A Practical Black-box Attack on Source Code Authorship Identification Classifiers

Qianjun Liu, Shouling Ji, *Member, IEEE,* Changchang Liu, Chunming Wu

***Abstract*—Existing researches have recently shown that adversarial stylometry of source code can confuse source code authorship identification (SCAI) models, which may threaten the security of related applications such as programmer attribution, software forensics, etc. In this work, we propose source code authorship disguise (SCAD) to automatically hide programmers' identities from authorship identification, which is more practical than the previous work [1] that requires to known the output probabilities or internal details of the target SCAI model. Specifically, SCAD trains a substitute model and develops a set of semantically equivalent transformations, based on which the original code is modified towards a disguised style with small manipulations in lexical features and syntactic features. When evaluated under totally black-box settings, on a real-world dataset consisting of 1,600 programmers, SCAD induces state-of-the-art SCAI models to cause above 30% misclassification rates. The efficiency and utility-preserving properties of SCAD are also demonstrated with multiple metrics. Furthermore, our work can serve as a guideline for developing more robust identification methods in the future.**

***Index Terms*—source code, authorship identification, adversarial stylometry.**

I. Introduction

PROGRAMMERS share programs and contribute to projects through various channels. Even if the source code is published anonymously, its authorship can still be revealed by the stylistic features exhibited naturally in the code [2–9]. Recent efforts on source code authorship identification (SCAI) can achieve over 90% accuracy in distinguishing thousands of programmers [2, 3], which benefit realistic applications such as programmer attribution, plagiarism detection, software forensics and copyright dispute investigation. However, these identification systems have been shown to be vulnerable to intentionally modified coding styles [1, 10, 11]. It is worthy noting that generating adversarial examples in source code is even more difficult than that in the domain of images and texts. For images, adversarial perturbations can be achieved by directly changing the input features (e.g., increasing the density of pixels [12]). For texts, adversarial examples are generated by changing characters, words or phrases of the original text [13, 14]. However, it is difficult to directly apply existing adversarial perturbation methods in texts/images to perturb source code due to the following two reasons. Firstly, different from the domain of images, there is no map between the feature space and the instance space of source code, i.e., we cannot create a code sample from a given feature representation. Secondly, different from the domain of texts, we cannot directly replace, insert, or modify words in source code which may cause error or malfunctions. The only work previous to ours, in automatically generating adversarial coding style is [1], which, however, relies on strong assumptions that the classifier outputs the probabilities along with the labels or the model internals (structure, feature representation, etc.) is priorly known.

To address challenges of adversarially transforming source code and overcome the weaknesses of previous works, we propose a practical and automatic system for source code authorship disguise (SCAD). SCAD works under a totally black-box setting without any prior knowledge of the identification models. In SCAD, we first train a substitute model for the unknown SCAI classifier. We then adopt a set of equivalent transformation rules that do not change the functionality of the original code. To select the most effective transformation at each step, we use a customized Jacobian-based saliency map approach (JSMA) to measure the influence of each transformation rule based on the substitute classifier. Finally, the transformed source code is verified on the SCAI classifier to check if a false classification occurs. We evaluate SCAD on a large-scale C++ code dataset against the most state-of-the-art classifiers. The misclassification rate of the original identification system under disguised code is over 30%. Further, the performance of SCAD is comparable with the previous work [1] when the output probabilities or model internals are accessible.

Contributions. To summarize, our work has made the following contributions:

(1) **A practical method to evade source code authorship identification.** We propose SCAD, a practical approach that can automatically generate source code with adversarial stylistic patterns to evade authorship identification systems. Our method does not rely on the knowledge of the output probabilities or model internals of the identification systems as previous methods did.

(2) **A substitute NARN model and a set of well-designed transformation rules.** We extract path features from abstract syntax trees and train the neural attentional regression network (NARN) upon path-based representations, which can capture

Shouling Ji and Chunming Wu are the corresponding authors.

Qianjun Liu is with the College of Computer Science & Technology at Zhejiang University, Hangzhou, Zhejiang 310027, China. (E-mail: liuqj0522@zju.edu.cn)

Shouling Ji is with the College of Computer Science and Technology, Zhejiang University, and the Zhejiang University NGICS Platform, Hangzhou, Zhejiang 310027, China. (E-mail: sji@zju.edu.cn)

Changchang Liu is with the Department of Distributed AI, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA. (E-mail: changchang.liu33@ibm.com)

Chunming Wu is with the College of Computer Science & Technology at Zhejiang University, Hangzhou, Zhejiang 310027, China. (E-mail: wuchunming@zju.edu.cn)

subtle stylistic patterns that are important to authorship identification. To manipulate composite stylistic features such as identifiers, data types, control flows and functions, we design 37 transformation rules to flexibly transform source code, without changing the functionality of the original code.

(3) **Large-scale real world evaluation.** We use a code corpus collected from 1,600 programmers, which, to our best knowledge, is the largest dataset for evaluating authorship disguise. Extensive experiments demonstrate the effectiveness of SCAD in evading the state-of-the-art authorship identification methods. We also show that SCAD is robust to models that are retrained based on adversarial code. We further measure the efficiency of SCAD and the utility of transformed code, to show its feasibility in real world applications.

II. Background and Related Work

De-anonymizing programmers through code stylometry. Programmer de-anonymization is also termed as code authorship identification, authorship attribution or code author profiling [9]. Generally, those works used different stylistic features to represent programs, and then learned a classification model based on the feature representations. Several early works used a limited number of features or similarity-based metrics, and got low accuracy even on tens of programmers [4–8]. Frantzeskou et al. used character-level n-grams to capture indentation, variable naming and other programming preference [9]. Their method can achieve 96.9% accuracy when identifying 30 programmers, which, however, degrade seriously when applying to large datasets with tens of thousands of n-gram strings. The most state-of-the-art works are [2] and [3], both of which achieved over 90% accuracy in identifying thousands of programmers. Caliskan-Islam et al. developed code stylometry feature set (CSFS), which includes layout features, lexical features and syntactic features [2]. They utilized a Random Forest Classifier (RFC) to attribute source code and got 92.83% accuracy on the code samples belonging to 1,600 programmers. Abuhamad et al. extracted term frequency–inverse document frequency (TF-IDF) values of word-level n-grams from source code [3]. The classification was finally done by a random forest classifier attached to recurrent neural network (RNN) layers. The RNN-RFC model achieved 92.3% accuracy on a set of 8,903 programmers, the largest scale of programmer de-anonymization by far.

Compared with source code, binary code preserves not only the programmer's style, but also the "fingerprints" remained by the operating systems, the compilers and other options in the intermediate tool chains. Although there have been several works [15, 16] targeted at de-anonymizing binaries, our work focuses on source code de-anonymization for two reasons: ① there exist certain scenarios where authorship identification must be implemented at the source code level (e.g., student assignments, open source software, incomplete code fragments, etc.); ② for existing works on de-anonymizing binary code, the features are still extracted from the disassembled and decompiled code [15]. Therefore, our study in adversarial source code would even benefit works in adversarial binary code, which may be another direction for protecting anonymity.

Adversarial examples for images. Adversarial examples are first proposed in the computer vision domain [17–19], for which many approaches have been designed to craft adversarial images, including Limited memory Broyden-FletcherGoldfarb-Shanno (L-BFGS) [17], Fast Gradient Sign Method (FGSM) [18], Jacobian-based Saliency Map Approach (JSMA) [12], C&W attack [19], etc. The objective of these works is to inject a minimum amount of perturbation (under various limitations of the adversary's capability) to an image so that its predicted result can be misled. However, they cannot be directly applied to source code because image inputs are continuous (e.g., pixel values) while code are discrete (e.g., tokens, statements). In addition, small changes in pixels will not change the semantics of images while perturbation on code are more perceivable.

Adversarial stylometry for natural language. An empirical study [20] showed natural language processing (NLP)-based methodologies were vulnerable to manually disguised writing style, which raised research interests of adversarial stylometry. Anonymouth [21] and Unstyle [22] are authorship circumventing tools which help users anonymize their documents. Both tools advise users which features are required to change and interact with users until the identification confidence is reduced to a low value. More advanced methods attempted to automatically transfer writing style by simply replacing, deleting or inserting words in the original text [23]. However, those methods are not suitable for transforming code since direct word replacement/insertion/deletion may cause syntax errors and semantic inconsistency.

Adversarial stylometry for source code. Authorship confusion or authorship disguise is far more difficult for code, since code is a kind of structured text. There were only a few works in this domain, which can be classified into two categories: the manual approaches and the automatic approaches. **(1) Manual approaches.** Recently, an experiment recruited 28 experienced programmers to conduct authorship forgery experiments, in which 76.6% of the participants were able to confuse the random forest classifier trained on 20 programmers' code [10]. Another work analyzed the split points in random forest classifiers to aid programmers in adjusting their inherent styles [11]. **(2) Automatic approaches.** To the best of our knowledge, Quiring et al. proposed so far the only work in automatically misleading source code authorship identification classifiers, where they designed 36 transformation rules and used Monte-Carlo tree search to guide the transformations [1]. Strictly speaking, their approach worked in gray-box settings. They achieved 99% misclassification rate on 204 programmers when the prediction scores (i.e., the probabilities of being classified as a certain class) of the original model was accessible, and 52% misclassification rate when the internal details (model structure, feature representation methods, etc.) of the original model was known. However, in realistic applications such as online deep learning API, the classifiers more often outputs classification labels than prediction scores and adversaries do not know the model's inner design. **Compared with [1], we have the following two advantages**: ① practical attack. Our approach does not require prior knowledge of the prediction scores or internal details of the original model, which is a

significant characteristic of fully black-box attacks. ② larger-scale and thoughtful evaluation. Our main experiment tests attack on 1,600 programmers on 6 different SCAI classifiers, while [1] only evaluated on 204 programmers on 2 SCAI classifiers. Besides, SCAD are evaluated under different overlap ratios between the attacker's and the target classifier's training sets. We also show the robustness of SCAD against classifiers that are re-trained on adversarial code. Based on the above advantages, our method is more realistic for disguising source code in practice.

III. Threat Model and Overview of SCAD

A. Threat Model

The possibility of evading source code authorship attribution brings several important security issues for a range of forensic applications, particularly when the source code can be automatically transformed by an attacker's tool. For example, a ghostwriter can use an automatic tool to transform his/her code into different styles to bypass plagiarism detection; developers can copy open source projects and generate a bunch of pirated softwares for profits. There are two roles in the process of disguising code: the SCAI classifier (or the detector) and the programmer (or the attacker). We explain the capacities and goals of each role as below.

The SCAI classifier. The SCAI classifier is trained on previous code samples collected from all the candidate programmers. We denote the attacker's classifier as C and its classification result for s as $C(s)$. **The programmer.** For some purposes, a programmer wants to distribute some code anonymously without being identified. So the programmer needs to transform the original code to get rid of his/her personal coding style. Meanwhile, the transformed code should have the same functions as the original code in order to be correctly used. We denote a source code sample as s and the adversarially transformed code as s'. The authorship disguise is successful if: $C(s') \neq C(s)$.

In the design of our attack, there are two challenges for the adversary:

• **Challenge 1**: The attacker works in a fully black-box setting, i.e., he/she has no prior knowledge of the SCAI classifer C (e.g., the features used, the structure/parameters of C, the prediction scores of C for a given sample under certain classes).

• **Challenge 2**: How to transform s into s' with small perturbations so that equivalent functionality is maintained?

B. Overview of SCAD

To solve the two challenges above, we propose a practical method to evade SCAI classifiers, named as SCAD. As shown in Figure 1, SCAD consists of four components: the substitute classifier, the transformation decision algorithm, the code transformation module and authorship disguise verification. Below, we show how the four components connect and work together in the process of source code authorship disguise.

Step (1): substitute classifier training. The attacker trains its own classifier C_a, as a substitute of the SCAI classifier C. C_a learns a classification function $\mathcal{F} : \mathcal{X} \rightarrow \mathcal{R}^{|\mathcal{Y}|}$, which outputs the probability of an input belonging to each author. In our model, for any code instance s, its feature representation $\boldsymbol{x}$ and the originally classified author $C(s)$, $\mathcal{F}$ is trained to have the same classification results as the original SCAI classifier: $\arg\max_{y \in \mathcal{Y}}(\mathcal{F}_y(x)) = C(s)$.

Step (2): transformation decision. The transformation decision algorithm iteratively selects the most effective transformation rule for the current code based on the output of the substitute model in step (1). Specifically, it measures the influence of a transformation rule by calculating the JSMA-based saliency scores of the changed features that contribute to the output probability of a target author, i.e., $\mathcal{F}_{y_t}(x)$.

Step (3): code transformation. The code transformation module implements a set of source-to-source transformation rules while maintaining semantic equivalence. The original code s is transformed into the adversarial code s' by applying the transformation rule decided in Step (2).

Step (4): authorship disguise verification. The transformed code s' is finally passed to the original SCAI classifier to check whether the attack succeeds, that is, $C(s') \neq C(s)$. If it fails, we repeat Step (2) and Step (3) until a successful evasion happens or the pre-defined number of iterations is reached.

IV. SCAD: Source Code Authorship Disguise

A. Substitute Model Training

Training a substitute model is motivated by the transferability of adversarial examples [24], i.e., the code instances that can mislead the substitute model C_a may also affect the original classifier C even if the two models have different architectures or use different training sets. We design a three-layer model which tailors the feature extraction and feature selection methods in previous works to learn different style patterns. Our model follows a general machine learning pipeline:

(1) Feature extraction. Caliskan-Islam et al. used manually defined lexical features (e.g., $numFunctions/length$) and simple syntactic features (e.g., ASTNodes, ASTBigrams) [2]. They split lexical features and syntactic features, which might be improper since the two features are closely related in source code. Abuhamad et al. directly segmented code to extract lexical token features [3]. Their features are discriminating but not comprehensive because a variable name is different from a function name. We extract **tokenized path** from AST, which links lexical tokens to its syntactic path thus expressing in a subtle way. A fragment of the entire AST together with the corresponding code is shown in Figure 2. The nodes in the tree represent elements of a programming language. The edges in the tree connect child nodes to parent nodes. We refer to the gray padded nodes in the AST as **tokenized node**, which usually contain tokens of identities, literals, operators or some built-in names such as int. The nodes which do not have tokens in usually stand for language constructs such as *FunctionCallExpression* and *SimpleDeclaration*. A tokenized path starts from a tokenized node, traverses from bottom to up until it reaches another token (Path 1 in Figure 2) or arrives at the root node (Path 2 in Figure 2). The path-based feature is similar to node n-grams but more expressive because

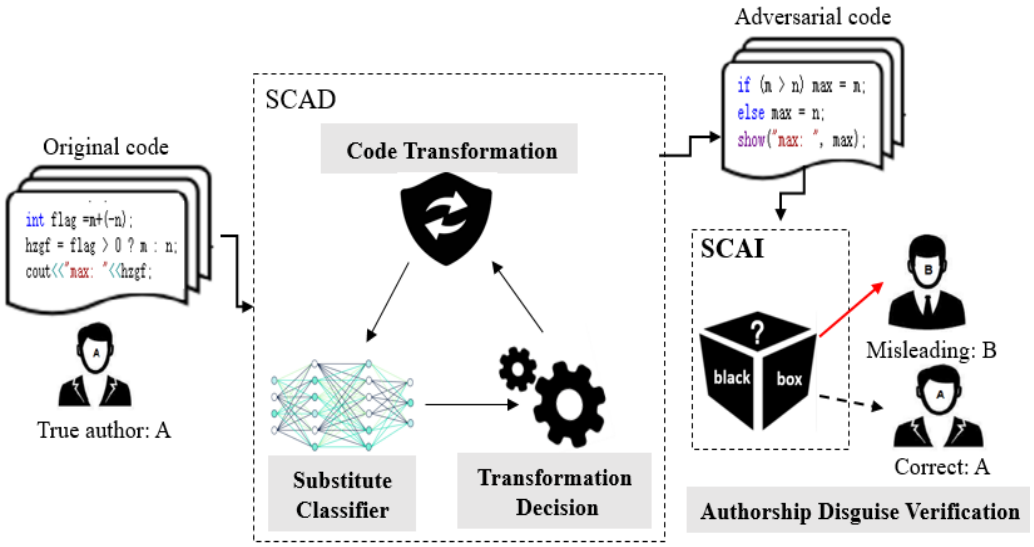


Fig. 1. The architecture of SCAD.

it combines a lexical token with its corresponding syntax constructs. For example, the "*int*" token in "*int main()*" is different from that in "*int div=gcd(18,66)*". We then represent the source code as a bag of AST paths $\mathcal{B} = \{< p_1, x_1 >, < p_2, x_2 >, ..., < p_k, x_k >, ...\}$, where each p_k is a unique path and x_k represents its occurrences (an int value) in the AST. We do not consider layout features such as tabs, spaces, brackets, braces, etc., since these features are less informative and we can uniformly format all the code in data preprocessing (see Section V-A).

(2) Feature selection. As there exist a wide variety of paths in the corpus, our bag-of-paths representation exhibits a large and sparse feature space. In order to improve efficiency, we need select the most important and discriminating features. Previous works used information gain (IG) [2] or term frequencies (TF) [3] to reduce feature dimensions. Unfortunately, using the two metrics alone may filter out valuable features due to the sparsity (some infrequent features may have low values of IG and TF). Thus, we design a novel feature selection metric through an effective combination of IG, TF and inverse document frequency (IDF) of the paths:

$$score(p_i) = \frac{TF(p_i) \times IDF(p_i) \times IG(p_i)}{\sqrt{\sum_{j=1}^{n} TF(p_j)^2 \times IDF(p_j)^2 \times IG(p_j)^2}} \quad (1)$$

where N is the number of all the paths, $TF(p_i)$ is the frequency of path p_i in the whole corpus, and $IDF(p_i)$ is the inverse document frequency[1] of p_i. The IG value of p_i is calculated as: $IG(p_i) = H(Y) - H(Y|p_i)$, where $H(\cdot)$ denotes the entropy and it evaluates the difference of information contained in Y with and without observing the occurrence ofp_i. We sort all the paths with their scores and leave the top n paths as the final feature set. Compared to IG, our metric is able to eliminate the sparsity: there are about 60% of the features with a score higher than 10^{-4} using our metric while only 4% of the features using IG. Moreover, when setting n to 10,000, the paths selected by our method cover 58 kinds of AST nodes while the paths selected by TF metric only cover 38 kinds (more details will be discussed in Section V-A).

[1]https://en.wikipedia.org/wiki/Tf–idf#Inverse_document_frequency

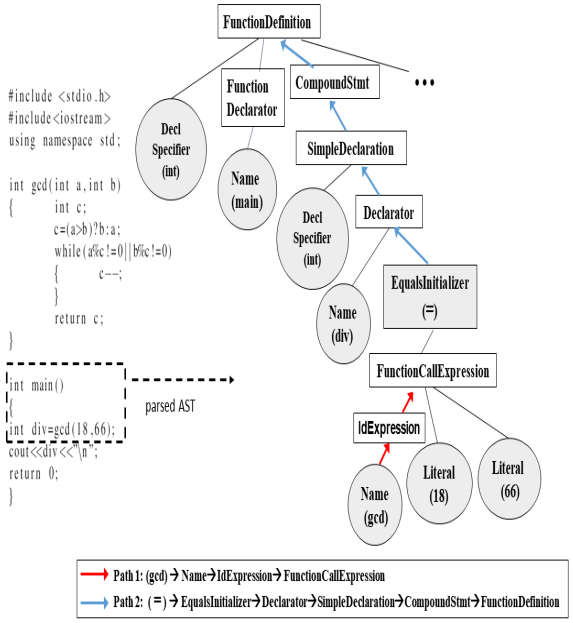


Fig. 2. An example of AST and tokenized path.

(3) Model training. After feature selection, the input vector for each code sample is represented as $\boldsymbol{x} = [x_1, x_2, ..., x_n]^{\mathsf{T}}$, where n is the number of selected path features and each element x_i ($x_i \geq 0, i \in [1, n]$) is an int value which represents the occurrences of the i^{th} path. To learn the programmers' preference on different tokenized paths, we design a neural attentional regression network (NARN) with three layers: the fully connected layer, the attention layer and the regression layer.

The attention layer We employ the main idea of a location-

TABLE I
TRIVIAL TRANSFORMATIONS

ID	Rule Name	Description and Examples
1	Remove unused code	Removes variables, functions and included libraries that are never used or called.
2	Split/aggregate declarations	Splits a one-line declaration into multiple lines or otherwise aggregates multiple declarations into one line. e.g., *int a,b → int a; int b;*
3	Separate/attach elaborated type declaration	Splits an composite type declaration into an elaborated type declaration or otherwise attaches an elaborated type declaration to its definition. e.g., *struct mynode {int x,y;}snode; → struct mynode {int x,y;}; struct mynode snode;*
4	Replace identifiers	Changes variable names or function names into another one across the whole code instance.
5	Undo type alias	Replaces the declared type alias with the original type name. e.g., *typedef long long LL; LL c=a+b; → long long c=a+b;*
6	Use alternative tokens	Replaces operators or symbols with their alternative tokens. e.g, (&& → and), (!= → not_eq). There are 26 such pairs of alternative tokens.
7	Swap operands	Swaps the operands of relational operators. e.g, $a < b \rightarrow b > a$. This rule is allowed for 8 operators, including ">, <,≥,≤, ==, ! =, &&, ∥".
8	Use converse-negative expressions	Rewrites condition statements into converse-negative expressions using inverse operators. e.g., $a < b \rightarrow !\ (a \geq b)$, vice versa. We support 4 pairs of inverse operators: (>→ ≤), (<→ ≥), (==→! =), (&& → ∥).
9	Use equivalent computations	Uses equivalent mathematical expressions. e.g., $a = b + c \rightarrow a = b - (-c)$. We implement equivalent computations for 20 arithmetic operators, assigning operators and relational operators.

TABLE II
DATA TRANSFORMATIONS

ID	Rule Name	Description and Examples
10	Use typeid expression	Uses the keyword "*typeid*" to dynamically decide data types. e.g., *int b=0 → typeid (a) b=0*, where *a* is an already defined int variable.
11	Use cast expressions	Although cast expressions are normally used to explicitly convert a data value to a new type, we can utilize the property to transform the same types like *int b=(int) a*, where *a* is an int variable.
12	Convert int literals into expressions	Gets an int literal by using mathematical expressions such as addition, subtraction, multiplication and division. e.g., *int b=8; → int b=2*4;*
13	Convert integers into hexadecimal numbers	e.g., *int b=48; → int b=0x30*;
14	Convert char literals into ASCII values	e.g., *char c='A'; → char c=65;*
15	Convert string literals to char arrays	Uses a char array to initialize a string variable.
16	Convert between bool literals and int literals	E.g., uses "1" for "true" and "0" for "false".

based global attention mechanism [25], which puts different weights on different steps in the source and target hidden states. To simplify, we put the attention weights directly on the input sequence. Each x_i is influenced by the input $\boldsymbol{x}$ and the global attention vector $\boldsymbol{W}$, which is calculated as a softmax function:

$$a_i = \frac{\exp\left(W_i \cdot x_i\right)}{\sum_{j=1}^{n} \exp\left(W_j \cdot x_j\right)} \tag{2}$$

Then all the elements are aggregated into a context vector $\boldsymbol{v}$ ($\boldsymbol{v} \in \mathcal{R}^{n\times 1}$) using the attention weights: $\boldsymbol{v} = [a_1 \cdot x_1, ..., a_n \cdot x_n]$.

The attention mechanism is different from the traditional weighted average because since the weight of x_i is dynamically decided (and not fixed) by the global attention vector, the element itself and other elements in $\boldsymbol{x}$ (see Eq. (2)).

The fully connected layer combines the context vector output by the attention layer into a high-level stylistic representation $\boldsymbol{r}$: $\boldsymbol{r} = \tanh\left(\boldsymbol{W_c} \times \boldsymbol{v} + \boldsymbol{b_c}\right)$, where $\boldsymbol{W_c} \in \mathcal{R}^{d\times n}$ and $\boldsymbol{b_c} \in \mathcal{R}^{d\times 1}$ are model parameters. With the hyperbolic tangent function ($\tanh$), the code instance is finally mapped into a vector $\boldsymbol{r}$ ($\boldsymbol{r} \in \mathcal{R}^{d\times 1}$), of which each element is in the scale of $[-1, 1]$. This facilitates NARN to learn a classifier within the space of a unit sphere. For each author y_j ($j \in [1, |\mathcal{Y}|]$), NARN also establishes a label embedding $\boldsymbol{e}_j \in \mathcal{R}^{d\times 1}$, which is trained together with model parameters. The stylistic representation $\boldsymbol{r}$ and the label embedding $\boldsymbol{e}_j$ are then mapped into a shared hidden vector $\boldsymbol{h}_j$ by element-wise product: $\boldsymbol{h}_j = \boldsymbol{r} \cdot \boldsymbol{e}_j$.

The regression layer outputs a confidence score p_j, which indicates the probability that the input code belongs to the j^{th} author: $p_j = \boldsymbol{W_r} \times \boldsymbol{h}_j + b_r$, where $\boldsymbol{W_r} \in \mathcal{R}^{1\times d}$ and $b_r \in \mathcal{R}$ are parameters of the regression layer. The predicted label is taken as the author who has the greatest probability: $y = \arg\max_j(p_j)$.

By using the tokenized path features and neural attentional regression network, we are able to profile different authors' coding style and to craft adversarial examples without any knowledge of the SCAI classifier, thus addressing Challenge 1 in Section III-A.

B. Code Transformation Rules

To transform the source code without changing functionality, we design 5 types of equivalent transformations, containing 37 transformation rules in total. Our transformations mainly focus on the lexical and syntactic elements, which are briefly introduced below.

Type-1: trivial transformations. As listed in Table I, trivial transformations change low-level features of source code, such as symbols, operators or locations. The simplest transformation rule replacing identifiers (Rule 4), which scrambles variable names or function names. Undo type alias (Rule 5) also change identifiers because this rule replaces the declared type alias with the original type name. It is different from deleting type alias declaration in that we do not replace all the usage back to the original type. This enables flexible change of specific tokens.

Type-2: data transformations. Data transformations are listed in Table II, which mainly manipulate data types or data structures. For example, using *typeid* expression (Rule 10) can replace the type name "*int*" with the expression "*typeid (a)*", where *a* is an already defined int variable. We can also convert int literals into hexadecimal numbers (Rule 13) or convert bool literals to int literals (Rule 16).

Type-3: control flow transformations. This type of transformations usually influence the execution process of a pro-

TABLE III
CONTROL FLOW TRANSFORMATIONS

ID	Rule Name	Description and Examples
17	Convert for-statement to while-statement	*for(init; condition; update){ ...;}* $\rightarrow$ *init; while(condition) {...; update;}*
18	Convert while-statement to for-statement	On the contrary to Rule 17.
19	Convert if-else to switch-case	e.g., *if (a > b) {...;}* $\rightarrow$ *switch (a > b) {case true: ...; }*
20	Convert switch-case to if-else	On the contrary to Rule 19.
21	Convert if-else to conditional expression	e.g., *if (a > b) {max=a;} else {max=b;}* $\rightarrow$ *max= a > b ? a : b*
22	Convert conditional expression to if-else	On the contrary to Rule 22.
23	Split conditions of if-statements	Splits the conditions of an if-statement into multiple statements when the logical operator is one of "$\|\|, \&\&, \leq, \geq$".
24	Swap if-else bodies	e.g., *if A do B; else do C;* $\rightarrow$ *if !A do C; else do B;*

TABLE IV
FUNCTION TRANSFORMATIONS

ID	Rule Name	Description and Examples
25	Reorder function arguments	*e.g., saveText(string s, ofstream outfile)* $\rightarrow$ *saveText(ofstream outfile, string s)*
26	Add function arguments	Adds extra arguments which are simply initialized without changing the function output.
27	Merge function arguments	Merges function arguments into a struct type and then use them as struct members. e.g, *int add (int a, int b) {return a+b;}* $\rightarrow$ *int add (args ab) {return ab.a+ab.b;}*, where *ab* is a struct type with two members.
28	Convert statements into functions	Removes initialization, assignment statements into a function, whose arguments are pointers to the influenced variables and return value is void.
29	Convert binary expressions into functions	Extracts variables in the binary expressions as function arguments and returns the computing result. e.g., *c=a+b* $\rightarrow$ *c= add(a,b)*
30	Merge functions	Merges two functions that have the same return type into a combined function and add a switching argument to decide which sub-function is called.
31	Hide API calls	Wraps API calls into a user-defined function. e.g., call "*freopen*" in the body of "*my_open*" function.

TABLE V
ADD BOGUS CODE

ID	Rule Name	Description and Examples
32	Add temp variables	Introduces a temp variable for expressions in array indexes, return statements, if statements, loop statements and function arguments. e.g., *return 0;* $\rightarrow$ *int t=0; return t;*
33	Add redundant operands	Uses redundant operands for addition, subtraction, multiplication and division expressions. e.g., *a * b* $\rightarrow$ *a * b * c/2*, where *c* is initialized to 2.
34	Add libraries	Adds extra libraries that are used by other programmers.
35	Add type alias	Adds extra type alias with the corresponding variable declaration.
36	Add global declarations	Adds global variables and initialize them in the main function.
37	Add function declarations in classes	Declares a function with a class without defining them later.

gram by changing the loop statements or altering the order of computations. As show in Table III, we can convert for-statement to while-statement, and vice versa (Rule 17-18). It is also allowed to convert between if-else to conditional expression (Rule 21-22).

Type-4: function transformations. Function transformations (see Table IV) influence function arguments, function definitions, function calls, number of functions and so on. For example, we can merge function arguments by aggregating function arguments into a struct type and then use them as struct members (Rule 27). Hiding API calls (Rule 31) by wrapping some standard library functions into a user-defined function, is also found to be useful in our attack.

Type-5: add bogus code. This kind of transformations create some extra element in the code without changing the result of computation (see Table V). We can introduce temp variables (Rule 32), add extra type libraries (Rule 34) or add extra type alias (Rule 35). Add bogus function declarations within class (Rule 37) is also allowed as long as we do not call them in the main function.

Our designed set of transformations influence a wide range of elements, including identifiers, operators, keywords, libraries, expressions, functions, etc. Different transformation rules have different influence scope on lines of code (LoC). In our paper, we use changed LoC to represent modified, removed and added LoC. Moreover, there are 10 overlapped rules and 27 different rules compared to the transformations in [1]. The difference between transformations and advantages of our rules are analyzed in Appendix A.

C. Transformation Rule Selection

To decide which transformation rules are useful and the order they are used, we need to know which stylistic features are most important to the output of the substitute model ($\mathcal{F}$). We utilize the Jacobian-based Saliency Map Approach (JSMA) [12], which iteratively changes features that have large adversarial saliency scores until the classifier outputs the target class y_t. Specifically, the value of a feature could be decreased or increased, corresponding to the removal or addition of a tokenized path as in our substitute model. If the occurrence of a path feature decreases, the saliency map computes the following score for the feature x_i:

$$S^{-}(\boldsymbol{x}, y_t)[i] = \begin{cases} 0 \text{ , if } \frac{\partial \mathcal{F}_t(\boldsymbol{x})}{\partial x_i} > 0 \text{ or } \sum_{j \neq t} \frac{\partial \mathcal{F}_j(\boldsymbol{x})}{\partial x_i} < 0 \\ \left|\frac{\partial \mathcal{F}_t(\boldsymbol{x})}{\partial x_i}\right| \left(\sum_{j \neq t} \frac{\partial \mathcal{F}_j(\boldsymbol{x})}{\partial x_i}\right) \text{ , otherwise} \end{cases} \tag{3}$$

where $\mathcal{F}_j$ is the confidence score of the j^{th} class output by the model. The Jacobian value $\frac{\partial \mathcal{F}_j}{\partial x_i}(\boldsymbol{x})$ is the partial derivative of $\mathcal{F}_j$ with regard to the input x_i, i.e., it measures the contribution of each input component x_i to the output y_j. Hence, features with high saliency scores will either increase the possibility of the target class or decrease the possibility of other classes, or both. Similarly, if the occurrence of a path feature increases, a counterpart saliency score is given as Eq. (4).

$$S^+(\boldsymbol{x}, y_t)[i] = \begin{cases} 0 \text{ , if } \frac{\partial \mathcal{F}_t(\boldsymbol{x})}{\partial x_i} < 0 \text{ or } \sum_{j \neq t} \frac{\partial \mathcal{F}_j(\boldsymbol{x})}{\partial x_i} > 0 \\ \left(\frac{\partial \mathcal{F}_t(\boldsymbol{x})}{\partial x_i}\right) \left| \sum_{j \neq t} \frac{\partial \mathcal{F}_j(\boldsymbol{x})}{\partial x_i} \right| \text{ , otherwise} \end{cases} \tag{4}$$

However, the original crafting algorithm in [12] takes pixel intensities as input and iteratively updates the most important feature by a fixed change θ until the target misclassification is achieved or the feature search domain is empty. Their algorithm generates brightened images when $\theta > 0$ and darkened images when $\theta < 0$. The generation approach for images cannot be directly applied to source code for two reasons: ① the value of θ cannot be fixed because our code transformations may increase some feature values while decreasing other values at the same time; ② the value of θ cannot be set to an arbitrary value as it may not produce realistic code instances. Therefore, we design a customized JSMA method, which directly measures the influence of a transformation rule r using a combined saliency score (CSS) of the features it changes: $CSS(\boldsymbol{x}, y_t, r) = \sum_{p \in DX} S^-(\boldsymbol{x}, y_t)[p] + \sum_{q \in IX} S^+(\boldsymbol{x}, y_t)[q]$, where DX is the set of indexes of all the decreased features, and IX is the set of indexes of all the increased features.

Our JSMA-based algorithm for source code transformation works by iteratively selecting the most effective transformation. At each step, it calculates the CSS value for each possible transformation (supposing that the transformation rule is applied).Then it accepts the rule with the highest CSS score and updates the code by applying the transformation. To imitate an targeted author y_t, the transformation decision is repeated until the substitute authorship classifier C_a outputs: $\mathcal{F}(\boldsymbol{x}') = y_t$. To mask a programmer's real style, the algorithm can be easily applied to untargeted adversarial examples by changing the terminal condition into $\mathcal{F}(\boldsymbol{x}') = y' \neq \mathcal{F}(\boldsymbol{x})$. By selecting the most effective transformations, SCAD aims to change the style of source code while modifying as few code as it can, thus addressing Challenge 2 in Section III-A.

D. Authorship Disguise Verification

After we obtain the transformed code s', we need to verify its effectiveness by checking whether it can be attributed to a false author by the original identification classifier. Next, we describe four kinds of authorship disguise according to classification settings and the false output.

Closed-world classification and open-world classification. In a closed-world classification problem, all the test samples are classified as candidate categories in the training set, while in open-world classification a test sample may belong to an unknown category that is not in the training set. If the classification confidence of a test sample is low, we attribute it to an unknown class (denoted by a "-1" label). Under closed-world settings, we deploy targeted authorship disguise (TAD) and untargeted authorship disguise (UAD), i.e., choose a targeted programmer or an untargeted programmer in the original classifier's training set as the required adversarial output. Under open-world settings, we deploy inclusion authorship disguise (IAD) and exclusion authorship disguise (EAD). In IAD, SCAD tries to induce programmers who are not in the original classifier's training set to be classified as those that are in. Oppositely, for EAD, SCAD tries to disguise programmers who are in the original classifier's training set to be classified as those that are not. The four kinds of authorship disguise are thus formalized as below. ① UAD: $C(s') \neq C(s)$; ② TAD: $C(s') = y_t \wedge y_t \neq C(s)$; ③ IAD: $C(s') \neq -1 \wedge C(s) = -1$; ④ EAD: $C(s') = -1 \wedge C(s) \neq -1$.

Indeed, TAD, IAD and EAD are all subcases of UAD. The programmer can choose one of the four kinds to disguise his/her identity. In our experiments, we evaluated TAD/UAD under closed-word setting and IAD/EAD under open-world setting (see Section V-B).

V. Evaluation

A. Experimental Setup

Building corpus. Following the prior works [1–3], we utilize Google Code Jam [26], a coding competition platform on which individual programmers solve coding challenges within limited time, to build our datasets. In this paper, we focus on C++ code, the most popular language used in the competition. We collect C++ solutions in GCJ competitions from 2012 to 2017. As shown in [3], having more code samples per author enables better authorship classification performance and 7 samples per author is enough to achieve an accuracy above 90%. Motivated by these observations, we select 7 samples for each author (denoted as S_{train}) for training the classifier while using another 4 samples (denoted as S_{test}) for testing authorship disguising attacks. After these selections, there are 6,005 programmers who have more than 11 code instances remaining in our final dataset.

Reproducing the original SCAI classifiers. Considering the state-of-the-art classifiers in Section II, the features that are utilized include: CSFS features, word-level n-grams and character-level n-grams. For simplicity, we refer to them as CSFS, WORD and CHAR, respectively. The two classification models used in these works are: RFC and RNN-RFC. Different combinations of these features and models would constitute six SCAI classifiers: CSFS+RFC, CSFS+RNN-RFC, WORD+RFC, WORD+RNN-RFC, CHAR+RFC, and CHAR+RNN-RFC. We also improve these models by setting a threshold of confidence to identify unknown authors that are not included in the training set (labeled as "-1").

Implementing SCAD. Before extracting features, we use Artistic Style [27] to organize code in a uniform format: indenting with spaces, limiting the maximal line length, using Linux braces [28], spacing between operators and brackets, etc. We also remove the comments and empty lines, after which the average LoC of each code file is 72.11. Then we use

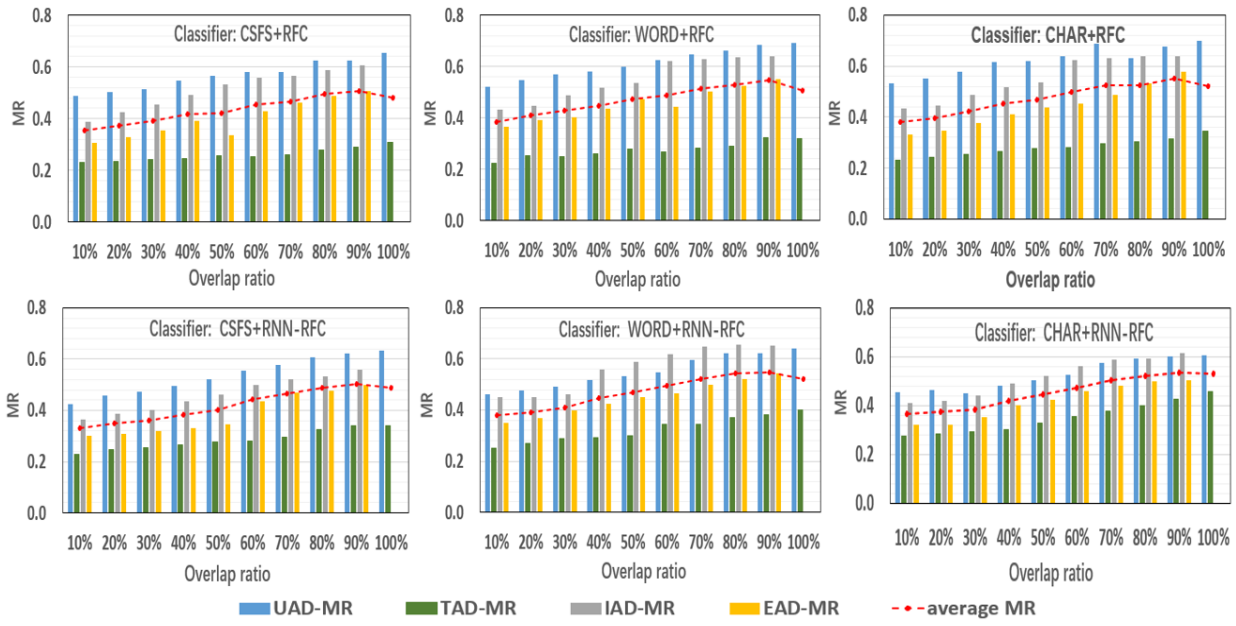


Fig. 3. MR varying with different overlap ratios under different classifiers.

C++ Development Tooling (CDT) plug-ins [29] for Eclipse to parse AST. Specifically, we expand macros and replace literals with specific tags such as *INT_LITERAL*, *STRING_LITERAL*, etc. There are 81 types of AST nodes, of which the most frequent nodes are *Name*, *SimpleDeclSpecifier* and *SimpleDeclaration*. There are 103,345 tokenized paths extracted from the whole corpus and the average length (number of nodes on a path) is 8. We select the top n paths using the normalized feature score as defined in Eq.(1) on the whole dataset. We then evaluate the accuracy varying with different n when training NARN, from which we observe that 6K to 10K features are enough to acquire over 90% accuracy. So we set n to 10K to cover as many types of nodes (58 out of 81) as possible while achieving the highest accuracy. The NARN model is implemented with Tensorflow [30], on a server with 2 cores, 64 GB memory, and a GeForce GTX 1080 Ti GPU. The size of NARN is: $n = 10,000$ (the dimension of the attention vector), $d = 1024$ (the rows of $\boldsymbol{W_c}, \boldsymbol{b_c}, \boldsymbol{e}_j$ and the columns of $\boldsymbol{W_r}$). We leverage the cross entropy loss [31] and Adam optimizer [32] to train our model. During the training process, we use a learning rate of 10^{-4}, a batch size of 128, and a dropout rate of 0.25 to avoid overfitting.

B. Effectiveness in Evading Authorship Identification Systems

In our experiments, we use misclassification rate (the ratio of adversarial examples being misclassified by the target SCAI classifier) to quantify the effectiveness of our disguise algorithm. For brevity, we use MR to denote misclassification rate in the following analysis.

Experiment 1: effectiveness in practical black-box settings. As a practical black-box attack, the model structure and training set of the target classifier are not known. So we should build our own substitute model, varying the overlap ratio between programmers in the training set of the SCAI model and the substitute model (while in [1] they assumed the two training sets are the same). In order to show the effectiveness of SCAD as well as its advantages over previous methods, we select 1,600 programmers for both models with different overlap ratios, ranging from 10% to 100% (attack under 0% overlap ratio is meaningless since each adversarial target selected from the substitute training set is not included in the original training set). Specifically, we conduct all the four kinds of authorship disguise and the attack performance is shown in Figure 3. From Figure 3, we have the following important observations. ① SCAD can achieve over 30% MR for all settings and even over 60% for UAD. In comparison, the state-of-the-art automatic method [1] achieves 52% MR when all the original training data (204 programmers) and the architecture of original training model are known, and 99% MR when the prediction scores of the original training model are known. Therefore, our work is more effective in practice when there is a larger number of programmers or the architecture/prediction scores of the original model are unknown. ② MR increases with a larger overlap ratio, indicating that SCAD can generate stronger adversarial source code when more information regarding the original training data is accessible. ③ MR of UAD is much higher than that of TAD for all the six classifiers, implying that masking a programmer's styple is easier than impersonating someone else. ④ MR of IAD is higher than that of EAD for all the six classifiers, since IAD only requires to pull the original author into anyone in the original training set while in comparison EAD requires to pull the original author away from all the other authors in the training set. ⑤ CSFS+RFC classifier is more robust than other classifiers. One possible reason is that the CSFS features are usually more representative than the text-based features such as WORD features.

Experiment 2: compare with other baselines. The only existing approach related to adversarial code stylometry was Code-Imitator, proposed by Quiring et al. [1]. Their attacks

TABLE VI
ATTACK PERFORMANCE OF DIFFERENT APPROACHES WITH PREDICTION SCORES ACCESSIBLE

Programmers	Approaches	Misclassification Rate	
		Targeted	Untargeted
204	Code-Imitator	**71.25%**	88.79%
	SCAD	70.59%	**90.13%**
1600	Code-Imitator	55.10%	75.34%
	SCAD	**64.66%**	**80.80%**

TABLE VII
ATTACK PERFORMANCE OF DIFFERENT APPROACHES WITH CLASSIFIER INTERNALS KNOWN

Programmers	Approaches	Misclassification Rate	
		Targeted	Untargeted
204	Code-Imitator	47.96%	**70.75%**
	SCAD	**50.76%**	69.06%
1600	Code-Imitator	40.99%	62.40%
	SCAD	**44.14%**	**65.35%**

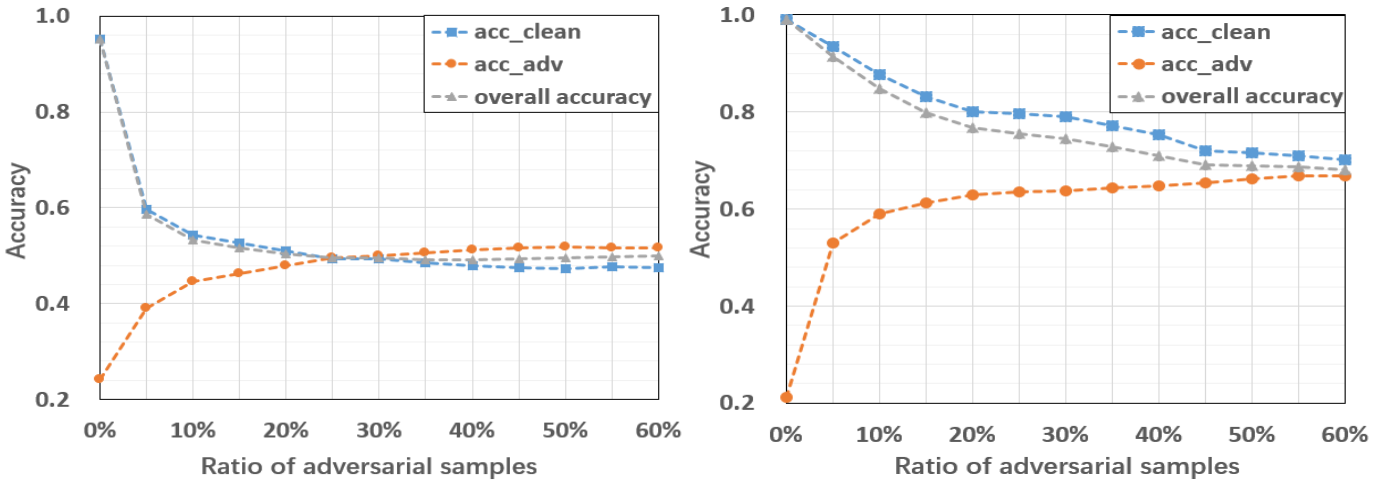
were conducted under two gray-box settings. That is, only the prediction scores accessible or the internal details of the target classifier known. Specifically, when adversaries can retrieve the classification labels along with the corresponding prediction scores, they designed Monte-Carlo Tree Search to backpropagate the classifier scores in nodes and select the next transformation. When prediction scores cannot be accessed directly, they conducted one supplementary experiment where a substitute model was implemented with the same internals of the target SCAI classifier (i.e., using the same features and model architectures, with 100% overlap ratio of programmers). To fairly compare SCAD and Code-Imitator, we follow the same gray-box scenarios. When prediction scores are accessible, we train our NARN substitute model based on the code samples and the prediction scores output by the target classifier. When the internals of the target classifier are known, we train a substitute model following the same way in [1]. The attacks are conducted on the small dataset used in [1] (204 programmers, 8 coding challenges) and another larger dataset (1600 programmers, 8 coding challenges), against the CSFS+RFC classifier, which is the most robust as shown in Experiment 1. Specifically, we use 4 code instances of each programmer to train the original model and use the other 4 to train the substitute model, with same feature representation methods and model architectures. As shown in Table VI and VII, SCAD achieves higher misclassification rates than Code-Imitator in most of the cases. Especially on the larger dataset, when the prediction scores of the CSFS+RFC classifier can be utilized, SCAD outperforms Code-Imitator (80.80% vs 75.34% for untargeted attacks, 64.66% vs 55.10% for targeted attacks). The improved performance of SCAD, compared to the results in Experiment 1, is possibly because the NARN model trained with the same prediction scores of the original classifier tends to learn similar decision boundaries.

Besides, we compare with Stunnix [33], a popular C/C++ obfuscator that scrambles identifier names by using md5, adding prefixes or changing characters. We also validate the effectiveness of the JSMA-based rule selection, comparing with applying random transformation rules (RandTrans) and all the transformations (AllTrans). We directly apply these transformations on a random selection of 100 code samples to see whether they can make any misclassification. The results of these simple baselines are shown in Table VIII. For Stunnix, the MRs are less than 10%, which shows simple obfuscations that change the layout of code is not enough to hide programming styles. Both AllTrans and RandTrans have low MRs, which shows our JSMA-based algorithm guides the transformation selection more efficient. We do not consider sophisticated code obfuscation techniques such as control flow flattening and using opaque predicates, since such obfuscation is perceptibly strange and less readable for humans, usually with abnormal instructions and obscure loops. Further, the previous work [3] has proven that authorship identification classifiers were able to maintain above 90% accuracy on 120 programmers, with obfuscation introduced in their C code.

TABLE VIII
ATTACK PERFORMANCE OF SIMPLE TRANSFORMATIONS

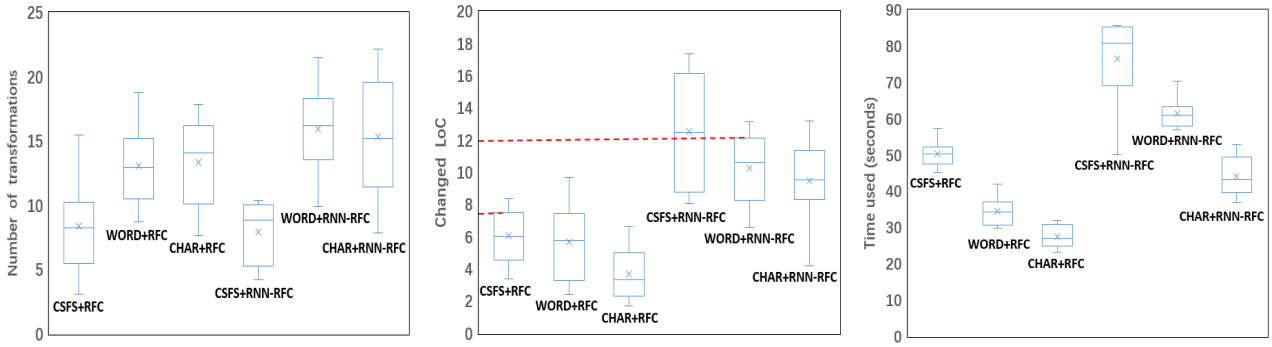
	Misclassification Rate (Untargeted)
Stunnix	5.60%
RandTrans	12.64%
AllTrans	22.46%

Experiment 3: robustness against model re-training. A potential countermeasure to improve the robustness of authorship classifiers is to re-train their models by leveraging adversarial samples [18, 34]. Although incorporating adversarial samples with correct authorship labels is rather difficult in practice, we still illustrate the impact of re-training on the performance of SCAD. The intuition for this experiment is that a programmer with randomized coding styles would be more indistinguishable. Therefore, for each author, the adversarial target y_t is randomly selected so an author may be disguised as multiple targets. We thus generate multiple adversarial samples for each of the original samples belonging to 250 programmers and add them to the training set, with the ratio of adversarial samples ranging from 0% to 60%. The performance of SCAD after re-training is shown in Figure 4, from which we observe that the accuracy on the adversarial samples (acc_adv) increases while the accuracy on the clean samples (acc_clean) decreases with a larger ratio of adversarial samples. Specifically, the accuracy of CSFS+RFC classifier decreases from $\sim$90% to $\sim$50% and the accuracy of CSFS+RNN-RFC classifier decreases from $\sim$100% to $\sim$70%. The overall accuracy drops and then tends to remain unchanged when the ratio of adversarial samples exceeds 35% (for RFC in Figure 4(a)) and 45% (for RNN-RFC in Figure 4(b)), demonstrating the inherent robustness of SCAD. An interesting observation here is that RNN-RFC suffers smaller accuracy degradation than RFC in classifying clean samples. A possible reason is that RNN-RFC adjusts its feature re-construction kernel (which is missing in RFC) in its shallow layers (e.g., the fully connected layer) to pull the adversarial example back to a deep representation located



(a) Classifier: CSFS+RFC (b) Classifier: CSFS+RNN-RFC

Fig. 4. Accuracy varying with different ratios of adversarial samples for authorship under different authorship identification classifiers that leverage (a) CSFS features with RFC models and (b) CSFS features with RNN-RFC models.



(a) Times of transformations (b) Percentage of changed LoC (c) Running time

Fig. 5. The efficiency of SCAD in evading different classifiers. The two whiskers outside the box correspond to the maximum and minimum values, respectively. The bottom and the top of the box are the first quartile and the third quartile. The band within the box is the median value and the cross symbol shows the average value.

close to the original class.

C. Efficiency in Practical Applications

Next, we analyze the efficiency of SCAD by leveraging the following three metrics. **(a) Times of transformations.** This metric measures how many times the code is transformed using the rules. For example, if Rule 1 is used once and Rule 2 is used for three times, the total number of transformations is four. **(b) Percentage of changed LoC. (c) Time used to craft an adversarial code instance.** Note that we only consider the time for rule selection and code transformation since the substitute model is trained offline.

All the metrics are calculated over all the adversarial samples generated from the 4 test codes belonging to 6,005 programmers, and the corresponding statistics are shown in Figure 5. From the box-plots we have the following important observations: ① most adversarial samples only involve a few transformations and a small fraction of changed code. For the three RFC classifiers, there are 5-15 times of transformation and less than 7 LoC that are changed (the percentage is about 9.7%). For the three RNN-RFC classifiers, the cost is a bit higher with 5-20 times of transformation and less than 16 changed LoC. Specifically, for CSFS+RFC, 75% of the adversarial code have less than 7 changed LoC as compared to 10 Loc in [1]. For WORD+RNN-RFC, 75% of the adversarial codes have less than 12 changed LoC, which is similar to [1]; ② in general, 60 seconds are enough for most of the adversarial samples to be generated, which is comparative to the time required to generate adversarial text in [14]; ③ evading RNN-RFC models are more time-consuming than evading RFC models.

D. Utility-Preserving Properties

The utility of adversarial samples is another important property that we need to take into consideration. In the domain of text, the utility of adversarial samples mainly relies in semantic similarity, i.e., the adversarial text should not be changed too much in semantic level to maintain its original functionality. However, when applied to analyze code semantics, similarity-based methods may be indicative but not provable, because programs with even 99% similarity may still have different functionalities. Thus, we measure the

utility of adversarial code from two aspects: stealth [35] and functionality equivalence.

Stealth. It is required that the adversarial sample should not be changed so much in its appearance as compared to the original sample. As the change of codes is not quantitative, we design two metrics to measure the difference between their feature vectors: the percentage of changed features and the normalized L_1-distance. The percentage of changed features is the percentage of different elements between the original feature vector $\boldsymbol{x}$ and the adversarial feature vector $\boldsymbol{x'}$. The normalized L_1-distance is calculated as: $d(\boldsymbol{x}, \boldsymbol{x'}) = \sum_{i=1}^{n} |u_i - u'_i|$, where $u_i = \frac{x_i}{\sum_{i=1}^{n} |x_i|}$, $u'_i = \frac{x'_i}{\sum_{i=1}^{n} |x'_i|}$ are L_1-normalized values in the two vectors. Both the two metrics are averaged over all the adversarial samples against the same classifier. Figure 6(a) shows the distribution of adversarial code with different percentages of changed features. For both classifiers, the majority of adversarial examples have less than 50% changed features. Furthermore, we observe two peaks for the result of CSFS+RFC classifier. The first peak shows that a few of adversarial examples have small percentages of changed features (0%-30%). The second peak shows that a large number of adversarial examples are altered by 40%-60% of all the features. While for WORD+RNN-RFC classifier, the majority of adversarial examples are altered by 0%-30% of features. Therefore, we can conclude that CSFS+RFC classifier requires more feature change than WORD+RNN-RFC classifier. Figure 6(b) shows the distribution of the normalized L_1-distance. For both classifiers, the majority of adversarial examples are close to the original examples with a distance of less than 0.5. CSFS+RFC classifier requires a larger change of distance than WORD+RNN-RFC classifier for 60% of the adversarial examples. From the two figures, we can observe that CSFS+RFC classifier is more robust against disguised code style than WORD+RNN-RFC classifier. One possible reason is that the CSFS features are more powerful to capture complicated stylistic patterns than WORD features.

Functionality equivalence. Although our transformations are carefully designed to preserve the code semantic, we still need to check if the same outputs are preserved for two code instances. As all the problems selected from GCJ competitions provide test cases with pairs of input-output values, we can simply use the test inputs and check whether the outputs have changed before and after transformation. In our experiments, all the adversarial code samples pass the output verification, validating the function-preserving property of adversarial codes generated by SCAD.

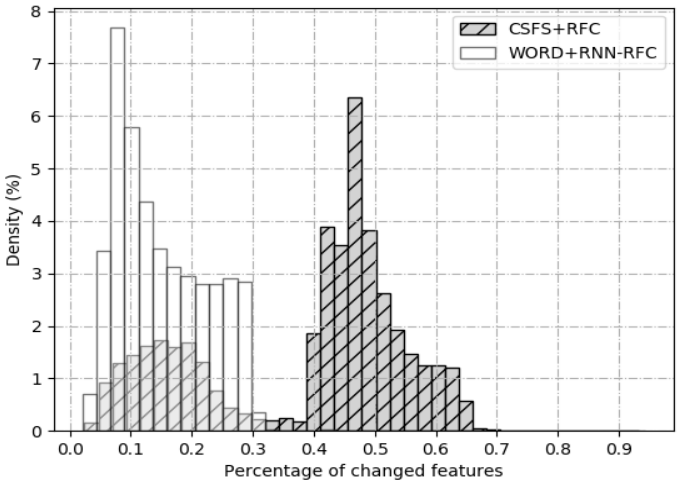
E. Summary of Evaluation Results

Based on our analysis above, we summarize important observations in our evaluation as follows.

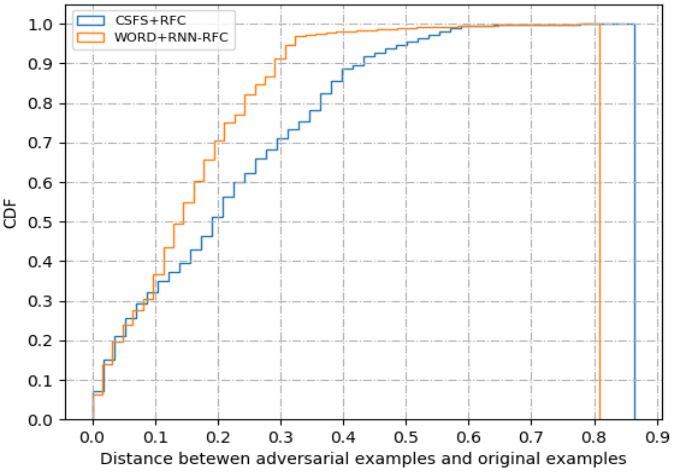
(1) SCAD can effectively deduce six source code authorship classifiers to have significant misclassification rates (above 30%), even against the most robust classifier (CSFS+RNN-RFC) and on datasets consisting of more than thousands of programmers.

(2) SCAD can efficiently craft code samples with adversarial stylistic features, through changing a small fraction of code by leveraging a set of well-designed transformations. Considering that SCAD is automatic and offline, the implementation time (approximately 60 seconds) is acceptable in practice.

(3) SCAD well preserves the utility of source code, where the transformed code only presents small perturbations in its feature space while maintaining the same functionality as compared to the original source code.



(a) Changed features



(b) Normalized L_1-distance

Fig. 6. The stealth of SCAD in evading different classifiers.

VI. Further Analysis and Limitations

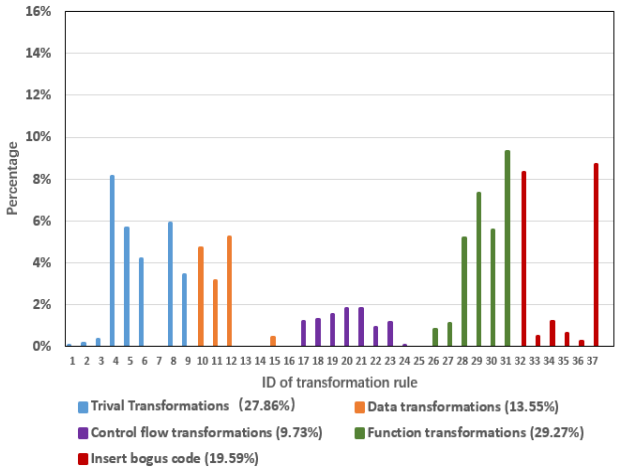
A. Futher Analysis of SCAD

Contribution of each transformation. To show which transformations are useful for authorship disguise, we measure the contribution of each transformation rule/type by the ratio of the times of using this rule/type among all the transformations. In Figure 7(a), the targeted classifier is RFC model using CSFS features, which assigns importance on manually defined lexical features and syntactic features, such as TF/TF-IDF of tokens, AST leaves, AST unigrams, AST bigrams and number of functions, average arguments, depth of AST node, etc. The most useful transformation type for misleading CSFS+RFC is function transformation (29.27%), which significantly changes

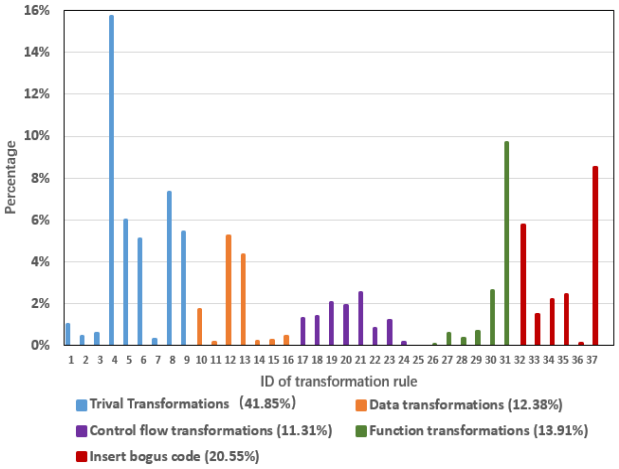
the number of functions, average arguments as well as other syntactic features influenced by the transformed functions. The trivial transformation is also useful because it mainly changes features related to identifiers, type names and operators. The most useful transformation rule is Rule 31 (hide API calls). In Figure 7(b), the targeted classifier is RNN-RFC model using n-gram features, which assigns importance to certain keywords and phrases such as "*return 0*". The most useful transformation types against this classifier are trivial transformations (41.85%) and adding bogus code (20.55%). The most useful transformation rule against this classifier is Rule 4 (replace identifiers). For both classifiers, control flow transformation is the least useful one. Moreover, there exist some transformations that are rarely or never used, such as Rule 13, Rule 14 and Rule 16 in Figure 7(a) since they are all literal transformations which are usually ignored by CSFS features. Rules 24-25 are also less useful as shown in Figure 7(b) because they change the order of if-else or operands while the WORD features do not focus too much on the order information.

Composite transformations by combing different rules. In our experiments, we find some interesting modifications in the adversarial code which are formed by applying multiple rules continuously. For example, using Rule 29 on the expression "*x+y>c*" continuously will convert it into wrapped function calls such as "*isLarger(add(x,y),c)*", where *is_larger* and *add* are two functions. Another example is that combining Rule 12 and Rule 29 will convert int literals into functions. We believe that the flexible combination of different transformations strengthens the power of single rules in disguising coding style, thus increasing the difficulty of recovering the transformed code especially when the combination is not known.

Based on the above analysis, we can observe that the 5 types of transformations which change a wide range of code elements are adversarially selected to mask or impersonate programmers' coding styles. These properties enable SCAD to work effectively under totally black-box settings, when the prediction scores or structures of the original authorship classifiers are difficult to obtain in practice.



(a) Classifier: CSFS + RFC



(b) Classifier: WORD + RNN-RFC

Fig. 7. Usage of transformation rules against different authorship identification classifiers that leverage: (a) CSFS features with RFC models and (b) WORD features with RNN-RFC models.

B. Limitations and Future Works

There exist several limitations of SCAD. Firstly, the accuracy of SCAI classifiers under our attack is higher than random guess, which is not sufficient to achieve anonymity. This is possibly because our transformation rules are mainly local transformations which change one or several statements, thus resulting in limited influence on complex code logic. Secondly, the output verification may not completely guarantee function equivalence of two programs, since we can only check a limited cases of test inputs and our analysis is not applicable to programs whose output is not a value (e.g., programs written to create a process or to delete a file). A possible countermeasure is that programmers can define (expected) unchanged objects and check them by inserting "assert" statements.

In the future, we also plan to incorporate more global transformations to further enhance the performance of disguising the source code. How to adapt generative adversarial networks (GAN) [36] to source code and insert functions as bogus bode in the header file would also be another interesting direction.

VII. Conclusion

In this paper, we design an effective authorship disguising approach (SCAD) for source code stylometry, which is automatic, learning-based and utility-preserving at the same time. In SCAD, we train a substitute authorship attribution model and propose a customized JSMA algorithm to automatically transform source code according to 37 well-designed rules. Our approach works under a totally black-box setting, without any prior knowledge of the prediction scores or model structure of the original classifier as previous work did. Our work reveals that although existing code authorship identification classifiers have gained high accuracy, they are vulnerable to practical adversarial manipulations. More robust stylistic features and identification models should be developed to avoid authorship disguise attack.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments to improve this article. This work was partly supported by NSFC under No. U1936215, 61772466, and U1836202, the National Key Research and Development Program of China under No. 2020YFB2103802, 2018YFB0804102, and 2020AAA0140004, the Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under No. LR19F020003, and the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform).

APPENDIX
DIFFERENCE BETWEEN TRANSFORMATION RULES

We summarize the following important discussions for the practical adoption of our transformation rules and the difference with those of [1].

• Transformation scope. We use a certain rule once at a position instead of applying global transformation. For example, we can convert part of the for-statements while keeping the others. While [1] applies global transformation, such as converting all the "for" loops and using type alias for

TABLE IX
COMPARE WITH OUR TRANSFORMATION RULES WITH [30]

Rules of SCAD	Overlaps or Differs	Rules of [30]
remove unused code^{I}	■	unused code transformer & include-remove transformer
replace identifiers^{I}	■	identifier transformer
uniform schemes of using braces^{I}	■	compound statement transformer (insert option/ delete option)
convert between string objects and char arrays^{II}	■	string transformer (array option / string option)
convert between bool literals and integers^{II}	■	boolean transformer (bool option / int option)
convert for statements into while statements^{III}	■	for statement transformer
convert while statements into for statements^{III}	■	while statement transformer
split conditions of if statements^{III}	■	if statement transformer
add include libraries^{V}	■	include transformer
add global declarations^{V}	■	global declaration transformer
undo type alias^{I}	⊡	typedef transformer (delete option)
convert statements/expressions into functions^{IV}	⊡	function creator transformer
add type alias^{V}	⊡	typedef transformer (convert option)
add temp variables^{V}	⊡	literal transformer
split/aggregate declarations^{I}	⊕	/
separate/attach elaborated type declaration^{I}	⊕	/
use alternative tokens^{I}	⊕	/
swap operands^{I}	⊕	/
use converse-negative expressions^{I}	⊕	/
use equivalent computations ^{I}	⊕	/
use typeid expression^{II}	⊕	/
use cast expressions^{II}	⊕	/
convert int literals into expressions^{II}	⊕	/
convert integers into hexadecimal numbers^{II}	⊕	/
convert char literals into ASCII values^{II}	⊕	/
convert if-else to switch-case^{III}	⊕	/
convert switch-case to if-else^{III}	⊕	/
convert if-else to conditional expression^{III}	⊕	/
convert conditional expression to if-else^{III}	⊕	/
swap if-else bodies^{III}	⊕	/
reorder function arguments^{IV}	⊕	/
add function arguments^{IV}	⊕	/
merge function arguments^{IV}	⊕	/
merge functions^{IV}	⊕	/
hide API calls^{IV}	⊕	/
add redundant operands^{V}	⊕	/
add function declarations in classes^{V}	⊕	/
/	⊖	deepest block transformer
/	⊖	array transformer
/	⊖	integral type transformer
/	⊖	floating-point type transformer
/	⊖	init-decl transformer (move-in option / move-out option)
/	⊖	input interface transformer (stdin option / stdout option)
/	⊖	output interface transformer (stdin option / stdout option)
/	⊖	input API transformer (C++ option/ C option)
/	⊖	output API transformer (C++ option / C option)
/	⊖	sync-with-stdio transformer
/	⊖	return statement transformer

I: Trivial transformations
II: Data transformations
III: Control transformations
IV: Function transformations
V: Bogus code transformations

■ the overlapped rules
⊡ the rules are generally the same but have some differences
⊕ the rules uniquely used in SCAD
⊖ the rules uniquely used in [30]

all the "int" types, which may lead more changed LoC than applying a transformation once at a time as we did.

• Number of transformations. Among all our transformations, many rules have sub-cases. For example, we can choose one out of the 26 specific symbols to replace with its alternative token. Therefore, there exist more than 37 transformation options in our approach. While there are only 36 transformation options in [1].

• Overlaps and differences between rules. As illustrated in Table IX, we have only 10 rules overlapped with [1] and 27 rules partly or totally different from theirs. Of the 27 different rules, 4 rules have some slight difference with [1]. For example, we add temp variables into various return statements while the return statement transformer in [1] only substitutes a return statement that returns an integer literal by a statement that returns a variable. Besides, our function transformations can change function arguments, number of functions, depth of functions and the API calls, while [1] only transforms between C++-style APIs and C-style APIs. The other 23 rules are uniquely designed for SCAD, which are composed of 6 new trivial transformations, 5 new data transformations, 5 new control transformations, 5 new function transformations and 2 bogus code transformations.

REFERENCES

[1] E. Quiring, A. Maier, and K. Rieck, "Misleading authorship attribution of source code using adversarial learning," in *USENIX Security Symposium*, 2019.

[2] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code stylometry," in *USENIX Security Symposium*, 2015, pp. 255–270.

[3] M. Abuhamad, T. AbuHmed, A. Mohaisen, and D. Nyang, "Large-scale and language-oblivious code authorship identification," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 101–114.

[4] I. Krsul and E. H. Spafford, "Authorship analysis: Identifying the author of a program," *Computers & Security*, vol. 16, no. 3, pp. 233–257, 1997.

[5] B. S. Elenbogen and N. Seliya, "Detecting outsourced student programming assignments," *Journal of Computing Sciences in Colleges*, vol. 23, no. 3, pp. 50–57, 2008.

[6] S. Burrows, A. L. Uitdenbogerd, and A. Turpin, "Application of information retrieval techniques for source code authorship attribution," in *International Conference on Database Systems for Advanced Applications*. Springer, 2009, pp. 699–713.

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

[8] 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 *Proceedings of the International Conference on Neural Information Processing*. IEEE, 1999.

[9] G. Frantzeskou, E. Stamatatos, S. Gritzalis, and S. Katsikas, "Effective identification of source code authors using byte-level information," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 893–896.

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

[11] C. McKnight and I. Goldberg, "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*. ACM, 2018, pp. 138–142.

[12] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 372–387.

[13] B. Liang, H. Li, M. Su, P. Bian, X. Li, and W. Shi, "Deep text classification can be fooled," in *International Joint Conferences on Artificial Intelligence Organization (IJCAI)*, 2018, pp. 4208–4215.

[14] J. Li, S. Ji, T. Du, B. Li, and T. Wang, "Textbugger: Generating adversarial text against real-world applications," in *Network and Distributed System Security Symposium*, 2018.

[15] A. Caliskan, F. Yamaguchi, E. Dauber, R. Harang, K. Rieck, R. Greenstadt, and A. Narayanan, "When coding style survives compilation: De-anonymizing programmers from executable binaries," in *Network and Distributed System Security Symposium*, 2018.

[16] N. Rosenblum, X. Zhu, and B. P. Miller, "Who wrote this code? identifying the authors of program binaries," in *European Symposium on Research in Computer Security*. Springer, 2011, pp. 172–189.

[17] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," *Computer Science*, 2013.

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

[19] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *S&P*, 2017.

[20] M. Brennan, S. Afroz, and R. Greenstadt, "Adversarial stylometry: Circumventing authorship recognition to preserve privacy and anonymity," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 3, p. 12, 2012.

[21] A. W. McDonald, S. Afroz, A. Caliskan, A. Stolerman, and R. Greenstadt, "Use fewer instances of the letter "i": Toward writing style anonymization," in *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 2012, pp. 299–318.

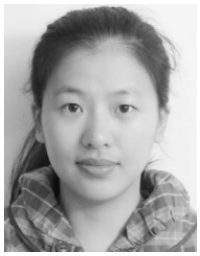
[22] Unstyle. https://github.com/pagea/unstyle.

[23] R. Shetty, B. Schiele, and M. Fritz, "A4nt: Author attribute anonymity by adversarial training of neural machine translation," in *USENIX Security Symposium*,

2018, pp. 1633–1650.
[24] N. Papernot, P. McDaniel, and I. Goodfellow, "Transferability in machine learning: from phenomena to black-box attacks using adversarial samples," *arXiv preprint arXiv:1605.07277*, 2016.
[25] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *arXiv preprint arXiv:1508.04025*, 2015.
[26] Google Gode Jam. https://code.google.com/.
[27] Artistic Style. http://astyle.sourceforge.net/astyle.html/.
[28] Linux Kernel Coding Style. [Online]. Available: http://biocluster.ucr.edu/ jhayes/slurm/coding_style.pdf.
[29] Eclipse CDT plug-ins. https://github.com/eclipse-cdt/cdt.
[30] Tensorflow. https://github.com/tensorflow/tensorflow.
[31] R. Y. Rubinstein and D. P. Kroese, *The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation and machine learning*. Springer Science & Business Media, 2013.
[32] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.
[33] Stunnix. http://stunnix.com/.
[34] S. W. Akhtar, S. Rehman, M. Akhtar, M. A. Khan, F. Riaz, Q. Chaudry, and R. Young, "Improving the robustness of neural networks using k-support norm based adversarial training," *IEEE Access*, vol. PP, no. 99, pp. 1–1, 2016.
[35] 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 *Acm Sigsac Conference on Computer & Communications Security*, 2016.
[36] Y. Song, R. Shu, N. Kushman, and S. Ermon, "Generative adversarial examples," *arXiv:1805.07894*, 2018.

Qianjun Liu is currently a Ph.D. student in the College of Computer Science and Technology at Zhejiang University. She received her Bachelor's degree in Information Security from Nanjing University of Posts and Telecommunications. Her current research interests include data security, natural language processing and code analysis.

Shouling Ji is a ZJU 100-Young Professor in the College of Computer Science and Technology at Zhejiang University and a Research Faculty in the School of Electrical and Computer Engineering at Georgia Institute of Technology. He received a Ph.D. in Electrical and Computer Engineering from Georgia Institute of Technology and a Ph.D. in Computer Science from Georgia State University. His current research interests include AI Security, Data-driven Security and Data Analytics. He is a member of IEEE and ACM and was the Membership Chair of the IEEE student Branch at Georgia State (2012-2013)



Changchang Liu received the Ph.D. degree in electrical engineering from Princeton University. She is currently a Research Staff Member of the Department of Distributed AI, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA. Her current research interests include federated learning, big data privacy, and security.

Chunming Wu is a Professor in the College of Computer Science and Technology, Zhejiang University. His research interests include Network System Architecture, Software Defined Netork and Network Security.