When Coding Style Survives Compilation: De-anonymizing Programmers from Executable Binaries

Aylin Caliskan*, Fabian Yamaguchi†, Edwin Dauber‡,
Richard Harang§, Konrad Rieck¶, Rachel Greenstadt‡ and Arvind Narayanan*
*Princeton University, {aylinc@, arvindn@cs}.princeton.edu
†Shiftleft Inc, fabs@shiftleft.io
‡Drexel University, {egd34, rachel.a.greenstadt}@drexel.edu
§Sophos, Data Science Team, rich.harang@sophos.com
¶TU Braunschweig, k.rieck@tu-bs.de

Abstract—**The ability to identify authors of computer programs based on their coding style is a direct threat to the privacy and anonymity of programmers. While recent work found that source code can be attributed to authors with high accuracy, attribution of executable binaries appears to be much more difficult. Many distinguishing features present in source code, e.g. variable names, are removed in the compilation process, and compiler optimization may alter the structure of a program, further obscuring features that are known to be useful in determining authorship. We examine programmer de-anonymization from the standpoint of machine learning, using a novel set of features that include ones obtained by decompiling the executable binary to source code. We adapt a powerful set of techniques from the domain of source code authorship attribution along with stylistic representations embedded in assembly, resulting in successful de-anonymization of a large set of programmers.**

We evaluate our approach on data from the Google Code Jam, obtaining attribution accuracy of up to 96% with 100 and 83% with 600 candidate programmers. We present an executable binary authorship attribution approach, for the first time, that is robust to basic obfuscations, a range of compiler optimization settings, and binaries that have been stripped of their symbol tables. We perform programmer de-anonymization using both obfuscated binaries, and real-world code found "in the wild" in single-author GitHub repositories and the recently leaked Nulled.IO hacker forum. We show that programmers who would like to remain anonymous need to take extreme countermeasures to protect their privacy.

I. Introduction

If we encounter an executable binary sample in the wild, what can we learn from it? In this work, we show that the programmer's stylistic fingerprint, or coding style, is preserved in the compilation process and can be extracted from the executable binary. This means that it may be possible to infer the programmer's identity if we have a set of known potential candidate programmers, along with executable binary samples (or source code) known to be authored by these candidates.

Programmer de-anonymization from executable binaries has implications for privacy and anonymity. Perhaps the creator of a censorship circumvention tool distributes it anonymously, fearing repression. Our work shows that such a programmer might be de-anonymized. Further, there are applications for software forensics, for example to help adjudicate cases of disputed authorship or copyright.

The White House Cyber R&D Plan states that "effective deterrence must raise the cost of malicious cyber activities, lower their gains, and convince adversaries that such activities can be attributed [42]." The DARPA Enhanced Attribution calls for methods that can "consistently identify virtual personas and individual malicious cyber operators over time and across different endpoint devices and C2 infrastructures [25]." While the forensic applications are important, as attribution methods develop, they will threaten the anonymity of privacy-minded individuals at least as much as malicious actors.

We introduce the first part of our approach by significantly overperforming the previous attempt at de-anonymizing programmers by Rosenblum et al. [39]. We improve their accuracy of 51% in de-anonymizing 191 programmers to 92% and then we scale the results to 83% accuracy on 600 programmers. First, whereas Rosenblum et al. extract structures such as control-flow graphs directly from the executable binaries, our work is the first to show that *automated decompilation* of executable binaries gives additional categories of useful features. Specifically, we generate *abstract syntax trees* of decompiled source code. Abstract syntax trees have been shown to greatly improve author attribution of source code [16]. We find that syntactical properties derived from these trees also improve the accuracy of executable binary attribution techniques.

Second, we demonstrate that using multiple tools for disassembly and decompilation in parallel increases the accuracy of de-anonymization by generating different representations of code that capture various aspects of the programmer's style. We present a robust machine learning framework based on entropy and correlation for dimensionality reduction, followed

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23304
www.ndss-symposium.org

by random-forest classification, that allows us to effectively use disparate types of features in conjunction without overfitting.

These innovations allow us to de-anonymize a large set of real-world programmers with high accuracy. We perform experiments with a controlled dataset collected from Google Code Jam (GCJ), allowing a direct comparison to previous work that used samples from GCJ. The results of these experiments are discussed in detail in Section V. Specifically; we can distinguish between *thirty times* as many candidate programmers (600 vs. 20) with higher accuracy, while utilizing less training data and much fewer stylistic features (53) per programmer. The accuracy of our method degrades gracefully as the number of programmers increases, and we present experiments with as many as 600 programmers. Similarly, we are able to tolerate scarcity of training data: our accuracy for de-anonymizing sets of 20 candidate programmers with just a single training sample per programmer is over 75%.

Third, we find that traditional binary obfuscation, enabling compiler optimizations, or stripping debugging symbols in executable binaries results in only a modest decrease in de-anonymization accuracy. These results, described in Section VI, are an important step toward establishing the practical significance of the method.

The fact that coding style survives compilation is unintuitive, and may leave the reader wanting a "sanity check" or an explanation for why this is possible. In Section V-J, we present several experiments that help illuminate this mystery. First, we show that decompiled source code *is not* necessarily similar to the original source code in terms of the features that we use; rather, the feature vector obtained from disassembly and decompilation can be used to *predict*, using machine learning, the features in the original source code. Even if no individual feature is well preserved, there is enough information in the vector as a whole to enable this prediction. On average, the cosine similarity between the original feature vector and the reconstructed vector is over 80%. Further, we investigate factors that are correlated with coding style being well-preserved, and find that more skilled programmers are more fingerprintable. This suggests that programmers gradually acquire their own unique style as they gain experience.

All these experiments were carried out using the GCJ dataset; the availability of this dataset is a boon for research in this area since it allows us to develop and benchmark our results under controlled settings [39], [9]. Having done that, we present the first ever de-anonymization study on an uncontrolled real-world dataset collected from GitHub in Section VI-D. This data presents difficulties, particularly noise in ground truth because of library and code reuse. However, we show that we can handle a noisy dataset of 50 programmers found in the wild with 65% accuracy and further extend our method to tackle open world scenarios. We also present a case study using code found via the recently leaked Nulled.IO hacker forum. We were able to find four forum members who, in private messages, linked to executables they had authored (one of which had only one sample). Our approach correctly attributed the three individuals who had enough data to build a model and correctly rejected the fourth sample as none of the previous three.

We emphasize that research challenges remain before programmer de-anonymization from executable binaries is fully ready for practical use. For example, programs may be authored by multiple programmers and may have gone through encryption. We have not performed experiments that model these scenarios which require different machine learning and segmentation techniques and we mainly focus on the privacy implications. Nonetheless, we present a robust and principled programmer de-anonymization method with a new approach and for the first time explore various realistic scenarios. Accordingly, our effective framework raise immediate concerns for privacy and anonymity.

The remainder of this paper is structured as follows. We begin by formulating the research question investigated throughout this paper in Section II, and discuss closely related work on de-anonymization in Section III. We proceed to describe our novel approach for binary authorship attribution based on instruction information, control flow graphs, and decompiled code in Section IV. Our experimental results are described in Section V, followed by a discussion of results in Section VII. Finally, we shed light on the limitations of our method in Section VIII and conclude in Section IX.

II. Problem Statement

In this work, we consider an analyst interested in determining the author of an executable binary purely based on its style. Moreover, we assume that the analyst only has access to executable binary samples each assigned to one of a set of candidate programmers.

Depending on the context, the analyst's goal might be defensive or offensive in nature. For example, the analyst might be trying to identify a misbehaving employee that violates the non-compete clause in his company by launching an application related to what he does at work. By contrast, the analyst might belong to a surveillance agency in an oppressive regime who tries to unmask anonymous programmers. The regime might have made it unlawful for its citizens to use certain types of programs, such as censorship-circumvention tools, and might want to punish the programmers of any such tools. If executable binary stylometry is possible, it means that compiled and cryptic code does not guarantee anonymity. Because of its potential dual use, executable binary stylometry is of interest to both security and privacy researchers.

In either (defensive or offensive) case, the analyst (or adversary) will seek to obtain labeled executable binary samples from each of these programmers who may have potentially authored the anonymous executable binary. The analyst proceeds by converting each labeled sample into a numerical feature vector, and subsequently deriving a classifier from these vectors using machine learning techniques. This classifier can then be used to attribute the anonymous executable binary to the most likely programmer.

Since we assume that a set of candidate programmers is known, we treat our main problem as a closed world, supervised machine learning task. It is a multi-class machine learning problem where the classifier calculates the most likely author for the anonymous executable binary sample among multiple authors. We also present experiments on an open-world scenario in Section VI-E.

Stylistic Fingerprints. An analyst is interested in identifying stylistic fingerprints in binary code to show that compiling source code does not anonymize it. The analyst engineers the numeric representations of stylistic properties that can be derived from binary code. To do so, the analyst generates representations of the program from the binary code. First, she uses a disassembler to obtain the low level features in assembly code. Second, she uses a decompiler to generate the control flow graph to capture the flow of the program. Lastly, she utilizes a decompiler to convert the low level instructions to high level decompiled source code in order to obtain abstract syntax trees. The analyst uses these three data formats to numerically represent the stylistic properties embedded in binary code. Given a set of labeled binary code samples with known authors, the analyst constructs the numeric representation of each sample. The analyst determines the set of stylistic features by calculating how much entropy each numeric value has in correctly differentiating between authors. She can further analyze how programmers' stylistic properties are preserved in a transformed format after compilation. Consequently, the analyst is able to quantify the level of anonymization and the amount of preserved stylistic fingerprints in binary code that has gone through compilation.

Additional Assumptions. For our experiments, we assume that we know the compiler used for a given program binary. Previous work has shown that with only 20 executable binary samples per compiler as training data, it is possible to use a linear Conditional Random Field (CRF) to determine the compiler used with accuracy of 93% on average [41], [27]. Other work has shown that by using pattern matching, library functions can be identified with precision and recall between 0.98 and 1.00 based on each one of three criteria; compiler version, library version, and linux distribution [23].

In addition to knowing the compiler, we assume to know the optimization level used for compilation of the binary. Past work has shown that toolchain provenance, including compiler family, version, optimization, and source language, can be identified with a linear CRF with accuracy of 99% for language, compiler family, and optimization and with 92% for compiler version [40]. Based on this success, we make the assumption that these techniques will be used to identify the toolchain provenance of the executable binaries of interest and that our method will be trained using the same toolchain.

III. Related Work

Any domain of creative expression allows authors or creators to develop a unique style, and we might expect that there are algorithmic techniques to identify authors based on their style. This class of techniques is called stylometry. Natural-language stylometry, in particular, is well over a century old [31]. Other domains such as source code and music also have stylistic features, especially grammar. Therefore stylometry is applicable to these domains as well, often using strikingly similar techniques [45], [10].

Linguistic stylometry. The state of the art in linguistic stylometry is dominated by machine-learning techniques [6], [32], [7]. Linguistic stylometry has been applied successfully to security and privacy problems, for example Narayanan et al. used stylometry to identify anonymous bloggers in large datasets, exposing privacy issues [32]. On the other hand, stylometry has also been used for forensics in underground cyber forums. In these forums, the text consists of a mixture of languages and information about forum products, which makes it more challenging to identify personal writing style. Not only have the forum users been de-anonymized but also their multiple identities across and within forums have been linked through stylometric analysis [7].

Authors may deliberately try to obfuscate or anonymize their writing style [12], [6], [30]. Brennan et al. show how stylometric authorship attribution can be evaded with adversarial stylometry [12]. They present two ways for adversarial stylometry, namely obfuscating writing style and imitating someone else's writing style. Afroz et al. identify the stylistic changes in a piece of writing that has been obfuscated while McDonald et al. present a method to make writing style modification recommendations to anonymize an undisputed document [6], [30].

Source code stylometry. Several authors have applied similar techniques to identify programmers based on source code [16], [34], [15]. Source code authorship attribution has applications in software forensics and plagiarism detection[1].

The features used for machine learning in source code authorship attribution range from simple byte-level [20] and word-level n-grams [13], [14] to more evolved structural features obtained from abstract syntax trees [16], [34]. In particular, Burrows et al. present an approach based on n-grams that reaches an accuracy of 77% in differentiating 10 different programmers [14].

Similarly, Kothari et al. combine n-grams with lexical markers such as the line length, to build programmer profiles that allow them to identify 12 authors with an accuracy of 76% [26]. Lange et al. further show that metrics based on layout and lexical features along with a genetic algorithm reach an accuracy of 75% in de-anonymizing 20 authors [28]. Finally, Caliskan-Islam et al. incorporate abstract syntax tree based structural features to represent programmers' coding style [16]. They reach 94% accuracy in identifying 1,600 programmers of the GCJ data set.

Executable binary stylometry. In contrast, identifying programmers from compiled code is considerably more difficult and has received little attention to date. Code compilation results in a loss of information and obstructs stylistic features. We are aware of only two prior works, both of which perform their evaluation and experiments on controlled corpora that are not noisy, such as the GCJ dataset and student homework assignments [39], [9]. Our work significantly overperforms previous work by using different methods and in addition we investigate noisy real-world datasets, an open-world setting, effects of optimizations, and obfuscations.

[9] present an onion approach for binary code authorship attribution. [39] identify authors of program binaries. Both Alrabaee et al. and Rosenblum et al. utilize the GCJ corpus.

Rosenblum et al. present two main machine learning tasks based on programmer de-anonymization. One is based on

[1] Note that popular plagiarism-detection tools such as Moss are not based on stylometry; rather they detect code that may have been copied, possibly with modifications. This is an orthogonal problem [8].

supervised classification with a support vector machine to identify the authors of compiled code [18]. The second machine learning approach they use is based on clustering to group together programs written by the same programmers. They incorporate a distance based similarity metric to differentiate between features related to programmer style to increase clustering accuracy. They use the Paradyn project's Parse API for parsing executable binaries to get the instruction sequences and control flow graphs whereas we use four different resources to parse executable binaries to generate a richer representation. Their dataset consists of submissions from GCJ and homework assignments with skeleton code.

Malware attribution. While the analysis of malware is a well developed field, authorship attribution of malware has received much less attention. Stylometry may have a role in this application, and this is a ripe area for future work that requires automated packer and encryption detection along with binary segment and metadata analysis. The difficulty in obtaining ground truth labels for malware samples has led much work in this area to focus on clustering malware in some fashion, and the wide range of obfuscation techniques in common use have led many researchers to focus on dynamic analysis rather than the static features we consider. The work of [29] examines several static features intended to provide credible links between executable malware binary produced by the same authors, however many of these features are specific to malware, such as command and control infrastructure and data exfiltration methods, and the authors note that many must be extracted by hand. In dynamic analysis, the work of [35] examines information obtained via both static and dynamic analysis of malware samples to organize code samples into lineages that indicate the order in which samples are derived from each other. [11] convert detailed execution traces from dynamic analysis into more general behavioral profiles, which are then used to cluster malware into groups with related functionality and activity. Supervised methods are used by [38] to match new instances of malware with previously observed families, again on the basis of dynamic analysis.

IV. Approach

Our ultimate goal is to automatically recognize programmers of compiled code. We approach this problem using supervised machine learning, that is, we generate a classifier from training data of sample executable binaries with known authors. The advantage of such learning-based methods over techniques based on manually specified rules is that the approach is easily retargetable to any set of programmers for which sample executable binaries exist. A drawback is that the method is inoperable if samples are not available or too short to represent authorial style. We study the amount of sample data necessary for successful classification in Section V.

Data representation is critical to the success of machine learning. Accordingly, we design a feature set for executable binary authorship attribution with the goal of faithfully representing properties of executable binaries relevant for programmer style. We obtain this feature set by augmenting lower-level features extractable from disassemblers with additional string and symbol information, and, most importantly, incorporating higher-level syntactical features obtained from decompilers.

In summary, such an approach results in a method consisting of the following four steps (see Figure 1) and the code is available at https://github.com/calaylin/bda.

- **Disassembly.** We begin by disassembling the program to obtain features based on machine code instructions, referenced strings, symbol information, and control flow graphs (Section IV-A).
- **Decompilation.** We proceed to translate the program into C-like pseudo code via decompilation. By subsequently passing the code to a fuzzy parser for C, we thus obtain abstract syntax trees from which syntactical features and n-grams can be extracted (Section IV-B).
- **Dimensionality reduction.** With features from disassemblers and decompilers at hand, we select those among them that are particularly useful for classification by employing a standard feature selection technique based on information gain and correlation based feature selection (Section IV-C).
- **Classification.** Finally, a random-forest classifier is trained on the corresponding feature vectors to yield a program that can be used for automatic executable binary authorship attribution (Section IV-D).

In the following sections, we describe these steps in greater detail and provide background information on static code analysis and machine learning where necessary.

A. *Feature extraction via disassembly*

As a first step, we disassemble the executable binary to extract low-level features that have been shown to be suitable for authorship attribution in previous work. In particular, we follow the basic example set by Rosenblum et al. and extract raw instruction traces from the executable binary [39]. In addition to this, disassemblers commonly make symbol information available, as well as strings referenced in the code, both of which greatly simplify manual reverse engineering. We augment the feature set accordingly. Finally, we can obtain control flow graphs of functions from disassemblers, providing features based on program basic blocks. The required information necessary to construct our feature set is obtained from the following two disassemblers.

We use two disassemblers to generate two sets of instructions for each binary. We disassemble the binary with the Netwide Disassembler (ndisasm) which is a widely available x86 disassembler. We then use the open source *radare2 disassembler* to get more detailed and higher level instructions than ndisasm's disassembly.

- ***The netwide disassembler.*** We begin by exploring whether simple instruction decoding alone can already provide useful features for de-anonymization. To this end, we process each executable binary using the netwide disassembler (*ndisasm*), a rudimentary disassembler that is capable of decoding instructions but is unaware of the executable's file format [44]. Due to this limitation, it resorts to simply decoding the executable binary from start to end, skipping bytes when invalid instructions are encountered. A problem with

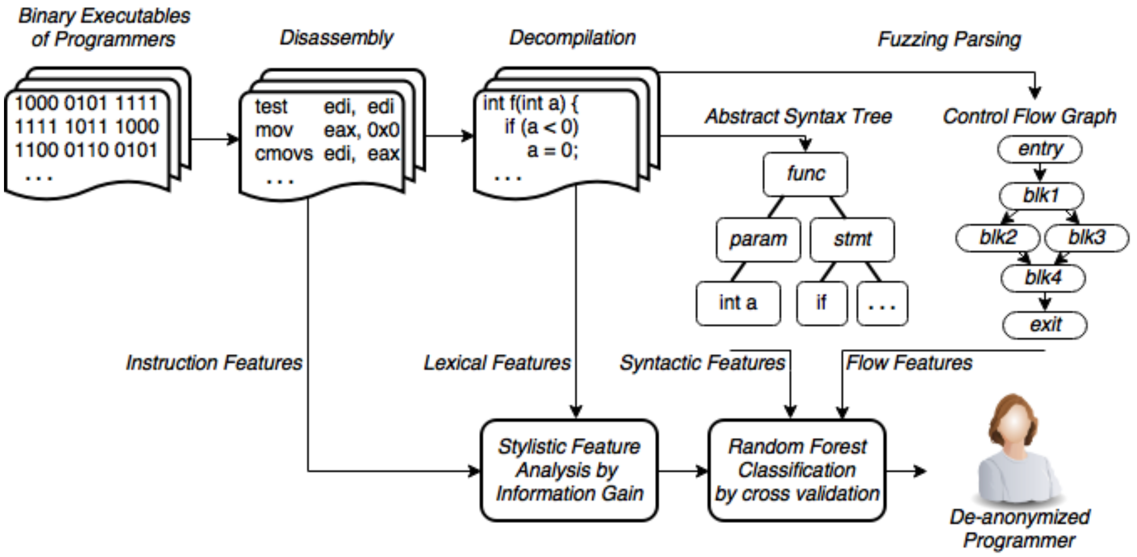

Fig. 1: Overview of our method. Instructions, symbols, and strings are extracted using disassemblers (1), abstract syntax tree and control-flow features are obtained from decompilers (2). Dimensionality reduction first by information gain criteria and then by correlation analysis is performed to obtain features that represent programmer style (3). Finally, a random forest classifier is trained to de-anonymize programmers (4).

this approach is that no distinction is made between bytes that represent data versus bytes that represent code. Nonetheless, we explore this simplistic approach as these inaccuracies may not degrade a classifier, given the statistical nature of machine learning.

- ***The radare2 disassembler.*** We proceed to apply *radare2* [33], a state-of-the-art open-source disassembler based on the capstone disassembly framework [37]. In contrast to *ndisasm*, *radare2* understands the executable binary format, allowing it to process relocation and symbol information in particular. This allows us to extract symbols from the dynamic (`.dynsym`) as well as the static symbol table (`.symtab`) where present, and any strings referenced in the code. Our approach thus gains knowledge over functions of dynamic libraries used in the code. Finally, *radare2* attempts to identify functions in code and generates corresponding control flow graphs.

Firstly, we strip the hexadecimal numbers from assembly instructions and replace them with the uni-gram $number$, to avoid overfitting that might be caused by unique hexadecimal numbers. Then, information provided by the two disassemblers is combined to obtain our disassembly feature set as follows: we tokenize the instruction traces of both disassemblers and extract token uni-grams, bi-grams, and tri-grams within a single line of assembly, and 6-grams, which span two consecutive lines of assembly. We cannot know exactly what each 6-gram corresponds to in assembly code but for most assembly instructions, a meaningful construct is longer than a line of assembly code. In addition, we extract single basic blocks of *radare2*'s control flow graphs, as well as pairs of basic blocks connected by control flow.

B. Feature extraction via decompilation

Decompilers are the second source of information that we consider for feature extraction in this work. In contrast to disassemblers, decompilers do not only uncover the program's machine code instructions, but additionally reconstruct higher level constructs in an attempt to translate an executable binary into equivalent source code. In particular, decompilers can reconstruct control structures such as different types of loops and branching constructs. We make use of these syntactical features of code as they have been shown to be valuable in the context of source code authorship attribution [16]. For decompilation, we employ the Hex-Rays decompiler [1].

Hex-Rays is a commercial state-of-the-art decompiler. It converts executable programs into a human readable C-like pseudo code to be read by human analysts. It is noteworthy that this code is typically significantly longer than the original source code. For example, decompiling an executable binary generated from 70 lines of source code with Hex-Rays produces on average 900 lines of decompiled code. We extract two types of features from this pseudo code: lexical features, and syntactical features. Lexical features are simply the word unigrams, which capture the integer types used in a program, names of library functions, and names of internal functions when symbol information is available. Syntactical features are obtained by passing the C-pseudo code to *joern*, a fuzzy parser for C that is capable of producing fuzzy abstract syntax trees (ASTs) from Hex-Rays pseudo code output [47]. We derive syntactic features from the abstract syntax tree, which represent the grammatical structure of the program. Such features are (illustrated in Figure 2) AST node unigrams, labeled AST edges, AST node term frequency inverse document frequency, and AST node average depth. Previous work on source code authorship attribution [16], [46] shows that these features are highly effective in representing programming style.

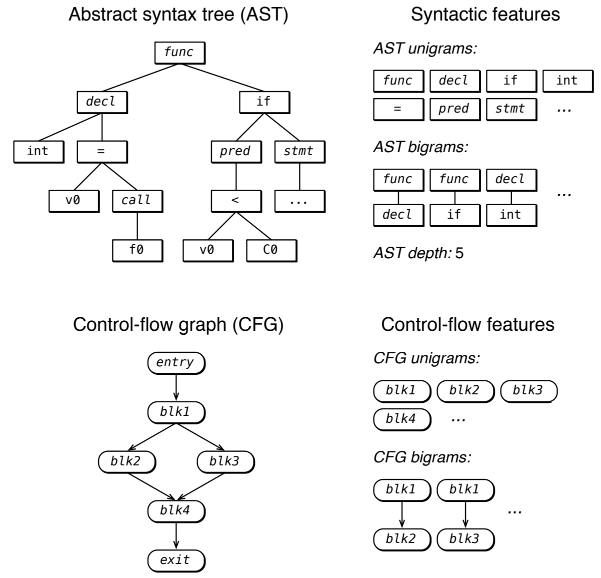

Fig. 2: Feature extraction via decompilation and fuzzy parsing: C-like pseudo code produced by Hex-Rays is transformed into an abstract syntax tree and control-flow graph to obtain syntactic and control-flow features.

C. Dimensionality reduction

Feature extraction produces a large amount of features, resulting in sparse feature vectors with thousands of elements. However, not all features are equally informative to express a programmer's style. This makes it desirable to perform feature selection to obtain a compact representation of the data to reduce the computational burden during classification as well as the chances of overfitting. Moreover, sparse vectors may result in a large number of zero-valued attributes being selected during random forest's random subsampling of the attributes to select a best split. Reducing the dimensions of the feature set is important for avoiding overfitting. One example to overfitting would be a rare assembly instruction uniquely identifying an author. For these reasons, we use information gain criteria followed by correlation based feature selection to identify the most informative attributes that represent each author as a class. This reduces vector size and sparsity while increasing accuracy and model training speed. For example, we get 705,000 features from the 900 executable binary samples of 100 programmers. If we use all of these features in classification, the resulting de-anonymization accuracy is slightly above 30% because the random forest might be randomly selecting features with values of zero in the sparse feature vectors. Once information gain criteria is applied, we get less than 2,000 features and the correct classification accuracy of 100 programmers increases from to 90%. Then, we identify locally predictive features that are highly correlated with classes and have low intercorrelation. After this second dimensionality reduction method, we are left with 53 predictive features and no sparsity remains in the feature vectors. Extracting 53 features or training a machine learning model where each instance has 53 attributes is computationally efficient. Given such proper representation of instances, the correct classification accuracy of 100 programmers reaches 96%.

We applied the first dimensionality reduction step using WEKA's information gain attribute selection criterion [21], which evaluates the difference between the entropy of the distribution of classes and the Shannon entropy of the conditional distribution of classes given a particular feature [36].

The second dimensionality reduction step was based on correlation based feature selection, which generates a feature-class and feature-feature correlation matrix. The selection method then evaluates the worth of a subset of attributes by considering the individual predictive ability of each feature along with the degree of redundancy between them [22]. Feature selection is performed iteratively with greedy hillclimbing and backtracking ability by adding attributes that have the highest correlation with the class to the list of selected features.

D. Classification

We used random forests as our classifier which are ensemble learners built from collections of decision trees, where each tree is trained on a subsample of the data obtained by random sampling with replacement. Random forests by nature are multi-class classifiers that avoid overfitting. To reduce correlation between trees, features are also subsampled; commonly $(logM)+1$ features are selected at random (without replacement) out of M, and the best split on these $(logM)+1$ features is used to split the tree nodes.

The number of selected features represents one of the few tuning parameters in random forests: increasing it increases the correlation between trees in the forest which can harm the accuracy of the overall ensemble, however increasing the number of features that can be chosen between at each split also increases the classification accuracy of each individual tree making them stronger classifiers with low error rates. The optimal range of number of features can be found using the out of bag error estimate, or the error estimate derived from those samples not selected for training on a given tree.

During classification, each test example is classified via each of the trained decision trees by following the binary decisions made at each node until a leaf is reached, and the results are aggregated. The most populous class is selected as the output of the forest for simple classification, or classifications can be ranked according to the number of trees that 'voted' for the label in question when performing relaxed attribution for *top-n* classification.

We employed random forests with 500 trees, which empirically provided the best tradeoff between accuracy and processing time. Examination of out of bag error values across multiple fits suggested that $(logM)+1$ random features (where M denotes the total number of features) at each split of the decision trees was in fact optimal in all of the experiments listed in Section V, and was used throughout. Node splits were selected based on the information gain criteria, and all trees were grown to the largest extent possible, without pruning.

The data was analyzed via *k*-fold cross-validation, where the data was split into training and test sets stratified by author (ensuring that the number of code samples per author in the training and test sets was identical across authors). The parameter *k* varies according to datasets and is equal to the number of instances present from each author. The cross-validation procedure was repeated 10 times, each with a different random seed, and average results across all iterations are reported, ensuring that results are not biased by improbably easy or difficult to classify subsets.

We report our classification results in terms of kappa statistics, which is roughly equivalent to accuracy but subtracts the random chance of correct classification from the final accuracy. As programmer de-anonymization is a multi-class classification problem, an evaluation based on accuracy, or the true positive rate, represents the correct classification rate in the most meaningful way.

V. Google Code Jam experiments

In this section, we go over the details of the various experiments we performed to address the research question formulated in Section II.

A. *Dataset*

We evaluate our executable binary authorship attribution method on a controlled dataset based on the annual programming competition *GCJ* [5]. It is an annual contest that thousands of programmers take part in each year, including professionals, students, and hobbyists from all over the world. The contestants implement solutions to the same tasks in a limited amount of time in a programming language of their choice. Accordingly, all the correct solutions have the same algorithmic functionality. There are two main reasons for choosing GCJ competition solutions as an evaluation corpus. First, it enables us to directly compare our results to previous work on executable binary authorship attribution as both [9] and [39] evaluate their approaches on data from GCJ. Second, we eliminate the potential confounding effect of identifying programming task rather than programmer by identifying functionality properties instead of stylistic properties. GCJ is a less noisy and clean dataset known definitely to be single authored. GCJ solutions do not have significant dependencies outside of the standard library and contain few or no third party libraries.

We focus our analysis on compiled C++ code, the most popular programming language used in the competition. We collect the solutions from the years 2008 to 2014 along with author names and problem identifiers. In GCJ experiments we are assuming that the programmers are not deliberately trying to hide their identity. Accordingly, we show results without excluding symbol information.

B. *Code Compilation*

To create our experimental datasets, we first compiled the source code with GNU Compiler Collection's gcc or g++ without any optimization to Executable and Linkable Format (ELF) 32-bit, Intel 80386 Unix binaries. The training set needs to be compiled with the same compiler and settings otherwise we might end up detecting the compiler instead of the author. Passing the training samples through the same encoder preserves mutual information between code style and labels and accordingly we can successfully de-anonymize programmers.

Next, to measure the effect of different compilation options, such as compiler optimization flags, we additionally compiled the source code with level-1, level-2, and level-3 optimizations, namely the O1, O2, and O3 flags. O3 is a superset of O2 optimization flags and similarly O2 is a superset of O1 flags. The compiler attempts to improve the performance and/or code size when the compiler flags are turned on but at the same time optimization has the expense of increasing compilation time and complicating program debugging.

C. *53 features represent programmer style.*

We are interested in identifying features that represent coding style preserved in executable binaries. With the current approach, we extract 705,000 representations of code properties of 100 authors, but only a subset of these are the result of individual programming style. We are able to capture the features that represent each author's programming style that is preserved in executable binaries by applying information gain criteria to these 705,000 features. After applying information gain to effectively represent coding style, we reduce the feature set to contain approximately 1,600 features from all feature types. Furthermore, correlation based feature selection during cross validation eliminates features that have low class correlation and high intercorrelation and preserves 53 of the highly distinguishing features which can be seen in Table I along with their authorial style representation power.

Considering the fact that we are reaching such high accuracies on de-anonymizing 100 programmers with 900 executable binary samples (discussed below), these features are providing strong representation of style that survives compilation. The compact set of identifying stylistic features contain features of all types, namely disassembly, CFG, and syntactical decompiled code properties. To examine the potential for overfitting, we consider the ability of this feature set to generalize to a different set of programmers (see Section V-G), and show that it does so, further supporting our belief that these features effectively capture coding style. Features that are highly predictive of authorial fingerprints include file and stream operations along with the formats and initializations of variables from the domain of ASTs whereas arithmetic, logic, and stack operations are the most distinguishing ones among the assembly instructions.

D. *We can de-anonymize programmers from their executable binaries.*

This is the main experiment that demonstrates how de-anonymizing programmers from their executable binaries is possible. After preprocessing the dataset to generate the executable binaries without optimization, we further process the executable binaries to obtain the disassembly, control flow graphs, and decompiled source code. We then extract all the possible features detailed in Section IV. We take a set of 100 programmers who all have 9 executable binary samples. With 9-fold-cross-validation, the random forest correctly classifies 900 test instances with 95% accuracy, which is significantly higher than the accuracies reached in previous work.

There is an emphasis on the number of folds used in these experiments because each fold corresponds to the implementation of the same algorithmic function by all the programmers in the GCJ dataset (e.g. all samples in fold 1 may be attempts by the various authors to solve a list sorting problem). Since we know that each fold corresponds to the same Code Jam

Feature	Source	Number of Possible Features	Selected Features / Information Gain
Word unigrams	decompiled code*	29,278	6/5.75
AST node TF†	decompiled code*	5,278	3/1.85
Labeled AST edge TF†	decompiled code*	26,783	0/0
AST node TFIDF‡	decompiled code*	5,278	1/0.75
AST node average depth	decompiled code*	5,278	0/0
C++ keywords	decompiled code*	73	0/0
radare2 disassembly unigrams	radare disassembly	21,206	3/1.61
radare2 disassembly bigrams	radare disassembly	39,506	1/0.62
radare2 disassembly trigrams	radare disassembly	112,913	0/0
radare2 disassembly 6-grams	ndisasm disassembly	260,265	0/0
radare2 CFG node unigrams	radare disassembly	5,297	3/1.98
radare2 CFG edges	radare disassembly	10,246	1/0.63
ndisasm disassembly unigrams	ndisasm disassembly	5,383	2/1.79
ndisasm disassembly bigrams	ndisasm disassembly	14,305	5/2.95
ndisasm disassembly trigrams	ndisasm disassembly	5,237	4/1.44
ndisasm disassembly 6-grams	ndisasm disassembly	159,142	24/16.08
Total		705,468	53/35
**hex-rays decompiled code*		*TF†= term frequency*	
TFIDF‡= term frequency inverse document frequency			

TABLE I: Programming Style Features and Selected Features in Executable Binaries

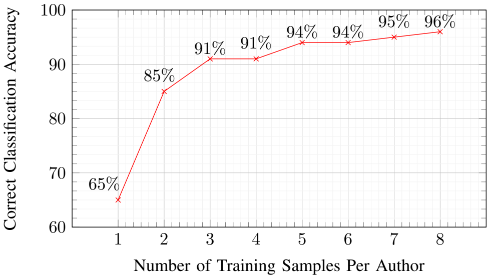

Fig. 3: Amount of Training Data Required for De-anonymizing 100 Programmers

problem, by using stratified cross validation without randomization and preserving order, we ensure that all training and test samples contain the same algorithmic functions implemented by all of the programmers. The classifier uses the excluded fold in the testing phase, which contains executable binary samples that were generated from an algorithmic function that was *not* previously observed in the training set for that classifier. Consequently, the only distinction between the test instances is the coding style of the programmer, without the potentially confounding effect of identifying an algorithmic function.

E. *Even a single training sample per programmer is sufficient for de-anonymization.*

A drawback of supervised machine learning methods, which we employ, is that they require labeled examples to build a model. The ability of the model to accurately generalize is often strongly linked to the amount of data provided to it during the training phase, particularly for complex models. In domains such as executable binary authorship attribution, where samples may be rare and obtaining "ground truth" for labeling training samples may be costly or laborious, this can pose a significant challenge to the usefulness of the method.

We therefore devised an experiment to determine how much training data is required to reach a stable classification accuracy, as well as to explore the accuracy of our method with severely limited training data. As programmers produce a limited number of code samples per round of the GCJ competition, and programmers are eliminated in each successive round, the GCJ dataset has an upper bound in the number of code samples per author as well as a limited number of authors with a large number of samples. Accordingly, we identified a set of 100 programmers that had exactly 9 program samples each, and examined the ability of our method to correctly classify each author out of the candidate set of 100 authors when training on between 1 and 8 files per author.

As shown in Figure 3, the classifier is capable of correctly identifying the author of a code sample from a potential field of 100 with 65% accuracy on the basis of a single training sample. The classifier also reaches a point of dramatically diminishing returns with as few as three training samples, and obtains a stable accuracy by training on 6 samples. Given the complexity of the task, this combination of high accuracy with extremely low requirement on training data is remarkable, and suggests the robustness of our features and method. It should be noted, however that this set of programmers with a large number of files corresponds to more skilled programmers, as they were able to remain in the competition for a longer period of time and thus produce this large number of samples.

F. *Relaxed Classification: In difficult scenarios, the classification task can be narrowed down to a small suspect set.*

In Section V-A, the previously unseen anonymous executable binary sample is classified such that it belongs to the most likely author's class. In cases where we have too many classes or the classification accuracy is lower than expected, we can relax the classification to *top–n* classification. In *top–n* relaxed classification, if the test instance belongs to one of the most likely n classes, the classification is considered correct. This can be useful in cases when an analyst or adversary is interested in finding a suspect set of n authors, instead of a direct *top–1* classification. Being able to scale down an authorship investigation for an executable binary sample of interest to a reasonable sized set of suspect authors among hundreds of authors greatly reduces the manual effort required by an analyst or adversary. Once the suspect set size is reduced, the analyst or adversary could adhere to content based dynamic approaches and reverse engineering to identify the author of the executable binary sample. Figure 4 shows how correct classification accuracies approach 100% as the classification is relaxed to top-10.

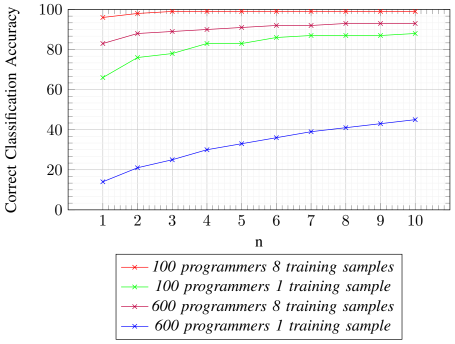

Fig. 4: Reducing Suspect Set Size: Top-n Relaxed Classification

It is important to note from Figure 3 that, by using only a single training sample in a 100-class classification task, the machine learning model can correctly classify new samples with 75.0% accuracy. This is of particular interest to an analyst or adversary who does not have a large amount of labeled samples in her suspect set. Figure 3 shows that an analyst or adversary can narrow down the suspect set size from 100 or 600 to a significantly smaller set.

G. *The feature set selected via dimensionality reduction works and is validated across different sets of programmers.*

In our earlier experiments, we trained the classifier on the same set of executable binaries that we used during feature selection. The high number of starting features from which we select our final feature set via dimensionality reduction does raise the potential concern of overfitting. To examine this, we applied this final feature set to a different set of programmers and executable binaries. If we reach accuracies similar to what we got earlier, we can conclude that these selected features do generalize to other programmers and problems, and therefore are not overfitting to the 100 programmers they were generated from. This also suggests that the final set of features in general capture programmer style.

Recall that analyzing 900 executable binary samples of the 100 programmers resulted in about 705,000 features, and after dimensionality reduction, we are left with 53 important features. We picked a different (non-overlapping) set of 100 programmers and performed another de-anonymization experiment in which the feature selection step was omitted, using instead the information gain and correlation based features obtained from the original experiment. This resulted in very similar accuracies: we de-anonymized programmers in the validation set with 96% accuracy by using features selected via the main development set, compared to the 95% de-anonymization accuracy we achieve on the programmers of the main development set. The ability of the final reduced set of 53 features to generalize beyond the dataset which guided their selection strongly supports the assertion that these features obtained from the main set of 100 programmers are not overfitting, and they actually represent coding style in executable binaries, and can be used across different datasets.

H. *Large Scale De-anonymization: We can de-anonymize 600 programmers from their executable binaries.*

We would like to see how well our method scales up to 600 users. An analyst with a large set of labeled samples might be interested in performing large scale de-anonymization. For this experiment, we use 600 contestants from GCJ with 9 files. We only extract the reduced set of features from the 600 users. This decreases the amount of time required for feature extraction. On the other hand, this experiment shows how effectively overall programming style is represented after dimensionality reduction. The results of large scale programmer de-anonymization in Figure 5, show that our method can scale to larger datasets with the reduced set of features with a surprisingly small drop on accuracy.

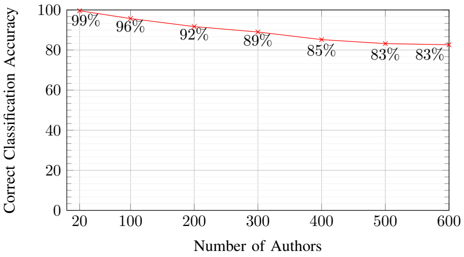

Fig. 5: Large Scale Programmer De-anonymization

I. *We advance the state of executable binary authorship attribution.*

Rosenblum et al. presented the largest scale evaluation of executable binary authorship attribution on 191 programmers each with at least 8 training samples [39]. We compare our results with Rosenblum et al.'s in Table II to show how we advance the state of the art both in accuracy and on larger datasets. Rosenblum et al. use 1,900 coding style features to represent coding style whereas we use 53 features, which might suggest that our features are more powerful in representing coding style that is preserved in executable binaries. On the other hand, we use less training samples as opposed to Rosenblum et al., which makes our experiments more challenging from a machine learning standpoint. Our accuracy in authorship attribution is significantly higher than Rosenblum et al.'s, even when we use an SVM as our classifier, showing that our different approach is more powerful and robust for de-anonymizing programmers. Rosenblum et al. suggest a linear SVM is the appropriate classifier for de-anonymizing programmers but we show that our different set of techniques and choice of random forests is leading to superior and larger scale de-anonymization.

Related Work	Number of Programmers	Number of Training Samples	Accuracy	Classifier
Rosenblum [39]	20	**8-16**	77%	SVM
This work	20	**8**	90%	SVM
This work	20	**8**	99%	RF
Rosenblum [39]	100	8-16	**61%**	SVM
This work	100	8	**84%**	SVM
This work	100	8	**96%**	RF
Rosenblum [39]	191	8-16	**51%**	SVM
This work	191	8	**81%**	SVM
This work	191	8	**92%**	RF
This work	**600**	8	**71%**	SVM
This work	**600**	8	**83%**	RF

TABLE II: Comparison to Previous Results

J. *Programmer style is preserved in executable binaries.*

We show throughout the results that it is possible to de-anonymize programmers from their executable binaries with a high accuracy. To quantify how stylistic features are preserved in executable binaries, we calculated the correlation of stylistic source code features and decompiled code features. We used the stylistic source code features from previous work on de-anonymizing programmers from their source code [16]. We took the most important 150 features in coding style that consist of AST node average depth, AST node TFIDF, and the frequencies of AST nodes, AST node bigrams, word unigrams, and C++ keywords. For each executable binary sample, we have the corresponding source code sample. We extract 150 information gain features from the original source code. We extract decompiled source code features from the decompiled executable binaries. For each executable binary instance, we set one corresponding information gain feature as the class to predict and then we calculate the correlation between the decompiled executable binary features and the class value. A random forest classifier with 500 trees predicts the class value of each instance, and then Pearson's correlation coefficient is calculated between the predicted and original values. The correlation has a mean of 0.32 and ranges from -0.12 to 0.69 for the most important 150 features.

To see how well we can reconstruct the original source code features from decompiled executable binary features, we reconstructed the 900 instances with 150 features that represent the highest information gain features by predicting the original features from decompiled code features. We calculated the cosine similarity between the original 900 instances and the reconstructed instances after normalizing the features to unit distance. The cosine similarity for these instances is in Figure 6, where a cosine similarity of 1 means the two feature vectors are identical. The high values (average of 0.81) in cosine similarity suggest that the reconstructed features are similar to the original features. When we calculate the cosine similarity between the feature vectors of the original source code and the corresponding decompiled code's feature vectors (*no predictions*), the average cosine similarity is 0.35. In summary, reconstructed features are much more similar to original code than the raw features extracted from decompiled code. 5% of the reconstructed features have less than 60% similarity based on the cosine similarity between original and decompiled source code features. At the same time, the de-anonymization accuracy of 900 executable binaries is 95% by using source code, assembly, CFG, and AST features. This might indicate that some operations or code sequences cannot be preserved after compilation followed by decompilation, due to the nature of transformations during each process.

VI. Real-World Scenarios

A. *Programmers of optimized executable binaries can be de-anonymized.*

In Section V, we discussed how we evaluated our approach on a controlled and clean real-world dataset. Section V shows how we advance over previous methods that were all evaluated with clean datasets such as GCJ or homework assignments. In this section, we investigate a complicated dataset which has been optimized during compilation, where the executable binary samples have been normalized further during compilation.

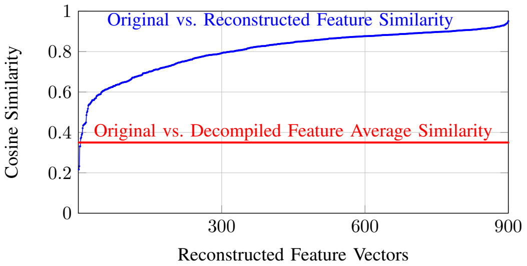

Fig. 6: Feature Transformations: Each data point on the x-axis is a different executable binary sample. Each y-axis value is the cosine similarity between the feature vector extracted from the original source code and the feature vector that tries to *predict* the original features. The average value of these 900 cosine similarity measurements is 0.81, suggesting that decompiled code preserves transformed forms of the original source code features well enough to reconstruct the original source code features.

Compiling with optimization tries to minimize or maximize some attributes of an executable program. The goal of optimization is to minimize execution time or the amount of memory a program occupies. The compiler applies optimizing transformations which are algorithms that transform a program to a semantically equivalent program that uses fewer resources.

GCC has predefined optimization levels that turn on sets of optimization flags. Compilation with optimization level-1, tries to reduce code size and execution time, takes more time and much more memory for large functions than compilation with no optimizations. Compilation with optimization level-2 optimizes more than level-1, uses all level-1 optimization flags and more. Level-2 optimization performs all optimizations that do not involve a space-speed tradeoff. Level-2 optimization increases compilation time and performance of the generated code when compared to level-1 optimization. Level-3 optimization yet optimizes more than both level-1 and level-2.

So far, we have shown that programming style features survive compilation without any optimizations. As compilation with optimizations transforms code further, we investigate how much programming style is preserved in executable binaries that have gone through compilation with optimization. Our results summarized in Table III show that programming style is preserved to a great extent even in the most aggressive level-3 optimization. This shows that programmers of optimized executable binaries can be de-anonymized and optimization is not a highly effective code anonymization method.

Number of Programmers	**Number of Training Samples**	**Compiler Optimization Level**	**Accuracy**
100	8	None	96%
100	8	1	93%
100	8	2	89%
100	8	3	89%

TABLE III: Programmer De-anonymization with Compiler Optimization

B. *Removing symbol information does not anonymize executable binaries.*

To investigate the relevance of symbol information for classification accuracy, we repeat our experiments with 100 authors presented in the previous section on *fully stripped executable binaries*, that is, executable binaries where symbol information is missing completely. We obtain these executable binaries using the standard utility *GNU strip* on each executable binary sample prior to analysis. Upon removal of symbol information, without any optimizations, we notice a decrease in classification accuracy by 24%, showing that stripping symbol information from executable binaries is not effective enough to anonymize an executable binary sample.

C. *We can de-anonymize programmers from obfuscated binaries.*

We are furthermore interested in finding out whether our method is capable of dealing with simple binary obfuscation techniques as implemented by tools such as Obfuscator-LLVM [24]. These obfuscators substitute instructions by other semantically equivalent instructions, they introduce bogus control flow, and can even completely flatten control flow graphs.

For this experiment, we consider a set of 100 programmers from the GCJ data set, who all have 9 executable binary samples. This is the same data set as considered in our main experiment (see Section V-D), however, we now apply all three obfuscation techniques implemented by Obfuscator-LLVM to the samples prior to learning and classification.

We proceed to train a classifier on obfuscated samples. This approach is feasible in practice as an analyst who has only non-obfuscated samples available can easily obfuscate them to obtain the necessary obfuscated samples for classifier training. Using the same features as in Section V-D, we obtain an accuracy of 88% in correctly classifying authors.

D. *De-anonymization* **in the Wild**

To better assess the applicability of our programmer de-anonymization approach *in the wild*, we extend our experiments to code collected from real open-source programs as opposed to solutions for programming competitions. To this end, we automatically collected source files from the popular open-source collaboration platform GitHub [4]. Starting from a seed set of popular repositories, we traversed the platform to obtain C/C++ repositories that meet the following criteria. Only one author has committed to the repository. The repository is popular as indicated by the presence of at least 5 *stars*, a measure of popularity for repositories on GitHub. Moreover, it is sufficiently large, containing a total of 200 lines at least. The repository is not a fork of another repository, nor is it named 'linux', 'kernel', 'osx', 'gcc', 'llvm', 'next', as these repositories are typically copies of the so-named projects.

We cloned 439 repositories from 161 authors meeting these criteria and collect only C/C++ files for which the main author has contributed at least 5 commits and the commit messages do not contain the word 'signed-off', a message that typically indicates that the code is written by another person. An author and her files are included in the dataset only if she has written at least 10 different files. In the final step, we manually verified ground truth on authorship for the selected files to make sure that they do not show any clear signs of code reuse from other projects. The resulting dataset had 2 to 344 files and 2 to 8 repositories from each author, with a total of 3,438 files.

We developed our method and evaluated it on the GCJ dataset, but collecting code from open source projects is another option for constructing a dataset. Open source projects do not guarantee ground truth on authorship. The feature vectors might capture topics of the project instead of programming style. As a result, open source code does not constitute the ideal data for authorship analysis; however, it allows us to better assess the applicability of programmer de-anonymization in the wild. We therefore present results from a dataset collected from the hosting platform GitHub, which we obtain by spidering the platform to collect C and C++ repositories.

We subsequently compile the collected projects to obtain object files for each of the selected source files. We perform our experiment on object files as opposed to entire binaries, since the object files are the binary representations of the source files that clearly belong to the specified authors.

For different reasons, compiling code may not be possible for a project, e.g., the code may not be in a compilable state, it may not be compilable for our target platform (32 bit Intel, Linux), or the files to setup a working build environment can no longer be obtained. Despite these difficulties, we are able to generate 1,075 object files from 90 different authors, where the number of object files per author ranges from 2 to 24, with most authors having at least 9 samples. We used 50 of these authors that have 6 to 15 files to perform a machine learning experiment with more balanced class sizes.

We extract the information gain features that were selected from GCJ data from this GitHub dataset. GitHub datasets are noisy for two reasons since the executable binaries used in de-anonymization might contain properties from third party libraries and code. For these two reasons, it is more difficult to attribute authorship to anonymous executable binary samples from GitHub, but nevertheless we reach 65% accuracy in correctly classifying these programmers' executable binaries. Another difficulty in this particular dataset is that there is not much training data to train an accurate random forest classifier that models each programmer. For example, we can de-anonymize the two programmers with the most samples, one with 11 samples and one with 7, with 100% accuracy.

Being able to de-anonymize programmers in the wild by using a small number of features obtained from our clean development dataset is a promising step towards attacking more challenging real-world de-anonymization problems.

E. *Have I seen this programmer before?*

While attempting to de-anonymize programmers in real-world settings, we cannot be certain that we have formerly encountered code samples from the programmers in the test set. As a mechanism to check whether an anonymous test file belongs to one of the candidate programmers in the training set, we extend our method to an open world setting by incorporating classification confidence thresholds. In random forests, the class probability or classification confidence $P(B_i)$ that executable binary B is of class i is calculated by taking the percentage of trees in the random forest that voted for class i during classification.

$$P(B_i) = \frac{\sum_j V_j(i)}{|T|_f} \qquad (1)$$

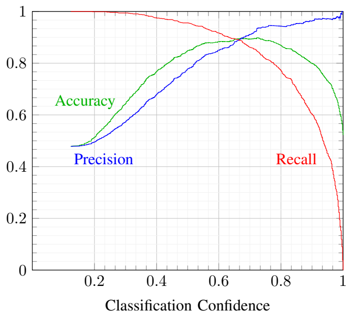

Fig. 7: Confidence Thresholds for Verification

There are multiple ways to assess classifier confidence and we devise a method that calculates the classification confidence by using classification margins. In this setting, the classification margin of a single instance is the difference between the highest and second highest $P(B_i)$. The first step towards attacking an open world classification task is identifying the confidence threshold of the classifier for classification verification. As long as we determine a confidence threshold based on training data, we can calculate the probability that an instance belongs to one of the programmers in the training set and accordingly accept or reject the classification.

We performed 900 classifications in a 100-class problem to determine the confidence threshold based on the training data. The accuracy was 95%. There were 40 misclassifications with an average classification confidence of 0.49. We took another set of 100 programmers with 900 samples. We classify these 900 samples with the closed world classifier that was trained in the first step on samples from a disjoint set of programmers. All of the 900 samples are attributed to a programmer in the closed world classifier with a mean classification confidence of 0.40. We can pick a verification threshold and reject all classifications with confidence below the selected threshold. Accordingly all the rejected open world samples and misclassifications become true negatives, and the rejected correct classifications end up as false negatives. Open world samples and misclassifications above the threshold are false positives and the correct classifications are true positives. Based on this, we generate an accuracy, pecision, and recall graph with varying confidence threshold values in Figure 7. This figure shows that the optimal rejection threshold to guarantee 90% accuracy on 1,800 samples and 100 classes is around confidence 0.72. Other confidence thresholds can be picked based on precision and recall trade-offs. These results are encouraging for extending our programmer de-anonymization method to open world settings where an analyst deals with many uncertainties under varying fault tolerance levels.

The experiments in this section can be used in software forensics to find out the programmer of a piece of malware. In software forensics, the analyst does not know if source code belongs to one of the programmers in the candidate set of programmers. In such cases, we can classify the anonymous source code, and if the majority number of votes of trees in the random forest is below a certain threshold, we can reject the classification considering the possibility that it might not belong to any of the classes in the training data. By doing so, we can scale our approach to an open world scenario, where we might not have encountered the suspect before. As long as we determine a confidence threshold based on training data [43], we can calculate the probability that an instance belongs to one of the programmers in the set and accordingly accept or reject the classification. We performed 270 classifications in a 30-class problem using all the features to determine the confidence threshold based on the training data. The accuracy was 96.67%. There were 9 misclassifications and all of them were classified with less than 15% confidence by the classifier.

Where $V_j(i) = 1$ if the j^{th} tree voted for class i and 0 otherwise, and $|T|_f$ denotes the total number of trees in forest f. Note that by construction, $\sum_i P(C_i) = 1$ and $P(C_i) \geq 0$ $\forall$ i, allowing us to treat $P(C_i)$ as a probability measure.

F. Case Study: Nulled.IO Hacker Forum

On May 6, 2016 the well known ‘hacker’ forum *Nulled.IO* was compromised and its forum dump was leaked along with the private messages of its 585,897 members. The members of these forums share, sell, and buy stolen credentials and cracking software. A high number of the forum members are active developers that write their own code and sell them, or share some of their code for free in public GitHub repositories along with tutorials on how to use them. The private messages of the sellers in the forum include links to their products and even to screenshots of how the products work, for buyers. We were able to find declared authorship along with active links to members’ software on sharing sites such as FileDropper[2] and MediaFire[3] in the private messages.

For our case study, we created a dataset from four forum members with a total of thirteen Windows executables. One of the members had only one sample, which we used to test the open world setting described in Section VI-E. A challenge encountered in this case study is that the binary programs obtained from Nulled.IO do not contain native code, but bytecode for the Microsoft Common Language Infrastructure (CLI). Therefore, we cannot immediately analyze them using our existing toolchain. We address this problem by first translating bytecode into corresponding native code using the Microsoft Native Image Generator (ngen.exe), and subsequently forcing the decompiler to treat the generated output files as regular native code for binaries. On the other hand, *radare2* is not able to disassemble such output or the original executables. Consequently we had access to a subset of the information gain feature set obtained from GCJ. We extracted a total of 605 features consisting of decompiled source code features and *ndisasm* disassembly features. Nevertheless, we are able to de-anonymize these programmers with 100% accuracy while the one sample from the open world class is classified in all cases with the lowest confidence, such as 0.4, which is below the verification threshold and is recognized by the classifier as a sample that does not belong to the rest of the programmers.

A larger de-anonymization attack can be carried out by collecting code from GitHub users with relevant repositories and identifying all the available executables mentioned in the public portions of hacker forums. GitHub code can be compiled with necessary parameters and used with the approach described in Section VI-D. Incorporating verification thresholds from Section VI-E can help handle programmers with

[2] www.filedropper.com: ‘Simplest File Hosting Website..’
[3] www.mediafire.com: ‘All your media, anywhere you go’

only one sample. Consequently a large number of members can be linked, reduced to a cluster or directly de-anonymized.

The countermeasure against real-world programmer de-anonymization attacks requires a combination of various precautions. Developers should not have any public repositories. A set of programs should not be released by the same online identity. Programmers should try to have a different coding style in each piece of software they write and also try to code in different programming languages. Software should utilize different optimizations and obfuscations to avoid deterministic patterns. A programmer who accomplishes randomness across all potential identifying factors would be very difficult to de-anonymize. Nevertheless, even the most privacy savvy developer might be willing to contribute to open source software or build a reputation for her identity based on her set of products, which would be a challenge for maintaining anonymity.

Some of these developers obfuscate their code with the primary goal of hiding the source code and consequently they are experienced in writing or using obfuscators and deobfuscators. An additional challenge encountered in this case study is that the binary programs obtained from NULLED.io do not contain native code, but bytecode for the Microsoft Common Language Infrastructure (CLI). Therefore, we cannot immediately analyze them using our existing toolchain. We address this problem by first translating bytecode into corresponding native code using the Microsoft Native Image Generator (ngen), and subsequently forcing the decompiler to treat the generated output files as regular native code binaries.

VII. Discussion

Our experiments are devised for a setting where the programmer is not trying to hide her coding style, and therefore, only basic obfuscation techniques are considered in our experiments. Accordingly, we focus on the general case of executable binary authorship attribution, which is a serious threat to privacy but at the same time an aid for forensic analysis.

We consider two data sets: the GCJ dataset, and a dataset based on GitHub repositories. Using the GitHub dataset, we show that we can perform programmer de-anonymization with executable binary authorship attribution in the wild. We de-anonymize GitHub programmers by using stylistic features obtained from the GCJ dataset. Using the same small set of features, we perform a case study on the leaked hacker forum Nulled.IO and de-anonymize four of its members. The successful de-anonymization of programmers from different sources supports the supposition that, in addition to its other useful properties for scientific analysis of attribution tasks, the GCJ dataset is a valid and useful proxy for real-world authorship attribution tasks.

The advantage of using the GCJ dataset is that we can perform the experiments in a controlled environment where the most distinguishing difference between programmers' solutions is their programming style. Every contestant implements the same functionality, in a limited amount of time while at each round problems are getting more difficult. This provides the opportunity to control the difficulty level of the samples and the skill set of the programmers in the dataset. In source code authorship attribution, programmers who can implement more sophisticated functionality have a more distinct programming style [16]. We observe the same pattern in executable binary samples and gain some software engineering insights by analyzing stylistic properties of executable binaries. In contrast to GCJ, GitHub and Nulled.IO offer noisy samples. However, our results show that we can de-anonymize programmers with high accuracy as long as enough training data is available.

Previous work shows that coding style is quite prevalent in source code. We were surprised to find that it is also preserved to a great degree in compiled source code. Coding style is not just the use of particular syntactical constructs but also the AST flows, AST combinations, and preferred types of operations. Consequently, these patterns manifest in the binary and form a coding fingerprint for each author. We can de-anonymize programmers from compiled source code with great accuracy, and furthermore, we can de-anonymize programmers from source code compiled with optimization or after obfuscation. In our experiments, we see that even though basic obfuscation, optimization, or stripping symbols transforms executable binaries more than plain compilation, stylistic features are still preserved to a large degree. Such methods are not sufficient on their own to protect programmers from de-anonymization attacks.

In scenarios where authorship attribution is challenging, an analyst or adversary could apply relaxed attribution to find a suspect set of n authors, instead of a direct *top–1* classification. In *top–10* attribution, the chances of having the original author within the returned set of 10 authors approaches 100%. Once the suspect set size is reduced to 10 from $hundreds$, the analyst or adversary could adhere to content based dynamic approaches and reverse engineering to identify the author of the executable binary sample. However, our experiments in these cases are performed using the information-gain features determined from the unoptimized case with symbol tables intact. Future work that customizes the dimensionality reduction step for these cases (for example, removing features from the trees that are no longer relevant) may be able to improve upon these numbers, especially since dimensionality reduction was able to provide such a large boost in the unoptimized case.

Even though executable binaries look cryptic and difficult to analyze, we can still extract many useful features from them. We extract features from disassembly, control flow graphs, and also decompiled code to identify features relevant to only programming style. After dimensionality reduction, we see that each of the feature spaces provides programmer style information. The initial development feature set contains a total of 705,000 features for 900 executable binary samples of 100 authors. Approximately 50 features from abstract syntax trees and assembly instructions suffice to capture enough key information about coding style to enable robust authorship attribution. We see that the reduced set of features are valid in different datasets with different programmers, including optimized or obfuscated programmers. Also, the reduced feature set is helpful in scaling up the programmer de-anonymization approach. While we can identify 100 programmers with 96% accuracy, we can de-anonymize 600 programmers with 83% accuracy using the same reduced set of features. 83% is a very high number for such a challenging task where the random chance of correctly identifying an author is 0.17%.

VIII. Limitations

Our experiments suggest that our method is able to assist in de-anonymizing a much larger set of programmers with significantly higher accuracy than state-of-the-art approaches. However, there are also assumptions that underlie the validity of our experiments as well as inherent limitations of our method which we discuss in the following paragraphs. First, we assume that our ground truth is correct, but in reality programs in GCJ or on GitHub might be written by programmers other than the stated programmer, or by multiple programmers. Such a ground truth problem would cause the classifier to train on noisy models which would lead to lower de-anonymization accuracy and a noisy representation of programming style. Second, many source code samples from GCJ contestants cannot be compiled. Consequently, we perform evaluation only on the subset of samples which can be compiled. This has two effects: first, we are performing attribution with fewer executable binary samples than the number of available source code samples. This is a limitation for our experiments but it is not a limitation for an attacker who first gets access to the executable binary instead of the source code. If the attacker gets access to the source code instead, she could perform regular source code authorship attribution. Second, we must assume that whether or not a code sample can be compiled does not correlate with the ease of attribution for that sample. Third, we mainly focus on C/C++ code compiled (except Nulled.IO samples) using the GNU compiler `gcc` in this work, and assume that the executable binary format is the Executable and Linking Format. This is important to note as dynamic symbols are typically present in ELF binary files even after stripping of symbols, which may ease the attribution task relative to other executable binary formats that may not contain this information. We defer an in depth investigation of the impact that other compilers, languages, and binary formats might have on the attribution task to future work.

Finally, while we show that our method is capable of dealing with simple binary obfuscation techniques, we do not consider binaries that are heavily obfuscated to hinder reverse engineering. While simple systems, such as packers [2] or encryption stubs that merely restore the original executable binary into memory during execution may be analyzed by simply recovering the unpacked or decrypted executable binary from memory, more complex approaches are becoming increasingly commonplace. A wide range of anti-forensic techniques exist [19], including methods that are designed specifically to prevent easy access to the original bytecode in memory via such techniques as modifying the process environment block or triggering decryption on the fly via guard pages. Other techniques such as virtualization [3] transform the original bytecode to emulated bytecode running on virtual machines, making decompilation both labor-intensive and error-prone. Finally, the use of specialized compilers that lack decompilers and produce nonstandard machine code (see [17] for an extreme but illustrative example) may likewise hinder our approach, particularly if the compiler is not available and cannot be fingerprinted. We leave the examination of these techniques, both with respect to their impact on authorship attribution and to possible mitigations, to future work.

IX. Conclusion

De-anonymizing programmers has direct implications for privacy and anonymity. The ability to attribute authorship to anonymous executable binaries has applications in software forensics, and is an immediate concern for programmers that would like to remain anonymous. We show that coding style is preserved in compilation, contrary to the belief that compilation wipes away stylistic properties. We de-anonymize 100 programmers from their executable binaries with 96% accuracy, and 600 programmers with 83% accuracy. Moreover, we show that we can de-anonymize GitHub developers or hacker forum members with high accuracy. Our work, while significantly improving the limited approaches in programmer de-anonymization, presents new methods to de-anonymize programmers in the wild from challenging real-world samples.

We discover a small set of features that effectively represent coding style in executable binaries. We obtain this precise representation of coding style via two different disassemblers, control flow graphs, and a decompiler. With this comprehensive representation, we are able to re-identify GitHub authors from their executable binary samples in the wild, where we reach an accuracy of 65% for 50 programmers, even though these samples are noisy and products of collaborative efforts.

Programmer style is embedded in executable binary to a surprising degree, even when it is obfuscated, generated with aggressive compiler optimizations, or symbols are stripped. Compilation, binary obfuscation, optimization, and stripping of symbols reduce the accuracy of stylistic analysis but are not effective in anonymizing coding style.

In future work, we plan to investigate snippet and function level stylistic information to de-anonymize multiple authors of collaboratively generated binaries. We also defer the analysis of highly sophisticated compilation and obfuscation methods to future work. Nevertheless, we show that identifying stylistic information is prevalent in real-world settings and accordingly developers cannot assume to be anonymous unless they take extreme precautions as a countermeasure. Examples to possible countermeasures include a combination of randomized coding style, different programming language usage, and employment of indeterministic set of obfuscation methods. Since incorporating different languages or obfuscation methods is not always practical, especially in open source software, our future work would focus on completely stripping stylistic information from binaries to render them anonymous.

We also plan to look at different real-world executable binary authorship attribution cases, such as identifying authors of malware, which go through a mixture of sophisticated obfuscation methods by combining polymorphism and encryption. Our results so far suggest that while stylistic analysis is unlikely to provide a “smoking gun” in the malware case, it may contribute significantly to attribution efforts.

Moreover, we show that attribution is sometimes possible with only small amounts of training binaries, however, having more binaries for training helps significantly. In addition, we observe that advanced programmers (as measured by progression in the GCJ contest) can be attributed more easily than their less skilled peers. Our results present a privacy threat for people who would like to release binaries anonymously.

References

[1] "Hex-rays decompiler," November 2015. [Online]. Available: https://www.hex-rays.com/

[2] "Upx: the ultimate packer for executables," upx.sourceforge.net, November 2015. [Online]. Available: http://upx.sourceforge.net/

[3] "Oreans technology: Code virtualizer," 2015 November. [Online]. Available: http://www.oreans.com/codevirtualizer.php

[4] "The github repository hosting service," http://www.github.com, visited, November 2015.

[5] "Google Code Jam Programming Competition," code.google.com/codejam, visited, November 2015.

[6] S. Afroz, M. Brennan, and R. Greenstadt, "Detecting hoaxes, frauds, and deception in writing style online," in *Proc. of IEEE Symposium on Security and Privacy*. IEEE, 2012.

[7] S. Afroz, A. Caliskan-Islam, A. Stolerman, R. Greenstadt, and D. McCoy, "Doppelgänger finder: Taking stylometry to the underground." in *Proc. of IEEE Symposium on Security and Privacy*, 2014.

[8] A. Aiken *et al.*, "Moss: A system for detecting software plagiarism," *University of California–Berkeley. See www. cs. berkeley. edu/aiken/moss. html*, vol. 9, 2005.

[9] S. Alrabaee, N. Saleem, S. Preda, L. Wang, and M. Debbabi, "Oba2: an onion approach to binary code authorship attribution," *Digital Investigation*, vol. 11, 2014.

[10] E. Backer and P. van Kranenburg, "On musical stylometry—a pattern recognition approach," *Pattern Recognition Letters*, vol. 26, no. 3, pp. 299–309, 2005.

[11] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering." in *NDSS*, vol. 9. Citeseer, 2009, pp. 8–11.

[12] M. Brennan, S. Afroz, and R. Greenstadt, "Adversarial stylometry: Circumventing authorship recognition to preserve privacy and anonymity," *ACM TISSEC*, vol. 15, no. 3, pp. 12–1, 2012.

[13] S. Burrows and S. M. Tahaghoghi, "Source code authorship attribution using n-grams," in *Proc. of the Australasian Document Computing Symposium*, 2007.

[14] S. Burrows, A. L. Uitdenbogerd, and A. Turpin, "Application of information retrieval techniques for source code authorship attribution," in *Database Systems for Advanced Applications*, 2009.

[15] ——, "Comparing techniques for authorship attribution of source code," *Software: Practice and Experience*, vol. 44, no. 1, pp. 1–32, 2014.

[16] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code stylometry," in *Proc. of the USENIX Security Symposium*, 2015.

[17] C. Domas, "M/o/vfuscator," November 2015. [Online]. Available: https://github.com/xoreaxeaxeax/movfuscator

[18] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "Liblinear: A library for large linear classification," *Journal of Machine Learning Research (JMLR)*, vol. 9, 2008.

[19] P. Ferrie, "Anti-unpacker tricks–part one." Virus Bulletin (2008): 4.

[20] G. Frantzeskou, E. Stamatatos, S. Gritzalis, and S. Katsikas, "Effective identification of source code authors using byte-level information," in *Proc. of the International Conference on Software Engineering*. ACM, 2006.

[21] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *SIGKDD Explor. Newsl.*, vol. 11, 2009.

[22] M. A. Hall, "Correlation-based feature selection for machine learning," Ph.D. dissertation, The University of Waikato, 1999.

[23] E. R. Jacobson, N. Rosenblum, and B. P. Miller, "Labeling library functions in stripped binaries," in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, 2011.

[24] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *Proc. of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15*, 2015.

[25] A. Keromytis, "Enhanced attribution," DARPA-BAA-16-34, 2016.

[26] J. Kothari, M. Shevertalov, E. Stehle, and S. Mancoridis, "A probabilistic approach to source code authorship identification," in *Information Technology, 2007. ITNG'07.* IEEE, 2007.

[27] J. Lafferty, A. McCallum, and F. C. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," 2001.

[28] R. C. Lange and S. Mancoridis, "Using code metric histograms and genetic algorithms to perform author identification for software forensics," in *Proceedings of the Annual Conference on Genetic and Evolutionary Computation*. ACM, 2007.

[29] M. Marquis-Boire, M. Marschalek, and C. Guarnieri, "Big game hunting: The peculiarities in nation-state malware research," in *Proc. of Black Hat USA*, 2015.

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

[31] T. C. Mendenhall, "The characteristic curves of composition," *Science*, pp. 237–249, 1887.

[32] A. Narayanan, H. Paskov, N. Z. Gong, J. Bethencourt, E. Stefanov, E. C. R. Shin, and D. Song, "On the feasibility of internet-scale author identification," in *Proc. of IEEE Symposium on Security and Privacy*, 2012.

[33] pancake, "Radare," radare.org, visited, October 2015.

[34] B. N. Pellin, "Using classification techniques to determine source code authorship," 2000.

[35] A. Pfeffer, C. Call, J. Chamberlain, L. Kellogg, J. Ouellette, T. Patten, G. Zacharias, A. Lakhotia, S. Golconda, J. Bay *et al.*, "Malware analysis and attribution using genetic information," in *2012 7th International Conference on Malicious and Unwanted Software*. IEEE, 2012.

[36] J. Quinlan, "Induction of decision trees," *Machine learning*, 1986.

[37] N. A. Quynh, "Capstone," capstone-engine.org, visited, October 2015.

[38] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and classification of malware behavior," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2008.

[39] N. Rosenblum, X. Zhu, and B. Miller, "Who wrote this code? Identifying the authors of program binaries," *ESORICS*, 2011.

[40] N. Rosenblum, B. P. Miller, and X. Zhu, "Recovering the toolchain provenance of binary code," in *Proc. of the International Symposium on Software Testing and Analysis*. ACM, 2011.

[41] N. E. Rosenblum, B. P. Miller, and X. Zhu, "Extracting compiler provenance from program binaries," in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 2010.

[42] N. Science and T. Council", "Federal cybersecurity research and development strategic plan," whitehouse.gov/files/documents, 2016.

[43] A. Stolerman, R. Overdorf, S. Afroz, and R. Greenstadt, "Classify, but verify: Breaking the closed-world assumption in stylometric authorship attribution," in *Working Group 11.9 on Digital Forensics*. IFIP, 2014.

[44] S. Tatham and J. Hall, "The netwide disassembler: NDISASM," http://www.nasm.us/doc/nasmdoca.html, visited, October 2015.

[45] P. van Kranenburg, "Composer attribution by quantifying compositional strategies." in *ISMIR*, 2006.

[46] W. Wisse and C. Veenman, "Scripting dna: Identifying the javascript programmer," *Digital Investigation*, 2015.

[47] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. of IEEE Symposium on Security and Privacy*, 2014.