Automated Detection of Password Leakage from Public GitHub Repositories

Runhan Feng, Ziyang Yan, Shiyan Peng, Yuanyuan Zhang
Shanghai Jiao Tong University
Shanghai, China
{fengrunhan,yzy123,peng-shiyan,yyjess}@sjtu.edu.cn

ABSTRACT

The prosperity of the GitHub community has raised new concerns about data security in public repositories. Practitioners who manage authentication secrets such as textual passwords and API keys in the source code may accidentally leave these texts in the public repositories, resulting in secret leakage. If such leakage in the source code can be automatically detected in time, potential damage would be avoided. With existing approaches focusing on detecting secrets with distinctive formats (e.g., API keys, cryptographic keys in PEM format), textual passwords, which are ubiquitously used for authentication, fall through the crack. Given that textual passwords could be virtually any strings, a naive detection scheme based on regular expression performs poorly. This paper presents PassFinder, an automated approach to effectively detecting password leakage from public repositories that involve various programming languages on a large scale. PassFinder utilizes deep neural networks to unveil the intrinsic characteristics of textual passwords and understand the semantics of the code snippets that use textual passwords for authentication, i.e., the contextual information of the passwords in the source code. Using this new technique, we performed the first large-scale and longitudinal analysis of password leakage on GitHub. We inspected newly uploaded public code files on GitHub for 75 days and found that password leakage is pervasive, affecting over sixty thousand repositories. Our work contributes to a better understanding of password leakage on GitHub, and we believe our technique could promote the security of the open-source ecosystem.

CCS CONCEPTS

• **Security and privacy → Security services**.

KEYWORDS

Password, Mining Software Repositories, Deep Learning, GitHub

ACM Reference Format:
Runhan Feng, Ziyang Yan, Shiyan Peng, Yuanyuan Zhang. 2022. Automated Detection of Password Leakage from Public GitHub Repositories. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510150

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510150

1 INTRODUCTION

GitHub has received much attention since its establishment in 2007 [9]. More and more developers choose to create and host their repositories on GitHub because of its attractive features like free code hosting and collaborative development. Especially in the recent two years, the GitHub community has proliferated. As of September 2020, there are 56 million registered developers on GitHub. In 2020 only, 65 million new repositories were created [4]. With so many repositories hosted publicly on GitHub, security and privacy issues come naturally. These repositories may contain important sensitive authentication information such as textual passwords, API keys, private cryptography keys, etc. Once such sensitive information is leaked, it will bring serious security threats to the owners.

Due to some developers' lack of security consciousness, it is not uncommon for such authentication information to be exposed to the public as part of GitHub repositories [2] [3]. Among all these authentication information, textual passwords are of great value. Although been blamed for security pitfalls, textual passwords have been a mainstream authentication method for decades because of their low deployment cost and simplicity. Especially in development, developers tend to use textual passwords to authenticate themselves to remote services. If such textual passwords are used for authenticating to third-party services, once they are leaked, all the data protected by the passwords would be compromised. For example, a leaked Gmail password can be used to log into many Google services, such as Gmail, Google Driver, etc. It can also be used to send spam and phishing e-mails spoofed from the owner. Besides, the disclosure places the owners' all accounts in jeopardy since security-insensitive developers tend to reuse their passwords across different services [35]. Worse still, if the textual passwords are for enterprise use, i.e., they are hard-coded in source code to access or manage the network, database, or other system infrastructure, such leakage could lead to severe security breach [6] [7]. Despite the potential threat, it remains unknown to what extent passwords are leaked on GitHub and whether the service provider can effectively detect and protect leaked passwords, or in other words, whether attackers can efficiently harvest them.

Both academia and industry have made great efforts to understand and mitigate the leakage of various credentials on GitHub. Sinha et al. [66] investigated Java files in 84 repositories to identify AWS keys using regular expressions and lightweight static analysis. Meli et al. [49] performed a large-scale analysis on private key files and API Keys with distinctive formats to characterize the leakage of the secrets on GitHub. Since 2019, GitHub introduced its secret scanning function, which scans public repositories to prevent fraudulent use of secrets that are committed accidentally [10]. So far, secret scanning supports a set of credentials, including Alibaba Cloud

Access Key, Azure Access Token, AWS Access Key, and private keys in PEM format, etc. Besides, many open source tools [15] [17] [18] [20] [25] were developed to help prevent developers from pushing their sensitive authentication information to public repositories. However, existing work mainly focuses on structured secrets like API keys, which could be easily detected with well-crafted regular expressions. Relatively little attention has been paid to textual passwords. Given the serious consequences that password leakage may bring, an in-depth study to understand the scope and magnitude of textual password leakage on public GitHub repositories at a large scale is necessary. Therefore, automated mining techniques towards detecting passwords from GitHub repositories containing code files written in multiple programming languages are badly needed. They could help better understand and promote the security of the open-source ecosystem. However, it is quite challenging to achieve due to the following barriers.

Heterogeneity of passwords. Unlike structured secrets like API keys, etc., textual passwords could be virtually any strings and do not conform to the distinct structure, thus making them hard to detect with high accuracy. Existing techniques mainly rely on well-crafted regular expressions [17] [61] [62] or entropy-based heuristics [20]. Specifically, regular-expression-based password detection techniques take keywords such as "password", "pwd", which are usually used altogether with the real passwords, as a significant basis for identification, lacking the ability to model the passwords themselves. Entropy-based techniques calculate the Shannon entropy of each string, implying a hypothesis that passwords have higher entropy than ordinary strings. However, in practice, many passwords are chosen by humans but not randomly generated by machines, thus violating the assumption [58]. In general, these techniques perform poorly since they cannot properly deal with the heterogeneity of passwords, which hinders them from being adopted in practice.

Multiformity of source code. GitHub repositories consist of source code files written in different programming languages [5], which follow diverse grammar and styles. Such multiformity also diminishes the effectiveness of regular-expression-based technology. Some work leverages specific static analyzers to bypass the impact of different grammar and styles. Specifically, SLIC considers only Puppet, an infrastructure as code (IaC) script language, and is built based on a dedicated parser [61]. CredMiner focuses on Android smali code [11], using backward slicing and forward simulation execution engine to reconstruct credentials used as parameters for predefined sink APIs in application code [74]. However, this work relies heavily on the sophisticated static analysis tools designed for specific programming languages (which are not available for unpopular programming languages), thus lacking the generalization ability to analyze files written in different languages. To deal with the multiformity problem, a programming language-agnostic technique is needed. Besides, such static-analysis-based methods usually have to statically analyze each file in the codebase to construct interprocedural dataflow, lacking scalability to ultra-large-scale codebases on GitHub [60].

This paper presents PassFinder, an automated approach to detecting passwords in code files written in different programming languages. Considering the two barriers mentioned above, we decompose the password detection problem from two aspects. On the one hand, for the heterogeneity of passwords, we construct the Password Model to extract the intrinsic characteristics of textual passwords. Specifically, we divide textual passwords into two main categories, i.e., human-chosen passwords and machine-generated random passwords. PassFinder models the distribution of human-chosen passwords corpus, random passwords corpus, and ordinary strings corpus with a deep neural network, which further guides predicting strings being passwords or not. On the other hand, considering the multiformity of source code, the critical insight is that no matter how the programming language and style changes, the context of each password, i.e., the surrounding code snippets share similar semantics and structures, thus could be leveraged to determine whether the usage scenario is authentication-related or not. Specifically, we train the Context Model to classify the code snippets based on the semantics of program elements (methods, variables, constants, etc.). Figure 1 shows a sample code snippet with the password "joe****" leaked on line 8. Typically, when authenticating with passwords, developers have to provide extra information such as identity, remote host, connection configuration, etc., which constitute the context of the passwords. Combining intrinsic characteristics and context information, PassFinder can finally identify passwords in source code files. Intuitively, our design reflects how a human identifies passwords in source code, i.e., by understanding individual strings and reasoning about the meaning of the context. Essentially, compared with existing regular-expression-based approaches [17] [20] [24], PassFinder expands and makes better use of the information sources for detection in a data-driven way.

```
1  ....
2  HtmlEmail email = new HtmlEmail();
3  email.setHostName("smtp.gmail.com");
4  email.setCharset("utf8");
5  email.setSSL(true);
6  email.setSocketConnectionTimeout(30);
7  email.setAuthentication("fromaddr@gmail.com",
8                          "joe****");
9  email.setSubject("Example subject");
10 email.setMsg("Example message");
11 ....
```

Figure 1: Code example based on real leaks

We evaluate the approach on our manually labeled ground truth dataset. The experimental results show that PassFinder accurately discovers leaked passwords in source code files (with a precision of 81.54% and a recall of 80.51%), significantly outperforming existing regular-expression-based approaches. Armed with PassFinder, we can harvest passwords from public GitHub repositories on a large scale, gaining new insights into the current situation of password leakage. This paper presents the first comprehensive, longitudinal analysis of password leakage on GitHub. We examine 1,476,692 files representing 539,012 repositories during 75 days, from which we identify 142,479 passwords from 64,045 (11.88%) repositories. Our results indicate that numerous developers on GitHub are faced with a massive threat since data in public GitHub repositories can be accessed by anyone, including malicious attackers who use GitHub as a source for collecting Open Source Intelligence (OSINT). Worse still, we investigate the lifecycle of leaked passwords and find that

over 82.32% of the passwords remain available in the repositories, not being removed within 16 days of being detected, which leaves a long window for attacking.

In summary, our work makes the following contributions:

- *New technique for password detection.* We design and implement a novel technique for automatically detecting passwords in source code. Our approach leverages both intrinsic characteristics of textual passwords and semantic information of program elements to accurately identify the hard-coded passwords from code files written in different programming languages.
- *Large-scale password leakage analysis.* Using our new technique, we investigate the password leakage in public GitHub repositories for 75 days. This is the first large-scale systematic study that measures password leakage on GitHub to the best of our knowledge. We find that thousands of passwords are leaked daily and that the majority of leaked passwords remain available for weeks or longer.

We organize the reminder of this paper as follows: we introduce related work of our research in Section 2. We describe the design of PassFinder in Section 3. We present the implementation and evaluation of PassFinder in Section 4. We report our large-scale password leakage analysis in Section 5, followed by a discussion in Section 6. We conclude our paper in Section 7.

2 RELATED WORK

This section briefly reviews prior research on password security and machine learning on code.

2.1 Password Security

Textual passwords have been ubiquitously used for authentication between end-users and computer systems since the 1960s. To make textual passwords more secure in practical use, many password guidelines are proposed by academia and industry [14] [53]. These guidelines suggest that passwords should be at least eight characters in length, should not contain common and easily-guessed words, should contain multiple character types, and ideally be randomly chosen. Despite good-faith efforts in the security of their accounts and data, users struggle to comply with password creation and management guidelines [45].

Ever since the seminal work by Morris et al. on password cracking [51], researchers have sought to understand how users create their passwords, i.e., the intrinsic characteristic of passwords, which could further guide password-cracking attacks [41] [58]. Matt et al. created a probabilistic context-free grammar (PCFG) based upon disclosed passwords and generate word-mangling rules automatically, which could be used to create password guesses [69] [70]. Wang et al. performed an empirical study on Chinese passwords and improved the PCFG-based algorithm to more accurately capture passwords that are of a monotonically long structure [68]. Ma et al. developed a novel method using a 6-gram Markov model with additive smoothing for modeling English-language passwords [48]. Conceptually, Markov models predict the probability of the next character in a password based on the previous characters or context characters.

In recent years, researchers have proved that deep neural network models have a good performance for modeling the characteristics of passwords [55] [56]. Melicher et al. constructed a recurrent neural network that outputs the number of guesses likely needed to guess a given password [50]. Hitaj et al. used deep generative adversarial networks (GAN) to learn the distribution of passwords to generate high-quality password guesses [40]. PassFinder also learns the intrinsic characteristics of passwords with a deep neural network model, but different from existing work, which focuses on password guessing (thus using generative models), we model the password detection problem as a classification task thus using a discriminative model.

2.2 Learning on "Big Code"

Learning on "Big Code" is a new trend, since the "naturalness" of software [39] could be used to assist software engineering and security tasks, including identifying code with vulnerability [47], deobfuscation [31] [67] and code summarization [27] [29] [30] [43], among many others [26]. These approaches analyze large amounts of source code, ranging from hundreds to thousands of software projects, building machine learning models of source code properties inspired by techniques from natural language processing (NLP). It has been proved that NLP models have the ability to learn both semantics and properties of source code snippets [63] [65]. Nan et al. utilized NLP techniques to automatically locate the program elements (variables, methods, etc.) of interest in Android APKs, and then perform a learning-based program structure analysis to accurately identify those carrying sensitive content to detect privacy leakage on Android platform [52]. Yu et al. used deep neural network models to extract the semantic information of the binary code for similarity detection [71]. Further, they used a deep pyramid convolutional neural network (DPCNN) and graph neural network (GNN) for modeling source code and binary code respectively, and achieved binary source code matching in function level [72]. Existing work focuses mainly on one programming language, most of which were built on a specific language's Abstract Syntax Tree (AST), thus lacking the generalization to multiple languages. In contrast, the model used by the PassFinder applies to different programming languages.

3 PASSFINDER DESIGN

Detecting hard-coded passwords in source code is by no means trivial. Existing heuristic regular-expression-based approaches [17] [20] [24] perform poorly on both false positives and false negatives, especially when it comes to multi-million source code files written in different programming languages, following diverse programming styles. In this section, we introduce the design of PassFinder, which addresses the challenges of processing different languages and precisely detects passwords in source code.

3.1 Design Overview

We model password detection in source code as a discriminative problem. Specifically, PassFinder inspects each hard-coded string to classify it as an ordinary string or a password, leveraging both the intrinsic characteristics of the string sequence and the semantics of the surrounding code snippets, i.e., the context of the string.

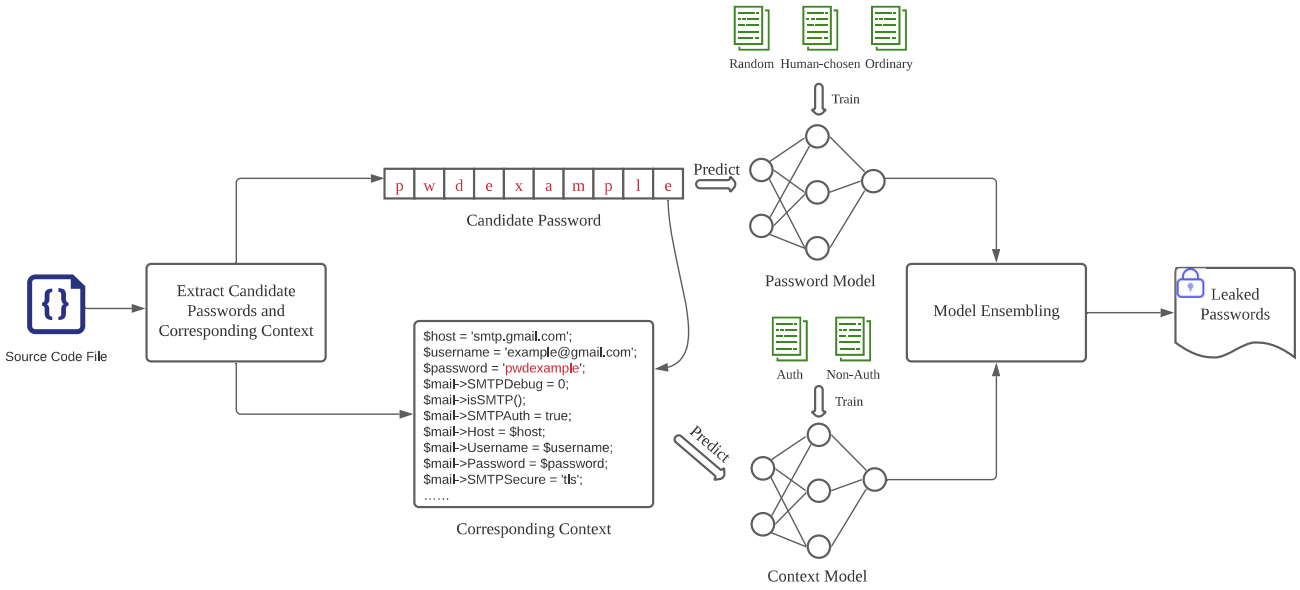

Figure 2: Design of PassFinder

Figure 2 shows a high-level overview of this decomposition. To empower PassFinder with the ability to distinguish between passwords and ordinary strings, we train two deep learning models, i.e., the Password Model and the Context Model, which will be detailed in 3.3 and 3.4 respectively. Intuitively, this decomposition reflects how a human identifies a password, i.e., by understanding individual sequence and reasoning about the meaning of multiple lines. Combining the output of both the models, PassFinder can finally accurately identify hard-coded textual passwords in source code. The motivation for choosing a learning-based approach over, e.g., regular-expression-based heuristics or static dataflow analysis is three-fold. First, deep learning methods have been proved to be capable of performing automatic feature extraction from raw data. Second, learning from data retains the possibility of generalizing the approach to multiple languages. Third, the models can continuously evolve by re-training them with more recent data.

3.2 Extracting Candidate Passwords

Given a source code file, PassFinder firstly preprocesses the whole file by removing non-ASCII characters. As for low-frequency tokens that cannot contribute to the result of classification, i.e., URL and e-mail address, we replace them with specific tokens $\langle URL \rangle$ and $\langle EMAIL \rangle$ respectively. Typically, passwords exist in source code in the form of hard-coded strings, usually wrapped in quotes to distinguish them from other program elements. Therefore, we can easily extract hard-coded strings precisely with well-crafted regular expressions. Note that we focus on strings with the length of 6 to 30, since passwords within this length interval account for the vast majority in previously leaked password datasets [1] [68]. The exact proportion is 99.95% in LinkedIn [1] dataset, which will be used for our Password Model. Besides, PassFinder filters out strings that are in the common passwords dictionary [13] because concerning these passwords, even human beings who have inspected the context cannot distinguish whether it is an unintentional leakage or the developers deliberately using these common passwords to replace their own passwords. The only way to verify the validity of these passwords is to manually use them for authentication, which is not allowed in our experiment for ethics problems. The remaining passwords are considered candidate passwords, which will be further inspected by PassFinder. As for each candidate password, we also extract the code snippets before and after the line where it is located as the context.

3.3 Modeling the Intrinsic Characteristics

As for a set of strings hard-coded in source code files, a human observer can tell which strings are more likely to be passwords because the corpus of passwords (either human-chosen or machine-generated) has a different distribution against ordinary strings used in source code. Intuitively, PassFinder imitates this ability of human beings. However, such characteristic (e.g., character frequency and order, etc.) is hard to model with specific rules. Therefore, we use a deep learning model, i.e., the Password Model, to automatically extract the features and further guide prediction. Different from existing work that models only the human-chosen passwords to construct a generative model that outputs high-quality password guesses, in this work, we model password detection as a three-classification problem. Specifically, strings in source code are classified by the Password Model into human-chosen passwords, machine-generated random passwords, and ordinary strings since they follow different distributions.

To train the Password Model, three types of data are needed. For human-chosen passwords, we use the previously leaked LinkedIn[1] dataset to model. However, after manually scrutinizing the LinkedIn dataset, we find that although the LinkedIn dataset contains almost

exclusively human-chosen passwords, it still contains a small portion of machine-generated random passwords. We have to filter out these random passwords. However, it is a non-trivial task since there is no guaranteed mathematical test for randomness. Similar to previous work [49], we use Shannon Entropy as an estimator for randomness. Intuitively, random passwords should have higher entropy, therefore deviating significantly from the dataset's average entropy. Specifically, for each password in the LinkedIn dataset, we calculate its entropy and eliminate it if its entropy is more than 3 standard deviations from the mean of all passwords, classifying it as an outlier. Then after deduplication, we get a human-chosen passwords dataset with over 58 million passwords. As for machine-generated random passwords, we choose the union of the character sets adopted by popular password managers [8] [22] and use secure pseudorandom number generator (PRNG) to randomly generate 25 million strings with a length between 6 to 30, one million for each length. Concerning the ordinary strings in source code, we used hard-coded strings from popular projects on GitHub. Specifically, for the ten most popular programming languages used on GitHub [5], we crawled 3,000 repositories with the most stars for each and got a total of 30,000 repositories. We recursively traversal each repository and extract hard-coded strings inside. With all these hard-coded strings, we can model the distribution of ordinary strings used in practice. However, after manually scrutinizing these popular repositories, we find that there exist some hard-coded passwords used for testing or demonstration purposes. Based on the observation that these popular repositories usually have an appropriate directory configuration, we filter out files that contain "test" or "example" in their full path to avoid mixing these dummy passwords into the ordinary strings dataset. We also filter out strings already in the human-chosen passwords dataset to avoid overlapping between datasets. In total, we extract over 16 million unique ordinary strings with a length between 6 to 30. Finally, we have a training dataset that consists of 58 million human-chosen passwords, 25 million machine-generated passwords, and 16 million ordinary strings. Although the dataset is not balanced between different categories, our Password Model learns all classes reasonably well (see Section 4.2 for more details).

As for each string in the training dataset, we tokenize it into characters as the basic terms and encode them with "one-hot" encoding. We train our discriminative model with a Text Convolution Neural Network (TextCNN) [73], which is nine layers deep with six convolutional layers and three fully connected layers. We choose TextCNN since it has been proven to be good at extracting local features, which is suitable for human-chosen passwords, which typically consist of sub-tokens. Finally, our Password Model will give the probability of strings being human-chosen passwords, machine-generated passwords, or ordinary strings. Directly using the label with the highest probability as the output is acceptable. However, as for our Password Model, we focus more on precision to alleviate false positives. Therefore, we fine-tune the threshold for predicting ordinary strings. Finally, we set the threshold as 0.2, i.e., any strings with the probability over 0.2 to be an ordinary string fall into this category. Otherwise, same as normal, we choose the label with the highest probability as the output. In this way, our Password Model achieves a higher precision, introducing only a small impact on the model's overall performance. Both those classified as human-chosen and machine-generated passwords are taken as positive results, i.e., textual passwords.

3.4 Modeling the Context Semantics

Besides the intrinsic characteristics, PassFinder also inspects the code snippet surrounding each candidate password, i.e., the context, to gain more insight about the usage scenario of the candidate password and gives a prediction that whether a code snippet is used for authentication or not. This can be taken as an essential criterion for judging whether the string is a password. Since there is no available open dataset on the code snippets related to authentication, we have to establish a dataset on our own for model training.

For clarity, we focus on the ten most popular programming languages on GitHub, which account for over 88% in all languages [5]. These ten languages include JavaScript, Python, Java, Golang, C++, TypeScript, Ruby, PHP, C#, and C. However, manually inspecting all newly uploaded files on GitHub to pinpoint code snippets related to authentication is inefficient since code for authentication use occupies only a tiny part of all files. To improve the effectiveness of collected files and labeling efficiency, we filter out non-related files with a heuristic method. The filtering strategy will be detailed in Section 5.1. We continuously collected candidate files and reviewed all content inside to pinpoint authentication-related code snippets for three weeks. In the labeling process, we consider only whether the code snippets are used for authentication, even if no actual password leakage occurs. To guarantee the labeling results' correctness, we built an inspection team, which consisted of two Ph.D. students with four master students. All of them have done intensive research work with software development. We divided the team into two groups. Each group consisted of a leader and two members. The leaders reviewed the labeling results from the members. We only accepted and included code snippets to our dataset when the code snippets received full agreement among the groups. When a code snippet received different labeling results, we hosted a discussion with all six people to decide through voting. In total, we collected 6,000 code snippets for authentication, 600 for each language. We also selected 600 code snippets for each language that are not related to authentication as negative samples. The labeling procedure took 1,200 person-hours.

With the labeled data, we can train the Context Model. To generalize our Context Model to files written by different languages, we did not laboriously parse the code into corresponding ASTs, which is prevalently used as the input of the deep neural network model in previous work that learns from code [28] [29] [32] [64]. Instead, we use the raw source code directly as input, relying on the neural network to extract semantics. Like our Password Model, we use the TextCNN model on character-level to avoid the out-of-vocabulary (OOV) phenomena [44], which might weaken the robustness of models. We do not use the Recurrent Neural Network (RNN) models since it is harder for RNN models to model the dependence in character level, and training RNN models requires more computing resources [34] [42]. As for the context width, i.e., the number of lines of code snippets used as context, we set it as 6. In other words, PassFinder extracts code snippets 6 lines before and after the candidate passwords for inspecting. We made a comparison between

the context width ranging from 3 to 7 in our experiment, and the context width of 6 achieves the best performance.

3.5 Ensembling Models

To combine the results of our Password Model and Context Model, we used model ensembling [36] technology. Specifically, we use the majority voting method [54], i.e., only when the Password Model and the Context Model agree will PassFinder treat it as a textual password. Specifically, as for each candidate password, only if the Password Model predicts it as a human-chosen or machine-generated password and the Context Model classifies its context as authentication-related will it be identified as a textual password. This is a relatively conservative strategy, but it could effectively reduce the false positives. Besides, it is relatively easier to deploy since it does not require extra training data.

4 EVALUATION OF PASSFINDER

We evaluate PassFinder on different settings and addresses the following three research questions:

- RQ1: How effective are the Password Model and the Context Model respectively?
- RQ2: How effective is PassFinder at detecting passwords?
- RQ3: How effective is the Context Model in cross-language validation?

4.1 Experiment Setup

We implement PassFinder in Python. The deep learning models are implemented with PyTorch [57] and Scikit-Learn [59]. The experiment environment is a 32-core server equipped with a Tesla V100 GPU, 64GB RAM, running on Ubuntu OS with a Linux 5.4.0 kernel. We use grid search [46] as the hyper-parameters selection method to obtain the best performance. The Password and Context model are trained for 16 and 32 epochs respectively, using the Adam optimizer with a learning rate of 0.0001.

4.2 RQ1: Effectiveness of the Models

In our evaluation of the Password Model, we divide the dataset collected in Section 3.3 into training set, test set and validation set at a ratio of 8:1:1. We fine-tune the hyper-parameters on the test set and finally evaluate on the validation set. As is shown in Table 1, on average, our Password Model achieves a precision of 96.35%, a recall of 97.29%, and an F1-score of 96.79%. Figure 3 shows a confusion matrix of the Password Model normalized by row. As we can see, the model achieves excellent performance in all three categories.

Table 1: Performance of the Password Model

	Precision	Recall	F1-score
Human-chosen	98.54%	96.86%	97.69%
Random	99.21%	98.92%	99.06%
Ordinary	91.29%	96.08%	93.62%
Marco-Average	96.35%	97.29%	96.79%

As for the evaluation of the Context Model, we train and test the model on the dataset collected in Section 3.4 with 10-fold validation.

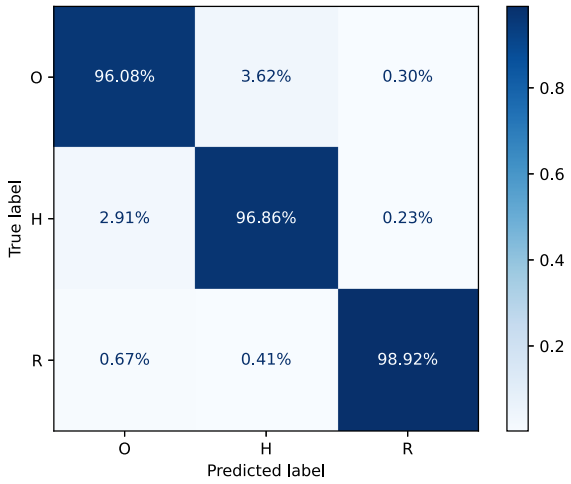

Figure 3: Normalized confusion matrix of the Password Model. "O" denotes ordinary strings, "H" denotes Human-chosen passwords and "R" denotes random passwords.

For comparison, we examine the performance of four widely used text classification approaches, including Naive Bayes (NB), Logistic Regression (LR), K-Nearest-Neighbors (KNN), and Support Vector Machine (SVM). For these four approaches, we consider both camel-case and underscore-case naming conventions and split each code snippet into word-level tokens with NLTK [33]. To alleviate the influence of word morphology, we also perform lemmatization and lowercasing. After preprocessing, we extract the Term Frequency and Inverse Document Frequent (TF-IDF) as feature vectors for each code snippet. We train and fine-tune hyper-parameters by a grid search to achieve their best performances. We assess the classification performance of these four approaches using the same dataset for our Context Model. The detailed precision, recall, and F1-score is shown in Figure 4. For the context classification task, our context model achieves a precision of 95.49%, a recall of 94.95%, and an F1 score of 95.22%. Overall, our Context Model performances best among all classification approaches.

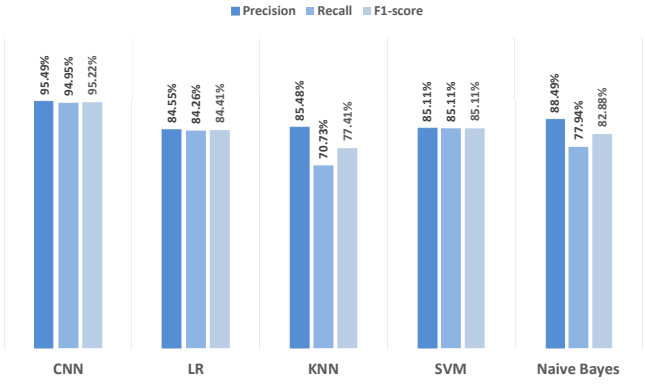

Figure 4: Performances of different classification methods

Table 2: Detailed regular expressions from detect-secrets, which are used for comparison in our experiment. denylist = [db_pass,password,passwd,secrets], closing = []\'"]{0,2}, whitespace = \s*?, quote = [\'"], secret = [^\s]+, nonWhitespace = [^\s]*, square_brackets = (\[\])

Target Regular Expression	Example
({denylist})({closing})?{whitespace}:=?{whitespace}({quote}?)({secret})(\3)	password:="foobar"
({denylist})({closing})?:{whitespace}({quote}?)({secret})(\3)	password: foobar
({denylist})({closing})?:({whitespace})({quote})({secret})(\4)	password: "foobar"
({denylist})({square_brackets})?{optional_whitespace}={optional_whitespace}(@)?(")({secret})(\5)	password[] = "foobar"
({denylist})({closing})?{whitespace}={whitespace}({quote}?)({secret})(\3)	password = foobar
({denylist})({closing})?{whitespace}={whitespace}({quote})({secret})(\3)	password = "foobar"
({denylist}){nonWhitespace}{whitespace}({quote})({secret})(\2);	password "foobar";

4.3 RQ2: End-to-End Effectiveness

End-to-End Ground Truth. To measure how effective PassFinder is at detecting passwords in source code, we built an end-to-end ground truth. Specifically, we randomly selected 5,000 files from our candidate files (The collection of candidate files will be detailed in Section 5). We used the same setting as the labeling process for the training dataset of our Context Model, i.e., we divided the inspection team into two groups (each consisted of 1 leader and 2 members) for the end-to-end ground truth. Passwords used for authentication are accepted and included in our ground truth. As for API keys with a length in the range of 6 to 30, we included them also since they are essentially machine-generated passwords and share similar context semantics with passwords. In total, we spent 180 person-hours annotating 395 passwords in 5,000 randomly selected source code files written in ten languages.

We compared PassFinder to regular-expression-based approaches, i.e., the state-of-the-art approaches that apply for multiple programming languages. The particular patterns used for matching in the regular-expression-based approaches affect their performance. Therefore, we inspected the source code of the most popular tools [17] [18] [12] [20] [24] that claim to be capable of detecting passwords in code to analyze the respective rules used for detection, so that we can find the approach with the best performance for comparison. We finally chose detect-secrets [7] from Yelp, a popular and continuously maintained project with over 2,000 stars as the baseline. It comes from two reasons. Firstly, the regular expressions in detect-secrets cover all the patterns supported by the remaining projects. That means these are the most well-crafted regular expressions that can be found in related open source tools. More importantly, it contains many heuristic strategies summarized by the maintainers to filter out false positives, making it stand out among all these tools. We kept these strategies in the comparison. Since detect-secrets is designed to detect secrets on a repository basis, containing features irrelevant to password detection. In our experiment, we extracted the code related to password detection from detect-secrets (commit `488334f`) and modified it to be consistent with our file-based detection workflow, with the regex part unchanged. The specific regular expressions is detailed in Table 2. As we can see, such regular expressions target assignment statements in one line in the source code. However, when encountered a more complex situation where passwords are defined by APIs or irregularly organized, as is shown in Figure 1, regular expression-based approaches will fail. Given that no conspicuous patterns exist, it is unrealistic to use regular expressions to match all possible situations.

Table 3: Comparison with regex-based methods

	PassFinder	Regex-based
Detected passwords	390	3,621
# True Positives	318	224
# False Positives	72	3,397
# False Negative	77	171
Precision	81.54%	6.19%
Recall	80.51%	56.71%
F1-score	81.02%	11.16%

As Table 3 shows, among the 5,000 files in the groundtruth, PassFinder reports 94 (41.96%) more leaked passwords than regular expression-based method, resulting a much higher coverage. This is mainly due to PassFinder's in-depth modeling for characteristics of human-chosen passwords. Besides, PassFinder has a significantly lower false-positive rate than the regular-expression-based method because our approach analyzes more semantic information. In contrast, due to lacking the ability to deal with the multiformity of source code, regular-expression-based method has extremely many false positives

4.3.1 False positives and false negatives. The false positive rate is 18.46% (72/390). In most cases, this is caused by PassFinder erroneously predicting the passwords' neighbors, such as username, database name, etc., which are used together for authentication, like passwords, since such elements might have a similar distribution as human-chosen passwords. Even in some cases, the developers use the same database name and passwords, which makes it difficult for PassFinder to distinguish. We consider such false alarms as acceptable because such false alarm implies PassFinder successfully locates authentication-related code, and the real passwords are probable to be identified by PassFinder as well. When it comes to false negatives, in many cases, it is because the Password Model fails to correctly classify the hard-coded passwords. With more data used for training, the performance of our models will continuous improve.

4.4 RQ3: Cross-language Effectiveness

We select code snippets from the ten most popular programming languages on GitHub, including JavaScript, Python, Java, etc., for the training of the Context Model. To validate whether our approach is generalizable to different languages, we iteratively use nine languages for training and the reserving one for testing. Table 4 presents the detailed performance for each language, i.e., the one for testing. We can see that our Context Model can also perform well in cross-language settings. Performance only slightly declines by 2.42% over average F1-scores. The result shows that PassFinder can learn the common patterns in code snippets written by different languages and be generalized to new languages.

Table 4: Performance of the cross-language effectiveness

Languages	Precision	Recall	F1-score
PHP	94.62%	94.62%	94.62%
JavaScript	95.59%	95.43%	95.51%
Ruby	95.29%	96.10%	95.69%
Python	94.41%	93.93%	94.17%
Java	94.76%	95.08%	94.92%
C#	95.72%	90.40%	92.98%
C++	91.81%	87.16%	89.42%
TypeScript	92.46%	89.32%	90.86%
Golang	87.94%	93.27%	90.53%
C	88.89%	89.64%	89.26%
Average	93.15%	92.50%	92.80%

5 LARGE-SCALE PASSWORD LEAKAGE ANALYSIS

This section reports our empirical study on password leakage from public GitHub repositories.

5.1 Candidate Files Collection

To investigate the password leakage in public GitHub repositories, we have to access the content of source code files on GitHub. A large-scale investigation is by no means trivial since it revolves around multi-million repositories [4]. In this work, instead of cloning each repository from GitHub and recursively traversing all inside files, we leverage the observation that only a tiny fraction of source code files are indeed authentication-related, leaving numerous files unnecessary to dig. Therefore, we firstly collect files of our interest, namely candidate files from GitHub, and then further inspect the content of the files to determine whether password leakage happens. Generally, there are two ways to collect files on GitHub directly: with GitHub mirrors like GHTorrent[38], BigQuery [21] or with the Search API [19] provided by GitHub, which supports key words querying with search qualifiers. We choose the latter because it can return the latest pushed files indexed by the GitHub Search engine, i.e., we can get results in near real-time as files are pushed to GitHub. In contrast, GitHub mirrors tend to contain much historical data and are not so frequently updated.

To get files that have a higher chance of containing passwords, i.e., candidate files, firstly, we have to query GitHub Search API with keywords. A naive way is to directly query GitHub API with password-related keywords like "password", "pwd" etc. However, it could bring many unnecessary indexing results since these keywords are prevalent in source code that implements authentication-related features (actually not real authentication operations) such as user registering. To avoid such query results to the greatest extent, we use the combination of keywords to generate target queries. Specifically, as is shown in Table 5, we select keywords according to different authentication scenes, covering both e-mail service, IaC, and general authentication. As for the selection of e-mail service providers, we consider choosing the most popular e-mail service since there are numerous providers. We reference the usage of e-mail service from two leaked datasets: Adobe [37] and CSDN and use a weighted sum model to calculate the final proportion of users. Note that CSDN [16] is a website for programmers, which could give us an insight into the e-mail usage among developers. We use the 6 most popular e-mail services, including Gmail, Yahoo, etc., which account for 73.02% in the weighted model. We also add "@outlook.com" and "@icloud.com", which are popular in recent years but not in the leaked dataset for complementary.

With the list of keywords, we combine keywords in the category "Password" with keywords in the remaining three categories to generate target query, such as "@gmail.com password". The keywords in the "Password" category are collected by inspecting keywords used in existing regex-based techniques. We specify the sort type for querying as "indexed" so that the GitHub Search API returns the most recently indexed results, ensuring we receive the latest results. For clarity, we focus on the ten most popular programming languages on GitHub, as is detailed in Section 3.4. We use the language type as search qualifier to ensure the coverage of our crawling since GitHub search API returns only a maximum of 1,000 results. The GitHub API returns a collection of files and their metadata for each query, including filename, URL, last modified time, etc. We then perform a content request to get the file's content finally. We build our file crawling system based on PyGithub [23]. In our crawling process, four GitHub API tokens are enough to eliminate the rate limit restrictions on GitHub API.

Table 5: Keywords used for generating target query

Category	Keywords
Password	password, passwd, pwd, secret, token, auth, access
E-mail	@gmail.com, @yahoo.com, @hotmail.com, @outlook.com, @icloud.com, @qq.com, @163.com, @126.com
IaC	host, server, ip, port
General	username, account

During the crawling procedure, even with the combination of keywords strategy, our target query could still match some files that are not likely to contain passwords. Therefore, we also filtered these files based on the file path in the repositories. We list the keywords we used for filtering in Table 7, which could mainly be divided into four categories. In this way, we avoid capturing many forked popular open-source projects and third-party libraries used in projects. There keywords were summarized by manually

Table 6: The statistics of the large-scale analysis

Language	# Cand. Files	% Cand. Files	# Cand. Repos	% Cand. Repos	# Det. Pwd	% Det. Pwd	# Det. Files	% Det. Files	# Det. Repos	% Det. Repos
Python	248,214	16.81%	103,286	19.16%	48,942	34.35%	32,036	33.99%	21,246	33.17%
Java	235,728	15.96%	76,942	14.27%	16,591	11.64%	11,333	12.03%	8,548	13.35%
JavaScript	220,963	14.96%	118,618	22.01%	22,107	15.52%	15,129	16.05%	10,879	16.99%
C#	164,310	11.13%	45,651	8.47%	9,272	6.51%	6,232	6.61%	4,398	6.87%
Typescript	142,557	9.65%	47,023	8.72%	6,741	4.73%	4,740	5.03%	3,378	5.27%
PHP	134,175	9.09%	63,857	11.85%	24,464	17.17%	15,162	16.09%	9,551	14.91%
Golang	116,026	7.86%	24,260	4.50%	3,845	2.70%	2,746	2.91%	1,874	2.93%
C++	93,155	6.31%	21,448	3.98%	6,288	4.41%	4,051	4.30%	2,617	4.09%
Ruby	68,582	4.64%	30,965	5.74%	2,900	2.04%	2,041	2.17%	1,493	2.33%
C	52,982	3.59%	11,865	2.20%	1,329	0.93%	768	0.81%	614	0.96%
Total	1,476,692	100%	539,012†	/†	142,479	100%	94,238	100%	64,045†	/†

† One Repository may contains different languages, we use the number of distinct repositories here.

inspecting content of files collected by our crawler. Our heuristic filtering strategy effectively filters out non-related files, allowing us to perform large-scale analysis on all files pushed to public GitHub repositories.

Table 7: Sample Keywords used to filter out unrelated files

Category	Sample Keywords
Third Party	site-packages, lib, include, sdk, third_party, 3rdParty, plugin, vendor, external, etc.
Software	symfony, openjdk, openssl, freebsd, linux, framworks_base, openwrt, jquery, jdk, etc.
Internationalization	lang, locale, i18n, zh_cn, etc.
Others	example, github.io, test, seed, mock, etc.

The GitHub Search API collection began on June 6, 2021, and finished on August 23, 2021. During this period of 75 days, we captured 1,476,692 distinct candidate files representing 543,915 repositories. Table 6 shows the distribution statistic of candidate files from different programming languages. Note that the candidate files used for labeling training dataset for Context Model and end-to-end ground truth were collected before the large-scale analysis, but with the same strategy.

5.2 Landscape

As is shown in Table 6, PassFinder totally discovered 142,479 passwords in 94,238 (6.38%) files representing 64,045 (11.88%) repositories in 75 days. On average, nearly 1,900 passwords are leaked per day. This indicates that such password leakage is indeed pervasive. Table 6 breaks down the total number of leaked passwords by language. Source code files written by Python leaked the most passwords, followed by PHP. Among all ten languages, the passwords leaked by most four languages account for 78.68% of all leaks. This may be caused by the usage scenarios of different programming languages.

5.3 Manual Review

To further inspect the leaked passwords detected by PassFinder, we carried out a rigorous manual review of the leakage dataset. Specifically, we randomly selected 4,000 distinct leaks and examined the file and repository containing passwords on GitHub's website.

Sensitivity. PassFinder is able to detect the passwords used for authentication in source code. However, it is not known whether the detected passwords are sensitive or not. Therefore, two Ph.D. students (leaders in previous labeling work) inspected each leak to evaluate each leak as sensitive, non-sensitive, or not a password. Specifically, we thoroughly examined the context of the passwords and related files in the repositories. Passwords determined to be used only for testing or demonstration are considered non-sensitive. Among all 4,000 detected leaks, 820 (20.50%) are evaluated as not a password, i.e., false positives of PassFinder. This is consistent with our previous evaluation data. As for the remaining 3,180 true positive leaks, 3,138 are sensitive. Therefore, we can estimate that the overall sensitivity of our entire data is 78.48%. This indicates that most of the discovered secrets are sensitive, resulting in developers and services at risk of compromise.

Authentication Scenarios. We also paid attention to the specific authentication scenarios in the manual review process. Among the 3,138 sensitive leaks, 2,272 passwords are used for e-mail authentication, accounting for the most (72.24%). Developers who integrate features involving automating e-mail sending may accidentally leave their accounts and passwords in the source code. The ratio of passwords for IaC use (including use for database, SSH, FTP, etc.) is 12.59%. Among all these 395 IaC leakages, we found 89 public hosts/IPs leaked altogether with the passwords, which means the attackers who get the passwords can directly connect to the database without further scouting the deployment information

of the project if no additional security measures are taken. There also exist 170 API keys detected by PassFinder, which does not follow any format. PassFinder can also detect API keys since they are machine-generated strings with similar context as password authentication. We also found 157 Wi-Fi passwords, which appear primarily in C/C++ projects that are related to IoT. The 144 remaining passwords have different authentication scenarios. Some are used to authenticate to platforms targeted to which the developers develop spiders using tools like Selenium. In some cases, the developers authenticate themselves to a remote HTTP server to further carry out automated operations. Also, some developers use passwords to request services that deal with CAPTCHA and SMS, etc.

5.4 Repository Characteristic

Popularity. The popularity of repositories can be characterized with the number of stars, forks, and watchers. Specifically, for each repository that contains passwords, we calculate the sum of the number of stars, forks, and watchers by August 24, 2021. By this time, 3,950 repositories have been removed from GitHub by the owners. As for the remaining 60,095 repositories, repositories with no star, fork, and watcher account for over 75.97% (45,653), which shows that the vast majority of repositories are unpopular, e.g., probably neglected by all other users except the owners. However, lacking popularity does not mean that the severity of leaks may cause is reduced. Although it is not likely for a curious user to catch such leakage, malicious attackers can harvest passwords efficiently, as proved in this work. In a sense, such unpopular repositories should be paid more attention since they might contain real-world deployed projects.

Activeness. We characterize the activeness of repositories from two dimensions. One direct dimension is the commit history. The more commits a repository has, the more active it is. We collected the commit counts for all repositories by August 24, 2021 and calculate the Cumulative Distribution Function (CDF) of the commit counts. As is shown in Figure 5, the commit count of 12,894 (21.46%) repositories is no more than 2 (It is a common practice that developers create the initialization commit on GitHub website and push all files from local, which results in 2 commits), which means the authors just upload the repositories to GitHub without further maintenance. Over 19,719 (32.81%) repositories are under continuous maintenance (with a commit count over 30). On the other hand, we investigated the existence of the README file in these repositories since this information implies whether the authors want to promote the repositories or just host them on GitHub. We found that over one-half (31,316, 52.11%) of the repositories have no README file or have a simple README file containing only the repository name with a simple description. This shows that the vast majority of owners are not willing to promote their repositories.

5.5 Password Lifetime

Besides the coverage of password leakage, we also tried to characterize the lifetime of these leaked passwords, i.e., after how long would the owners realize the exposure and remove the passwords from public repositories. We began our monitoring on passwords lifetime since August 12, 2021. For each leaked password since then,

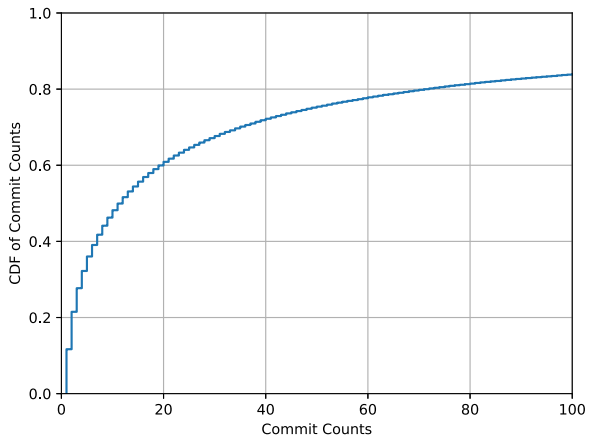

Figure 5: The CDF of commit counts of different repositories

we queried GitHub daily to determine if the passwords still existed on the head of the default branch. The result of daily monitoring is shown in Figure 6. As we can see, in the first two days, the curve drops quickly, and 7.92% of the passwords are removed. Then the curve becomes flat. However, after 16 days the passwords are detected, less than 17.68% were removed, which leaves a long window for attackers. The proportion of passwords remaining exposed is in correspondence to API Keys leakage in previous work [49]. Further, we examined whether the users removed their passwords thoroughly, i.e., removed the commits that contain passwords. After manually inspected the commit history (of repositories that were not deleted by the owner), to our surprise, none of these owners rewrote the repository history, leaving these passwords still accessible.

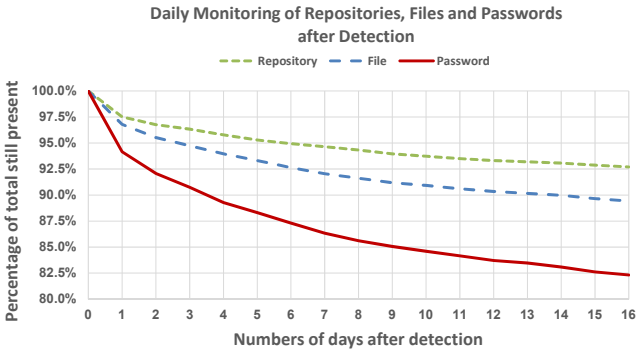

Figure 6: Password lifetime.

6 DISCUSSION

This section discusses ethics consideration, mitigations of password leakage problems, as well as limitations of our work.

6.1 Ethics Consideration

In this paper, we conducted experiments collecting 142,479 leaked passwords that revolved 64,045 public repositories on GitHub that could have severe consequences if abused. Apart from our search queries, our methodology is passive. All secrets that we collect were already exposed when we find them. Thus this research does not create vulnerabilities where they did not already exist. Furthermore, we never attempt to use any of the discovered passwords other than for the analytics in this paper, even for innocuous purposes like merely verifying the passwords' validity. It prevents us from obtaining any sensitive, proprietary, or personal information from the password owners. As of the camera-ready, we are currently working to notify vulnerable repository owners of our findings.

6.2 Mitigations

We have shown in this work that an attacker with minimal resources could compromise assets of numerous developers and enterprises by stealing leaked passwords. We discuss some mitigations for such attacks here. A naive approach is to impose strict audits for queries to GitHub Search API. If some abnormal behavior is detected, GitHub could further block related tokens or accounts. In this way, it would be more difficult for active attackers harvesting passwords from public repositories. However, a motivated attacker could use numerous tokens or accounts to achieve his goal. Such a block strategy could also lead to a decline in common users' experience for false-positive cases. What's more, this approach does not solve the most fundamental problem. i.e., the passwords are still exposed in public GitHub repositories. It could be noticed by any curious users, leaving a security threat for password owners.

The key to thoroughly solve the problem of password leakage is to detect the leakage timely. GitHub's Secret Scanning [10] has been proven to be effective in detecting secrets with distinct formats. We hope GitHub could integrate our detection techniques into Secret Scanning and instantly alert developers once the password leakage is caught. In this way, the developers could realize the problem. Also, we suggest developers using tools such as Vault to store secrets.

6.3 Limitations of Our Work

In this section, we briefly detail the limitations of our work. First, we do not have ground-truth knowledge of whether the passwords we discover are exploitable. Though great efforts were paid to exclude them, some passwords may be stale or simply invalid. Without actually testing such passwords (which we do not do for ethical reasons), it is not possible to have certainty that a secret is exploitable.

Second, we proved the feasibility of our approach in this paper, i.e. leveraging both intrinsic and contextual characteristic for password detection in source code. However, our two models can continuously evolve with more high-quality data used for training as well as more sophisticated algorithms that make better use of collected data. Also, our approach can seamlessly be combined with existing methods like regular expression-based and entropy-based techniques to improve the coverage further.

Finally, we focus only on password leakage on GitHub in this work. While GitHub is the largest public code hosting platform, many other code hosting services like GitLab or BitBucket. The leakage in such platforms has not been systematically investigated. We leave this for future work.

7 CONCLUSION

This paper gives our research on detecting password leakage on GitHub repositories at a large scale. To address the main challenge that existing approaches cannot effectively identify passwords in source code files, we propose PassFinder, a new technique for automated password detection. PassFinder leverages the intrinsic characteristics of passwords, together with the semantic information of code snippet, which is authentication-related, to accurately identify passwords in code files written in different languages. The evaluation results showed PassFinder achieves high precision and outperforms existing heuristic approaches. Using this technique, we performed the first large-scale and longitudinal analysis of password leakage on GitHub. Our result shows that hundreds of thousands of passwords are leaked at a rate of thousands per day. Our result also highlights the importance of data protection in today's open-source ecosystem.

8 ACKNOWLEDGEMENTS

The authors thank the anonymous reviewers for their constructive comments. We also thank the other participants of this project for their labeling efforts. This work is, in part supported by the National Science Foundation of China (No. 61872237). Yuanyuan Zhang is the corresponding author.

REFERENCES

[1] Linkedin leaked passwords. https://hashes.org/public.php, 2012. [Online; Accessed on 02/04/2022].
[2] Github kills search after hundreds of private keys exposed. https://it.slashdot.org/story/13/01/25/132203/github-kills-search-after-hundreds-of-private-keys-exposed, 2013. [Online; Accessed on 02/04/2022].
[3] Aws urges devs to scrub secret keys from github. https://developers.slashdot.org/story/14/03/24/0111203/aws-urges-devs-to-scrub-secret-keys-from-github, 2014. [Online; Accessed on 02/04/2022].
[4] The 2020 state of the octoverse. https://octoverse.github.com/, 2020. [Online; Accessed on 02/04/2022].
[5] Github language stats. https://madnight.github.io/githut/#/pull_requests/2020/1, 2020. [Online; Accessed on 02/04/2022].
[6] How i got my first big bounty payout with tesla. https://medium.com/heck-the-packet/how-i-got-my-first-big-bounty-payout-with-tesla-8d28b520162d, 2020. [Online; Accessed on 02/04/2022].
[7] Solarwinds github password. https://www.theregister.com/2020/12/16/solarwinds_github_password/, 2020. [Online; Accessed on 02/04/2022].
[8] 1password. https://1password.com/, 2021. [Online; Accessed on 02/04/2022].
[9] About github. https://github.com/about, 2021. [Online; Accessed on 02/04/2022].
[10] About secret scanning. https://docs.github.com/en/github/administering-a-repository/about-secret-scanning, 2021. [Online; Accessed on 02/04/2022].
[11] Android dalvik bytecode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode, 2021. [Online; Accessed on 02/04/2022].
[12] Bandit. https://github.com/PyCQA/bandit, 2021. [Online; Accessed on 02/04/2022].
[13] Common-credentials. https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/10-million-password-list-top-1000.txt, 2021. [Online; Accessed on 02/04/2022].
[14] Creating a strong password. https://support.google.com/accounts/answer/32040?hl=en, 2021. [Online; Accessed on 02/04/2022].
[15] Credscan. https://secdevtools.azurewebsites.net/helpcredscan.html, 2021. [Online; Accessed on 02/04/2022].
[16] csdn. https://www.csdn.net/, 2021. [Online; Accessed on 02/04/2022].
[17] Detect-secrets release version. https://github.com/Yelp/detect-secrets/releases, 2021. [Online; Accessed on 02/04/2022].
[18] git-secrets. https://github.com/awslabs/git-secrets, 2021. [Online; Accessed on 02/04/2022].
[19] Github search api. https://docs.github.com/en/github/searching-for-information-on-github, 2021. [Online; Accessed on 02/04/2022].

[20] gitleaks. https://github.com/zricethezav/gitleaks, 2021. [Online; Accessed on 02/04/2022].

[21] Google bigquery. https://cloud.google.com/bigquery, 2021. [Online; Accessed on 02/04/2022].

[22] Keepassx. https://www.keepassx.org/, 2021. [Online; Accessed on 02/04/2022].

[23] pygithub. https://github.com/PyGithub/PyGithub, 2021. [Online; Accessed on 02/04/2022].

[24] shhgit. https://github.com/eth0izzle/shhgit, 2021. [Online; Accessed on 02/04/2022].

[25] Trufflehog. https://github.com/dxa4481/truffleHog, 2021. [Online; Accessed on 02/04/2022].

[26] Allamanis, M., Barr, E. T., Devanbu, P., and Sutton, C. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR) 51*, 4 (2018), 1–37.

[27] Allamanis, M., Peng, H., and Sutton, C. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning* (2016), pp. 2091–2100.

[28] Alon, U., Zilberstein, M., Levy, O., and Yahav, E. A general path-based representation for predicting program properties. *ACM SIGPLAN Notices 53*, 4 (2018), 404–419.

[29] Alon, U., Zilberstein, M., Levy, O., and Yahav, E. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages 3*, POPL (2019), 1–29.

[30] Bavishi, R., Pradel, M., and Sen, K. Context2name: A deep learning-based approach to infer natural variable names from usage contexts. *arXiv preprint arXiv:1809.05193* (2018).

[31] Bichsel, B., Raychev, V., Tsankov, P., and Vechev, M. Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), pp. 343–355.

[32] Bielik, P., Raychev, V., and Vechev, M. Phog: probabilistic model for code. In *International Conference on Machine Learning* (2016), PMLR, pp. 2933–2942.

[33] Bird, S., Klein, E., and Loper, E. *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.", 2009.

[34] Bojanowski, P., Joulin, A., and Mikolov, T. Alternative structures for character-level rnns. *arXiv preprint arXiv:1511.06303* (2015).

[35] Das, A., Bonneau, J., Caesar, M., Borisov, N., and Wang, X. The tangled web of password reuse. In *NDSS* (2014), vol. 14, pp. 23–26.

[36] Dietterich, T. G. Ensemble methods in machine learning. In *International workshop on multiple classifier systems* (2000), Springer, pp. 1–15.

[37] Foster, I. D., Larson, J., Masich, M., Snoeren, A. C., Savage, S., and Levchenko, K. Security by any other name: On the effectiveness of provider based email security. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), pp. 450–464.

[38] Gousios, G., and Spinellis, D. Ghtorrent: Github's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)* (2012), IEEE, pp. 12–21.

[39] Hindle, A., Barr, E. T., Gabel, M., Su, Z., and Devanbu, P. On the naturalness of software. *Communications of the ACM 59*, 5 (2016), 122–131.

[40] Hitaj, B., Gasti, P., Ateniese, G., and Perez-Cruz, F. Passgan: A deep learning approach for password guessing. In *International Conference on Applied Cryptography and Network Security* (2019), Springer, pp. 217–237.

[41] Hranický, R., Zobal, L., Ryšavý, O., Kolář, D., and Mikuš, D. Distributed pcfg password cracking. In *European Symposium on Research in Computer Security* (2020), Springer, pp. 701–719.

[42] Hwang, K., and Sung, W. Character-level language modeling with hierarchical recurrent neural networks. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2017), IEEE, pp. 5720–5724.

[43] Iyer, S., Konstas, I., Cheung, A., and Zettlemoyer, L. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (2016), pp. 2073–2083.

[44] Karampatsis, R.-M., Babii, H., Robbes, R., Sutton, C., and Janes, A. Big code!= big vocabulary: Open-vocabulary models for source code. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)* (2020), IEEE, pp. 1073–1085.

[45] Kuo, C., Romanosky, S., and Cranor, L. F. Human selection of mnemonic phrase-based passwords. In *Proceedings of the second symposium on Usable privacy and security* (2006), pp. 67–78.

[46] LaValle, S. M., Branicky, M. S., and Lindemann, S. R. On the relationship between classical grid search and probabilistic roadmaps. *The International Journal of Robotics Research 23*, 7-8 (2004), 673–692.

[47] Li, Z., Zou, D., Xu, S., Jin, H., Qi, H., and Hu, J. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (2016), pp. 201–213.

[48] Ma, J., Yang, W., Luo, M., and Li, N. A study of probabilistic password models. In *2014 IEEE Symposium on Security and Privacy* (2014), IEEE, pp. 689–704.

[49] Meli, M., McNiece, M. R., and Reaves, B. How bad can it git? characterizing secret leakage in public github repositories. In *NDSS* (2019).

[50] Melicher, W., Ur, B., Segreti, S. M., Komanduri, S., Bauer, L., Christin, N., and Cranor, L. F. Fast, lean, and accurate: Modeling password guessability using neural networks. In *25th USENIX Security Symposium (USENIX Security 16)* (2016), pp. 175–191.

[51] Morris, R., and Thompson, K. Password security: A case history. *Communications of the ACM 22*, 11 (1979), 594–597.

[52] Nan, Y., Yang, Z., Wang, X., Zhang, Y., Zhu, D., and Yang, M. Finding clues for your secrets: Semantics-driven, learning-based privacy discovery in mobile apps. In *NDSS* (2018).

[53] Oesch, S., and Ruoti, S. That was then, this is now: A security evaluation of password generation, storage, and autofill in browser-based password managers. In *Proc. of USENIX Security Symp* (2020).

[54] Opitz, D., and Maclin, R. Popular ensemble methods: An empirical study. *Journal of artificial intelligence research 11* (1999), 169–198.

[55] Pal, B., Daniel, T., Chatterjee, R., and Ristenpart, T. Beyond credential stuffing: Password similarity models using neural networks. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 417–434.

[56] Pasquini, D., Ateniese, G., and Bernaschi, M. Interpretable probabilistic password strength meters via deep learning. In *European Symposium on Research in Computer Security* (2020), Springer, pp. 502–522.

[57] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. In *NIPS-W* (2017).

[58] Pearman, S., Thomas, J., Naeini, P. E., Habib, H., Bauer, L., Christin, N., Cranor, L. F., Egelman, S., and Forget, A. Let's go in for a closer look: Observing passwords in their natural habitat. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 295–310.

[59] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[60] Pradel, M., Murali, V., Qian, R., Machalica, M., Meijer, E., and Chandra, S. Scaffle: bug localization on millions of files. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2020), pp. 225–236.

[61] Rahman, A., Parnin, C., and Williams, L. The seven sins: security smells in infrastructure as code scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019), IEEE, pp. 164–175.

[62] Rahman, M. R., Rahman, A., and Williams, L. Share, but be aware: Security smells in python gists. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2019), IEEE, pp. 536–540.

[63] Raychev, V., Bielik, P., and Vechev, M. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices 51*, 10 (2016), 731–747.

[64] Raychev, V., Bielik, P., Vechev, M., and Krause, A. Learning programs from noisy data. *ACM Sigplan Notices 51*, 1 (2016), 761–774.

[65] Raychev, V., Vechev, M., and Krause, A. Predicting program properties from'big code'. *Communications of the ACM 62*, 3 (2019), 99–107.

[66] Sinha, V. S., Saha, D., Dhoolia, P., Padhye, R., and Mani, S. Detecting and mitigating secret-key leaks in source code repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories* (2015), IEEE, pp. 396–400.

[67] Vasilescu, B., Casalnuovo, C., and Devanbu, P. Recovering clear, natural identifiers from obfuscated js names. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering* (2017), pp. 683–693.

[68] Wang, D., Wang, P., He, D., and Tian, Y. Birthday, name and bifacial-security: understanding passwords of chinese web users. In *28th USENIX Security Symposium* (2019), pp. 1537–1555.

[69] Weir, M., Aggarwal, S., Collins, M., and Stern, H. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), pp. 162–175.

[70] Weir, M., Aggarwal, S., De Medeiros, B., and Glodek, B. Password cracking using probabilistic context-free grammars. In *2009 30th IEEE Symposium on Security and Privacy* (2009), IEEE, pp. 391–405.

[71] Yu, Z., Cao, R., Tang, Q., Nie, S., Huang, J., and Wu, S. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2020), vol. 34, pp. 1145–1152.

[72] Yu, Z., Zheng, W., Wang, J., Tang, Q., Nie, S., and Wu, S. Codecmr: Cross-modal retrieval for function-level binary source code matching. *Advances in Neural Information Processing Systems 33* (2020).

[73] Zhang, X., Zhao, J., and LeCun, Y. Character-level convolutional networks for text classification. *arXiv preprint arXiv:1509.01626* (2015).

[74] Zhou, Y., Wu, L., Wang, Z., and Jiang, X. Harvesting developer credentials in android apps. In *Proceedings of the 8th ACM conference on security & privacy in wireless and mobile networks* (2015), pp. 1–12.