Lucy Simko*, Luke Zettlemoyer, and Tadayoshi Kohno

Recognizing and Imitating Programmer Style: Adversaries in Program Authorship Attribution

Abstract: Source code attribution classifiers have recently become powerful. We consider the possibility that an adversary could craft code with the intention of causing a misclassification, i.e., creating a *forgery* of another author's programming style in order to hide the forger's own identity or blame the other author. We find that it is possible for a non-expert adversary to defeat such a system. In order to inform the design of adversarially resistant source code attribution classifiers, we conduct two studies with C/C++ programmers to explore the potential tactics and capabilities both of such adversaries and, conversely, of human analysts doing source code authorship attribution. Through the quantitative and qualitative analysis of these studies, we (1) evaluate a state-of-the-art machine classifier against forgeries, (2) evaluate programmers as human analysts/forgery detectors, and (3) compile a set of modifications made to create forgeries. Based on our analyses, we then suggest features that future source code attribution systems might incorporate in order to be adversarially resistant.

Keywords: Authorship Attribution, Source Code Attribution, Machine Learning, Adversarial Stylometry, Privacy, Computer Security

DOI 10.1515/popets-2018-0007
Received 2017-05-31; revised 2017-09-15; accepted 2017-09-16.

1 Introduction

With recent publicity surrounding cyber attacks and corresponding investigations into attribution of the attacks, the computer security research community has renewed its attention on authorship attribution of programs. This line of research seeks to answer the following question: given source code (or a binary) of unknown authorship, as well as examples of source code (or binaries) from a set of authors, is it possible to identify the original author of that program? Prior research on this question has shown significant promise — it *is* possible to attribute the authorship of programs with a high degree of accuracy under realistic assumptions. For example, Caliskan-Islam et al. [5] propose a classifier that achieves 96% accuracy over 250 programmers.

Computer security involves adversaries trying to defeat security mechanisms, and the designers of those security mechanisms seeking to defend against adversaries. We argue that the next phase in the evolution of program authorship attribution is therefore considering potential adversaries — parties seeking to fool authorship attribution systems. Specifically, *because* strong program authorship attribution systems now exist, albeit not designed for adversarial contexts, we can experimentally explore how adversaries might seek to defeat those systems and whether those adversaries might be successful. This paper explores both the *methods* that an adversary might use to intentionally subvert an attribution system, as well as a conservative estimate of the adversary's *capabilities*. By understanding the adversaries' tactics and capabilities, as well as their effect on a (public) state-of-the-art attribution system, our results can inform the design of adversarially resistant authorship attribution systems.

Adversarial Goals. We consider two related goals for an adversary: to create a *forgery* of another author, and to *mask* the original author. A successful *masking* means that an adversary has taken a program by some *original author* and modified it to conceal the true authorship. A successful *forgery* means that an adversary has fooled an attribution system into believing that some program was written by a targeted, victim author who did *not* write the program. A *forgery* attack is a targeted attack, while a *masking* attack is untargeted. Security under the latter goal (masking) implies security under the former (forgery) because an adversary able to accomplish forgery will by definition be able to accomplish masking.

For example, consider an attribution system trained on five authors (A-E) with program P_a written by au-

***Corresponding Author: Lucy Simko:** Paul G. Allen School of Computer Science & Engineering, University of Washington, E-mail: simkol@cs.washington.edu
Luke Zettlemoyer: Paul G. Allen School of Computer Science & Engineering, University of Washington, E-mail: lsz@cs.washington.edu
Tadayoshi Kohno: Paul G. Allen School of Computer Science & Engineering, University of Washington, E-mail: yoshi@cs.washington.edu

thor A, and program $P_{a'}$ written by author A but modified by an adversary. An adversary launching a forgery attack with $P_{a'}$ would have the goal of $P_{a'}$ being labeled as written by a specific author, say, C. An adversary launching a masking attack with $P_{a'}$ has a goal of $P_{a'}$ being labeled as written by any of B, C, D, or E.

Our Forgery Studies. To study the capabilities and methods of an adversary with these goals, we conducted two studies with programmers; specifically, we study *forgery* attempts, as a successful forgery implies a successful masking. In our first study, the *forgery creation study*, programmers play the adversary, modifying code to fool a machine classifier. In our second study, the *forgery detection study*, we present a different set of programmers, acting as human analysts, with a series of attribution tasks over a set of code samples that include forgeries.

With the results of these studies, we then conduct *quantitative* analyses to explore the potential capabilities of an adversary against both existing classifiers and human analysts; we also conduct *qualitative* analyses to understand the potential methods of both an adversary creating attacks and a human analyst doing attribution. Finally, we analyze the modifications from the point of view of the machine classifier, to better understand the success of the forgeries and suggest features that future, robust machine classifiers might add.

Both these studies, along with our analysis of the machine classifier, give us a greater richness in our understanding of *how* people might attempt to create forgeries — as well as a baseline of their success — and provide us with insights on how one might design adversarially robust program authorship attribution systems, including systems that one might design to work with human analysts.

Contributions. We conduct the first public analysis of the robustness of a state-of-the-art source code authorship attribution system to adversarial attempts at creating forgeries. We also conduct the first, to our knowledge, public study of the efficacy of human analysts at identifying forgeries. The following contributions arise from these experiments and our resulting analyses.

- We experimentally evaluate the *robustness* of a public state-of-the-art attribution system to adversarial attempts at creating forgeries. In doing so, we find that both expert and non-expert programmers *can* successfully fool the attribution system.
- We provide a set of modifications that forgers made to create their forgeries. We also identify the strategies that human analysts (in our study) use to make attribution decisions.
- We experimentally evaluate programmers as analysts for source code authorship attribution and forgery detection. We find that forgery detection is difficult for humans as well as for machines, but that humans can adapt their strategies after being warned about adversarial input, without being specifically trained on it. From this analysis, we draw lessons for machines.
- We analyze strategies from both the forgers and the analysts, and provide recommendations to the designers of future program authorship attribution systems, in an effort to make them robust against the type of adversaries we consider.
- We investigate the effect of the forgeries on the machine classifier and, through this analysis, suggest lessons for future, more robust source code attribution classifiers

2 Related Work

Natural Language Processing: adversarial stylometry and authorship obfuscation. Authorship attribution is a well-studied topic within the Natural Language Processing (NLP) community. See [25] for a survey. In addition to attributing authorship of written text, the NLP community has recently turned to authorship obfuscation and forgery.

Kacmarcik and Gamon [15] explored the idea of purposeful anonymization in NLP, with a focus on quantifying the effort necessary to anonymize against attribution systems. Later, Brennan et al. [4] introduced *adversarial stylometry*: the notion that authors of natural language text may intentionally obfuscate (*mask*) their identity, or imitate (*forge*) another's identity. They asked a group of writers to obfuscate their own writing and then to imitate a famous author, and found that classifiers that typically had high accuracy did poorly on the obfuscated and imitated samples.

Our experiments and goals are similar to those of Brennan et al. [4], except our domain is source code instead of English. Just as code attribution research borrows ideas from NLP, our research on subverting source code attribution classifiers is both analogous to Brennan et al. [4] and novel within the security community.

Afroz et al. [1] analyzed the data from [4] to identify linguistic features that change when writers are being deceptive about their identities. This work introduced a

set of features with which they train a classifier to recognize when an author is hiding their identity. Concurrent work by Juola et al. [14] found similar results.

More recently, Potthast, et al. [20] published a survey of authorship obfuscation and imitation, as well as large scale evaluation of obfuscation techniques and detection algorithms.

Source code attribution. Compared to natural languages, authorship attribution for source code is a much younger field. Early works include Spafford and Weeber [23], Pellen [19], Kothari et al. [16], and Frantzeskou et al. [10]. Recently, Caliskan-Islam et al. [5] improved the state-of-the-art by using a combination of syntactic, lexical, and layout features and achieved higher accuracy over a much larger group of programmers than previous works. We use this classifier when testing the strength of forgeries.

Hayes and Offutt [13] suggest a number of high-level features to distinguish programmers. They find that the most distinguishing features are concerning the occurrence of operators, constructs, and lint warnings. They consider high-level algorithmic and engineering features such as testability and code coverage, but do not find that these features distinguish amongst the five programmers in their study. Our results suggest revisiting these high-level features in the context of forgeries.

Additionally, Dauber et al. [8] develop an ensemble classifier to do highly accurate attribution on short (partial) code samples.

Plagiarism detection for source code. Plagiarism detection is a sub-problem of authorship attribution. MOSS ("Measure Of Software Similarity") [22], a well-known publicly available tool, measures the similarity between two sets of source code and outputs a percent match. Forgery and plagiarism are related but also have a clear difference: in plagiarism, multiple instances of the plagiarized program are often available. In forgery and masking, the adversary creates a *new* program, intending to cause the attribution system to misattribute.

Classifying binaries. There is also work on attribution of binaries. Rosenblum et al. [21] apply machine learning to style features extracted from binaries; Caliskan-Islam et al. [6] build on this work. Muir and Wikström [17] find that changing compiler settings and linking statically can be used to decrease attribution accuracy and obfuscate authorship on binary attribution classifiers — the only other work, to our knowledge, focused on authorship obfuscation for programs.

3 Threat Model and Goals

We now elaborate on our goals and underlying threat model, beginning with several motivating scenarios.

3.1 Motivating Scenarios

Consider the following scenarios. In each case, the programmer or adversary knows about the existence of program authorship attribution systems but not the details. The adversary is not necessarily malicious, but is always acting adversarially to the attribution system.

1. **Forgery.** Suppose an adversary wants to insert malicious code into an open-source code repository. One avenue the adversary might take is to obtain the credentials of an established contributor and commit code that introduces a vulnerability. The adversary's goal is to commit code that appears to have come from the legitimate programmer so as to not raise the suspicion of the other contributors to the repository, or trigger any automated alarms — i.e., the adversary seeks to create a *forgery*.
2. **Masking.** Suppose that a programmer wants to write and release code for a new digital currency but does not want anyone to know who wrote the program. The history of the Bitcoin digital currency, and its anonymous inventor Satoshi, inspire this scenario [3]. The programmer, desiring privacy, may seek to intentionally modify their code to make it unidentifiable — i.e., to *mask* their code.
3. **Masking.** Consider a programmer developing software in a country where censorship is rampant. The programmer wants to develop software that is illegal in that country, such as anti-censorship messaging apps, and may use Tor and other network tools to obfuscate any network indicators. However, the programmer also needs to ensure that the software does not look like it was written by them — i.e., this programmer seeks to *mask* their code, to protect their anonymity. This example is inspired by Saeed Malekpour [26], an Iranian web developer sentenced to death for writing photo sharing software used to distribute pornography.
4. **Forgery.** Suppose now that an adversary wants to make it *appear* that a programmer developed anti-censorship software. The adversary, perhaps a work colleague or someone with access to software that the intended victim had written, e.g., via Git, mod-

ifies their own code to look like the victim's, and the victim is blamed. This represents a *forgery* attack.

In any of these scenarios, the adversaries may either write their own code or modify code written by others, depending on their skill levels and access to code that already fulfills their purpose.

3.2 Threat Model

As captured in the preceding scenarios, we consider two key (related) adversarial goals.

1. *Forgery.* An adversary creates a successful *forgery* if they create code such that the attribution system attributes that code to a specific *target author*.
2. *Masking.* An adversary successfully *masks* the authorship of a program if the adversary modifies a program written by some *original author* such that the attribution system does not identify the original author when given the modified program as input.

Actors. To fully understand the threat model, we define several additional parties.

- *Adversary:* the party seeking to create a forgery or mask and thwart the *attribution system*.
- *Original Author:* the person who first wrote the code modified in an attack. The original author may or may not be the *adversary*.
- *Target Author:* the victim of a forgery attack. The adversary modifies code written by the *original author* to make the *attribution system* identify the target author. We envision adversaries who are *not* the target authors.
- *Attribution System:* one or more classifiers that determine the authorship of the code. Inspired by prior works, we typically use this term to refer to a machine classifier that does authorship attribution. However, as we discuss, human analysts may augment these systems.

We next elaborate on our assumptions, drawn from our goal of developing a conservative estimate of an adversary's capabilities.

Training set. First, we assume that both the adversary and the attribution system have labeled samples of code written by the original and target author (if a forgery attack). Both parties may have samples from other authors, i.e., the classifier may be trained on multiple authors. We assume the training set contains no forgeries.

Classifier. We assume that the only information available to the attribution system are the programs in question: network traces, memory dumps of machines, etc., could help detect authorship but are orthogonal to our study of source code attribution.

Adversary skill. We assume adversaries with a solid, but not necessarily expert, knowledge of the language used by the authors (C/C++ in our case), and no specific training on forgery, masking, or attribution. We think it advantageous to begin with the simple case (in this case, *experienced* but not *expert* programmers): if even these adversaries can fool a state-of-the-art attribution system, then the attribution system will likely be vulnerable to more sophisticated adversaries, and a less sophisticated attribution system will probably be more vulnerable to the same adversaries.

Adversary's knowledge of the classifier. We assume that the adversary lacks specific knowledge about the classifier in question, e.g., classification algorithm, features, size, or contents of training set. This assumption gives the classifier the greatest power and likelihood of success even in the face of an adversary.

3.3 Goals

Informed by these motivating scenarios and qualified by the bounds of our threat model, we present our goals as a series of guiding questions:

1. **Deceptibility of classifiers under adversarial input.** What is a realistic conservative estimate for the success of forgery attacks in the wild? Section 6 addresses this question using our experiments with both a state-of-the-art classifier and human analysts.
2. **Observed methods of forgery creation.** What modifications might adversaries make to code to create forgeries? Section 7.1 explores the forgery creation process to inform the design of future attribution systems with increased resiliency to forgery.
3. **Observed methods of forgery detection and authorship attribution**. What features might human analysts look for when making attribution decisions? Are these sufficient for attribution when forgeries are *not* present? When forgeries *are* present? Could these features be potentially adopted by machine classifiers? Section 7.2 explores the forgery detection insights gleaned from human analysts.

4. **Effect of forgery on machine classifier features.** How do the code modifications made by forgers affect the features that a machine classifier might examine? Section 8 addresses this by comparing the machine classifier's view of original files and their corresponding forgeries.

4 Classifier and Data

Before discussing our methodology, we give background information on the classifier we use and our data set.

State of the art classifier. To facilitate the exploration of our goals, we needed a classifier against which to test forgeries. Industry attribution systems, to the extent that they exist, are proprietary, with little public knowledge of their capabilities or methods. We therefore focused on a publicly available attribution system.

A clear choice for a state-of-the-art code stylometry classifier was the classifier introduced by Caliskan-Islam et al. [5], since it achieved remarkable increases in accuracy over prior work on source code attribution. This model performs attribution on C/C++ code by extracting lexical, layout, and syntactic features, which are derived from an abstract syntax tree of the code.

We replicated Caliskan-Islam et al.'s setup with the open source version of the classifier available at [7] and followed their directions to set up using the Code Stylometry Feature Set. As they did, we used Weka's implementation of a Random Forest with 300 trees.

Dataset. For consistency, we used the same dataset as Caliskan-Islam et al. [5], Google Code Jam (GCJ) [11], a programming competition. GCJ invites participants to code to solve a series of problems. GCJ data is well-suited for code stylometry and authorship attribution work because each program is guaranteed to be single-author and it provides a body of functionally equivalent programs as solved by different authors. However, the programs differ from much code in the wild because there are no style guidelines and the code is not meant to be maintained or used over long periods, unlike much professionally developed software. We used GCJ despite this difference because it is consistent with previous work and provided our participants with functionally equivalent programs from which to learn authors' styles.

We used C code because the classifier is built to attribute C code. The dataset consisted of 8661 programs from 3940 authors (averaging 2.2 each), but we primarily used code from the five authors with the most files: 214 total, or an average of 42.8 files each.

	$\mathcal{C}_5$	$\mathcal{C}_{20}$	$\mathcal{C}_{50}$
Precision	**100%**	**87.6%**	**82.3%**
Recall	**100%**	**88.2%**	**84.5%**

Table 1. Precision and recall for our classifiers, calculated using 10-fold cross-validation.

Data Modification. The forgery creation study aimed to understand the methods and capabilities of potential adversaries. A secondary goal was respecting participants' time. Pilot experiments showed participants created forgeries by making many tedious typographical modifications and then losing interest and motivation.

To encourage participants to focus on other modifications, we normalized typographical style to a large extent by running all code though a code linter, *astyle* [2], a simple static analyzer and code beautifier. By doing this, we support our goal of understanding sophisticated methods of forging and attribution rather than methods that would be obliterated by a linter.

Our Classifiers. We created a suite of three training sets on which this classifier achieved high accuracy in order to (1) give the classifier an advantage where possible and realistic and (2) allow us to discuss general trends over training sets.

The three training sets differ by the number of authors in the sets: 5, 20, and 50. We name them accordingly as $\mathcal{C}_5$, $\mathcal{C}_{20}$, and $\mathcal{C}_{50}$. All classifiers include data from the five authors with the most files; additional authors are chosen randomly from those with at least five files (to ensure sufficient training data). We chose the five most prolific authors because we wanted to give the classifier as much training data as possible in order to give it an advantage over the forgers. Table 1 gives the precision and recall of each classifier.

Although we mirrored Caliskan-Islam et al.'s methodology until this point, here we differ. Caliskan-Islam et al. constructed training sets that had equal amounts of code from each author in the training set to investigate how the amount of training data affects attribution success; for our goals, we wanted to have a classifier with as high accuracy as possible in order to obtain a conservative estimate of adversarial success. We achieve this by not withholding training data from any author and creating forgeries of the five most prolific authors. We did not control the number of training files per author, as downsampling would not improve classification accuracy or fairness.

Additionally, we remove all programs that we gave participants as a possible starting point for forgeries,

a small subset of all code by each author. We remove these files from the training sets because these *original files* are similar to the forgeries created.

5 Methodology

This section explains our design decisions for the studies and presents a summary of participants' demographics.

We refer to the author or authors that we asked a participant to focus on as 'X', 'Y', and 'notX'. Recall that there are five possible authors, A-E, so five possible values for each X and Y. 'notX' takes on the value of the other four authors; for example, if 'X' is author B, then 'notX' is the combination of A, C, D, and E. Additionally, participants in the forgery creation study compared different author pairs: some compared authors A and E while others compared B and C (and so on).

Running studies that explored all possible combinations of authors and participant skill with the number of choices of code to forge and attribute would have proven prohibitively large. Given the lack of prior work on code style forgery, our intentions were two-fold: 1) develop a conservative estimate of adversarial capabilities by measuring forgeries against a state-of-the-art machine classifier, and, 2) qualitatively analyze the methods used to forge and attribute, as well as the effects they had on the features extracted by the machine classifier.

We designed and conducted both studies with the ethical considerations required when working with human participants. Our institution's human subjects institutional review board approved both studies. Neither study asked participants to incur any risk greater than normal while programming or reading code.

5.1 Forgery Creation Study Design

In our forgery creation study, we conducted sessions with 28 programmers, who produced 29 forgery attempts (one participant misunderstood the instructions and created two forgeries). We excluded two forgeries because they differed dramatically from the original code in functionality, which was explicitly against the instructions. Thus, we have 27 total forgeries.

The tasks in this study were:

1. **Train.** Given code from two authors, X and Y (chosen from authors A-E), compare functionally equivalent code by both authors to learn the authors' styles. Participants were not told how to think about style, but they did know that subsequent tasks would ask them to attribute and forge.
2. **Attribute.** Attribute 10 new samples of code to either X or Y. We asked participants to: (1) give an attribution decision, (2) indicate their confidence on a scale of 1-10, (3) briefly state their reasoning.
3. **Forge.** Modify code written by one author to look like it was written by the other author. We asked participants to imagine the best possible classifier: human, machine, or some combination thereof. Participants chose whether to forge X's style or Y's. In this task, we learn the level of success after participants *think* they have made enough transformations to fool a classifier — i.e., the capabilities of an adversary with no information about a classifier.
4. **Forge with Oracle.** Test the forgeries from the previous step against the classifiers. Participants then continued to modify their forgeries until all versions of the classifier produced the target misclassification, or until they chose to stop. The purpose of this task was to encourage participants to continue making more transformations if their first forgeries did not fool all the classifiers.

5.2 Forgery Detection Study Design

Our forgery *detection* study used the forgeries created in our forgery *creation* study (Section 5.1) to explore both the success of programmers as attribution analysts and the methods used to successfully detect forgeries.

This study had 21 participants. We gave the participants no specific instructions concerning attribution or forgery detection, since we did not want to bias their attribution methods with our preexisting knowledge of the forgeries. At first, participants were not aware that this study was about forgeries.

Participants completed the following tasks, in which they acted as an analyst looking to detect forgeries of a certain author, X; X varied between authors A-D (we did not receive enough forgeries of E):

1. **Train.** Given code from the same five authors used in the forgery creation study, with four of the authors combined to be 'notX', and the remaining author as 'X', learn the style of author 'X'. As in the forgery creation study, there were two examples of each problem, and there were no forgeries in the training set. Participants did know that there were multiple authors in the 'notX' category. We trained participants on one author versus all the others, a departure from the training phase in the forgery cre-

ation study, because we thought it would be easier for participants completing these tasks to focus on only one author's style — i.e., the one being forged.

2. **Simple attribution.** Given 12 new samples of code, answer the question "who wrote this code?" With possible answers 'X' and 'notX', random guessing would be expected to achieve 50% accuracy. Four of the 12 samples were written by X, and eight were written by one of the notX authors. Of the eight, four were forgeries of X, and four were not. We count a 'notX' label on 'forgery of X' as correct, and an 'X' label incorrect, as the task was to identify the person who wrote the code.
3. **Attribution with knowledge of forgery.** Attribute the same set of 12 samples again, with the knowledge that some may be forgeries. Possible answers were 'X', 'notX', and 'forgery of X'. In this task, random guessing would be expected to achieve 44%, as there are two correct labels for a forgery of X: both 'notX' (the original author) and 'forgery of X' (a forgery detection) are correct, since either would be a win from a classifier.
4. **Find the forgery.** Given 3 or 4 pairs of functionally equivalent programs side by side, one written by author X and the other a forgery of author X, decide which is the forgery.

5.3 Recruitment and Demographics

Recruitment. For both studies, we required participants to self-report a good working knowledge of C/C++. As we elaborate below, participants in the forgery-creation study were a mix of undergraduate Computer Science students and current or former professional software developers. Participants in our forgery detection study were primarily upper-level undergraduates and graduate students in Computer Science. No one participated in both studies.

Demographics. Of the 28 forgery creation participants, 10 were students, 16 were professional computer scientists, and two were unemployed or had an occupation unrelated to computer science. Of the 21 forgery detection participants, 14 were students, and three were professional computer scientists.

We also asked participants to report experience as a professional C/C++ programmer. Although the forgery creation participants were, in general, more experienced with programming, we can compare the relative skill levels of the two groups at attribution specifically by comparing their performance on the two attribution tasks: the attribution task in the forgery creation study (Step 2 in Section 5.1) was nearly identical to the simple attribution task in the forgery detection study (Step 2 in Section 5.2), if we count attribution decisions on only non-forgeries. Despite their demographic differences, both groups of participants attained high accuracy on these tasks, suggesting that they were similarly skilled at thinking about programming style.

6 Quantitative Results

We now turn to our results regarding the capabilities of both the participants in the forgery study (the forgers) and the participants in the forgery detection study (the analysts). Broadly, we find that *the forgeries created by our participants, who had no specific training in forgery creation, are largely successful against machine classifiers and human analysts. However, when analysts are told to look for forgeries, forgery success decreases rapidly, suggesting that future, robust attribution systems should learn from the strategies that the human analysts developed.* In this section, we refer to *success* as the success of the forgery attacks, not the success of the classifier.

6.1 Deceptibility of machine classifier

We find that the current state-of-the-art machine classifier [5] — though it achieves high accuracy on large numbers of programmers — is not robust to forgery attempts. The ability of adversaries to defeat this classifier is perhaps unsurprising given that the classifier was not designed to be robust against adversaries. However, we find it informative to study whether it actually fails with adversarial input in practice and (later) evaluate both how people created those forgeries and how the forgeries affected the machine classifier features in order to inform the design of future, adversarially-robust classifiers.

Attack type	Initial attack success	Final attack success
Forgery	61.1%	70.0%
Masking	73.3%	80.0%

Table 2. The percent of attempts that were a successful attack on both the forgers' initial and final attempts.

Attack type	$\mathcal{C}_5$	$\mathcal{C}_{20}$	$\mathcal{C}_{50}$	Average
Forgery	66.6%	70.0%	73.3%	70.0%
Masking	76.6%	76.6%	86.6%	80.0%

Table 3. Breakdown of attack success of *final* attempts against the machine classifiers.

Task	Simple attribution	Attribution with knowledge of forgery	Find the forgery
Forgery attack success	56.6 %	23.7%	53.1%

Table 4. The percent of forgeries that were a successful attack in each phase of the forgery detection study.

Initial forgery attempts. Table 2 shows the percent of forgery attempts that yielded successful forgeries or masks against the machine classifiers. On the forgers' first try, 61.1% of forgery attempts were successful forgery attacks, and 73.3% of forgery attempts resulted in successful masks. (Recall that forgery is a subset of masking, so successful mask attempts include all successful forgery attempts).

Final forgery attempts. Table 3 shows that 70.0% of final forgery attempts were successful, and 80.0% of final attempts were successful as masking attacks. That is, for 80.0% of final forgery attempts, the classifier did not identify the original author, meaning that the average accuracy of the classifier on the forgeries was 20.0%.

Training data variation: number of authors. The percentage of successful attacks increases with the number of authors in the classifier's training; in our data, the percent of successful forgery attacks increases from 66.6% on $\mathcal{C}_5$ to 73.3% on $\mathcal{C}_{50}$. Similarly, masking attacks increase from 76.6% to 86.6% on the same classifiers. We expect that as the number of authors in the classifier's training set rises, the ease of masking would continue to rise, while the ease of forging would depend on the strength of the classifier's model of the target author.

Training data variation: presence of unmodified forgeries. Recall, from Section 4, that the training sets did not contain the *original files* from which the forgeries were created. We also test the forgeries against three classifiers trained on the same authors, but with the addition of the original forgery files, so the classifiers have access to all authors' entire body of work. Intuitively, forgeries have a lower level of success when attacking these versions of the classifier: forgery success is 65.6% overall, and masking is 73.3% (down from 70.0% and 80.0%, respectively). This implies that the classifier has a better chance of recognizing the original author if the actual original file is in its training set.

6.2 Success against human analysts

The results of our forgery creation study, detailed in Section 6.1, reveal that an adversary can deceive the state-of-the-art machine classifier. Our forgery detection study finds that *while our participants do not spot forgeries when given no information at all, they can develop successful forgery detection strategies without examples of forgeries or instructions about forgery creation.* As above, success refers to the forgery attack success, *not* the success of the participants.

Recall, from Section 5.2, that in the forgery detection study, participants had no knowledge of the forgeries in the *simple attribution* task, and later knew that there *might* be forgeries present in the next task, the *attribution-with-knowledge-of-forgery* task. The final task, *find-the-forgery*, presented participants with two functionally equivalent programs, and instructed them to determine which one was the forgery. Table 4 summarizes the rate of success of forgery attacks in each of the attribution tasks.

We find that attack success decreases dramatically in the second task, after participants are told about the possibility of forgeries. We also find that humans and machines are not always fooled by the same forgeries.

Simple attribution task results. In the *simple attribution* task, the forgeries fooled the humans 56.6% of the time, meaning that the participants' accuracy on forgery was statistically equivalent to random guessing. This indicates that in this task, when participants had no reason to expect that some of the programs were forgeries, they were often fooled by the forgeries (as was the machine classifier). The attribution accuracy on non-forgeries attained by the participants, 92.7%, is high.

Attribution-with-knowledge-of-forgery task results. In the *attribution-with-knowledge-of-forgery* task (Step 3 in Section 5.2), forgery attack success decreases drastically, to 23.7%. That is, participants were able to correctly detect 76.3% of forgeries. This dramatic drop in forgery success shows that when analysts are made aware of the possibility of forgery, they are more likely to accurately detect or attribute forgeries. However, although many forgeries were correctly detected, a number of code samples attributed as forgeries were not ac-

tually forgeries: where previously the analysts correctly attributed non-forgeries with 92.7% accuracy, they only achieved 76.2% accuracy on non-forgeries in this task. This can be attributed to increased suspicion about inconsistencies in every program, as participants noted.

Find-the-forgery task results. The third and final task of the study presented participants with two functionally equivalent programs, one written by X, and the other a forgery attempt of X solving the same problem. Success on this task was 53.1% overall, statistically equivalent to random guessing (50%). However, the results were not random: some forgeries consistently fooled the humans while others tended to be detected. Furthermore, although some forgeries in this task fooled the participants, they were not strictly the same forgeries that fooled the machine classifiers, or that fooled the participants in the previous task.

6.3 Lessons

Our results enable us to draw the following lessons regarding the capabilities of forgers and analysts:

1. The current state-of-the-art attribution classifier [5] is not robust to adversarial stylometry attacks, even when those attacks are created by non-experts. This suggests that other source code attribution classifiers may be vulnerable to the same type of attacks.
2. Training sets matter: classifiers trained on fewer authors are more robust, and classifiers that have the original, unmodified files in their training sets are more robust. This suggests having a variety of training sets on which to test new code.
3. Augmenting attribution systems with human analysts, though expensive, may increase the success of the overall system's forgery detection and attribution capabilities. Attribution analysts are subject to the same attacks that machine classifiers are, but they *may* be sensitive to different attack strategies.
4. Telling the analysts about the potential adversarial input was enough to significantly increase forgery detection, though the increased suspicion caused non-forgeries to be marked as forgeries. Attribution systems employing human analysts should consider how to train analysts to increase the rate of forgery detection without decreasing attribution accuracy.
5. Some forgeries fooled all the machine classifiers and many or all of the humans. Others fooled all the machine classifiers and none of the humans, and others still fooled many or all of the humans, but none of the machine classifiers. Forgeries that fool *all* the classifiers, human and machine, are the best forgeries. However, the small number of forgeries the fool either humans or machines but not both suggests two things: (1) that as the quality of the forgery declines, it may only fool some of the classifiers, but we cannot say which ones; and (2) some forgeries are easier for different classifiers or people to recognize.

7 Qualitative Results

To understand how to improve forgery detection, we first analyze forgeries to understand the changes forgers made. Second, we analyze the participants' notes from the attribution phases in both studies to understand how they thought about author style, attribution, and forgery detection.

7.1 Forgery Creation

Implementation involves three types of decisions: typographic, control structure, and information structure [12, 18]. We further divide modifications into *local*, or implementation-level changes, and higher-level *algorithmic* changes. These categories exist on a continuum, but we define them roughly as follows: if one were implementing a program from pseudocode, a local modification would not be visible in the pseudocode, whereas an algorithmic change would be a departure from the pseudocode. For example, imagine pseudocode for a nested loop, along with the accompanying source code. In the source code, modifying the *type* of loop — i.e., `for` to `while` — would be a *local control flow modification*, since it does not affect the pseudocode, but control flow keywords are modified. Abstracting the inner loop to a helper function would be an *algorithmic control flow modification*, since it would affect the underlying pseudocode.

Type of change	Total modifications	Forgers who made this change
Var. names	168	25
Lines of code copied	113	23
Libraries imported	53	22
Indent scheme	1024	19
Var. decl. location	85	19

Table 5. Common modifications made by forgers

We find that *modifications made to create successful forgeries are overwhelmingly local control flow, local information structure changes, and typographical changes, and that the vast majority of participants did not make algorithmic modifications at all.*

The current state-of-the-art classifier [5] achieves high accuracy on non-adversarial input. However, our participants — without knowing what features the classification algorithm was using — modified code in such a way that they were able to fool the classifiers using *only* these local modifications, suggesting that local code features are not sufficient for an attribution system that may receive adversarial input. Section 8 explores the how these local code modifications affect the machine classifier features.

Next, we detail the types of changes made to forgeries, citing examples from the studies. First, we present an overview of the most common types of modifications; then we explore specific changes using the examples shown in Figures 1, 2, 3, and 4. Appendix A contains the entire list of modifications.

[**Original code**]

```
for(i=0;i<a1-1;i++) printf("%d",r1[i]);printf("%d\n",r1[i]);
for(i=0;i<a2-1;i++) printf("%d",r2[i]);printf("%d\n",r2[i]);
```

[**Modified code (forgery)**]

```
for (i = 0; i < cmx - 1; i++)
    {
        printf("%d ",curmx[i]);
    }
    printf("%d\n", curmx[i]);

for (i = 0; i < cmy - 1; i++)
    {
        printf("%d ", curmy[i]);
    }
    printf("%d\n", curmy[i]);
```

Fig. 1. Original and modified code, showing typographical and information structure modifications

7.1.1 Most Common Changes

Table 5 shows the most popular modifications, ranked by the number of participants who made at least one such modification. Changing variable names, variable declaration location, and libraries imported are local information structure modifications (Section 7.1.3), and modifying the indentation scheme is a typographical change (Section 7.1.2).

Copying entire lines of code written by the target author in the training set may be either a control flow or an information structure change: often, participants copied the logic of the main loop (control flow modification) and variable declarations for variables used globally or at a high level in the main function (information structure modification), suggesting that participants considered file- or function-level variables and loop logic and structure an essential marker of style.

7.1.2 Typographical Changes

Recall from Section 4 that the code given to participants had been normalized typographically to a large extent. Many participants still made some typographical changes, though some typographical modifications may not have been intentional if participants were working in an IDE that enforced a certain style. There is no way to determine which modifications were not deliberate, so we include all modifications in our analysis.

The most common typographical modifications were the usage or location of brackets and the indentation scheme, while less common modifications were comments and spacing between operators. Figure 1 illustrates some of these modifications. In this original-forgery pair, the forger added **brackets**, **newlines**, and **spaces between operators** to transform the code to look more like the target author's code. These modifications make the code look significantly different at first glance, but in reality these modifications could be easily automated with a code linter.

7.1.3 Information Structure Changes

Information structure changes cover variables, syntax, and data flow. Local modifications include changing the variable name or declaration location, ternary operator usage, and macro addition and removal. Algorithmic changes, which *no* participants made, would include modifications to the variable type, data structure usage, and static and dynamic memory usage.

Nearly all participants modified variable names and the location of variable declarations, typically either from or to a global variable, or inside a `for`-loop instantiation. Participants also frequently added or removed macros, libraries included (more often added), and swapped equivalent API calls, typically for I/O. Less common modifications were syntactic, such as ternary operator usage, the increment/decrement operator, and array indexing.

[**Original code**]

```
43 int main()
44 {
45  int i,j,k;
46  int cc,ca;
47  cin >> ca;
48  for(cc=1;cc<=ca;cc++)
49  {
50   cin >> D >> I >> M >> N;
51   for(i=0; i<N; i++)
52      cin >> original[i];
```

[**Modified code (forgery)**]

```
1   #define REP(i,a,b) for(i=a;i<b;i++)
2   #define rep(i,n) REP(i,0,n)
    ...
41  int main()
42  {
43     int i,j,k;
44   int size, count = 0;
45   scanf("%d", &size);
46   while (size--)
47   {
48     scanf("%d%d%d%d", &D, &I, &M, &N);
49     rep (i,N) scanf("%d", original+i);
```

Fig. 2. Information structure and control flow modifications

The transformation in Figure 2 illustrates changes to **variable names**, **API calls** (specifically, this occurred with I/O functions), and the addition of **macros** used by the target author. Figures 1 and 2 show modified **variable names**.

7.1.4 Control Flow Changes

Control flow modifications relate to the path of execution in a program. Local modifications include modifying the loop type, loop logic (i.e., increment to decrement), putting multiple assignments per line, breaking up a complex `if`-statement, and adding or removing control flow keywords. Algorithmic modifications include refactoring functions and any major addition or removal of control structures like loops and conditionals.

The majority of the control flow changes to forgeries are implemented as local modifications. Most common are changes to loop type or logic, breaking up an `if`-statement, and the addition or removal of control flow keywords (with minimal modifications to the surrounding algorithm). We do find some algorithmic changes, though they are much less common than the local modifications: some participants added or removed inlined versions of `memset` or common math functions. One participant undertook a major refactoring of a helper function; we highlight this unusual but impressive series of modifications after discussing local modifications.

```
8   int judge(int m) {
9    int j,k;
10   for(j=0;j<m;j++) for(k=0;k<m;k++) if (b[j][k]!=-1) {
11     if (b[k][j]==-1) b[k][j]=b[j][k]; else if
      (b[j][k]!=b[k][j])
   return 0;
12     if (b[m-1-j][m-1-k]==-1) b[m-1-j][m-1-k]=b[j][k]; else
      if (b[m-1-j][m-1-k]!=b[j][k]) return 0;
13     if (b[m-1-k][m-1-j]==-1) b[m-1-k][m-1-j]=b[j][k]; else
      if (b[m-1-k][m-1-j]!=b[j][k]) return 0;
14   }
15   return 1;
16  }
17
18   int cal(int m) {
...
21        memset(b,0xff,sizeof(b));
```

Fig. 3. Original code: algorithmic changes. The forger later refactored the function, `judge`

Control flow: local changes. Figure 2 illustrates local control flow modifications. Most prevalent is the addition or removal of control flow keywords, like `break`, `continue`, `assert`. This may be because it is often trivial to add or remove them by locally modifying the logic.

The modified code snippet in Figure 2 begins by adding **macros** for `for`-loops that the imitated author frequently used. Then, the **loop type** of the inner `for`-loop (original line 51, modified line 48) is changed to use the macro. The modified version also **removes a newline** so that both the loop initialization and body are on the same line, without brackets. The **loop type of the main loop** (original line 48, modified line 46) is changed from a `for`-loop to a `while`-loop. The **logic of the main loop** is also modified: in the modified version, the counter decreases; in the original version, the counter increases. The **input functions** at original lines 47/50 and modified lines 45/48 have been changed to use `scanf` instead of `cin`. Finally, some **entire lines** (44-46) in the modified version have been copied from programs written by the target author.

Control flow: algorithmic changes. Control flow changes that modify the underlying algorithm reflect a deeper understanding of both the original and target programmers' thought processes and goals. Only a few participants made these types of changes — for example, refactoring a function, replacing a `goto` with a non-trivial series of conditionals, or adding or removing inlined version of common math functions. Figures 3 and 4 show an example of inlining: the participant removes a call to `memset` (original line 21) and replaces it with an **inlined** version (modified line 41), a double loop through the array where each element is set to 256, as in the original version. This type of modification occurs in a few other forgeries and also signifies a more

```
2  #define REP(i,a,b) for(i=a;i<b;i++)
3  #define rep(i,n) REP(i,0,n)
...
10  int copy_maybe(int x, int y, int c, int d) {
11  if (b[x][y]==-1) {
12      b[x][y] = b[c][d];
13    }
14    else
15      {
16        if (b[x][y] != b[c][d]) return 0;
17      }
18    return 1;
19  }
20
21  int judge(int m) {
22      int j,k,id,ie,r;
23      rep(j,m) rep(k,m) if (b[j][k]!=-1) {
24       r = copy_maybe(k,j,j,k);
25    if (r == 0) return 0;
26    id = m - 1 - j;
27    ie = m - 1 - k;
28    r = copy_maybe(id,ie,j,k);
29    if (r == 0) return 0;
30    r = copy_maybe(ie,id,j,k);
31    if (r == 0) return 0;
32   }
33   return 1;
34  }
35
36 int calculate(int m) {
...
41      rep(o,256) rep(p,256) b[o][p] = MAX;
```

Fig. 4. Algorithmic changes to a forgery: the forger refactored some logic into a new function, `copy_maybe`

significant change to the path of execution, though it only requires only a local understanding of the code.

A more unusual, higher-level algorithmic modification in Figures 3 and 4 is the addition of a helper function, showing deep analysis of the algorithm. To create this forgery, the participant made significant changes to the function `judge`, primarily by adding **temp variables** and abstracting away much of the logic in the complex `if`-statements in the original code (lines 11-13) to a **new helper function**, called `copy_maybe`. These changes would require a forger to understand the function as a whole. This is the *only* forgery that shows a forger delving slightly deeper in the design choices made by both the original and target authors. The participant who created this forgery wrote that *"it required understanding the code more deeply than just at a stylistic level and trying to rewrite it from scratch."*

7.2 Features Used for Attribution

Through analysis of our study results, we find that analysts focus primarily on local implementation details instead of algorithmic decisions when making attribution decisions, just as the forgers focused on these when creating forgeries. That is, *participants overwhelmingly use the same set of features when trying to detect forgeries, but augment this set with higher-level reasoning about stylistic inconsistencies and the forgeability of features.*

These can be roughly divided into inconsistencies regarding the *presence* of a feature and inconsistencies regarding *use* of a feature. For example, imagine an author typically uses the variable name `aa` in their main loops, as seen in the training set. In a forgery, a presence inconsistency would be the lack of a variable named `aa`. A usage inconsistency would be a variable named `aa` used in a helper function and not in the main loop. Noticing a usage inconsistency requires the analyst to have a greater understanding of the author's style.

Presence of features. The inconsistent inclusion or exclusion of a feature indicative of the target author led participants to label code a forgery. Although noting these inconsistencies sometimes resulted in a correct label, it was not a reliable technique, as we discuss below.

A common reason for a correct forgery label was inconsistencies in spacing and bracket use. The success of these low-level typographic inconsistencies (in combination with the other features) may be attributed, to some extent, to the fact that style in the training set was largely normalized, but we made no normalizations to the forgeries because they reflect the forgers' idea of what will fool a classifier. So, if the forger did not attempt to imitate typographic style at all, the participant may have discerned typographical differences.

Sometimes, however, typographical inconsistencies lead to an incorrect label: participants wrote, about non-forgeries, that *"everything matches except there is a* `for` *loop missing its braces"*; and *"the discrepancies, particularly the multiple* `for`*-loop statements on the same line, make me think it's a forgery."*

Other participants justified the forgery label with anomalies in the code, or features that match or might have been copied from the target author's training set: *"The use of* `unsigned long` *is something B never did in the training data however the overall structure of the code looks like it could be from person B. Maybe this person didn't understand the difference between* `unsigned long` *and* `long` *and blindly copied over from code that used* `unsigned long`*'s?"*

Conversely, participants labeled as non-forgeries code that contained features consistently present in the training set. One participant wrote that the code *"uses __ and ___,* `asserts`*, has no inconsistencies with D's style"*, and assigned a 'X' label to a forgery of X. In both this example and the previous one, participants who based attribution decisions on the inconsistent presence of features assigned incorrect labels.

Forgeability of Features. Some participants reasoned about the fact that typographical and local features would be easy to forge but still falsely decided the code was not a forgery: "*For me this looked like D because of the* `asserts` *and curly braces, but it would be pretty easy to add those things in an attempted forgery....*" Similarly, another participant noted that the addition of simple macros would be trivial to add but assigned a non-forgery label because of the lack of unused variables. This shows that participants thought about the process of forgery creation — that is, that the addition of local and typographical features is simple — but sometimes still came to the wrong decision.

Usage of features. Some participants assigned correct forgery labels because they noticed features not used as the target author had. Higher-level features that participants used to detect forgeries include differences in modularity of the code, use of helper functions, and lines that appear in the training set, all inconsistencies in the usage of features.

A participant wrote *"the use of underscores as variable names suggests D, but normally D would only do that if the function is very short and delegates all the work to another helper function. Here, underscores are used in a very long loop, and somewhere in the middle an underscore variable is used. This seems unlike D's style. Also,* `assert.h` *is included but there are no* `asserts` *used, which seems like a sloppy imitation of D."*

This participant noted three simple features — variable names, function length, and libraries — and combined information about higher-level indicators (like typical variable names) with variable usage and information flow to discover inconsistencies in the usage of certain elements.

7.3 Lessons

Stepping back, we are now positioned to draw some key implications and lessons regarding the potential tactics and methods of forgers and analysts:

1. When creating forgeries, forgers may copy code from other programs written by the target of the forgery. This code, while visible in the source, might not actually be executed when the program is run. Therefore, an adversarially resilient attribution system might use a software analysis tool to determine which portions of code are reachable or unreachable and then treat these regions of code differently.
2. The vast majority of modifications made to create forgeries required only a local understanding of the code, and were made by forgers who had no knowledge of the features that the classifier was using for attribution. This suggests that forgeries in the wild, to the extent that they exist, might contain the same types of local modifications.
3. When told about forgeries, but without being shown examples, some participants developed reliable methods of forgery detection. These arose from the participants first considering how forgeries might be created and then developing an understanding of the target author's style at a higher level than the modifications that the forger made. This suggests that attribution system designers should aim to develop a high level understanding of authorship style.
4. Taken together, the previous two lessons suggest that adversarially resistant attribution systems might incorporate algorithmic features which require a deeper understanding of the target author's engineering process and programmatic goals.

8 Discussion

To extend the results in Sections 6 and 7, we analyze the features modified by the forgers, and then summarize our recommendations for a more robust classifier.

8.1 Machine Classifier Feature Analysis

To better understand the high rate of forgery success against the machine classifier, we explore the machine classifier's features. Each forgery has an associated original file that the forger modified to create the forgery; we compare the features and analyze how the modifications made by the forgers affected the features that the machine classifier extracted. In this section, we are specifically discussing machine classifier features, *not* 'features' used by the forgers or analysts.

Overview of classifier features. Caliskan-Islam et al's [5] classifier extracts features from an abstract syntax tree (AST) representation of the code. Leaf nodes of the AST are unigrams of code: variable names, API calls, strings, and literals (as well as C keywords). AST nodes are snippets of the syntactic structure of code, such as 'AdditiveExpression' or 'ForInit'. The classifier

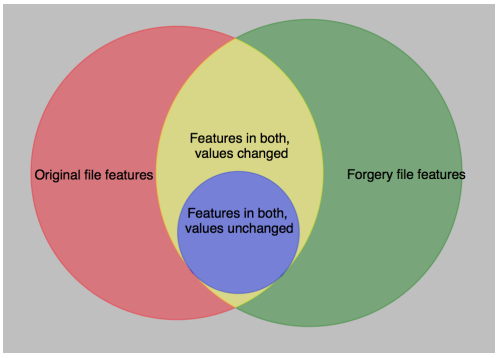

Fig. 5. This is a proportional view of the average number of features changed between the original and forgery. The large outer circles represent the original (left) and forgery (right) files. The sizes of the circles is proportional to the number of features that the machine classifier extracts. The overlap in the middle represents features that the machine classifier extracted from *both* the original and the forgery. The non-overlapped section of the left circle is 49.4% of its area, and the non-overlapped section of the right circle is 40.2% of its area. The small circle in the overlapped section represents the small proportion of features that the machine classifier extracted from both files *and* whose values did not change in either file: it is 43.0% of the overlapped portion.

traverses the AST to extract AST unigrams and bigrams. For an in-depth explanation, see [5].

On average, for classifier $\mathcal{C}_5$, 55.0% of features include a variable name, API call, macro, or literal. By making small, local changes to *only* variable names, macros, literals, or API calls, forgers had access to over half of the features. Recall from Section 7.1 that swapping variable names was the most common modification that forgers made, and they also commonly modified macros and API calls: these modifications had the potential to alter half the features extracted from the file.

Overview of changes to features. Figure 5 shows an overview of the proportion of features shared between the original and the forgery. On average, 49.4% of features extracted by the machine classifier from original file are absent in the corresponding forgery, meaning that the forgers' modifications completely removed any instances of nearly half of the features. Similarly, 40.2% of the features in a forgery do not exist in the original file. The type of features that exist in only one file are proportional to the overall feature makeup, meaning that forgers affected both a large percentage and variety of features: although forgers primarily made modifications to originals that did not require a high-level understanding of the authors' decisions, they were still able to affect a vast majority *and variety* of features, in line with the high rate of forgery success against the classifier.

The small circle contained in the overlapping middle portion of Figure 5 shows the proportion of features whose *values* remained the same throughout the change: 43.0% of the features that exist in both files. Combined with the features that were added to the forgery, this means that of the features in the forgery, only 23.5% have the same value as they did in the original. So, by making primarily local changes, forgers created forgeries that contained 76.5% features with different values.

Next, we take a deeper look at the features whose *values* change most dramatically. Adding or removing many instances of a feature changes its value more. For example, a forger who changes all `while`-loops to `for`-loops will increase the value of certain features (such as the AST node 'ForInit') more than a forger who only changes one `while` loop.

Features that exist in only one of the files. We both examine features removed *from* original files and features added *to* forgeries. Table 6 shows the most commonly added and removed features for authors A-D (there were not enough forgeries of author E). The majority of these features are variable names or macros, reflecting the modifications that forgers made to variable names and macros.

There are also AST unigrams such as 'ShiftExpression', 'UnaryOp', 'ArrayIndexing', and 'AssignmentExpr', which are typically close to the leaf nodes and thus more susceptible to local code changes. Moreover, these AST nodes indicate low level programming decisions, such as the use of a bitshift operator, or the choice to use array indexing rather than pointer addition to access array elements. There are only two AST bigrams that appear in these highly-changed features. 'ExpressionStatement AssignmentExpression' indicates assigning value to a variable, perhaps because the forger broke up multiline statements. 'ForInit IdentifierDeclStatement' indicates that the forger added variable declarations immediately after `for`-loops, another common modification. The AST bigrams sometimes capture programmatic elements across more than one line of programming, but even when they do — as with 'ForInit IdentifierDeclStatement' — they do not capture the higher level observations that participants made about the *use* of features rather than the *presence* of features.

Features that exist in both files. We now examine the features that exist in both files but have a high cumulative change in value between original and forgery files. Table 7 sorts these features by type and author. In contrast to features that only exist in one file, these con-

Feature Description	Feature	Authors
Var. name, macro, & literal unigrams	ca (avg depth), cc, w, ca, pos, color, plate, best, N, N, k, t, T, cc, rr, best, j, r, D, T (avg depth), t, c, N (avg depth), rep, size, count, tar, ll, mode, in, rep (avg depth), tm, size (avg depth), r, res, lim, ___, _, n, s, R, y, 2, tt, cmy, curmy, tn, MAXN, res	A, B, C, D
Var. name bigrams	t T, i n	B
API calls unigrams	cin, cin (avg depth), scanf_s, scanf, assert	A, B, C, D
AST unigrams	ShiftExpression, MultiplicativeExpression, cc (avg depth), ShiftExpression (avg depth), AndExpression, UnaryOp, AssignmentExpr, ArrayIndexing	A, C, D
AST — AST bigrams	ExpressionStatement AssignmentExpr, ForInit IdentifierDeclStatement	A, B

Table 6. These are features that most often occur in either the original by an author, or the forgery of that author, but *not* both.

Feature Description	Feature	Authors
Var. name, macro, & literal unigrams	i, j, 0, 1, 1	all, C, D
AST unigrams	Argument, AssignmentExpr, ExpressionStatement, Condition, AdditiveExpression, RelationalExpression, CallExpression, IncDecOp, Callee, ArgumentList, ArrayIndexing, CallExpression, Callee, ArgumentList, IncDecOp, ArrayIndexing, CallExpression, IdentifierDecl, Callee, ArrayIndexing, IdentifierDecl, IncDecOp, ForInit	all, A, B, C, D
C keyword unigrams	int	all

Table 7. These features often occur in both a forgery *and* its corresponding original, with the highest cumulative change in value.

tain no leaf nodes other than common variable names and literals used by most programmers. The majority of the features are AST unigrams.

Two of the AST nodes are typically relatively high in the AST: 'ExpressionStatement' and 'Condition'. A change in the value of 'ExpressionStatement', a relatively generic AST node, could indicate that the forger has expanded or contracted lines of code, or simply changed the length of the program, in part by adding or removing lines of code that are expression statements. Forgeries were commonly different lengths than the corresponding original files. A change in the value of 'Condition' may reflect that some forgers trivially added or removed `if`-statements.

One of the most popular AST unigrams, 'Argument' occurs for each argument in a function call, so its value will change if the forger uses equivalent API calls that have a different number of arguments, another common modification we observed. The rest of the AST node names are self explanatory and reflect code modifications that occur on one line only.

Combined with the features that occur in only one file, we observe that the features highly affected by forgers' changes are variable names and AST unigrams, typically nodes close to the leaf nodes. As shown in Figure 5, these are 76.5% of the features in the forgery. We now briefly turn our attention to the 23.5% of features that typically do not change in value between the original and the forgery.

There are only two features that occur in all original-forgery pairs and never change value. They are: AST — AST bigram 'FunctionDefinition CompoundStatement', and 'Max depth of AST leaf'. There is a long tail of features that stay constant in the one file in which they appear, primarily AST — AST and leaf — AST bigrams. Along with the high rate of feature modification, this long tail indicates that *the machine classifier features are not unforgeable*, and that a future, more robust machine classifier should extract features that are more difficult for forgers to modify.

8.2 Lessons

Based on our analysis of how the forgers affected the machine classifier's features, we draw the following lessons.

1. Over half of the machine classifier features included variable names, macros, literals, or API calls. Because these are typically simple to change by hand — and even to automate — we suggest that future classifiers should consider fewer of these features, or that these features could be contextualized with their usage in the program, as successful human analysts did in Section 7.2.

2. AST bigrams can capture some information about higher level programming decisions. Although these are not unforgeable, they may be a valuable part of a more robust classifier.
3. Combined with our finding from Section 7.3 that thinking about the *forgeability* of code properties helped analysts correctly label forgeries, we reiterate that the unchanged machine classifier features are *not* unforgeable and that the same common strategies that changed other features could modify them. This suggests that future, more robust classifiers may benefit from extracting *new* features that are not affected by the local modification strategies that the forgers employed, for example by including more complex AST features or by including forgeries (correctly labeled) in the training set.

9 Conclusion

We conduct two studies with C/C++ programmers in order to better understand the potential capabilities and methods of (1) adversaries creating forgery attacks, and (2) machine classifiers and human analysts trying to attribute source code which may have been modified by an adversary.

Through our forgery creation and detection studies, we find that programmers not specifically trained in attribution or forgery *can*, in many cases, fool a state-of-the-art source code attribution system. This work therefore provides a conservative estimate of the success of more skilled or trained adversaries.

We find that successful forgeries primarily contain modifications that required the forger to have only a local understanding of the code, and that human analysts who successfully detected forgeries understood style at a higher level than the modifications, suggesting that future adversarially robust attribution systems should additionally include features that capture high-level style.

This work additionally paves the way for follow-on research, such as research in which participants modify their *own* code to look like a target's style, or research in which forgeries are included in the training set.

This work, an exploration of the capabilities and methods of adversaries seeking to forge another programmer's style, is a starting point for the design of future adversarially resistant attribution systems. The designers of such systems should consider the lessons and implications detailed here, and build systems resilient to even more sophisticated forgery and masking attacks.

10 Acknowledgements

We are very grateful to Camille Cobb, Karl Koscher, Kiron Lebeck, Anna Kornfeld Simpson, Paul Vines, Karl Weintraub, and Eric Zeng for helping us pilot these studies and for their feedback on our paper drafts, and to Sandy Kaplan, Franziska Roesner, Brian Rogers, and Alison Simko for their feedback on drafts of this paper. We also are extremely thankful for Yejin Choi's early discussions on this project, and her feedback on our final paper.

We would also like to thank Aylin Caliskan and Rachel Greenstadt for their personal help setting up their classifier, and for some early conversations about research directions. Finally, a sincere thank you to our anonymous reviewers for their constructive feedback.

This work was supported in part by an NSF Graduate Research Fellowship and the Short-Dooley Professorship.

References

[1] Afroz, Sadia, Michael Brennan, and Rachel Greenstadt. "Detecting hoaxes, frauds, and deception in writing style online." IEEE Symposium on Security and Privacy. IEEE, 2012.

[2] astyle.sourceforge.net

[3] S, L. "Who Is Satoshi Nakamoto?" The Economist. The Economist, 02 Nov. 2015. Web. 28 Feb. 2017.

[4] Brennan, Michael, Sadia Afroz, and Rachel Greenstadt. "Adversarial stylometry: Circumventing authorship recognition to preserve privacy and anonymity." ACM Transactions on Information and System Security (TISSEC) 15.3 (2012): 12.

[5] Caliskan-Islam, Aylin, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. "De-anonymizing programmers via code stylometry." 24th USENIX Security Symposium (USENIX Security 15). 2015.

[6] Caliskan-Islam, Aylin, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. "When coding style survives compilation: De-anonymizing programmers from executable binaries." arXiv preprint arXiv:1512.08546 (2015).

[7] https://github.com/calaylin/CodeStylometry

[8] Dauber, Edwin, Aylin Caliskan-Islam, Richard Harang, and Rachel Greenstadt. "Git blame who?: Stylistic authorship attribution of small, incomplete source code fragments." arXiv preprint arXiv:1701.05681 (2017).

[9] DeMillo, Richard A., Richard J. Lipton, and Frederick G. Sayward. "Hints on test data selection: Help for the practic-

ing programmer." Computer 11.4 (1978): 34-41.
[10] Frantzeskou, Georgia, Stephen MacDonell, Efstathios Stamatatos, and Stefanos Gritzalis. "Examining the significance of high-level programming features in source code author classification." Journal of Systems and Software 81.3 (2008): 447-460.
[11] https://code.google.com/codejam/
[12] Gray, Andrew, Stephen MacDonell, and Philip Sallis. "Software forensics: Extending authorship analysis techniques to computer programs." (1997).
[13] Hayes, Jane Huffman, and Jeff Offutt. "Recognizing authors: an examination of the consistent programmer hypothesis." Software Testing, Verification and Reliability 20.4 (2010): 329-356.
[14] Juola, Patrick. "Detecting stylistic deception." Proceedings of the Workshop on Computational Approaches to Deception Detection. Association for Computational Linguistics, 2012.
[15] Kacmarcik, Gary, and Michael Gamon. "Obfuscating document stylometry to preserve author anonymity." Association for Computational Linguistics, 2006.
[16] Kothari, Jay, Maxim Shevertalov, Edward Stehle, and Spiros Mancoridis. "A probabilistic approach to source code authorship identification." Information Technology, 2007. ITNG'07. Fourth International Conference on. IEEE, 2007.
[17] Muir, Macaully, and Johan Wikström. "Anti-analysis techniques to weaken author classification accuracy in compiled executables." (2016).
[18] Oman, Paul W., and Curtis R. Cook. "A taxonomy for programming style." Proceedings of the 1990 ACM annual conference on Cooperation. ACM, 1990.
[19] Pellin, Brian N. "Using classification techniques to determine source code authorship." White Paper: Department of Computer Science, University of Wisconsin (2000).
[20] Potthast, Martin, Matthias Hagen, and Benno Stein. "Author obfuscation: attacking the state of the art in authorship verification." Working Notes Papers of the CLEF (2016).
[21] Rosenblum, Nathan, Xiaojin Zhu, and Barton P. Miller. "Who wrote this code? identifying the authors of program binaries." European Symposium on Research in Computer Security. Springer Berlin Heidelberg, 2011.
[22] Schleimer, Saul, Daniel S. Wilkerson, and Alex Aiken. "Winnowing: local algorithms for document fingerprinting." Proceedings of the 2003 ACM SIGMOD international conference on Management of data. ACM, 2003.
[23] Spafford, Eugene H., and Stephen A. Weeber. "Software forensics: Can we track code to its authors?" Computers & Security 12.6 (1993): 585-595.
[24] Laskov, Pavel and Nedim Srndic. "Practical evasion of a learning-based classifier: A case study." 2014 IEEE Symposium on Security and Privacy. IEEE, 2014.
[25] Stamatatos, Efstathios. "A survey of modern authorship attribution methods." Journal of the American Society for information Science and Technology 60.3 (2009): 538-556.
[26] "8 Years Later: Saeed Malekpour Is Still In An Iranian Prison Simply For Writing Open Source Software." Techdirt. Accessed February 20, 2017. https://www.techdirt.com/articles/20161005/10584235719/8-years-later-saeed-malekpour-is-still-iranian-prison-simply-writing-open-source-software.shtml.

A All Modifications

In Section 7.1 we discuss the set of modifications that participants made in our forgery creation study. In this appendix (Table 8), we provide the complete list of the different classes of modifications. This table splits modifications into three categories: control flow, information structure, and typographical. See Section 7.1 for additional discussions.

Type of change	Total number of modifications	Number of participants who made this type of change	Avg modifications per participant who made at least one
Control Flow modifications			
Control flow keywords	32	14	2.3
Loop type	48	12	4
Conditional statements	21	10	2.1
Loop logic	12	11	1.1
Functions added/removed	11	10	1.1
Multiple assignments per line	10	5	2
API calls - usage vs inlining	4	4	1
Information Structure modifications			
Variable names	168	25	6.7
Libraries included	53	22	2.4
Variable declaration locations	85	19	4.5
Macros	36	17	2.1
API calls - which one is used	67	17	3.9
Usage/placement of unary inc/dec operator	10	8	1.3
Array indexing vs pointer addition	6	4	1.5
Ternary operator usage	5	3	1.7
Typographical modifications			
Indent scheme	1024	19	53.9
Brackets - location	69	17	4.1
Brackets - use	78	13	6
Spaces between operators	628	12	52.3
Commented-out code	17	11	1.5
Comments (not code)	13	4	3.25
Other			
Entire lines of code copied	113	23	4.9

Table 8. All modifications made to forgeries