenevereqian.chen@utsa.edu,chen.chen@crcv.ucf.edu
20185258@stu.neu.edu.cn,sxu@uccs.edu

## ABSTRACT

Source code authorship attribution is an important problem often encountered in applications such as software forensics, bug fixing, and software quality analysis. Recent studies show that current source code authorship attribution methods can be compromised by attackers exploiting adversarial examples and coding style manipulation. This calls for *robust* solutions to the problem of code authorship attribution. In this paper, we initiate the study on making Deep Learning (DL)-based code authorship attribution robust. We propose an innovative framework called *Robust coding style Patterns Generation* (RoPGen), which essentially learns authors' unique coding style patterns that are hard for attackers to manipulate or imitate. The key idea is to combine *data augmentation* and *gradient augmentation* at the adversarial training phase. This effectively increases the diversity of training examples, generates meaningful perturbations to gradients of deep neural networks, and learns diversified representations of coding styles. We evaluate the effectiveness of RoPGen using four datasets of programs written in C, C++, and Java. Experimental results show that RoPGen can significantly improve the robustness of DL-based code authorship attribution, by respectively reducing 22.8% and 41.0% of the success rate of targeted and untargeted attacks on average.

## CCS CONCEPTS

• **Security and privacy → Software security engineering**.

## KEYWORDS

Authorship attribution, source code, coding style, robustness, deep learning

**ACM Reference Format:**
Zhen Li[*†], Guenevere (Qian) Chen[*], Chen Chen[♯], Yayi Zou[§], Shouhuai Xu[‡]. 2022. RoPGen: Towards Robust Code Authorship Attribution via Automatic Coding Style Transformation. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510181


*ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA*

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510181

## 1 INTRODUCTION

Software forensics analysis aims to determine whether or not there is software intellectual property infringement or theft associated with some given software code. One useful technique for this purpose is source code authorship attribution [13, 27], which aims to identify the author(s) of a given software program [14, 25]. This technique has been used for many applications, such as code plagiarism detection, criminal prosecution (e.g., identifying the author of a piece of malicious code), corporate litigation (e.g., determining whether a piece of code is written by a former employee who violates any non-compete clause of contract), bug fixing [8, 38], and software quality analysis [46].

There are multiple approaches to source code authorship attribution, including statistical analysis [18, 26], similarity measurement [12, 21, 27], and machine learning [1, 4, 7, 11, 14, 24, 36, 47, 51]. Recent studies show that current source code authorship identification methods can be compromised by two classes of attacks: the ones exploiting *adversarial examples* [31, 37] and the ones exploiting *coding style imitation/hiding* [34, 35, 42]. For instance, leveraging adversarial examples [1, 24] can cause misattribution of more than 99% software programs in the GoogleCodeJam competition dataset [37]; whereas leveraging the coding style hiding [12, 18, 24] can cause misattribution of all of the software programs in a GitHub dataset [34]. The state-of-the-art is that current code authorship attribution methods are vulnerable to these attacks. This calls for research on enhancing the robustness of code authorship attribution methods against attacks.

**Our contributions**. In this paper, we initiate the study on enhancing the robustness of Deep Learning (DL)-based code authorship attribution methods. We choose to focus on this family of methods because they can automatically learn coding style patterns (i.e., avoiding laborious involvement of domain experts) and are very promising for real-world adoption [1, 4, 7, 11, 47, 51]. Effectively, we tackle the following problem: *How can we enhance the robustness of DL-based code authorship attribution against attacks?* For this purpose, we need to address *two challenges.*

The first challenge is to consider more attacks than what have been investigated in the literature; otherwise, the resulting defenses would be specific to the known attacks and will soon become obsolete when new attacks are introduced. This is especially true because the known attacks are geared towards domain expert-defined features [34], which may not be sustainable and would sooner or later need to be replaced by automatic feature learning. This inspires us to explore new/unknown attacks so that we can design defenses that can enhance robustness against both known and new attacks. For this purpose, we introduce two new attacks which exploit automatic coding style imitation and hiding; these attacks can be applied against both DL-based code authorship attribution and other methods. The new attacks leverage our systematization of semantics-preserving coding style attributes and transformations, which may be of independent value. The attacks are of black-box type because they do not need to know the target code authorship attribution methods; instead, they imitate the target author's coding style or hide the true author's.

The second challenge is to design effective defenses against the known and new attacks mentioned above, while accommodating a range of neural network structures (rather than a specific one). To address this challenge, it would be natural to leverage the idea of *adversarial training* because it has been widely used in other settings [9, 33, 40]. However, our experimental results show that such adversarial training approaches applied in these settings [9, 33, 40] cannot effectively mitigate the known and new attacks mentioned above (as what will be described in Table 11 of Section 5.4). This prompts us to propose an innovative framework, called *Robust coding style Patterns Generation* (RoPGen). The key idea is to incorporate *data augmentation* and *gradient augmentation* to learn robust coding style patterns which are difficult for attackers to manipulate or imitate. The role of *data augmentation* is to increase the amount and diversity of software programs for training purposes. This is achieved by augmenting programs in two ways: (i) imitating coding styles of other authors; and (ii) perturbing programs' coding styles to a small degree without changing their authorship. The role of *gradient augmentation* is to learn robust DL models with diversified representations by incurring perturbations to gradients of deep neural networks. This is achieved as follows: at each training iteration, we sample multiple sub-networks with a certain fraction of the nodes at each layer of the network; then, we use the sampled sub-networks to construct the network with diversified representations during the weights-sharing training process. The resulting model learns robust coding style patterns which would be difficult to exploit. It is worth mentioning that gradient augmentation has been used as a regularization method to alleviate *over-fitting* of deep neural networks in image classification [50]; we are the first to use it for *robust* authorship attribution.

To evaluate the effectiveness of RoPGen, we use four datasets of programs written in C, C++, and Java, namely GCJ-C++ [37], GitHub-Java [51], GitHub-C, and GCJ-Java. Among them, GCJ-C++ and GCJ-Java are two sets of programs written by authors who participate in programming competitions for solving a given set of problems; GitHub-Java and GitHub-C are two sets of real-world programs written by different programmers for varying purposes; GitHub-C and GCJ-Java are created for the purpose of the present paper. Experimental results show that RoPGen can significantly improve the robustness of DL-based code authorship attribution, respectively reducing the success rate of targeted and untargeted attacks by 22.8% and 41.0% on average. We have made the datasets available at `https://github.com/RoPGen/RoPGen`. We will publish the source code of RoPGen on the same website.

**Paper organization**. We discuss the notion of coding styles in Section 2, introduce two new attacks in Section 3, describe RoPGen in Section 4, present experimental results in Section 5, discuss limitations in Section 6 and related prior studies in Section 7, and conclude this paper in Section 8.

## 2 THE NOTION OF CODING STYLES

The problem of source code authorship attribution has two variants: single-authorship attribution [1, 2, 4, 7, 11, 12, 14, 18, 21, 24, 26, 27, 36, 47, 51] vs. multi-authorship attribution [3, 17]. Since most studies focus on the former variant while the latter is little understood, we focus on addressing the former variant.

**Coding style attributes**. The premise for achieving authorship attribution is that each author has a unique *coding style*, which can be defined based on four types of attributes related to programs' layout, lexical, syntactic, and semantic information. Layout attributes include code indentation, empty lines, brackets, and comments [24]. Lexical attributes describe tokens (e.g., identifier, keyword, operator, and constant), the average length of variable names, the number of variables, and the number of `for` loop statements [1, 24]. Syntactic attributes describe a program's *Abstract Syntax Tree* (AST), including syntactic constructs (e.g., unary and ternary operators) and tree structures (e.g., frequency of adjacent nodes and average depth of AST node types) [1, 11, 34]. Semantic attributes describe a program's control flows and data flows (e.g., "`for`", "`while`", "`if, else if`", "`switch, case`", and execution order of statements) [34].

Since coding styles and their attributes are related to programming languages, we focus on C, C++, and Java programs because they are widely used, while leaving the treatment of other languages to future studies. Even for these specific programming languages, their coding style attributes are scattered in the literature [31, 34, 37, 42]. This prompts us to systematize attributes according to the following observations: (i) layout attributes can be easily manipulated by code formatting tools [37] (e.g., Code Beautify [16] and Editor Config [20]); (ii) those attributes, whose values cannot be automatically modified without changing a program's semantics, would not be exploited by an attacker because they make imitation attacks hard to succeed; and (iii) those attributes, whose values are rarely used (e.g., making programs unnecessarily complicated), would not be exploited by an imitation attacker. As highlighted in Table 1, these observations lead to 23 coding style attributes, which span across lexical, syntactic, and semantic information.

**Leveraging coding style attributes as a starting point for robust authorship attribution**. For this purpose, we need to consider two issues. First, we consider *granularity* of coding style attributes, namely token vs. statement vs. basic block vs. function. This is important because code transformations on coarse-grained attributes may demand larger degrees of perturbations to programs.

- *Token*-level attributes (#1-#5 in Table 1): They describe the elements in a program's statements: identifier naming method (#1), usage of temporary variable names (#2), usage of non-temporary local identifier names (#3), usage of global declarations (#4), and

**Table 1: C, C++, and Java coding style attributes serving as a starting point for robust code authorship attribution**

| Granularity | Attribute # | Description | Value | Type | Exhaustive? | Language |
|---|---|---|---|---|---|---|
| Token | 1 | Identifier naming method | Camel case (e.g., `myCount`), Pascal case (e.g., `MyCount`), words separated by underscores, or identifiers starting with underscores. | Lexical | Yes | C, C++, Java |
| | 2 | Usage of temporary variable names | Variable names defined in a compound statement of a function. | Lexical | No | C, C++, Java |
| | 3 | Usage of non-temporary local identifier names | Variable names defined in functions but not defined in compound statements, or user-defined function calls. | Lexical | No | C, C++, Java |
| | 4 | Usage of global declarations | Global constants declared outside of functions. | Lexical | No | C, C++ |
| | 5 | Access of array/pointer elements | Use the form of array indexes or pointers, e.g., `arr[i]` and `*(arr+i)`. | Lexical | Yes | C, C++ |
| Statement | 6 | Location of defining local variables | Local variables are defined at the beginning of the variable scope, or each local variable is defined when used for the first time. | Syntactic | Yes | C, C++, Java |
| | 7 | Location of initializing local variables | Local variables are initialized and defined in same statements, or in different statements. | Syntactic | Yes | C, C++, Java |
| | 8 | Definition (and initialization) of multiple variables with same types | Multiple variables with same types are defined (and initialized) in a statement or in multiple statements. | Syntactic | Yes | C, C++, Java |
| | 9 | Variable assignment | Multiple variable assignments are in a statement (e.g., `tmp=++i;`) or multiple statements (e.g., `++i; tmp=i;`). | Syntactic | Yes | C, C++, Java |
| | 10 | Increment/decrement operation | Use increment (or decrement) operator with different locations, e.g., (i) `i++;` (ii) `++i;` (iii) `i=i+1;` (iv) `i+=1;`. | Syntactic | Yes | C, C++, Java |
| | 11 | User-defined data types | Use `typedef` to rename a data type or not. | Syntactic | No | C, C++ |
| | 12 | Macros | Use macros to replace constants and expressions or not. | Syntactic | No | C, C++ |
| | 13 | Included header files or imported classes | Header files included in C/C++ programs and classes imported in Java programs. | Semantic | No | C, C++, Java |
| | 14 | Usage of return statements | Use `return 0;` to explicitly return success in `main` function or not. | Semantic | Yes | C, C++ |
| | 15 | Usage of namespaces | Use namespace `std` or not. | Semantic | Yes | C++ |
| | 16 | Synchronization with stdio | Enable or remove the synchronization of C++ streams and C streams. | Semantic | Yes | C++ |
| | 17 | Stream redirection | Use `freopen` to redirect predefined streams to specific files or not. | Semantic | Yes | C, C++ |
| | 18 | Library function calls | C++ library function calls (e.g., `cin`, `cout`) or corresponding C library function calls with the same functionalities (e.g., `scanf`, `printf`). | Semantic | Yes | C++ |
| | 19 | Memory allocation | Static array allocation (e.g., `int arr[100];`) or dynamic memory allocation (e.g., `int *arr=malloc(100*sizeof(int));`). | Semantic | Yes | C, C++ |
| Basic block | 20 | Loop structures | Use `for` structure or `while` structure. | Semantic | Yes | C, C++, Java |
| | 21 | Conditional structures | Use conditional operator, `if-else`, or `switch-case` structure. | Semantic | Yes | C, C++, Java |
| | 22 | Compound `if` statements | Use a logical operator in an `if` condition (e.g., `if(a && b)`) or use multiple `if` conditions (e.g., `if(a){if(b){...}}`). | Semantic | Yes | C, C++, Java |
| Function | 23 | Usage of functions | The maximum layer number of control statements and loops that are nested within each other, or the number of lines of code in the function. | Semantic | No | C, C++, Java |

access of array/pointer elements (#5). For instance, attribute #2 of the program shown in Figure 1(a) is described by temporary variable names `case_it`, `st`, `ss`, `ans`, `pos`, and `i`.

- *Statement*-level attributes (#6-#19 in Table 1): They describe the location of defining local variable (#6), the location of initializing local variables (#7), the definition (and initialization) of multiple varialbles with same types (#8), variable assignment (#9), increment/decrement operation (#10), user-defined data types (#11), macros (#12), included header files or imported classes (#13), Usage of return statements (#14), usage of namespaces (#15), synchronization with stdio (#16), stream redirection (#17), library function calls (#18), and memory allocation (#19). For instance, attribute #18 of the program shown in Figure 1(a) is described by library functions `cin` (Line 7) and `cout` (Line 20).
- *Basic block*-level attributes (#20-#22 in Table 1): They describe loop structures (#20), conditional structures (#21), and compound `if` statements (#22). For instance, attribute #20 of the program shown in Figure 1(a) is described by two `for` structures (Line 4 and Line 13) and a `while` structure (Line 11).
- *Function*-level attribute (#23 in Table 1): At this granularity, coding styles describe the usage of functions, namely (i) the maximum number of layers of nested compound statements (e.g., control statements and loops) or (ii) the number of lines of code in a function. For instance, attribute #23 of the program shown in Figure 1(a) is the maximum number of layers of nested compound statements, which is 3 in this case (i.e., `for-while-for`).

Second, we propose distinguishing those coding style attributes whose domains are exhaustive from those that are not; the term "exhaustive" means that an attribute's domain contains few values (e.g., the kinds of loop structures), and a domain is treated as non-exhaustive if its domain contains many values (e.g., the number of possible variable names can be very large). This is important because a non-exhaustive attribute would naturally demand more perturbed examples for adversarial training purposes. As shown in Table 1, exhaustive attributes include attributes #1 and #5 at the token-level granularity, #6-#10 and #14-#19 at the statement-level granularity, and #20-#22 at the basic block-level granularity. For instance, #20 (i.e., loop structures) has only two values in C, C++, and Java programs: `for` and `while`. Non-exhaustive attributes include attributes #2-#4 at the token level, #11-#13 at the statement level, and #23 at the function level. For instance, #2 (i.e., usage of temporary variable names) is non-exhaustive because temporary variables can have arbitrary names. For the program described in Figure 1(a), the value of attribute #2 includes `case_it`, `st`, `ss`, `ans`, `pos`, and `i`.

## 3 TWO NEW ATTACKS

We investigate two new attacks against code authorship attribution, one is coding style *imitation* attack and the other is coding style *hiding* attack. These attacks are new and can make our defense widely applicable because they are waged *automatically* and are waged against both DL-based code authorship attribution and other methods. In contrast, attacks presented in the literature are manual [42], semi-automatic [35], or automatic but not applicable to DL-based code authorship attribution [34].

Denote by $\mathcal{A} = \{A_1, \ldots, A_\delta\}$ a finite set of authors and by $M$ the code authorship attribution method in question. The attacker has black-box access to $M$, meaning: (i) the attacker can query any program $p$ to $M$ which returns the author of $p$ or $M(p)$; and (ii) how $M$ is obtained is unknown to the attacker. In the threat model, the attacker manipulates $p$ written by $A_s$ (e.g., Alice) into a variant program $p'$ via semantics-preserving code transformations, where $p' \neq p$. The attacker's goal is:

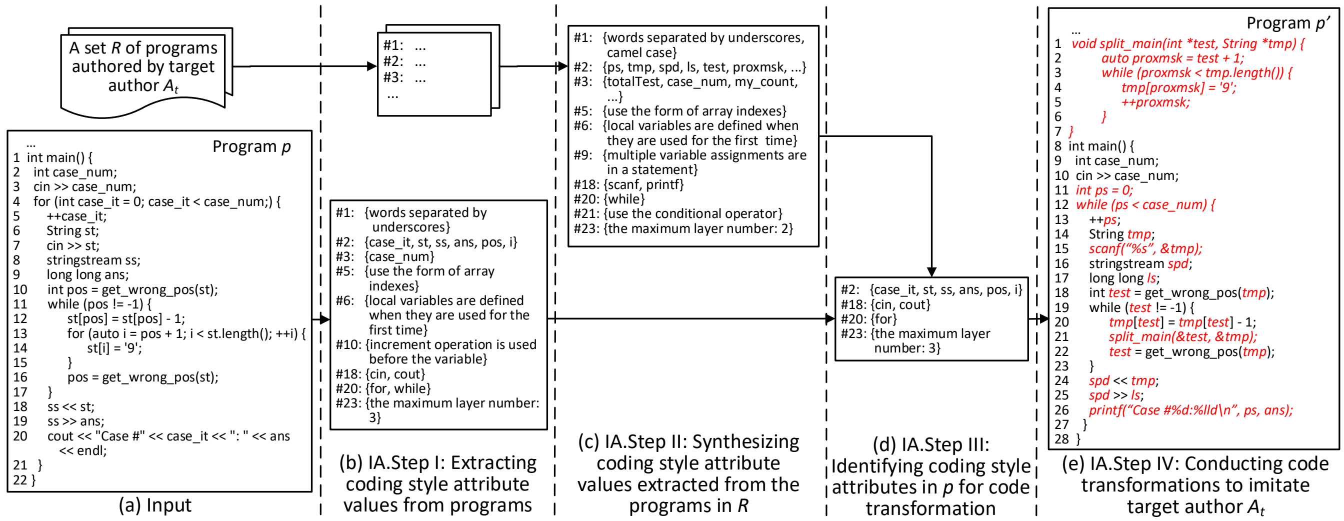


**Figure 1: An example showing generation of C++ program $p'$ for targeted attack (modified code is highlighted in red and italics)**

- In a *targeted* attack with target author $A_t$ (e.g., Bob) where $t \neq s$, the attacker's goal is to make $M$ misattribute $p'$ to $A_t$, namely $M(p') = A_t$ while noting that $M$ would correctly attribute $p$ to $A_s$, namely $M(p) = A_s$. That is, the attacker attempts to manipulate a program written by Alice into a semantically-equivalent program which will be misattributed to Bob.
- In an *untargeted* attack, the attacker's goal is to make $M$ misattribute $p'$ to any other author $A_u$ than $A_s$, namely $M(p') = A_u$ where $A_u \in \mathcal{A} - \{A_s\}$.

## 3.1 Automatic Coding Style Imitation Attack

In this attack, the attacker, $A_s \in \mathcal{A}$ in typical use cases, takes as input: (i) the set $\mathcal{A}$ of authors; (ii) a program $p$ authored by $A_s$; and (iii) a set $R$ of programs authored by target author $A_t \in \mathcal{A}$ where $t \neq s$. The goal of $A_s$ is to *automatically* transform program $p$ to program $p'$ such that $p'$ preserves $p$'s functionality and $M$ misattributes $p'$ to $A_t$. The attack proceeds as follows.

- **IA.Step I: *Extracting coding style attribute values from program p and the programs in R (authored by target author $A_t$).*** Attacker $A_s$ generates the coding styles of program $p$ and all programs in $R$ by leveraging the 23 attributes mentioned above (Table 1). As a running example, Figure 1(a) shows $A_s$'s program $p$ and Figure 1(b) shows the values of the 9 applicable attributes of $p$. For instance, in order to obtain the value of attribute #1 (i.e., identifier naming method), $A_s$ can identify all of the user-defined variable and function call names used in $p$ (i.e., `case_num`, `case_it`, `st`, `ss`, `ans`, `pos`, `get_wrong_pos`, and `i` in this case). Then, $A_s$ can obtain the identifier naming method for each user-defined variable and function call name. Specifically, the value of attribute #1 corresponding to `case_num`, `case_it`, and `get_wrong_pos` is "words separated by underscores"; the other variable and function call names (i.e., `st`, `ss`, `ans`, `pos`, and `i`) cannot be represented by attribute #1 because these identifiers have no naming rules. Therefore, the value of attribute #1 of program $p$ is "words separated by underscores".
- **IA.Step II: *Synthesizing coding style attribute values extracted from the programs in R.*** Having extracted attribute values from *individual* programs in $R$, we need to synthesize them into a single value for each attribute to obtain target author $A_t$'s coding style. In the case an attribute is numeric, we propose using the average of an attribute's values (as observed from the programs in $R$) to represent $A_t$'s coding style with respect to the attribute. In the case an attribute is non-numeric, we propose using the ordered set of an attribute's distinct values in the descending order of their frequency to represent $A_t$'s coding style with respect to the attribute. As a running example, Figure 1(c) illustrates $A_t$'s coding style attributes synthesized from the programs in $R$. For instance, the synthesized value of numeric attribute #23 (usage of function) is 2, which is the average of values observed from the programs in $R$. Non-numeric attribute #1 (identifier naming method) takes two distinct values: "words separated by underscores" (as observed from most programs in $R$) and "camel case" (as observed from the other programs in $R$); the synthesized value of attribute #1 is the ordered set "{words separated by underscores, camel case}" as the former has a higher frequency.
- **IA.Step III: *Identifying coding style attributes in p for code transformation.*** Having obtained attacker $A_s$'s coding style attributes from program $p$ (IA.Step I) and target author $A_t$'s coding style attributes from $R$ (IA.Step II), we identify the *discrepant* attributes, namely the attributes that take different values with respect to $A_s$ and $A_t$, as candidates for code transformation to make $p$ imitate $A_t$'s coding style. For a numeric attribute, discrepancy means that the difference between its value derived from $p$ and its value derived from $R$ is above a given threshold $\tau$. For a non-numeric attribute, discrepancy means that its value derived from $p$ is not a subset of its value derived from $R$. As a running example, Figure 1 (b) and (c) show that the value of numeric attribute #23 derived from $p$ is discrepant with the value derived from $R$ because their difference, 1, is larger than the threshold $\tau = 0$; the values of non-numeric attributes #2, #18, and #20 derived from $p$ are discrepant with their counterparts derived from $R$ because the former is not a subset of the latter, respectively.

As shown in Figure 1 (d), these four discrepant attributes are candidates for code transformations to imitate $A_t$'s coding style.

- **IA.Step IV:** ***Conducting code transformations to imitate target author $A_t$.*** This step is to change the values of the discrepant attributes identified in IA.Step III to imitate target author $A_t$, leading to a transformed (or manipulated) program $p'$ which preserves $p$'s functionality. We conduct code transformations on individual program files based on *srcML* [44], which can preserve program functionalities while supporting multiple programming languages. As a running example, Figure 1 (e) shows the manipulated program $p'$ obtained by sequentially transforming the values of attributes #2, #18, #20, and #23 derived from program $p$, while assuring that each transformation preserves the functionality of the program in question. Take attribute #23 for example. The `main` function (Line 1 in Figure 1 (a)) is split into two functions `main` (Line 8 in Figure 1 (e)) and `split_main` (Line 1 in Figure 1 (e)).

## 3.2 Automatic Coding Style Hiding Attack

In this attack, attacker $A_s \in \mathcal{A}$ takes as input the set $\mathcal{A}$ of authors and a program $p$ authored by $A_s$. As mentioned above, the goal of $A_s$ is to manipulate program $p$ to another program $p'$, which preserves $p$'s functionality but will not be attributed to $A_s$. To achieve this, we propose leveraging the preceding imitation attacks by choosing a target author with the highest misattribution probability. Details follow.

- **HA.Step I:** ***Extracting coding style attribute values from program $p$.*** This is the same as IA.Step I.
- **HA.Step II:** ***Obtaining the coding style of each author $A_d$.*** For each $A_d \in \mathcal{A} - \{A_s\}$, we generate $A_d$'s coding style as IA.Step II by treating $A_d$ as the target author.
- **HA.Step III:** ***Identifying the coding style attributes in p for each $A_d$.*** For each author $A_d \in \mathcal{A} - \{A_s\}$, we identify the coding style attributes extracted from $p$ that are discrepant with $A_d$'s. This is the same as IA.Step III by treating $A_d$ as the target author.
- **HA.Step IV:** ***Selecting author $A_u$ for transformation.*** For each $A_d \in \mathcal{A} - \{A_s\}$, we compute the number of lines of code that need to be changed to make $p'$ imitate $A_d$'s coding style. Changing more lines of code in $p$ (e.g., involving attributes #11, #12, and #13) may make $p$ retain fewer original coding styles and thus make an untargeted attack successful with a higher misattribution probability. We select author $A_u \in \mathcal{A} - \{A_s\}$ with the highest misattribution probability as the target author.
- **HA.Step V:** ***Conducting code transformations to imitate author $A_u$.*** This is the same as IA.Step IV with target author $A_u$.

# 4 THE ROPGEN FRAMEWORK

In DL-based authorship attribution, the input at the training phase is a set of $\eta$ training programs with labels, denoted by $P = \{p_k, q_k\}_{k=1}^{\eta}$, where $p_k$ is a training program and $q_k$ is its label (i.e., author). The output is a DL model $M$. Given a finite set of authors $\mathcal{A} = \{A_1, \ldots, A_\delta\}$ and a program $p_k$ authored by $A_s \in \mathcal{A}$, let $\Pr(M, p_k, A_s)$ denote the probability that $M$ predicts that $p_k$ is authored by $A_s$. The attacker manipulates $p_k$ to a different program, denoted by $p'_k$. As discussed above, an *imitation* attacker succeeds when $\Pr(M, p'_k, A_t) = \max_{1\leq z\leq\delta} \Pr(M, p'_k, A_z)$ for a given $t \neq s$; a *hiding* attacker succeeds when $\Pr(M, p'_k, A_s) \neq \max_{1\leq z\leq\delta} \Pr(M, p'_k, A_z)$.

Figure 2 highlights the training phase of RoPGen framework, which trains an enhanced model of $M$, denoted by $M^+$. The input to RoPGen includes: (i) a set $P$ of $\eta$ training programs and their labels, (ii) a set $T \subseteq \mathcal{A}$ of target authors, and (iii) a set $E$ of adversarial examples against model $M$. The basic idea behind RoPGen is to leverage ideas of *data augmentation* and *gradient augmentation*:

- *Data augmentation* aims to increase the amount and diversity of training programs. We achieve this via two ideas: (i) imitating coding styles of the other authors, which is elaborated in Step 1 below; (ii) changing programs' coding styles with small perturbations, which is elaborated in Step 2 below.
- *Gradient augmentation* aims to learn a robust deep neural network with diversified representations by generating meaningful perturbations to gradients. We achieve this by sampling multiple sub-networks, with each involving the first $w_j \times 100\%$ nodes at each layer of the network, where $w_j \in [\alpha, 1]$ and $\alpha$ $(0 < \alpha < 1)$ is the width lower bound. This allows a larger sub-network to contain the representation of a smaller sub-network during weights-sharing training, enabling the former to leverage the representations learned by the latter to construct robust networks with diversified representations. This is elaborated in Step 3 below.

## 4.1 Step 1: Extending the Training Set by Coding Style Imitation

Given a set $T$ of target authors, this step is to extend $P$ by generating programs to imitate the coding styles of the authors in $\mathcal{A}$. We first generate a set $P_1$ of programs imitating the coding styles of the authors in $\mathcal{A}$. Specifically, for each program $p_k \in P$ with label (i.e., authored by) $q_k \in T$, we transform $p_k$ to imitate the coding style of each of the other $\delta - 1$ authors in $\mathcal{A} - \{q_k\}$, while preserving $p_k$'s label. This essentially repeats the imitation attack described in Section 3 for $\delta - 1$ times. Then we obtain the extended set $U = P \cup P_1$ of training programs with labels, which is the input to Step 3 below.

## 4.2 Step 2: Generating Manipulated Programs by Coding Style Perturbation

This step is to generate manipulated programs by coding style perturbation. We consider two situations. First, we can generate a set $E$ of adversarial examples against $M$ and then obtain a set $U'$ of manipulated programs by leveraging $E$ as follows. For each adversarial example $e_r \in E$, we obtain a sequence $T_r$ of transformations which led to $e_r$. Then, for each program $p_k \in P$, we generate a manipulated program $p_{k,r}$ by conducting the sequence $T_r$ of transformations. This leads to $|U'| = |E| \times |P|$ manipulated programs. Second, if it is not easy to generate adversarial examples, we can generate manipulated programs $p_k^1, \ldots, p_k^z$ by perturbing program $p_k$, namely by changing the value of each of the $z$ attributes for each program $p_k \in P$. This leads to a set $U'$ of manipulated programs, where $|U'| = z \times |P|$. Specifically, we first extract $p_k$'s coding style attributes as in IA.Step I (see Section 3). Corresponding to each attribute $c_j$ $(j = 1, \ldots, z)$, we generate a manipulated program $p_k^j$ by randomly selecting a value of $c_j$ and changing it to another value, while preserving $p_k$'s label. For instance, consider program $p$ in

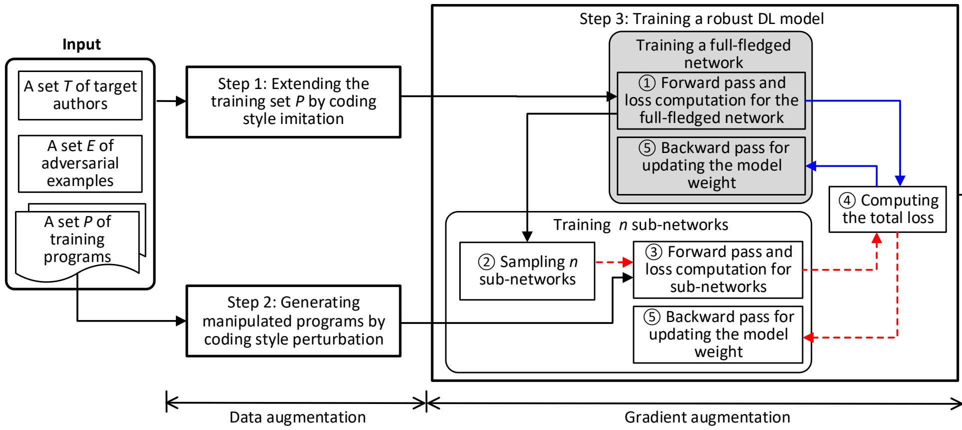


**Figure 2: The RoPGen framework is an enhanced training model, involving data augmentation (Steps 1 and 2) and gradient augmentation (Step 3). Since the data flows share ③ in Step 3, we use solid blue arrows and dotted red arrows to distinguish the training processes of the full-fledged network and sub-networks. The original DL-based training model (baseline) is highlighted with shaded boxes.**

Figure 1 (a). For an exhaustive attribute (e.g., attribute #20), its value (e.g., `while`) can be transformed to another value (e.g., `for`), causing the `while` structure (Lines 10 and 11 in Figure 1 (a)) to be transformed to the `for` structure (i.e., "`for(pos=get_wrong_pos(st); pos!=-1;){`"). For a non-exhaustive attribute (e.g., attribute #2), its value can be transformed to the value corresponding to another randomly selected author's, causing the temporary variable names to become another author's. Finally, we obtain $U'$ which contains manipulated programs with labels.

## 4.3 Step 3: Training a Robust DL Model $M^+$

This step trains a robust model $M^+$ by sampling multiple sub-networks in each training iteration for *gradient augmentation* and generating meaningful perturbations to the gradients of the model. RoPGen uses the extended training set $U$ as the input to the full-fledged network and the set $U'$ of manipulated programs as the input to the sub-networks. Denote by $\mathcal{N}$ the deep neural network and $\theta$ its model parameter. Each training iteration has five substeps:

**Step ①: Forward pass and loss computation for the full-fledged network.** We use the extended set $U$ of training programs (obtained in Step 1) as the input to the full-fledged network. For each training program with its label $(u, v) \in U$, we conduct the forward pass and obtain the predicted value of the full-fledged $\mathcal{N}(\theta, u)$. We compute the full-fledged network's loss using the standard

$$L_{std} = l(\mathcal{N}(\theta, u), v) \tag{1}$$

and loss function $l$ (e.g., cross entropy).

**Step ②: Sampling $n$ sub-networks.** We sample $n$ sub-networks $\mathcal{N}_1, \ldots, \mathcal{N}_n$ from the full-fledged network $\mathcal{N}$. To obtain $\mathcal{N}_j$ ($j = 1, \ldots, n$), we sample the first $w_j \times 100\%$ nodes in each layer of the full-fledged network. The order of nodes at each layer is naturally determined by the full-fledged network (i.e., top-to-bottom in the standard representation of neural networks). We use this order to sample the first $w_j$-fraction of nodes at a layer to obtain a sub-network. These sub-networks will be used to learn different representations from manipulated programs and enhance the robustness of the full-fledged network.

**Step ③: Forward pass and loss computation for sub-networks.** We use $U'$ obtained in Step 2 as the input to each sub-network $\mathcal{N}_j$ because programs in $U'$ are generated with small perturbations and thus suitable for fine-tuning the full-fledged network. Let $\theta_{w_j}$ be the parameter of the sub-network $\mathcal{N}_j$. For each program with its label $(u', v') \in U'$, we conduct the forward pass and obtain prediction $\mathcal{N}(\theta_{w_j}, u')$. The loss $L_{subnet}$ of the $n$ sub-networks is

$$L_{subnet} = \sum_{j=1}^{n} l(\mathcal{N}(\theta_{w_j}, u'), v'). \tag{2}$$

**Step ④: Computing the total loss.** The total loss $L_{RoPGen}$ is the sum of the loss of the full-fledged network and the loss of the sub-networks:

$$L_{RoPGen} = L_{std} + L_{subnet}. \tag{3}$$

**Step ⑤: Updating the model weights**. We conduct the backward pass and leverage the total loss to update model weights, which are shared by the full-fledged network and $n$ sub-networks. This allows different parts of the network to learn diverse representations.

Steps ① to ⑤ are iterated until the model converges to $M^+$.

**Gradient property analysis.** To show how Step 3 augments the gradient, it suffices to consider the full-fledged network $\mathcal{N}$ with one layer. Based on Eq. (1), the full-fledged network $\mathcal{N}$'s gradient $g_{std}$ is

$$g_{std} = \frac{\partial l(\mathcal{N}(\theta, u), v)}{\partial \theta}. \tag{4}$$

Based on Eq. (2), the $n$ sub-networks' gradient $g_{subnet}$ is

$$g_{subnet} = \sum_{j=1}^{n} \frac{\partial l(\mathcal{N}(\theta_{w_j}, u'), v')}{\partial \theta_{w_j}}. \tag{5}$$

Based on Eq. (3), Eq. (4), and Eq. (5), RoPGen's gradient $g_{RoPGen}$ is

$$g_{RoPGen} = g_{std} + g_{subnet}, \tag{6}$$

$g_{subnet}$ can be seen as an augmentation to the raw gradient $g_{std}$, explaining the term "gradient augmentation".

# 5 ROPGEN EXPERIMENTS AND RESULTS

Our experiments aim to answer three Research Questions (RQs):

- **RQ1**: Are the existing DL-based authorship attribution methods robust against the known and new attacks? (Section 5.2)
- **RQ2**: How robust are RoPGen-enabled authorship attribution methods against the known and new attacks? (Section 5.3)
- **RQ3**: Are RoPGen-enabled methods more effective than other adversarial training methods? (Section 5.4)

## 5.1 Experimental Setup

**Datasets.** Our experiments use four datasets: the first two are used in the literature and the last two are introduced in this paper.

- ***GCJ-C++ dataset***. *Google Code Jam* (GCJ) [23] is an annual international programming competition of multiple rounds; each round requires participants to solve some programming challenges. This dataset is created from GCJ in [37] and consists of 1,632 C++ program files from 204 authors. Each author has 8 program files, corresponding to 8 programming challenges, with an average of 74 lines of code per program file.
- ***GitHub-Java dataset***. This dataset is created from GitHub in [51] and consists of 2,827 Java program files from 40 authors, with an average of 76 lines of code per program file.
- ***GitHub-C dataset***. We create this dataset from GitHub, by crawling the C programs of authors who contributed between 11/2020 and 12/2020. We filter the repositories that are marked as forks (because they are duplicates) and the repositories that simply duplicate the files of others. We preprocess these files by removing the comments; we then eliminate the resulting files that (i) contain less than 30 lines of code because of their limited functionalities or (ii) overlap more than 60% of its lines of code with other files. The resulting dataset has 2,072 C files of 67 authors, with an average of 88 lines of code per file.
- ***GCJ-Java dataset***. We create this dataset from GCJ between 2015 and 2017. Since some authors participate in GCJ for multiple years, we merge their files according to their IDs. We select the authors who have written at least 30 Java program files. The dataset has 2,396 Java files of 74 authors, with an average of 139 lines of code per file.

**Evaluation metrics.** To evaluate effectiveness of code authorship attribution methods, we adopt the widely-used accuracy and attack success rate metrics [19]. Recall that $M$ is a DL-based attribution method, $M^+$ is the RoPGen-enabled version of $M$, and $G$ is an attack method. The accuracy of $M$, denoted by $Acc(M)$, is the fraction of the test programs that are correctly labelled by $M$. The attack success rate of an imitation attack $G$ against model $M$, denoted by $Asr_{tar}(M, G)$, is the fraction of the manipulated programs that are misattributed to the target author by $M$, among all of the test programs. The attack success rate of a hiding attack $G$ against model $M$, denoted by $Asr_{unt}(M, G)$, is the fraction of the manipulated programs that are misattributed to another author by $M$, among the correctly classified test programs.

**Implementation.** We choose the following two DL-basd attribution methods reported in [1, 11] because they represent the state-of-the-art and are open-sourced as well as language-agnostic.

**Table 2: Accuracies of two DL-based attribution methods on four datasets (metrics unit: %)**

| Method | GCJ-C++ | GitHub-C | GCJ-Java | GitHub-Java |
|---|---|---|---|---|
| DL-CAIS | 88.2 | 79.9 | 98.5 | 88.4 |
| PbNN | 84.8 | 76.7 | 86.2 | 95.4 |

- ***DL-CAIS*** [1]. This method adopts lexical features to represent programs, leverages recurrent neural network and fully-connected layers to learn representations, and uses random forest to predict authorship.
- ***PbNN*** [11]. This method adopts code2vec [6] to represent programs. It decomposes a program to multiple paths in its AST, transforms the path-contexts to vectors, and uses a fully-connected layer with softmax activation to predict authorship.

We use a stratified $\kappa$-fold cross validation, where the dataset is split into $\kappa$-1 subsets for training and the rest for testing. Following the training strategy of PbNN [11], we set $\kappa$=10 for the GitHub-C, GCJ-Java, and GitHub-Java datasets. Following the training strategy of DL-CAIS [1], we set $\kappa$=8 for the GCJ-C++ dataset. This cross validation is repeated $\kappa$ times, where each subset is used for testing the model trained from the other $\kappa$-1 subsets. The evaluation metrics are computed as the average of the $\kappa$ validations. We use the method reported in [37] to generate adversarial examples and leverage *srcML* [44] to generate manipulated programs and launch coding style imitation/hiding attacks. We choose *srcML* because it can conduct code transformations on an individual program file and can support multiple programming languages. We conduct experiments on a computer with a NVIDIA GeForce GTX 3080 GPU and an Intel i9-10900X CPU running at 3.70GHz.

## 5.2 Robustness of Existing Methods (RQ1)

To determine whether existing authorship attribution methods are robust against the known and new attacks, we attack two DL-based attribution methods (i.e., DL-CAIS [1] and PbNN [11]) on four datasets (i.e., GCJ-C++, GitHub-Java, GitHub-C, and GCJ-Java), corresponding to eight DL models.

Table 2 shows that DL-CAIS and PbNN on four datasets achieve 88.8% and 85.8% accuracies on average. For the *known* attacks, we use the Monte-Carlo tree search to generate adversarial examples [37] for each program in the test set of the GCJ-C++ and GitHub-C datasets, since the approach focuses on C/C++ programs. To preserve the main coding styles of the original authors, we leverage the notion of $\varphi$*-adversary*, which means a program can apply at most $\varphi$ code transformations when generating adversarial examples [39]. For the *new* attacks, we use the automatic coding style imitation and hiding attacks we propose to generate manipulated programs.

**Robustness against targeted attacks.** Due to the quadratic number of pairs, we perform targeted attacks on 20 random authors for each dataset and use two program files as the external source (i.e., not part of the training or test set) for extracting each target author's coding style, as per [37]. For each program authored by these 20 authors in the test set, we respectively take the 19 authors other than the author to whom the program is attributed as the target author. For generating adversarial examples, we set $\varphi = 3$ (i.e., 3-adversary when generating adversarial examples. We will discuss the impact of different choices of $\varphi$. Table 3 depicts the attack success rates of two DL-based attribution methods on four datasets. We observe that the success rate of the targeted attack exploiting adversarial examples is 20.3% lower than that of the targeted attack exploiting coding style imitation on average. This can be attributed to the fact that adversarial examples obtained by conducting more than three code transformations are not valid attacks with respect to the notion of 3-adversary. In terms of the time complexity for generating manipulated programs, we consider DL-CAIS on GCJ-C++ dataset as an example. On average, it takes 2,417 seconds to generate an adversarial example of a program; whereas, it only takes 1.5 seconds on average to generate a manipulated program via the coding style imitation method. This large discrepancy can be attributed to the fact that the former method needs to call the attribution model to test candidate examples (possibly multiple rounds in order to generate an adversarial example); whereas, this is not needed in the latter method. For different datasets, the attack success rate of two attribution methods ranges from 9.4% to 74.6%, which are related to the number of programs in the dataset and the coding styles of different authors.

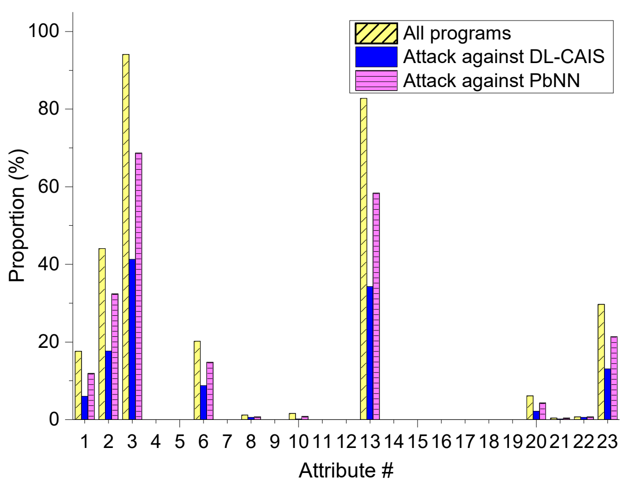


**Figure 3: Illustrating (i) the proportion of the manipulated programs in the test set involving a coding style attribute's transformation among all manipulated programs in the test set (denoted by "all programs") and (ii) the proportion of the manipulated programs that involve a coding style attribute's transformation and can attack successfully in the test set among all manipulated programs in the test set for DL-CAIS and PbNN (denoted by "attack against DL-CAIS" and "attack against PbNN" respectively).**

To see which attributes are changed when generating manipulated programs and the impact of the choice of attributes, let us consider the GCJ-Java dataset. For each coding style attribute $r$, Figure 3 illustrates (i) the proportion of the manipulated programs in the test set involving $r$'s transformation among all manipulated programs in the test set and (ii) the proportion of the manipulated programs that involve $r$'s transformation and can attack successfully in the test set among all manipulated programs in the test set for two DL-based attribution methods. We observe that most manipulated programs involve attributes #1, #2, #3, #6, #13, and #23, indicating that these coding style attributes have more significant differences among different authors than other coding style attributes. We also observe that the fraction of the manipulated programs that are successful targeted attacks against PbNN is on average 14.4% higher than that of the successful targeted attacks against DL-CAIS, where manipulations are on attributes #1, #2, #3, #6, #13, and #23. This indicates that for Java programs, the path-based representation, which is used by PbNN, can transfer the prediction from one author to another more easily than the token-based representation, which is used by DL-CAIS.

**Table 3: Attack success rates of two DL-based attribution methods, where "-" means the method cannot be used on the dataset (metrics unit: %).**

| Method | GCJ-C++ | GitHub-C | GCJ-Java | GitHub-Java |
|---|---|---|---|---|
| Targeted attacks by exploiting adversarial examples ($Asr_{tar}$) | | | | |
| DL-CAIS | 22.2 | 18.2 | - | - |
| PbNN | 9.7 | 9.4 | - | - |
| Targeted attacks by coding style imitation ($Asr_{tar}$) | | | | |
| DL-CAIS | 43.9 | 24.3 | 17.7 | 45.1 |
| PbNN | 36.8 | 18.4 | 21.0 | 74.6 |
| Untargeted attacks by exploiting adversarial examples ($Asr_{unt}$) | | | | |
| DL-CAIS | 87.7 | 15.7 | - | - |
| PbNN | 81.3 | 53.7 | - | - |
| Untargeted attacks by coding style hiding ($Asr_{unt}$) | | | | |
| DL-CAIS | 94.8 | 75.0 | 66.3 | 45.0 |
| PbNN | 95.0 | 42.7 | 60.3 | 64.5 |

**Robustness against untargeted attacks.** We apply the untargeted attack to the correctly classified test programs of authors which are randomly selected in targeted attacks. Table 3 shows the success rate of untargeted attacks for two DL-based attribution methods on four datasets. We observe that the average success rate of untargeted attacks is 36.8% higher than that of targeted attacks, which can be attributed to the fact that untargeted attacks, which misattribute program as any author other than the true author, is easier than targeted attacks, which misattribute program to the target author. To compare the effectiveness of different methods for coding style hiding attacks, we consider as the baseline a random replacement method, which transforms each coding style attribute value in the program to another random value. We choose the random replacement method because it is an intuitive way to make the manipulated program's coding style deviate more from the original author's coding style.

Table 4 summarizes the average results of random replacements five times for each DL model. Our untargeted attack method is significantly better than the random replacement method with 12.7% higher attack success rate on average. This can be explained by the fact that the random replacement method may make the manipulated programs easier to be attributed as the original author because there are some coding style attributes in the program that cannot be automatically transformed. If we do not purposely transform the program's coding style to a target author's, the manipulated program's coding style is more similar to the original author's, causing a failed untargeted attack.

**Table 4: Attack success rates of two methods for coding style hiding attacks (metrics unit: %)**

| Method | GCJ-C++ | GitHub-C | GCJ-Java | GitHub-Java |
|---|---|---|---|---|
| Our untargeted attacks | | | | |
| DL-CAIS | 94.8 | 75.0 | 66.3 | 45.0 |
| PbNN | 95.0 | 42.7 | 60.3 | 64.5 |
| Untargeted attacks by randomly replacement | | | | |
| DL-CAIS | 77.9 | 41.4 | 45.7 | 42.3 |
| PbNN | 84.0 | 38.0 | 57.1 | 55.7 |

**Table 5: Attack success rates of DL-CAIS method for different $\varphi$-adversaries on the GCJ-C++ dataset (metrics unit: %)**

| Attack type | $\varphi = 1$ | $\varphi = 3$ | $\varphi = 5$ |
|---|---|---|---|
| Targeted attack | 5.8 | 22.2 | 38.8 |
| Untargeted attack | 45.2 | 87.7 | 90.5 |

**Table 6: Accuracies of RoPGen-enabled attribution methods on 4 datasets (metrics unit: %)**

| Method | GCJ-C++ | GitHub-C | GCJ-Java | GitHub-Java |
|---|---|---|---|---|
| DL-CAIS | 92.1 | 84.9 | 98.5 | 90.0 |
| PbNN | 67.6 | 79.7 | 83.6 | 86.1 |

To show the impact of $\varphi$ (in $\varphi$-adversary) when generating adversarial examples, we consider DL-CAIS [1] on the GCJ-C++ dataset, while noting that a similar phenomenon is observed for the other DL models. Table 5 summarizes the attack success rates of DL-CAIS with $\varphi = 1, 3, 5$. We observe that when increasing $\varphi$ from 1 to 5, the attack success rate increases from 5.8% to 38.8% for the targeted attack and from 45.2% to 90.5% for the untargeted attack. This indicates that applying more code transformations can increase the success of imitating or hiding coding styles.

INSIGHT 1. *Existing DL-based attribution models are far from robust against the known and new attacks; the success rate of the untargeted attack is much higher than that of the targeted attack because the attacker has more options in the former case.*

## 5.3 Robustness of RoPGen (RQ2)

To evaluate the effectiveness of RoPGen-enabled authorship attribution methods against known and new attacks, we train eight RoPGen-enabled models involving two DL-based methods on four datasets. We choose the hyperparameters leading to the best accuracy. Take RoPGen-enabled DL-CAIS on the GCJ-C++ dataset as an example. The main hyperparameters are: the batch size is 128, the learning rate is 0.0001, the number of recurrent neural network layers is 3, the width lower bound $\alpha$ is 0.8, and the number of sub-networks is 3. We set $\varphi = 3$ for generating adversarial examples.

Table 6 shows the accuracies of eight RoPGen-enabled models. We observe that the average accuracy of the RoPGen-enabled DL-CAIS models is 2.6% higher than that of the DL-based models and the average accuracy of the RoPGen-enabled PbNN models is 6.5% lower than that of the DL-based models, indicating a strong impact of the attribution method.

Table 7 summarizes the attack success rates of RoPGen-enabled methods against attacks. Compared with DL-based attribution methods, RoPGen-enabled methods can reduce the success rates of targeted and untargeted attacks (based on exploiting adversarial examples and coding style imitation/hiding) respectively by 22.8% and 41.0% on average. This means that the RoPGen significantly improves the robustness of DL-based attribution methods against attacks, which can be attributed to the data augmentation and gradient augmentation for learning robust coding style patterns. By taking PbNN on the GCJ-C++ dataset as an example, we observe the following. For PbNN, the training phase takes 65.5 seconds; for RoPGen-enabled PbNN, the training phase takes 5,876 seconds (including 5,810.5 seconds incurred by data augmentation and gradient augmentation). This extra training cost is paid for gaining robustness, while noting that the test cost is almost the same (i.e., 0.010 vs. 0.012 seconds). Since we do not need to train models often, our method is arguably practical.

**Table 7: Attack success rates of RoPGen-enabled attribution methods (metrics unit: %)**

| Method | GCJ-C++ | GitHub-C | GCJ-Java | GitHub-Java |
|---|---|---|---|---|
| Targeted attacks by exploiting adversarial examples ($Asr_{tar}$) | | | | |
| RoPGen-enabled DL-CAIS | 19.4 | 3.7 | - | - |
| RoPGen-enabled PbNN | 5.1 | 1.8 | - | - |
| Targeted attacks by coding style imitation ($Asr_{tar}$) | | | | |
| RoPGen-enabled DL-CAIS | 3.4 | 1.3 | 0.7 | 0.3 |
| RoPGen-enabled PbNN | 6.3 | 7.2 | 0.6 | 18.0 |
| Untargeted attacks by exploiting adversarial examples ($Asr_{unt}$) | | | | |
| RoPGen-enabled DL-CAIS | 58.3 | 9.0 | - | - |
| RoPGen-enabled PbNN | 60.0 | 23.5 | - | - |
| Untargeted attacks by coding style hiding ($Asr_{unt}$) | | | | |
| RoPGen-enabled DL-CAIS | 15.0 | 12.4 | 10.9 | 4.2 |
| RoPGen-enabled PbNN | 35.0 | 11.6 | 25.0 | 25.7 |

To study the contribution of data augmentation and gradient augmentation to the effectiveness respectively, we conduct the *ablation study* to investigate their effects, including three methods. The *first* method is that we exclude extending the training set by coding style imitation (denoted by "-CI"), namely the set $P$ of training programs is directly input to the full-fledged network of Step 3. The *second* method is that we exclude the gradient augmentation (denoted by "-GA"), namely the extended training set $U$ obtained from Step 1 and the set $U'$ of manipulated programs generated from Step 2 together are input to the deep neural network. The *third* method is that we exclude both coding style perturbation and gradient augmentation from RoPGen (denoted by "-CP-GA"), namely the extended training set $U$ obtained from Step 1 is input to the deep neural network.

Table 8 presents the results of applying DL-CAIS [1] to the GCJ-C++ dataset. We observe that the "-CI" method can reduce the success rate of untargeted attacks by exploiting adversarial examples, but are not very effective against targeted attacks by exploiting adversarial examples and coding style imitation and hiding attacks. The "-CP-GA" method can greatly reduce the success rate of coding style imitation and hiding attacks, but are not effective against attacks by exploiting adversarial examples. The "-GA" method can reduce the success rate of both the coding style imitation and hiding attacks and the attacks by exploiting adversarial examples, but are not as effective as RoPGen. On average, RoPGen remarkably improves the baseline with a 21.7% lower success rate of the targeted attack and a 54.6% lower success rate of the untargeted attack,

**Table 8: Ablation analysis results for DL-CAIS on the GCJ-C++ dataset (metrics unit: %)**

| Method | Adversarial examples | | Coding style imitation/hiding | |
|---|---|---|---|---|
| | $Asr_{tar}$ | $Asr_{unt}$ | $Asr_{tar}$ | $Asr_{unt}$ |
| RoPGen | 19.4 | 58.3 | 3.4 | 15.0 |
| -CI | 27.0 | 61.3 | 25.0 | 65.0 |
| -GA | 21.3 | 62.7 | 3.8 | 15.4 |
| -CP-GA | 25.7 | 80.6 | 3.2 | 15.8 |
| Baseline | 22.2 | 87.7 | 43.9 | 94.8 |

**Table 9: Attack success rates of RoPGen-enabled DL-CAIS for different $\varphi$ on the GCJ-C++ dataset (metrics unit: %)**

| Attack type | $\varphi = 1$ | $\varphi = 3$ | $\varphi = 5$ |
|---|---|---|---|
| Targeted attack | 5.7 | 19.4 | 37.7 |
| Untargeted attack | 28.3 | 58.3 | 66.6 |

owing to the incorporation of data augmentation and gradient augmentation.

We evaluate the impact of $\varphi$ in attacks exploiting adversarial examples on the effectiveness of RoPGen-enabled methods. Table 9 presents the attack success rate of RoPGen-enabled DL-CAIS on the GCJ-C++ dataset, with $\varphi = 1, 3, 5$. We observe that the attack success rate increases with $\varphi$, exhibiting a similar phenomenon to DL-CAIS; on average, the attack success rate of the RoPGen-enabled DL-CAIS method for targeted and untargeted attacks improves 1.3% and 23.4% with $\varphi$, respectively, compared with the DL-CAIS method (Table 5). This shows the effectiveness of RoPGen-enabled methods against the attacks that exploit adversarial examples.

Insight 2. *RoPGen-enabled authorship attribution methods are substantially more robust than the original DL-based methods. In particular, the success rate of targeted and untargeted attacks on RoPGen-enabled methods is respectively reduced by 22.8% and 41.0% on average.*

## 5.4 Comparing Adversarial Trainings (RQ3)

To compare the effectiveness of RoPGen-enabled attribution methods with other adversarial training methods, we consider two adversarial training methods from text/source code processing and image classification as baselines, since there have been no defense methods against code authorship attribution attacks so far. The *first* method is basic adversarial training, which is widely used in text processing and source code processing [30, 53]. The basic idea is to generate a set of adversarial examples and adding them to the training set. We test two kinds of adversarial examples. One is the adversarial examples generated by [37] (denoted by "Basic-AT-AE"); the other one is the combination of the adversarial examples generated by [37] and the programs generated by imitating the coding styles of the authors in $\mathcal{A}$ (denoted by "Basic-AT-COM"). The *second* method is PGD-AT [32], which is a widely-used baseline in image classification. It improves the adversarial robustness by solving the composition of an inner maximization problem and an outer minimization problem. When used to code authorship attribution, PGD-AT has an extremely large search space to search for the coding style transformation with the maximum loss for a program. We use the coding style transformation of a single coding style attribute instead.

**Table 10: Accuracies of DL-CAIS with 4 adversarial training methods on GCJ-C++ and GitHub-C datasets (metrics unit: %)**

| Method | GCJ-C++ | GitHub-C |
|---|---|---|
| None | 88.2 | 79.9 |
| Basic-AT-AE | 92.6 | 81.5 |
| Basic-AT-COM | 89.2 | 78.2 |
| PGD-AT | 86.2 | 76.1 |
| RoPGen | 92.1 | 84.9 |

**Table 11: Attack success rates of DL-CAIS with 4 adversarial training methods on the GCJ-C++ and GitHub-C datasets (metrics unit: %)**

| Method | GCJ-C++ | GitHub-C |
|---|---|---|
| Targeted attacks by exploiting adversarial examples ($Asr_{tar}$) | | |
| None | 22.2 | 18.2 |
| Basic-AT-AE | 20.4 | 16.5 |
| Basic-AT-COM | 25.4 | 4.2 |
| PGD-AT | 20.6 | 6.9 |
| RoPGen | 19.4 | 3.7 |
| Targeted attacks by coding style imitation ($Asr_{tar}$) | | |
| None | 43.9 | 24.3 |
| Basic-AT-AE | 45.7 | 19.9 |
| Basic-AT-COM | 5.1 | 4.2 |
| PGD-AT | 24.2 | 6.9 |
| RoPGen | 3.4 | 1.3 |
| Untargeted attacks by exploiting adversarial examples ($Asr_{unt}$) | | |
| None | 87.7 | 15.7 |
| Basic-AT-AE | 61.4 | 14.8 |
| Basic-AT-COM | 63.5 | 18.5 |
| PGD-AT | 81.7 | 15.0 |
| RoPGen | 58.3 | 9.0 |
| Untargeted attacks by coding style hiding ($Asr_{unt}$) | | |
| None | 94.8 | 75.0 |
| Basic-AT-AE | 100.0 | 72.9 |
| Basic-AT-COM | 15.8 | 27.9 |
| PGD-AT | 94.2 | 68.0 |
| RoPGen | 15.0 | 12.4 |

Table 10 shows the accuracies of DL-CAIS method with four adversarial training methods on the GCJ-C++ and GitHub-C datasets, while noting that PbNN exhibits similar phenomena. We observe that the accuracies of these adversarial training methods come close to each other, which means these methods have little effect on the accuracy. Table 11 shows the attack success rates of DL-CAIS with four adversarial training methods. For *Basic-AT-AE* and *PGD-AT* methods, the success rate of targeted and untargeted attacks by exploiting adversarial examples is averagely 4.1% and 8.5% lower than the original DL-CAIS because a number of manipulated programs with small perturbations are used to improve the model. However, the success rate of coding style imitation/hiding attacks is even a little worse than the original DL-CAIS on some datasets, which means directly extending the training set by programs with small perturbations cannot defend coding style imitation/hiding attacks. For *Basic-AT-COM* method, the success rate of coding style imitation and hiding attacks is 29.5% and 63.1% lower than the original

DL-CAIS on average. However, the success rate of attacks by exploiting adversarial examples is even a little worse than the original DL-CAIS on some datasets, which means the training set extension with the adversarial examples and the coding styles imitation of other authors cannot defend the attacks by exploiting adversarial examples. Compared with the original DL-CAIS method, RoPGen can reduce the average success rate of targeted and untargeted attacks based on exploiting adversarial examples by 8.7% and 18.1% respectively, and reduce the average success rate of targeted and untargeted attacks based on coding style imitation and hiding by 31.8% and 71.2% respectively. This attributes to the coding style imitation of other authors, the coding style perturbation, and the gradient augmentation.

INSIGHT 3. *Owing to the data augmentation and gradient augmentation, RoPGen substantially outperforms the other adversarial training methods for attacks by both exploiting adversarial examples and coding style imitation/hiding.*

## 6 LIMITATIONS

The present study has several limitations. **First**, we focus on improving the robustness of source code authorship attribution methods for a single author owing to its popularity, but the methodology can be adapted to cope with the DL-based multi-authorship attribution methods. Experiments need to be conducted for multi-authorship attribution methods. **Second**, to evaluate the effectiveness of RoPGen for DL-based attribution methods with different languages, we use two open-source and language-agnostic DL-based attribution methods for evaluation. Future studies should investigate other DL-based attribution methods for certain programming languages. **Third**, though the RoPGen framework is promising, there is much room for pursuing robust code authorship attribution. Future research should investigate other methods to find the best possible result in defending against attacks. **Fourth**, for coding style imitation/hiding attacks, we focus on automatic attack methods against code authorship attribution owing to their reproducibility. It is an interesting future work to investigate whether manual transformation is more powerful than automatic transformation, while noting (i) the manual transformation needs Institutional Review Boards (IRB) approval and (ii) the results would depend on the coding skill of programmers. **Fifth**, we do not know how to rigorously prove the soundness of various program transformations, but our empirical results provide some hints. **Sixth**, it is important to assure the adequacy of threat models.

## 7 RELATED WORK

**Prior studies on *non-adversarial* source code authorship attribution.** Prior studies on non-adversarial authorship attribution can be divided into two categories: *single-authorship* attribution [1, 2, 4, 7, 11, 11, 12, 14, 18, 21, 24, 26, 27, 36, 47, 51] vs. *multi-authorship* attribution [3, 17]. There are three approaches to non-adversarial single-authorship attribution [14]: (i) the *statistical* approach aims to identify important features for discriminant analysis [18, 26]; (ii) the *similarity* approach uses ranking methods to measure the similarity between test examples and candidate examples in the feature space [12, 21, 27]; (iii) the *machine learning* approach achieves attribution via random forests [11, 24], support vector machines [14, 36], and deep neural networks [1, 2, 4, 7, 11, 47, 51]. Whereas, multi-authorship attribution is still largely open [3, 17]. When compared with these studies, we focus on *adversarial* single-authorship attribution.

**Prior studies on *adversarial* source code authorship attribution.** There are two attacks against authorship attribution, which exploit *adversarial examples* or *coding style imitation/hiding*. The former performs functionality-preserving perturbations to a target program to cause misattribution [31, 37]. The latter can be characterized by what the attacker knows (i.e., black-box [35, 42] vs. white-box [34]) and what the attacker does (i.e., manual mimicry attacks [42] vs. semi-automatically or automatically leveraging weaknesses of an attribution method [34, 35]). The most closely related prior study is [34], which presents a white-box attack leveraging human-defined features of the code authorship attribution method. In contrast, RoPGen deals with black-box attacks which do not know or need such information. The present study is complementary, or orthogonal, to [34] because we focus on coping with black-box attacks against DL-based attribution methods; whereas, [34] cannot deal with DL-based attribution methods because automatically learned features are not human-defined or human-understandable.

**Prior studies on adversarial training.** From a technical standpoint, RoPGen leverages adversarial training [9, 33, 40]. The basic idea is to augment training data with adversarial examples, analogous to "vaccination". This approach has been extensively investigated in a number of applications, including: image processing [22, 32, 41, 49], neural language processing [30, 48, 54], malware detection [5, 15, 28, 29], and source code processing (e.g., functionality classification, method/variable name prediction, and code summarization) [10, 39, 43, 45, 52, 53]. To the best of our knowledge, RoPGen is the first robustness framework for coping with attacks against source code authorship attribution.

## 8 CONCLUSION

We presented the RoPGen framework for enhancing robustness of a range of DL-based source code authorship attribution methods. The key idea behind RoPGen is to learn coding style patterns which are hard to manipulate or imitate. This is achieved by leveraging data augmentation and gradient augmentation to train attribution models. We presented two automatic coding style imitation and hiding attacks. Experimental results show that RoPGen can substantially improve the robustness of DL-based code authorship attribution. The limitations of the present study discussed in Section 6 provide interesting problems for future research.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments, which guided us in improving the paper. This work was supported in part by the National Science Foundation under Grants #1812599, #2122631, and #2115134, Army Research Office Grant #W911NF-17-1-0566, and Colorado State Bill 18-086. Zhen Li was supported in part by the National Natural Science Foundation of China under Grant U1936211. Any opinions, findings, conclusions or recommendations expressed in this work are those of the authors and do not reflect the views of the funding agencies in any sense.

# REFERENCES


[1] Mohammed Abuhamad, Tamer AbuHmed, Aziz Mohaisen, and DaeHun Nyang. 2018. Large-Scale and Language-Oblivious Code Authorship Identification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS), Toronto, ON, Canada.* 101–114.

[2] Mohammed Abuhamad, Tamer Abuhmed, David Mohaisen, and Daehun Nyang. 2021. Large-scale and Robust Code Authorship Identification with Deep Feature Learning. *ACM Trans. Priv. Secur.* 24, 4 (2021), 1–35.

[3] Mohammed Abuhamad, Tamer AbuHmed, DaeHun Nyang, and David A. Mohaisen. 2020. Multi-$\chi$: Identifying Multiple Authors from Source Code Files. *Proc. Priv. Enhancing Technol.* 2020, 3 (2020), 25–41.

[4] Mohammed Abuhamad, Ji-su Rhim, Tamer AbuHmed, Sana Ullah, Sanggil Kang, and DaeHun Nyang. 2019. Code Authorship Identification Using Convolutional Neural Networks. *Future Gener. Comput. Syst.* 95 (2019), 104–115.

[5] Abdullah Al-Dujaili, Alex Huang, Erik Hemberg, and Una-May O'Reilly. 2018. Adversarial Deep Learning for Robust Detection of Binary Encoded Malware. In *Proceedings of 2018 IEEE Security and Privacy Workshops, San Francisco, CA, USA.* 76–82.

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, POPL (2019), 40:1–40:29.

[7] Bander Alsulami, Edwin Dauber, Richard E. Harang, Spiros Mancoridis, and Rachel Greenstadt. 2017. Source Code Authorship Attribution Using Long Short-Term Memory Based Networks. In *Proceedings of the 22nd European Symposium on Research in Computer Security (ESORICS), Oslo, Norway.* 65–82.

[8] John Anvik, Lyndon Hiew, and Gail C. Murphy. 2006. Who Should Fix This Bug?. In *Proceedings of the 28th International Conference on Software Engineering (ICSE), Shanghai, China.* 361–370.

[9] Tao Bai, Jinqi Luo, Jun Zhao, Bihan Wen, and Qian Wang. 2021. Recent Advances in Adversarial Training for Adversarial Robustness. *CoRR* abs/2102.01356 (2021).

[10] Pavol Bielik and Martin T. Vechev. 2020. Adversarial Robustness for Code. In *Proceedings of the 37th International Conference on Machine Learning (ICML), Virtual Event.* 896–907.

[11] Egor Bogomolov, Vladimir Kovalenko, Yurii Rebryk, Alberto Bacchelli, and Timofey Bryksin. 2021. Authorship Attribution of Source Code: A Language-Agnostic Approach and Applicability in Software Engineering. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Athens, Greece.* 932–944.

[12] Steven Burrows and Seyed MM Tahaghoghi. 2007. Source Code Authorship Attribution Using n-grams. In *Proceedings of the 12th Australasian Document Computing Symposium, Melbourne, Australia, RMIT University.* Citeseer, 32–39.

[13] Steven Burrows, Seyed M. M. Tahaghoghi, and Justin Zobel. 2007. Efficient Plagiarism Detection for Large Code Repositories. *Softw. Pract. Exp.* 37, 2 (2007), 151–175.

[14] Steven Burrows, Alexandra L. Uitdenbogerd, and Andrew Turpin. 2014. Comparing Techniques for Authorship Attribution of Source Code. *Softw. Pract. Exp.* 44, 1 (2014), 1–32.

[15] Yizheng Chen, Shiqi Wang, Dongdong She, and Suman Jana. 2020. On Training Robust PDF Malware Classifiers. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security).* 2343–2360.

[16] Code Beautify 2020. https://codebeautify.org/c-formatter-beautifier.

[17] Edwin Dauber, Aylin Caliskan, Richard E. Harang, Gregory Shearer, Michael Weisman, Frederica Free-Nelson, and Rachel Greenstadt. 2019. Git Blame Who?: Stylistic Authorship Attribution of Small, Incomplete Source Code Fragments. *Proc. Priv. Enhancing Technol.* 2019, 3 (2019), 389–408.

[18] Haibiao Ding and Mansur H. Samadzadeh. 2004. Extraction of Java program fingerprints for software authorship identification. *J. Syst. Softw.* 72, 1 (2004), 49–57.

[19] Yinpeng Dong, Qi-An Fu, Xiao Yang, Tianyu Pang, Hang Su, Zihao Xiao, and Jun Zhu. 2020. Benchmarking Adversarial Robustness on Image Classification. In *Proceedings of the 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Seattle, WA, USA.* 318–328.

[20] EditorConfig 2020. https://editorconfig.org/.

[21] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, and Sokratis K. Katsikas. 2006. Effective Identification of Source Code Authors Using Byte-level Information. In *Proceedings of the 28th International Conference on Software Engineering (ICSE), Shanghai, China.* 893–896.

[22] Xiang Gao, Ripon K. Saha, Mukul R. Prasad, and Abhik Roychoudhury. 2020. Fuzz Testing Based Data Augmentation to Improve Robustness of Deep Neural Networks. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE), Seoul, South Korea.* 1147–1158.

[23] Google Code Jam 2020. https://codingcompetitions.withgoogle.com/codejam.

[24] Aylin Caliskan Islam, Richard E. Harang, Andrew Liu, Arvind Narayanan, Clare R. Voss, Fabian Yamaguchi, and Rachel Greenstadt. 2015. De-anonymizing Programmers via Code Stylometry. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security), Washington, D.C., USA.* 255–270.

[25] Vaibhavi Kalgutkar, Ratinder Kaur, Hugo Gonzalez, Natalia Stakhanova, and Alina Matyukhina. 2019. Code Authorship Attribution: Methods and Challenges. *ACM Comput. Surv.* 52, 1 (2019), 3:1–3:36.

[26] Ivan Krsul and Eugene H. Spafford. 1997. Authorship Analysis: Identifying the Author of a Program. *Comput. Secur.* 16, 3 (1997), 233–257.

[27] Robert Charles Lange and Spiros Mancoridis. 2007. Using Code Metric Histograms and Genetic Algorithms to Perform Author Identification for Software Forensics. In *Proceedings of Genetic and Evolutionary Computation Conference (GECCO), London, England, UK.* 2082–2089.

[28] Deqiang Li, Qianmu Li, Yanfang Ye, and Shouhuai Xu. 2020. SoK: Arms Race in Adversarial Malware Detection. *CoRR* abs/2005.11671 (2020). arXiv:2005.11671 https://arxiv.org/abs/2005.11671

[29] Deqiang Li, Qianmu Li, Yanfang Ye, and Shouhuai Xu. 2021. A Framework for Enhancing Deep Neural Networks Against Adversarial Malware. *IEEE Trans. Netw. Sci. Eng.* 8, 1 (2021), 736–750. https://doi.org/10.1109/TNSE.2021.3051354

[30] Jinfeng Li, Shouling Ji, Tianyu Du, Bo Li, and Ting Wang. 2019. TextBugger: Generating Adversarial Text Against Real-world Applications. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA.*

[31] Qianjun Liu, Shouling Ji, Changchang Liu, and Chunming Wu. 2021. A Practical Black-box Attack on Source Code Authorship Identification Classifiers. *IEEE Trans. Inf. Forensics Secur.* 16 (2021), 3620–3633.

[32] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards Deep Learning Models Resistant to Adversarial Attacks. In *Proceedings of the 6th International Conference on Learning Representations (ICLR), Vancouver, BC, Canada.*

[33] Pratyush Maini, Eric Wong, and J. Zico Kolter. 2020. Adversarial Robustness against the Union of Multiple Perturbation Models. In *Proceedings of the 37th International Conference on Machine Learning (ICML), Virtual Event.* 6640–6650.

[34] Alina Matyukhina, Natalia Stakhanova, Mila Dalla Preda, and Celine Perley. 2019. Adversarial Authorship Attribution in Open-Source Projects. In *Proceedings of the 9th ACM Conference on Data and Application Security and Privacy (CODASPY), Richardson, TX, USA.* 291–302.

[35] Christopher McKnight and Ian Goldberg. 2018. Style Counsel: Seeing the (Random) Forest for the Trees in Adversarial Code Stylometry. In *Proceedings of the 2018 Workshop on Privacy in the Electronic Society (WPES@CCS), Toronto, ON, Canada.* 138–142.

[36] Brian N Pellin. 2000. Using Classification Techniques to Determine Source Code Authorship. *White Paper: Department of Computer Science, University of Wisconsin* (2000).

[37] Erwin Quiring, Alwin Maier, and Konrad Rieck. 2019. Misleading Authorship Attribution of Source Code using Adversarial Learning. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security), Santa Clara, CA, USA.* 479–496.

[38] Foyzur Rahman and Premkumar T. Devanbu. 2011. Ownership, Experience and Defects: A Fine-grained Study of Authorship. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE), Waikiki, Honolulu , HI, USA.* 491–500.

[39] Goutham Ramakrishnan, Jordan Henkel, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas W. Reps. 2020. Semantic Robustness of Models of Source Code. *CoRR* abs/2002.03043 (2020).

[40] Lukas Schott, Jonas Rauber, Matthias Bethge, and Wieland Brendel. 2019. Towards the First Adversarially Robust Neural Network Model on MNIST. In *Proceedings of the 7th International Conference on Learning Representations (ICLR), New Orleans, LA, USA.*

[41] Ali Shafahi, Mahyar Najibi, Amin Ghiasi, Zheng Xu, John P. Dickerson, Christoph Studer, Larry S. Davis, Gavin Taylor, and Tom Goldstein. 2019. Adversarial Training for Free!. In *Proceedings of Annual Conference on Neural Information Processing Systems (NeurIPS), Vancouver, BC, Canada.* 3353–3364.

[42] Lucy Simko, Luke Zettlemoyer, and Tadayoshi Kohno. 2018. Recognizing and Imitating Programmer Style: Adversaries in Program Authorship Attribution. *Proc. Priv. Enhancing Technol.* 2018, 1 (2018), 127–144.

[43] Jacob M. Springer, Bryn Marie Reinstadler, and Una-May O'Reilly. 2020. STRATA: Building Robustness with a Simple Method for Generating Black-box Adversarial Attacks for Models of Code. *CoRR* abs/2009.13562 (2020).

[44] srcML 2020. https://www.srcml.org/.

[45] Shashank Srikant, Sijia Liu, Tamara Mitrovska, Shiyu Chang, Quanfu Fan, Gaoyuan Zhang, and Una-May O'Reilly. 2021. Generating Adversarial Computer Programs Using Optimized Obfuscations. In *Proceedings of the 9th International Conference on Learning Representations (ICLR), Virtual Event, Austria.*

[46] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. 2016. Revisiting Code Ownership and Its Relationship with Software Quality in the Scope of Modern Code Review. In *Proceedings of the 38th International Conference on Software Engineering (ICSE), Austin, TX, USA.* 1039–1050.

[47] Farhan Ullah, Junfeng Wang, Sohail Jabbar, Fadi Al-Turjman, and Mamoun Alazab. 2019. Source Code Authorship Attribution Using Hybrid Approach of Program Dependence Graph and Deep Learning Model. *IEEE Access* 7 (2019), 141987–141999.

[48] Wenqi Wang, Lina Wang, Run Wang, Zhibo Wang, and Aoshuang Ye. 2019. Towards a Robust Deep Neural Network in Texts: A Survey. *CoRR* abs/1902.07285 (2019).
[49] Eric Wong, Leslie Rice, and J. Zico Kolter. 2020. Fast is Better than Free: Revisiting Adversarial Training. In *Proceedings of the 8th International Conference on Learning Representations (ICLR), Addis Ababa, Ethiopia*.
[50] Taojiannan Yang, Sijie Zhu, and Chen Chen. 2020. GradAug: A New Regularization Method for Deep Neural Networks. In *Proceedings of Annual Conference on Neural Information Processing Systems (NeurIPS), virtual*.
[51] Xinyu Yang, Guoai Xu, Qi Li, Yanhui Guo, and Miao Zhang. 2017. Authorship Attribution of Source Code by Using Back Propagation Neural Network Based on Particle Swarm Optimization. *PloS one* 12, 11 (2017), e0187204.
[52] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial Examples for Models of Code. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 162:1–162:30.
[53] Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. 2020. Generating Adversarial Examples for Holding Robustness of Source Code Processing Models. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI), New York, NY, USA*. 1169–1176.
[54] Yuhao Zhang, Aws Albarghouthi, and Loris D'Antoni. 2020. Robustness to Programmable String Transformations via Augmented Abstract Training. In *Proceedings of the 37th International Conference on Machine Learning (ICML), Virtual Event*. 11023–11032.