UNIVERSITY OF CALIFORNIA
RIVERSIDE

A Systematic Approach for Finding and Profiling Malware Source Code in Public Archives

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Md. Omar Faruk Rokon

June 2022

Acknowledgments

I am grateful to my advisor, Prof. Michalis Faloutsos without whom this journey would not have been possible. First and foremost, I would like to thank him for giving the opportunity to begin my Ph.D. here. I found him as the most supportive and patient throughout the time. His critical thinking and life guidance have helped me to grow not only as researcher but also as a better human being. I will carry this learning for the rest of my life and will try to improve myself.

I would also like to express my gratitude to all my committee members, Prof. Nael Abu-Ghazaleh, Prof. Vagelis Papalexakis, and Prof. Manu Sridharan for their time and valuable suggestion in my research which greatly helped me to complete my Ph.D.

I am very much grateful to my family. Their endless support and belief always encouraged me and helped me go through this long journey. I would like to remember my late father Moktal Hossain who would have been the happiest person to see me finish my Ph.D. I will always be indebted to my mother Rokeya Khatun, my big brother Asaduzzaman Asad, and my dear wife Nishat Ara Tania for their unconditional and unlimited support.

I can not forget my close friends in Riverside who were there for me. They have gone out of their way to drag me into their life, accepted me as their friend and for that, I will always be grateful to them. Especially, Risul Islam, C M Sabbir Ahmed, Jawad Bappy, and Rimi Apu - you guys have supported me and loved me beyond my imagination. I will cherish your friendship for the rest of my life.

Finally, my labmates, Dr. Joobin, Dr. Darki, Dr. Shaila, Dr. Jakapun, Dr. Sina, Masud, Ben, and Arman. Thank you for all your support and help. I could not have asked for any better lab-mates and friends than you. You have accepted me and valued me more than I deserve. Thank you very much and I hope we all will stay connected just like we are now.

For everyone else I haven't mention, and/or I can not mention, thank you, everyone. This journey was never easy and it is impossible to walk alone in this path. I stand where I am because of all of you. I am grateful and I thank you. May God bless all of you.

To my lovely wife, Nishat and my family for all the love and support.

ABSTRACT OF THE DISSERTATION

A Systematic Approach for Finding and Profiling Malware Source Code in Public Archives

by

Md. Omar Faruk Rokon

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2022
Dr. Michalis Faloutsos, Chairperson

How can we find malware source code and establish the similarity, influence, and phylogeny of these malware? This question is motivated by a real need: there is a dearth of malware source code, which impedes various types of security research. Our work is driven by the following insight: public archives, like GitHub, have a surprising number of malware repositories. This thesis spans three interrelated problems in this space. First, we address the problem of scarcity of malware source code. We propose, SourceFinder, a supervised-learning approach to identify repositories of malware source code efficiently. We evaluate and apply our approach using 97K repositories from GitHub. Second, we propose Repo2Vec, a comprehensive embedding approach to represent a repository as a distributed vector by combining features from three types of information sources. As our key novelty, we consider three types of information: (a)metadata, (b) the structure of the repository, and (c) the source code. It enables ML techniques for similarity identification, clustering, and classification tasks. We evaluate our approach with 1013 java repositories to find similarities and clusters among them. Third, we propose PIMan, a systematic approach to quantify the influence among the repositories in a software archive by focusing on the social level interactions. We introduce the concept of Plausible Influence which considers three types of information: (a) repository level interactions, (b) author level interactions, and (c) temporal considerations. We evaluate and apply our method using 2089 malware repositories from GitHub spanning approximately 12 years. In our thesis, we use the data from GitHub and three security forums. We show that our approach, SourceFinder identifies malware repositories with 89% precision and 86% recall using a labeled dataset. We use SourceFinder to identify 7504 malware source code repositories, which arguably constitute the largest malware source code database. Second, we show that our method outperforms previous methods in terms of precision (93%vs 78%), with nearly twice as many Strongly Similar repositories and 30% fewer False Positives. We show how Repo2Vec provides a solid basis for: (a) distinguishing between malware and benign repositories, and (b) identifying a meaningful hierarchical clustering. For example, we achieve 98% precision and 96% recall in distinguishing malware and benign repositories. We study the social level interaction between two repositories and establish a plausible influence network among them. We find that there is a significant collaboration and influence among the repositories in our dataset. We argue that our approach and our large repository of malware source code can be a catalyst for research studies, which are currently not possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"Where can we find malware source code?" This question is motivated by a real need: there is a dearth of malware source code, which impedes various types of security research. Security researcher are always searching for an an extensive database of malware source code for various real need, which is currently unavailable. Security researchers can leverage malware source code to: (a) understand malware behavior and techniques, and (b) evaluate security methods and tools. Also, the source code can be used to create the groundtruth for assessing the effectiveness of different techniques, such as reverse engineering methods.

Surprisingly, open source software archives, like GitHub, host many publicly-accessible malware repositories. Security community has a fairly limited idea about the opportunity to find malware source code and they are yet to explore the platform. In this work, we focus on GitHub which is arguably the largest software storing and sharing platform. As of October 2019, GitHub reports more than 34 million users [67] and more than

32 million public repositories [65]. As we will see later in Chapter 3, there are thousands of repositories that have malware source code, which seem to have escaped the radar of the research community so far. We use a broad definition of malware to include any repository containing software that can participate in compromising devices and supporting offensive, undesirable and parasitic activities ranging from simply script-kiddies to tools and it may contain the new, old, popular, variant of well-known malware to hacking tools or malware framework.

Therefore, the question that we answer in this thesis is: "*How can we find malware source code and establish similarity, influence and phylogeny these malware?*" The goal here is to identify malware source code repositories, and establish similarity and influence among them in a large archive, like GitHub and BitBucket. The input to the problem is an online archive and the desired output is a database of malware repositories. The problem consists of challenging sub-problems: (a) collecting an appropriate set of repositories from the potentially vast archive, (b) identifying the repositories that contain malware, and (c) profiling the repositories, for example, determining the similarity among the repositories, establishing the influence and phylogeny of malware. In more detail, there are three distinct types of information for each repository: (a) metadata (title, description, etc), (b) the actual source code, and (c) the project directory structure. The problem statement can be captured well in the following motivating questions:

Question 1: How can we identify malware repositories in online software archives?

Question 2: How can we profile the malware repositories, in terms of determining similarity?

Question 3: How can we establish plausible influence among the malware repositories?

We face several practical challenges in this thesis. First, we need to collect an appropriate set of repositories from the potentially vast archive. Second, repository contains unstructured and heterogeneous data. There is a lot of "noise", lack of structure, and an abundance of informal and hastily written text. The key challenge is to identify appropriate data in a repository. Third, there is no ground-truth dataset. We need to create the ground truth for evaluating the identification and classification methods. Fourth, another practical challenge here is to represent the repository data into a numeric feature vectors to enable ML approaches to compute the similarities and cluster among repositories. In addition, combining vectors from different types of information, as we will do here, is also a challenge. Also, another key challenge is developing a comprehensive approach for defining influence, and even further, estimating the possibility that such an influence has occurred. Combining these different types of information is a non-trivial task.

There is limited work for the problems as defined above. Malware repositories have only been used opportunistically, and there has not been a study that focuses on a systematic identification of malware repositories in online archives. There are several efforts that maintain malware binary databases [5, 9] and extract higher-level information from binaries [34, 53]. There are relatively few efforts [103, 133, 148, 205, 236] that focus on establishing similarity between repositories, and most of them use either metadata or source code level information, while none of them use the three types of information or provide embedding which enables downstream ML tasks. We elaborate on previous works later.

Our work is arguably the first effort to systematically identify malware source code repositories from a massive public archive like GitHub. The key contributions in this thesis are as follows. First, we propose a systematic approach, SourceFinder, to detect malware source code repositories in public archive. We describe this in Chapter 3. Second, we design and develop an embedding approach, Repo2Vec, to present the GitHub repositories in a fixed length vector, and to quantify similarities between two repositories. We discuss this in details in Chapter 4. Third, we propose PIMan, a systematic approach to establish influence and phylogeny among a set of malware repositories in GitHub. We include this in Chapter 5. Our key results are summarized in the following points.

(a) **We identify 7504 malware source-code repositories** with 89% precision, which is arguably the largest malware source-code database available to the research community. We also create a curated database of 250 malware repositories, manually verified and spanning a wide range of malware families. We study the fundamental properties and trends of the malware repositories and their authors. The number of such repositories appears to be growing by an order of magnitude every 4 years, and 18 malware authors seem to be "professionals" with a well-established online reputation.

(b) **Repo2Vec outperforms prior works by a huge margin.** We show that our method outperforms previous methods in terms of precision (93% vs 78%), with nearly twice as many Strongly Similar repositories and 30% fewer False Positives. We show how Repo2Vec provides a solid basis for (a) distinguishing between malware and benign repositories, and (b) identifying a meaningful hierarchical clustering. For example, we achieve 98% precision and 96% recall in distinguishing malware and benign repositories.

(c) **PIMan models influence flexibly with a directed graph.** Our approach captures repository-level influence relationships with a flexible and informative plausible influence graph (PIGraph) which correlates with code-level similarity. We observe significant collaboration and influence among the repositories in our dataset. Analyzing the influence graph, we can find interesting lineage and clusters of influence. We find 19 repositories that influenced at least 10 other repositories directly and spawned at least two "families" of repositories.

We develop a systematic suit of capabilities for studying the malware repository in online platforms. Our tools are capable of: (a) identifying malware repositories, and (b) establishing similarity and influence among repositories. We present the largest malware source code database with 7504 repositories to the security research community. We also provide an embedding approach, Repo2Vec to enable ML for any repository analysis tasks. Follow up research can expand on our work to (a) identify more malware source code repositories in GitHub, GitLab, Gitee, and so on, (b) extensively classify the malware dataset to families and target platform, (c) extend Repo2Vec capabilities for other programming language such as Python, C/C++, etc. repositories. We believe that our methods can be seen as a great foundation for security researchers who are always looking for malware source code.

Chapter 2

Our Datasets

Our work focuses on repository data from GitHub, the largest software archive with roughly 30 million public repositories. We provide background information on GitHub and the type of information that repositories have.

GitHub is a massive world-wide software archive, which enables users to share code through its public repositories thus creating a global social network of interaction. For instance, first, users can collaborate on a repository. Second, users often "fork" projects: they copy and evolve projects. Third, users can follow projects, and "up-vote" projects using "stars" (think Facebook likes). Although GitHub has many private repositories, there are more than 32 million public software repositories.

We describe the key elements of a GitHub repository. A repository is equivalent to a project folder, and typically, each repository corresponds to a single software project. However, a repository could contain: (a) source code, (b) binary code, (c) data, (d) documents, such as latex files, and (e) all of the above.

A repository in GitHub has the following data fields: a) title, b) description, c) topics, d) README file, e) file and folders, f) date of creation and last modified, g) forks, h) watchers, i) stars, and j) followers and followings, which we explain below.

a. Repository title: The title is a mandatory field and it usually consists of less than 3 words.

b. Repository description: This is an optional field that describes the objective of the project and it is usually 1-2 sentences long.

c. Repository topics: An author can optionally provide topics for her repository, in the form of tags, for example, *"linux, malware, malware-analysis, anti-virus"*. Note that 97% of the repositories in our dataset have less than 8 topics.

d. README file: As expected, the README file is a documentation and/or light manual for the repository. This field is optional and its size varies from one or two sentences to many paragraphs. For example, we found that 17.48% of the README files in our repositories are empty.

e. File and folders: In a well-constructed software, the file and folder names of the source code can provide useful information. For example, some malware repositories contain files or folders with indicative names, such as "malware", "source code" or even specific malware types or names of specific malware, like *mirai*.

f. Date of creation and last modification: GitHub maintains the date of creation and last modification of a repository. We find malware repository created in 2008 are actively being modified by authors till present.

g. Number of forks: Users can fork a public repository: they can create a clone of the project. An user can fork any public repository to change locally and contribute to the original project if the owner accepts the modification. The number of forks is an indication of the popularity and impact of a repository. Note that the number of forks indicates the number of distinct users that have forked a repository.

h. Number of watchers: Watching a repository is equivalent to "following" in the social media language. A "watcher" will get notifications, if there is any new activity in that project. The numbers of watchers is an indication of the popularity of a repository [45].

i. Number of stars: A user can "star" a repository, which is equivalent to the "like" function in social media [16], and places the repository in the users favorite group, but does not provide constant updates as with the "watching" function.

j. Followers: Users can also follow other users' work. If A follows B, A will be added to B's followers and B will be added to A's following list. The number of followers is an indication of the popularity of a user [115].

Chapter 3

SourceFinder: Finding Malware Source-Code from Publicly Available Repositories in GitHub

Security research could greatly benefit by an extensive database of malware source code, which is currently unavailable. This is the assertion that motivates this work. First, security researchers can use malware source code to: (a) understand malware behavior and techniques, and (b) evaluate security methods and tools. In the latter, having the source code can provide the groundtruth for assessing the effectiveness of different techniques, such as reverse engineering methods. Second, currently, a *malware source code* database is not readily available. By contrast, there are several databases with *malware binary code*, as collected via honeypots, but even those are often limited in number and not widely available. We discuss existing malware archives in Section 3.7.

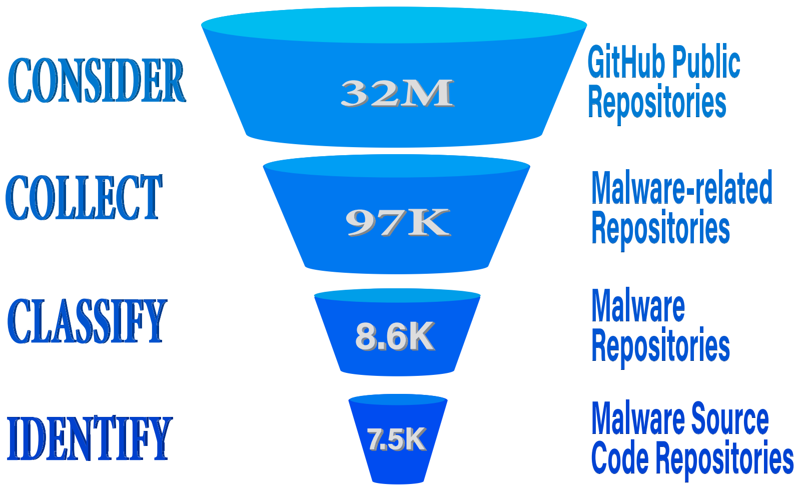

CONSIDER
32M
GitHub Public Repositories
COLLECT
97K
Malware-related Repositories
CLASSIFY
8.6K
Malware Repositories
IDENTIFY
7.5K
Malware Source Code Repositories

Figure 3.1: The steps of our work as a funnel: We identify 7.5K malware source code repositories in GitHub starting from 32M repositories based on 137 malware keywords (Q137).

A missed opportunity: Surprisingly, software archives, like GitHub, host many publicly-accessible malware repositories, but this has not yet been explored to provide security researchers with malware source code. In this work, we focus on GitHub which is arguably the largest software storing and sharing platform. As of October 2019, GitHub reports more than 34 million users [67] and more than 32 million public repositories [65]. As we will see later, there are thousands of repositories that have malware source code, which seem to have escaped the radar of the research community so far. We use a broad definition of malware to include any repository containing software that can participate in compromising devices and supporting offensive, undesirable and parasitic activities.

Why do authors create public malware repositories? This question mystified us: these repositories expose both the creators and the intelligence behind the malware. Intrigued, we conducted a small investigation on malware authors, as we discuss below.

Problem: How can we find malware source code repositories in a large archive, like GitHub? The input to the problem is an online archive and the desired output is a database of malware repositories. The challenges include: (a) collecting an appropriate set of repositories from the potentially vast archive, and (b) identifying the repositories that contain malware. Optionally, we also want to further help researchers that will potentially use these repositories, by determining additional properties, such as the most likely target platform, the malware type or family etc. Another practical challenge is the need to create the ground truth for validation purposes.

Related work: To the best of our knowledge, there does not seem to be any study focusing on the problem above. We group related work in the following categories. First, several studies analyze software repositories to find usage and limitations without any focus on malware [43]. Second, several efforts create and maintain databases of malware binaries but without source code [5, 9]. Third, many efforts attempt to extract higher-level information from binaries, such as lifting to Intermediate Representation (IR) [53], but it is really difficult to re-create the source code [34]. In fact, such studies would benefit from our malware source-code archive to evaluate and improve their methods. Taking a software engineering angle, an interesting work [27] compares the evolution of 150 malware source code repositories with that of benign software. We discuss related work in Section 3.7.

Contributions: Our work is arguably the first effort to systematically identify malware source code repositories from a massive public archive. The contribution of this work is three-fold: (a) we propose SourceFinder, a systematic approach to identify malware source-code repositories with high precision, (b) we create, arguably, the largest non-commercial malware source code archive with 7504 repositories, and (c) we study patterns and trends of the repository ecosystem including temporal and author-centric properties and behaviors. We apply and evaluate our method on the GitHub archive, though it could also be used on other archives, as we discuss in Section 3.6.

Our key results can be summarized in the following points, and some key numbers are shown in Figure 3.1.

a. We collect **97K malware-related repositories** from GitHub, namely repositories retrieved using malware keywords through GitHub's API and employing techniques to overcome several limitations. We also generate an extensive ground-truth with 2013 repositories, as we explain in Section 3.1.

b. **SourceFinder achieves 89% precision.** We systematically consider different Machine Learning approaches, and carefully-created representations for the different fields of the repository, such as title, description etc. We then systematically evaluate the effect of the different features, as we discuss in Section 3.3. We show that we classify malware repositories with a 89% precision, 86% recall and 87% F1-score using five fields from the repository.

c. **We identify 7504 malware source-code repositories**, which is arguably the largest malware source-code database available to the research community. We have already downloaded the contents in these repositories, in case GitHub decides to deactivate them. We also create a curated database of 250 malware repositories, manually verified and spanning a wide range of malware types.

d. **The number of new malware repositories in our data more than**

triples every four years. The increasing trend is interesting and alarming at the same time.

e. **We identify popular and influential repositories.** We study the malware repositories using three metrics of popularity: the number of watchers, forks and stars. We find 8 repositories that dominate the top-5 lists for all three metrics.

f. **We identify prolific and influential authors.** We find that 3% of the authors have more than 300 followers. We also find that 0.2% of the authors have more than 7 malware repositories, with the most prolific author *cyberthreats* having created 336 repositories.

g. **We identify and profile 18 professional hackers.** We find 18 authors of malware repositories, who seem to have created a brand around their activities, as they use the same user names in security forums. For example, user *3vilp4wn* (pronounced evil-pawn) is the author of a keylogger malware in GitHub, which the author is promoting in the *Hack This Site* forum using the same username. We present our study of malware authors in Section 3.5.

Open-sourcing for maximal impact: creating an engaged community. We intend to make our datasets and our tools available for research purposes at our website [74]. Our vision is to create community-driven reference platform, which will provide: (a) malware source code repositories, (b) community-vetted labels and feedback, and (c) open-source tools for collecting and analyzing malware repositories. Our goal is to expand our database with more software archives and richer information. Although authors could start hiding their repositories (see Section 3.6), we argue that our already-retrieved database could have significant impact in enabling certain types of security studies [59, 77, 95].

3.1 Data Collection

The first step in our work is to collect repositories from GitHub that have a higher chance of being related to malware. Extracting repositories at scale from GitHub hides several subtleties and challenges, which we discuss below. Using the GitHub Search API, a user can query with a set of keywords and obtain the most relevant repositories. We describe briefly how we select appropriate keywords, retrieve related repositories from GitHub and how we establish our ground truth.

Set	**Descriptions**	**Size**
Q1	Query set = {"malware"}	1
Q50	Query with 50 keywords with Q1⊂Q50	50
Q137	Query with 137 keywords with Q50⊂Q137	137
RD1	Retrieved repositories from query Q1	2775
RD50	Retrieved repositories from query Q50	14332
RD137	Retrieved repositories from query Q137	97375
LD1	Labeled subset of RD1 dataset	379
LD50	Labeled subset of RD50 dataset	755
LD137	Labeled subset of RD137 dataset	879
M1	Malware source code repositories in RD1	680
M50	Malware source code repositories in RD50	3096
M137	Malware source code repositories in RD137	7504
MCur	Manually verified malware source code dataset	250

Table 3.1: Datasets, their relationships, and their size.

A. Selecting keywords for querying: In this step, we want to retrieve repositories from GitHub in a way that: (a) provides as many as possible malware repositories, and (b) provides a wide coverage over different types of malware. For this reason, we select keywords from three categories: (a) malware and security related keywords, such as malware and virus, (b) malware type names, such as ransomware and keylogger, and (c) popular malware names, such as mirai. Due to space limitations, we will provide the full list of keywords in our website at publication time for repeatability purposes.

We define three sets of keywords that we use to query GitHub. The reason is that we want to assess the sensitivity of the number of keywords on the outcome. Specifically, we use the following query sets: (a) the **Q1 set**, which only contains the keyword "malware"; (b) the **Q50 set**, which contains 50 keywords, and (c) the **Q137 set** which contains 137 keywords. The Q137 keyword set is a super-set of Q50, and Q50 is a superset of Q1. As we will see below, using the query set Q137 provides wider coverage, and we recommend in practice. We use the other two to assess the sensitivity of the results in the initial set of keywords. We list our datasets in Table 3.1.

B. Retrieving related repositories: Using the Search API, we query GitHub with our set of keywords. Specifically, we query GitHub with every keyword in our set separately. In an ideal world, this would have been enough to collect all related repositories: a query with "malware" (Q1) should return the many thousands related repositories, but this is not the case.

The search capability hides several subtleties and limitations. First, there is a limit of 1000 repositories that a single search can return: we get the top 1000 repositories ordered by relevancy to the query. Second, the GitHub API allows 30 requests per minute for an authenticated user and 10 requests per minute for an unauthenticated user.

Bypassing the API limitations. We were able to find a work around for the first limitation by using ranking option. Namely, a user can specify her preferred ranking order for the results based on: (a) best match, (b) most stars, (c) fewest stars, (d) most forks, (e) fewest forks, (f) most recently updated, and (g) the least recently updated order. By repeating a query with all these seven ranking options, we can maximize the number of distinct repositories that we get. This way, for each keyword in our set, we search with these seven different ranking preferences to obtain a list of GitHub repositories.

C. Collecting the repositories: We download all the repositories identified in our queries using PyGithub [169], and we obtain three sets of repositories RD1, RD50 and RD137. These retrieved datasets have the same "subset" relationship that they query sets have: RD1 $\subset$ RD50 $\subset$ RD137. Note that we remove pathological repositories, mainly repositories with no actual content, or repositories "deleted" by GitHub. For each repository, we collect and store: (a) repository-specific information, (b) author-specific information, and (c) all the code within the repository.

As we see from Table 3.1, using more and specialized malware keywords returns significantly more repositories. Namely, searching with the keyword "malware" does return 2775 repositories, but searching with the Q50 and Q137 returns 14332 and 97375 repositories respectively.

D. Establishing the groundtruth: As there was no available groundtruth, we needed to establish our own. As this is a fairly technical task, we opted for domain experts

Labeled Dataset	**Malware Repo.**	**Benign Repo.**
LD137	313	566
LD50	326	429
LD1	186	193

Table 3.2: Our groundtruth: labeled datasets for each of the three queries, for a total of 2013 repositories.

instead of Mechanical Turk users, as recommended by recent studies [62]. We use three computer scientists to manually label 1000 repositories, which we selected in a uniformly random fashion, from each of our dataset RD137 and RD50 and 600 repositories from RD1. The judges were instructed to independently investigate every repository thoroughly.

Ensuring the quality of the groundtruth. To increase the reliability of our groundtruth, we took the following measures. First, we asked judges to label a repository *only*, if they were certain that it is malicious or benign and distinct, and leave it unlabeled otherwise. We only kept the repositories for which the judges agreed unanimously. Second, duplicate repositories were removed via manual inspection, and were excluded from the final labeled dataset to avoid overfitting. It is worth noting that we only found very few duplicates in the order of 3-5 in each dataset with hundreds of repositories.

With this process, we establish three separate labeled datasets named LD137, LD50, and LD1 starting from the respective malware repositories from each of our queries, as shown in Table 3.2. Although the labeled datasets are not 50-50, they are representing both classes reasonably well, so that a naive solution that will label everything as one class, would perform poorly. By contrast, our approach performs sufficiently well, as we will see in Section 3.3.

As there is no available dataset, we argue that we make a sufficient size dataset by manual effort.

3.2 Overview of our Identification Approach

Here, we describe our supervised learning algorithm to identify the repositories that contain malware.

Step 1. Data preprocessing: As in any Natural Language Processing (NLP) method, we start with some initial processing of the text to improve the effectiveness of the solution. We briefly outline three levels of processing functionality.

a. Character level preprocessing: We handle the character level ”noise” by removing special characters, such as punctuation and currency symbols, and fix Unicode and other encoding issues.

b. Word level preprocessing: We eliminate or aggregate words following the best practices of Natural Language Processing [97]. First, we remove article words and other words that don't carry significant meaning on their own. Second, we use a stemming technique to handle inflected words. Namely, we want to decrease the dimensionality of the data by grouping words with the same ”root”. For example, we group the words ”organizing”, ”organized”, ”organize” and ”organizes” to one word ”organize”. Third, we filter out common file and folder names that we do not expect to help in our classification, such as ”LEGAL”, ”LICENSE”, ”gitattributes” etc.

c. Entity level filtering: We filter entities that are likely not helpful in describing the scope of a repository. Specifically, we remove numbers, URLs, and emails, which are often found in the text. We found that this filtering improved the classification performance. In the future, we could consider mining URLs and other information, such as names of people, companies or youtube channels, to identify authors, verify intention, and find more malware activities.

Step 2. The repository fields: We consider fields from the repositories that can be numbers or text. Text-based fields require processing in order to turn them into classification features and we explain this below. We use and evaluate the following text fields: title, description, topics, file and folder names and README file fields.

Text field representation: We consider two techniques to represent each text field by a feature in the classification.

a. Bag of Words (BoW): The bag-of-words (BoW) model is among the most widely used representations of a document. The document is represented as the number of occurrences of its words, disregarding grammar and word order [234]. This model is commonly used in document classification where the frequency of each word is used as feature value for training a classifier [134]. We use the model with the count vectorizer and TF-IDF vectorizer to create the feature vector.

In more detail, we represent each text field in the repository with a vector $V[K]$, where $V[i]$ corresponds to the significance of word i for the text. There are several ways to assign values $V[i]$: (a) zero-one to account for presence, (b) number of occurrences, and (c) the TF-IDF value of the word. We evaluated all the above methods.

Fixing the number of words per field. To improve the effectiveness of our approach using BoW, we conduct a feature selection process, χ^2 statistic following best practices [174].

The χ^2 statistic measures the lack of independence between a word (feature) and a class. A feature with lower chi-square score is less informative for that class, and thus not useful in the classification. We discuss this further in Section 3.3. For each text-based field f, we select the top K_f words for that field, which exhibit the highest discerning power in identifying malware repositories. Note that we set a value for K_f during the training stage For each field, we select the value K_f, as we explain in Section 3.3.

b. Word embedding: The word embedding model is a vector representations of each word in a document: each word is mapped to an M-dimensional vector of real numbers [137], or equivalently are projected in an M-dimensional space. A good embedding ensures that words that are close in meaning have nearby representations in the embedded space. In order to create the document vector, word embedding follows two approaches (i) frequency-based vectorizer(unsupervised) [182] and (ii) content-based vectorizer(supervised) [113]. Note that in this type of representation, we do not use the *word level processing*, which we described in the previous step, since this method can leverage contextual information.

We use frequency-based word embedding with word average and TF-IDF vectorizer. We also use pre-trained model of Google word2vec [136] and Stanford (Glov) [161] to create the feature vector.

Finally, we create the vector of the repository by concatenating the vectors of each field of that repository.

Step 3. Selecting the fields: Another key question is which fields from the repository to use in our classification. We experiment with all of the fields listed in the

Data Collection Section and we explain our findings in the next Section.

Step 4. Selecting a ML engine: We design ML model to classify the repositories into two classes: (i) malware repository and (ii) benign repository. We systematically evaluate many machine learning algorithms [21, 140]: Naive Bayes (NB), Logistic Regression (LR), Decision Tree (CART), Random Forest(RF), K-Nearest Neighbor (KNN), Linear Discriminant Analysis (LDA), and Support Vector Machine (SVM).

Step 5. Detecting source code repositories: In this final step, we want to identify the presence of source code in the repositories. By June 2020, GitHub started labeling repositories that contain source code. Therefore, one can simply filter out all repositories that are not labelled as such.

As our study predates this GitHub feature, we developed a heuristic approach to identify source code repositories independently, which we describe below. Our heuristic exhibits 100% precision as validated by GitHub's classification, as we will see in Section 3.3.

Our source-code classification heuristics works in two steps. First, we identify files in the repository that contain source code. To do this, we start by examining their file extension. If the file extension is one of the known programming languages: *Assembly, C, C++, Batch File, Bash Shell Script, Power Shell Script, Java, Python, C#, Objective-C, Pascal, Visual Basic, Matlab, PHP, Javascript, and Go*, we label it as a source file. Second, if the number of source files in a repository exceeds the **Source Percentage threshold (SourceThresh)**, we consider that the repository contains source code.

3.3 Evaluation: Choices and Results

In this section, we evaluate the effectiveness of the classification based on the proposed methodology defined in Section 3.2. More specifically, our goal here is to answer the following questions:

1. **Repository field selection:** Which repository fields should we consider in our analysis?

2. **Field representation:** Which feature representation is better between bag of words (BoW) and word embedding and considering several versions of each?

3. **Feature selection**: What are the most informative features in identifying malware repositories?

4. **ML algorithm selection:** Which ML algorithm exhibits the best performance?

5. **Classification effectiveness:** What is the precision, recall and F1-score of the classification?

6. **Identifying malware repositories:** How many malware repositories do we find?

7. **Identifying malware source code repository:** How many of the malware repositories have source code?

Note that we have a fairly complex task: we want to identify the best fields, representation method and Machine Learning engine, while considering different values for parameters. What complicates matters is that all these selections are interdependent. We

Representation	**Classification Accuracy Range**
Bag of Words with Count Vectorizer	86%-51%
Bag of Words with Count Vectorizer + Feature Selection	91%-56%
Bag of Words with TF-IDF vectorizer	82%-63%
Word Embedding with Word Average	85%-72%
Word Embedding with TF-IDF	85%-74%
Pretrained Google word2vec Model	76%-64%
Pretrained Stanford (Glov) Model	73%-62%

Table 3.3: Selecting the feature representation model: We evaluate all the representations across seven machine learning approaches and report the range of the overall accuracy.

present our analysis in sequence, but we followed many trial and error and non-linear paths in reality.

1. Selecting repository fields: We evaluated all the repository fields mentioned earlier. In fact, we used a significant number of experiments with different subsets of the features, not shown here due to space limitations. We find that the title, description, topics, README file, and file and folder names have the most discerning power. We also considered number of forks, watchers, and stars of the repository and the number of followers and followings of the author of the repository. We found that not only it did not help, but it usually decreased the classification accuracy by 2-3%. One possible explanation is that the numbers of forks, stars and followers reflect the popularity rather than the content of a repository.

2. Selecting a field representation: The goal is to find, which representation approach works better. In Table 3.3, we show the comparison of the range of classification accuracy across the 7 different ML algorithms that we will also consider below. We find that Bag of Words with the count vectorizer representation reaches 86% classification accuracy, with the word embedding approach nearly matching that with 85% accuracy. Note that we finetune the selection of words to represent each field in the next step.

Why does not the embedding approach outperform the bag of words? One would have expected that the most complex embedding approach would have been the winner and by a significant margin. We attribute this to the relatively small text size in most text fields, which also do not provide well-structured sentences (think two-three words for the title, and isolated words for the topics). Furthermore, the word co-occurrences does not exist in the topics and file names fields, which is partly what makes embedding approaches work well in large and well structured documents [68, 123].

In the rest of this paper, we use the Bag of Words with count vectorizer to represent our text fields, since it exhibits good performance and is computationally less intensive than the embedding method.

3. Fixing the number of words per field. We want to identify the most discerning words from each text field, which is a standard process in NLP for improving the scalability, efficiency and accuracy of a text classifier [36]. Using the χ^2 statistic, we select the top K_f best words from each field.

To select the appropriate number of words per field, we followed the process below. We vary K_f = 5,10,20,30,40 and 50 for title, topic and README file, and we find that the top 30 words in title, 10 words in topic and 10 words in README file exhibit the highest accuracy. Similarly, we try $K_f = 80$, 90, 100, 110 and 120 for file names and $K_f = 300$, 325, 350, 375, 400, 425, 450 and 475 for the description field. We find that the top 100 words for file and folder names and top 400 words for description field give the highest accuracy. Note that we do this during training and refining the algorithm, and then we continue to use these words as features in testing.

Thus, we select the top: (a) 30 words from the title, (b) 10 words from the topics, (c) 400 words from the description, (d) 100 words from the file names, and 10 words from the README file. This leads to a total of 550 words across all fields. For reference, we find 9253 unique words in the repository fields of our training dataset. Reducing the focus on the top 550 most discerning words per field increases the classification accuracy by as much as 20% in some cases.

4. Evaluating and selecting ML algorithms: We find that Multinomial Naive Bayes exhibits the best F1-score with 87%, striking a good balance between 89% precision and 86% recall for the malware class among other machine learning classifier which we considered. Detecting the benign class, we do even better with 92% precision, 94% recall and 93% F1-score. By contrast, the F1-score of the other algorithms is below 79%. Note that KNN, LR and LDA methods provide higher precision, but with significantly lower recall. Thus, one could use these algorithms to get higher precision at the cost of lower total number of repositories.

We use Multinomial Naive Bayes as our classification engine for the rest of this study. We attempt to explain the superior F1-Score of the Naive Bayes in our context. The main advantage of Naive Bayes over other algorithms is that it considers the features

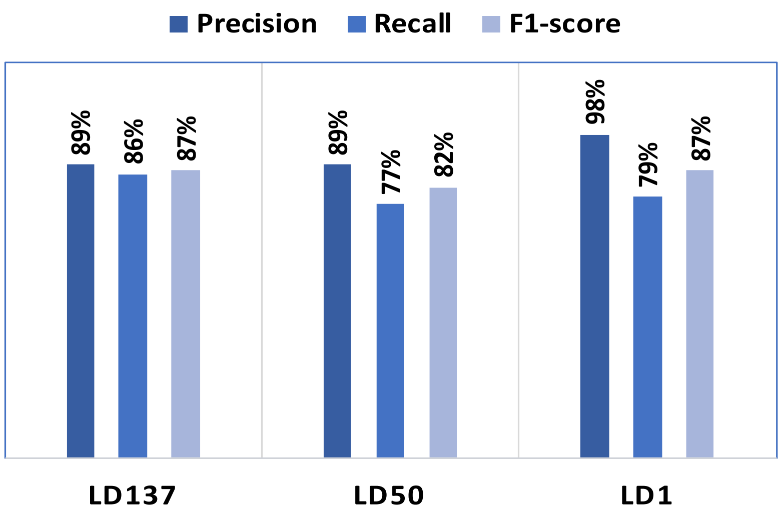


Figure 3.2: Assessing the effect of the number of keywords in the query: Precision, Recall and F1-score of our approach on the LD137, LD50 and LD1 labeled datasets.

independently of each other for a given class and can handle large number of features better. As a result, it is more robust to noisy or unreliable features. It also performs well in domains with many equally important features, where other approaches suffer, especially with a small training data, and it is not prone to overfitting [209]. As a result, the Naive Bayes is considered a dependable algorithm for text classification and it is often used as the benchmark to beat [227].

5. Assessing the effect of the query set: We have made the following choices in the previous steps: (a) 5 text-based fields, (b) bag of words with count vectorization, (c) 550 total words across all the fields, and (d) the Multinomial Naive Bayes. We perform 10-fold cross validation and report the precision, recall and F1-score in Figure 3.2 for our three different labeled data sets. We see that the precision stays above 89% for all three datasets, with a recall above 77%.

Dataset	**Initial**	**Malware**	**Mal. + Source**
RD1	2775	809	680
RD50	14332	3615	3096
RD137	97375	8644	7504

Table 3.4: The identified repositories per dataset with: (a) malware, and (b) malware and source code.

It is worth noting the relative stability of our approach with respect to the keyword set for the initial query especially between LD50 and LD137 datasets. The LD1 dataset we observe higher accuracy, but significantly less recall compared to LD137. We attribute this fact to the single keyword used in selecting the repositories in LD1, which may have lead to a more homogeneous group of repositories. Interestingly, LD50 seem to have the lower recall and F1-score even though the differences are not that large.

6. Identifying 8644 malware repositories: We use LD137 to train our Multinomial Naive Bayes model and apply it on RD137 dataset. We find 8644 malware repositories. We also apply the same trained model on RD1 and RD50 and find 809 and 3615 malware repositories respectively, but this repositories are included in the 8644. (Recall that RD1 and RD50 are subsets of RD137).

7. Identifying 7504 malware source code repositories: As of June 2020, we can use the source code labelling to identify such repositories. Here, we use this labelling to validate our heuristic approach for completeness.

In deploying our heuristic, we set our Source Percentage threshold to 75%, meaning that: if more than 75% of files in a repository are source code files, we label it as a source code repository. Applying this heuristic, we find that 7504 repositories are most likely source code repositories in RD137. We use the name **M137** to refer to this group of malware source code repositories. We find 680 and 3096 malware source code repositories in RD1 and RD50 as shown in Table 3.4. However, these are subset of M137, given that RD1 and RD50 are subsets of RD137.

We find that 100% of our source code repositories are also labeled as such by GitHub. We argue that our heuristic could be useful for other software archives, which may not provide the "source code" label.

8. A curated malware source code dataset- MCur: As a tangible contribution, we provide, MCur, a dataset of 250 repositories from the M137 dataset, which we manually verify for containing malware source code and relating to a particular malware type. Opting for diversity and coverage, the dataset spans all the identified types: virus, backdoor, botnet, keylogger, worm, ransomware, rootkit, trojan, spyware, spoof, ddos, sniff, spam, and cryptominer. We intend to constantly update and make our labeled malware repositories publicly available [74].

3.4 A large scale study of malware

Encouraged by the substantial number of malware repositories, we study the distributions and longitudinal properties of the identified malware repositories in M137.

Caveat: We provide some key observations in this section, but they should be viewed as indicative and approximate trends and only within the context of the collected repositories and with the general assumption that repository titles and descriptions are reasonably accurate. In Section 3.6, we discuss issues around the biases and limitations that our dataset may introduce.

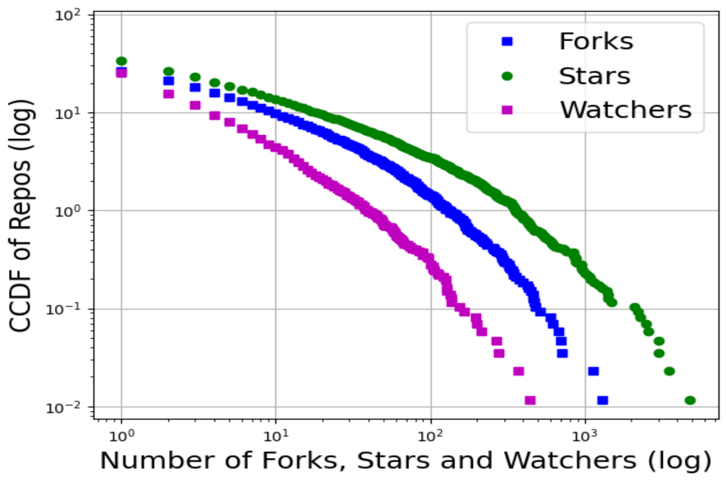


Figure 3.3: CCDF distributions of forks, stars and watchers per repository.

A. Identifying influential repositories. The prominence of a repository can be measured by the number of *forks*, *stars*, and *watchers.* In Figure 3.3, we plot the complementary cumulative distribution function (CCDF) of these three metrics for our malware repositories.

Fork distribution: We find that 2% of the repositories seem quite influential with at least 100 forks as shown in Figure 3.3. Recall that the fork counter indicates the number of distinct users that have forked a repository. For reference, 78% of the repositories have less than 2 forks.

Star distribution: We find that 2% of the repositories receive more than 250 stars as shown in Figure 3.3. For reference, 75% of the repositories have less than 3 stars.

Watcher distribution: In Figure 3.3, we find that 1% of the repositories have more than 50 watchers. For reference, we observe that 84% of the repositories have less

R ID	Author	# Star	# Fork	# Watcher	Content of the Repository
1	ytisf	4851	1393	730	80 malware source code and 140 Binaries
2	n1nj4sec	4811	1307	440	Pupy RAT
3	Screetsec	3010	1135	380	TheFatRat Backdoor
4	malwaredllc	2515	513	268	Byob botnet
5	RoganDawes	2515	513	268	USB attack platform
6	Visgean	626	599	127	Zeus trojan horse
7	Ramadhan	535	283	22	30 malware samples
8	dana-at-cp	1320	513	125	backdoor-apk backdoor

Table 3.5: The profile of the top 5 most influential malware repositories across all three metrics with 8 unique repositories.

than 3 watchers. Note that these distributions are skewed, and follow patterns that can be approximated by a log-normal distribution.

Which are the most influential repositories? We find that 8 repositories dominate the top 5 spots across all three metrics: stars, forks, and watchers. We present a short profile of these dominant repositories in Table 3.5. Most of the repositories contain a single malware project, which is an established practice among the authors in GitHub [152, 212]. We find that the repository ”theZoo” [228], created by *ytisf* in 2014 is the most forked, watched, and starred repository with 1393 forks, 730 watchers and 4851 stars as of October, 2019. However, this repository is quite unique and was created with the intention of being a malware database with 140 binaries and 80 source code repositories.

Influence metrics are correlated: As one would expect, the influence and popularity metrics are correlated. We use a common correlation metric, the Pearson Correlation Coefficient (r) [17], measured in a scale of $[-1, 1]$. We calculate the metric for all pairs of our three popularity metrics. We find that all of them exhibit higher positive correlation: stars vs. forks ($r = 0.92$, $p < 0.01$), forks vs. watchers ($r = 0.91$, $p < 0.01$) and watchers vs. stars ($r = 0.91$, $p < 0.01$).

B. Malware type and target platform. We wanted to get a feel for what type of malware we have identified. As a first approximation, we use the keywords found in the text fields to relate repositories in M137 with the type of malware and the intended target platform. Our goal is to create the two-dimensional distribution per malware type and the target platform as shown in Table 3.6. To create this table, we associate a repository with keywords in its title, topics, descriptions, file names and README file fields of: (a) the 6 target platforms, and (b) the 13 malware type keywords.

How well does this heuristic approach work? We provide two different indications of its relative effectiveness. First, the vast majority of the repositories relate to one platform or type of malware: (a) less than 8% relate to more than one platform, and (b) less than 11% relate to more than one type of malware. Second, we manually verify the 250 repositories in our curated data MCur and find a 98% accuracy.

Below, we provide some observations from Table 3.6.

a. Keyloggers reign supreme. We see that one of the largest categories is the keylogger malware with 679 repositories, which are mostly affiliated with Windows and Linux platforms. We discuss the emergence of keyloggers below in our temporal analysis.

Types	**Target Platform**						
	Wind.	Linux	Mac	IoT	Andr.	iOS	**Total**
Total	**1592**	**1365**	**380**	**108**	**442**	**131**	**4018**
keylogger	396	209	42	2	27	3	**679**
backdoor	181	227	37	11	51	4	**511**
virus	235	131	34	2	51	16	**469**
botnet	153	154	43	36	64	17	**467**
trojan	133	70	24	16	67	19	**329**
spoof	76	115	88	2	20	9	**310**
rootkit	55	163	13	2	19	3	**255**
ransomware	117	67	14	1	33	13	**245**
ddos	71	95	20	10	9	3	**208**
worm	61	45	18	5	25	18	**172**
spyware	45	22	6	6	38	16	**133**
spam	40	29	18	14	23	5	**129**
sniff	29	38	23	1	15	5	**111**

Table 3.6: Distribution of the malware repositories from M137 dataset based on the malware type and malware target platform. This table demonstrates the repositories that fit with the criteria defined in Section 3.4.

b. Windows and Linux are the most popular targets. Not surprisingly, we find that the majority of the malware repositories are affiliated with these two platforms: 1592 repositories for Windows, and 1365 for Linux.

c. MacOS-focused repositories: fewer, but they exist. Although MacOS platforms are less commonly targeted, we find a non-trivial number of malware repositories for MacOS. As shown in Figure 3.6, there are 380 MacOS malware repositories, which is roughly an order of magnitude less compared to those for Windows and Linux.

C. Temporal analysis. We want to study the evolution and the trends of malware repositories. We plot the number of new malware repositories per year: a) total malware in Figure 3.4, b) per type of malware in Figure 3.5, and c) per target platform in Figure 3.6. We discuss a few interesting temporal behaviors below.

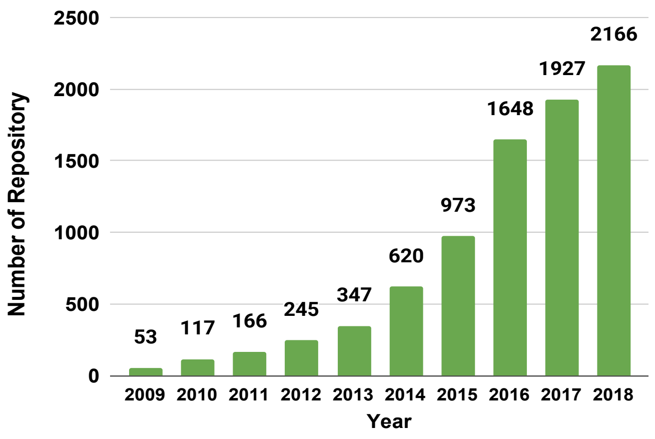


Figure 3.4: New malware repositories created per year.

a. The number of new malware repositories more than triples every four years. We see an alarming increase from 117 malware repositories in 2010 to 620 repositories in 2014 and to 2166 repositories in 2018. We also observe a sharp increase of 70% between 2015 to 2016 shown in Figure 3.4.

b. Keyloggers started a super-linear growth since 2010 and are by far affiliated with the most new repositories per year since 2013, but their rate of growth reduced in 2018.

c. Ransomware repositories emerge in 2014 and gain momentum in 2017. Ransomware experienced their highest growth rate in 2017 with 155 new repositories, while that number dropped to 103 in 2018.

d. Malware activity slowed down in 2018 across the board. It seems that 2018 is a slower year for all malware even when seen by type (Figure 3.5) and target platform (Figure 3.6). We find that the number of new malware repositories has dropped significantly in 2018 for most types of malware except virus, keylogger and trojan.

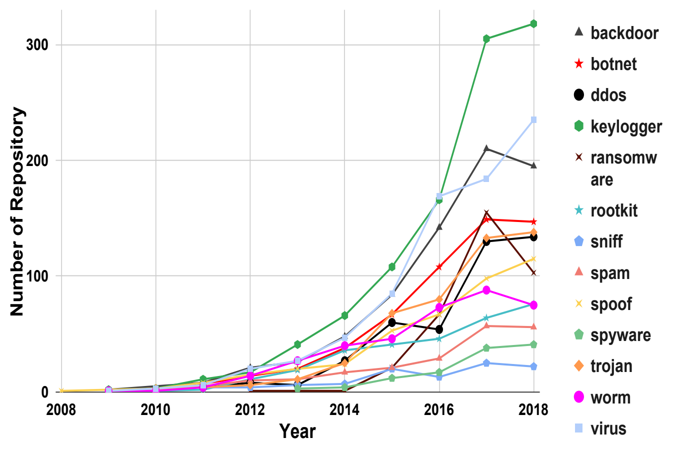


Figure 3.5: New repositories per type of malware per year.

e. IoT and iPhone malware repositories become more visible after 2014. We find that IoT malware emerges in 2015 and iPhone malware sees an increase after 2014 in Figure 3.6. We conjecture that this is possibly encouraged by the emergence and increasing popularity of specific malware: (a) WireLurker, Masque, AppBuyer malware [39] for iPhones, and (b) BASHLITE [222], a Linux based botnet for IoT devices. We find the names of the aferemntioned malware in many repositories starting in 2014. Interestingly, the source code of the original BASHLITE botnet is available in a repository created by *anthonygtellez* in 2015.

f. Windows and Linux: dominant but slowing down. In Figure 3.6, we see that windows and linux malware are flattened between 2017 and 2018. By contrast, IoT and android repositories have increased.

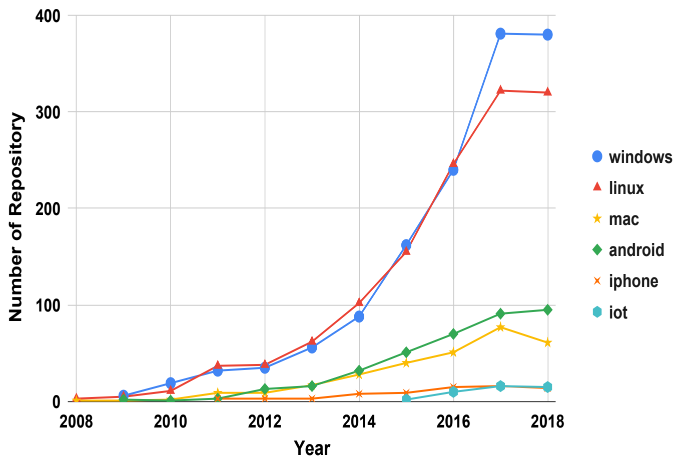


Figure 3.6: New malware repositories per target platform per year.

3.5 Understanding malware authors

Intrigued by the fact that authors create public malware repositories, we attempt to understand and profile their behavior.

As a first step towards understanding the malware authors, we want to assess their popularity and influence. We use the following metrics: (a) number of malware repositories which they created, (b) number of followers, (c) total number of watchers on their repositories, and (d) total number of stars. We focus on the first two metrics here. We use the notation *top k authors* for any of the metrics above, where k can be any positive integer to referring to ”heavy-hitters”.

A. Finding influential malware authors. We study the distribution of the number of malware repositories created and the number followers per author in following.

First, we find that 15 authors are contributing roughly 5% of all malware repositories by examining the CCDF of the created repositories in Figure 3.7. From the figure, we find an outlier author, *cyberthreats*, who doesn't follow power law distribution [57], has

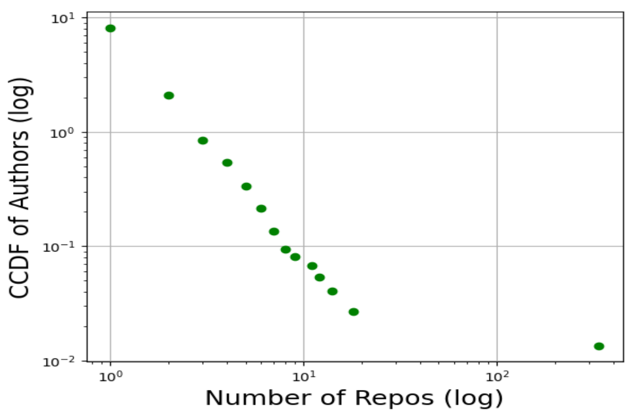


Figure 3.7: CCDF of malware repositories per author.

created 336 malware repositories. We also find that 99% authors have less than 5 repositories.

Second, we study the distribution of the number of followers per author, but omit the plot due to space limitations. The distributions is skewed with 3% (221) of the authors having more than 300 followers each, while 70% of the authors have less than 16 followers.

B. Malware authors strive for an online "brand": In an effort to understand the motive of sharing malware repositories, we make the following investigation.

a. Usernames seem persistent across online platforms. We find that many malware authors use the same username consistently across many online platforms, such as security forums. We conjecture that they are developing a reputation and they use their username as a "unique" identifier.

We identify 18 malware authors[1], who are active in at least one of the three security forums: Offensive Community, Ethical Hacker and Hack This Site, for which we happen to

[1]Note that this does not mean that the other authors are not doing the same, but they could be active in other security forums or online platforms.

have access to their data. We conjecture that at least some of these usernames correspond to the same users based on the following two indications. First, we find direct connections between the usernames across different platforms. For example, user *3vilp4wn* at the "Hack This Site" forum is promoting a keylogger malware by referring to a GitHub repository [1] whose author has the same username. Second, these usernames are fairly uncommon, which increases the likelihood of belonging to the same person. For example, there is a GitHub user with the name *fahimmagsi*, and someone with the same username is boasting about their hacking successes in the "Ethical Hacker" forum. As we will see below, *fahimmagsi* seems to have a well-established online reputation.

b. "Googling" usernames reveals significant hacking activities. Given that these GitHub usernames are fairly unique, it was natural to look them up on the web at large. Even a simple Internet search with the usernames reveals significant hacking activities, including hacking websites or social networks, and offering hacking tutorials in YouTube.

We investigate the *top 40* most prolific malware authors using a web search with a *single* simple query: "hacked by <username>". We then examine only the first page of search results. Despite all these self-imposed restrictions, we identify three users with substantial hacking related activities across Internet. For example, we find a number of news articles for hacking a series of websites by GitHub users *fahimmagsi* and *CR4SH* [211] [44]. Moreover, we find user *n1nj4sec* sharing a multi-functional Remote Access Trojan (RAT) named "Pupy", developed by her, which received significant news coverage in security articles back in March of 2019 [143] [171]. We are confident that well-crafted and targeted searches can connect more malware authors with hacking activities and usernames in other online forums.

3.6 Discussion

We discuss the effectiveness and limitations of SourceFinder.

a. Why is malware publicly available in the first place? Our investigation in Section 3.5 provides strong indications that malware authors want to actively establish their hacking reputation. It seems that they want to boost their online credibility, which often translates to money. Recent works [48, 166, 181] study the underground markets of malware services and tools: it stands to reason that notorious hackers will attract more clients. At the same time, GitHub acts as a collaboration platform, which can help hackers improve their tools.

b. Do we identify every malware repository in GitHub? Our tool can not guarantee that it will identify every malware repository in GitHub. First, we can only identify repositories that "want to be found": (a) they must be public, and (b) they must be described with the appropriate text and keywords. Clearly, if the author wants to hide her repository, we won't be able to find it. However, we argue that this defeats the purpose of having a public archive: if secrecy was desired, the code would have been shared through private links and services. Second, our approach is constrained by GitHub querying limitations, which we discussed in Section 3.1, and the set of 137 keywords that we use. However, we are encouraged by the number and the reasonable diversity of the retrieved repositories we see in Table 3.6.

c. Are our datasets representative? This is the typical hard question for any measurement or data collection study. First of all, we want to clarify that our goal is to create a large database of malware source code. So, in that regard, we claim that we accomplished our mission. At the same time, we seem to have a fair number of malware samples in each category of interest, as we see in Table 3.6.

Studying the trends of malware is a distant second goal, which we present with the appropriate caveat. On the one hand, we are limited by GitHub's API operation, as we discussed earlier. On the other hand, we attempt to reduce the biases that are under our control. To ensure some diversity among our malware, we added as many words as we could in our 137 malware, which is likely to capture a wide range of malware types. We argue that the fairly wide breadth of malware types in Table 3.6 is a good indication. Note that our curated dataset MCur with 250 malware is reasonably representative in terms of coverage.

d. What is the overlap among the identified repositories? Note that our repository does not include forked repositories, since **GitHub does not return forked repositories as answers to a query**. Similarly, the breadth of the types of the malware as shown in Table 3.6 hints at a reasonable diversity. However, our tool cannot claim that the identified repositories are distinct nor is it attempting do so. GitHub does not restrict authors from copying (downloading), and uploading it as a new repository. In the future, we intend to study the similarity and evolution among these repositories.

e. Are the authors of repositories the original creator of the source code? This is an interesting and complex question that goes beyond the scope of this work. Identifying the original creator will require studying the source code of all related repositories, and analyzing the dynamics of the hacker authors, which we intend to do in the future.

f. Are all the malware authors malicious? Not necessarily. This is an interesting question, but it is not central to the main point of our work. On the one hand, we find some white hackers or researchers, such as Yuval Nativ [230], or Nicolas Verdier [150]. On the other hand, several authors seem to be malicious, as we saw in Section 3.5.

g. Are our malware repositories in "working order"? It is hard to know for sure, but we attempt to answer indirectly. First, we pick 30 malware source codes and all of them compiled and a subset of 15 of them actually run successfully in an emulated environment as we already mentioned. Second, these public repositories are a showcase for the skills of the author, who will be reluctant to have repositories of low quality. Third, public repositories, especially popular ones, are inevitably scrutinized by their followers.

h. Can we handle evasion efforts? Our goal is to create the largest malware source-code database possible and having collected 7504 malware repositories seems like a great start. In the future, malware authors could obfuscate their repositories by using misleading titles, and description, and even filenames. We argue that authors seem to want their repositories to be found, which is why they are public. We also have to be clear: it is easy for the authors to hide their repositories, and they could start by making them private or avoid GitHub altogether. However, both these moves will diminish the visibility and "street-cred" of the authors.

i. Will our approach generalize to other archives? We believe that SourceFinder can generalize to other archives, which provide public repositories, like GitLab and BitBucket. We find that these sites allow public repositories and let the users retrieve repositories. We have also seen equivalent data fields (title, description, etc). Therefore, we are confident that our approach can work with other archives.

3.7 Related Work

There are several works that attempt to determine if a piece of software is malware, usually focusing on a binary, using static or dynamic analysis [10, 46, 107, 186]. However, to the best of our knowledge, no previous study has focused on identifying malware source code in public software archives, such as GitHub, in a systematic manner as we do in this work. We highlight the related work in the following categories:

a. Studies that need malware source code. Several studies [120, 189, 238] use malware source code that are manually retrieved from GitHub repositories. Some studies [27] [26] compare the evolution and the code reuse of 150 malware source codes (with only some from GitHub) with that of benign software from a software engineering perspective and study the code reuse. Overall, various studies [59, 95] can benefit from malware source code to fine-tune their approach.

b. Mining and analyzing GitHub: Many studies have analyzed different aspects of GitHub, but not with the intention of retrieving malware repositories. First, there are efforts that study the user interactions and collaborations on GitHub and their relationship to other social media in [80,109,164]. Second, some efforts discuss the challenges in extracting and analyzing data from GitHub with respect to sampling biases [43,69]. Other works [99,100] study how users utilize the various features and functions of GitHub. Several studies [81, 170, 214] discuss the challenges of mining software archives, like *SourceForge* and GitHub, arguing that more information is required to make assertions about users and software projects. Finally, some efforts [187, 194, 235, 236] study GitHub repositories, but they focus on establishing a systematic method for identifying similarities, and use it to identify classes of repositories (e.g. Android versus web applications). Most of these studies use topic modeling, which is one of the approaches that we considered initially, but gave poor results in our context, but we will revisit in the future.

c. Databases of malware source code: At the time of writing this paper, there are few malware source code databases and are rarely updated such as project *theZoo* [228]. To the best of our knowledge, there does not exist an active archive of malware source code, where malware research community can get an enough number of source code to analyze.

d. Databases of malware binaries: There are well established malware binary collection initiatives, such as Virustotal [213] which provides analysis result for a malware binary. There are also community based projects such as VirusBay [216] that serve as malware binary sharing platform.

e. Converting binaries to source code: A complementary approach is to try to generate the source code from the binary, but this is a very hard task. Some works [52, 53] focus on reverse engineering of the malware binary to a high-level language representation, but not source code. Some other efforts [35, 77, 183] introduce binary decompilation into readable source code. However, malware authors use sophisticated obfuscation techniques [178] [34, 229] to make it difficult to reverse engineer a binary into source code.

f. Measuring and modeling hacking activity. Some other studies analyze the underground black market of hacking activities but their starting point is security forums [48, 166, 181], and as such they study the dynamics of that community but without retrieving any malware code.

3.8 Conclusion

Our work capitalizes on a great missed opportunity: there are thousands of malware source code repositories on GitHub. At the same time, there is a scarcity of malware source code, which is necessary for certain research studies.

Our work is arguably the first to develop a systematic approach to extract malware source-code repositories at scale from GitHub. Our work provides two main tangible outcomes: (a) we develop SourceFinder, which identifies malware repositories with 89% precision, and (b) we create, possibly, the largest non-commercial malware source code archive with 7504 repositories. Our large scale study provide some interesting trends for both the malware repositories and the dynamics of the malware authors.

We intend to open-source both SourceFinder and the database of malware source code to maximize the impact of our work. Our ambitious vision is to become the authoritative source of malware source code for the research community by providing tools, databases, and benchmarks.

Chapter 4

Repo2Vec: A Comprehensive Embedding Approach for Determining Repository Similarity

Establishing a way to measure similarity between software repositories is an essential building block for studying the plethora of repositories in online Open Source Software (OSS) platforms. These OSS platforms contain a massive number of repositories and engagement of millions of users [139]. There are significant collaborations and code reuses [60,198] on these platforms, which are openly supported and encouraged. Researchers are interested in studying the dynamics of such repositories, which include the ability to identify: (a) derivative repositories, (b) families of repositories, (c) the evolution of software projects, and (d) coding and technology trends. GitHub is arguably the largest such platform with more than 32 million repositories and 34 million users exhibiting significant collaborative

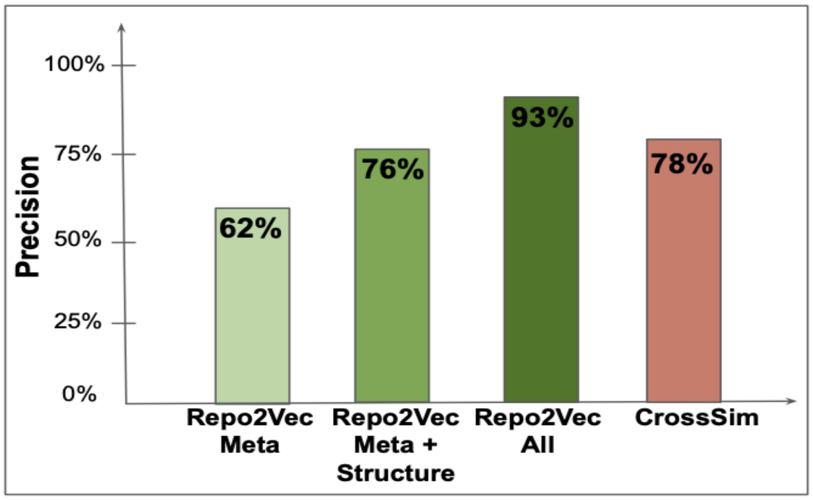


Figure 4.1: Our approach outperforms the state of the art approach CrossSim in terms of precision using CrossSim dataset. We also see the effect of different types of information that Repo2Vec considers: metadata, adding structure, and adding source code information.

interactions [138].

How can we quantify the level of similarity between two repositories? This is the problem that we address here. Focusing on GitHub, every repository consists of metadata, source code, and auxiliary files. Given a repository, how can we identify the most similar repositories among a large set? The input here is a large number of repositories and a set of queries. The desired output is: (a) the most similar repositories for a given query repository and (b) clusters of similar repositories. The key challenge here is to represent the repository data into a numeric feature vectors to enable ML approaches to compute the similarities and cluster among repositories. In addition, combining vectors from different types of information, as we will do here, is also a challenge.

There are relatively few efforts that focus on establishing similarity between repositories, and most of them use either metadata or source code level information, while none of them use the three types of information that we do here. First, LibRec [206], SimApp [37], Collaborative Tagging [205], and RepoPal [236] utilize only metadata to find similarity among repositories. Second, MUDABLUE [103] and CLAN [133] are two similarity computation approaches using only source code of repositories as plain text. Third, CrossSim [148, 149] proposes a graph representation to compute similarity between repositories using both metadata and source code. We discuss the related work in more detail in Section 4.6.

As our key contribution, we propose Repo2Vec, an embedding approach to represent software repositories with a multi-dimensional distributed continuous vector which can be used to measure the similarity between repositories. We briefly describe the key features of our approach. First, our method represents a repository as a distributed continuous vector in an embedding space. Second, we consider three types of information: (a) metadata, (b) source code, and (c) the repository directory structure. Our approach provides a flexible way to combine these three types using our default values which can be customized to match the niche needs of a savvy user. The significance of our approach is that it generates a relatively-low dimensional vector that can enable follow up repository analysis. Such follow up studies can leverage the plethora of ML techniques: we provide a proof of concept for two such applications here.

We deploy our approach and study the similarity on a malware dataset of 433 repositories and a benign dataset of 580 repositories. First, we demonstrate the effectiveness of our method by comparing it against state of the art works. Second, we show how our Repo2Vec can enable algorithms for: (a) distinguishing between malware and benign repositories, and (b) identifying a meaningful hierarchical clustering. The key results are briefly discussed below.

a. Repo2Vec outperforms prior works. For this comparison, we select the best approach to date, CrossSim, which has been shown to outperform previous approaches [103, 133, 236]. For consistency, we also follow their evaluation methodology and use their dataset with 580 benign repository. We show that our approach identifies similar repositories with 93% precision compared to 78% as shown in Figure 4.1. Further, our approach finds nearly **twice as many strongly similar** repositories and 30% fewer False Positives, as we see in Figure 4.6.

b. Metadata and structure provide significant performance. We assess the information contribution of three types of information. Interestingly, we can identify similarity fairly well without the use of source code as shown in Figure 4.1. Using only metadata and structure leads to a 76% precision, which is comparable to the previous best method, which uses source code.

c. Application: identifying malware repositories accurately. We show that our approach can enable a supervised classification approach. We focus on distinguishing malware from benign repositories, which is a practical problem [175]. Using our embedding, we can identify malware repositories with 98% precision and 97% recall, which outperforms the previous approaches.

d. Application: identifying a meaningful hierarchy. We show that our approach can form the basis for a meaningful (unsupervised) hierarchical clustering of repositories. We show that the emerging structure aligns with their purpose and lineage. In our evaluation, we focus at two levels of granularity: a coarse and a fine level with 3 and 26 clusters respectively. Using an LDA-based topic extraction method, we find that the clusters are cohesive: more than 80% of the repositories per cluster have the same focus. We discuss the clustering in Section 4.4.

Our work in perspective. Our approach can be seen as a first step towards the use of embedding approaches in repository analysis. In fact, it can be seen as a general framework where the selection of individual features can be driven by the intention of the application. For example, one can focus on different primary features depending on whether we want to identify: (a) plagiarism or function level similarity, (b) programming styles, or (c) software intention.

4.1 Background

We provide some background on GitHub and describe embedding approaches, which we extend and use later.

A. GitHub and its features. GitHub is a massive software archive, which enables users to store and share code creating a global social network of interaction. Users can collaborate on a repository by raising issues or forking projects, where they copy and evolve projects. Users can follow projects, and "up-vote" projects using "stars". We describe the key elements of a GitHub repository here. A repository contains three types of information (a) metadata, (b) project directory, and (c) source code files, which we explain below.

a. Metadata: A repository in GitHub has a large number of metadata fields. Most notable are: (a) title, (b) descriptions, (c) topics, and (d) readme file. All these fields are optional and they are provided by the author. Commit and issues are other sources of textual metadata which include messages about the specific functionality of the repository. At the same time, there are metrics that capture the popularity of a repository including: (a) stars, (b) forks, and (c) watches. As the text fields are provided by the repository author, they can be unstructured, noisy, or missing altogether.

b. Source code: It is the core element of a software repository. A repository contains software projects written in various programming languages such as C/C++, Java, Python, and so on. These source codes are the logical centre of a software stored in a repository.

c. Project directory structure: A well-crafted software repository follows a best-practices directory structure containing dataset, source code, and other auxiliary files. We hypothesize that the structure could be useful in establishing similarity between repositories.

B. Embedding approaches. An embedding (a.k.a. distributed representation) is an unsupervised approach for mapping entities, such as words or images, into a relatively low-dimensional space by using a deep neural network on a large training corpus [114,137]. Although the method is unsupervised, it relies on ideally a large dataset, which is used to "train" the neural network. The neural network develops a model of the dataset, which we can think of as probabilities and correlations of its entities. Embedding approaches have revolutionized research in several fields, such as Natural Language Processing (NLP) [114, 137, 161, 162], computer vision [110], graph mining [70, 145], and software analysis [6].

The power of an embedding is twofold: (a) it can simplify the representation of a complex entity with diverse features, including categorical, and (b) it provides a way to quantify entity similarity as a function of the distance of their corresponding vectors. An efficient embedding has the following properties: (a) it gives a fixed and low dimensional vector, and (b) it ensures that semantically similar objects are “close by” in the embedding space.

a. Word embedding: word2vec. In the seminal word2vec work [137], we map words to vectors in a way that similar words, such as “father” and “parent”, map to nearby vectors. This similarity is established by “feeding” a large corpus of documents to the deep neural network. In other words, the model captures word correlations by calculating the probability with which a word can appear within a given neighborhood of words.

b. Document embedding: doc2vec. The doc2vec [114] is an unsupervised embedding model for a variable length paragraph or document. The model takes a document as input and maps it to an M-dimensional embedding vectors while doing a proxy task, predicting target word or sampled words in the document.

In more detail, the document embedding model is based on the word embedding [137] model. The main difference between them is the introduction of the document id vector. Like word2vec, there are two types of doc2vec available: (a) Distributed Memory Model of Paragraph Vectors (PV-DM) and (b) Distributed Bag of Words version of Paragraph Vector (PV-DBOW). PV-DM is similar to the Continuous Bag of Words (CBOW) model in word2vec. The PV-DBOW model is similar to the skip-gram model of word2vec.

The document vector is calculated at the same time as the word vectors of the document. Note that, PV-DM performs better for large, and well-structured documents. On the other hand, PV-DBOW is considered a better choice for small and defective documents, as it is computationally fast.

c. Code embedding: code2vec. Embedding approaches have also been proposed for detecting code similarity. A recent approach is code2vec which maps a method (or more generally a code snippet) of arbitrary length to an M-dimensional vector [6, 42]. The code2vec approach uses program structure explicitly to predicting program properties and uses an attention based neural network that learns a continuous distributed vector representation for the code snippet. As a result, one can compare and group code snippets. The process is fairly involved as it attempts to capture the logical structure and flow of the program and the sequence of commands. For example, the code is decomposed into a set of paths based on its abstract syntax tree. The neural network learns simultaneously: the representation of each path and how to aggregate a set of them. Due to space limitations, we refer the interested reader to the original work [6].

d. Node embedding: node2vec. The node2vec [70] is a graph embedding approach for mapping a node in a network to an M-dimensional embedding vector. The model maximizes the likelihood of preserving network neighborhoods of nodes using Stochastic Gradient Descent (SGD).

In more detail, the model computes the embedding based on nodes neighborhoods. First, the network structure is converted to a set of paths (node sequences) using a biased random walk sampling strategy which combines Depth-First Sampling (DFS) and Breadth-

First Sampling (BFS) for every nodes. The sampling strategy efficiently explores diverse neighborhoods of a given node. These sets of paths can be analogized to the sentences in a document. Then the model is trained on these node sequences with the skip-gram models presented in word2vec [137] to get the vector representations for each node. For more details about the model, we refer to the original paper [70].

4.2 Proposed Method

The main idea behind Repo2Vec is to combine the metadata, source code, and directory structure of a repository and provide an embedding representation for the whole repository. In fact, we create an embedding for each type of data, which we refer to as: (i) meta2vec for metadata, (ii) source2vec for the source code, and (iii) struct2vec for the directory structure. Our approach follows these four steps. In the first three steps, we create an embedding vector for each of the three types of data, and in the fourth step, we combine these into a repository embedding. The Repo2Vec pipeline is shown in Figure 4.2. We explain each step of our approach in more detail below.

Step 1. Metadata embedding: meta2vec. We define meta2vec as mapping all the metadata in a repository to an R_M-dimensional embedding vector, M_{R_M}. In meta2vec, we follow three steps. First, we select the fields of metadata that we want to "summarize" in embedding. Second, we preprocess the metadata text to remove noise. Finally, we adapt the doc2vec approach to compute the embedding vector. The overview of meta2vec is shown in Figure 4.3.

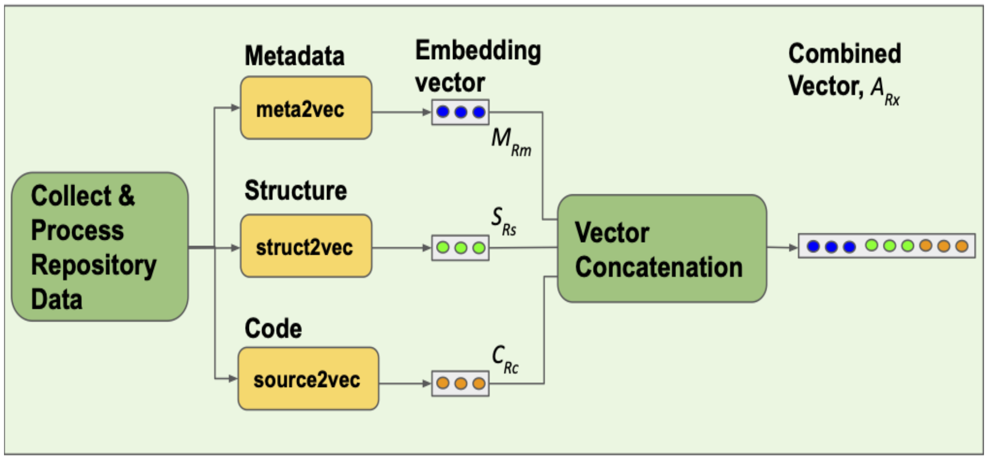


Figure 4.2: Overview of the Repo2Vec embedding: (a) we create an embedding representation for metadata, structure, and source-code, and (b) we combine them into an embedding that captures all three types of information. Each embedding hides significant subtleties and challenges.

a. Field selection: We consider all the fields of metadata that contain descriptive information regarding the content of a repository such as title, description, topics (or tags), and readme file. Recall that all this information is provided by the author. There are many ways to extract and combine textual information from each field. Here, we opt to treat each metadata field as a paragraph and concatenate them to generate a document, which we process as below. Note that we do not consider metrics that relates to the popularity of a repository, since our intuition and initial results suggest that it is less helpful in determining similarity.

b. Text preprocessing: Like any Natural Language Processing (NLP) method, we start with necessary preprocessing of the text to improve the effectiveness of our approach. As metadata in a repository text fields are often noisy, we follow the NLP best practices step which include removal of: (i) special characters e.g. '?', and '!', (ii) irrelevant

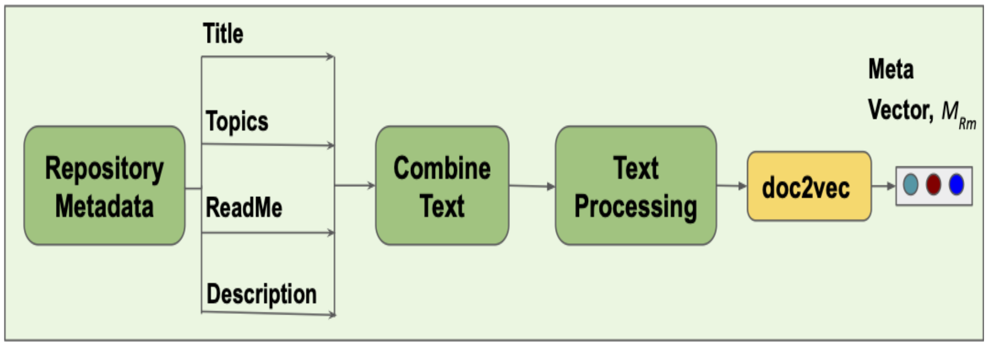


Figure 4.3: The overview of the meta2vec embedding: (a) we collect the text from metadata fields, (b) we combine them into a single document, (c) we preprocess the text in the document, and (d) we map the document to a vector using an approach inspired by doc2vec.

words and numbers e.g. "URL", "Email", "123", (ii) stopping words.

c. Repository meta vector generation: We map the metadata in a repository to an R_M-dimensional distributed vector, M_{R_M} in this step. Following the basic principles of doc2vec [114] approach, we adapt it to our needs and constraints here. Specifically, as metadata in a repository often consists of unstructured text and is small in size, we employ PV-DBOW, discussed in Section 4.1, because it performs better for small text dataset.

Step 2. Directory structure embedding: struct2vec. We define struct2vec as mapping of repository directory structure to an R_S-dimensional embedding vector, S_{R_S}. We compute struct2vec following three steps. First, we represent the directory structure into a tree representation. Second, we generate node vectors employing node2vec. Third, we synthesize node vectors into a single structure vector. The overview of struct2vec is shown in Figure 4.4.

a. Directory tree representation: A software repository in GitHub consists of a standard directory structure with necessary data files and source code files. We consider the directory structure and transform it into Tree representation to enable node2vec on it. Note that, in order to nullify the effect of directory or file names in the mapping, the representation does not include directory or file names in the tree.

b. Node vector generation: We map all nodes in the tree into an R_S-dimensional node embedding vector, N_{R_S}, in this step. Following the properties of node2vec, first, we convert the trees into a set of paths using a biased random walk sampling strategy to include a diverse set of neighborhood nodes for a node. Then, we apply skip-gram models on these paths to get vectors for all nodes.

c. Repository directory structure vector generation: We compute repository directory structure embedding vector, S_{R_S}, by synthesizing the node vectors, N_{R_S}, in the tree. We follow column-wise aggregation method to synthesize these into a single vector. In order to do that, we employ six aggregation functions: mean, mode, max, min, sum, and standard deviation to compute a value for a column in the resultant vector.

Step 3. Source code embedding: source2vec. We define source2vec as an embedding approach to represent the source code in a repository to an R_C-dimensional embedding vector. In source2vec, we employ the Java method embedding techniques and a trained model with 15.3M methods discussed in Section 4.1. We follow three steps in source2vec. First, we compute the R_C-dimensional method code vectors for each method in the source file available in a repository. Second, we aggregate these method vectors in a single R_C-dimensional file code vector. Finally, we compute the final R_C-dimensional

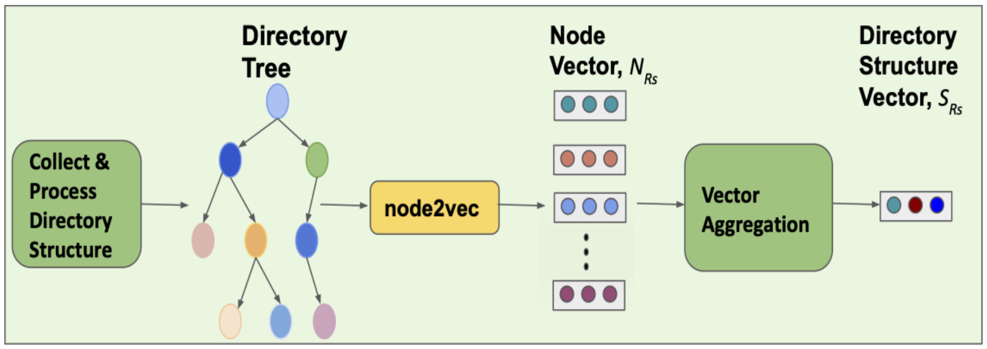

Collect & Process Directory Structure
Directory Tree
node2vec
Node Vector, N_{R_S}
Vector Aggregation
Directory Structure Vector, S_{R_S}

Figure 4.4: The overview of our struct2vec embedding: (a) we extract directory tree structure of the repository, (b) we map each node into a vector following a node2vec approach, (c) we combine the node embedding to create the structure embedding for the repository.

repository code vector for all the source files by another level of vector aggregation. The pipeline of our approach is shown in Figure 4.5 and discussed below in details.

a. Method code vector generation: A software repository may have multiple source code files and other files. First, each source file is decomposed into its methods. Next, methods are preprocessed into AST paths, and context vectors which are the input to the code2vec model. The model maps each method into an R_C-dimensional embedding code vector, MC_{R_C}. These method vectors are then passed to the next stage of pipeline to be aggregated into a single vector.

b. File code vector generation: After generating the method code vectors, MC_{R_C}, in a file, the task is now to aggregate them into an R_C-dimensional file code vector, FC_{R_C}. We follow a number of column-wise aggregation functions. The aggregation functions that we investigate are mean, mode, max, min, sum, and standard deviation. Following the procedure, the pipeline creates a single file code vector, FC_{R_C}, and passes it

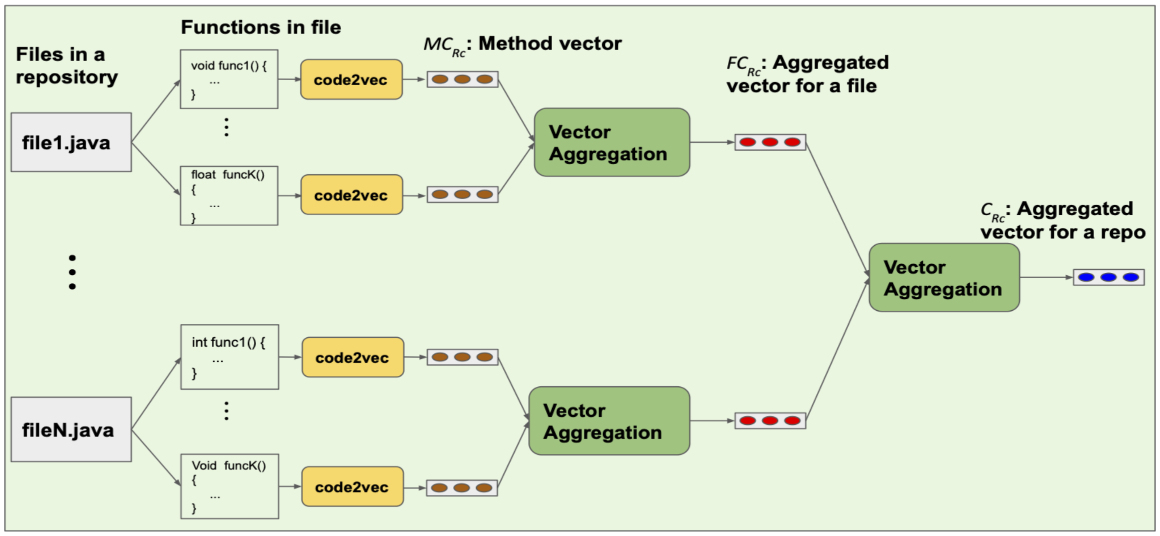


Figure 4.5: The overview of our source2vec embedding: (a) we extract functions (methods) from each source file, (b) we embed each function, (c) we combine each function embedding to create an embedding for each file, and (c) we aggregate each file embedding to create the source-code embedding for the repository.

to the next stage to create a single repository vector.

c. Repository code vector generation: At this stage of the pipeline, source2vec aggregates all the R_C-dimensional file code vectors, FC_{R_C} for all source code files available in the repository to a single R_C-dimensional repository code vector, C_{R_C}. The pipeline follows same procedure like previous step, column-wise aggregation function to get the repository code vector.

Step 4. Repo2Vec: Repository embedding.

We propose Repo2Vec to present a GitHub repository in an embedding vectors using features from three types of information sources: metadata, source code, and project directory structure following the pipeline shown in Figure 4.2.

In this step, we combine metadata vector M_{R_M}, directory structure vector S_{R_S}, and source code vector C_{R_C} into repository vector A_{R_x}.

Combining the vectors of each information type is a challenge as many methods exist following two types of approaches: (a) merging the numerical values into a single vector, using the sum, average or median, etc, and (b) concatenating vectors to create a "longer" vector. In both approaches one can consider weighting and normalizing to ensure "fairness". Here, we opt to use the concatenation approach as follows:

$$A_{R_x} = w_M * M_{R_M} + w_S * S_{R_S} + w_C * C_{R_C} \tag{4.1}$$

where w_M, w_S and w_C are the weights for the meta vector M_{R_M}, structure vector S_{R_S}, and source code vector C_{R_C} respectively, and these weights are in the range of $[0, 1]$.

4.3 Experiments and Evaluation

We evaluate the effectiveness of Repo2Vec using real data and answer two questions.

Q.1: What is the effect of each information type? We want to quantify the effect and contribution of the three information types in determining similarity.

Q.2: How does Repo2Vec compare to prior art? We compare our method with CrossSim [148, 149], which is arguably the state of the art approach and was shown to outperform previous approaches [103, 133, 236].

4.3.1 Experimental Setup

We present the datasets and our evaluation approach.

1. Datasets. We consider two datasets in our evaluation: (a) a dataset of benign repositories, D_ben, which was used in prior work [148, 149], and (b) a dataset of malware repositories, D_mal, collected by a prior repository analysis study [175].

a. Benign repositories D_ben: This dataset consists of 580 Java repositories from GitHub and was used in an earlier study introducing CrossSim [148]. We select this dataset in order to make a fair and reproducible comparison with CrossSim. The dataset spans various software categories such as: PDF processors, JSON parsers, Object relational mapping projects, Spring MVC related tools, SPARQL and RDF, Selenium test, Elastic search, Spring MVC, Hadoop, and Music player.

b. Malware repositories D_mal: This dataset consists of 433 Java malware repositories. The dataset is provided by the SourceFinder project [175], whose goal is to identify and provide malware source code repositories. Here, we choose only the Java language repositories, which are the focus of the CrossSim approach. The repositories have a fairly wide coverage across malware families including: Botnets, Keyloggers, Viruses, Ransomware, DDoS, Spyware, Exploits, Spam, Malicious code injections, Backdoors, and Trojans.

2. Query-based evaluation. For consistency and fairness, we follow the evaluation methodology and similarity metrics of prior work [148]. We conduct our evaluation by using similarity queries as follows: a given repository, we want to identify its five most similar repositories.

a. The query-set Q_ben: For the sake of compatibility with CrossSim, *Q_ben* consists of the same query set of 50 repositories as CrossSim. The query set spans various domains e.g. SPARQL and RDF, Selenium test, Elastic search, Spring MVC, Hadoop, and Music player.

b. The query-set Q_mal: For the D_mal dataset, we create a query-set by selecting 50 repositories uniformly at random. The query set includes various malware families such as Keylogger, Botnet, DDoS, Ransomware, Virus, Backdoor, Trojan, etc.

3. Ground truth generation. We establish the groundtruth for each dataset by manual evaluation and follow the scoring framework, which was used in prior work [148]. Namely, we use four categories of scores to label the level of similarity:

- Category 4: Strongly Similar (**SS**) repositories.
- Category 3: Weakly Similar (**WS**) repositories.
- Category 2: Weakly Dissimilar (**WD**) repositories.
- Category 1: Strongly Dissimilar (**SD**) repositories.

For consistency, we follow the convention of the previous study [148]: a repository in category 3 or 4 is considered (sufficiently) similar or a True Positive. Conversely, a repository in category of 1 or 2 is considered dissimilar or a False Positive.

For the evaluation, we opted to use experts, who are more reliable compared to a Mechanical Turk platform for highly technical questions [62] . Specifically, we recruited three computer science researchers with at least 3 years of Java programming experience. The evaluators are given the target repository and the response of 5 repositories per query. Note that the five repositories in each response are in random order to avoid introducing biases. The evaluators assign a score among the four categories of scores to each repository in the responses. The evaluators were provided with context and information in order to calibrate their criteria. The first and second evaluators independently assign a score to each repository in the response. Later, the third evaluator acts as judge by rechecking and finalizing their scores if their scores are not same for a query.

4. Evaluation metrics. For consistency, we adopt the metrics used in related work [148], which we describe below.

a. Success rate: We say an answer to a query is successful, if one or more of the returned repositories is similar to the above definition of similarity. The success rate is the percentage of successful queries.

b. Precision: Precision is the percentage of the returned repositories which are similar to their query repository. We compute the precision following the equation,

$$precision = \frac{SS + WS}{SS + WS + WD + SD} \tag{4.2}$$

c. True and False Positives: Following the standard definitions, True Positives for a query-set is the total number of similar repositories returned, while False Positives is the number of non-similar repositories in the answers.

d. Ranking order correlation (ROC): We quantify the quality of the ranked answer to the query using again a metric introduced in prior work. The intuition is to "reward" an algorithm that returns highly similar repositories ranked higher. To quantify this, we calculate the widely-used Spearman's rank correlation coefficient r [197], which is defined as:

$$r = 1 - \frac{6 \sum (d_i)^2}{n(n^2 - 1)} \tag{4.3}$$

where r is the coefficient, d_i is the difference between the two ranks of each repository, and n is the number of ranked repositories. The coefficient is in the range of $[-1, 1]$, with 1 implying perfect agreement, and -1 disagreement between the two rankings.

Comment: Given the way we formulate the query, the use of Recall is less relevant here: we ask the algorithms to report only the top five most similar repositories. Formulating a query we expect the methods to return all similar repositories is challenging for two reasons. First, we would need an established ground-truth, since manual validation would be labor-intensive. Second, there is no absolute way to define what constitutes "sufficiently similar" repositories, while relative similarity is easier to define.

4.3.2 Deploying Repo2Vec

We implement our method, which we described in Section 4.2 using Python3.6 packages: TensorFlow2.0.0, gensim PV-DBOW doc2vec. We discuss some implementation details and parameter choices.

Selecting the embedding dimensions. We select 128 as the embedding vector dimension for R_M, R_S, and R_C, since well-established embedding techniques [6,70,114,137] recommend this number for striking a balance between computational cost and effectiveness. We use the same number of dimensions for the vector of each type of information for fairness. Concatenating these three vectors creates a single Repo2Vec vector with R_x=384 dimensions. The above choices give good results as we will see later. In the future, we will explore the effect of different vector dimensions.

Exploring the solution space via weight selection. The weights in equation 4.1 give us the ability to control the "contribution" of each information type. Here, we focus on the following weight combinations, which give rise to three derivative algorithms: (a) **Repo2Vec_M** using only metadata with weights $w_M = 1, w_S = 0, w_C = 0$; (b) **Repo2Vec_MS** using metadata and structure with weights $w_M = 1, w_S = 1, w_C = 0$; and, (c) **Repo2Vec_All** using all three types of information with weights $w_M = 1, w_S = 1, w_C = 1$.

In other words, we explore the effect of weights but in a coarse way. In the future, we intend to explore non-integer weight combinations. Overall, our results suggest that equal weights seem to work quite well, but a savvy user can customize them to achieve optimal performance for niche problems.

Calculating the similarity. There are many different ways to calculate the similarity in an embedding space as the inverse of their distance in that space. Here, we use the widely used cosine similarity, which is often recommended for high dimensional spaces [192], and yields great results here as well.

Selecting the right aggregation function to aggregate multiple vectors into a single vector. As we see in Section 4.2, we introduce six column-wise aggregation functions to aggregate vectors into a single vector. We find that mean aggregation function performs better than others. In more detail, we evaluate the performance of all aggregation functions: average, max, min, mode, sum, and standard deviation. We find that embedding with mean aggregation shows highest 93% precision for D_ben dataset and 95% precision for D_mal dataset. Max aggregation function shows the second best result 88% and 91% precision for benign and malware dataset respectively. Other aggregation functions show relatively lower precision for both dataset. In the remaining of the work, we use the mean

Method	D_ben Dataset		D_mal Dataset	
	Success Rate	Precision	Success Rate	Precision
Repo2Vec_M	100%	62%	100%	67%
Repo2Vec_MS	100%	76%	100%	82%
Repo2Vec_All	**100%**	**93%**	**100%**	**95%**

Table 4.1: Performance comparison of our three variants of Repo2Vec. Using all three information types (metadata, structure, and source code) provides significantly better results.

aggregation function.

4.3.3 Evaluation

We evaluate Repo2Vec in two ways. First, we assess the effect of each type of information on the performance. Second, we compare our method against CrossSim [148], which is the state of the art approach.

a. The effect of the information types: We evaluate the effect of information types by comparing the performance of our three variants: Repo2Vec_M, Repo2Vec_MS, and Repo2Vec_All, which we defined earlier. We report the result in Table 4.1 for our three Repo2Vec variations and both datasets. This evaluation leads to two main observations:

Observation 1: Using all three data types provides significantly better performance. In the table, we see that Repo2Vec_All achieves 93% and 95% precision compared to 76% and 82% when only metada and structure information are used.

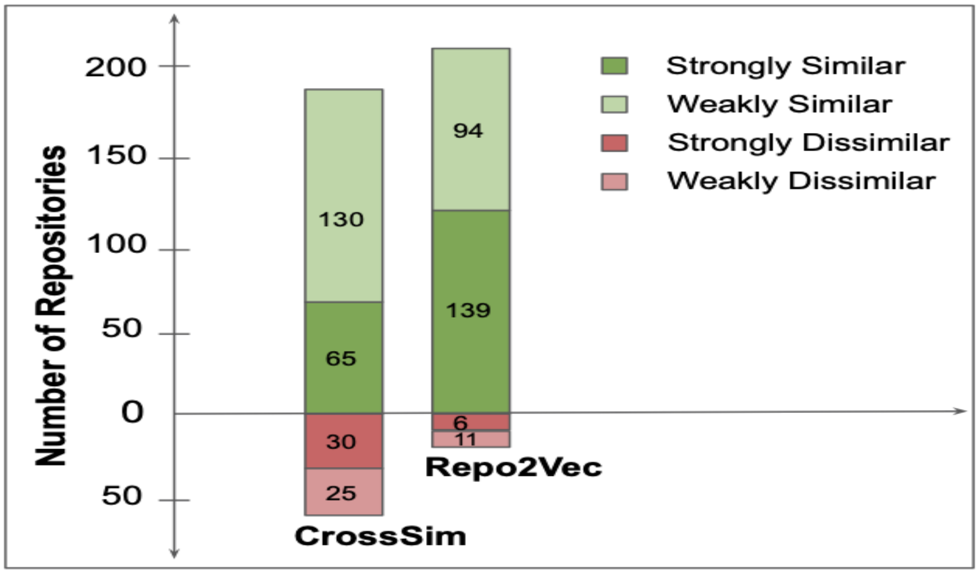


Figure 4.6: Repo2Vec outperforms CrossSim significantly: it finds nearly twice as many Strongly Similar repositories and 30% fewer False Positives.

Observation 2: Metadata and structure provide fairly good results. Although Repo2Vec_All performs best, Repo2Vec_MS performs quite well especially if we compare it with CrossSim on the same benign dataset and query-set shown in Table 4.2. Note that the computational effort for using metadata and structure is significantly less compared to analyzing the code.

b. Comparing Repo2Vec to the state of the art. We compare the best configuration, Repo2Vec_All, with CrossSim with respect to success rate, precision, confidence, and ROC for the benign dataset D_ben. We find that Repo2Vec outperforms CrossSim in terms of precision and ROC and has the same success rate as CrossSim.

Method	**Success Rate**	**Precision**	**Spearman's Coefficient (r)**
CrossSim	100%	78%	0.23
Repo2Vec_All	100%	93%	0.59

Table 4.2: Repo2Vec performs better in comparison of similarity approaches between Repo2Vec and CrossSim for the D_ben dataset.

Observation 3: Repo2Vec: higher precision and better ranking. The results are presented in Table 4.2. Although **CrossSim** does well in terms of success rate, its precision of 78% is significantly lower compared to the precision of 93% of Repo2Vec_All. Also, the ranking of similar repositories identified by Repo2Vec_All is better than **CrossSim**. We find that ROC = 0.59 for Repo2Vec_All, and ROC = 0.23 for **CrossSim**, which further suggests that Repo2Vec_All is better at computing similarity among repositories.

Observation 4: Repo2Vec provides better quality results. Given that we have four categories of similarity, we assess the quality of the results as follows. We plot the returned repositories from each method per category in Figure 4.6. Considering category 4 (strong similarity) only, Repo2Vec_All identifies nearly 100% more such repositories! Similarly, CrossSim reports 5 times more repositories in the strong dissimilarity category.

In conclusion, our comparison suggests that Repo2Vec outperforms CrossSim. The evaluation is summarized in Table 4.2 and Figure 4.6. In addition, CrossSim was shown to perform better than other related work RepoPal, CLAN, and MUDABLUE [148].

4.4 Case Studies

In this section, we want to showcase how Repo2Vec can facilitate repository mining studies for specific applications considering both unsupervised and supervised techniques. We consider two likely case studies: a) classifying repositories as benign or malicious, and b) clustering a set of repositories.

4.4.1 Identifying malware repositories

We showcase the usefulness of our Repo2Vec in a supervised classification problem, which is of interest to practitioners [175,194,235]. The question is to identify whether a repository contains malware or benign code. We assess the effectiveness of our approach and we also compare it with the state of the art method [175].

We create a dataset of 580 benign repositories from D_ben and 433 malware repositories from D_mal collected and discussed in Section 4.3.

Method	**Accuracy**	**Precision**	**Recall**	**F1 Score**
SourceFinder	90%	89%	99%	94%
Repo2Vec	97%	98%	96%	97%

Table 4.3: Repo2Vec outperforms SourceFinder in malware repository classification

Using our Repo2Vec, we determine the embedding vector for each repository. For the classification, one can use a plethora of ML approaches. Here, we use the Naive Bayes, which is widely used for NLP classification problems [227], and, more importantly, it is also

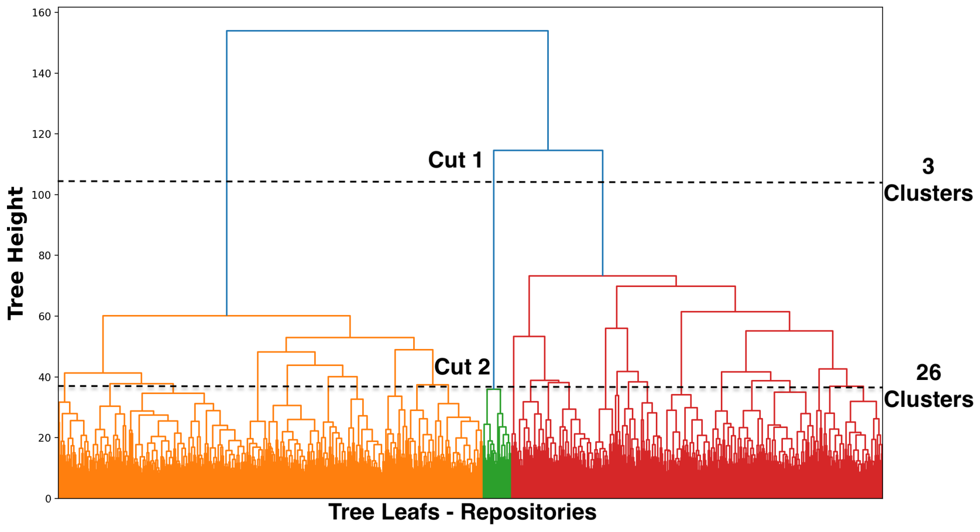


Figure 4.7: Hierarchical clustering of malware repositories. Horizontal line 1 cuts into 3 distinct cluster of repositories and line 2 cuts into 26 distinct cluster of repositories

used by the most recent SourceFinder study [175]. With this selection, we want to focus more on the effect of the features when comparing to the SourceFinder classification. We implement the SourceFinder classifier, and apply it on our dataset.

We assess the classification performance using 10-fold cross validation. The results are shown in Table 4.3. Our model classifies the malware and benign repositories with 98% precision and 96% recall which clearly outperforms the previous malware repository classification study by SourceFinder [175].

4.4.2 Hierarchical clustering

Here we showcase whether our approach can lead to a meaningful clustering of repositories creating the basis for an unsupervised solution. We consider the union of our two datasets, D_mal and D_ben dataset with a total of 1013 repositories.

First, we apply Repo2Vec on all the repositories and get the embedding vectors. Second, we apply the widely-used agglomerative hierarchical clustering (AGNES) [142] on the vectors of the repositories. Clearly, there are many different clustering techniques, but note that our goal is to showcase the capability and not to propose a clustering method. We show the resulting hierarchical clustering in Figure 4.7.

How meaningful is this clustering? Assessing the effectiveness of a hierarchical clustering is challenging and it can depend on specific focus of a study. A related question is at what levels of granularity we should focus. We provide indirect proof that our clustering provides meaningful results.

Considering two levels of granularity. We analyze our hierarchical clustering at two different levels of granularity, which are represented by two horizontal lines in Figure 4.7. The first line (Cut 1) corresponds to a **coarse level** of granularity and yields three large clusters. The second line (Cut 2) corresponds to **fine level** of granularity and yields 26 smaller clusters.

We elaborate on how we select the two cuts in the dendogram in Figure 4.7. First, we select Cut 1 to see if the clustering distinguishes the malware from the benign repositories. Second, we select a Cut 2 in a way that optimizes the number of clusters. A commonly-used approach is the elbow method [106]. The elbow or knee of a curve is a cutoff point in the number of clusters versus sum of squared error (SSE) graph, where increasing the number of cluster shows diminishing returns. Figure 4.8 shows that the elbow lies at around K=26 clusters, which is how we select Cut 2.

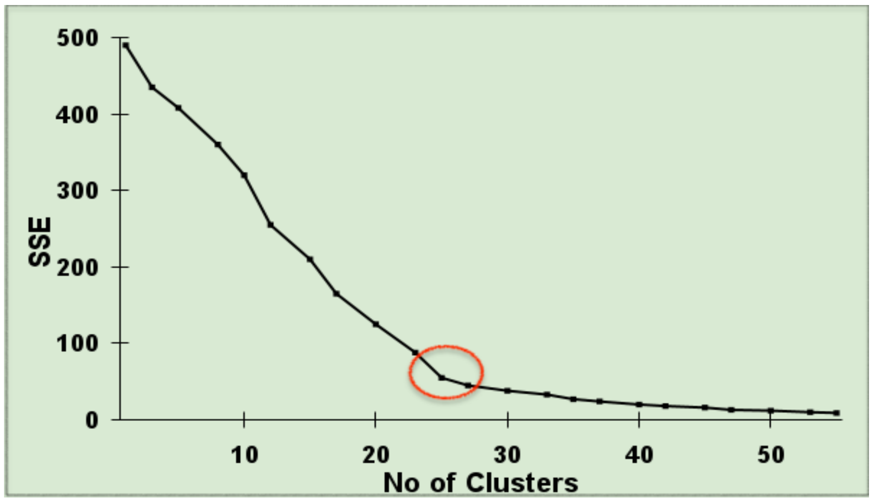


Figure 4.8: Determining optimal number of clusters. Diminishing returns of sum of squared error (SSE) is shown at red circle.

Our goal is to profile the identified clusters at both levels of granularity. The results are shown in Table 4.4 and 4.5.

1. Fine level cluster profiling: We want to evaluate the nature and the cohesiveness of the 26 clusters at this level. We extract the profile of each cluster in terms of its focus and we present the results in Table 4.4. Our profiling consists of two steps: (a) we identify the dominant keywords of the cluster and (b) we assess how aligned its repository is to the profile cluster. In more detail, we identify the cluster topics using Latent Dirichlet Analysis (LDA) topic modeling [22] on the metadata of each repository. Note that we use a randomly selected subset of half of the repositories in the cluster. Second, we want to identify the most dominant topic among all the candidate topics. The most dominant topic is the one that appears in the most repositories of the cluster. We report that topic

Cluster No.	Number of Repos	Dominant Repo Family	Cluster No.	Number of Repos	Dominant Repo Family
1	25	DDoS	14	10	Virus
2	27	Android Keylogger	15	58	Trojan and Spyware
3	42	Backdoor	16	33	REST API
4	32	Worms	17	48	Hadoop
5	44	Android Botnet	18	36	JSON Parser
6	55	Android Malware	19	45	Music Player
7	31	Rootkit	20	71	SPARQL
8	24	Java Keylogger	21	146	Elastic Search
9	32	Ransomware	22	54	Object Relational Mapping
10	24	Whitehat Hacking	23	27	PDF Processor
11	15	Malicious Code Injection	24	25	Graph-Aided Searc
12	8	Android Trojan	25	31	Selenium
13	6	Android Backdoor	26	56	Spring MVC

Table 4.4: Fine-level clustering: the profile of the 26 repository clusters using a topic extraction method. The color of the cluster is similar to that of Figure 4.7.

Cluster No.	Number of Repos	Cluster Type	Cluster Description
1	433	Malware	The D_mal malware repositories
2	33	Benign	Cluster 16 from the fine granularity with REST API repositories
3	547	Benign	The D_ben repositories.

Table 4.5: Coarse-level clustering: the profile of the three clusters. The color of the cluster is similar to that of Figure 4.7.

in table 4.4. The cohesiveness of the cluster is substantial: at least 80% of the inspected repositories are clearly members of the family of the cluster. Finally, as an extra optional step, we manually investigate the repositories to verify the accuracy of the profile.

This process gives us both cohesive and "focused" clusters. Most of the clusters contain repositories from narrowly-defined malware or benign software families, such as Android Botnet, Keyloggers, Trojan, DDoS, Backdoor, Hadoop, Json parser, Elastic Search, and Spring MVC.

We provide an indication of an insight that can be extracted here. Interestingly, the largest malware cluster (cluster 15) with 58 repositories contains repositories from Trojan and spyware malware families. A Trojan malware program is similar to spyware except that it is packaged as another program. This observation can give rise to the following hypothesis: could Trojan and Spyware have more in common than we thought?

2. Coarse level cluster profiling: The overarching observation is that the three clusters of this level correspond correctly to different software domains as shown in Table 4.5. We find that following clusters: (a) the D_mal, malware repositories, (b) the D_ben, benign repositories, and (c) REST API related benign repositories, which correspond to cluster 16 in the fine granularity clustering. The fact that the unsupervised clustering separated malware and benign repositories suggests that malware and benign software are different. The only exception seems to be the REST API cluster 16, which would have been bundled with the malware repositories if we have created a two cluster decomposition. We argue that the REST API repositories seem to resemble ddos and botnet malware (opening and listening to ports etc).

4.5 Discussion

In this section, we discuss the scope, extensions, and limitation of our study.

a. What are the limitations of Repo2Vec? As Repo2Vec is a comprehensive approach with data from three different sources, it performs even if every data source is not present. However, we believe unstructured software repositories with evasive metadata and obfuscated source code might fool Repo2Vec. In this case, previous works might perform better as these mostly depend on the graph connection of repositories.

b. Will our approach generalize to other programming language repositories? Our approach is generalizable and extendable for all programming languages, though accuracy levels may vary. First, code2vec [6] can be extended to other programming languages, and the researchers seem to have plans to expand to other languages. Second, two information types, metadata, and structure, are fairly programming-language independent. Furthermore, from Table 4.1, we can see that even using only these two information types, we can achieve reasonably good performance.

c. What will happen if the quality of metadata is low or misleading? If metadata becomes unreliable, we could decrease its weight in our algorithm. At the same time, we find that developers have an inherent motivation to provide quality metadata. First, these repositories are part of the developers professional persona, and part of one's professional portfolio or resume. Second, these repositories are public, therefore there is an intention to make them both easy to find and easy to use. The bragging rights of having a popular repository is a strong motivation to provide informative metadata. Hence, the number of these type of repositories tend to be very low. We only have 1 in 580 (0.17%)

repositories in D_ben, and 3 in 433 (0.69%) repositories in D_mal with an empty metadata. Also, as Repo2Vec is a comprehensive approach with data from three information sources, even if metadata is unavailable, it will perform sufficiently.

d. Why is GitHub search not sufficient to identify similar repositories? GitHub only allows the retrieval of repositories based on the keywords. Though it is very useful, GitHub's query capability is not answering the problem that we address here. First, it does not support query by example: "find the most similar repositories to this repository". Second, it does not provide the ability to measure similarity between a pair of repositories or rank a group of repositories based on similarity to a given repository. Third, the service does not seem to use source-code which as we saw, provides significant improvement.

e. Are our datasets representative? This is the typical hard question to answer for any measurement study. We attempt to answer the question by making two statements. First, we evaluate our approach with the same dataset of 580 repositories (D_ben) used by well-known prior studies [148, 149]. This dataset attempts to include repositories from ten different families as listed in Table 4.4. Second, our D_mal dataset includes 13 types of malware families listed in the same table. In the future, we intend to collect more repositories in our dataset and include more programming languages. The key bottleneck is the creation of groundtruth.

f. Should we consider the popularity metrics? So far, we did not consider the popularity metrics of the repositories, such as the number of stars, watches, and forks. While we intend to examine what information we can extract from such metrics, we argue that they will mostly help in finding the representative or influential repositories. Our preliminary analysis suggests that popularity does not provide information w.r.t. the type of the repository. As a proof of concept, we can consider an initial and a forked repository: they are most likely nearly identical, but their popularity metrics can vary significantly.

4.6 Related Work

Studying the similarity among software repositories has gained significant attention in the last few years. Most studies differ from our approach in that: (a) they do not incorporate all types of data present in a repository, (b) they do not present a feature vector keeping the semantic meaning of the metadata, source code, and structure of a repository, and (c) their approaches are not suitable for other ML classification tasks such as repository family classification, malware and benign repository classification, etc. We discuss the related work briefly below.

a. Software similarity computation: The prior studies in software similarity computation can be classified mainly into three groups based on the data they use: (a) high level meta data [37, 205, 206, 236], (b) low level source code [103, 133], and (c) the combination of both high and low level data [148, 149].

In an earlier study [205], authors utilize repository tags to compute the similarity among repositories written in different languages. Capturing the weights of tags present in a repository, they create the feature vector and apply cosine similarity to compute the similarity. Later, [206] proposes a library recommendation method, LibRec, using association rule mining and collaborative filtering techniques. It searches for the similar repositories to recommend related libraries for developers. Another effort [37] proposes SimApp to identify mobile applications with similar semantic requirements. A recent approach, RepoPal [236], utilizes readme file, and stars property of GitHub repositories to compute the similarity between two repositories.

On the other hand, MUDABLUE [103] is the first automatic approach to categorize the software repositories using Latent Semantic Analysis (LSA) on source code. Considering the source code as plain text, they create a identifiers-software matrix and apply LSA on it to compute the similarity. Later, another study [207] categorizes the software repositories applying Latent Dirichlet Allocation (LDA) on the source codes. A recent study named CLAN [133] computes the similarity between repositories by representing the source code files as a term-document matrix (TDM) where every class represents a row and the repositories are the columns.

Finally, a very recent study [148, 149] proposes CrossSim, a graph based similarity computation approach using both high level star property and API call references in source code files in a repository. Utilizing the mutual relationship, they represent a set of repositories as a graph and compute the similar repositories of a given repository from the graph. However, their work is limited by the external library call which may fool as the similarity will largely depends on it. Another study [28] has confirmed that CrossSim may identify dissimilarity based on external API usage while internally implementing similar functionalities.

b. Embedding approaches: NLP-based techniques have been well established to mine and represent summarized information from GitHub repositories [86–88, 90, 91]. However, a recent advancement in NLP has opened a whole new way of feature representation, a neural network based feature learning approach for discrete objects. First, introduction of word2vec [137], a continuous vector representation of words from very large corpus, has paved the way. Later, another study named doc2vec [114] introduces a distributed representation of variable length paragraph or documents. More recently, the embedding concept is being shared in other domains and has gained enormous success in effective feature representation such as graph embedding [70, 145], topic embedding [153], tweet embedding [51], and code embedding [6, 79, 101, 202].

4.7 Conclusions

We present Repo2Vec, an approach to represent a repository in an embedding vector utilizing data from three types of information sources: (a) metadata, (b) repository structure, and (c) source code available in a repository. The main idea is to aggregate the embedding representations from these three types of information. Our work can be summarized in the following points:

1. **A highly effective embedding:** Repo2Vec is a comprehensive embedding approach, which enables us to determine similar repositories with 93% precision.

2. **Improving the state of the art:** Our approach outperforms the best known method, CrossSim, by a margin of 15% in terms of precision. Also, it finds nearly twice as many Strongly Similar repositories and 30% less False Positives.

3. **Facilitating the identification of malware**: Our approach can classify the malware and benign repositories with 98% precision outperforming previous studies.

4. **Enabling meaningful clustering**: Our approach identifies a tree hierarchy of repositories that aligns well with their purpose and lineage.

In the future, we first plan to extend the work with a larger dataset and a more extensive ground truth dataset. In fact, we would like to help develop a community-wide benchmark that will facilitate further research. Second, we would like to extend our work to other programming languages, which hinges mostly on developing a code2vec capability for other languages. It would be interesting to see if different languages lend themselves to embedding representations the same way we are able to do here with Java.

Chapter 5

PIMan: A Comprehensive Approach for Establishing Plausible Influence among Software Repositories

Determining the influence among software repositories is an essential building block for studying the dynamics of software evolution and collaboration in online Open Source Software (OSS) platforms. These OSS platforms contain a massive number of repositories and facilitate the engagement of millions of users [139]. GitHub is arguably the largest such platform with more than 32 million repositories and 34 million users exhibiting significant collaborative interactions [176]. As these are open source coding platforms, there are no restrictions for users to create new repositories. Naturally, it attracts users with varied

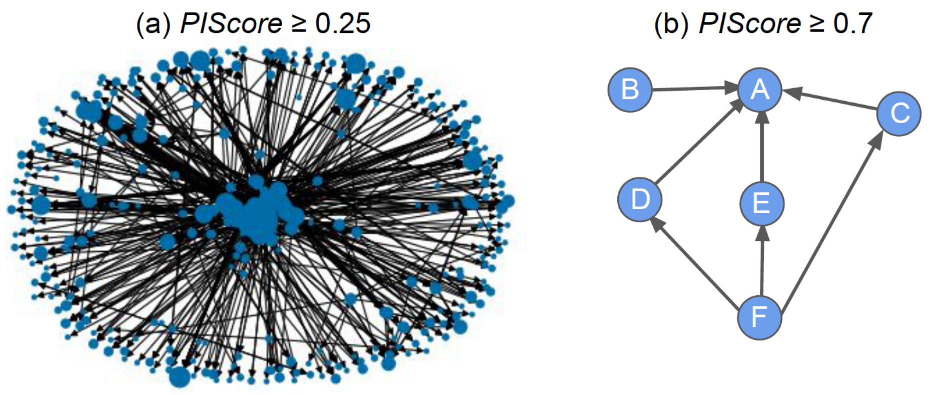


Figure 5.1: Our approach captures plausible influence between repositories in a tuneable and visually powerful way. We show: (a) the dense PIGraph with lower influence threshold, $PIScore \geq PIT = 0.25$ with 426 nodes, and 1191 edges, and (b) the sparse PIGraph with higher influence threshold, $PIScore \geq PIT = 0.7$ with 6 nodes, and 7 edges.

expertise levels, and they develop their own software, and also copy and duplicate other repositories. As a result, there is a significant collaboration and code reuse [60, 198] in these platforms. Researchers are interested in studying the dynamics of the ecosystem at the repository level, which could reveal insights into software evolution. Interestingly, there is also a significant malware development activity within public repositories, which could be valuable for security analysts [86, 175]. Therefore, we decide to focus on establishing influence among malware repositories in our work.

Given two repositories, how can we quantify the level of influence between them by analyzing their platform-level interactions? This is the problem that we address here. We can identify the following types of interactions: (a) an author can “appreciate” the repository of another author by starring it, watching it, and commenting on it, (b) an author can *follow* another author, and (c) an author can fork popular repositories in the archive. The key challenge is developing a comprehensive approach for defining influence, and even further, estimating the possibility that such an influence has occurred.

Combining these different types of information is a non-trivial task. Note that here we define influence which is a much broader concept than, say, code-level similarity which obviously overlaps with influence but is ultimately different. In fact, our intention is to explore the interplay between influence and code-level similarity. Establishing similarity at the code level is an open research question in its own right and here we only employ it as an indirect validation of our influence metric in this study. Once we can quantify the influence between two repositories, we can understand the influence interactions for a group of repositories.

The challenge: the elusive nature of influence. We want to stress that the goal is to identify the **likelihood** of influence between repositories as it is hard to prove influence with certainty. First, the concept of influence is inherently challenging and goes beyond the code-level similarity. For example, author A can be inspired by repository R, even copy it initially, but then improve it substantially. The final repository can have minimal code level similarity with the repository R. Second, it is nearly impossible to establish influence even if we define it in a very strict sense: author A copies (parts of) repository R of author B. For example, we can think of a scenario where authors A and B emulate or copy a third repository (or some other source). So even if two copied repositories are identical, that does not prove that one has influenced the other in a strict sense. However, we argue that the likelihood of influence will be very helpful in studying software evolution and collaboration patterns.

There are relatively few efforts that focus on establishing influence among repositories. Most efforts typically focus on one or a few high-level metrics such as forking relationships and number of stars, or the focus on influence and popularity of authors. First, there are efforts that study the popularity and importance of repositories [84, 173] using a limited number of readily available metrics such as star and fork relationships. However, these works do not focus on the likelihood of influence between a pair of repositories. Second, there are studies [83, 86, 175] that focus on the author-author level interactions and popularity and not on the repository-repository level. Finally, there are efforts that study code-level similarity, which we view as complementary to our work. We discuss related work in more detail in the Section 5.5.

As our key contribution, we propose PIMan[1], a comprehensive multi-dimensional approach to identify pair-wise **plausible influence** at the repository level. First, our method quantifies the pair-wise influence of repositories considering most social-interaction dimensions comprehensively: (a) repository level interactions, (b) author level interactions, and (c) temporal considerations. Second, our method generates a Plausible Influence Graph (**PIGraph**) for a group of repositories, where an edge between two repositories exists if the **Plausible Influence Score (*PIScore*)** of these repositories is greater than the **Plausible Influence Threshold (*PIT*)**. Our approach is a comprehensive and flexible way to combine the multifaceted information using either our default or customized parameter values, tuned to match the niche needs of study.

We deploy our approach on a dataset of 1013 malware repositories from GitHub [175]

[1]PIMan stands for Plausible Influence Modeling and Analysis.

and study the influence relationship of its repositories. These repositories have been created during a span of approximately 12 years which can capture the long-term effects and phenomena. Although our approach can be applied to any type of repository, focusing on malware repositories: (a) hones in on a rather well-defined community, and (b) could manifest practical value in combating cyber-crime. Our key results can be summarized in the following points.

a. PIMan models influence flexibly with a directed graph. Our approach captures repository-level influence relationships with a flexible and informative **plausible influence graph (PIGraph)**. In Figure 5.1, we illustrate the descriptive power of our approach by looking at different levels of influence using the *PIT* threshold. For $PIT = 0.25$, we get a dense graph of 426 nodes and 1191 edges, whereas for $PIT = 0.7$, the graph contains 6 nodes and 7 edges. By tuning this threshold, we can hone in on different intensity levels of influence. In addition, we show that our *PIScore* is significantly different from other straightforward metrics of popularity (see Figure 5.9).

b. Plausible influence as a proxy for code-level similarity. We show that our definition of influence correlates with code-level similarity as shown in Figure 5.5 (Spearman coefficient, $\rho = 0.79$, and $p - value = 1.26e - 19$). We consider the following as an indirect validation of our approach: (a) our quantification of influence is reasonable and (b) it can provide useful results, e.g. pointing us to repositories with actual collaboration and overlap at the code level.

c. Finding evidence of significant collaboration. We observe significant collaboration and influence among the repositories in our dataset. First, we identify 28 connected components in our plausible influence graph ($PIT = 0.25$). We find that 71% of the components have less than 5 repositories, while 7% components have more than 15 repositories. In addition, the top 10 most influential repositories have directly influenced 260 repositories in a non-trivial way ($PIT = 0.25$). In fact, the most influential repository has directly influenced 67 repositories.

d. Revealing emerging structures and families. Analyzing the influence graph, we can find interesting lineage and clusters of influence. We find 19 repositories that influenced at least 10 other repositories directly and spawned at least two "families" of repositories. For example, the repository *"vaginessa/android-overlay-malware-example"* is a highly influential information-crawler android malware created on June 17, 2015. It influenced 10 repositories directly and spawned three families of malware: (a) ransomware with 3 repositories, (b) malware for stealing user credentials, such as keyloggers, with 4 repositories, and (c) RAT malware with 3 repositories.

Our work in perspective. Our approach is an essential capability towards understanding the dynamics and evolution of online platforms at the repository level. In fact, it can be seen as a powerful component that can complement other features such as popularity or similarity at the code level, which capture related but different aspects of the repositories.

5.1 Background

Our work focuses on GitHub, the largest open-source software archive. Here, we provide some background on the repository information that are available in GitHub.

A. The information in GitHub. GitHub is a massive software hosting platform, that enables users to create public repositories to store, share, and collaborate on projects, and provisions a good number of features for the users to do different social networking interactions. The following describes the key elements of a GitHub repository and its author.

1. Repository features: A repository contains the following types of information.

a. Metadata: A repository in GitHub has a large number of metadata fields. The most notable ones are: (a) title, (b) description, (c) topics, and (d) readme file. All these fields are optional, and are provided by the author. As the text fields are provided by the repository author, they can be most often unstructured, noisy, or missing altogether.

b. Source code: The core element of a software repository is its source code. The repository can contain the software projects which are written in various programming languages such as C/C++, Java, and Python.

2. Social interaction features: It is helpful to group social interactions into repository and author level features.

a. Repository level interaction: GitHub provides functionality for social interaction at the repository level. A repository can be (a) starred, (b) watched to get notification about the updates, (c) receive comments, and (d) forked by other authors.

b. Author level interaction: GitHub enables authors to create a profile by adding social information. Authors can follow other authors which is a direct indication of interest and appreciation. As such, one would expect that followers are likely to be influenced by that author and her repositories.

These two types of interaction define the repository popularity in GitHub which we quantify as *RepoPop.* Note that the repository versus author level features is not that strict; as for example starring a repository by author A implicitly conveys appreciation for both the repository and the author.

B. Fundamental techniques and algorithms. We provide an overview of two fundamental techniques that we leverage in our work: (i) Repo2Vec [176] to represent a repository into a vector and (ii) HackerScope [85,86] to identify popular authors in GitHub.

1. Code-level similarity - RepoSimScore: Quantifying repository similarity at the code level is not trivial. For validation purposes, we will rely on Repo2Vec [176], an embedding approach that represents a GitHub repository in an M-dimensional vector utilizing data from three types of information sources that enables the repository similarity computation, classification, and clustering tasks. An embedding (a.k.a. distributed representation) is an unsupervised approach for mapping entities, such as words or images, into a relatively low-dimensional space by using a deep neural network on a large training corpus [114, 137]. The approach combines the repository metadata, the code, and the directory structure of the repository to estimate *RepoSimScore*, the similarity between two repositories.

2. Determining node significance in a directed graph: Several approaches exist for capturing the significance of interacting nodes in a complex network. In our case, the interactions are captured by a directed graph, which points to the use of hyperlink-induced topic search algorithm [85, 86, 122]. The algorithm used in these previous studies identifies influential authors by incorporating a HITS approach on the Author-Author graph which captures the interactions of the authors as we will discuss later.

C. Datasets. Our main dataset is D_All, which consists of 2089 Java malware repositories collected by a prior study [176] to whom we are grateful whose goal is to transform a GitHub repository into an M-dimensional embedding vector and determine the similarity between two repositories. The dataset covers a fairly wide range of malware families including: Botnets, Keyloggers, Viruses, Ransomwares, DDoS, Spywares, Exploits, Spam, Malicious code injections, Backdoors, and Trojans. We conduct some additional crawling of data (Using GitHub REST API) for each repository, as the initial dataset was missing some repository features that we intend to use here, namely the identity of authors interacting with a repository. The dataset contains 433 original repositories and 1656 fork repositories. In Section 5.2, we discuss how we leverage the existence of these fork relationships in our study.

We create three datasets which we use to tune parameters and validate assumptions. First, we create D_50 by randomly selecting 50 pairs of repositories with *RepoSimScore* $\geq$ 0.8. Second, we create D_F50 by randomly selecting 50 pairs of forked repository. Third, we create dataset D_3Levels as follows. We randomly select 90 pairs of repositories from three ranges of *RepoSimScore*: (a) 30 pairs from range [0-0.25), (b) 30 pairs from range [0.25-0.75), and (c) 30 pairs from range [0.75-1.00].

5.2 Proposed Method

The main idea behind PIMan is to create a directed weighted graph among repositories by computing Plausible Influence Score, *PIScore*. Our approach can be summarized in three steps. In the first step, we compute the influence score across three dimensions: (a)

Symbol	**Description**
PIMan	Plausible Influence Modeling and Analysis
PIGraph	Plausible Influence Graph
PIScore	Plausible Influence Score between two repositories
PIT	Plausible Influence Threshold to create PIGraph
TPIScore	Total Plausible Influence Score for a repository
RepoSimScore	Code-level Similarity Score between two repositories
RepoPop	Repository Popularity combining number of stars, forks, and watches
RAI	Repository-Author interaction score
AAI	Author-Author interaction score
APop	Author Popularity score in the network
D_All	Dataset with 2089 repositories
D_50	Dataset of 50 repository pairs with *RepoSimScore* ≥ 0.8
D_F50	Dataset of random 50 forked pairs of repository
D_3Levels	Dataset of 90 pairs of repositories within 3 ranges of *RepoSimScore*

Table 5.1: Table of symbols used in this work.

repository-author interaction, (b) author-author interaction, and (c) author popularity in the network. In the second step, we calculate the Plausible Influence Score (*PIScore*) using the weighted summation of influence scores from these three dimensions. Finally, we create a directed graph where an edge is added between two repositories if their *PIScore* satisfies a defined threshold, *PIT*. The overview of our approach is shown in Figure 5.2.

Step 1. Computing three influence scores. For a given ordered pair of repositories R1 by author A1 and R2 by author A2, we want to quantify the likely influence of repository R1 on R2. We compute the influence score for the repositories considering three dimensions: R1-A2 interaction, A1-A2 interaction, and popularity score of author A1.

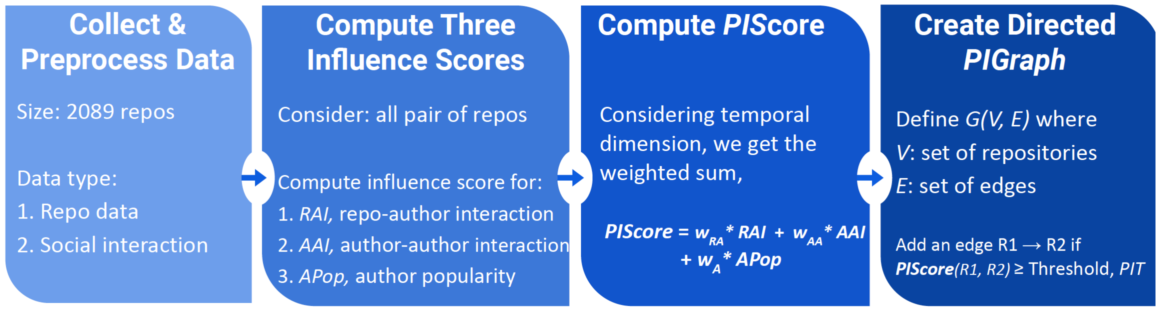

Collect & Preprocess Data
Size: 2089 repos
Data type:
1. Repo data
2. Social interaction
Compute Three Influence Scores
Consider: all pair of repos
Compute influence score for:
1. RAI, repo-author interaction
2. AAI, author-author interaction
3. APop, author popularity
Compute PIScore
Considering temporal dimension, we get the weighted sum,
PIScore = w_RA * RAI + w_AA * AAI + w_A * APop
Create Directed PIGraph
Define G(V, E) where
V: set of repositories
E: set of edges
Add an edge R1 → R2 if PIScore(R1, R2) ≥ Threshold, PIT

Figure 5.2: The overview of PIMan: (a) we define and collect dataset, (b) for all pairs of repositories in the dataset D_All, we compute three influence scores RAI, AAI and $APop$ from repository-author relationship, author-author relationship, and author popularity in the network, (c) we combine three influence scores to get the Plausible Influence Score ($PIScore$) for each pair, and (c) we create the directed influence graph among the repositories using the $PIScore$ value.

Handling forked repositories. A forked repository starts as an exact replica of the initial repository, and therefore we consider this as a substantial influence, although a forked repository can evolve over time. Therefore, our approach would translate fork repositories to a higher $PIScore$ value.

In the remainder of this section, we focus on the more challenging case of non-forked repositories. In the context of our study, we opt to use these fork repositories for validation. Specifically, we pretend that we are not aware of forking relationships, and conduct our influence estimation independently. However, we leverage the knowledge of fork relationships in selecting weight for Equation 5.7 using our dataset D_F50.

a. Repository-Author interaction: We consider all repository level interactions from author A2 to repository R1, and calculate the influence score for repository-author interaction, RAI. First, we compute the Starring Score (SS), Forking Score (FS), Watching Score (WS), and Commenting Score (CS). Here, the Starring Score is equal 1 ($SS = 1$) if author A2 stars repository R1, otherwise it will be 0 ($SS = 0$). Similarly, we compute (FS), (WS), and (CS) to capture forking, watching, and issue commenting interaction score. Finally, we normalize the score to keep it in the [0,1] range by arithmetic mean of these four scores.

We can combine these scores into a single score using many different ways and by giving different weights to individual scores. Here we decide to first explore using the equal weights to all scores, therefore using the following formula:

$$RAI = \frac{SS + FS + WS + CS}{4} \tag{5.1}$$

As we will see later, this way of calculating the score gives good results. In the future, we explore the use of weights and other ways to combine the individual scores.

b. Author-Author interaction: We consider significant interactions from author A2 to author A1 to calculate an influence score (AAI) based on the author-author interactions. First, we compute the Following Score (FS), Other Repository Fork Score (FS_{O_R}), Other Repository Star Score (SS_{O_R}), Other Repository Watch Score (WS_{O_R}), Other Repository Comment Score (CS_{O_R}) if author A2 follows author A1, A2 forks, stars, watches any repository of A1 (except R1), and A2 comments on any repository of A1 (except R1), respectively. Finally, we assign the normalized mean influence score to AAI for the aforementioned interactions.

As above, we combing the individual scores giving the same importance using the formula below:

$$AAI = \frac{FS + FS_{O_R} + SS_{O_R} + WS_{O_R} + CS_{O_R}}{5} \tag{5.2}$$

In future, we intend to consider other ways to combine these interactions.

c. Author popularity: In GitHub, popular authors get more attention and are more likely to influence other authors. The prominence of an author here can be captured by several aggregate metrics such as the number of followers, the total number of stars across all their repositories, etc. as we described previously. As a result, quantifying the overall prominence of an author is not trivial. In order to compute the score, we extend the approach that we mentioned earlier in the Section 5.1.

c.1. Generating the author-author interaction graph. We create a graph to capture the network-wide interaction among authors. In more detail, we define a weighted labeled multi-digraph where the nodes are the authors, and we consider six types of relationships that are represented by directed edges with different labels (u, v) from author u to v. These edges can be: (a) a follower edge: when u follows v, (b) a fork edge: when u forks a repository of v, (c) a star edge: when u stars a repository of v, (d) a watch edge: when u watches a repository of v, (e) a contribution edge: when u contributes code in a repository of v, and (f) a comment edge: when u raise a issue comment in a repository of v. These relationships capture the most substantial author-level interactions.

c.2. Edge weight calibration. The above multigraph consists of six different relationships, whose "significance" as an interaction differs. For example, it is "cheaper" to star a repository compared to forking it, which shows intention to use and modify the original code. We want to appropriate weigh the importance of each relationships and, to do this,

```
Algorithm 1:
Compute_Producer-Connector_Score: An algorithm to compute Producer Score, $PS$ and Connector Score, $CS$ in a weighted multi-digraph of repository authors.

Input: A directed multi-digraph, G
Output: Producer Score $PS$, Connector Score $CS$
1  for each node u in G do
2      $PS_u = 1.0$
3      $CS_u = 1.0$
4  end
5  while convergence != True do
6      for each nodes u in G do
7          $PS_u \leftarrow \sum_v w(v,u) * CS_v$, for all $v$ pointing to $u$
8          $CS_u \leftarrow \sum_z w(u,z) * PS_z$, for all $z$ pointed by $u$
9      end
10     Normalize $PS$ and $CS$ so that $\sum_u PS_u + \sum_u CS_u = 1$
11 end
12 return $PS$ and $CS$
```

Figure 5.3: An algorithm to compute Producer Score, PS and Connector Score, CS in a weighted multi-digraph of repository authors.

we consider how rare each relationship is. Intuitively, a rare relationship should get higher importance. Specifically, we consider the weight of a type of edge inversely proportional to a measure of its relative frequency. We calculate the average degree d_{type} for each type of edge, and normalize it dividing by the minimum average degree d_{min}. We get the weight for each type of edge following the equation $w_{type} = \frac{d_{min}}{d_{type}}$: (i) follower edge weight, $w_{follower} = \frac{d_{min}}{d_{follower}}$, (ii) fork edge weight, $w_{fork} = \frac{d_{min}}{d_{fork}}$, (iii) star edge weight, $w_{star} = \frac{d_{min}}{d_{star}}$, (iv) watch edge weight, $w_{watch} = \frac{d_{min}}{d_{watch}}$, (v) contribution edge weight, $w_{contribution} = \frac{d_{min}}{d_{contribution}}$, and (vi) comment edge weight, $w_{comment} = \frac{d_{min}}{d_{comment}}$

c.3. Author popularity score computation. We define two roles of an author in the ecosystem: (a) **producer**, who creates repositories, and (b) **connector**, who interacts with the other authors by following them, and starring, forking, watching, and commenting on their repositories. To quantify the popularity of the author depending on the roles played, we associate each node u with two values: (a) producer score, PS_u, and (b) connector score, CS_u. The algorithm iteratively updates the producer and connector scores until (i) they converge, or (ii) tolerance threshold is reached. Algorithm 5.3 provides the high-level pseudo-code of our approach to compute PS and CS. First, we initialize PS_u and CS_u to value 1.0. Second, we iteratively update values as follows:

(i) for all nodes v with a directed edge to u, (v, u):

$$PS_u = \sum_v w(v, u) * CS_v \tag{5.3}$$

(ii) for all nodes z with a directed edge from u, (u, z):

$$CS_u = \sum_z w(u, z) * PS_z \tag{5.4}$$

(iii) we normalize PS_u and CS_u,

$$\sum_u PS_u + \sum_u CS_u = 1 \tag{5.5}$$

We repeat this step until the values converge. For convergence, we set a tolerance threshold for the change of the value of any node.

Finally, we take a threshold T_P for PS_u score and T_C for CS_u score, and (i) if $PS_u \geq T_P$ and $CS_u \geq T_C$, the author is highly active and popular, (ii) if $PS_u \geq T_P$ and $CS_u < T_C$, the author is highly active by just creating repositories, (iii) if $PS_u < T_P$ and $CS_u \geq T_C$, the author is highly active in starring, forking, watching and commenting, and (iv) if $PS_u < T_P$ and $CS_u < T_C$, the author is not active. These two scores capture different aspect of authors popularity. Hence, we will get the combined network-wide author

```
Algorithm 2:
Get_PIScore: An algorithm to compute the Plausible Influence Score of repository R1 to R2.
Input: Two repositories: R1 and R2
Output: Plausible influence score, PIScore
1  if R2 is created earlier than R1 then
2      $PIScore \leftarrow 0.0$
3  end
4  else
5      $A1, A2 \leftarrow$ get author name of repository R1 and R2
6      $RAI \leftarrow$ compute interaction score of R1 and A2 interactions
7      $AAI \leftarrow$ compute interaction score of A1 and A2
8      $APop \leftarrow$ compute popularity score of A1 in the authors network
9      $PIScore \leftarrow w_{RA} * RAI + w_{AA} * AAI + w_A * APop$
10 end
11 return PIScore
```

Figure 5.4: An algorithm to compute the Plausible Influence Score of repository R1 to R2.

popularity score as follows,

$$APop = PS_u + CS_u \tag{5.6}$$

while in the future we will consider other ways to combine these two scores.

Step 2. Plausible influence score (*PIScore*). We attribute the influence score of repository R1 to repository R2 as *PIScore* $(R1, R2)$.

a. Combining the influence scores. Algorithm in Figure 5.4 provides the high-level pseudo-code of the basic workflow to get the score. We define the *PIScore* of $R1$ to $R2$ to be the weighted sum of the three scores from repository-author interaction, author-author interaction, and author popularity score. We compute *PIScore* using the following equation,

$$PIScore = w_{RA} * RAI + w_{AA} * AAI + w_A * APop \tag{5.7}$$

where w_{RA}, w_{AA} and w_A are the weights for the score derived from the repository-author interaction, author-author interaction, and author popularity. We discuss in detail how we calibrate these weights in the next section.

b. Temporal considerations. In general, we propose to adjust the influence score by considering other practical considerations. The most critical is time. The key idea is simple: a recent repository cannot have influenced a repository in the past. However, the implementation can hide several subtleties. We outline two approaches.

Approach 1. We can simply consider the creation time of a repository as a sufficient indication for creating a temporal order. In this approach, if the creation time of repository R2 (T2) is earlier than that of repository R1 (T1), the plausible influence score (*PIScore*) of R1 to R2 is set to zero. Otherwise, we use the influence score as calculated above.

Approach 2. We can consider a "temporal phases of influence" where we recognize that: (a) repositories are created over time, (b) the effect of time can be a real value between zero or one. In other words, we can have a multi-step weight where for different time differences of the repository creation $DT_{creation} = T1 - T2$ we can have different values for a modifying factor **Temporal Modifying Factor (*TMF*)** within [0,1]. For example: one rule could be: if $DT_{creation} > -2$ weeks, then $TMF = 1$, which means we "allow" a repository to influence the repository that was created 2 weeks earlier. The rationale is that software development takes time. Another thought is to consider that a really old repository is less likely to influence a recent repository, say 8 years later, given the fast pace of evolution in software and techniques, so if $DT_{creation} > 8$ years, then $TMF = 0.2$. We can then adjust the *PIScore* by multiplying it with TMF.

Given time and space constraints, we adopted approach 1 in our work, which seems to give meaningful results. In the future, we intend to develop a sophisticated temporal consideration framework. However, such a framework will need to be grounded on observed properties of repositories, such typical duration, temporal properties of the intensity of development as seen by the commits in the code, and observations on how authors interact with other repositories, e.g. how often does an author stars a 8-year-old repository.

Step 3. Creating the PIGraph: We create the PIGraph as a directed weighted graph that captures the plausible influence among repositories. Formally, we define a directed weighted graph: $PIGraph(V, E)$, where V is the repository set, E is the set of edges, and we denote the weight of an edge e as $w(e)$. We consider an edge e between repositories $R1$ to $R2$, if *PIScore*$(R1, R2)$ (the influence score of $R1$ to $R2$) is greater than or equal to a threshold *PIT*, and assign the weight of the edge $w(e) =$ *PIScore*$(R1, R2)$.

After generating the PIGraph, each node can be an influencer (having outgoing edges), an influencee (having incoming edges), or both. We use the term **influence outdegree** to refer to the number of outgoing edges of a node. We also define **Total Plausible Influence score,** *TPIScore*, of a repository to be the sum of the weights of all its outgoing edges. Both these metrics capture the network-wide influence of a node as we discuss later.

5.3 Tuning and Evaluating Our Approach

We present a systematic approach to select appropriate values for the weight parameters of our approach and we evaluate the effectiveness of PIMan.

A. Tuning the author popularity parameters. As we saw in the Section 5.2, the author network consists of six different relationships which show significantly different distribution. To provide appropriate importance, we make the weight of a type of edge inversely proportional to the measure of its relative frequency. We calculate the average degree of d_{type} for each type of edge: (i) $d_{follower} = 0.96$, (ii) $d_{star} = 0.60$, (iii) $d_{fork} = 1.98$, (iv) $d_{watch} = 1.01$, (v) $d_{contributor} = 0.36$, and (vi) $d_{comment} = 0.29$. We define the weight for each type, w_{type}, by normalizing the average degree using the minimum average degree, that is $w_{type} = \frac{d_{min}}{d_{type}}$. Here, the minimum average degree $d_{min} = d_{comment} = 0.29$. This way, we set (i) following edge weight: $w_{follower} = 0.30$, (ii) star edge weight: $w_{star} = 0.48$, (iii) fork edge weight: $w_{fork} = 0.15$, (iv) watch edge weight: $w_{watch} = 0.29$, (v) contributor edge weight: $w_{contributor} = 0.8$, and (vi) comment edge weight: $w_{comment} = 1.0$.

Furthermore, we calculate the popularity of authors, $APop$, by combining the Producer Score (PS) and Connector Score (CS) which are calculated following the Algorithm 5.3. The algorithm iteratively updates PS and CS for each node in the network. For the convergence, we set a tolerance threshold of 10^{-10} for the change of the value of any node. After 522 iterations, we obtain the converged values of PS and CS for each author. Finally, we assign the sum of PS and CS as the author popularity ($APop$) and rank the authors based on the derived popularity scores.

B. Tuning the weights for the *PIScore* computation. Here, we explain how we can systematically determine appropriate weights to ensure that each type of influence is considered adequately in Equation 5.7.

a. Weight selection: We choose the weight for each type of influence by "training" the weights to reflect the likelihood that there is code-level similarity between the repositories. Specifically, we use a set of repositories for which we have *RepoSimScore*, the code-level similarity, as we discussed in the Section 5.1. We use the Spearman correlation coefficient [197] between the influence score of each dimension and code-level similarity. Note that we only do this once to calibrate the weights.

In more detail, we calculate the weights in two steps by using our D_50 dataset. First, we calculate the correlation coefficient between *RepoSimScore* and influence score of each dimension for each pair of repository in D_50 dataset. We find that RAI (Influence Scores of repository-author interaction) is positively correlated to the *RepoSimScore* (Spearman coefficient, $\rho_{RA} = 0.38$ and p-value = 3.82e-9). By contrast, the correlation coefficient of Influence Scores of author-author interaction (AAI) and author popularity ($APop$) to the *RepoSimScore* are $\rho_{AA} = 0.15$ with p-value = 1.53e-12 and $\rho_A = 0.06$ with p-value = 1.92e-8 respectively. Finally, we measure the weights in a way that reflects the ratio of the corresponding ρ values, while the sum of the three weights should be equal to one, which leads us to the following weights: $w_{RA} = 0.65, w_{AA} = 0.25$ and $w_A = 0.10$, which we use in Equation 5.7.

b. Validating our weight selection. We further evaluate the effectiveness of the weight selection of Equation 5.7 with dataset D_F50 which consists of pairs of forked repositories. The assumption is that forked repositories are supposed to be highly

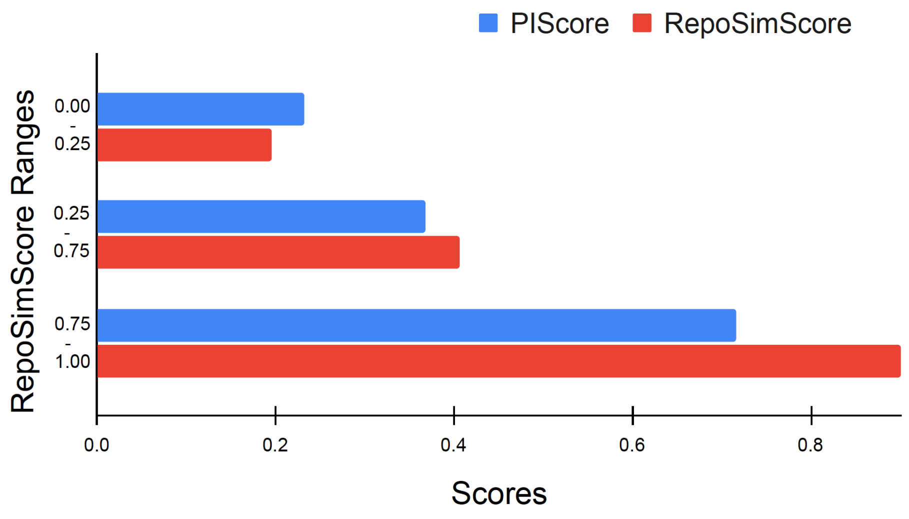


Figure 5.5: The Plausible Influence Score (*PIScore*) is highly correlated with the code-level similarity (*RepoSimScore*) (Spearman coefficient, ρ = 0.79, and p-value = 1.26e-19) using the D_3Levels dataset.

influenced by the original repositories as we discussed earlier. We find that the repository-author interaction score (*RAI*) is the most relevant dimension as the Spearman correlation coefficient with respect to *RepoSimScore* is $\rho = 0.52$ with p-value = 3.32e-11, whereas the values for AAI and $APop$ are $\rho = 0.22$ with p-value = 1.46e-8 and $\rho = 0.09$ with p-value = 2.73e-8 respectively. These coefficient scores validate that the repository-author relationship is the most relevant dimension in identifying influence among repositories which is why we correctly consider it with a higher weight in our approach as we describe above.

C. Evaluating our approach: We present our effort to establish whether our influence metric provide reasonable results.

Plausible Influence and code-level similarity. We find that our definition of influence correlates relatively strongly with code-level similarity as shown in Figure 5.5. In more detail, we use the dataset D_3Levels, where each level in that group corresponds to low, medium, and high *RepoSimScore* as explained earlier. (Note that dataset D_3Levels is different than D_50, which we used earlier to determine the weights.) We then calculate the influence score between every pair of repositories in D_3Levels. (Note that we select the relationship with the highest influence between R1-R2 and R2-R1). We plot the average influence score per pair and *RepoSimScore* per pair grouped by level for ease of viewing. We see that the two scores are strongly correlated. Using the original data points, we find a Spearman coefficient $\rho = 0.79$ with p-value = 1.26e-19, which indicates a robust correlation.

To investigate further, we manually assess 10 randomly selected pairs with high influence scores. For example, we find that *"androidtrojan1/android_trojan"* and *"vaginessa/android-overlay-malware-example"* have high *RepoSimScore*. We also find that author *"vaginessa"* follows, stars and forks 5 of repositories of author *"androidtrojan1"*, which leads to a high influence score.

Note that, in the next section, we will provide additional indications that our approach provides interesting and meaningful results. Specifically, we apply our method on the D_All dataset which provides several interesting observations. We argue that: (a) these observations are useful and insightful, (b) they are aligned with and corroborated by other approaches.

5.4 Study: Results and Insights

We apply PIMan on our dataset and discuss the key findings.

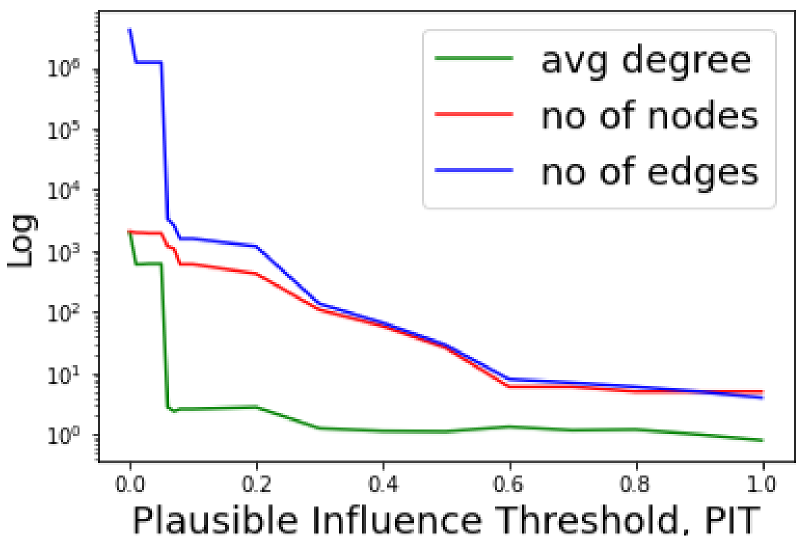


Figure 5.6: Increasing the *PIT* threshold reduces the PIGraph network size keeping only higher influence edges.

5.4.1 Part 1. Studying the PIGraph

A. The effect of the influence threshold *PIT* on the PIGraph. We create the directed influence graph among the repositories in the dataset following the steps described in the Section 5.2. We add an edge from repository $R1$ to repository $R2$ if $PIScore(R1, R2) \geq PIT$. We study the PIGraph for different threshold values and plot the graph properties: average degrees, number of nodes, and number of edges in Figure 5.6. It implies that increasing the threshold reduces the size of the network. In addition, it also exhibits the highly influential characteristics when the threshold $PIT \geq 0.40$. Figure 5.1 shows that our model produces (a) the dense PIGraph having the threshold $PIT \geq 0.25$ with 426 nodes and 1191 edges, and (b) the sparse PIGraph having the threshold $PIT \geq 0.7$ with 6 nodes and 7 edges.

Observation: The above plot in Figure 5.6 provides some guidance for selecting a value for the threshold parameter *PIT*. We observe that the distribution of the graph properties creates a knee in the range between 0.1 and 0.35. In order to ensure non-trivial influence, we use a value of 0.25 in the rest of our work unless otherwise stated.

Indirect validation: We argue that this analysis suggests that our influence metric captures a reasonable breadth of behaviors contingent on the *PIT* threshold. Capturing a breadth of behaviors is a desirable property for a modeling approach.

B. The distribution of repository influence. The number of directly influenced repositories follows a skewed distribution with several extremely influential repositories. Here, we focus on studying the PIGraph ($PIT = 0.25$) and we focus on the edges that represent direct influence: we use the term **influence outdegree** of a repository to refer to the number of directly influenced repositories for a given PIGraph. We find 39% of the repositories having zero direct influence on other repositories while 8% of the repositories influenced at least 20 repositories. In aggregate, the top 10 most influential repositories have directly influenced 260 repositories in a non-trivial way. Furthermore, the most influential repository has directly influenced 67 repositories.

C. Influence: intensity versus the number of repositories. The Total Plausible Influence score (*TPIScore*) provides a different way to capture influence by also considering the intensity of influence. We explore the relationships between *TPIScore* and number of directly influenced repositories (outdegree) by producing the scatterplot shown in Figure 5.7. There is a strong, arguably linear, correlation between the two metrics. In addition, this plot can help us identify "niche" repositories with a "cult" following:

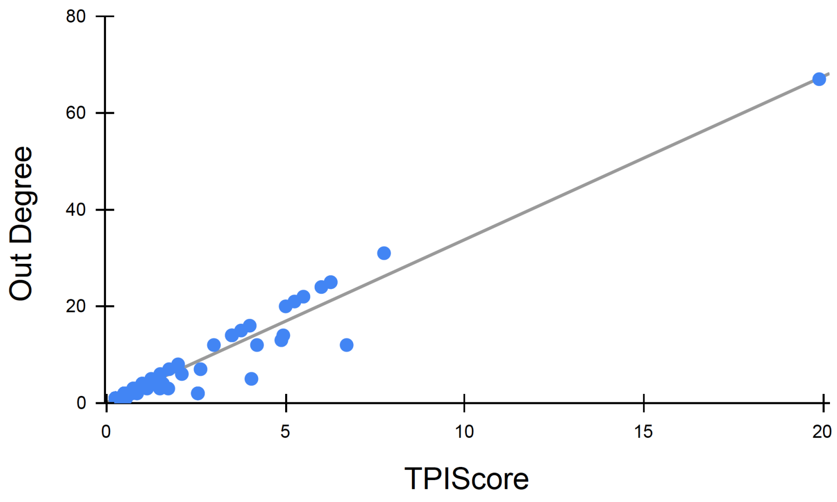


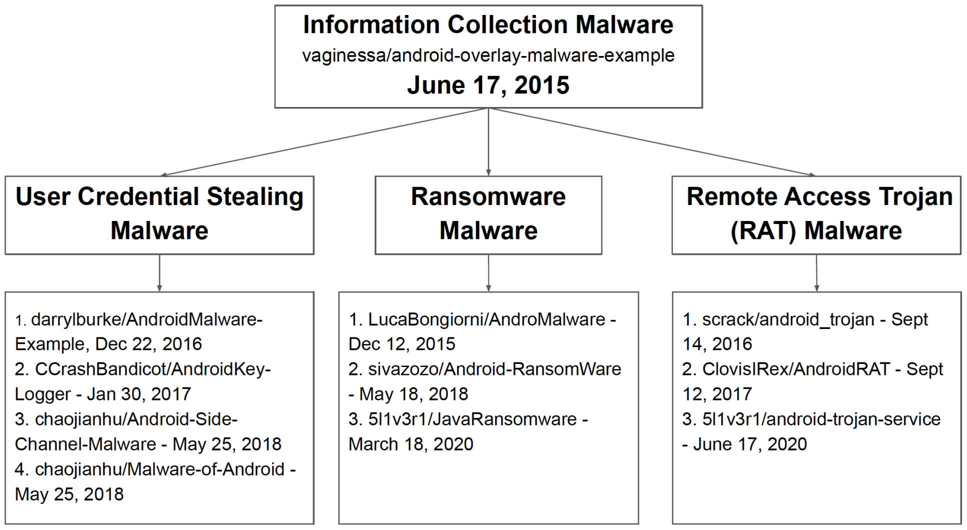
Figure 5.7: Number of directly influenced repositories (Outdegree) vs Total Plausible Influence ($TPIScore$) exhibits a linear correlation for D_50 dataset.

repositories with relatively small outdegree but high influence. As an example, repository *"tiagorlampert/sAINT"* is highly influential ($TPIScore = 6.71$) with only 12 influencees.

5.4.2 Part 2. Clusters and Lineage of Influence

A. Finding evidence of collaboration. We want to identify the relationships and groups of high influence. Overall, we observe significant collaboration and influence among the repositories in our dataset. First, we identify 28 connected components in PIGraph ($PIT = 0.25$). We find that 71% of the components have less than 5 repositories while 7% of components have more than 15 repositories. This is a strong indication of substantial collaboration among the repositories, especially if we consider that we have already set a high threshold for the influence in the graph.

Indirect validation: How cohesive are these components? To answer this question,



Information Collection Malware
vaginessa/android-overlay-malware-example
June 17, 2015
User Credential Stealing Malware
Ransomware Malware
Remote Access Trojan (RAT) Malware
1. darrylburke/AndroidMalware-Example, Dec 22, 2016
2. CCrashBandicot/AndroidKey-Logger - Jan 30, 2017
3. chaojianhu/Android-Side-Channel-Malware - May 25, 2018
4. chaojianhu/Malware-of-Android - May 25, 2018
1. LucaBongiorni/AndroMalware - Dec 12, 2015
2. sivazozo/Android-RansomWare - May 18, 2018
3. 5l1v3r1/JavaRansomware - March 18, 2020
1. scrack/android_trojan - Sept 14, 2016
2. ClovisIRex/AndroidRAT - Sept 12, 2017
3. 5l1v3r1/android-trojan-service - June 17, 2020

Figure 5.8: Lineage and influence: an influential information-collection malware repository spawned three fairly distinct malware sub-families. Here we show only the directly influenced repositories.

we manually analyze a set of components selected randomly. We find one component with 16 repositories exclusively focused on Android malware, while another component with 235 repositories contained three different families of malware. We argue that this is an additional indication that our approach provides meaningful results.

B. Lineage: highly influential repositories spawn multiple repository “families”. In our analysis, we investigate the effect of highly influential repositories, and observe the following phenomenon. We find 19 repositories that have influenced at least 10 repositories directly, and have spawned at least two malware “families”. For example, the repository *vaginessa/android-overlay-malware-example* is a highly influential information crawler android malware created on June 17, 2015. It influences 10 repositories directly and has spawned three families of malware: (a) user credential stealing malware, (b) ransomware malware, and (c) remote access trojan (RAT) malware. Figure 5.8 depicts the lineage story of *vaginessa/android-overlay-malware-example*.

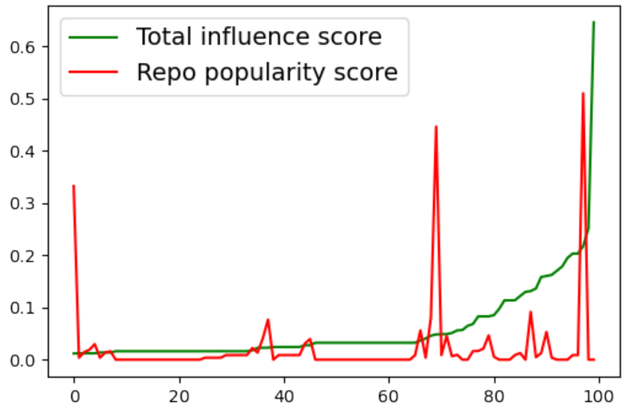


Figure 5.9: The Total Plausible Influence score (*TPIScore*) of a repository is significantly different from the popularity of the repository (*RepoPop*). We show the normalized scores for dataset D_50.

5.4.3 Part 3. Repository Influence and Popularity

A. Total influence score, *TPIScore* **captures different aspects compared to repository popularity metrics.** We compare the repository influence computed by PIMan and GitHub popularity metrics, *RepoPop*. First, we create the PIGraph for our dataset with a reasonable influence score, $PIT = 0.25$. This creates an influence graph with 426 nodes and 1191 edges. We calculate *TPIScore*, the total influence score considering the outgoing edge weights for all nodes. Second, we identify the influential repositories based on *RepoPop*, the popularity metrics of GitHub. We compute the sum of: (i) number of stars, (ii) number of forks, and (iii) number of watches to determine the total popularity score for a repository. We rank the repositories according to the popularity score.

In Figure 5.9, we show the comparison between total influence score, *TPIScore*, and repository popularity score, *RepoPop* for the repositories in the D_50 dataset. The two scores exhibit significant differences, and seem fairly uncorrelated. We argue that influence provides a significantly different perspective compared to popularity.

No	**Influential repositories using PIMan**	**Popular repositories using** *RepoPop*
1	00aj99/AndroidMalware-Example	tiagorlampert/sAINT
2	CCrashBandicot/android-_trojan	adonespitogo/AdoBot
3	CCrashBandicot/Android-KeyLogger	M1Dr05/IsTheApp
4	molotof/sAINT	tomgersic/AndroidKey-Logger
5	5l1v3r1/AndroidRansom-Ware	Mandyonze/Droid-Sentinel
6	CristianTuretta/MAD-Spy	PanagiotisDrakatos/Java-Ransomware
7	tiagorlampert/sAINT	harshalbenake/Android-Elite-Virus
8	Mandyonze/Droid-Sentinel	moloch–/Yoshimi-Botnet
9	androidtrojan1/ android_trojan	androidtrojan1/ android_trojan
10	un4ckn0wl3z/Psyber-Project	siberas/sjet

Table 5.2: Top 10 influential repositories identified by PIMan and popularity metric *RepoPop* (which combines stars, forks, and watches) in our D_All.

B. Influential repositories by ***TPIScore*** **versus those ranked by repository popularity.** We want to understand the relationship between influence and popularity of repositories. We show the top 10 repositories identified by both approaches in Table 5.2. We find that the top 10 most influential repositories have influenced 260 repositories when the influence is substantial, $PIT = 0.25$. Comparing the two lists in the table, we see that the two approaches have identified different sets of repositories. They have only three repositories in common, and they also differ in their ranking ([7,8,9] versus [1,5,9]). This indicates that the concept of influence captures a different perspective than popularity. That is not to say that one is better than the other: the two concepts are related but not identical.

5.5 Related Work

There are relatively few efforts that focus on establishing influence between repositories, especially at the "social" level that we consider here. We discuss the related work below grouped in broad areas of focus.

A. Studies of author level roles in GitHub. The efforts in this group have focused on identifying influential authors and not repositories, as we do here. There are some works [31, 125] that studies the ecosystem of developers to measure the social-coding collaboration in GitHub. Focusing on popularity at the author level, some efforts [23, 96] have surveyed developers to study influential users to understand how normal users are influenced by highly influential users on GitHub. Another effort [124] has proposed a ranking-based approach to identify influential authors which can be applied to a heterogeneous network. A recent work [83] has proposed a Following-Star-Fork-Activity based approach to measure user influence in the GitHub developer social network. A more recent work [86] studies influential authors in network considering a Following-Fork-Contribution-Comment relationship in a hacker ecosystem in GitHub. However, our work focuses on establishing influence in the repository level considering (i) repository-author interaction, (b) author-author interaction, and (iii) author popularity in the network.

B. Studies on repository popularity. Most prior efforts focus on quantifying and predicting repository popularity, which is not exactly the same as influence. A recent work [173] has proposed an approach to predict repository popularity using starring and following relationships. There are efforts [13, 204] who have used PageRank to identify popular repositories by analyzing the social coding interaction graph, where two nodes are connected, if the corresponding projects have at least one common developer. Another work [128] uses network centrality measures to identify influence among Python language repositories where the network is created based on the inter-dependencies between projects which are parsed from the setup.py files. Another effort [84] has studied repository influence, but focused *only* on starring relationships, which ignores many other interactions.

None of these works have addressed the problem as formulated here and in the comprehensive fashion of all the relationships that we use in our work.

C. Studies on code-level similarity of repositories on GitHub. Several efforts study code-level similarity, which we view as complementary to our work. First, there are efforts that utilize: (a) metadata [206, 236], (b) the source code [8, 103, 133], (c) both metadata and source code [148, 149]. A recent work [176], Repo2Vec, proposes an embedding approach to measure the similarity between repositories considering three types of information: (a) metadata, (b) source code, and (c) the repository directory structure. This is the approach that we use to compute similarity at the code level.

D. Studies of malware repositories on GitHub. For completeness, we provide a quick overview of studies that focus on malware on GitHub. Some earlier efforts [120,238] have manually collected a small number of malware repositories from GitHub for research purposes. More recently [175] developed a systematic method to classify repositories as malware or benign and identified the malware source code database that we are using here. Other studies [85, 86] have analyzed the hacker ecosystem in GitHub and other online security forums and identified the roles and dynamics among authors.

5.6 Conclusions

We present PIMan, a comprehensive approach to establish plausible influence among a set of repositories. Our approach combines three types of information: (a) repository level interaction, (b) author level interaction, and (c) temporal considerations. Once we determine the pair-wise influence score, we can create a network-wide influence interaction for the repositories.

The high-level contribution of our work can be summarized in the following points:

a. A step towards defining inter-repository social-level influence: With our definition of influence, we capture social interactions in a comprehensive way. Our initial results show that the influence is correlated with code-level similarity. Our plan is to thoroughly study the interplay between the influence defined in our study and the code-level similarity.

b. Flexible and powerful representation of influence using PIGraph: We develop the Plausible Influence Graph (PIGraph) in a systematic way, and propose different algorithmic and analysis techniques to extract useful information and insights.

c. Identifying lineage and families of influence: We showcase the capabilities of our approach by identifying interesting lineage relationships and repository groups rendering significant influence among them. The intention is to highlight the great potential for useful and insightful analysis that our approach can enable.

In the future, we plan to expand the work as follows. First, we will study the relationship between influence and code-level similarity. Second, we will expand our analysis to other types of software: (a) we will compare benign software development with malware software and (b) we will analyze in depth focused software branches, e.g. data mining software, android apps, etc. Third, we will closely study the malware ecosystem on GitHub as this could provide significant information in combating cyber-crime.

Finally, we intend to open-source our code and datasets to maximize the impact of our work and facilitate follow up research.

Chapter 6

Conclusions

Our thesis proposes and develops a systematic suit of methods to extract actionable information from online platforms. We develop robust tools to (a) mine important "events", (b) facilitate a hierarchical cluster extraction, and (c) model the dynamics of an ecosystem (here, malware authors). Our approaches have the following main advantages: (a) we develop complete tools for each of our methods which provide visual and intuitive information and can be operated even by a savvy users, (b) our tools can operate in unsupervised way without any apriori knowledge, (c) critical hierarchical patterns can also be discovered using one of our methods, and (d) we can track the hackers across platforms and understand their dynamics by profiling them.

Our study concludes in three key takeaway messages: (a) online platforms hide significant amount of security related important information which can be mined systematically and can be used even for early detection of critical events, (b) hackers leave online footprints, collaborate among themselves, brag about their hacking successes even in security forums and can be spotted using our tool, (c) malware development are on the rise targetting even macOS and iOS which are thought to be safer. Our initial findings are just the beginning of a promising future effort that can shed light on this online malware author ecosystem, which spans software repositories and security forums.

The current work thus can be seen as a building block that can enable new research directions. Follow-up efforts can use our approach to (a) detect emerging trends, (b) monitor malicious activity, (c) develop new capabilities on top of our methods, and (d) identify influential hackers towards safeguarding the Internet.

Bibliography

[1] 3vilp4wn. Hacking tool of 3vilp4wn. `https://github.com/3vilp4wn/CryptLog/`, 2013. [Online; accessed 08-February-2020].

[2] Aaron Holmes. 17 years old boy tried to hack twitter. `https://www.businessinsider.com/twitter-hacker-florida-teen-past-minecraft-bitcoin-scams-2020-8/`, August 2020.

[3] A. Abbasi, W. Li, V. Benjamin, S. Hu, and H. Chen. Descriptive analytics: Examining expert hackers in web forums. In *2014 IEEE Joint Intelligence and Security Informatics Conference*, pages 56–63, Sep. 2014.

[4] Sara Abdali, Neil Shah, and Evangelos E Papalexakis. Hijod: Semi-supervised multi-aspect detection of misinformation using hierarchical joint decomposition. *ECML-PKDD*, 2020.

[5] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471. IEEE, 2016.

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[7] Taher Alzahrani and Kathy J Horadam. Community detection in bipartite networks: Algorithms and case studies. In *Complex systems and networks*, pages 25–50. Springer, 2016.

[8] Wolfram Amme, Thomas S Heinze, and André Schäfer. You look so different: Finding structural clones and subclones in java source code. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 70–80. IEEE, 2021.

[9] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[10] John Aycock. *Computer viruses and malware*, volume 22. Springer Science & Business Media, 2006.

[11] Ali Sajedi Badashian and Eleni Stroulia. Measuring user influence in github: the million follower fallacy. In *2016 IEEE/ACM 3rd International Workshop on Crowd-Sourcing in Software Engineering (CSI-SE)*, pages 15–21. IEEE, 2016.

[12] Paul Baecher, Markus Koetter, Thorsten Holz, Maximillian Dornseif, and Felix Freiling. The nepenthes platform: An efficient approach to collect malware. In *International Workshop on Recent Advances in Intrusion Detection*, pages 165–184. Springer, 2006.

[13] Riyu Bana and Anuja Arora. Influence indexing of developers, repositories, technologies and programming languages on social coding community github. In *2018 Eleventh International Conference on Contemporary Computing (IC3)*, pages 1–6. IEEE, 2018.

[14] MohammadHossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Raimondas Kiveris, Silvio Lattanzi, and Vahab Mirrokni. Affinity clustering: Hierarchical clustering at scale. In *Advances in Neural Information Processing Systems*, pages 6864–6874, 2017.

[15] Joseph Bayer, Nicole Ellison, Sarita Schoenebeck, Erin Brady, and Emily B Falk. Facebook in context: Measuring emotional responses across time and space. *new media & society*, 20(3):1047–1067, 2018. SAGE.

[16] Andrew Begel, Jan Bosch, and Margaret-Anne Storey. Social networking meets software development: Perspectives from github, msdn, stack exchange, and topcoder. *IEEE Software*, 30(1):52–66, 2013.

[17] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson correlation coefficient. In *Noise reduction in speech processing*, pages 1–4. Springer, 2009.

[18] Austin R Benson, David F Gleich, and Jure Leskovec. Tensor spectral clustering for partitioning higher-order network structures. In *Proceedings of the 2015 SIAM International Conference on Data Mining*, pages 118–126. SIAM, 2015.

[19] Derya Birant and Alp Kut. St-dbscan: An algorithm for clustering spatial–temporal data. *Data & Knowledge Engineering*, 60(1):208–221, 2007.

[20] Christian Bird, Peter C Rigby, Earl T Barr, David J Hamilton, Daniel M German, and Prem Devanbu. The promises and perils of mining git. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 1–10. IEEE, 2009.

[21] Christopher M Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[22] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[23] Kelly Blincoe, Jyoti Sheoran, Sean Goggins, Eva Petakovic, and Daniela Damian. Understanding the popular users: Following, affiliation influence and leadership on github. *Information and Software Technology*, 70:30–39, 2016.

[24] Logsign Blog. Major ransomware events:. `https://blog.logsign.com/10-worst-ransomware-attacks-in-the-last-five-years/`, March 2018.

[25] Brian Krebs. Spyeye v. zeus rivalry ends in quiet merger. `https://krebsonsecurity.com/2010/10/spyeye-v-zeus-rivalry-ends-in-quiet-merger/`, 2010. [Online; accessed 14-March-2020].

[26] Alejandro Calleja, Juan Tapiador, and Juan Cabalero. The malsource dataset: Quantifying complexity and code reuse in malware development. *IEEE Transactions on Information Forensics and Security*, 14(12):3175–3190, 2018.

[27] Alejandro Calleja, Juan Tapiador, and Juan Caballero. A look into 30 years of malware development from a software metrics perspective. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 325–345. Springer, 2016.

[28] Andrea Capiluppi, Davide Di Ruscio, Juri Di Rocco, Phuong T Nguyen, and Nemitari Ajienka. Detecting java software similarities by using different clustering techniques. *Information and Software Technology*, 122:106279, 2020.

[29] Charlotte Carter. Romantic scamming in gaming forum. `https://www.stuff.co.nz/auckland/106254141/romantic-scammers-preying-on-players-of-online-game-words-with-friends/`.

[30] Ferhat Ozgur Catak and Ahmet Faruk Yazı. A benchmark api call dataset for windows pe malware classification. *arXiv preprint arXiv:1905.01999*, 2019.

[31] Dorota Celińska. Coding together in a social network: collaboration among github users. In *Proceedings of the 9th International Conference on Social Media and Society*, pages 31–40, 2018.

[32] Moses Charikar and Vaggos Chatziafratis. Approximate hierarchical clustering via sparsest cut and spreading metrics. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 841–854. SIAM, 2017.

[33] Fragkiskos Chatziasimidis and Ioannis Stamelos. Data collection and analysis of github repositories and users. In *2015 6th International Conference on Information, Intelligence, Systems and Applications (IISA)*, pages 1–6. IEEE, 2015.

[34] Gengbiao Chen, Zhengwei Qi, Shiqiu Huang, Kangqi Ni, Yudi Zheng, Walter Binder, and Haibing Guan. A refined decompiler to generate c code with high readability. *Software: Practice and Experience*, 43(11):1337–1358, 2013.

[35] Gengbiao Chen, Zhuo Wang, Ruoyu Zhang, Kan Zhou, Shiqiu Huang, Kangqi Ni, and Zhengwei Qi. A novel lightweight virtual machine based decompiler to generate c/c++ code with high readability. *School of Software, Shanghai Jiao Tong University, Shanghai, China*, 11, 2010.

[36] Jingnian Chen, Houkuan Huang, Shengfeng Tian, and Youli Qu. Feature selection for text classification with naïve bayes. *Expert Systems with Applications*, 36(3):5432–5435, 2009.

[37] Ning Chen, Steven CH Hoi, Shaohua Li, and Xiaokui Xiao. Simapp: A framework for detecting similar mobile applications by online kernel learning. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, pages 305–314, 2015.

[38] Dongdong Cheng, Sulan Zhang, and Jinlong Huang. Dense members of local cores-based density peaks clustering algorithm. *Knowledge-Based Systems*, page 105454, 2020.

[39] Chris Stobing. ios malwares in 2014. `https://www.digitaltrends.com/computing/decrypt-2014-biggest-year-malware-yet/`, 2015. [Online; accessed 08-February-2020].

[40] Ashley Claster. News of hacking by vandathegod:. `https://www.databreaches.net/dozens-of-government-websites-defaced-by-vandathegod-hacktivists/`, March 2019. [accessed March-2020].

[41] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical review E*, 70(6):6, 2004.

[42] Rhys Compton, Eibe Frank, Panos Patros, and Abigail Koay. Embedding java classes with code2vec: Improvements from variable obfuscation. *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020.

[43] Valerio Cosentino, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Findings from github: methods, datasets and limitations. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 137–141. IEEE, 2016.

[44] CR4SH. Hacking tool of cr4sh. `https://github.com/Cr4sh/s6_pcie_microblaze/`, 2017. [Online; accessed 08-February-2020].

[45] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on computer supported cooperative work*, pages 1277–1286. ACM, 2012.

[46] Anusha Damodaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H Austin, and Mark Stamp. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, 13(1):1–12, 2017.

[47] David Bisson. News of ransomware outbreak. `https://www.tripwire.com/state-of-security/security-data-protection/cyber-security/top-10-ransomware-strains-2016/`. [Online; accessed 08-February-2020].

[48] Ashok Deb, Kristina Lerman, Emilio Ferrara, Ashok Deb, Kristina Lerman, and Emilio Ferrara. Predicting Cyber-Events by Leveraging Hacker Sentiment. *Information*, 9(11):280, nov 2018.

[49] Tsuyoshi Deguchi, Katsuhide Takahashi, Hideki Takayasu, and Misako Takayasu. Hubs and authorities in the world trade network using a weighted hits algorithm. *PloS one*, 9(7), 2014.

[50] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[51] Bhuwan Dhingra, Zhong Zhou, Dylan Fitzpatrick, Michael Muehl, and William W Cohen. Tweet2vec: Character-based distributed representations for social media. *arXiv preprint arXiv:1605.03481*, 2016.

[52] Lukás Ďurfina, Jakub Křoustek, and Petr Zemek. Psybot malware: A step-by-step decompilation case study. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 449–456. IEEE, 2013.

[53] Lukáš Ďurfina, Jakub Křoustek, Petr Zemek, Dušan Kolář, Tomáš Hruška, Karel Masařík, and Alexander Meduna. Design of a retargetable decompiler for a static platform-independent malware analysis. In *International Conference on Information Security and Assurance*, pages 72–86. Springer, 2011.

[54] EH. Ethical hacker:. `https://www.ethicalhacker.net/`, March, 2020 2020. [accessed July-2020].

[55] Neil A Ernst, Steve Easterbrook, and John Mylopoulos. Code forking in open-source software: a requirements perspective. *arXiv preprint arXiv:1004.2889*, 2010.

[56] Nicolaas Klaas M Faber, Rasmus Bro, and Philip K Hopke. Recent developments in candecomp/parafac algorithms: a critical review. *Chemometrics and Intelligent Laboratory Systems*, 65(1):119–137, 2003.

[57] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. *ACM SIGCOMM computer communication review*, 29(4):251–262, 1999.

[58] Joseph L Fleiss and Jacob Cohen. The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability. *Educational and psychological measurement*, 33(3):613–619, 1973.

[59] Sri Shaila G, Ahmad Darki, Michalis Faloutsos, Nael Abu-Ghazaleh, and Manu Sridharan. Idapro for iot malware analysis? In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*, Santa Clara, CA, August 2019. USENIX Association.

[60] Mohammad Gharehyazie, Baishakhi Ray, and Vladimir Filkov. Some from here, some from there: Cross-project code reuse in github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 291–301. IEEE, 2017.

[61] Joobin Gharibshah, Evangelos E Papalexakis, and Michalis Faloutsos. RIPEx: Extracting malicious ip addresses from security forums using cross-forum learning. In *PAKDD*. Springer, 2018.

[62] Joobin Gharibshah, Evangelos E Papalexakis, and Michalis Faloutsos. Rest: A thread embedding approach for identifying and classifying user-specified information in security forums. *arXiv preprint arXiv:2001.02660*, 2020.

[63] Joobin Gharibshah, Evangelos E Papalexakis, and Michalis Faloutsos. Rest: A thread embedding approach for identifying and classifying user-specified information in security forums. In *Proceedings of the International AAAI Conference on Web and Social Media*, volume 14, pages 217–228, 2020.

[64] Joobin Gharibshah, Evangelos E Papalexakis, and Michalis Faloutsos. REST: A thread embedding approach for identifying and classifying user-specified information in security forums. *ICWSM*, 2020.

[65] GitHub. Repository search for public repositories: Showing 32,107,794 available repository results. `https://github.com/search?q=is:public/`. [Online; accessed 13-October-2019].

[66] GitHub. Search api v3. `https://help.github.com/en/github/searching-for-information-on-github/searching-for-repositories`. [Online; accessed 13-October-2019].

[67] GitHub. User search: Showing 34,149,146 available users. `https://github.com/search?q=type:user&type=Users/`. [Online; accessed 13-October-2019].

[68] Amir Globerson, Gal Chechik, Fernando Pereira, and Naftali Tishby. Euclidean embedding of co-occurrence data. *Journal of Machine Learning Research*, 8(Oct):2265–2295, 2007.

[69] Georgios Gousios and Diomidis Spinellis. Mining software engineering data from github. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 501–502. IEEE, 2017.

[70] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

[71] Romain Guigoures, Marc Boullé, and Fabrice Rossi. A triclustering approach for time evolving graphs. In *2012 IEEE 12th International Conference on Data Mining Workshops*, pages 115–122. IEEE, 2012.

[72] Ekta Gujral and Evangelos E Papalexakis. Smacd: Semi-supervised multi-aspect community detection. In *ICDM*, pages 702–710. SIAM, 2018.

[73] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

[74] HackerChatter. UCR hacker forum webtool for extracting useful information from security forums! `http://www.hackerchatter.org/`. [Online; accessed 22-July-2020].

[75] Mahmud Hasan, Mehmet A Orgun, and Rolf Schwitter. Real-time event detection from the twitter data stream using the twitternews+ framework. *Information Processing & Management*, 56:1146, 2019. Elsevier.

[76] Claudia Hauff and Georgios Gousios. Matching github developer profiles to job advertisements. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 362–366. IEEE, 2015.

[77] Richard Healey. Source code extraction via monitoring processing of obfuscated byte code, August 27 2019. US Patent 10,394,554.

[78] Brandon Heller, Eli Marschner, Evan Rosenfeld, and Jeffrey Heer. Visualizing collaboration and influence in the open-source software community. In *Proceedings of the 8th working conference on mining software repositories*, pages 223–226, 2011.

[79] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 518–529, 2020.

[80] Sameera Horawalavithana, Abhishek Bhattacharjee, Renhao Liu, Nazim Choudhury, Lawrence O Hall, and Adriana Iamnitchi. Mentions of security vulnerabilities on reddit, twitter and github. In *IEEE/WIC/ACM International Conference on Web Intelligence*, pages 200–207. ACM, 2019.

[81] James Howison and Kevin Crowston. The perils and pitfalls of mining sourceforge. In *MSR*, pages 7–11. IET, 2004.

[82] HTS. Hack this site:. `https://www.hackthissite.org/`, March, 2020 2020. [accessed July-2020].

[83] Yan Hu, Shanshan Wang, Yizhi Ren, and Kim-Kwang Raymond Choo. User influence analysis for github developer social networks. *Expert Systems with Applications*, 108:108–118, 2018.

[84] Yan Hu, Jun Zhang, Xiaomei Bai, Shuo Yu, and Zhuo Yang. Influence analysis of github repositories. *SpringerPlus*, 5(1):1–19, 2016.

[85] Risul Islam, Md Omar Faruk Rokon, Ahmad Darki, and Michalis Faloutsos. Hackerscope: The dynamics of a massive hacker online ecosystem. In *Proceedings of International Conference on Advances in Social Network Analysis and Mining (ASONAM)*. IEEE/ACM, 2020.

[86] Risul Islam, Md Omar Faruk Rokon, Ahmad Darki, and Michalis Faloutsos. Hackerscope: The dynamics of a massive hacker online ecosystem. *Social Network Analysis and Mining*, 11(1):1–12, 2021.

[87] Risul Islam, Md Omar Faruk Rokon, Evangelos E. Papalexakis, and Michalis Faloutsos. Tenfor: A tensor-based tool to extract interesting events from security forums. In *2020 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 515–522, 2020.

[88] Risul Islam, Md Omar Faruk Rokon, Evangelos E. Papalexakis, and Michalis Faloutsos. Recten: A recursive hierarchical low rank tensor factorization method to discover hierarchical patterns from multi-modal data. *Proceedings of the International AAAI Conference on Web and Social Media*, 15(1):230–241, May 2021.

[89] Risul Islam, Md Omar Faruk Rokon, Evangelos E. Papalexakis, and Michalis Faloutsos. Recten: A recursive hierarchical low rank tensor factorization method to discover hierarchical patterns in multi-modal data. In *Proceedings of the International AAAI Conference on Web and Social Media*, 2021.

[90] Risul Islam, Md Omar Faruk Rokon, Evangelos E. Papalexakis, and Michalis Faloutsos. Tenfor: Tool to mine interesting events from security forums leveraging tensor decomposition. *Lecture Notes in Social Networks*, 2021.

[91] Risul Islam, Ben Treves, Md Omar Faruk Rokon, and Michalis Faloutsos. Linkman: Hyperlink-driven misbehavior detection in online security forums. In *Proceedings of International Conference on Advances in Social Network Analysis and Mining (ASONAM)*. IEEE/ACM, 2021.

[92] Jack T. Youtube video of wifiphisher. `https://www.youtube.com/watch?v=pRtxFWJTS4k`. [Online; accessed 14-March-2020].

[93] Jack T. Youtube video of wifiphisher. `https://www.youtube.com/watch?v=tCwclyurB8I`. [Online; accessed 14-March-2020].

[94] Stefanie Jegelka, Suvrit Sra, and Arindam Banerjee. Approximation algorithms for tensor clustering. In *International Conference on Algorithmic Learning Theory*, pages 368–383. Springer, 2009.

[95] James A Jerkins. Motivating a market or regulatory solution to iot insecurity with the mirai botnet code. In *2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 1–5. IEEE, 2017.

[96] Jing Jiang, David Lo, Jiahuan He, Xin Xia, Pavneet Singh Kochhar, and Li Zhang. Why and how developers fork what from whom in github. *Empirical Software Engineering*, 22(1):547–578, 2017.

[97] Anjali Ganesh Jivani et al. A comparative study of stemming algorithms. *Int. J. Comp. Tech. Appl*, 2(6):1930–1938, 2011.

[98] Mainduddin Ahmad Jonas, Md Shohrab Hossain, Risul Islam, Husnu S Narman, and Mohammed Atiquzzaman. An intelligent system for preventing ssl stripping-based session hijacking attacks. In *MILCOM 2019-2019 IEEE Military Communications Conference (MILCOM)*, pages 1–6. IEEE, 2019.

[99] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101. ACM, 2014.

[100] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. An in-depth study of the promises and perils of mining github. *Empirical Software Engineering*, 21(5):2035–2071, 2016.

[101] Hong Jin Kang, Tegawendé F Bissyandé, and David Lo. Assessing the generalizability of code2vec token embeddings. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1–12. IEEE, 2019.

[102] Md Rezaul Karim, Oya Beyan, Achille Zappa, Ivan G Costa, Dietrich Rebholz-Schuhmann, Michael Cochez, and Stefan Decker. Deep learning-based clustering approaches for bioinformatics. *Briefings in Bioinformatics*, 2020.

[103] Shinji Kawaguchi, Pankaj K Garg, Makoto Matsushita, and Katsuro Inoue. Mudablue: An automatic categorization system for open source repositories. *Journal of Systems and Software*, 79(7):939–953, 2006.

[104] Yong-Deok Kim and Seungjin Choi. Nonnegative tucker decomposition. In *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2007.

[105] Jon M Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, 46(5):604–632, 1999.

[106] Trupti M Kodinariya and Prashant R Makwana. Review on determining number of cluster in k-means clustering. *International Journal*, 1(6):90–95, 2013.

[107] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *USENIX security symposium*, volume 4, pages 351–366, 2009.

[108] Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[109] Bence Kollanyi. Automation, algorithms, and politics: Where do bots come from? an analysis of bot codes shared on github. *International Journal of Communication*, 10:20, 2016.

[110] Satwik Kottur, Ramakrishna Vedantam, José MF Moura, and Devi Parikh. Visual word2vec (vis-w2v): Learning visually grounded word embeddings using abstract scenes. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4985–4994, 2016.

[111] Da Kuang and Haesun Park. Fast rank-2 nonnegative matrix factorization for hierarchical document clustering. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 739–747, 2013.

[112] Sanjeev Kumar, Rakesh Sehgal, and JS Bhatia. Hybrid honeypot framework for malware collection and analysis. In *2012 IEEE 7th International Conference on Industrial and Information Systems (ICIIS)*, pages 1–5. IEEE, 2012.

[113] Matt Kusner, Yu Sun, Nicholas Kolkin, and Kilian Weinberger. From word embeddings to document distances. In *International conference on machine learning*, pages 957–966, 2015.

[114] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196, 2014.

[115] Michael J Lee, Bruce Ferwerda, Junghong Choi, Jungpil Hahn, Jae Yun Moon, and Jinwoo Kim. Github developers use rockstars to overcome overflow of news. In *CHI'13 Extended Abstracts on Human Factors in Computing Systems*, pages 133–138. ACM, 2013.

[116] Roy Ka-Wei Lee and David Lo. Github and stack overflow: Analyzing developer interests across multiple social collaborative platforms. In *International Conference on Social Informatics*, pages 245–256. Springer, 2017.

[117] William Leibzon. Social network of software development at github. In *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 1374–1376. IEEE, 2016.

[118] Corrado Leita, Marc Dacier, and Frederic Massicotte. Automatic handling of protocol dependencies and reaction to 0-day attacks with scriptgen based honeypots. In *International Workshop on Recent Advances in Intrusion Detection*, pages 185–205. Springer, 2006.

[119] Corrado Leita, VH Pham, Olivier Thonnard, E Ramirez-Silva, Fabian Pouget, Engin Kirda, and Marc Dacier. The leurre. com project: collecting internet threats information using a worldwide distributed honeynet. In *2008 WOMBAT Workshop on Information Security Threats Data Collection and Sharing*, pages 40–57. IEEE, 2008.

[120] Toomas Lepik, Kaie Maennel, Margus Ernits, and Olaf Maennel. Art and automation of teaching malware reverse engineering. In *International Conference on Learning and Collaboration Technologies*, pages 461–472. Springer, 2018.

[121] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: an approach to modeling networks. *Journal of Machine Learning Research*, 11(2), 2010.

[122] Longzhuang Li, Yi Shang, and Wei Zhang. Improvement of hits-based algorithms on web documents. In *Proceedings of the 11th international conference on World Wide Web*, pages 527–535, 2002.

[123] Yitan Li, Linli Xu, Fei Tian, Liang Jiang, Xiaowei Zhong, and Enhong Chen. Word embedding revisited: A new representation learning and explicit matrix factorization perspective. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[124] Zhifang Liao, Haozhi Jin, Yifan Li, Benhong Zhao, Jinsong Wu, and Shengzong Liu. Devrank: Mining influential developers in github. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1–6. IEEE, 2017.

[125] Antonio Lima, Luca Rossi, and Mirco Musolesi. Coding together at scale: Github as a collaborative social network. In *Eighth International AAAI Conference on Weblogs and Social Media*, 2014.

[126] Yuxuan Liu, Guanghui Yan, Jianyun Ye, and Zongren Li. Community evolution based on tensor decomposition. In *ICPCSEE*, pages 62–75. Springer, 2019.

[127] Tuong Luu. Approach to evaluating clustering using classification labelled data. Master's thesis, University of Waterloo, 2011.

[128] Wanwangying Ma, Lin Chen, Yuming Zhou, and Baowen Xu. What are the dominant projects in the github python ecosystem? In *2016 Third International Conference on Trustworthy Systems and their Applications (TSA)*, pages 87–95. IEEE, 2016.

[129] M Madheswaran et al. An improved frequency based agglomerative clustering algorithm for detecting distinct clusters on two dimensional dataset. *Journal of Engineering and Technology Research*, 9(4):30–41, 2017.

[130] GS Mahalakshmi, G MuthuSelvi, and S Sendhilkumar. Gibbs sampled hierarchical dirichlet mixture model based approach for clustering scientific articles. In *Smart Computing Paradigms: New Progresses and Challenges*, pages 169–177. Springer, 2020.

[131] E. Marin, J. Shakarian, and P. Shakarian. Mining key-hackers on darkweb forums. In *2018 1st International Conference on Data Intelligence and Security (ICDIS)*, pages 73–80, April 2018.

[132] Ericsson Marin, Jana Shakarian, and Paulo Shakarian. Mining key-hackers on darkweb forums. In *ICDIS*, pages 73–80. IEEE, 2018.

[133] Collin McMillan, Mark Grechanik, and Denys Poshyvanyk. Detecting similar software applications. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 364–374. IEEE, 2012.

[134] Michael Frederick McTear, Zoraida Callejas, and David Griol. *The conversational interface*, volume 6. Springer, 2016.

[135] Sneha Mehta, Mohammad Raihanul Islam, Huzefa Rangwala, and Naren Ramakrishnan. Event detection using hierarchical multi-aspect attention. In *WWW*, pages 3079–3085, 2019.

[136] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[137] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[138] Audris Mockus. Large-scale code reuse in open source software. In *First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007)*, pages 7–7. IEEE, 2007.

[139] Audris Mockus, Diomidis Spinellis, Zoe Kotti, and Gabriel John Dusing. A complete set of related git repositories identified via community detection approaches based on shared commits. In *IEEE International Working Conference on Mining Software Repositories*, 2020.

[140] Kevin P Murphy. *Machine learning: a probabilistic perspective.* MIT press, 2012.

[141] Fionn Murtagh and Pedro Contreras. Algorithms for hierarchical clustering: an overview. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(1):86–97, 2012.

[142] Fionn Murtagh and Pierre Legendre. Ward's hierarchical agglomerative clustering method: which algorithms implement ward's criterion? *Journal of classification*, 31(3):274–295, 2014.

[143] n1nj4sec. Pupy tool. `https://github.com/n1nj4sec/pupy/wiki/`, 2015. [Online; accessed 08-February-2020].

[144] Annamalai Narayanan, Mahinthan Chandramohan, Lihui Chen, Yang Liu, and Santhoshkumar Saminathan. subgraph2vec: Learning distributed representations of rooted sub-graphs from large graphs. *arXiv preprint arXiv:1606.08928*, 2016.

[145] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. graph2vec: Learning distributed representations of graphs. *arXiv preprint arXiv:1707.05005*, 2017.

[146] Abdelmonim Naway and Yuancheng Li. Android malware detection using autoencoder. *arXiv preprint arXiv:1901.07315*, 2019.

[147] Hoang Nguyen, Xuan-Nam Bui, Quang-Hieu Tran, and Ngoc-Luan Mai. A new soft computing model for estimating and controlling blast-produced ground vibration based on hierarchical k-means clustering and cubist algorithms. *Applied Soft Computing*, 77:376–386, 2019.

[148] Phuong T Nguyen, Juri Di Rocco, Riccardo Rubei, and Davide Di Ruscio. Crosssim: exploiting mutual relationships to detect similar oss projects. In *2018 44th Euromicro conference on software engineering and advanced applications (SEAA)*, pages 388–395. IEEE, 2018.

[149] Phuong T Nguyen, Juri Di Rocco, Riccardo Rubei, and Davide Di Ruscio. An automated approach to assess the similarity of github repositories. *Software Quality Journal*, pages 1–37, 2020.

[150] Nicolas Verdier. Security researcher. `https://www.linkedin.com/in/nicolas-verdier-b23950b6/`. [Online; accessed 14-February-2020].

[151] Feiping Nie, Wei Zhu, and Xuelong Li. Unsupervised large graph embedding based on balanced and hierarchical k-means. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[152] Nikhil Gupta. Should we create a separate git repository of each project or should we keep multiple projects in a single git repo? `https://www.quora.com/Should-we-create-a-separate-git-repository-of-each-project-or-should-we-keep-multi answer/Nikhil-Gupta-55`, 2019. [Online; accessed 14-February-2020].

[153] Liqiang Niu, Xinyu Dai, Jianbing Zhang, and Jiajun Chen. Topic2vec: learning distributed representations of topics. In *2015 International conference on asian language processing (IALP)*, pages 193–196. IEEE, 2015.

[154] OffCom. Offensive community website. `http://offensivecommunity.net/`, March, 2020 2020. [accessed July-2020].

[155] Online Forums. Ethical hacker, hack this site, offensive community, wilders security, mpgh. `https://www.ethicalhacker.net/`, `https://www.hackthissite.org/`, `http://offensivecommunity.net/`, `https://www.wilderssecurity.com/`, `http://mpgh.net/`.

[156] Evangelos Papalexakis and A Seza Doğruöz. Understanding multilingual social networks in online immigrant communities. In *WWW*, page 865, 2015.

[157] Evangelos E Papalexakis. Automatic unsupervised tensor mining with quality assessment. In *SDM16*, pages 711–719. SIAM, 2016.

[158] Sergio Pastrana and Guillermo Suarez-Tangil. A first look at the crypto-mining malware ecosystem: A decade of unrestricted wealth. *arXiv preprint arXiv:1901.00846*, 2019.

[159] Sergio Pastrana, Daniel R Thomas, Alice Hutchings, and Richard Clayton. Crimebb: Enabling cybercrime research on underground forums at scale. In *WWW*, pages 1845–1854, 2018.

[160] Paul Roberts. Spyeye and zeus malware: Married or living separately? `https://threatpost.com/spyeye-and-zeus-malware-married-or-living-separately-101411/75755/`, 2011. [Online; accessed 14-March-2020].

[161] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[162] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

[163] Antoine Pietri, Guillaume Rousseau, and Stefano Zacchiroli. Forking without clicking: on how to identify software repository forks. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 277–287, 2020.

[164] Daniel Pletea, Bogdan Vasilescu, and Alexander Serebrenik. Security and emotion: sentiment analysis of security discussions on github. In *Proceedings of the 11th working conference on mining software repositories*, pages 348–351. ACM, 2014.

[165] Daniel Plohmann, Martin Clauss, Steffen Enders, and Elmar Padilla. Malpedia: a collaborative effort to inventorize the malware landscape. *Proceedings of the Botconf*, 2017.

[166] Rebecca S. Portnoff, Sadia Afroz, Greg Durrett, Jonathan K. Kummerfeld, Taylor Berg-Kirkpatrick, Damon McCoy, Kirill Levchenko, and Vern Paxson. Tools for Automated Analysis of Cybercriminal Markets. *Proceedings of the 26th International Conference on World Wide Web - WWW '17*, pages 657–666, 2017.

[167] Rebecca S Portnoff, Sadia Afroz, Greg Durrett, Jonathan K Kummerfeld, Taylor Berg-Kirkpatrick, Damon McCoy, Kirill Levchenko, and Vern Paxson. Tools for automated analysis of cybercriminal markets. In *WWW*, page 657, 2017.

[168] Anuja Priyam, GR Abhijeeta, Anju Rathee, and Saurabh Srivastava. Comparative analysis of decision tree classification algorithms. *International Journal of current engineering and technology*, 3(2):334–337, 2013.

[169] PyGithub. A python libraray to use github api v3. `https://github.com/PyGithub/PyGithub/`. [Online; accessed 13-October-2019].

[170] Austen Rainer and Stephen Gale. Evaluating the quality and quantity of data on open source software projects. In *Procs 1st int conf on open source software*, 2005.

[171] Raj Chandel. Article on pupy. `https://www.hackingarticles.in/command-control-tool-pupy/`, 2019. [Online; accessed 08-February-2020].

[172] William M Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical association*, 66(336):846–850, 1971.

[173] Leiming Ren, Shimin Shan, Xiujuan Xu, and Yu Liu. Starin: An approach to predict the popularity of github repository. In *International Conference of Pioneering Computer Scientists, Engineers and Educators*, pages 258–273. Springer, 2020.

[174] Monica Rogati and Yiming Yang. High-performing feature selection for text classification. In *Proceedings of the eleventh international conference on Information and knowledge management*, pages 659–661, 2002.

[175] Md Omar Faruk Rokon, Risul Islam, Ahmad Darki, Evangelos E Papalexakis, and Michalis Faloutsos. Sourcefinder: Finding malware source-code from publicly available repositories in github. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*, pages 149–163, 2020.

[176] Md Omar Faruk Rokon, Pei Yan, Risul Islam, and Michalis Faloutsos. Repo2vec: A comprehensive embedding approach for determining repository similarity. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021.

[177] Aurko Roy and Sebastian Pokutta. Hierarchical clustering via spreading metrics. *The Journal of Machine Learning Research*, 18(1):3077–3111, 2017.

[178] Hassen Saıdi, Phillip Porras, and Vinod Yegneswaran. Experiences in malware binary deobfuscation. *Virus Bulletin*, 2010.

[179] A. Sapienza, A. Bessi, S. Damodaran, P. Shakarian, K. Lerman, and E. Ferrara. Early warnings of cyber threats in online discussions. In *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 667–674, Nov 2017.

[180] Anna Sapienza, Alessandro Bessi, and Emilio Ferrara. Non-negative tensor factorization for human behavioral pattern mining in online games. *Information*, 9(3):66, 2018. Multidisciplinary Digital Publishing Institute.

[181] Anna Sapienza, Sindhu Kiranmai Ernala, Alessandro Bessi, Kristina Lerman, and Emilio Ferrara. Discover: Mining online chatter for emerging cyber threats. In *Companion Proceedings of the The Web Conference 2018*, pages 983–990. International World Wide Web Conferences Steering Committee, 2018.

[182] Tobias Schnabel, Igor Labutov, David Mimno, and Thorsten Joachims. Evaluation methods for unsupervised word embeddings. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, pages 298–307, 2015.

[183] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. Evolving exact decompilation. In *Workshop on Binary Analysis Research (BAR)*, 2018.

[184] Security Forums. Ethical hacker, hack this site, offensive community, wilders security. https://www.ethicalhacker.net/, https://www.hackthissite.org/, http://offensivecommunity.net/, https://www.wilderssecurity.com/.

[185] Neil Shah, Danai Koutra, Tianmin Zou, Brian Gallagher, and Christos Faloutsos. Timecrunch: Interpretable dynamic graph summarization. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1055–1064, 2015.

[186] Madhu K Shankarapani, Subbu Ramamoorthy, Ram S Movva, and Srinivas Mukkamala. Malware detection using assembly and api call sequences. *Journal in computer virology*, 7(2):107–119, 2011.

[187] Abhishek Sharma, Ferdian Thung, Pavneet Singh Kochhar, Agus Sulistya, and David Lo. Cataloging github repositories. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pages 314–319, 2017.

[188] Deepak Sharma, Bijendra Kumar, and Satish Chand. A survey on journey of topic modeling techniques from svd to deep learning. *IJMECS*, 9(7):50, 2017. Modern Education and Computer Science Press.

[189] Victor RL Shen, Chin-Shan Wei, and Tony Tong-Ying Juang. Javascript malware detection using a high-level fuzzy petri net. In *2018 International Conference on Machine Learning and Cybernetics (ICMLC)*, volume 2, pages 511–514. IEEE, 2018.

[190] Kevin Sheridan, Tejas G Puranik, Eugene Mangortey, Olivia J Pinon-Fischer, Michelle Kirby, and Dimitri N Mavris. An application of dbscan clustering for flight anomaly detection during the approach phase. In *AIAA Scitech 2020 Forum*, page 1851, 2020.

[191] Lei-Lei Shi, Lu Liu, Yan Wu, Liang Jiang, Muhammad Kazim, Haider Ali, and John Panneerselvam. Human-centric cyber social computing model for hot-event detection and propagation. *IEEE Transactions on CSS*, 6(5):1042–1050, 2019. IEEE.

[192] Grigori Sidorov, Alexander Gelbukh, Helena Gómez-Adorno, and David Pinto. Soft similarity and soft cosine measure: Similarity of features in vector space model. *Computación y Sistemas*, 18(3):491–504, 2014.

[193] Simone Catania. News of simplelocker outbreak. https://www.internetx.com/en/news-detailview/die-10-gefaehrlichsten-ransomware-varianten-der-letzten-jahre/. [Online; accessed 08-February-2020].

[194] Marcus Soll and Malte Vosgerau. Classifyhub: an algorithm to classify github repositories. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, pages 373–379. Springer, 2017.

[195] Sophron. Wifiphisher. https://github.com/wifiphisher/wifiphisher, 2014. [Online; accessed 14-March-2020].

[196] SourceFinder. Finding malware source-code from publicly available repositories in github. `http://hackerchatter.org/sourcefinder/`. [Online; accessed 12-January-2021].

[197] Charles Spearman. The proof and measurement of association between two things. *Appleton-Century-Crofts*, 1961.

[198] Diomidis Spinellis, Zoe Kotti, and Audris Mockus. A dataset for github repository deduplication: Extended description. In *IEEE International Working Conference on Mining Software Repositories*, 2020.

[199] Will Wei Sun and Lexin Li. Dynamic tensor clustering. *Journal of the American Statistical Association*, 114(528):1894–1907, 2019.

[200] Xiaoyan Sun, Yang Wang, Jie Ren, Yuefei Zhu, and Shengli Liu. Collecting internet malware based on client-side honeypot. In *2008 The 9th International Conference for Young Computer Scientists*, pages 1493–1498. IEEE, 2008.

[201] Dean Teffer, Ravi Srinivasan, and Joydeep Ghosh. Adahash: hashing-based scalable, adaptive hierarchical clustering of streaming data on mapreduce frameworks. *International Journal of Data Science and Analytics*, 8(3):257–267, 2019.

[202] Bart Theeten, Frederik Vandeputte, and Tom Van Cutsem. Import2vec: Learning embeddings for software libraries. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 18–28. IEEE, 2019.

[203] Sachin Thukral, Hardik Meisheri, Tushar Kataria, Aman Agarwal, Ishan Verma, Arnab Chatterjee, and Lipika Dey. Analyzing behavioral trends in community driven discussion platforms like reddit. In *ASONAM*, pages 662–669. IEEE, 2018.

[204] Ferdian Thung, Tegawende F Bissyande, David Lo, and Lingxiao Jiang. Network structure of social coding in github. In *2013 17th European conference on software maintenance and reengineering*, pages 323–326. IEEE, 2013.

[205] Ferdian Thung, David Lo, and Lingxiao Jiang. Detecting similar applications with collaborative tagging. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 600–603. IEEE, 2012.

[206] Ferdian Thung, David Lo, and Julia Lawall. Automated library recommendation. In *2013 20th Working conference on reverse engineering (WCRE)*, pages 182–191. IEEE, 2013.

[207] Kai Tian, Meghan Revelle, and Denys Poshyvanyk. Using latent dirichlet allocation for automatic categorization of software. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 163–166. IEEE, 2009.

[208] Kai Tian, Shuigeng Zhou, and Jihong Guan. Deepcluster: A general clustering framework based on deep learning. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 809–825. Springer, 2017.

[209] SL Ting, WH Ip, and Albert HC Tsang. Is naive bayes a good classifier for document classification. *International Journal of Software Engineering and Its Applications*, 5(3):37–46, 2011.

[210] Gülen Toker and O Kirmemis. Text categorization using k nearest neighbor classification. *Survey Paper, Middle East Technical University*, 2013.

[211] Tom K. Hacking news of fahim magsi. `https://www.namepros.com/threads/hacked-by-muslim-hackers.950924/`, 2016. [Online; accessed 08-February-2020].

[212] Tommy Hodgins. Choosing between "one project per repository" vs "multiple projects per repository" architecture. `https://hashnode.com/`, 2017. [Online; accessed 14-February-2020].

[213] Virus Total. Virustotal-free online virus, malware and url scanner. *Online: https://www. virustotal. com/en*, 2019.

[214] Christoph Treude, Larissa Leite, and Maurício Aniche. Unusual events in github repositories. *Journal of Systems and Software*, 142:237–247, 2018.

[215] VentureBeat. Github passes 100 million repositories. `https://venturebeat.com/2018/11/08/github-passes-100-million-repositories/`. [Online; accessed 13-October-2019].

[216] VirusBay. A web-based, collaboration platform for malware researcher. *Online: https://beta.virusbay.io/*, 2019.

[217] Hanna M Wallach. Topic modeling: beyond bag-of-words. In *Proceedings of the 23rd international conference on Machine learning*, pages 977–984. ACM, 2006.

[218] Chi Wang, Xueqing Liu, Yanglei Song, and Jiawei Han. Towards interactive construction of topical hierarchy: A recursive tensor decomposition approach. In *ACM SIGKDD*, pages 1225–1234, 2015.

[219] Joe H Ward Jr. Hierarchical grouping to optimize an objective function. *Journal of the American statistical association*, 58(301):236–244, 1963.

[220] Dawid Weiss. Quantitative analysis of open source projects on sourceforge. In *Proceedings of the First International Conference on Open Source Systems, Genova*, pages 140–147. Citeseer, 2005.

[221] Jianshu Weng, Ee-Peng Lim, Jing Jiang, and Qi He. Twitterrank: finding topic-sensitive influential twitterers. In *Proceedings of the third ACM international conference on Web search and data mining*, pages 261–270, 2010.

[222] Wikipedia. Linux based botnet bashlite. `https://en.wikipedia.org/wiki/BASHLITE/`. [Online; accessed 08-February-2020].

[223] WS. Wilders security website. `https://www.wilderssecurity.com/`, March, 2020 2020. [accessed July-2020].

[224] Joicymara Xavier, Autran Macedo, and Marcelo de Almeida Maia. Understanding the popularity of reporters and assignees in the github. In *SEKE*, 2014.

[225] Hangjun Xu. An algorithm for comparing similarity between two trees. *arXiv preprint arXiv:1508.03381*, 2015.

[226] Ji Xu, Guoyin Wang, and Weihui Deng. Denpehc: Density peak based efficient hierarchical clustering. *Information Sciences*, 373:200–218, 2016.

[227] Shuo Xu. Bayesian naïve bayes classifiers to text classification. *Journal of Information Science*, 44(1):48–59, 2018.

[228] Y. Nativ and S. Shalev. Github repository: thezoo. `https://github.com/ytisf/theZoo`, 2014. [Online; accessed 14-March-2020].

[229] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 158–177. IEEE, 2016.

[230] Yuval Nativ. Security researcher. `https://morirt.com/`. [Online; accessed 14-February-2020].

[231] Lenny Zeltser. Free malware sample sources for researchers. `https://zeltser.com/malware-sample-sources/`. [Online; accessed 13-October-2019].

[232] Jixin Zhang, Zheng Qin, Hui Yin, Lu Ou, and Yupeng Hu. Irmd: malware variant detection using opcode image recognition. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1175–1180. IEEE, 2016.

[233] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing*, 18(6):1245–1262, 1989.

[234] Yin Zhang, Rong Jin, and Zhi-Hua Zhou. Understanding bag-of-words model: a statistical framework. *International Journal of Machine Learning and Cybernetics*, 1(1-4):43–52, 2010.

[235] Yu Zhang, Frank F Xu, Sha Li, Yu Meng, Xuan Wang, Qi Li, and Jiawei Han. Higitclass: Keyword-driven hierarchical classification of github repositories. In *2019 IEEE International Conference on Data Mining (ICDM)*, pages 876–885. IEEE, 2019.

[236] Yun Zhang, David Lo, Pavneet Singh Kochhar, Xin Xia, Quanlai Li, and Jianling Sun. Detecting similar repositories on github. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 13–23. IEEE, 2017.

[237] Yong-liang Zhao and Quan Qian. Android malware identification through visual exploration of disassembly files. *International Journal of Network Security*, 20(6):1061–1073, 2018.

[238] Xingsi Zhong, Yu Fu, Lu Yu, Richard Brooks, and G Kumar Venayagamoorthy. Stealthy malware traffic-not as innocent as it looks. In *2015 10th International Conference on Malicious and Unwanted Software*, pages 110–116. IEEE, 2015.

[239] Jianwei Zhuge, Thorsten Holz, Xinhui Han, Chengyu Song, and Wei Zou. Collecting autonomous spreading malware using high-interaction honeypots. In *International Conference on Information and Communications Security*, pages 438–451. Springer, 2007.