Optimizing Leak Detection in Open-Source Platforms with Machine Learning Techniques

Sofiane Lounici[1], Marco Rosa[1], Carlo Maria Negri[1], Slim Trabelsi[1] and Melek Önen[2]
[1]*SAP Security Research, France*
[2]*EURECOM, France*

Keywords: Data mining, security tool, machine learning

Abstract: Public code platforms like GitHub are exposed to several different attacks, and in particular to the detection and exploitation of sensitive information (such as passwords or API keys). While both developers and companies are aware of this issue, there is no efficient open-source tool performing leak detection with a significant precision rate. Indeed, a common problem in leak detection is the amount of false positive data (i.e., non critical data wrongly detected as a leak), leading to an important workload for developers manually reviewing them. This paper presents an approach to detect data leaks in open-source projects with a low false positive rate. In addition to regular expression scanners commonly used by current approaches, we propose several machine learning models targeting the false positives, showing that current approaches generate an important false positive rate close to 80%. Furthermore, we demonstrate that our tool, while producing a negligible false negative rate, decreases the false positive rate to, at most, 6% of the output data.

1 Introduction

Data protection has become an important issue over the last few years. Despite the multiplication of awareness campaigns and the growth of good development practices, we observe a major rise of data leaks in 2019, with passwords representing 64% of all data compromised[1]. It has become a huge concern for companies to protect themselves and to efficiently detect these data leaks.

GitHub[2] is a hosting platform for software development version control. With more than 100 million repositories (with at least 28 million public ones), it is the largest host of source code in the world. Users can use GitHub to publish their code, to collaborate on open-source projects, or simply to use publicly available projects. In such an environment, one of the most critical threats is represented by hardcoded (or plaintext) credentials in open-source projects (MITRE, 2019). Indeed, when developers integrate an authentication process in their source code (e.g., a database access), a common practice is the use of password or authentication tokens (also known as API Keys). In this process, there is a risk that secrets may be unintentionally published in publicly available open-source projects, possibly leading to data breaches. For instance, Uber sustained in 2016 a massive data leak[3], affecting 57 million customers by revealing personal data such as names, and phone numbers. This attack was originating from a password found in a private GitHub repository.

Several tools are already available to detect leaks in open-source platforms such as GitGuardian[4] or TruffleHog[5]. Nevertheless, the diversity of credentials, depending on multiple factors such as the programming language, code development conventions, or developers' personal habits, is a bottleneck for the effectiveness of these tools. Their lack of precision leads to a very high number of pieces of code detected as leaked secrets, even though they consist in perfectly legitimate code. Data wrongly detected as a leak is called *false positive data*, and compose the huge majority of the data detected by currently available tools. Thus, various companies (including GitHub itself[6]), are starting to automate the detection of leaks while reducing false positive data.

In this paper, we present a novel approach to analyze GitHub open-source projects for data leaks, with a significant decrease in false positives thanks to the use of machine learning techniques. First, a **Regex**

[1]https://preview.tinyurl.com/y7bygg8d
[2]https://www.github.com
[3]https://tinyurl.com/yd3c37lc
[4]https://www.gitguardian.com/
[5]https://github.com/dxa4481/truffleHog
[6]https://preview.tinyurl.com/ycnllvfd

Scanner searches through the source code for potential leaks, looking for any correspondence with a set of programming patterns. Then, machine learning models filter the potential leaks by detecting false positive data, before a human reviewer can check the classified data manually to correct possible wrongly classified data. These machine learning models are using various techniques such as data augmentation (Shorten and Khoshgoftaar, 2019), code stylometry (Long et al., 2017; Quiring et al., 2019) and reinforcement learning (Watkins and Dayan, 1992).

The main contributions of this paper can be summarized as follows.

- We present an automated leak detector for passwords and API Keys in open-source platforms, with low false positive rate.
- We evaluate our solution by scanning 1000 public GitHub and 300 company-owned repositories, and we show that the classic regular expression approaches generate a high false positive rate, that we estimate close to 82%.
- We manually assess the results of this scan, proving that our solution reaches a negligible false negative rate.
- We investigate the false positives induced by the machine learning models, and we show it is between 5% and 32% of the filtered data (hence between 1% and 6% of the overall data)

Outline. We introduce an overview of the problem of leak detection in Section 2.1, alongside an architecture of our framework in Section 2.2. We further detail the different modules: We describe the Path Model in Section 3, the Snippet models in Section 4, and the Similarity model in Section 5. We present an evaluation of our approach, focusing on the false positive rate induced by the machine learning models, in Section 6. We discuss the related work in Section 7. We finally address potential privacy concerns in Section 8.

2 Overview

2.1 Problem statement

A leak is a piece of information in a source code, published on open-source platforms such as GitHub, disclosing personal and sensitive data. Data leaks can be caused by any type of developer, such as independent developers or important corporations. For instance, a password published on GitHub by an Uber's employee led to the disclosure of personal information of 57 millions customers[7].

Several types of data leaks exist: API Keys (e.g., AWS credentials), email passwords, database credentials, etc. Although detection techniques exist, current approaches do not achieve a satisfying precision rate, leading to a high false positive rate, i.e., non-negligible part of data is wrongly classified as leak. A high false positive rate implies an important workload for reviewers who manually check the accuracy of the classification.

In this paper, we present an automated leak detector for open-source platforms with low false positive rate, powered by machine learning.

We identify three main problems we intend to tackle. To begin with, we notice that open-source projects often provide the documentation of their code, together with tutorials, tests, and example files. These situations are easily recognizable by the actual path name (e.g., `src/Example.py`, `connectionTutorial.java`, etc.). An important amount of passwords or database credentials are located in these type of files and are never used in production, increasing the false positive rate.

Moreover, current solutions such as GitGuardian, Trufflehog, S3Scanner, GitHub Token Scanning or others in (Sinha et al., 2015) consist of regular expression classifiers and exclusively focus on API Keys, ignoring passwords as a category of leak. Indeed, the detection of API Keys creates a negligible amount of false positive data (due to the particular patterns). Thus, it is easier to handle them with simple regular expression classifiers. Passwords, on the other hand, are difficult to identify with classic methods, even though they account for the majority of leaks, leading to a high false positive rate. Current solutions offer little to no automated false positive filtering (except with simple heuristics) because they discard the most important source of false positive data in their analysis.

Additionally, the detection of leaks with low false positive rate is usually performed using supervised machine learning techniques which by definition incur the need for labelled training data. The collection of leak data in this context remains a challenge for several reasons: (i) on a theoretical point of view, passwords/credentials are privacy sensitive data, (ii) on a practical point of view the training dataset needs to satisfy general properties such as balance or diversity, and current machine learning approaches cannot guarantee these properties while maintaining a reasonable manual workload to sanitize, anonymize and

[7] https://tinyurl.com/yd3c37lc

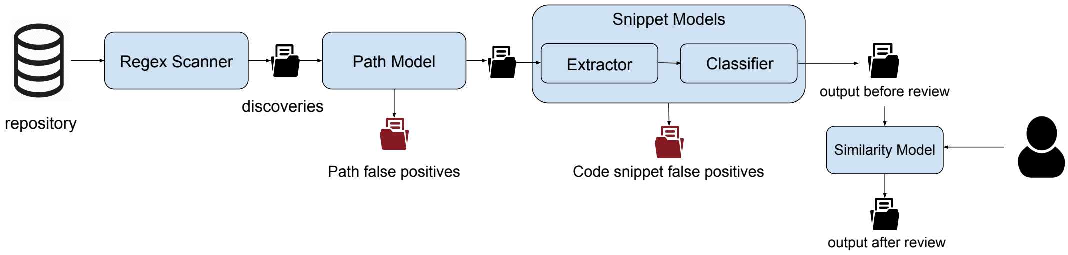

Figure 1: Architecture of our approach

label data.

2.2 Our approach

In order to detect leaks with high precision and low false positive rate, we begin with the use of a regular expression scanner similar to classical approaches. We further propose to make the distinction between two sources of false positives: *Path false positives* (e.g., data located in documentation or example files) and *Code snippet false positives* (e.g., dummy credentials or initialization variables). These two sources of false positives can be tackled by two separate machine learning models: the Path model and the Snippet model. Consequently, our solution regroups the following components.

Regex Scanner: Given an open-source repository, the Regex Scanner searches through the source code history to detect any credential, API Key or plaintext password, and is considered as the default component in classic approaches. The Regex Scanner analyzes each source code modification by a developer over time, retrieving the link between these modifications and a set of regular expressions. The output of the Regex Scanner over a repository R is a set of m discoveries $D = \{d_1, ..., d_m\}$, each discovery containing a path f_i and a code snippet s_j.

Path model: The Path model analyzes each path f_i to reduce Path false positives, and outputs a list of filtered discoveries. We propose to make use of the Linear Continous Bag-of-Words model to represent and link words to the actual context. Thanks to this model, we already reduce false positives by 69%.

The **Snippet models** filter false positives related to code snippets. A code snippet is more complex to analyze than a file path (more diversity, more irregular patterns, etc.), and may contain non-negligible amount of irrelevant data for leak classification (function names, type names, method names, symbols, etc.). Compared to the Path model, an additional preprocessing step is needed before the actual leak detection. Therefore, the Snippet models consist of two main components:

Extractor: The Extractor identifies relevant information in the snippets, i.e., the variable name and the value assigned. As mentioned before, it is difficult to collect relevant data to train the Extractor. Thus, we implement data augmentation techniques through reinforcement learning.

Classifier: The Classifier takes the relevant information extracted as inputs to classify a code snippet as a leak or as a false positive. At this step, we consider again a LCBOW model, leading to a reduction of 13% of the discoveries with the combination of the Extractor and the Classifier.

As a final step, once automated components output the leaks they have detected, a human reviewer manually checks the accuracy of the classification by *flagging* (i.e., re-classifying manually) a leak as a false positive.

Similarity model: The Similarity model can assist the human reviewer by flagging similar discoveries as false positives to reduce her workload.

Figure 1 gives an overview of the architecture of the proposed framework. In the following sections, we describe the design choices for each of these components while illustrating their use with three example scenarios.

Scenario 1: Consider the code snippet `String password = "Ub4!l"`, located in the file `src/Example.py`. The Regex Scanner identifies the key word *password*, so that the discovery is classified as a leak. Then, the Path model analyzes the file path, and discards the leak as a Path false positive (due to the word `Example`).

Scenario 2: Consider the code snippet `String password = "Ub4!l"`, located in the file `src/run.py`. The Regex Scanner still identifies the key word *password*, while the Path model does not discard the leak due to its Path. The Extractor outputs the combination *(password, Ub4!l)*, and the Classifier classifies this code snippet as a leak.

Scenario 3: Consider the code snippet `String password = "INSERT_CREDENTIAL_HERE"`, located in the file `src/run.py`. The Extractor outputs the combination *(password, INSERT_CREDENTIAL_HERE)*, and the Classifier classifies the code snippet as a false positive.

3 Path Model

The goal of the Path model is to reduce the Path false positives. This model analyzes where a leak is identified in an open-source repository (i.e., its file path), and gives a first classification on whether the leak is relevant or not. The Path model relies on a basic machine learning technique called Linear Continous Bag-of-Words (LCBOW).

3.1 LCBOW model

In the field of Natural Language Processing, there exist many possible choices for a text representation method among which the *word embedding* one where words are mapped to vectors like in word2vec (Mikolov et al., 2013b) or Bag-of-Words (BoW) (Ma et al., 2019) representations. In this work, we consider the use of the Linear Continuous Bag-of-Words (LCBOW) model (Mikolov et al., 2013a; Joulin et al., 2016), especially for its efficiency. We briefly explain how the LCBOW model is built.

Let's denote a list of words as a document corpus of size N. A sentence in the document corpus is composed of N-gram features $\{w_1, w_2, ..., w_N\}$. We obtain the feature representations via a weight matrix U to obtain $x_i = U \cdot w_i$. Then, we define y as the linear Bag-of-Words of the document, by averaging all the feature representations x_i:

$$y = \frac{1}{N}\sum_{i=1}^{N} x_i$$

y is the input of a hidden layer associated to a weight matrix V, such that output $z = V \cdot y$. We can compute the probability that a word vector belongs to the j^{th} class as $p_j = \sigma(z_j)$, with $\sigma(z_j)$[8] being the softmax function. Finally, the weight matrices U and V are computed by minimizing the negative log-likelihood of the probability distribution, using stochastic gradient descent, namely:

$$-\frac{1}{N}\sum_{k=1}^{N} y_k \cdot log\big(\sigma(V \cdot U \cdot w_i)\big)$$

[8] $\sigma(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{m} e^{z_k}}$

In the remaining of the paper, we will use the notation $LCBOW(w)$ to describe the vector representation of the word w.

3.2 Data pre-processing

The Regex Scanner outputs a list of discoveries, each discovery containing a path f_i (used as an input for the Path model) and a code snippet s_j (used as an input for the Snippet models). Some pre-processing phase is needed for both these data: First, we remove non-alphanumerical characters, before applying stemming and lemmatization, which are natural language processing techniques (Sun et al., 2014). We split the input data in words to obtain $f_{preprocc} = \{f_i^1, ..., f_i^{k_f}\}$. In order to respect common coding conventions while standardizing the input data, we apply the Java coding convention to each word in f_i (the choice of coding convention is irrelevant as long as it is standardized for all inputs).

Example: If we consider *Scenario 1*, with f = `src/Example.py` and s = `String password = "Ub4!l"`, the pre-processing phase outputs $f_{preproc} = \{src, Example, py\}$ and $s_{preproc} = \{String, password, Ub4!l\}$

3.3 Training phase

The workload to gather sufficient training data and to review labeled items can be handled by a human reviewer. Since the path name is not a sensitive piece of information, the data sanitization aspect can be reduced to a minimum. We collected 100k file names from 1000 GitHub repositories (analyzed in our evaluation in Section 6), which we labeled using regular expressions and manual checks. We applied the data pre-processing techniques and we train a LCBOW model, achieving 99% of accuracy on this dataset.

4 Snippet Models

In this section, we detail the design choices for the Snippet models: the Extractor and the Classifier. To fully understand our approach, we propose to introduce several concepts, aimed to be used as building blocks for these models.

4.1 Building blocks

4.1.1 Code stylometry

Each developer has her own coding habits, depending on many factors such as the coding language or the occurrences of given key words. We introduce a concept called *code stylometry*, aiming to encapsulate into a vector the main characteristics of these coding habits.

Example: Consider a Python developer, focused on software development. This developer will probably use key words such as `password` or `pass_word` to do password assignments (e.g., `password = "Ub4!l"`). A different developer, focused on database management, might prefer keywords such as `root` or `db` (like `db.root = "Ub4!l"`. These design choices will result in two different code stylometry vectors.

Supposing that we have extracts of code belonging to a developer (denoted $\mathcal{E}$), we compute her code stylometry based of these extracts[9].

4.1.2 Data augmentation

As previously mentioned in Section 2.1, obtaining a dataset for leaks on GitHub is complicated. Indeed, since we are dealing with sensitive data, we have to follow and comply with privacy guidelines, e.g., performing data sanitization. The collected data also needs to be labelled, which may require a significant manual workload. In addition, the diversity of leaks in open-source repositories usually follows the Pareto rule, meaning that 80% of the data leaks are originating from the same few programming patterns (for instance, `password="1234"` is extremely common). Therefore, collecting a diverse dataset in order to train a machine learning model (to have good generalization properties and avoid overfitting (Shorten and Khoshgoftaar, 2019)) would be difficult to reach from a practical point of view. For these reasons, we propose to use data augmentation techniques in order to enhance the size and the diversity of the dataset with no extra cost in labelling or sanitization.

Data augmentation is a set of techniques to enhance the diversity of a dataset without new data. It is particularly used in image processing (Shorten and Khoshgoftaar, 2019), by applying filters to images in order to produce new training samples. The main benefit is to expand a dataset (fixing class imbalance or adding diversity in the training samples) with limited pre-processing cost. Data augmentation can also prevent overfitting (i.e, when a machine learning model is not able to generalize from the training data).

Example: Consider two leaks `password="Ub4!"` and `mypass="1234"`. If we switch the variable names to obtain `password="1234"` and `mypass="Ub4!"`, we have in fact created two new leaks. In general, given a pattern `key="value"`, any pair of $(key, value)$ can be chosen to obtain a new leak. Every time another variable name is collected, data augmented leaks can be obtained by the re-arrangement of already existing data. More specifically, when a new programming pattern is collected for password assignment (e.g., `DataBase.key="value"`) additional leaks can be obtained, creating diversity from limited dataset.

Data: $\mathcal{D}$, π, $style_{ref}$
Result: Training Data for π T_π
while *condition is True* **do**
  $style \leftarrow choose\_actions(\pi, \mathcal{D})$;
  $reward_{sim} \leftarrow similarity(style, style_{ref})$;
  $update\_choices(reward_{sim})$;
end
$T_\pi \leftarrow choose\_actions(\pi, \mathcal{D})$

Algorithm 1: Q-learning algorithm

Data: Collected data $\mathcal{D}$, patterns Π, extracts $\mathcal{E}$
Result: *model*
$style_{ref} \leftarrow stylometry(\mathcal{E})$;
for π *in* Π **do**
  $T_\pi \leftarrow QLearning(\pi, \mathcal{D}, \mathcal{E}, style_{ref})$;
  $T_{tot} \leftarrow T_\pi \cup T_{tot}$
end
$model \leftarrow train_{LCBOW}(T_{tot})$;

Algorithm 2: Extractor model algorithm

In the context of this work, we have an important number of alternatives to enhance our dataset, such as replacing variable names by synonyms, modifying function names (e.g., from `set_password()` to `os.setPass()`), or replacing `'[]'` with `'()'`. Since there is no clear algorithm to choose which actions (or combination of actions) will output the best suited dataset for the training phase, we consider the Q-learning algorithm (Watkins and Dayan, 1992).

4.1.3 Q-learning

Q-learning algorithm is a reinforcement learning algorithm, where an agent learns, through interactions with its environment, actions to take to maximize a reward. A classic example is a game of chess: the Q-learning algorithm will compute the list of moves the player needs to perform to win the game (to checkmate her opponent).

Similarly, in the data augmentation process, some actions can be applied to the collected data, such as

[9] The complete list of the features we consider for code stylometry can be found in the Appendix

modifying variable names (like in Example A), selecting different functions names, or considering object-oriented programming patterns. Since different combinations of actions lead to different datasets, it will also lead to different code stylometry vectors. The goal for data augmentation is to converge to a particular code stylometry of the transformed dataset, called *reference stylometry*.

We define three primitives to build the Q-learning algorithm, that we show in Algorithm 1.

- $style \leftarrow choose\_actions(\pi, \mathcal{D})$: The agent can choose a combination of actions she intends to perform on data $\mathcal{D}$ (collected data from an empirical study) for a given pattern π. These actions produce a new dataset, from which we can compute the resulting stylometry *style*. In this paper, we consider a list of 28 programming patterns[10]
- $similarity(style, style_{ref})$: The similarity function computes the cosine distance between the current stylometry and the reference stylometry (computed through extracts $\mathcal{E}$). The output corresponds to the reward (which we want to maximize)
- $update\_choices(reward_{sim})$: Based on the reward, the Q-learning algorithm will update the available choices of actions. This update is ruled by the Bellman equation (Bellman, 1957).

After several iterations of the algorithm, the Q-learning will apply the optimal choices of combinations of actions, to compute the training dataset for a given programming pattern T_π. The stopping condition can be time-based (e.g., maximum number of iterations) or a threshold reward value.

4.2 Extractor

The main objective for the Extractor is to remove unnecessary elements in a code snippet, taking as inputs a list of discoveries (corresponding to the output of the Path model), and it outputs, for each code snippet, a tuple containing a variable name and a variable value. If no tuple can be found in a code snippet, then it is automatically discarded (because no variable assignment has been found).

The training data for the Extractor is obtained through the augmentation of collected data $\mathcal{D}$ from GitHub, like variable names, function names, etc., used for variable assignments. Data augmentation is performed before the training phase. Simultaneously, the Extractor has access to a collection of code extracts $\mathcal{E}$; these extracts are not discoveries, but simply randomly chosen pieces of code, from which we can compute a reference code stylometry. Hence, an Extractor model can be trained for every developer (because each of them has a different code stylometry) or for a group of developers (considering their global code stylometry).

The training phase for the Extractor is shown in Algorithm 2. For each collected programming pattern π, we apply the Q-learning algorithm, while considering the stylometry of the developer ($style_{ref}$) as the reference stylometry. We obtain the training data T_{tot} on which a LCBOW model is trained to obtain the Extractor.

4.3 Classifier

The Classifier takes as input a list of tuples, each of them containing a variable name and a variable value (which corresponds to the output of the Extractor) and classifies the tuple as a leak or as a Code snippet false positive. The training data for the Classifier is different from the training data of the Extractor. We retrieved an open-source list of the most commonly used passwords[11] (used by multiple tools when attempting to guess credentials for a given targeted service), and collected (through an empirical study) a list of commonly used variable names (such as *root, admin, pass*, etc.). The design of the Classifier is similar to the design of the Path model, with a LCBOW model. The Classifier achieves on this dataset of (variable name, variable value) 98% of accuracy.

5 Similarity model

In the manual review phase, a user can classify a potential leak containing a code snippet s_j as false positive. We assume that we have the set of LCBOW word representations of code snippets of discoveries $\{LCBOW(s_1), ...LCBOW(s_k)\}$. To reduce the workload of a human reviewer, we introduce a Similarity model, taking the code snippet $LCBOW(s_j)$ as input and automatically classifying discoveries containing similar code snippets as false positives, denoted $\{LCBOW(s_i), ...LCBOW(s_{k'})\}$ with $0 \leq k' \leq k$.

Definition. *Let* η *be a similarity threshold. Two code snippets LCBOW representation* $LCBOW(s_i)$ *and* $LCBOW(s_j)$ *are similar if* $cosine(LCBOW(s_i), LCBOW(s_j)) \leq \eta$

A similarity threshold $\eta = 1$ means that, for a flagged discovery $\{f_i, s_j\}$, the Similarity model flags

[10] see Appendix. The list of actions can also be found in the Appendix.

[11] https://github.com/danielmiessler/SecLists/tree/master/Passwords/Common-Credentials

Repository type	Discoveries	File path FP	Code snippet FP	Total FP
public	13.6	9.35 (69%)	1.79 (13%)	11.11 (82%)
proprietary	0.259	0.091 (35%)	0.064 (25%)	0.155 (60%)

Table 1: FP by models (in millions of discoveries)

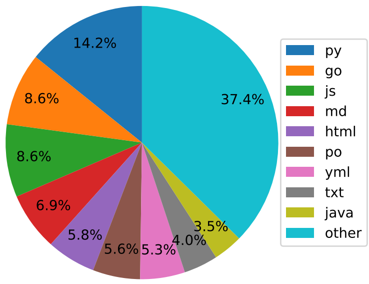

Figure 2: Most common files containing secrets

all the duplicates of the code snippets. The impact of η is analyzed in Section 6.3.

6 Experiments

In this section, we present an evaluation of our solution, divided into three major parts. Firstly (in Section 6.1), we evaluate the rate of false positive data on the output of the Regex Scanner (as proposed by the solutions in the literature). With this goal, we scan a dataset of 1000 repositories from the public GitHub (i.e., github.com), and 300 repositories from a GitHub-like code versioning platform owned by a private company. In the remainder of this section, we refer to github.com as *public github* and to the repositories publicly available on this platform as *public repositories*, while we refer to the privately owned GitHub platform as *proprietary github* and to its repositories as *proprietary repositories*. Next, in Section 6.2, we manually assess the false positive rate as well as the false negative rate induced by the machine learning models, and we show that the false negative rate is negligible (meaning that no leak on the output of the Regex Scanner is discarded by the models). Finally, in Section 6.3 we estimate the impact of the data augmentation algorithm parameters on the precision of our solution.

The tool that we have developed, and that we have used for the experimental evaluation of our proposal, is available open source together with the machine learning models[12].

classification		machine learning models	
		potential leak	non critical data
manual	leak	20% (true positives)	1% (false negatives)
	non critical data	80% (false positives)	99% (true negatives)

Table 2: Manual assessment of 2000 discoveries

6.1 Regex Scanner false positive rate

For this experiment, we randomly selected and scanned 1000 public repositories on GitHub. The list of regular expressions used by the Regex Scanner can be found in the Appendix. Over 14 million discoveries have been found, with 13.6 million in 579 out of 1000 public GitHub repositories (58%) and 260k discoveries in 268 out of 300 proprietary repositories (89%). Our discoveries cover more than 30 programming languages, and represent more than 300 file types. Figure 2 shows the 10 most common file extensions containing leaks in our dataset. The number of contributors and the sizes of the repositories have been chosen equally distributed.

We notice that API keys are still widely published in open-source projects, as shown also in (Meli et al., 2019). Nevertheless, they do not represent the majority of the discoveries. Indeed, in our study, we notice a more important number of passwords giving access to local and remote databases, or to e-mail accounts. We observe that the vast majority of these passwords is not critical (i.e., false positives), which seriously increases the load of a developer to review each of them manually. These passwords are mostly undetectable by traditional scanning tools, but they are still easy to find for someone using a simple search tool in the commit message (with keywords such as *remove credentials*, *delete password*, etc.). We found many passwords that we suppose to be real (even if we cannot have the certainty of this, since we are not allowed to test these passwords). This is a very important concern not only because passwords are still widely reused (Pearman et al., 2019), but also because two-factor authentication is still scarcely known (and thus activated) (Milka, 2018; Center, 2019), and scarcely supported by services (Bursztein,).

To summarize, the vast majority of the discoveries detected with the Regex Scanner consists of false positive data. In order to reduce the false positive rate, as described in section 3, we apply the Path model and the Snippet models sequentially, and finally evaluate the newly obtained false positive rates. As shown in Table 1, the Path model classifies almost 70% of the discoveries as false positives in the public dataset. This score is halved with the proprietary dataset. Together with the Snippets models, we see that up to

[12] https://github.com/SAP/credential-digger

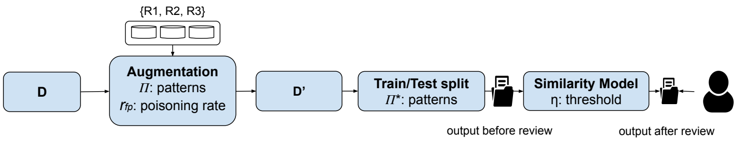

Figure 3: Data augmentation on $\mathcal{D}$ to assess the performance of the Extractor with the train/test split technique

82% of the discoveries are classified as false positives without a human intervention.

6.2 Models false negatives

In order to assess the behavior of our models, we decided to perform a manual review of a limited number of discoveries (we recall that the Regex Scanner found 14 millions discoveries in the previous experiment). To do so, we consider a sampling method, randomly selecting 100 discoveries classified as potential leaks by the models and 100 discoveries classified as non critical data by the models, and we manually analyze each of them. We repeat this process 10 times (covering 0.01% of all the discoveries from the previous experiment). The results are shown in Table 2.It is visible that 99% of the discoveries classified as non critical data by the models are real-life true negatives. The remaining percentage (corresponding to false negatives) corresponds to edge cases, where developers inserted (seemingly) real credentials in dummy files. Thus, in the scope of our study, we can state that the unclassified leak rate is negligible. Given the discoveries classified as potential leaks, 80% of them are non critical (i.e., false positives non detected by the models), and 20% of them are actual leaks (i.e., true positives). If we project the results of this manual assessment to the complete list of discoveries, we can assume that (i) our models do not create false negatives and (ii) they provide an efficient reduction of the false positive data on the output of the Regex Scanner.

6.3 Models false positives

In the previous section, we notice that it is difficult to assess the false positive rate of the Snippet Models (especially the Extractor) with precise metrics since we do not have a ground truth for the majority of the leaks detected in open-source repositories. In the previous section, we had to consider other evaluation techniques (e.g., sampling) to evaluate the false positive rate in real-life conditions, or to manually label the discoveries, which represents an important workload. Furthermore, due to the limited size of labeled data that we manage to collect, we cannot apply the *train/test* split (Bronshtein, 2017) technique in order

Repository	Language	Contributors
rhiever/MarkovNetwork[13]	Python	3
bradtraversy/vanillawebprojects[14]	Javascript	8
AGWA/git-crypt[15]	C++	15

Table 3: Description of the three repositories

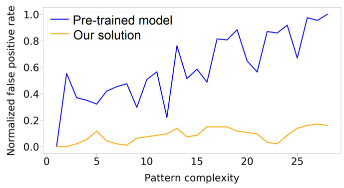

Figure 4: Normalized FP rate by pattern for the pre-trained model and the Extractor trained with Q-learning

to evaluate our models on them. The train/test split technique is a well-known process to assess the validity of machine learning models, splitting the data into two distinct subsets: training data (on which we will fit our model) and testing data (on which we will evaluate our model). As mentioned before, the size of the collected labeled data $\mathcal{D}$ is too small to accurately evaluate the Extractor using the train/split technique.

Nevertheless, Section 4.2 shows that we can apply data augmentation techniques to expand the size of our training dataset, as long as we have a reference stylometry. Hence, the goal of this section is to evaluate the false positive rate induced by the Extractor itself (independently from the false positives induced by the Regex Scanner) on several open-source repositories, with a train/test split approach commonly used in supervised learning on an data augmented dataset. To achieve this goal, we consider three different repositories $\{R_1, R_2, R_3\}$, each of them containing source code written in different programming languages by different developers (and different code stylometries) as shown in Table 3. The main idea is to use the stylometries of these repositories to obtain an augmented dataset where the train/test split technique is possible, and to see the impact of the augmentation process on accuracy metrics such as precision or recall.

Situation	Precision	Recall
Pre-trained Extractor	55.56	100
Extractor with Q-learning	71.66	100
Extractor + Similarity model	74.52	99.71

Table 4: Impact of data augmentation with $\Pi^*_{0.80}$

6.3.1 Train/test split

We propose an experiment to evaluate the false positive rate on the Snippet Models with respect to $R \in \{R_1, R_2, R_3\}$, as illustrated in Figure 3.

- To begin with, we obtain an augmented dataset $\mathcal{D}'$ from the collected data $\mathcal{D}$, the patterns Π, and the extracts $\mathcal{E}$ of the repository R. We can select the leak percentage in $\mathcal{D}'$ with parameter r_{fp} ($r_{fp} = 0.5$ corresponds to a balanced dataset).
- Next, we split $\mathcal{D}'$ into a training and a testing dataset. We also perform the split on the patterns to obtain $\Pi^* \subset \Pi$: this ensures that the patterns used to perform the training (Π^*) are different from the patterns used to do data augmentation (Π).
- Finally, after the training phase, we compute metrics such as precision, recall, and f_1 score on the testing dataset. A manual reviewer manually flags the false positives, and she is assisted by the similarity model (with threshold η). We consider that the manual reviewer flags 0.1% of the discoveries.

There are mainly three hyper-parameters that have an impact on the precision of the Extractor: r_{fp} (the percentage of leaks over the size of $\mathcal{D}'$), how we choose the subset of patterns Π^* used to train the Extractor in the train/test phase, and the similarity threshold η from the similarity model. In the following subsections, we show the effects of these three hyper-parameters on the accuracy of our solution. We propose to first study the impact of the Q-learning algorithm on the precision of the Extractor, and show that this technique significantly increases the precision (thus decreasing the false positive rate). We further evaluate the impact of the three hyper-parameters on the precision, recall and false positive rate of our approach.

6.3.2 Pre-trained model

To begin with, we study the impact of the data augmentation process. On the one hand, we have an Extractor model, pre-trained on the data we collected without any data augmentation process (called pre-trained Extractor). On the other hand, we have an Extractor model, trained with the Q-learning algorithm for data augmentation where $\Pi^* = \Pi^*_{0.8}$ (corresponding to a set of patterns, randomly chosen including 80% of the patterns in Π). In Table 4, we see the impact of the Q-learning algorithm, with a high precision score as opposed to the pre-trained model (it increases from 55.56% to 71.66%).

A recall close to 100% means that we detect almost all the leaks. However, when the user flags a discovery as false positive, the similarity model (with threshold parameter η) may classify an actual leak as non relevant (i.e., it may cause a false negative). If we select $\eta = 1$, we reach a recall of 100% but without any significant improvement of the precision score. To fix the recall drop, a possible remediation is to inform the user on what discoveries have been classified as non relevant by the similarity model, so that she can check whether or not an actual leak has been wrongly classified (it will improve the recall score, but will increase the manual workload also).

We also compare the precision score per pattern. Each pattern has a complexity value associated with its index (i.e., the pattern with index 1 is the simplest, and the pattern with index 28 is the most complex). As shown in Figure 4, we can observe a linear relationship between the pattern complexity and the false positive rate when we use the pre-trained Extractor (which seems natural for a global model, since more complex patterns are harder to detect, leading to more false positives). With the Extractor trained with the Q-learning algorithm, the false positive rate is independent from the complexity of the pattern (which means that no particular pattern will lead a higher false positive rate).

6.3.3 Extractor with Q-learning

In this section, we solely consider the Extractor trained with the Q-learning algorithm (excluding the pre-trained model), by presenting the impact of r_{fp} and Π^* on the false positive rate.

Impact of r_{fp}: First, we analyze the impact r_{fp} on the false positive rate in three different situations, i.e., with $r_{fp} = 0.5$ (balanced situation between leaks and false positive), $r_{fp} = 0.2$, and $r_{fp} = 0.05$ (unbalanced situation where leaks are scarce), while fixing the parameter Π^*. We present the results in Table 5a. We observe that:

- in a balanced situation, we achieve a false positive rate of 5.97%, considerably reducing the part of false positive data in the discoveries;
- in unbalanced situations, the results show that we manage an acceptable rate of false positives, below 12%.

Impact of Π^*: Next, we analyze the impact of the choice of Π^* on the false positive rate in several situations, while fixing the poisoning rate $r_{fp} = 0.5$. As mentioned before, each pattern has a complexity value. Thus, we can define the complexity of a set of patterns Π as the average complexity of these patterns (therefore, in our experiments, the complexity of our set of 28 patterns Π, is equal to 14.5). Let $\Pi^*_{0.5}$ be a set of patterns representing 50% of the set of patterns

Situation	$\Pi^*_{0.80}$			$\Pi^*_{0.5}$		
	Precision	Recall	F1	Precision	Recall	F1
Before review	89.33	100	**94.36**	71.66	100	**84.89**
After review	89.69	99.96	94.55	74.52	99.71	85.30

(a) Impact on the manual review on the metrics

Situation	$\Pi^* = \Pi^*_{0.80}$		
	$r_{fp} = 0.5$	$r_{fp} = 0.20$	$r_{fp} = 0.05$
FP Rate	**5.97**	12.09	11.03

Situation	$r_{fp} = 0.5$			
	$\Pi^*_{complex}$	$\Pi^*_{0.5}$	Π^*_{simple}	$\Pi^*_{0.25}$
FP Rate	9.36	**18.35**	31.99	27.86

(b) Impact of Π^s and r_{fp} on the FP rate

Table 5: Poisoning experiments. Results in bold in (a) correspond to experiments with identical parameters in (b)

in Π, with an equivalent pattern complexity. Table 5b presents the results.

For $\Pi^* = \Pi^*_{0.5}$, we obtain a false positive rate of 18.35% in this setting. Compared to $\Pi^*_{0.8}$, a 30% decrease in the number of patterns leads to a 15% increase of the false positive, proving that our approach is able to generalize unseen patterns while preserving low false positive rate. We also consider $\Pi^*_{0.25}$ corresponding to 25% of the patterns with an equivalent overall pattern complexity.

Furthermore, we decide to study set of patterns without conserving the overall pattern complexity, splitting Π^* into two sets $\Pi^* = \Pi^*_{simple} \cup \Pi^*_{complex}$, corresponding respectively to the first 14 patterns and to the last 14 patterns. The results of the experiment with Π^*_{simple}, $\Pi^*_{complex}$, Π^*_{simple} and $\Pi^*_{0.25}$ are also presented in Table 5b.

Although $\Pi^*_{0.5}$, Π^*_{simple} and $\Pi^*_{complex}$ contain the same number of programming patterns, the pattern complexity distribution greatly impacts the false positive rate. We reach an acceptable false positive rate with only 25% of the patterns, but more equally distributed in complexity. It is worth noting that the highest score is reached with the $\Pi_{complex}$ pattern set, with results close to the full pattern experiment. Indeed, as shown in Figure 4, the false positive rate per pattern is higher, on average, for complex patterns (i.e., with index above 14). Therefore, targeting only this class of patterns leads to a decrease of the global false positive rate.

With respect to Π^* and r_{fp}, we estimate the false positive rate induced by the Extractor between 6% and 32%. In Section 6.1, we showed that more than 80% of the false positive data (induced by the Regex Scanner) has already been discarded. Overall, we showed that the false positive rate of the whole solution (including the Regex Scanner and the machine learning models) represents between 1% and 6% of the output.

7 Related work

7.1 Research work

An important amount of work targets GitHub open-source projects, from vulnerability detection(Russell et al., 2018) to sentiment analysis (Guzman et al., 2014). Empirical studies also provide a more global overview of the data on GitHub (Kalliamvakou et al., 2014) and how to facilitate its access (Gousios et al., 2014).

With the advent of machine learning techniques in the researchers' toolkits, approaches for source code representation have been developed, proposing a language-agnostic representation of source code (Alon et al., 2018; Gelman et al., 2018). Leak detection can be also considered as a branch of data mining or code search tasks. Works on evaluating the state of the semantic code search (Husain et al., 2019), as well as works on deep learning applications for code search (Cambronero et al., 2019), emphasize the need for developing machine learning techniques for source code analysis. However, these previous works have different purposes from ours, especially regarding the criticality of the datasets, and they consider token-based representations (so language dependent) as opposed to our purely semantic approach.

Leak detection is connected to malware detection (Dahl et al., 2013; Pendlebury et al., 2019) addressing similar issues to solve privacy concerns in realistic settings, where the testing samples are not representative of real world distributions. Contrary to malware classification, we do not have a reference dataset to benchmark language specific approaches.

Code transformations based on stylometry have been tackled by other works (Long et al., 2017; Quiring et al., 2019). In particular, in (Quiring et al., 2019), the authors, given a list of code extracts $\{e_1, ..e_n\}$ developed by a list of developers $\{D_1, ...D_m\}$ and an authorship attribution classifier, transform each e_i to fool the classifier concerning the authorship of e_i. To do so, they use a Monte-Carlo Tree Search algorithm to compute the most optimal code transformations to perform the authorship attribution attack. In our work, we leverage the ideas

Category	Tool	Scanning process						User experience						Adoption		
		Regex	Entropy check	Heuristics	Path FP detection	Password detection	Machine learning	Free	User interface	Open-source	Repository management	Scan of private repositories	Authentication not required	Community	Scalability	Regular updates
Known algorithms	TruffleHog	●	◐	-	-	-	-	●	-	●	-	◐	●	●	●	●
	Git-secrets	●	●	●	-	-	-	●	●	●	-	-	-	◐	●	◐
	Gitrob	●	●	●	-	-	-	●	●	●	-	-	-	◐	●	◐
	(Meli et al., 2019)	●	●	●	◐	-	-	-	-	-	-	-	-	-	-	-
Commercial offers	GitGuardian	●	-	●	-	-	-	◐	●	-	●	●	-	◐	●	●
	Nighfall AI	●	-	●	-	-	●	-	●	-	●	●	-	-	●	-
Our approach		●	-	●	●	●	●	●	●	●	●	●	●	-	●	●

● = provides property; ◐ = partially provides property; - = does not provide property;

Figure 5: Comparison of available tools

developed in (Quiring et al., 2019) to perform our own code transformation to do data augmentation. We choose Temporal Difference (TD) learning over Monte-Carlo, due to its incremental aspect. Indeed, in the description of the Q-learning algorithm, there is a stopping condition in order to obtain the augmented data, whereas Monte-Carlo algorithms have to be run completely. We suppose that in our case the conditions for the convergence of TD algorithms are satisfied (Van Hasselt et al., 2018).

Two different studies have considered the state of data leakage in GitHub repositories. (Sinha et al., 2015) focuses on API Keys detection but the scope of their study is limited to Java files, and the remediation techniques are mainly composed of heuristics. In a more recent work (Meli et al., 2019), Meli et al. propose a study on the leak of API Keys, focusing on possible correlations between multiple features in a GitHub project to find root causes. Nevertheless, this work is limited to API Keys: it is explicitly stated that their analysis does not apply to passwords. Moreover, the focus of their study was on the characteristics of true secrets, with indications on contributors or persistence of secrets. Our focus dwells instead, on the false positive data, since it represents the vast majority of discoveries of any open-source project. Finally, they provide an extensive study of GitHub API Keys leaks by scanning an important number of repositories, close to 700,000. In our work, we chose not to conduct our GitHub leak status study with such a high number of repositories, because it would have led to a tremendous number of false positive discoveries, which would not have been possible to process.

7.2 Comparison with other tools

Since the problem of leak detection in public open-source projects is not new, open-source tools such as GitHub Token Scanning[16], GitLeaks[17] or S3Scanner [18] have been developed to tackle it alongside commercial platforms, namely GitGuardian and Gamma. However, to the best of our knowledge, there is no open-source tool which scans GitHub repositories and applies machine learning to decrease the false positive rate. Therefore, since the existing tools do not work in the same paradigm as our approach (not considering passwords, for instance), we do not provide a comparison of metrics to avoid any bias. Still, we can compare our approach with several tools we selected.

TruffleHog[19] is a very popular (5k stars on GitHub, at the time of writing) and open-source scanning tool. The user has to provide her own set of regular expressions to the tool in order to detect possible leaks. This tool does not use machine learning, and it is mostly targeted to detect API Keys. Its main advantage is surely its simplicity for developers. Similar tools have emerged with the same characteristics, such as *Gitrob*[20] and *git-secrets*[21].

GitGuardian[22] is a tool provided by the namesake company founded in 2016 and specialized in detection of leaks in open-source resources. Alongside their commercial offer, they provide free services to scan one's own GitHub repositories. They claim their tool is *machine learning powered* and that they can identify *more than 200 API Keys patterns*, but they do not mention passwords.

TruffleHog and its variants aim to be a strong baseline for scanning tools. For example, in (Meli et al., 2019) authors offer improvements to its core algorithm. Various heuristics can be implemented

[16]https://preview.tinyurl.com/ycnllvfd
[17]https://github.com/zricethezav/gitleaks
[18]https://github.com/sa7mon/S3Scanner
[19]https://github.com/dxa4481/truffleHog
[20]https://github.com/michenriksen/gitrob
[21]https://github.com/awslabs/git-secrets
[22]https://www.gitguardian.com/

to improve the accuracy of the tool, such as *entropy check*: if a string has high entropy, which means it consists of seemingly random characters, the probability that this string is an API Key is high. We perform several manual tests on the GitGuardian platform on various API Keys patterns and on plaintext passwords in order to understand the possibilities and the limitations of such a tool. According to our tests, the platform is not able to detect plaintext passwords, and it only detects a reduced sample of API Keys, excluding big API Keys providers such as Facebook and Paypal. We only tested the free version of GitGuardian, so it might be possible that the full capabilities of the platform are only enabled in the commercial offer. Another commercial tool called Nightfall AI[23] (formerly known as Watchtower) offers the same services, but no free version is available to test the platform.

We compared several tools, on different criteria, and show our results in Figure 5. For each scanning tool, we compare what techniques are used, and if there is any false positive reduction. The open-source tools do not perform false positive reduction (since most of them do not detect passwords), favoring the usage of heuristics which need less computational power. However, most of the heuristics are not adapted to all use cases, so the developer has to manually configure the tool without efficiency guarantees. In our approach, we choose to adapt the scanning process to each developer, thus the fine-tuning is performed by the Leak Generator rather than the user herself. The *continuous training* parameter is the ability for the tool to re-train the machine learning models when the user flags a discovery, so to improve future classifications. Open-source solutions are more focused on single use cases, offering limited interactions with the developers. Our approach, similar to the GitGuardian platform, is to improve the accuracy while reviewing, decreasing the monitoring time. The user experience is also a key point in order to be used efficiently. The price could represent an important barrier for small companies willing to protect themselves, encouraging bad development habits. Commercial products provide a user interface, making the tool more accessible to developers, and even to non-technical people. Since the origin of a leak does not depend on the level of expertise of the developers (Meli et al., 2019), tools with a user interface could be easily used also by beginners to protect their code.

[23] https://www.nightfall.ai/

8 Privacy concerns disclosure

In this paper, we deal with critical data, which could harm users' privacy in case they were used for malicious purposes. Thus, we need to discuss privacy issues in the scope of our research. First, with regard to the experiment shown in Section 6.1, public repositories represent open-source data found in public websites (in particular, github.com), while the access to the proprietary platform has been granted by the company that owns all the rights on it. In both cases, no intrusion or hacking techniques were used to obtain data. We ensure that data collected are only accessible to our working team, for analysis purposes only, and that sensitive information have not been used to train predictive models. The training of the models, together with the evaluation of our approach shown in Section 6.3, has been achieved using sanitized data. Furthermore, we did not attempt to use any actual leaks we discovered to verify their authenticity, and we tried, when possible, to notify the developer responsible for publishing credentials. Finally, all the real data we collected have been deleted after the experimental evaluation of our approach.

9 Conclusion

We proposed an approach to detect data leaks in open-source projects with a low false positive rate. Our solution improves classic regular expression scanning methods by leveraging machine models, filtering an important number of false positives. Through our series of experiments, we show that our approach outperforms classic scanning methods, produces a negligible amount of undetected leaks and results in a false positive rate of at most 6% of the output data.

10 Acknowledgments

We would like to thank Sabrina Kall for her help during the writing of this paper. We also would like to thank the Institue for artificial intelligence 3IA and the Councel of Industrial Resarch for Artificial Intelligence ICAIR for their support.

REFERENCES

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. (2018). code2vec: Learning distributed representations of code.

Bellman, R. (1957). *Dynamic Programming*.

Bronshtein, A. (2017). Train/test split and cross validation in python. *Understanding Machine Learning*.

Bursztein, E. The bleak picture of two-factor authentication adoption in the wild. https://tinyurl.com/yctk4aja.

Cambronero, J., Li, H., Kim, S., Sen, K., and Chandra, S. (2019). When deep learning met code search.

Center, P. R. (2019). Americans and digital knowledge. https://tinyurl.com/y8ftudoh.

Dahl, G. E., Stokes, J. W., Deng, L., and Yu, D. (2013). Large-scale malware classification using random projections and neural networks. In *ICASSP*.

Gelman, B., Hoyle, B., Moore, J., Saxe, J., and Slater, D. (2018). A language-agnostic model for semantic source code labeling. In *MASES*.

Gousios, G., Vasilescu, B., Serebrenik, A., and Zaidman, A. (2014). Lean ghtorrent: Github data on demand. In *MSR*, pages 384–387.

Guzman, E., Azócar, D., and Li, Y. (2014). Sentiment analysis of commit comments in github: an empirical study. In *MSR*, pages 352–355.

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. (2019). Codesearchnet challenge: Evaluating the state of semantic code search.

Joulin, A., Grave, E., Bojanowski, P., and Mikolov, T. (2016). Bag of tricks for efficient text classification.

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., and Damian, D. (2014). The promises and perils of mining github. In *MSR*.

Long, F., Amidon, P., and Rinard, M. (2017). Automatic inference of code transforms for patch generation. In *FSE*, pages 727–739.

Ma, S., Sun, X., Wang, Y., and Lin, J. (2019). Bag-of-Words as target for neural machine translation.

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

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient estimation of word representations in vector space.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013b). Distributed representations of words and phrases and their compositionality. In *NIPS*.

Milka, G. (2018). Anatomy of account takeover. In *Proceedings of Enigma*.

MITRE (2019). 2019 cwe top 25 most dangerous software errors. https://tinyurl.com/y73xa6qk.

Pearman, S., Zhang, S. A., Bauer, L., Christin, N., and Cranor, L. F. (2019). Why people (don't) use password managers effectively. In *USENIX SOUPS*.

Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., and Cavallaro, L. (2019). TESSERACT: Eliminating experimental bias in malware classification across space and time. In *USENIX Security Symposium*, pages 729–746.

Quiring, E., Maier, A., and Rieck, K. (2019). Misleading authorship attribution of source code using adversarial learning.

Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., and McConley, M. (2018). Automated vulnerability detection in source code using deep representation learning. In *ICMLA*.

Shorten, C. and Khoshgoftaar, T. M. (2019). A survey on image data augmentation for deep learning. *Journal of Big Data*, 6:60.

Sinha, V. S., Saha, D., Dhoolia, P., Padhye, R., and Mani, S. (2015). Detecting and mitigating secret-key leaks in source code repositories. In *MSR*, pages 396–400.

Sun, X., Liu, X., Hu, J., and Zhu, J. (2014). Empirical studies on the nlp techniques for source code data preprocessing. In *EAST*, pages 32–39.

Van Hasselt, H., Doron, Y., Strub, F., Hessel, M., Sonnerat, N., and Modayil, J. (2018). Deep reinforcement learning and the deadly triad.

Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8:279–292.

APPENDIX

A list of 29 regular expression used in the Regex Scanner is presented in Table 7. We collected 15 API Keys patterns, 3 RSA Key patterns and 1 access token pattern from (Meli et al., 2019). In addition to these, we also used patterns from TruffleHog. We augmented this dataset with 2 ssh-related patterns, alongside 8 passwords (or keywords) patterns. We did not optimize our regular expressions, since we implemented the scanner with Hyperscan, i.e., a regular expression library offering integrated optimization.

In Table 6 and Table 8, we present respectively the list of possible transformations on source code and the list of programming patterns used for the data augmentation process. We group the actions by *class* of actions: identity action (no modification on the source code), actions expanding (or reducing) the input length, actions changing the hypothetical type of an input, and actions impacting the pattern complexity.

Actions
identity
longer_key
longer_function
longer_method
longer_object
smaller_key
smaller_function
smaller_method
smaller_object
change_type
more_complex_pattern
simpler_pattern

Table 6: Actions which could be applied to a source code extract

We present the list of features considered to compute the stylometry of an extract in Figure 6.

Features
Word occurrences in the code snippet
List of keywords in the code snippet
Number of total symbols
Average length in characters
Standard Deviation length in characters
Number of spaces
Ratio between number of spaces and number of characters
Occurrences of specific symbols (parentheses, brackets, etc.)

Figure 6: Features used to compute the stylometry vector

<table>
<tr><th>Type</th><th>Pattern</th><th>Source</th></tr>
<tr><td>RSA Private Key</td><td>-----BEGIN RSA PRIVATE KEY-----
[\r\n]+(?:\w+:.+)*[\s]*(?:[0-9a-zA-Z+=]
{64,76}[\r\n]+)+[0-9a-zA-Z+=]+[\r\n]+
-----END RSA PRIVATE KEY-----</td><td>Meli et. al</td></tr>
<tr><td>RSA EC Key</td><td>-----BEGIN EC PRIVATE KEY-----
[\r\n]+(?:\w+:.+)*[\s]*(?:[0-9a-zA-Z+=]
{64,76}[\r\n]+)+[0-9a-zA-Z+=]+[\r\n]+
-----END EC PRIVATE KEY-----</td><td>Meli et. al</td></tr>
<tr><td>RSA PGP Key</td><td>-----BEGIN PGP PRIVATE KEY BLOCK-----
[\r\n]+(?:\w+:.+)*[\s]*(?:[0-9a-zA-Z+=]
{64,76}[\r\n]+)+[0-9a-zA-Z+=]+[\r\n]+
-----END PGP PRIVATE KEY BLOCK-----</td><td>Meli et. al</td></tr>
<tr><td>Access token</td><td>((?:\? | \& | \" | \')(?:access_token)(?:\" | \')?\s*(?:= | :))</td><td>Meli et. al</td></tr>
<tr><td>Token</td><td>EAACEdEose0cBA[0-9A-Za-z]+</td><td>Meli et. al</td></tr>
<tr><td>Token</td><td>AIza[0-9A-Za-z\-_]{35}</td><td>Meli et. al</td></tr>
<tr><td>Token</td><td>[0-9]+-[0-9A-Za-z_]{32}\.apps\.googleusercontent\.com</td><td>Meli et. al</td></tr>
<tr><td>Token</td><td>sk_live_[0-9a-z]{32}</td><td>Meli et. al</td></tr>
<tr><td>Token</td><td>sk_live_[0-9a-zA-Z]{24}</td><td>Meli et. al</td></tr>
<tr><td>Token</td><td>rk_live_[0-9a-zA-Z]{24}</td><td>Meli et. al</td></tr>
<tr><td>Token</td><td>sq0atp-[0-9A-Za-z\-_]{22}</td><td>Meli et. al</td></tr>
<tr><td>Token</td><td>sq0csp-[0-9A-Za-z\-_]{43}</td><td>Meli et. al</td></tr>
<tr><td>Token</td><td>access_token\$production\$[0-9a-z]{16}\$[0-9a-f]{32}</td><td>Meli et. al</td></tr>
<tr><td>Token</td><td>SK[0-9a-fA-F]{32}</td><td>Meli et. al</td></tr>
<tr><td>Token</td><td>key-[0-9a-zA-Z]{32}</td><td>Meli et. al</td></tr>
<tr><td>Token</td><td>AKIA[0-9A-Z]{16}</td><td>Meli et. al</td></tr>
<tr><td>Token</td><td>(xox[p|b|o|a]-[0-9]{12}-[0-9]{12}-[0-9]{12}-[a-z0-9]{32})</td><td>TruffleHog</td></tr>
<tr><td>Token</td><td>https://hooks.slack.com/services/T[a-zA-Z0-9_]{8}
/B[a-zA-Z0-9_]{8}/[a-zA-Z0-9_]{24}</td><td>TruffleHog</td></tr>
<tr><td>Key word</td><td>sshpass</td><td>Our contribution</td></tr>
<tr><td>Key word</td><td>sshpass -p.*['|\"]</td><td>Our contribution</td></tr>
<tr><td>Password</td><td>(root|admin|private_key_id|client_email|client_id|token_uri)
\s*((?:=|:|->|<-|=>|<=|==|<<))</td><td>Our contribution</td></tr>
<tr><td>Password</td><td>(password|new_pasword|username
\s*(?:=|:|->|<-|=>|<=|==|<<))</td><td>Our contribution</td></tr>
<tr><td>Password</td><td>(user|email|User|Pwd|UserName|user_name
\s*(?:=|:|->|<-|=>|<=|==|<<))</td><td>Our contribution</td></tr>
<tr><td>Password</td><td>(access_token|access_token_secret|consumer_key |consumer_secret
\s*(?:=|:|->|<-|=>|<=|==|<<))</td><td>Our contribution</td></tr>
<tr><td>Password</td><td>(FACEBOOK_APP_ID|ANDROID_GOOGLE_CLIENT_ID)
\s*(?:=|:|->|<-|=>|<=|==|<<))</td><td>Our contribution</td></tr>
<tr><td>Password</td><td>(authTokenToken|oauthToken|CODECOV_TOKEN
\s*(?:=|:|->|<-|=>|<=|==|<<))</td><td>Our contribution</td></tr>
<tr><td>Password</td><td>(IOS_GOOGLE_CLIENT_ID
\s*(?:=|:|->|<-|=>|<=|==|«))</td><td>Our contribution</td></tr>
<tr><td>Password</td><td>(sk_live|rk_live
\s*(?:=|:|->|<-|=>|<=|==|<<))</td><td>Our contribution</td></tr>
</table>

Table 7: Regular expression patterns

Id	Pattern
1	key = "value"
2	key[’value’]
3	key « object.method("value")
4	key.method(’value’)
5	Object.key = ’value@gmail.com’
6	key = type_1 function Password(’value’)
7	public type_1 type_2 int key = ’value’
8	key => method(’value’)
9	type_1 key = ’value’
10	Object[’key’] = ’value’
11	method.key : "value"
12	object: {email: user.email, key: ’value’}
13	key = setter(’value’)
14	key = os.env(’value’)
15	Object.method :key => ’value’"
16	key = Object.function(’value’)
17	User.function(email: ’name@gmail.com’, key: ’value’)
18	User.when(key.method_1()).method_2(’value’)
19	key.function().method_1(’value’)
20	type_1 key = Object.function_1(’value’)
21	method(’key’=>’value’)
22	public type_1 key { method_1 { method_2 ’value’ } }
23	private type_1 function_1 (type_1 key, type_2 password=’value’)
24	protected type_1 key = method(’value’)
25	type_1 key = method_1() credentials: ’value’.function_1()
26	type_1 key = function_1(method_1(type_2 credentials = ’value’))
27	Object_1.method_1(type_1 Object_2.key = Object_1.method_2(’value’))
28	type_1 Object_1 = Object_2.method(type_2 key_1=’value_1, type_3 key_2=’value_2’)

Table 8: Programming patterns used for the data augmentation process