# Style Counsel: Seeing the (Random) Forest for the Trees in Adversarial Code Stylometry[*]

Christopher McKnight
Magnet Forensics
chrisdm2010@gmail.com

Ian Goldberg
University of Waterloo
iang@cs.uwaterloo.ca

## ABSTRACT

The results of recent experiments have suggested that *code stylometry* can successfully identify the author of short programs from among hundreds of candidates with up to 98% precision. This potential ability to discern the programmer of a code sample from a large group of possible authors could have concerning consequences for the open-source community at large, particularly those contributors that may wish to remain anonymous. Recent international events have suggested the developers of certain anti-censorship and anti-surveillance tools are being targeted by their governments and forced to delete their repositories or face prosecution.

In light of this threat to the freedom and privacy of individual programmers around the world, we devised a tool, *Style Counsel*, to aid programmers in obfuscating their inherent style and imitating another, overt, author's style in order to protect their anonymity from this forensic technique. Our system utilizes the implicit rules encoded in the decision points of a random forest ensemble in order to derive a set of recommendations to present to the user detailing how to achieve this obfuscation and mimicry attack.

**ACM Reference Format:**
Christopher McKnight and Ian Goldberg. 2018. Style Counsel: Seeing the (Random) Forest for the Trees in Adversarial Code Stylometry. In *WPES '18: 2018 Workshop on Privacy in the Electronic Society Oct. 15, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3267323.3268951

## 1 INTRODUCTION

*Code stylometry* [3] attempts to identify the programmer that wrote some sample of computer code through an analysis of their programming style. It achieves this by examining their source code or executable artifacts in order to discover common features that together may reveal a "fingerprint" of the author's style, such as their preference for certain logical structures, comments, naming conventions, etc. Following the Internet/Morris Worm incident in 1988 [13], a report was published that attempted to profile the author of the worm based on an examination of the reverse-engineered code [17], casting style analysis as a forensic technique.

This technique, however, may be used to chill speech for software developers. There are several cases of developers being treated as individuals of suspicion, intimidated by authorities and/or coerced into removing their software from the Internet. In the US, Nadim Kobeissi, the Canadian creator of Cryptocat (an online secure messaging application) was stopped, searched, and questioned by Department of Homeland Security officials on four separate occasions in 2012 about Cryptocat and the algorithms it employs [16]. In November 2014, Chinese developer Xu Dong was arrested, primarily for political tweets, but also because he allegedly *"committed crimes of developing software to help Chinese Internet users scale the Great Fire Wall of China"* [4] in relation to proxy software he wrote. In August 2015, the Electronic Frontier Foundation (EFF) reported that Phus Lu, the developer of a popular proxy service hosted on Google's App Engine, called GoAgent, had been forced to remove all their code from GitHub and delete all their tweets on Twitter [12]. This followed a similar incident reported on greatfire.org a few days earlier involving the creator of ShadowSocks, another popular proxy used in China to "scale the wall", known pseudonymously as clowwindy. According to the article reporting this incident, clowwindy posted a note afterwards that was subsequently removed, which said: *"the police contacted him and asked him to stop working on the tool and to remove all of the code from GitHub"* [14]. The README file for the project now simply says *"Removed according to regulations"*. Earlier in March 2015, GitHub was subjected to *"the largest DDoS that they have ever dealt with"* [2], which has been linked to the Chinese government [8] and has been suggested was an attempt to bully the site into removing repositories that contravened Chinese censorship regulations.

As the environment turns hostile towards the developers, many of them may opt to disguise their identities, and authorship attribution techniques such as code stylometry could be deployed in order to identify them from other code they may have published using their real identities. Even the threat of such techniques could be enough to instill a chilling effect in open-source contributors who otherwise may have been willing to contribute their time and effort into assisting with censorship resistance tools and privacy enhancing technologies.

Despite a substantial body of prior work on program authorship attribution, only Simko *et al.* [15] (independent and concurrent work to ours) has yet investigated how robust the techniques are to *adversarial* modifications aimed at *defending* against code stylometry by obfuscation of style or imitation of someone else's style, and how difficult or realistic this is. They carried out two significant user studies looking into the robustness of a state-of-the-art source code stylometry system [3] to human adversaries attempting to

[*]More details on this work can be found in the extended version of this paper [11], as well as in the first author's Master's thesis [10]. We thank NSERC for grants STPGP-463324 and RGPIN-03858. This work was conducted while the first author was at the University of Waterloo. This work benefitted from the use of the CrySP RIPPLE Facility at the University of Waterloo.

perform targeted mimicry attacks and non-targeted obfuscation of style. The first user study had 28 participants who were asked to modify someone else's code with the express intent of imitating a third person's style. The second study, involving 21 participants (none of whom had taken part in the first study), examined their ability to attribute forged samples to their true author, initially without knowledge of forgery and then again after being informed of the potential forgeries. In both studies, data from the Google Code Jam competition[1] were used as the corpus, taken from the dataset used by Caliskan-Islam *et al.* [3]. Their paper offers insights into the vulnerabilities of even state-of-the-art classifiers, and highlights the problems that can arise by only evaluating classification systems (of any sort, not merely authorship attribution systems) in terms of their accuracy under ordinary conditions, assuming honest actors.

We go beyond prior work by establishing a method for *automated* extraction of adversarial modifications, and creating a developer tool named *Style Counsel*[2] that assists programmers in obscuring their coding style by mimicking someone else's, much in the vein of Anonymouth [9] by McDonald *et al.* for disguising one's natural-language writing style.

### 1.1 Contributions

While there are some papers [1, 6, 9] investigating natural language stylometry from an adversarial perspective, and one [15] about the source code equivalent, we look at *automating* the process of making suggestions for altering source code to imitate the style of another author. If code stylometry is truly feasible *en masse* against real-world data, it represents a threat to the safety of individuals online and therefore defences ought to be developed to assist programmers in protecting their identities against such a threat. To this end, this work offers the following contributions; for more details, see the extended version of this paper [11]:

(1) A new set of features for capturing elements of programming style.
(2) A novel, practical, algorithm for extracting a change set from a random forest classifier in order to produce a misclassification of a particular feature vector as an alternative, known class.
(3) A tool to assist developers in protecting their anonymity that integrates with a popular IDE and is able to perform feature extraction on their source code and recommend changes to both obfuscate their style and imitate the style of another, specific individual.
(4) A pilot user study evaluating the usability of the tool, the feasibility of manually imitating another's style, and the practicalities of using the tool for this task.

## 2 STYLE COUNSEL

### 2.1 Data Collection and Feature Extraction

One of the aims of our work was to perform a realistic authorship attribution study, to discover, highlight and hopefully overcome some of the practical challenges associated with carrying out a study such as this "in the wild". All prior studies into source code attribution have used corpora derived from student assignments, textbooks and programming competitions—but none of these sources presents a corpus such as one would encounter in a real attempt at performing large-scale deanonymization. Student assignments are often relatively short, all trying to achieve the same end result, and written by individuals from very similar backgrounds (particularly with regards to their education). Code from textbooks is likely to be proofread and edited, over-commented and, from the author's perspective, a model of perfection. Code from programming competitions is likely to contain much copy-and-pasted boilerplate code, taken from their other submissions, as well as being short, uncommented and probably not following the competitor's usual style—its purpose is to solve the problem as quickly as possible; it is not intended to be production quality, readable, or maintainable.

To this end, we chose to obtain a large corpus of source code from real projects that had been published on GitHub, with the caveat that the code belong to a single author (and truly written by that person), to ensure purity of style. Selecting a popular and public source for our data ensures a wide diversity of both developers, in terms of their background and demographic, and projects, in terms of purpose and size. Our evaluations were carried out on 1,261 C repositories from 525 authors; see the extended version [11] for details of the data set collection and cleaning process.

Our target platform is the Eclipse IDE, so we wanted to integrate the task of feature extraction within the plugin as much as possible to take advantage of the rich services provided by the IDE for code parsing. The Eclipse C Development Tools provide a convenient mechanism for traversing the abstract syntax tree (AST) it constructs internally, with an abstract class containing callback methods one can implement and pass as an argument to the AST interface. Our feature set is constructed largely from this tree traversal, while specialized feature extractors are used to parse comments and preprocessor directives, which are not present in the AST.

We extracted features in the following categories; see the extended version [11, App. A] for details: **node frequencies**—the relative frequency of AST node type unigrams in the AST; **node attributes**—the relative frequency of AST node attributes, which are dependent on the node type and provide more contextual information; **identifiers**—naming conventions, average length, etc.; **comments**—use of comments, average length, ratio of comments to other structures, etc.

These categories combined to give us a total of 265 features. We purposefully exclude typographical features, such as indentation and whitespace, as these inflate the accuracy of a classifier at the cost of susceptibility to trivial attacks. Furthermore, as Simko *et al.* [15] alluded to, asking users to make many minor typographical modifications is tedious and frustrating, while there would be little research novelty in automating such changes within our tool as code formatters are already very common and would make our adversarial attacks less compelling. Instead, we invoke Eclipse's built-in code formatter in order to provide default protection for our users against the weakest attribution systems, without considering such modifications as being successful defences. We also decided against counting node bigrams as used by Caliskan-Islam *et al.* [3], or character $n$-grams, as implemented by Frantzeskou *et al.* [5].

[1]https://code.google.com/codejam/
[2]*The Style Council* is also the name of a new wave pop band formed by Paul Weller in 1983 shortly after *The Jam* had split.

Node bigram-based features result in extremely high-dimensional feature vectors, while character *n*-grams would be completely impractical for producing recommendations to the user, being made up of combinations of partial words, tokens and whitespace.

As this is exploratory work whose main purpose is to investigate defences against attribution, rather than performing attribution itself, this comprehensive but not exhaustive set of features was chosen to be representative of the features one might employ if wishing to perform authorship attribution while simultaneously being of a high enough level to be the source of meaningful advice to present to the user. Our aim is to demonstrate the feasibility of an approach to parse the model generated by a learning algorithm to automatically produce change sets for misclassification. Because of the generality of this goal, we have provided a *flexible framework* that can accommodate varying feature sets; exploring such alternative feature sets to discover those that succinctly capture an author's style, while being amenable to producing actionable advice, would be an excellent avenue for future work.

## 2.2 Making Recommendations

A significant part of our system, and crucial to its effectiveness, is the ability to make recommendations to the user on what aspects of their code they should change in order to disguise their identity as a particular target author. Our reasons for imitating a specific individual, rather than just "any" author, or "no" author (obfuscation) are as follows: first, with obfuscation the aim is to reduce the classification confidence to some target value, preferably to that of a random guess or below that of some other author. This typically would involve perturbing the feature vector to a position just outside the boundaries of that class in the feature space. A second classifier trained on the same data, or with alternative background data, may derive a different boundary that places the perturbed feature vector within the bounds of its original class. Furthermore, there is the problem of selecting which features to perturb, and by how much. Imitation of "any" author suffers from many of the same drawbacks. Granted, the direction and magnitude of perturbations is now more clearly defined (toward the nearest other author in the feature space), but if it is known that the feature vector has been perturbed, the original author could be determined by finding what classes are nearest to the feature vector's position other than the given class. Indeed, following Kerckhoffs' Principle [7], we must assume everything is known about our defences, and design a system that is secure despite this knowledge.

We have the following requirements for our system:

(1) The advice should relate to something that is possible for the programmer to change, so not refer to something that is inherent to the programming language itself, or violate syntactical rules of the language.
(2) The recommendations should not contradict one another, so not advising the user to increase one feature while simultaneously decreasing another that is strongly positively correlated.
(3) The user should be presented only with changes that contribute to the desired misclassification—either reducing confidence in their classification or increasing it in the target author.
(4) There should be a minimum of effort on the part of the user; they should be presented with the minimum set of changes required to effect a misclassification as the target.
(5) The recommendations should make sense to the user; they should be able to understand what is required.
(6) Similarly, the advice should not be too vague; there should be a clear connection between the recommendation and the content of each file.
(7) As our tool is aimed at open-source developers, we want them to be able to implement the changes without having a large negative impact on readability of the code.

Of these requirements, the first two are the most important and possibly easiest to ensure. The first equates to *correctness* and is mostly a requirement of feature selection, extraction and representation. The second requirement equates to *consistency* and refers to our ability to analyze the dataset and the relationships between features.

The third requirement equates to *relevance* and can be met by only considering features that are actually used by the learning algorithm. With random forests, a form of feature selection occurs during induction, due to the algorithm selecting the best feature/value split at each node from among a random subset of the total features. Therefore, the more important and influential features will be seen with greater probability in each tree. This gives us the ability to "rank" recommendations according to their influence on the overall (mis)classification. By only making recommendations to the user that will actually affect their classification, we can maximize the effectiveness of the plugin, and reduce the impact on the original code. The fourth requirement equates to *efficiency* and can be met by calculating some form of *effort requirement* to transform the user's feature vector according to the recommendations, which we can then use to select the set of recommendations requiring the least effort.

The fifth requirement equates to *simplicity* of communication, and can be met by using language that is familiar to programmers, but without introducing too much jargon. The sixth requirement equates to *clarity* and is mostly related to the features used. Features based on vague patterns found in the file contents that are not tied to discrete semantic objects, such as character *n*-grams rather than words, are going to be hard to relate to real content.

The final requirement equates to *non-intrusiveness*, and is the most difficult of the requirements to meet. It is dependent, to a large extent, on the person implementing the change, and how exactly they choose to do it. However, it is also dependent on the feature set and the interpretation of the classification model. As mentioned above, vague recommendations are hard to relate to real content and can result in highly intrusive changes that affect readability and other desirable aspects of the code, possibly even to the detriment of performance and correctness.

For space reasons, we leave the details of our algorithm and implementation to the extended version [11].

## 2.3 Results

We evaluated our recommendation algorithm to demonstrate that it produces correct recommendations, in terms of eliciting a misclassification as a target author. We also wish to show that features not

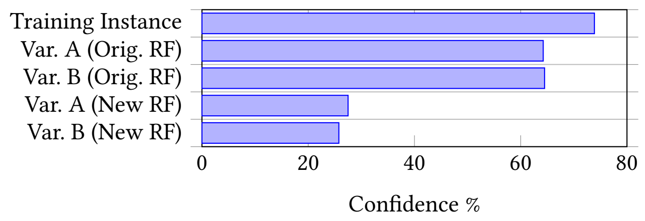

**Figure 1: Results of evaluating variation A and B feature vectors generated from recommendations. The fabricated feature vectors were evaluated by the random forest that produced them and an alternate random forest trained on the same data, using forests of 100 trees. The confidences reported here indicate how successfully the extracted intervals model the volume of feature space occupied by the training instance, with higher confidences being more successful.**

included in the recommendations do not contribute to the overall classification for that target, and can safely be ignored.

We carried out this evaluation by generating recommendations for each file in our corpus, as though the author of that file were the *target*. The values of each recommendation were used to perturb that file's existing feature vector. Two versions of these modified vectors were then produced, according to how the features with no recommendations were treated. In variation A, those features were left as their original values, and in variation B they were set to 0. If the variation A and B feature vectors return similar confidence labels, then we can say the recommendations only included the features that actually contribute to the classification. This is a desirable characteristic for our system as we want to minimize the changes we ask users to make. We then evaluated these feature vectors with both the random forest they were derived from and a second random forest trained on the same data (representing the fact that the defender aiming to hide her identity will not have access to the exact trained classifier being used by the stylometric analyst). The results of these evaluations are given in Figure 1. The confidence returned for the original training instances are averaged across both random forests, as the differences are negligible, while the confidence for the variation A and B feature vectors are averaged separately over the respective forests, as their differences are more significant. We report the *confidences*, rather than accuracy, because this more accurately reflects the closeness with which we are able to imitate the target, but note that with the relatively large number of classes in our corpus, any class receiving a confidence (i.e., votes) greater than 2% will typically become the overall label attached to that instance; a class receiving a confidence greater than 50% is guaranteed to become the overall class label. Therefore, the accuracy in our context is always strictly greater than the confidence.

Comparing the performance of the variation A and B vectors, we can see there is very little difference, which demonstrates that the subset of features used in deriving the recommendations for the user are the only features contributing to the classification. The differences in confidence between the training instance and the modified feature vectors on the original random forest are a result of the relaxing of the feature vector values from the original single point to a *volume* of the feature space encompassing a much greater number of potential feature vectors, each of which can expect to elicit a similar classification confidence from the random forest in question as its peers. Having a target volume to guide users toward instead of a single point in the feature space is far more flexible, providing our users with more options when it comes to deciding how to implement the suggestions offered by the tool, improving its usability at a cost of slightly lower overall confidence.

## 2.4 Pilot User Study

In order to help assess the usability and feasibility of our plugin, we conducted a small pilot user study in order to receive feedback from real users—extremely valuable for developing an effective tool. Our Office of Research Ethics approved our study (reference number ORE#22378).

For our pilot study, we chose participants that had C programming experience and a corpus of source code files they had authored. Three members of the CrySP (Cryptography, Security, and Privacy) lab at the University of Waterloo who satisfied these criteria volunteered for the study. Each participant was given two tasks; the first was to manually analyze another author's source code with the aim of identifying elements of their style and reproducing those elements in one of the participant's own files. The second task was to use our plugin to achieve the same goal, with a different author so as not to confer an advantage from carrying out the first task. The tasks were chosen in this order so that completion of the assisted task would not provide the user with insights into the feature set for the unassisted task. Feedback from participants indicated the automated analysis and frequent feedback on progress were the most important benefits, saving a great deal of time. The downsides given pertained to clarity and degradation of code quality. See the extended version [11] for more details and results from the pilot study.

## 3 CONCLUSIONS

Source code stylometry has been identified as a potential threat to the privacy of software developers, particularly those working in the open-source community. In addition, several recent cases have highlighted a worrying trend of governments targeting the developers of tools deemed to be used primarily for bypassing Internet censorship and surveillance. It is easy to see how these two separate phenomena could combine to threaten the safety and anonymity of current contributors, as well as push would-be contributors into silence. Alternatively, using authorship attribution has also been proposed as a means of identifying computer criminals and malware developers. Before we can reach any meaningful conclusions about its applications, however, it is important to understand its limitations with more research into its feasibility in real-world settings, its robustness in adversarial settings, and its ability to discern style from content.

To this end, we developed an algorithm for providing source code modification recommendations that will result in a successful imitation if followed, using a "human-in-the-loop" model, where it is down to the user's discretion whether and how to implement said recommendations. We presented our solution as a plugin called *Style Counsel* for the popular open-source IDE Eclipse.

## REFERENCES

[1] Michael Brennan, Sadia Afroz, and Rachel Greenstadt. 2012. Adversarial Stylometry: Circumventing Authorship Recognition to Preserve Privacy and Anonymity. *ACM Transactions on Information and System Security* 15, 3 (2012), 1–22. https://doi.org/10.1145/2382448.2382450

[2] Bill Budington. 2015. China Uses Unencrypted Websites to Hijack Browsers in GitHub Attack. https://www.eff.org/deeplinks/2015/04/china-Uses-Unencrypted-Websites-to-Hijack-Browsers-in-Github-Attack [Online; Accessed July 2018].

[3] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. 2015. De-Anonymizing Programmers via Code Stylometry. *24th USENIX Security Symposium (USENIX Security 15)* (2015), 255–270. https://doi.org/10.1145/2665943.2665958 arXiv:1512.08546

[4] China Change. 2014. Young IT Professional Detained for Developing Software to Scale GFW of China. https://chinachange.org/2014/11/12/young-It-Professional-Detained-for-Developing-Software-to-Scale-Gfw-of-China/ [Online; Accessed July 2018].

[5] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, Carole E. Chaski, and Blake Stephen Howald. 2007. Identifying Authorship by Byte-Level N-Grams: The Source Code Author Profile (SCAP) Method. *International Journal of Digital Evidence* 6, 1 (2007), 1–18.

[6] Gary Kacmarcik and Michael Gamon. 2006. Obfuscating Document Stylometry to Preserve Author Anonymity. *Proceedings of the COLING/ACL on Main Conference Poster Sessions -* (2006), 444–451. https://doi.org/10.3115/1273073.1273131

[7] Auguste Kerckhoffs. 1883. La Cryptographie Militaire. *Journal des Sciences Militaires* IX (1883), 5–83.

[8] Bill Marczak, Nicholas Weaver, Jakub Dalek, Roya Ensafi, David Fifield, Sarah Mckune, Arn Rey, John Scott-Railton, Ronald Deibert, and Vern Paxson. 2015. *China's Great Cannon.* Technical Report. Citizen Lab. https://citizenlab.ca/2015/04/chinas-Great-Cannon/, [Accessed Aug 2018].

[9] Andrew W.E. McDonald, Sadia Afroz, Aylin Caliskan, Ariel Stolerman, and Rachel Greenstadt. 2012. Use Fewer Instances of the Letter "i": Toward Writing Style Anonymization. In *Privacy Enhancing Technologies*, Vol. 7384. Springer, 299–318.

[10] Christopher McKnight. 2018. *StyleCounsel: Seeing the (Random) Forest for the Trees in Adversarial Code Stylometry.* Master's thesis. University of Waterloo. https://uwspace.uwaterloo.ca/handle/10012/12856.

[11] Christopher McKnight and Ian Goldberg. 2018. *Style Counsel: Seeing the (Random) Forest for the Trees in Adversarial Code Stylometry.* Technical Report 2018-08. CACR. http://cacr.uwaterloo.ca/techreports/2018/cacr2018-08.pdf.

[12] Danny O'Brien. 2015. Speech That Enables Speech: China Takes Aim at Its Coders. https://www.eff.org/deeplinks/2015/08/speech-Enables-Speech-China-Takes-Aim-Its-Coders [Online; Accessed July 2018].

[13] Hilarie Orman. 2003. The Morris Worm: A Fifteen-Year Perspective. *IEEE Security & Privacy* 99, 5 (2003), 35–43.

[14] Percy. 2015. Chinese Developers Forced to Delete Softwares by Police. https://en.greatfire.org/blog/2015/aug/chinese-Developers-Forced-Delete-Softwares-Police [Online; Accessed July 2018].

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

[16] Privacy SOS. 2012. Programmer and Activist Interrogated at the Border. https://privacysos.org/blog/programmer-and-Activist-Interrogated-at-the-Border/ [Online; Accessed July 2018].

[17] Eugene H. Spafford. 1989. The Internet Worm Program: An Analysis. *ACM SIGCOMM Computer Communication Review* 19, 1 (1989), 17–57.