POLITECNICO DI TORINO

Master's Degree Course in Computer Engineering

Master's Degree Thesis

Machine Learning based credential code scanner

Relatori
prof. Cataldo Basile
prof. Antonio Lioy

Candidato
Carlo Maria Negri

Supervisore Aziendale
Dr. Slim Trabelsi SAP LABs France

Anno accademico 2019-2020

Acknowledgements

A tutte le persone che hanno creduto in me,
che sono sempre state convinte potessi farcela.
Grazie.

Contents

List of Tables 8

List of Figures 9

1 Introduction 13

2 Background 17

2.1 Machine Learning introduction 17

2.2 Natural Language Processing introduction 21

2.2.1 NLP and Machine Learning 22

2.3 Text representation 23

2.3.1 Bag-of-Words (BoW) 24

2.3.2 Bag-of-N-grams 24

2.3.3 TF-IDF 25

2.3.4 Word Embedding 26

2.3.5 FastText for text representation 28

2.4 Text classification models 29

2.4.1 State of the art for text classification 29

2.4.2 FastText for classification 31

2.5 Performance Metrics 33

3 Secrets leakage problem 36
3.1 Definitions 37
3.2 Best practices in secrets management 38
3.3 State of the art 39
3.3.1 How bad can it Git? 39
3.3.2 Open-source existing tools 40
3.3.3 Proprietary existing tools 41
3.4 Problem approach 42
3.5 Machine Learning choice 44

4 Architecture design and high level components 46
4.1 High level design 46
4.2 GUI 47
4.3 Web Application Server 50
4.4 Database 50

5 GitHub Scanner 52
5.1 Hyperscan 52
5.2 Scan API 53
5.3 Scanning process 54

6 Path Model 57
6.1 Model idea 57
6.2 Path Analysis 58
6.3 Model building 62

7 Snippet Model 64
7.1 Model idea 64
7.2 Dataset creation 66
7.2.1 Extractor dataset 67
7.2.2 Classifier dataset 68

7.3 Data pre-processing 69
7.4 Model training 70
7.5 Model integration 70
7.6 Manual review 70

8 Evaluation 72
8.1 GitHub credentials leaks status 72
8.2 Leak Classifier evaluation 75
8.2.1 Poisoned repository generation 75
8.2.2 Evaluation 76
8.2.3 Discussion 76
8.3 Comparison with other tools 77

9 Conclusion 79
9.1 Further improvements 80

A Regular expressions 81

B Patterns 83

Bibliography 84

List of Tables

6.1 Comparison between tested architecture for building the classification models. 62

8.1 False positive classification in repositories. 74

8.2 Manual assessment of 200 discoveries after models classifications . . 74

8.3 Situation 1 evaluation metrics. 76

8.4 Situation 2 evaluation metrics. 76

8.5 Comparison of available tools 78

A.1 List of regular expressions used by the Scanner 82

B.1 List of pattern used in Snippet Model 83

List of Figures

2.1 Structure of an artificial neural network. 19
2.2 Structure of a neuron. 20
2.3 Relationship between AI, ML, NNs, DL. 21
2.4 Relationship between ML, DL and NLP. 23
2.5 Bag-of-Word example. 24
2.6 Bag-of-N-grams example with N = 2. 25
2.7 Word2Vec context window. 27
2.8 Skip-Gram and CBOW models. 28
2.9 RNN example. 31
2.10 ConvNet example. 31
2.11 FastText for classification example. 33
2.12 Confusion matrix. 34

3.1 Example of a discovery. 37
3.2 Examples of leaks and non critical data. 38
3.3 Static approach example. 43
3.4 Dynamic approach example. 43
3.5 Example of two different patterns for hardcoding secrets in Java. . . 44
3.6 Solution architecture. 45

4.1 Overall architecture. 47
4.2 SCCS Dashboard. 48
4.3 SCCS Add a rule. 48

4.4 SCCS Scan process. 49
4.5 SCCS Address a discovery. 49
4.6 Example of Python package usage. 51
4.7 Database structure. 51

5.1 GitHub network graph example. 54
5.2 GitHub diff example. 55
5.3 Example of discovery inserted into the db after the scanning process. 56

6.1 File extension distribution in paths. 58
6.2 Percentage of test, txt, and markdown files. 59
6.3 Word distribution in paths. 60
6.4 Pre-processed word distribution in all paths. 61
6.5 Bi-grams and tri-grams distribution in all paths. 61
6.6 Path model structure. 63

7.1 Key-value example in discoveries. 66
7.2 Classifier example. 66
7.3 Snippet model structure. 67
7.4 Example of training data for the Classifier. 68
7.5 Example of extractor training. 70
7.6 Example of similarity check. 71

8.1 Distribution of the most common extensions containing secrets . . . 73

Chapter 1

Introduction

Every day, millions of developers use source code repositories, such as GitHub, Bitbucket or GitLab, to manage and maintain their personal or company projects. These repositories also include a version control such as Git, which allows you to upload the project's source code and keep track of all changes that occur over time. Moreover, they are coding communities, so their usage can provide wide exposure for your project, and that's why developers use them also to let their works gain visibility. The more people review your project, the more popularity you get. Companies and organizations also have great advantages in publishing there their projects, especially in terms of:

1. *Collaboration:* every developer can give a contribution which can be accepted by the project owner;
2. *Adoption and remixing:* public projects can be used by everyone and changed according to your own needs;
3. *Transparency:* anyone can check a public project looking for errors or inconsistencies.

Thanks to their sharing capabilities, public source code repositories are nowadays a powerful tool, essential for every developer, but, unfortunately, that power comes with a risk. Looking through public code, it's easy to find things like:

- Hardcoded usernames, emails and passwords;
- Cloud services API tokens;
- Sensitive configuration files;

- Database connection strings;
- Private SSL or SSH keys;
- Uploaded Linux directories, with sensitive dotfiles;

In fact, there are many web and mobile based applications that interact with external services hosted by providers such as Facebook, Google, or Amazon through Web APIs. The mechanism for authentication between the application and the service is often through an API key or a pair of an API client identifier and a secret key. These API token, together with database connection strings, e-mails, and passwords, are often hardcoded by non-expert developers.

The problem of hardcoded or embedded credentials is a well-known problem in IT security, in fact, it ranks on the Common Weakness Enumeration (CWE) list of common security errors. Hardcoded secrets can be found everywhere, starting from software applications to hardware firmware and IoT devices. In 2016 it was exploited by Mirai malware to build a huge botnet of about 400.000 IoT devices, unbeknownst to most of their owners. Mirai-related botnets provoked one of the most disruptive DDOS attacks ever seen. Through a brute force attack, it was applied a table of 61 known hardcoded default usernames and passwords to attempt to access these IoT devices.

This problem became even more critical when these secrets, accidentally or intentionally, are made public as part of the repositories the developers push, and since these repositories are publicly available, anybody can find them. This leads to a huge and severe vulnerability that can be exploited even by inexperienced hackers. GitHub has a search API that can be used to search across all its repositories, and it happily delivers the secret key data. That's what happened in 2016 when data of 50.000 Uber drivers have been stolen by hackers that didn't have to do much hacking at all. They got into the company's database using login credentials they'd found on GitHub. Even more recently, in 2019, Canadian banking giant Scotiabank had to remove public GitHub repositories after being alerted about the exposition of some secrets in their source code.

Nowadays, there are many tools that can scan source code repositories looking for hardcoded secrets, GitHub itself has its own token scanning platform, but most of them just rely on regular expressions on well-known patterns leading to computational bottle-neck as well as an important number of false positive matches. These false positives are resulting from the lack of precision and granularity of regular expressions definition. We estimate that at least 80/90 % of the hits are false positives. Simultaneously, the outbreak of Machine Learning in cybersecurity,

especially Natural Language Processing, brought new tools to conduct a more precise analysis of security data.

The main goal of this thesis is to present and describe in detail SAP Credential Code Scanner (SCCS) platform, a scalable, public GitHub repositories scanning tool that outperforms the current state of the art. It analyzes discoveries and uses Machine Learning only to keep relevant secrets, automatically minimizing the rate of false positive data. Analyzing public repositories, we came up with a maximum reduction of 82% of the matches obtained through regular expression providing evidence of a more efficient way to find hardcoded secrets. Providing also a user interface, we are releasing a completely usable tool that developers can use to check if the code they published is secure.

The study was carried out starting from the analysis of the state of the art, understanding what the currently available tools that try to solve the problem are, and how they perform. Then we developed a platform that was able to scan GitHub repositories using regular expressions in order to look for hardcoded secrets. The value we thought to add was the use of Machine Learning for the reduction of false positive data. The key point was to create a filtering model, which takes as an input the matches, and classify them whether as real secrets or not critical data.

In the end, the tool was evaluated in three different situations. In the first one, about 1300 GitHub repositories were analyzed, showing a maximum false positives reduction of 82%. In the second one, the filtering model, due to a lack of labeled data, was tested on a controlled environment created injecting secrets in an already existing repository. In the third, the tool was compared with other available tools in terms of the provided features.

The main parts that compose the thesis are briefly outlined below:

- Chapter 2 *Background:* the knowledge required to understand the thesis. It mainly involves Machine Learning and Natural Language Processing techniques used for the filtering model;

- Chapter 3 *Secret leakage problem:* a presentation of the hardcoded secrets problem in public source code repositories and our approach to the solution;

- Chapter 4 *Architecture design and high level components:* gives an explanation of the architecture design of the solution including an overview of each single component.

- Chapter 5 *GitHub Scanner:* a detailed description about how the scanning process works.

- Chapter 6 *Path model:* description of the first part that compose the filtering model.
- Chapter 7 *Snippet model:* description of the first part that compose the filtering model.
- Chapter 8 *Evaluation:* experimental setup presentation for tool evaluation.
- Chapter 9 *Conclusion and future work:* final conclusion about the research work.

Chapter 2

Background

This chapter presents the required background to understand the content of the thesis.

Machine Learning techniques are used to reduce the number of false positives generated by scanning repositories just with regular expressions. The choice of Machine Learning came up because there was the necessity of some intelligent algorithms capable of analyzing the discovery and to output if it is a false positive or not.

This chapter starts with an introduction about Machine Learning, with its definitions and main concepts, then with a general explanation about Natural Language Processing and its relationship with Machine Learning. The following section is about *text classification*, an important NLP tasks solved through ML techniques. Than FastText is presented, a Python library developed by Facebook AI Research that solves this task and is adopted in the development of the tool. The last part is about performance metrics for classification models.

2.1 Machine Learning introduction

Nowadays we hear more and more talking about Artificial Intelligence, Machine Learning and Deep Learning, but what do these terms really mean? What is their relationship?

Artificial Intelligence (AI) can be classified as the discipline that involves the theories and practical techniques for the development of algorithms that allow machines (particularly “computers”) to show intelligent activity that simulates human behavior.

On the other hand, in the field of computer science, **Machine Learning (ML)** is an application of Artificial Intelligence that is based on learning and improving from experience autonomously in opposition to traditional programming. Machine learning focuses on the development of applications that can access data and use them to learn for themselves using statistical techniques.

The term Machine Learning was first coined in 1959 by Arthur Samuel and later taken up by Tom Mitchell who gave it a formal, contemporary definition:

> "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." [1]

In this context, Machine Learning algorithms are not kinds of algorithm explicitly programmed to perform a specific task as in traditional programming paradigm, but the key concept is to learn from data. Given a specific dataset, statistical methods are used to progressively improve the performances of the algorithm in the identification of data pattern.

Today there are different ways of learning which differ not only in the used algorithms, but especially for the purpose the machines are made for. Depending on the type of algorithm used, i.e. the way in which learning takes place, three different learning systems can be subdivided: supervised, unsupervised, and for reinforcement.

1. *Supervised Learning:* in this category of Machine Learning it is given to the algorithm both the input data and the information about the desired results, the labels. The objective is to identify a general rule that links the input data with the output data (i.e. given the input examples and its own labels it can learn the link between them), so it can then be reused for other similar tasks.

   In a more formal way, supervised learning is when you have input variables (x) and an output variable (y), and you use an algorithm to learn the mapping function from the input to the output.

   $$y = f(x)$$

   The goal is to approximate the mapping function so well that when you have new input data (x), you can predict the output variables (y) for that data.

   Supervised learning problems can be further grouped into regression and classification problems.

- *Classification*: the goal is to then take an input value and assign it a class or category. Examples of this algorithm are Linear Classifiers or Support Vector Machines (SVM).
- *Regression*: the goal is to predict a continuous number. Examples of this algorithm are Linear Regression or Logistic Regression.

2. *Unsupervised Learning:* as can be seen from the name, unsupervised learning, differently from the previous one, does not use previously classified and labeled data; therefore, we do not know to which category they belong. The machine is asked, therefore, to extract a rule that groups the data according to characteristics derived from the data itself.

   Unsupervised learning techniques are basically based on clustering or association rules. In clustering we start from an unlabelled dataset, in which, from the characteristics of a group of items, the items itself are grouped in a certain number of clusters, according to different ways such as similarity or distance.

3. *Reinforcement Learning:* it is a Machine Learning technique where an agent, by interacting with an environment, receives a feedback for each action it takes. This feedback can be a reward or a punishment and the goal of this kind of algorithm is to maximize the reward.

Another key concept in the Machine Learning field is **Neural Network (NN)** or Artificial Neural Network. As the name suggests, it is inspired to the human brain and tries to mimic how a biological neuron behaves. In fact, a NN has an essential building block as the human brain has called perceptron, which accomplishes the signal processing. More units are connected to a vast network.

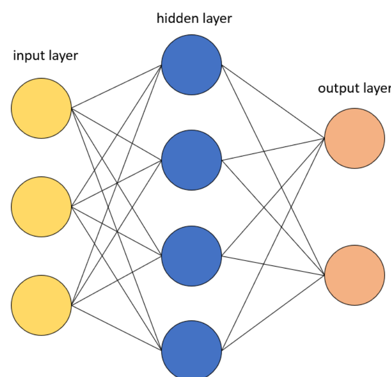


Figure 2.1: Structure of an artificial neural network.

A neuron has the aim to combine the data input with a set of coefficients, or weights, that amplify or reduce that input. In this way, the input of a neuron acquires meaning respect to the label it is trying to learn. The product between all the input and the weights in then summed, and the sum, plus a bias, is passed through an activation function to understand if the signal coming from that neuron should progress further through the network on not. If the signals pass through, the neuron has been "activated".

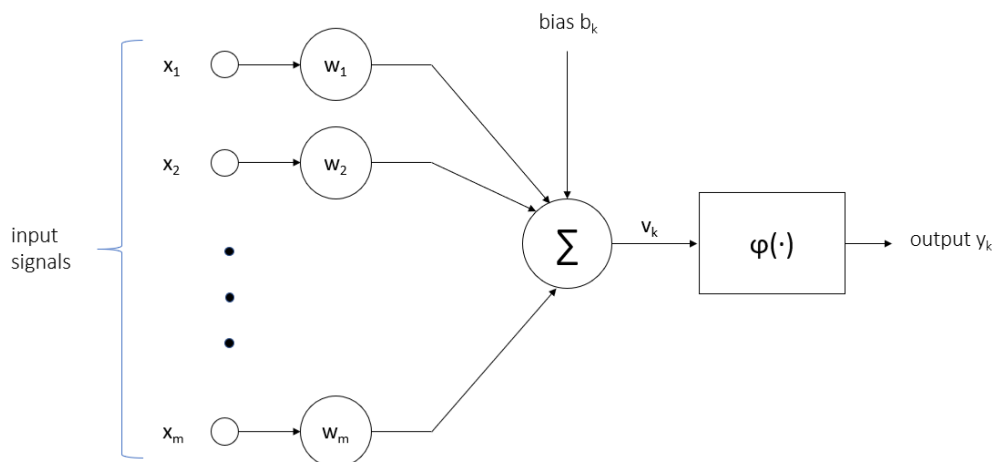


Figure 2.2: Structure of a neuron.

The learning phase in Neural Networks is divided into 2 phases that depend on each other:

- *forward propagation:* in this phase, starting from the input layer, a data sample is fed into the network and propagated till the output layer. Here the calculation and storage of intermediate variables (including outputs) take place;
- *backward propagation:* given the expected output y and the predicted output $\hat{y}$ a loss function is computed. After that, the weighs and the biases are modified proportionally to the gradient of this loss function.

From the concept of Neural Network, **Deep Learning (DL)** comes from. It is nothing more than "deep" Neural Networks, i.e., NNs composed by a large number of hidden layers and neurons. The recent success of deep learning can be attributed to the overcome of some obstacles that in the past precluded the achievement of desired results, such as the lack of data and adequate computational capacity.

Nowadays Deep Learning system make it possible to:

- Identify objects in images and videos;
- Translate speech into text;
- Identify and interpret the interests of online users, showing the most relevant results for their research.

Summarizing, AI tries to create general purpose intelligent machines; Machine Learning is a class of algorithms that AI uses, Neural Networks are a subset of models that are used in Machine Learning, they are characterized by their use of neurons. In the end, Deep Learning is a subset of NNs that are characterized by the use of multiple hidden layers of neurons.

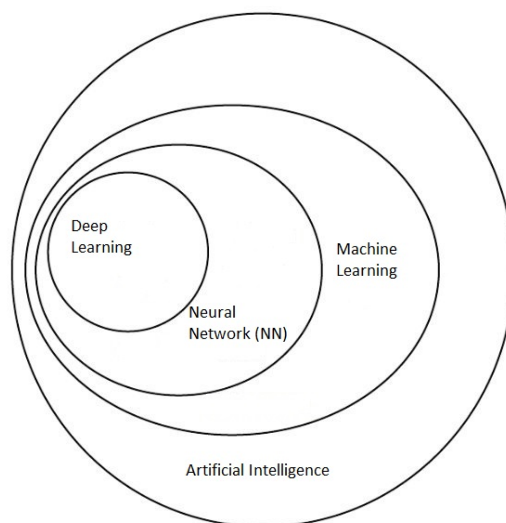


Figure 2.3: Relationship between AI, ML, NNs, DL.

2.2 Natural Language Processing introduction

The wider concept of Natural Language Processing falls within the thematic area of natural language. As the name suggests, Natural Language Processing refers to the computer processing of natural language for any purpose, independent of the level of analysis. Natural language refers to the language we use in everyday life, and it is synonymous with human language, mainly to distinguish it from computer language, which is called formal language. As it is, natural language is

the most natural and common form of human communication, in both his spoken and written version. Compared to formal language, natural language is much more complex, often containing sub understandings and ambiguities, which makes it very difficult to process.

Of course there are different aspect of natural language that are studied. Grammar is defined as the mechanism responsible for creating well formed structures in language and it consists of distinct modules. These includes:

- *syntax:* the study of how word are combined to form a sentence;
- *semantics:* the study of meaning in language;
- *morphology:* the study of units of meaning in a language;
- *phonology:* the study of how phones are used in different languages to create meaning;
- *phonetics:* the study of sound of human speech;
- *lexicon:* the study of the vocabulary of a language.

2.2.1 NLP and Machine Learning

A term that is used in parallel with NLP is Machine Learning. Anyway, Machine Learning and NLP are completely different concepts; the former refers to a type of approach, while the latter represents a thematic area. Nevertheless, the Machine learning approach has dominated the NLP scene in recent years, so sometimes many people forget or simply ignore the existence of other NLP approaches, i.e. language rules. So, it is no surprise that for these people, NLP is equivalent to Machine Learning, but it is not true.

There are a large variety of NLP tasks that can be solved through Machine Learning such as:

- *text classification:* capability to understand the content of a text and properly classify it with a label. These are often multi-label classification systems;
- *machine translation:* capability to translate a text in different languages;
- *question answering:* predict the answer given the question;

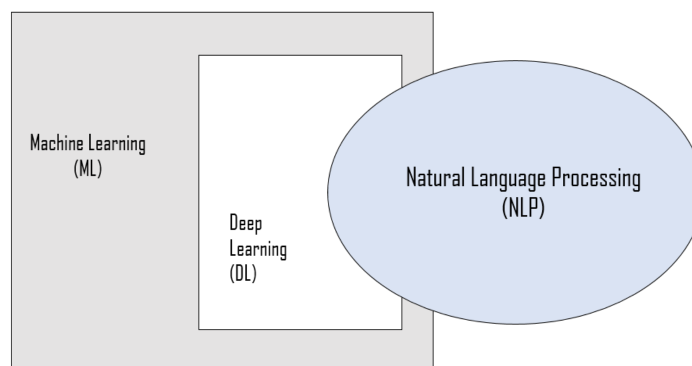


Figure 2.4: Relationship between ML, DL and NLP.

- *sentiment analysis:* the information extraction from a text about the general emotions transmitted by words. It is possible to identify offensive and spam comments and automatically delete them;
- *speech recognition:* recognition and translation of spoken language into text;
- *named entity recognition (NER):* identification of “named entity”. We define named entity as any object that exists and can be identified by a proper name.

2.3 Text representation

One of the basic building block in NLP that impact on the performance of Machine Learning and Deep Learning models is *text representation*. Giving a more formal definition, *text representation* is a set of modeling techniques in which words or phrases of vocabulary are mapped into real number vectors. That's because Machine Learning models work with numbers, and there is the necessity to have some techniques that transfer text from human language to a machine-readable format. To better understand, it is useful to explain also the concept of corpus.

A **corpus** is a collection of texts selected and organized in such a way that they meet specific criteria that make them functional for linguistic analyses. The most important characteristic it must have is that it needs to be representative of a language. For example, if we are building a Question/Answer system, a corpus of phone conversation would be appropriate.

It is important to notice that *text representation* methods work both on word or token level and on document level as these are both considered pieces of text.

Both of these techniques will be explained.

2.3.1 Bag-of-Words (BoW)

This is the simplest and most intuitive technique to generate a vector from a text. Given a vocabulary composed of V words, every word is represented by a binary vector that has the same dimension as the vocabulary and a 1 in the position of that specific word in the vocabulary and a 0 in all the others positions. This is also called **one-hot representation** of a word. The representation of a document using this technique will be the sum of the vectors related to the words in that document.

The two main problems of this representation are:

- for each text, it generates a large and sparse vector that requires large memory for computation;
- it is not possible to understand any relationship between words. The inner product between any two different words is always 0.

	Vocabulary	the	cat	is	on	table	hungry
0	the	1	0	0	0	0	0
1	cat	0	1	0	0	0	0
2	is	0	0	1	0	0	0
3	on	0	0	0	1	0	0
4	table	0	0	0	0	1	0
5	hungry	0	0	0	0	0	1

The cat is on the table

vector representation	2	1	1	1	1	0

Figure 2.5: Bag-of-Word example.

2.3.2 Bag-of-N-grams

This technique can be considered as an extension of the previous one because when we consider a Bag-of N-grams with $N = 1$ we are just referring to BoW. Starting

from the concept of N-gram, it is nothing more than a sequence of N token. Given the corpus and a specific value of N, all the concatenation of N token we find sliding a window of size N over the corpus are saved in the vocabulary. When we create the vector representation of those sentences, it is given a 1 if the N-gram is present in the sentence, 0 otherwise. It will be more precise with an example.

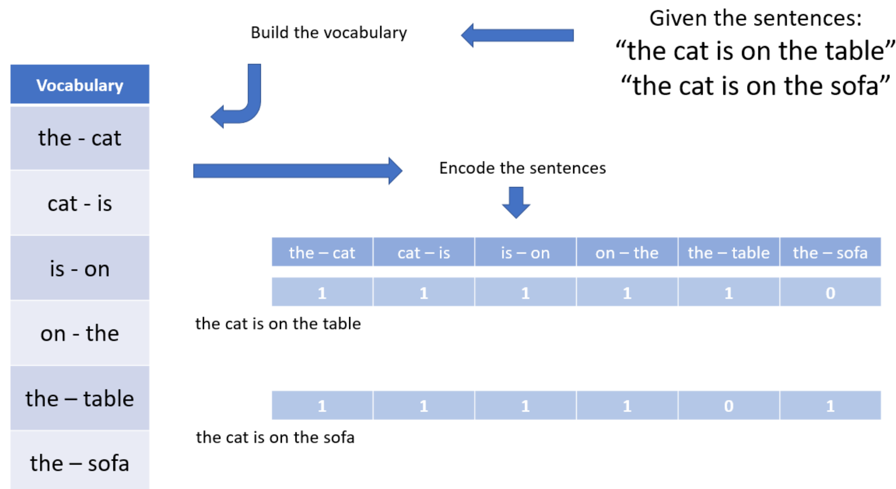


Figure 2.6: Bag-of-N-grams example with N = 2.

The main advantage of this kind of approach is that word order is now taken into consideration, so it encodes not just keywords but also word ordering automatically. The limitations are the same as BoW technique, and of course, long range dependencies are not captured.

2.3.3 TF-IDF

TF-IDF stands for Term Frequency - Inverse Document Frequency.

- *Term Frequency (TF):* gives the frequency of the word in each text in the corpus. It is the ratio between the number of times a word appears in a text and the total number of words in that text. It increases as the number of occurrences of that word within the text increases.

  $$tf_{i,j} = \frac{n_{i,j}}{|d_j|}$$

  In this formula, n is the number of occurrences of term i in document j, while the denominator d is simply the size, expressed in number of terms, of document j.

- *Inverse Document Frequency:* is the logarithm of the ratio between the number of documents and the number of document that contains a specific word. It determines the weight of rare words across all documents in the corpus.

$$idf_i = \log \frac{|D|}{|\{d : i \in d\}|}$$

  D is the number of documents in the collection, while the denominator is the number of documents containing the term i.

TF-IDF is simply the TF multiplied by IDF.

$$w_{i,j} = tf_{i,j} \cdot idf_i$$

The weight of a word increases proportionally to the number of times that word appears in a document but decreases if it is contained in many documents. So, words that are common in every document, such as stop-words, have a small weight even though they may appear many times since their relevance in the document is little. The main advantages between TF-IDF and the previous text representation methods are it can embed in its vector representation how useful a word is to a document, and it helps in ignoring words that are misspelled.

2.3.4 Word Embedding

To overcome the main disadvantages of sparse vector representation such as BoW or TF-IDF, *Word Embedding* methods have been developed. The main advantage of these techniques are:

1. the capability to catch the syntactic and semantic characteristic of a word in a document, embedding the meaning of the word in a feature vector, where each entry will be a hidden feature inside the word meaning. In this way, for example, the vectors associated with the words “cat” and “animal” have a high degree of similarity;

2. the generated vector will be denser respect to the previous representation, so it will be easy to use in terms of space and time.

Word Embedding can be better understood by looking at this relationship:

$$E_{Rome} = E_{Italy} + (E_{Paris} - E_{France})$$

The relationship between words can be obtained just with linear algebra. If we know the vectors that contain the embedding of Italy, Paris, and France, we can just guess which will be the capital of Italy solving this simple equation.

The main problems of this kind of representation are:

- *out of vocabulary words (OOV):* these models fail in generating the word representation for words that don't compare in the training set;
- *polysemy:* these models don't take into account the possibility that a word can have more than one meaning.

Word2Vec

T. Milkov et al. at Google Inc. proposes a word embedding model called Word2Vec that "can be used for learning high-quality word vector from huge dataset with billions of words and with millions of words in the vocabulary" and "with a modest dimensionality of word vector between 50-100" [2]. The two main architectures of Word2Vec are *Continuous Bag Of Word (CBOW)* and *Skip-Gram*. Learning word embedding should be an unsupervised learning process, but both these method convert the generation of a vector representation in a supervised learning problem that is solved using Neural Networks.

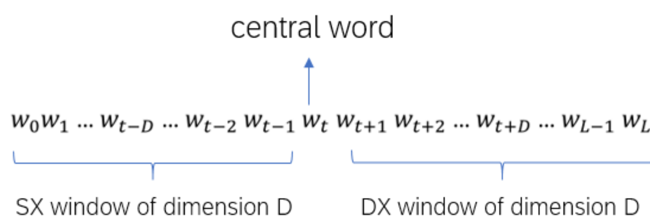


Figure 2.7: Word2Vec context window.

Given a document of length L as input and given w_t , the current central word with $0 \leqslant t \leqslant L-1$, and its relative context windows of length D composed by words w_{t-j} with $1 \leqslant j \leqslant D$ and $-D \leqslant j \leqslant -1$:

1. **CBOW:** the prediction of the central word w_t is made given the Bag-of-Words of the context words w_{t-j}.

2. **Skip-Gram:** the prediction of the context words w_{t-j} is mad given the central word w_t.

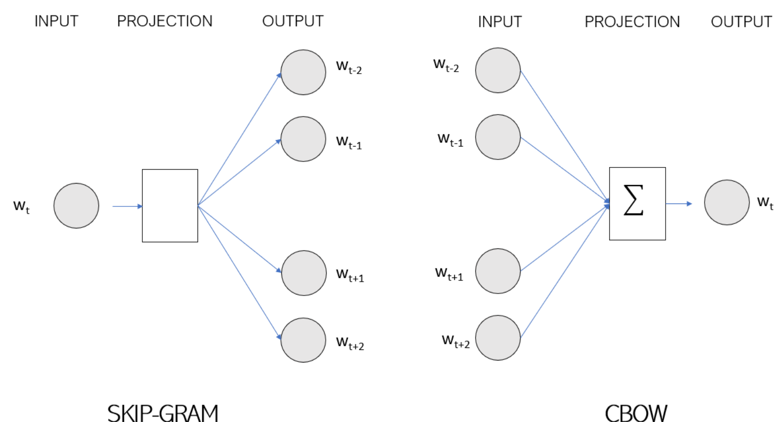


Figure 2.8: Skip-Gram and CBOW models.

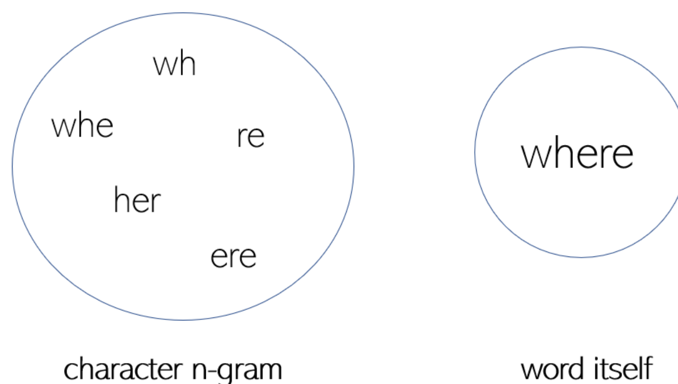
GloVe

GloVe stand for Global Vector and it is another word embedding method and was developed as an open source project at Stanford University [3]. It is an unsupervised learning algorithm for obtaining vector representations and it can be used to train word vectors on a new corpus but also pre-trained word vectors are available. It uses word-word co-occurrence statistics from a corpus for building a vector space with meaningful substructure.

2.3.5 FastText for text representation

FastText is a library for text representation developed by Facebook AI Research in 2016, and it partially avoids some problems that are proper of the word embedding methods described above.

They say: "Our main contribution is to introduce an extension of the continuous Skip-Gram model (Mikolov et al., 2013b), which takes into account subword information." [4] so their work can be considered as an improvement of Skip-Gram model. The latter builds a vector representation for each word, without taking into consideration the internal structure. In FastText, a word is represented using the word itself and a set of character n-gram. Considering the word "where":



It is important to notice that the english word “her” is still different from the character n-gram “her” obtained from “where”.

The main advantages of using FastText are:

- it works well with OOV (Out of Vocabulary) words. Using character level n-grams the model can generate a vector even for words that don’t appear in the vocabulary;
- the morphology of a word is considered into the representation. Even in this situation, character n-grams enrich the word representation with subword information;
- the model is built to be fast in the training phase.

2.4 Text classification models

In the previous step, the text is transformed from a human readable format to numbers by extracting features from it. The final step is to choose and train a classifier using the features created before. The aim is, of course, assigning a label to a given piece of text.

2.4.1 State of the art for text classification

A lot of different Machine Learning models can be applied in this situation, starting from the more simple classifier to advanced Deep Learning models.

Among the simplest we have:

- *Naive Bayes Classifier:* it is a classification technique based on Bayes' theorem that makes the hypothesis of feature independence.
- *Linear Classifier:* it measures the relationship between one or more independent variables and categorical dependent variables by estimating probabilities using a logistic function.
- *Support Vector Machine (SVM)*: it tries to generate the best hyper-plane line that separates two (or more) classes.
- *Bagging models:* it aims to create a set of classifiers of equal importance. At classification time, each model will vote on the outcome of the prediction, and the overall output will be the class that received the most votes. Random Forest Classifier is an example and belongs to the tree based model family.
- *Boosting models:* different from bagging, each classifier influences the final vote with a certain weight. Xtereme Gradient Boosting (XGBOOST) [5] is an example and belongs to the tree based model family.

On the other hand, in some studies Deep Learning approach was preferred to conventional classifiers. In mostly depend on the kind of problem approached, but the most used techniques that use DL involves:

- *RNN with LSTM:* where tokens are processed in sequential order, so a sentence is considered as a sequence of words (or a word is considered as a sequence of characters). Differently from feed forward neural network, where the information/signal can only go in one direction, and each neuron can be interconnected with one or more neurons of the next chain, in Recurrent Neural Networks (RNN), neurons can also admit loops and/or can be interconnected to neurons of a previous level. These characteristics allow neurons the ability to remember what it was learned before. RNNs suffer from a problem called Vanishing Gradient, so while learning, the larger the sequence of neurons is, the harder it is to learn from early layers. To solve this problem LSTM (Long Short Term Memory) cells have been created.
- *Convolutional Neural Network:* in this kind of network, that are feed forward, convolutions are computed on the input layer to produce the output. CNNs are widely used in image recognition due to the inner structure of an image, and they are starting to be used on text classification too. This approach was used in a paper of A. Conneau et al. where they say that text have same proprieties as images: "characters combine to form n-grams, stems, words, phrase, sentences, etc."[6]. In that work, a character level deep convolutional network with 29 layers was developed.

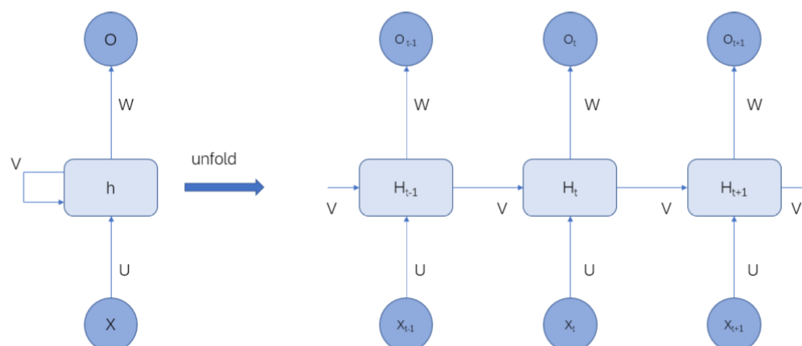


Figure 2.9: RNN example.

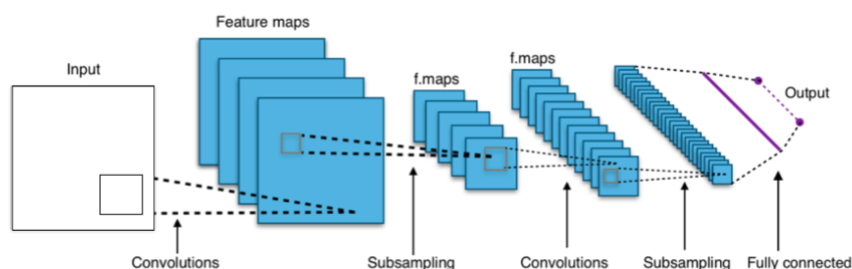


Figure 2.10: ConvNet example.

2.4.2 FastText for classification

Facebook AI Research, together with the text representation, developed a text classification library in 2016 that "is often on par with deep learning classifiers in terms of accuracy, and many orders of magnitude faster for training and evaluation" [7]. This model learns the words representations at character n-grams level in the same time as the classification task.

Given a sentence in the corpus, denoted as s, composed of N-grams features $f_1, f_2, ..., f_N$, the features representation is obtained via a weight matrix A to obtain:

$$x_i = A \cdot f_i$$

Then y is defined as the linear bag of words of the sentence by averaging all the feature representation x_i:

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

y is the input of an hidden layer associated with a weight matrix B, such as the output:

$$z = B \cdot y$$

We can compute the probability that a word vector belongs to the jth class M as follows:

$$p_j = \sigma(z_j)$$

with

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

the softmax function. The weight matrices A and B are computed by minimizing the negative log-likelihood of the probability distribution, using stochastic gradient descent:

$$\frac{1}{N}\sum_{k=1}^{N} y_k \cdot log\Big(\sigma(A \cdot B \cdot x_i)\Big)$$

As the word representation matrix A is computed in the same time as the weight classification matrix B the word representation is specifically designed for a specific task as opposed to stand alone word representation.

FastText also provides a set of API for training the text classification model. It also allows to change parameter for improving the performance of the classification such as:

- *Word n-gram:* for taking word order into consideration.
- *Hierarchical softmax*: that approximates the softmax with a faster computation.
- *Epochs.*
- *Learning rate.*

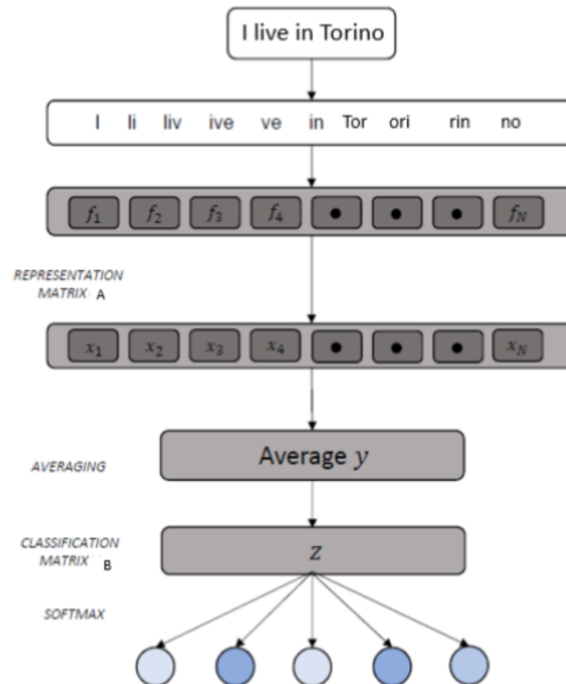


Figure 2.11: FastText for classification example.

2.5 Performance Metrics

Once the text representation technique is chosen, and the model is selected, it is important to test the overall architecture. Once the available data are carefully divided into training and test set, different metrics are available for the evaluation of Machine Learning algorithms. This part is focused on classification problems.

The key concept of performance metrics is the *confusion matrix* that is not a metric itself, but almost all the performance measures are based on it. Here it will be explained the two-class confusion matrix is used to predict the result of a binary classifier. Here, for simplicity's sake, we will illustrate the two-class confusion matrix, which is used to evaluate the result of a binary classifier.

Each column of the matrix represents the predicted values, those that the ML model outputs, while each row represents the real values. Using this matrix, it is observable if there is “confusion” in the classification of different classes.

If class labels assume only two values, that are indicated as true and false, all the instances of the test set can be divided into four groups:

- *True positive:* true class instances classified as true.
- *False positive:* false class instances classified as true.
- *True negative:* true class instances classified as false.
- *False negative:* false class instances classified as false.

Predicted

Actual		True	False
	True	True positive (TP)	False negative (FN)
	False	False positive (FP)	True negative (TN)

Figure 2.12: Confusion matrix.

Different metrics can easily be obtained from the confusion matrix:

- *Precision:* the percentage of positive cases correctly classified.

$$\frac{TP}{TP + FP}$$

- *Accuracy:* the percentage of correct predictions over all the prediction made.

$$\frac{TP + TN}{TP + FP + TN + FN}$$

- *Recall:* the percentage of correct predictions over the true cases.

$$\frac{TP}{TP + FN}$$

- *Specificity:* the percentage of correct predictions over the false cases, it is the opposite of recall.
$$\frac{TN}{TN + FP}$$
- *F1 score:* is calculated using the harmonic mean of precision and recall. It single score that represents both precision and recall.
$$2 \cdot \frac{precision \cdot recall}{precision + recall}$$

There are also other metrics related to classification problems like the AUC-ROC Curve, Log loss, F-Beta score, etc. , but are out of the scope of this thesis.

Chapter 3

Secrets leakage problem

Nowadays, one of the most critical types of data leaks is represented by plaintext (or hardcoded) credentials in an open-source project. GitHub itself, a platform that contains more than 100 million of repositories (with at least 28 million public ones), has become the most used place where users can publish their code and collaborate to open-source projects [8]. Meanwhile it also turned into an important source of secrets for malicious agents. Some developers, mainly because of inexperience or laziness, leave them hardcoded in the source code.

Even the use of GitHub search tool became popular for finding secrets. If we look for specific strings like "remove credential" or "password removed" in commit messages it is not difficult to find thousands of keys and passwords, as it happened in 2013 where GitHub Search has to be shut down after secret exposition [9]. This example shows how easy it can be to find leaks, there is no need to be an experienced hacker to find secrets and to exploit them.

It is not the first time the problem is approached, there are other open-source tools that try to tackle it using regular expressions but, due to a lack of granularity and precision of them, the amount of false positives they generate is very high. From our knowledge, there are also tools that are not open-source that scan GitHub repositories using Machine Learning but with a different approach respect to ours.

In this chapter at first are given some definitions and explained the best practices a developer should follow for handling secrets in his project, then we move to the state of the art that includes a study conducted on a large number of repositories and the related work about this problematic. In the end it is explained the chosen approach to solve the problem and the reason that brought to the use of Machine Learning.

3.1 Definitions

For a better understanding of the content of this thesis let's define at first what in this work it is considered as a **discovery**, a **leak** and a **secret**. We define a **discovery** or a **match** as the result of the application of a specific set of regular expressions on the content of a repository. As it is shown in Figure 3.1, a discovery is not just the text that matches a regular expression but it is the whole line where that text is found.

regular expression: password(*)=(*)

```
1 ...
2 username = "Carlo"
3 password = "qwerty123!"   → discovery
4 connection = db.connect(username , password)
5 ...
```

Figure 3.1: Example of a discovery.

A discovery can be a **leak** or a **non critical data**. A **leak** is a sensitive information found in open-source projects, which may compromise developers' privacy, a **non critical data**, on the other hand, is a piece of text that does not contain sensitive information. Since the purpose of the developed tool is to find leaks avoiding non-critical data we will refer to them as **false positives**.

A **secret** is, more generically, something that is kept or meant to be kept unknown or unseen by others. In this situation can be:

- *username/password/email:* are personal information about an user. Of course this kind of tokens can be potentially found in any string. The tool we build tries to catch them using regular expressions looking for specific patterns;
- *API token:* is a unique code, passed to an API to identify the calling application or the user. API keys are used to track and control how the APIs are used, for example, to prevent malicious use or abuse of the API. It is often used as both a unique identifier and a secret token for authentication and generally has a set of access specific to the identity with which it is associated;
- *cryptograpyc keys:* such as RSA or AES keys.

```
def define_secret(object):
    password = 'qwerty!@#123'                              → leak
    object.set_password('test')                            → non critical data
    object.update_credential('test', 'dummy')              → non critical data
    email = 'carlomngr@gmail.com'                          → leak
    username = 'I433344'                                   → leak
    private_key = """-----BEGIN RSA PRIVATE KEY-----       → leak
        MIIBOQIBAAJBAIOLepgdqXrM07O4dV/nJ5gSA12jcjBeBXK5mZO7Gc778HuvhJi+
        RvqhSi82EuN9sHPx1iQqaCuXuS1vpuqvYiUCAwEAAQJATRDbCuFd2EbFxGXNxhjL
        loj/Fc3a6UE8GeFoeydDUIJjWifbCAQsptSPIT5vhcudZgWEMDSXrIn79nXvyPy5
        BQIhAPU+XwrLGy0Hd4Roug+9IRMrlu0gtSvTJRWQ/b7m0fbfAiEAiVB7bUMynZf4
        SwVJ8NAF4AikBmYxOJPUxnPjEp8D23sCIA3ZcNqWL7myQ0CZ/W/oGVcQzhwkDbck
        3GJEZuAB/vd3AiASmnvOZs9BuKgkCdhlrtlM6/7E+y1p++VU6bh2+mI8ZwIgf4Qh
        u+zYCJfIjtJJpH1lHZW+A60iThKtezaCk7FiAC4=
        -----END RSA PRIVATE KEY-----"""
    api_key = "zaCELgL.0imfpc8mVLWwsAawjYr4Rx-Af50DDqtlx"      → leak
    object.update_credential(email, password)              → non critical data
    private_key = 'example'                                → non critical data
```

Figure 3.2: Examples of leaks and non critical data.

3.2 Best practices in secrets management

The main cause of leaks in public repositories can be attributed to the inexperience of developers or their laziness. Sometimes they do not know how to handle a secret in a project or maybe they do but, because of laziness, they just leave passwords or API keys hardcoded.

There are lots of ways a user can handle secrets:

- *ask the user:* he is the only one who should know the secrets, asking him is always a good solution;
- *configuration file:* it is the easiest way. A configuration file with all the secrets will be part of the project but will never be pushed on public source code repository by adding it in the .gitignore file. A sample configuration file will be pushed instead just to clarify the user about its usage;
- *encrypt:* it is based on the encryption of all the secret or of the file where the secrets are with a secret that the program already knows;
- *key management services:* the main cloud provider, such as AWS, Azure and Google Cloud, use a secure way to handle secrets. They use a secure solution for storing passwords for the applications that are hosted on their services.

If by mistake it happens that a secret is committed in a public source code repository it is important to be aware that the secret became compromised and it needs to be changed immediately. Moreover it has to been noticed that if another commit

is done to remove the hardcoded secret it will remain in the commit history of a repository. For this reason it is important to rewrite the history, by using the *filter-branch* command on Git, as GitHub suggests [10].

3.3 State of the art

In this section are presented some of the latest work, research papers, and methods that try to tackle the secret leakage problem. At first, in the paper of Meli et al. [11], it has been done a large-scale study about leaks in source code repositories, then other open-source tools that try to achieve good results in leak detection using regular expressions are analyzed. After that an overview of two existing tools that use Machine Learning is done. As they are not open-source it is explained what they pretend to do.

3.3.1 How bad can it Git?

In this paper, the authors propose "the first comprehensive, longitudinal analysis of secret leakage on GitHub" covering the 13% of open-source repositories [11]. They focus on what they called "distinct secrets", which are secrets that have a very distinct structure. These keys involve just a limited set of API keys and private keys (15 API keys and 4 asymmetric private keys) so they didn't consider other kinds of secrets such as usernames, passwords or email address.

Their secret detection is divided into 3 phases:

1. GitHub's Search API and Google BigQuery dataset are queried using specific strings in order to find "candidate files" that are files likely to contain secrets;
2. candidate files are scanned with a specific set of regular expressions looking for "candidate secrets";
3. candidate secrets are filtered using three different kinds of filters (entropy Filter, word filter, and pattern filter) in order to obtain "valid secrets" that can be used for the analysis.

Querying GitHub's Search API they got around 4 million candidate files and a hit rate of "candidate secrets" of 7%, where 99% of them "valid secrets". Querying the Google BigQuery dataset they got around 2 billion candidate files, a hit rate of 0.005% of "candidate secrets", where 99% of them "valid secrets". They also did a small manual review of this statistical approach on 240 candidate secrets estimating that 93.74% of API secrets and 76.24% of asymmetric were sensitive.

3.3.2 Open-source existing tools

In this part are presented all the open-source tools that try to find leaks in public source code repositories. As it was said, all of them are regular expression based and produces a high rate of false positive. All of them are available on GitHub and they are ordered by their popularity on the website.

Git-secrets (7.3k stars)

It is a tool developed by AWS Labs that "prevents you from committing passwords and other sensitive information to a git repository" [12]. It works on local files and not on published repositories and checks if the commit message and the content of a specific commit or branch match a prohibited set of regular expressions. The false positives problem is tackled using regular expressions. It does not provide a user interface.

TruffleHog (4.6k stars)

It is a tool used to scan GitHub repositories that goes deeper into the commit history of every branch of the repository it is chosen to scan. It computes the diff from each commit. It checks for commit both with regular expressions and by running entropy check on the git diff. It is one of the most popular because it is the only one that goes deeper into every piece of text published on GitHub [13].

Scumblr (2.6k stars)

It is a web application written in Ruby on Rails developed by Netflix and currently deprecated. It allows searching in GitHub repositories for secrets, patterns, and vulnerabilities and it is regular expressions based. It also allows tracking, ticketing and monitoring security vulnerabilities [14].

Repo Supervisor (359 stars)

"Serverless tool that detects secrets and passwords in your pull requests" [15]. It is regular expression based and it also performs entropy check. It outputs the results in a JSON format.

Gittyleaks (313 stars)

It is written in Python and it works on already existing repositories and tries to find API key, username email and password using regular expressions [16].

Git Hound (213 stars)

It is a git plugin written in golang that "prevents sensitive data from being committed into a repository by sniffing potential commits against regular expressions" [17]. The main difference between other tools is that it tries to prevent secrets to be published instead of just looking at the repository after it is committed.

3.3.3 Proprietary existing tools

In this part are presented all the proprietary tools that try to find leaks in public source code repositories. Two of them pretend to use Machine Learning but not for false positives reduction.

GitHub Security

GitHub team itself, joining with Semmle, is building his own tool for automatic token scanning [18]. Their security tool focuses only on API token detection and they support the API key of the most popular service providers. It is a regular expression based search that generates alerts in case of a match. They pretend also to look through the code for vulnerability providing an option to fix them [19].

GitGuardian

GitGuardian is a french start-up that provides a tool that can find "many different types of secrets: API keys, database connection strings, credentials, certificates ..." in source code repositories [20] . They also pretend to use Machine Learning in the detection phase and they also added a feature to improving training model: "we use a combination of regular expressions, entropy statistics and Machine Learning to achieve good performance. We're constantly improving our algorithms thanks to the labeling of our users." [21].

As it is a proprietary tool it was tested just a free version that wasn't as promising as they pretend to be. They were detecting just API tokens that are easy to recognize just with regular expressions.

NightfallAI

NightfallAI (formerly known as Watchtower Radar) is an American start-up and they developed a tool that "uses Machine Learning to identify business-critical data, like customer PII, across your SaaS, APIs, and data infrastructure" [22].

As they say "Nightfall's detectors are built via machine learning, so you'll receive more accurate, less noisy results than traditional approaches like regular expressions or high-entropy string detection" [23], so Machine Learning techniques are used just for detection and not for false positives reduction.

3.4 Problem approach

As it was described, the majority of available tools, both open-source and proprietary, are regular expressions based because of their good performances in terms of accuracy. On one side, if the case of API keys is considered, these are very easy to catch because the majority of them follow specific patterns (i.e. AWS API key that starts with "AKIA" and Google API key that start with "AIza") and with regular expressions, it is possible to catch these patterns. On the other side, there could be secrets that don't follow patterns like usernames or passwords and, in this case, regular expressions can not guarantee you have found a leak but they still can output a string that has a high probability of containing a leak.

Due to the efficiency of this approach it was decided to start from the development of an efficient **GitHub Scanner**. Receiving in input a set of regular expressions and the URL of a repository available on GitHub, the Scanner digs into every file of the repository and outputs the lines where these regular expressions are matched.

It is obvious, as the analyzed tools show, that regular expressions are not sufficient. If more specific patterns are used, they produce more accurate results but they can potentially exclude many data leaks and, in the opposite situation, if more generic patterns are used there is a rise of false positive rate and in case of big-sized source code it leads to a severe drop in terms of accuracy. If we consider that each secret detected with a scanning tool involves a manual check aimed at addressing it, it is clear that the lower is the number of false positive matches the lower is the overhead represented by the code revision activity.

Once the Scanner was developed, and it was understood that regular expressions were not sufficient for the secrets we want to target, two different kinds of approaches were possible, either we process the set of regular expressions used for the scan or we process the matches resulting from the scan.

- In the first approach, that we call a **static approach**, the core issue is trying to find a trade-off between the precision of regular expression and the need to detect all the leak. The more regular expression are accurate, the more it is needed to cover all the cases;

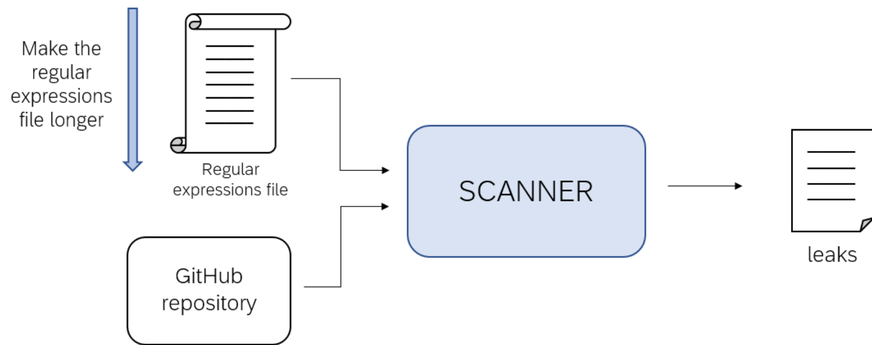


Figure 3.3: Static approach example.

- In the second approach, that we call **dynamic approach** the goal is to expand the set of regular expression and in the meanwhile build a filter. A filter can be considered as something that takes the match, analyze it and understand if it is a non critical secret or a leak.

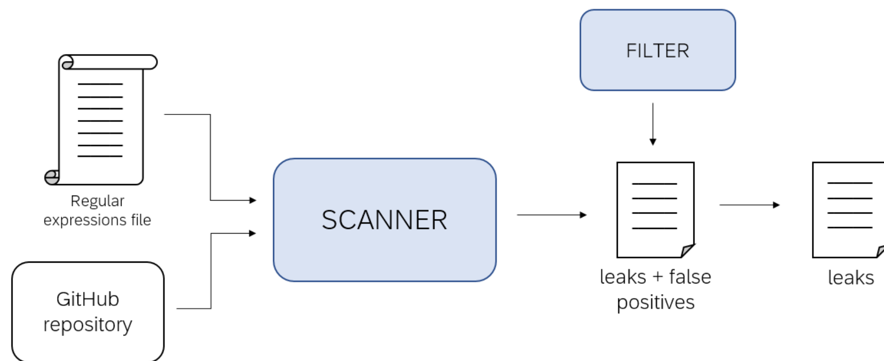


Figure 3.4: Dynamic approach example.

3.5 Machine Learning choice

Finding real secrets using a static approach could be a hard task because, by definition, they are a really small percentage of the amount of source code available on public source code repositories. For this reason it was chosen to adopt a dynamic approach. More than that, raising the amount of regular expression is not a scalable solution for a couple of reasons:

1. A lot of leaks are very generic, so it impossible to avoid a decrease in terms of accuracy of the tool. If we consider the example in Figure 3.5 where there are two lines written in Java, it can be understood that it is impossible to write a single regular expression that matches both these lines of code. If the same concept is extended to all the existing programming languages the task becomes non-trivial;

```
1 public static void main(String []args){
2
3     String password = "123qwerty!!!";
4
5     Credential myCredential = new
          Credential("carlomngr@gmail.com", "123qwerty!!!");
6
7 }
```

Figure 3.5: Example of two different patterns for hardcoding secrets in Java.

2. The tool should be constantly updated with new regular expressions to cover new secrets such as new API keys or new programming patterns.

The need for a filter that is able to analyze a match and classify it as a true secret or a false positive and the increasing use of Machine Learning techniques in security, lead us to the choice of Machine Learning. In this way it is possible to process the matches and classify them using Machine Learning models.

For this reason, once the Scanner was finished and a set of regular expression were written, a study on the matches was conducted in order to understand witch were the characteristic and the patterns that can be exploited in order to build what it was called a **Leak Classifier**.

More about this study will be explained in the next chapter but the main idea was to build two different models:

- **Path model**: it is the first one that is applied to the match. It takes as input the path of the repository where the match is found and classifies the discovery based on this input;
- **Snippet model:** it takes as input the code snippet of a discovery and classifies it.

Because of just using the Path model we have a good reduction in terms of false positives, the Snippet model is applied after the first one. So basically the output of the Leak Classifier is an OR function between the output of the Snippet Model output and the output of the Path Model.

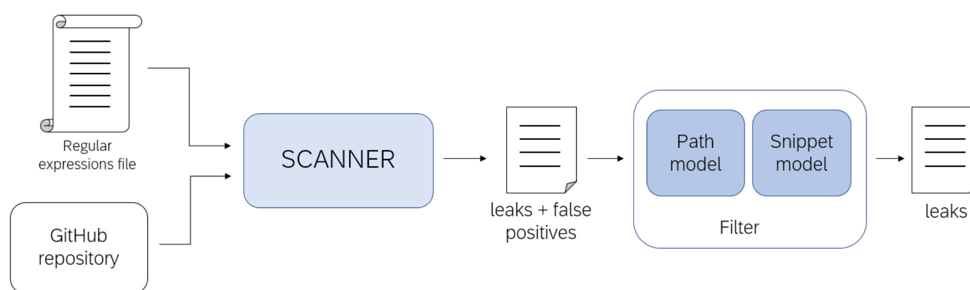


Figure 3.6: Solution architecture.

Both in case of a path false positive and code snippet false positive, the classification task is considered as a text classification task in the area of natural language processing.

Another natural language processing technique, in particular text representation, was also used to improve manual check. When a user tags a discovery as false positive, the vector representation of that leak can be computed and compared with other representations. If similar representations are found the tool tags more than one discovery at the same time.

Chapter 4

Architecture design and high level components

In this section, it is explained the high level design of the chosen architecture and a general overview of every single component is given.

Due to the decision to open-source the whole project the architecture structure is more similar to a prototype and not to a real proprietary software. For this reason, and because of the most famous Machine Learning libraries are available in Python, the whole project was written using this programming language.

4.1 High level design

It has been developed a web application based on the client - server model using REST architecture. The main components are:

- *web server:* Nginx, one of the most popular open-source web server, was used [24]. It contains the web application server, written in Python that perform all the request done by a user who interacts through a GUI;
- *database:* a relational database, PostgreSQL in particular, is used[25].

For a better deployment of the application, both the web server and the database were wrapped in two different docker containers that can be launched using the "docker-compose" command. For the database a docker volume was declared in order to guarantee the data persistence.

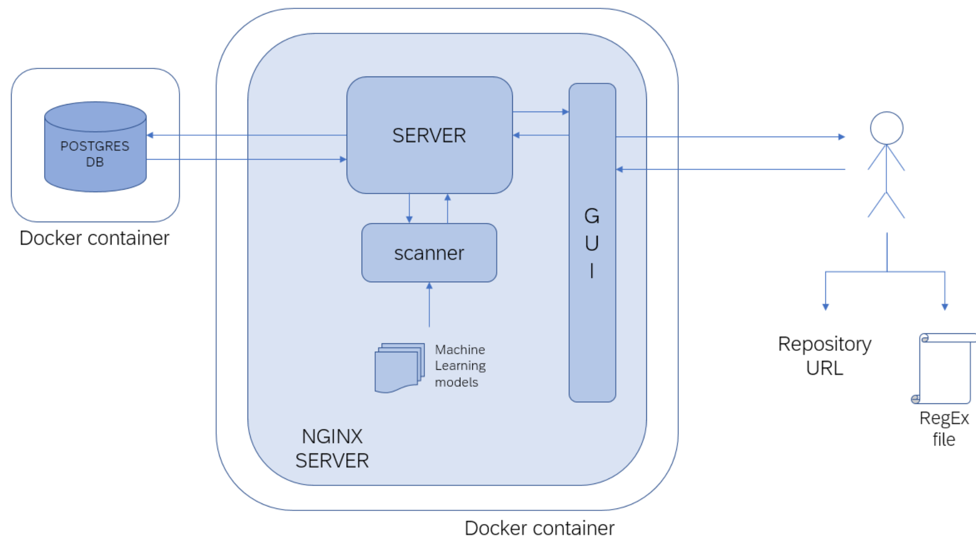


Figure 4.1: Overall architecture.

In the end, a user, having a set of regular expression file and the URL of the repository to scan, can analyze it and find if some leak is present.

The project is still under development and is not released yet as open-source.

4.2 GUI

The first component to be explained will be the Graphical User Interface. Explaining all the ways the user can interact with the GUI, it is easy to understand all the functionalities we gave to our tool.

In Figure 4.2 it is shown the dashboard of the tool. From here there is an overview of all the scanned repository, including the number of total discoveries (the matches) and the number of uploaded rules (regular expression for scanning). From this interface it is possible to scan a new repository or add a new rule using the related buttons.

At first it is necessary to add some new rules. This is possible from the dashboard, clicking on "Add Rule" button or going to "Rules" in the header bar where there is another button similar to the previous one. To add a rule, a category must also be specified. The category of a regular expression gives an idea of what kind of secret it will match (i.e. RSA key, AWS API token etc.). In "Rules" page

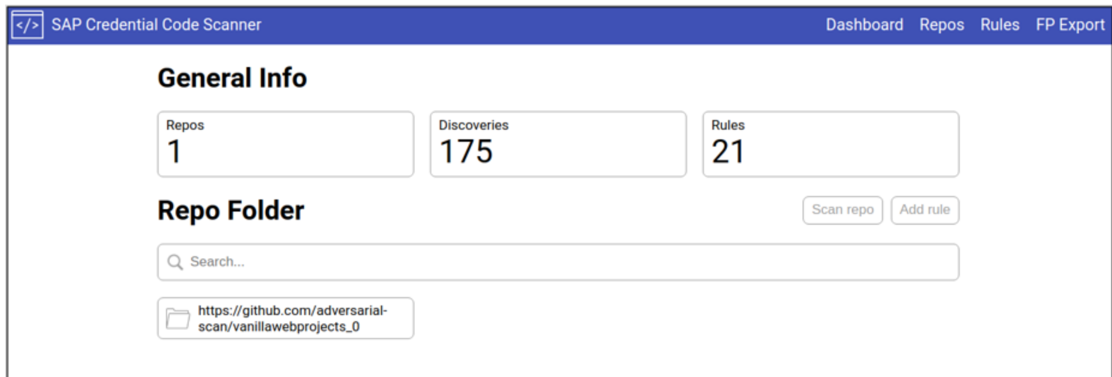


Figure 4.2: SCCS Dashboard.

there is also a button that allows you to upload a file containing regular expressions in a JSON format.

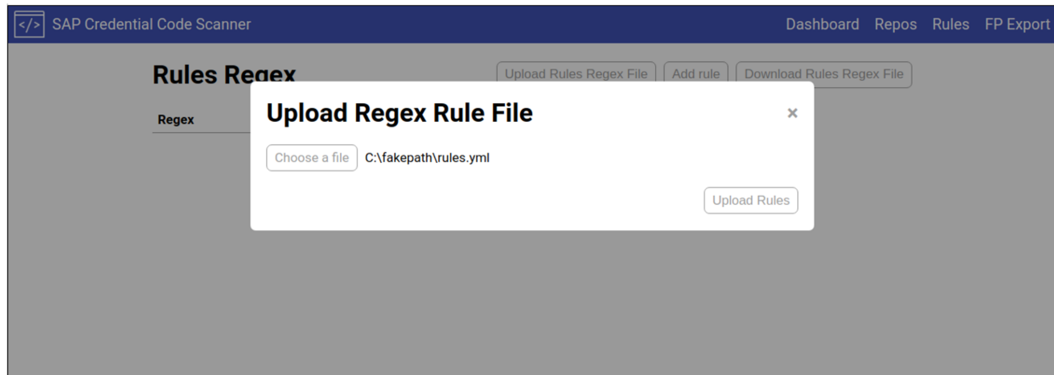


Figure 4.3: SCCS Add a rule.

After that, if the "Scan Repo" button is pressed, the interface shown in Figure 4.4 appears.

Here the user can insert the repository URL, select a certain category of regular expressions and just by clicking on "Start Scan" the scan begins. Of course, the Machine Learning models for filtering false positives are automatically applied. When the scan finishes, the scanned repository will appear in the dashboard and, by clicking on it, all the filtered matches are shown.

Clicking on a single discovery it is displayed the line of code, the regular expression that matched it, the commit ID and the path where it was found. Some options are available as it is shown in Figure 4.5:

- *Open commit:* open the GitHub page where the discovery was found;

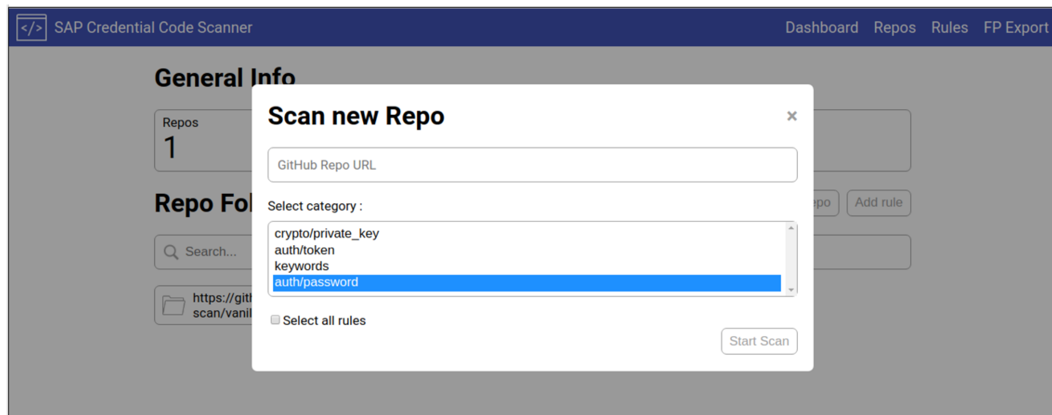


Figure 4.4: SCCS Scan process.

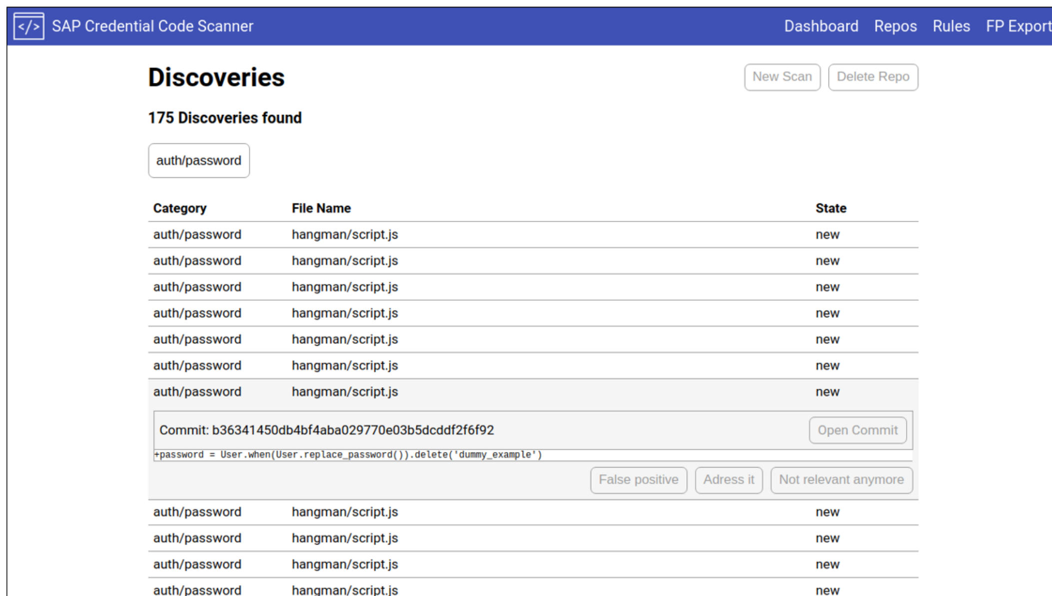


Figure 4.5: SCCS Address a discovery.

- *False positive:* user can manually tag a discovery as false positive. The tool doesn't have 100% precision so this action is required;
- *Address it:* when the developer fixes the code and removes the discovered leak, clicking on this button the discovery will disappear;
- *Not relevant:* the tool can find some matches that correspond to leaks but the user may not really care about them.

4.3 Web Application Server

The web application server involves two main parts. In the first one, contained in the “server.py” file, the Flask framework was used. It is a lightweight framework for Web Application “designed with the ability to scale up to complex systems” [26]. So when a user performs a request, the server use Flask to give a response. The second one was created in order to wrap all the logic of the tool and to communicate with the database, it is a Python package that exposes a set of API. These API are consumed when the Flask functions are triggered. If this package is used just with the database, the tool becomes completely usable even without all the web application infrastructure. These API involves:

- *Scan:* this function encapsulates the logic of the GitHub Scanner mentioned before. Given a URL and a set of regular expression returns in output a list of discoveries. The scan logic will be explained in detail in the next chapter. As it is shown from the code below, it is possible to choose the desired models that filter false positive just declaring them in the API;
- *Get/Add/Delete repository:* the API for all the operation related to the repository management;
- *Get/Add/Update discoveries:* the API for all the operation related to the discoveries;
- *Update similar discovery:* an API for updating all the similar discoveries between a single discovery tagged as false positive and all the discoveries related to the considered repository (the improved manual check mentioned before);
- *Get/Add/Add from file rules:* the API for all the operation related to the rules.

An example of the usage of the Python package can be found in Figure 4.6. It used for the database connection and scanning repositories and here it is shown how a scan process happens.

4.4 Database

As it was said before, a relational database was chosen, PostgreSQL in particular. It is an open-source database with “strong reputation for reliability, feature robustness, and performance.” [25].

```python
1 from sccsclient.cli import Client
2
3 URL = "https://github.com/adversarial-scan/VideoAnalytics_0"
4 c = Client(dbname="scs", dbuser="test", dbpassword="test")
5 c.add_rules_from_files("rules.yml")
6 c.scan(URL, models=["PathModel", "SnippetModel"], visualize=True)
7 discoveries = c.get_discoveries(URL)
8 print(discoveries)
```

Figure 4.6: Example of Python package usage.

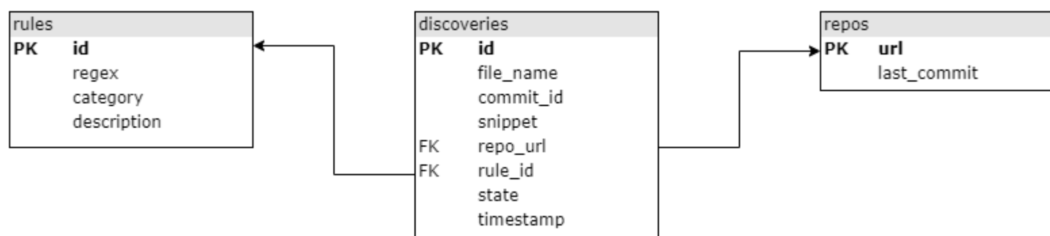


Figure 4.7: Database structure.

The database is structured in the following way (arrows point to foreign keys). Three tables were considered, on for the repository uniquely characterized by their URL, one for the rules, including a category and a description, and one for the discoveries. A discovery is related to one regular expression and a repository and these are the saved information:

- *ID:* it uniquely identify a discovery;
- *Commit ID:* it is the commit id of the founded discovery;
- *File name:* it is the path of the file where the discovery was found. This field is the one used in the Path Model;
- *Snippet:* it is the line that contains the regular expression match. This field is the one used in the Snippet Model;
- *State:* state of the discovery (i.e. new, false positive, not relevant, addressed);
- *Timestamp:* date and time when the discovery was found.

Chapter 5

GitHub Scanner

In this chapter it is be explained how the repository scanner works, so basically how the "scan" API seen before performs the real scan. The approach it was chosen to adopt is similar to the one of TruffleHog [13], the search goes through every commit on every single branch of a repository and the scanner is built to be as fast as possible. A library for fast regular expressions matching called Hyperscan is used [27].

5.1 Hyperscan

Hyperscan is "high-performance multiple regex matching library available as open-source with a C API", it "uses hybrid automata techniques to allow simultaneous matching of large numbers of regular expressions across streams of data" [27]. It was developed by Intel Corporation.

The choice of Hyperscan came out after an article where regular expression engines were compared was found. Starting from the same hardware and software configuration, lot of them were compared in terms of speed and execution time in matching a set of 17 regular expression. The result showed that Hyperscan was the one with the best performances [28]. The Hyperscan compiler API takes regular expressions as input and converts them into a compiled pattern database that can then be used to scan data. Hyperscan in the project is used when the set of regular expression the user upload have to be matched in a considered line of code.

Of course, all the project was developed in Python, but we found an unofficial CPython extension for Intel's Hyperscan that was integrated into the project [29].

```
1  def stream(self, rules):
2          """ Load the hyperscan database. """
3          self._stream =
              hyperscan.Database(mode=hyperscan.HS_MODE_BLOCK)
4          patterns = []
5          for r in rules:
6              rule_id, rule, _, _ = r.values()
7              patterns.append((rule.encode('utf-8'),
8                               rule_id,
9                               hyperscan.HS_FLAG_CASELESS |
10                              hyperscan.HS_FLAG_UTF8 |
11                              hyperscan.HS_FLAG_UCP))
12
13         expressions, ids, flags = zip(*patterns)
14         self._stream.compile(expressions=expressions,
15                              ids=ids,
16                              elements=len(patterns),
17                              flags=flags
```

5.2 Scan API

Starting from the description of the API:

```
1  new_discoveries = c.scan(repo_url=REPO_URL,
2                           category=CATEGORY,
3                           scanner=SCANNER,
4                           models=MODELS,
5                           exclude=NO_RULES,
6                           visualize=VISUALIZE)
```

The arguments are:

- *REPO_URL:* the URL of the repository that is going to be scanned;
- *CATEGORY:* the category of rules to be used for the scan. If no category is selected, the scanner uses all the rules that are stored in the database;

- *SCANNER:* the name of the regular expression matching engine. If nothing is specified, Hyperscan will be used;
- *MODELS:* a list of models that we want to apply to automatically classify false positives (the models are applied in cascade, sequentially). The available ones are the Path Model and the Snippet Model;
- *NO_RULES:* a list of IDs of rules that we do not want to use. For instance, if we want to use all the keywords rules except the one with id=5, then we set CATEGORY equal to keywords and NO_RULES equal to 5;
- *VISUALIZE:* True if we want visual feedbacks (progress bars) when the scan is in progress, False otherwise (the default choice).

5.3 Scanning process

The scanning process involves all the steps required for going from the URL of a repository, given as input, to the line of code that will be the input of the regular expression engine. It is important to navigate through every commit of every branch because a plaintext secret can potentially be in every part of the GitHub history. For this objective the GitPython library was used [30].

For a better understanding of the scanning process, let's consider an example of a GitHub network graph displaying all the branches and the commit of a repository (Figure 5.1).

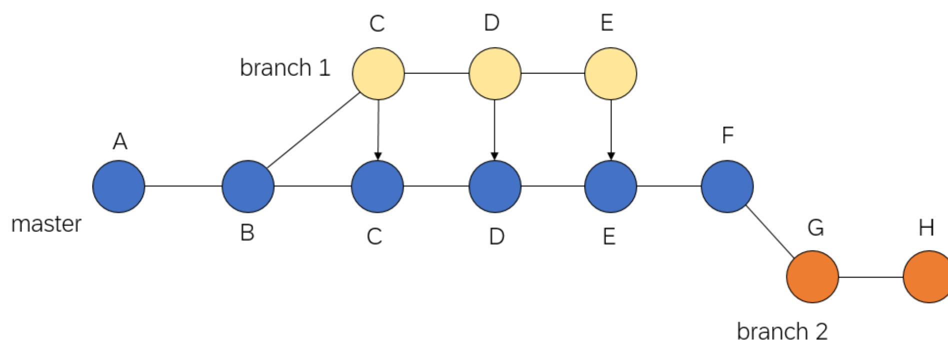


Figure 5.1: GitHub network graph example.

In this example we have three *branches*, "branch1", "branch2" and "master", and every letter represent a single *commit*.

The *diff* between two commits is the collection of all the changes from a commit to the previous one. Here is an example of a *diff*:

```
6 test.py
@@ -8,10 +8,10 @@
8     c = 6                                          8     c = 6
9                                                    9
10    # calculate the discriminant                   10    # calculate the discriminant
11  - d = (a**2) - (4*a*c)                           11  + d = (b**2) - (4*a*c)
12                                                   12
13    # find two solutions                           13    # find two solutions
14  - sol1 = (-d-cmath.sqrt(d))/(2*a)                14  + sol1 = (-b-cmath.sqrt(d))/(2*a)
15  - sol2 = (-d+cmath.sqrt(d))/(2*a)                15  + sol2 = (-b+cmath.sqrt(d))/(2*a)
16                                                   16
17    print('The solution are {0} and               17    print('The solution are {0} and
  {1}'.format(sol1,sol2))                             {1}'.format(sol1,sol2))
```

Figure 5.2: GitHub diff example.

In this case, all the line that start with a "+" (added lines) and the one that starts with a "-" (removed lines) constitute *diff*.

The scanning process involves the following steps:

1. *clone the repository:* in this way a copy is temporary available locally and can be scanned;

2. *loop over the branches:* so in the example case, the tool analyzes "branch1", "branch2" and "master" branch;

3. *compute the diff between commits iteratively:* starting from the last commit and going backward, the diff is computed between the last commit and the one before. In the case of "branch 1" the difference is computed at first between the commits E and D and then between D and C. The last diff, the one that involves the first commit, is calculated between the latter and a null commit;

4. *computed diff are saved:* this is done for avoiding checking a diff more than once. This is the case of "branch 1" and "master", they have the diff between E and D and the diff between D and C in common because "branch 1" was after merged into master. It is computed and saved in a global list the hash of the sum of the commit IDs of every analyzed diff. Before analyzing a new diff it is checked if its hash already exists in this list. If it does, the scan of

the diff is avoided, otherwise the hash is saved and the diff scanned. This operation speeds up the whole scanning process;

5. *diff are split into lines and filtered:* after splitting into lines the computed diff, the lines that start with '-' are filtered because are removed lines. Of course these lines are present with a '+' in some other diff, corresponding to the moment when they were added so it is just for avoiding analyzing the same line twice;

6. *single lines are passed to the regex engine:* in this final part, the computed lines are analyzed by the regular expression engine looking for matches;

7. *all the matches are saved:* they are saved in a data structure that contains for each line its commit ID and the file path where it was found. These discoveries are inserted into the database as "new" discoveries;

8. *remove the cloned repository:* after the insertion into the database the temporary cloned repository is deleted.

id [PK] integer	file_name text	commit_id text	snippet text	repo_url text	rule_id integer	state states	timestamp text
79	sortable-list/script.js	3030324c65b40828b27003ce9c985d...	+public float new_password : { modify { return 'testDummy' } }	https://githu...	21	new	Mon Feb 24 09:...
80	sortable-list/script.js	3030324c65b40828b27003ce9c985d...	+password = UserPwd.release_password('test_password')	https://githu...	21	new	Mon Feb 24 09:...
81	sortable-list/script.js	3030324c65b40828b27003ce9c985d...	+delete(new_password=>'abc123')	https://githu...	21	new	Mon Feb 24 09:...
82	sortable-list/script.js	3030324c65b40828b27003ce9c985d...	+password => permit('put_your_password_here')	https://githu...	21	new	Mon Feb 24 09:...
83	sortable-list/script.js	3030324c65b40828b27003ce9c985d...	+double password = return() {credentials: 'soccer'}.encrypt_password()	https://githu...	21	new	Mon Feb 24 09:...

Figure 5.3: Example of discovery inserted into the db after the scanning process.

Once the scan is finished and all the discoveries are correctly saved into the database, the models can be applied for false positives filtering.

Chapter 6

Path Model

In this section it will be explained how the Path Model works and how it is trained, starting from the concept behind, the training data, the model choice and the obtained results. It is the first model that compose the Leak Classifier, the filtering model applied once the Scanner finishes analyzing a repository. As the name suggests, starting from the discoveries saved in the database, it takes as input the path of them and performs binary classification, predicting if a discovery is a real leak or a false positive.

6.1 Model idea

When a lot of data were collected scanning repositories on GitHub, an analysis was done for investigating the nature of them. According to some observations a large part of the false positives is dummy data mainly used for code testing purposes. It is the case of pieces of code like this:

```
1 ...
2 # access to the database
3 url = "myDB/db"
4 login = "user"
5 password = "fakepass"
6 ...
```

It was discovered that an indication about the testing purpose of a source code can be captured from the file name or the path. It is not difficult to find pieces of

code like the one above in path like "/src/test/test.code". Another good percentage of false positives concerns documentation files. Documentations always contain fake tokens to explain how the application consumes them for security purposes. The rest of the false positives are related to the misinterpretation of correct token declarations. Starting from these intuitions and after the collection of a reasonable amount of discoveries and their related information about the path, the data were analyzed in order to find a hidden correlation between them.

6.2 Path Analysis

The repositories to scan were chosen randomly, so without specific criteria, choosing projects with different kinds of programming languages. The purpose was to conduct the analysis on all the file names and paths of discoveries that have been tagged as false positive.

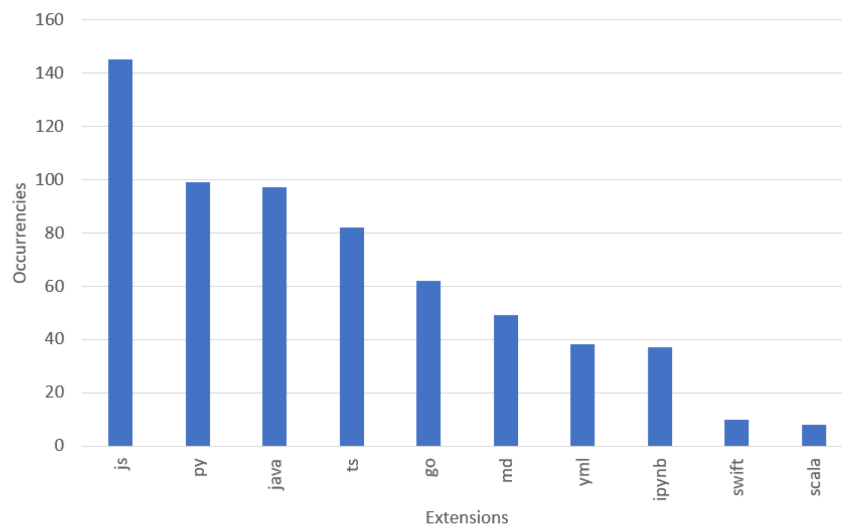


Figure 6.1: File extension distribution in paths.

One of the biggest problems at the beginning was that a lot of discoveries were available, but none of them were tagged. Of course, the Scanner alone is not able to catalog false positives and that's why some of the discoveries were manually tagged and the study was conducted on the paths of these.

The problem of tagging discoveries was not so hard because, fortunately, the amount of true positive found was a little percentage. After tagging around 10.000 discoveries the analysis was done using the two main libraries available in Python: *matplotlib* and *pandas*.

Of course, a lot of the paths of these tagged discoveries were duplicates because, for example, in a single file the scanner could detect more than one discovery. For this reason, starting from 10.000 tagged path, the number has been reduced to about 1.500. In Figure 6.1 the distribution of the file extension in the considered repositories.

As a first analysis it was checked what was the percentage of test files, the one that contains the word test in the path, and documentation files, such as markdown and textual files.

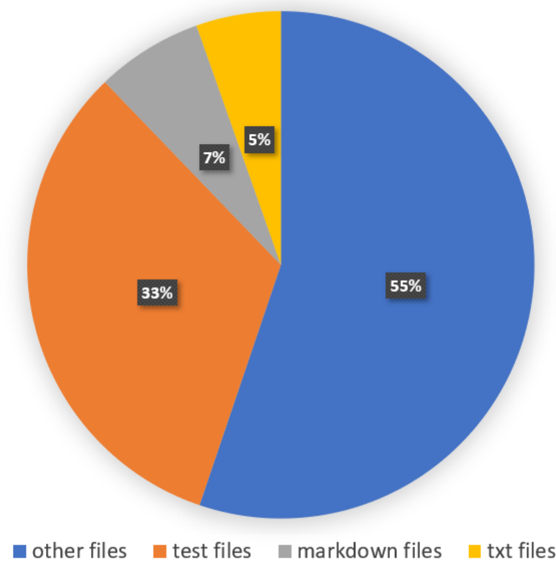


Figure 6.2: Percentage of test, txt, and markdown files.

From the graph above, it is possible to see that there is a high percentage of test files among the false positives, of course because of code used for testing purposes.

These results led to think that there are keywords contained in the paths that can be an indication of the nature of the discovery. For this reason it has been thought to divide every path in single words splitting it by the character "/" and then to count the occurrences of every single in the whole dataset of paths to determine the frequency of each word.

From Figure 6.3 it is clear that many words found can be traced back to the analysis made before. This is the case, for example, of words like "test", "readme" and "md". Other words, on the other hand, are of little significance for the analysis.

At this point, to extend the analysis, paths not manually tagged as false positives were also considered, for a total of about 45.000 distinct paths. A better pre-processing has also been applied to the path, in particular:

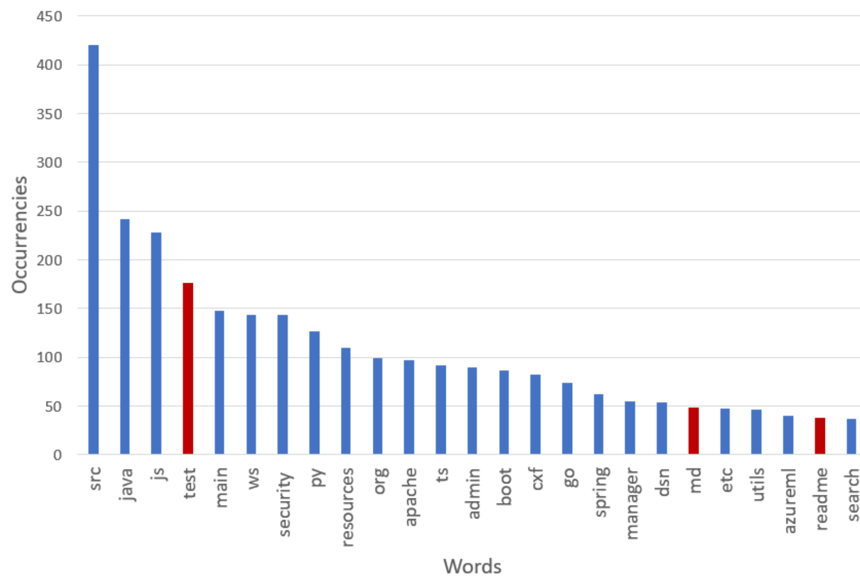


Figure 6.3: Word distribution in paths.

- Extensions were removed;
- Words were lowercased;
- Stemmization was applied. It is an algorithmic process that automatically removes the prefix and the suffix of a word in order to get the radix of it;
- Numbers were removed.

In this way (Figure 6.4), words like “tests”, “test123”, and “testing” will be considered as the single word “test”. Even lowercase is important, because file name like “README” and “readme” should not be treated differently.

A further analysis conducted concerns not only the words that make up the path taken as single units but taken as bi-grams and tri-grams. In this case stemmization was not applied .

After this analysis (Figure 6.5) it was found that the word “test” is the most common word in a path where the scanner detects a match, Also words like “example”, “dummy” or “tutorial” that are less frequent are still related to false positive data. For the bigrams, we have a lot of words that are associated with the word “test”, keywords like “lib” or “modules” that are related to libraries file and can be considered as false positives, and bigrams like “how to” that are releted to paths used for explanations. The conclusion was that there are specific keywords,

Word	Number				
test	22198	addon	2531	main	1773
src	7781	layout	2506	exmpl	1752
web	7357	node	2352	client	1753
modul	5750	blink	2232	kubernet	1729
lib	5449	platform	2179	fast	1664
third	5295	expect	2154	offscreen	1647
parti	5295	org	2148	user	1598
ansibl	4057	wpt	2135	network	1591
com	3201	servic	2023	plugin	1559
vendor	3127	canva	1931	auth	1536
github	2656	secur	1908	tutorial	1535
kit	2532	api	1836	integr	1405

Figure 6.4: Pre-processed word distribution in all paths.

WordCombination	Number				
third party	5266	expected txt	2066	wpt web	1785
github com	2582	web tests	2023	node modules	1778
party webkit	2526	blink web tests	2009	offscreen canvas	1646
third party webkit	2526	party blink web	2009	platform tests offscreen	1557
lib ansible	2474	blink web	2009	tests offscreen canvas	1557
ansible modules	2393	tests wpt	1854	tests offscreen	1557
party webkit layouttests	2384	platform tests	1785	vendor github com	1425
webkit layouttests	2384	web platform tests	1785	vendor github	1425
party blink	2158	wpt web platform	1785	http tests	1315
third party blink	2158	tests wpt web	1785	how to	1287
lib ansible modules	2099	web platform	1785	test go	1226

Figure 6.5: Bi-grams and tri-grams distribution in all paths.

bigrams or trigrams that can be related to false positives data and it is possible to classify a discovery thanks to these keywords.

6.3 Model building

This problem can be tackled just using regular expression for matching the specific keywords related to false positives or building a Machine Learning for path classification. ML model was preferred for two main reasons:

- The possibility to retrain a Machine Learning model with new data;
- Its speed in predicting respect to regular expression is better as the number of keywords to find increases.

Since the choice was to build a model that classifies a discovery based on its path, the main problem was that a labeled dataset was not available. The problem has been solved by assigning labels through keywords search within the pre-processed paths. Starting from the previous analysis, a set of specific keywords, that surely are related to a false positive match, were saved and used for labeling.

Once the dataset was ready with 4.5000 labeled entries, some data augmentation was performed at least to reach 50.000 entries. It was done taking an already false positive entry, removing a random word from its path and replacing it with a random keyword.

After that there was the need for a text representation method and a Machine Learning model to test if the idea was good.

Three different situations were compared:

	Precision	Recall	F1-score	Training time	Prediction time
BOW +XGBoost	100%	97.98%	98.98%	4119.3 s	2.563 s
TFIDF + XGBoost	100%	96.55%	98.24%	3531.6 s	3.437 s
fastText for classification	99.98%	99.55%	**99.76%**	**2048.4** s	**0.147 s**

Table 6.1: Comparison between tested architecture for building the classification models.

In all these three situations the models were trained in the same machine, with the same amount of data and the same true positive/false positive ratio. The best choice was the use of FastText not only because of its slightly better metrics but also because of a faster prediction time and training time. The training time was also a parameter of evaluation because an improvement in the architectures could be the possibility to retrain the model based on new data that a user tags as false positive. The reason behind its speed is that it learns word representation while

doing classification so it is better compared to the other tested models. FastText is also really useful in this situation because with its character level n-grams it can output vectors for out-of-dictionary words. It is an important advantage for path classification because a lot of out-of-dictionary words can appear doing this pre-processing. The hyper-parameters used for FastText were a learning rate of 0.1 and 100 epochs to allow the model to learn faster, and wordNgrams of 3 because the order was also important.

Once the model was trained on the data, it was saved and inserted in the tool just after the Scanner. So once a repository is scanned, the path of each discovery is pre-processed and sent to the model that predicts if it is a false positive or not. In the case of false positive the field "state" in the database is updated.

This is the final pipeline:

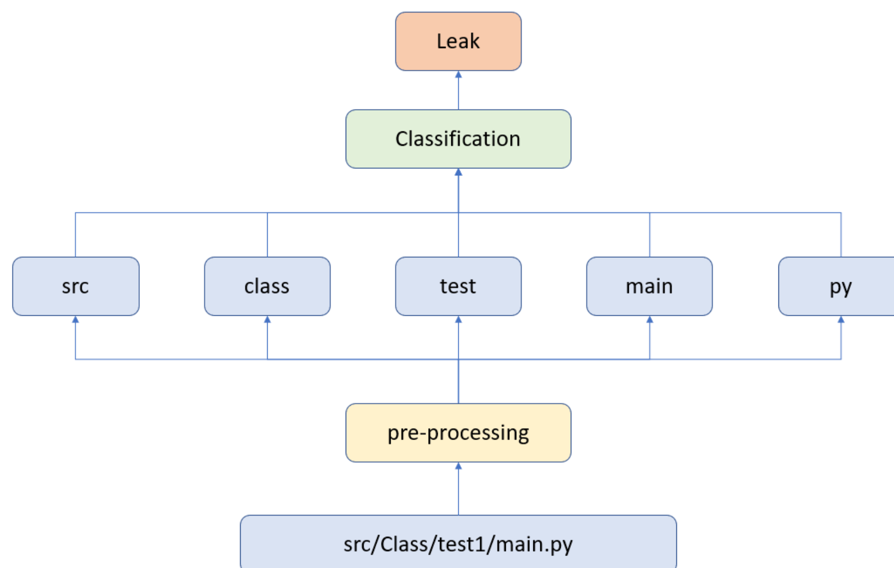


Figure 6.6: Path model structure.

Chapter 7

Snippet Model

This section covers the Snippet Model, the one that takes the code snippet as input and predicts non critical data after Path Model. It is also explained the idea behind, the dataset used for training, that is synthetic in this case, and the whole pipeline starting from the pre-processing to the training of the model and its parameters.

After that, the manual review phase is described and, in particular, how Machine Learning is used to optimize this part.

7.1 Model idea

The main problem to face for building an architecture capable to classify a snippet was the nature of the code snippet itself. It is a string obtained by matching a regular expression on a line of a file and, of course, it can be very heterogeneous. As a consequence, it can either be a line of code coming from a programming language file or, potentially, a line coming from any other kind of file as GitHub has no restriction in the type of file that can be uploaded.

An approach similar to Path Model does not fit this situation because there is no specific keyword to find, and data can not be labeled in the same way. In the meanwhile, no labeled data were available to start thinking about a model.

Doing some analysis, it was discovered that there are two different characteristics that a non critical data has:

- *The absence of a structure:* there is a high probability to be a false positive if the discovery doesn't present a real structure or follow a specific pattern. This

is because sometimes the Scanner, using regular expressions, matches some lines that are not that relevant, probably coming from a commented piece of code or some file with an uncommon extension. These kind of discoveries are difficult to analyze even for a human;

- *The presence of mock credentials:* these are easier to recognize because they follow a specific structure. Mock credentials are used in order to simplify the comprehension of some pieces of code.

On the other side it is easy to understand what is the structure of a discovery that has a high probability to be a leak. It has a *key-value* structure where the *key* is the variable where the secret is stored and the *value* is the secret itself.

It was found that the false positives characterized by an absence of a structure can simply be excluded with few heuristics. So, after that, the idea was to extract the key-value pairs from the snippet. Two approaches were possible, either to build a model for each programming language or build a unique model that is language agnostic. The first approach, used also for the “CodeSearchChallenge” [31], was not scalable for our solution because, even if it is built a tool that uses ten different models corresponding to the ten most common programming languages, a relevant percentage of files would not be covered anyway. Moreover, since we are analyzing simple lines of code and not entire pieces of code, different models were not really useful.

So the objective became to build a model that can be applied independently of the programming language and able to extract these kinds of entities. It was evaluated that the majority of false positives could be flagged as such only considering words (so excluding any particular or language-specific symbols). For this reason, a good pre-processing was done on the snippet in order to remove special characters and only to keep words. In this way, code snippets are analyzed purely on semantic level, by examining the meaning of the variable name, function name and their combination. It was called **Extractor**.

Once the key-value pairs were extracted by this first model the idea was to build another model, called **Classifier**, which takes as input the output of the previous model and classify the key-value pair as false positive or not.

The overall structure of the model can be found in Figure 7.3.

The two models are trained separately with different data, and they are used one after another. The big advantage of keeping them separate is that the whole architecture becomes more explainable and easier to test. In both the models the problem was treated as a classification task and FastText library was used.

```
1 ...
2
3 String username = "Carlo";
4
5 String password = "qwerty123!";
6
7 Credential = new Credential(username, password);
8
9 ...
```

(annotations: key = username, value = "Carlo")

Figure 7.1: Key-value example in discoveries.

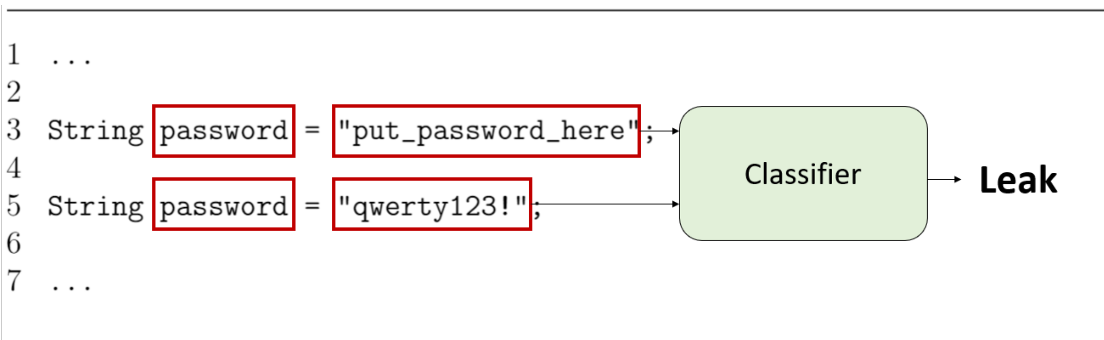


Figure 7.2: Classifier example.

7.2 Dataset creation

Another important issue that was faced was that no labeled data were available for training the models because the only available dataset was the one that came from the Scanner. The process of labeling data manually would have been too long and the model to work efficiently needs a lot of data.

In addition, this dataset contains critical data that could harm users' privacy in case they were used for malicious purposes and, for this reason, it is not secure to use this sensitive information for model training. There could be some adversarial attack that, starting from the models, can retrieve this information. Because of this, for both the models it was chosen to use a synthetic dataset.

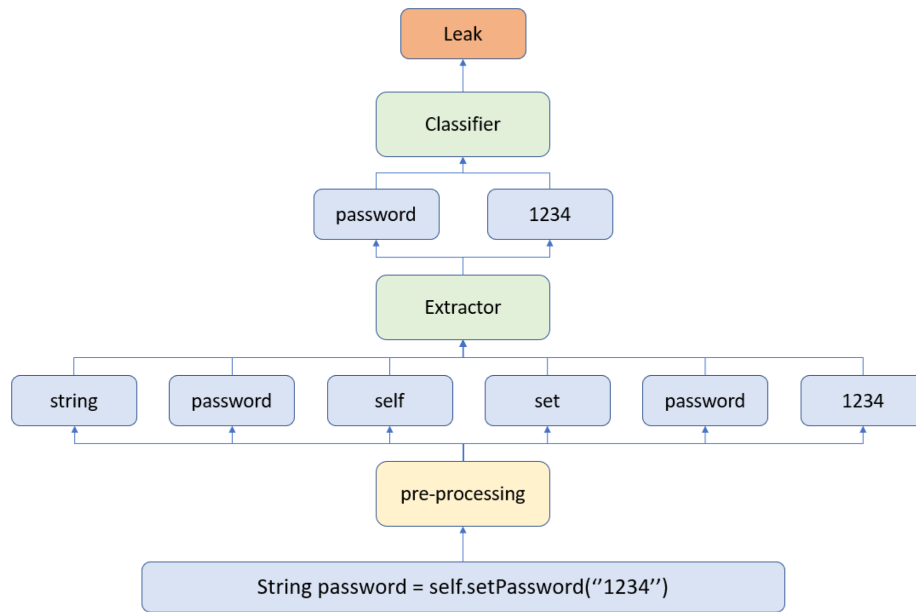


Figure 7.3: Snippet model structure.

7.2.1 Extractor dataset

For the extractor it was needed a code snippet with its relative key-value pair. For this reason, just analyzing the collected snippets, 28 code snippet patterns were collected, they can be found in Table B.1. These patterns are typical from different programming language or configuration files and each of them is characterized by a key-value pair.

Each pattern contains some elements that, if replaced with a proper value, generate a code snippet. These elements are:

- *Keys:* variables where secrets are stored.
- *Values:* random value for a secret. It is possible to generate a fake secret or a real secret depending on the used word.
- *Objects:* names of objects used in a snippet.
- *Methods:* names of objects used in a snippet.
- *Function:* names of objects used in a snippet.
- *API keys:* list of fake but realistic API keys from different providers.

Some examples of the generated code snippet could be:

```
1 token_uri = "DummyToken"
2 this->client_id = 'test_dummy'
3
4 # generated real leak
5 String password = Base64.replace_password('123456789')
6 self: {email: user.email, client_email: 'marie.curie@gmail.com'}
```

Both the patterns and the elements that compose each of them were generated after a careful observation of the code snippet caught with regular expressions.

Around 200.000 different code snippets were generated to use as a training set, with a 50% repartition between real secret and false positives. The labels assigned to each snippet were the key and the value elements placed in the pattern.

7.2.2 Classifier dataset

In this situation the needed dataset was a list of keyword that corresponds to a secret with an associated binary label, true if it is a false positive, false if it is not.

The data that corresponds to real secrets were generated based on realistic secret tokens such as API key, password or email addresses. On the other hand, data that corresponds to fake tokens were generated looking for them in the available code snippets. This is an example of tokens:

put_your_key_here
dummyToken
example
Fake Secrets

qwerty123!
AIzaSyClUpQXesfIbeargYUzT69KqImSGZEs7r4
dr.john.rabbit@outlook.com
Realistic Secrets

Figure 7.4: Example of training data for the Classifier.

7.3 Data pre-processing

Once the labeled dataset was built for both the models, data have to be pre-processed before using them for training the model. The process is similar to the one already applied for Path Model as we want to extract word units from a text. Only the Extractor needs the data to be pre-processed, the Classifier doesn't because its input is the output of the first one.

These are the applied transformations:

- Snake case strings are converted in camel case convention.
- Special characters are removed, such as dots, commas, parenthesis etc.
- Numbers are removed.
- Words inside the quotes are not pre-processed in order to preserve the possible presence of secrets.
- Lines that contains comments are splitted.

Here some examples:

```
1      raw_data = "this is my foo(secret_token_auth, )"
2      print(self._pre_process(raw_data))
3      #['this', 'is', 'my', 'foo', 'SecretTokenAuth']
4
5      raw_data = "self.do_something(password="3733tPwd!")2
6      print(self._pre_process(raw_data))
7      #['self', 'DoSomething', 'password', '3733tPwd!']
8
9      raw_data = ""password2": "#####", "!@AAA12")"
10     print(self._pre_process(raw_data))
11     #['password', '####', '!@AAA12']
```

During the pre-processing phase a lot of discoveries were also excluded because of the application of some heuristics. This is the case of code snippets that come from a file that belongs to the most common programming language (i.e Python, Java, c++, etc.), but don't contain quotes, that are for sure false positives and they're tagged as such. Also, lines with more than 400 character were excluded because almost all the time were meaningless.

7.4 Model training

The choice for the model to use for this task fell on FastText for reasons similar to those of the Path Model. FastText is faster respect to other kinds of models and can catch out of dictionary words.

The Extractor model has to predict the position in the sentence of the key-value pair. The training dataset corresponds to the pre-processed snippets with two labels for key and value positions. In this way the problem can be seen as multi-label classification. Hierarchical softmax is used due to a large number of possible classes. The hyper-parameters used were a learning rate of 0.1 and 50 epochs to allow the model to learn faster, and wordNgrams of 5.

```
Pre processing the extractor dataset...
100%|██████████| 200000/200000 [01:36<00:00, 2079.91it/s]
Read 1M words
Number of words:  516
Number of labels: 9
Progress: 100.0% words/sec/thread:  246811 lr:  0.000000 loss:  0.015874 ETA:   0h 0m
(40000, 0.999725, 0.5109044218165093)
```

Figure 7.5: Example of extractor training.

The Classifier model has to predict if a given key value pair is a false positive or not. It is a binary classification problem and it was also performed with FastText library. The hyper-parameters were the same as the Extractor except for the wordNgram that was set to 1 in this situation because not really useful.

7.5 Model integration

Once the models are trained, they are integrated into the tool after the scanner. After the Path Model, the Snippet model act as a function taking as input a snippet, pre-processing it as it was done in the training phase and using the models to obtain the output of the classification.

Once the key-value pair is extracted and classified, if a snippet, and consequently the whole discovery, is false positive, the state is changed in the database.

7.6 Manual review

Once Machine Learning models have been applied to the discoveries, the user can still label manually the remaining snippets that he considers as false positives. Also

this process has been improved with the use of FastText library, but this time it was used the API to train a model for text representation. FastText was chosen because, as it extracts features at character level it fits in this situation.

A separate model for learning the word representation was trained on the same corpus composed by 200000 code snippets used for the Extractor. This model, added in the project, is used when a user tag as false positive a discovery. At that moment the word representation of the snippet is computed and it is compared with the vector representation of every other snippet from the same repository. If the value of the cosine similarity is above a threshold, that was chosen to be 0.95, the algorithm tags as false positive even similar discoveries.

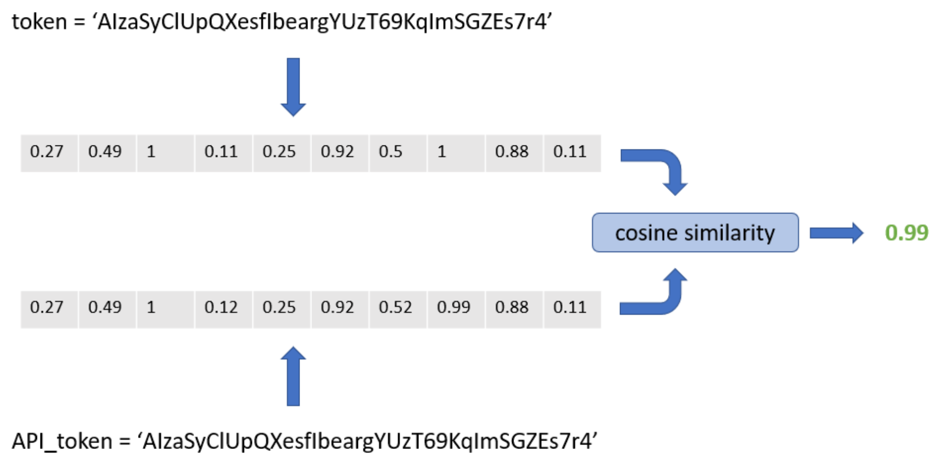


Figure 7.6: Example of similarity check.

In Figure 7.6 it is possible to see an example of similar strings. When a user tags as false positive the first one, the algorithm understands that it has a high degree of similarity with the second one so both are tagged as false positive.

This technique can speed up the manual review workload for removing false positives. Just with a single click lots of discoveries can be removed.

Chapter 8

Evaluation

In this section, it is presented an evaluation of the tool, divided into three major parts.

Firstly, in Section 8.1, the aim is to evaluate the current status of false positive data in repositories. 1000 repositories from the public GitHub and 300 repositories from a GitHub-like code versioning platform owned by a private company were scanned to do this analysis. For this reason, in the remainder of this section, we will refer to GitHub as *public GitHub* and to the repositories publicly available on this platform as *public repositories*, while we will refer to the privately owned GitHub platform as *proprietary GitHub* and to its repositories as *proprietary repositories*.

In the second part, in Section 8.2, the built Leak Classifier is evaluated in a controlled environment. That's because no labeled dataset is available where to test the models so it was decided to build our own poisoned project.

In the last evaluation, in Section 8.3, part the tool is compared with other available tools that try to tackle the same problem.

8.1 GitHub credentials leaks status

In three months, 1000 public repositories on GitHub were randomly selected and scanned. The list of regular expressions used can be found in Table A.1. Over 14 million discoveries have been found, with 13.6 million in public GitHub repositories. Our discoveries cover more than 30 programming languages and represent more than 300 file types. Figure 8.1 represents the 10 most common file extensions containing leaks in our dataset. The number of contributors and the sizes of the repositories has been chosen equally distributed.

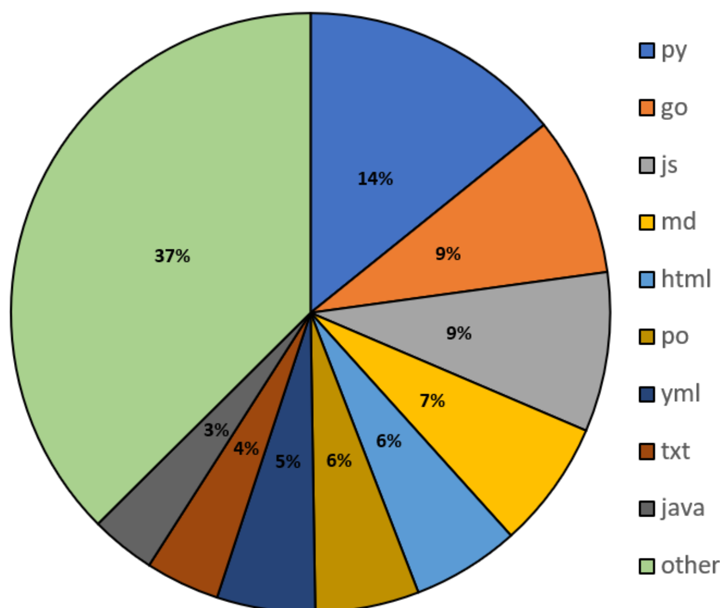


Figure 8.1: Distribution of the most common extensions containing secrets

It was noticed that API keys are still widely published in open-source projects, as shown also in [11], but they do not represent the majority of the discoveries. In our study, we find an important number of passwords giving access to local and remote databases, or to e-mail accounts. Unfortunately, these actual leaks are mixed with a vast majority of non critical ones, wrongly detected as leaks (false positives), which makes it impossible for a developer to review everything manually. These passwords are mostly undetectable by traditional scanning tools, but they are still easy to find for someone using a simple search tool with keywords such as *remove credentials*, *password removed*, etc., in the commit message. Furthermore, when an actual password is detected, the access granted may be far more dangerous than an API Key, possibly giving access to e-mail accounts, social media or banking details. In the evaluation, many passwords that it was supposed to be real were found (even if there is no absolute certainty because they haven't been tested).

In the experiment, there were detected more than 40 million secrets in 579 out of 1000 public repositories (about 58% of them), with more than 26 million belonging to a very limited number of huge repositories. Thus, we also provide an evaluation based on a filtered set of public repositories, which may be pointed out as more realistic. Besides, we detected about 260k secrets in 268 out of 300 proprietary repositories (about 89% of them). As previously discussed, the vast majority of the secrets detected with a regex scanner consist of false positive discoveries. Since

these discoveries are supposed to be manually checked by developers, to the aim of reducing the burden caused by such an operation (that is prone to errors due to the high number of interactions needed), we have applied the path model and the snippet model to automatically classify the discoveries produced by our scan. The results are shown in Table 8.1.

Repositories	Discoveries	Path Model	Snippet Model	Total FP
public	40467007	17520412 (43%)	9936550 (25%)	22737099 (56%)
public (filtered)	13620876	9350402 (69%)	3285207 (24%)	11110534 (82%)
proprietary	259225	91180 (35%)	85599 (33%)	154521 (60%)

Table 8.1: False positive classification in repositories.

As expected, there is an overlap in the classification of the false positives produced by our models. This is due to the fact that a discovery may be classified as a false positive both according to its file path and to its code snippet.

Overall, both the models have classified more than 30% of the discoveries as false positive. In particular, the path model reaches almost 70% of false positive classification in the filtered dataset. This score is halved with the proprietary dataset. The reasons behind this are due to the fact that in these proprietary repositories test files and documentation are not used as commonly as in public repositories. In fact, this company provides a support service for their software, and this policy relapse on the bare documentation integrated into their codebase.

In order to assess the behavior of our models, we decided to perform a manual review of a limited number of discoveries. To do so, we randomly selected 100 discoveries automatically classified as potential leaks and 100 discoveries classified as non critical data by the models, and we manually analyzed each of them. The results are shown in Table 8.2.

<table>
<tr><td colspan="2" rowspan="2">classification</td><td colspan="2">models</td></tr>
<tr><td>potential leak</td><td>non critical data</td></tr>
<tr><td rowspan="2">manual</td><td>leak</td><td>20
(true positive)</td><td>1
(false negative)</td></tr>
<tr><td>non critical data</td><td>80
(false positive)</td><td>99
(true negative)</td></tr>
</table>

Table 8.2: Manual assessment of 200 discoveries after models classifications

Given 100 discoveries classified as non critical data by the models, we manually classified 99 of them as such (i.e., true negative), and we found only 1 occurrence of

a discovery being a potential leak (i.e., false negative). Yet, this discovery represents an edge case: the developer stated to use a certain password (in plaintext) to log into a service in a readme file (which violates the assumption that real credentials cannot be found in example or dummy files). Thus, in the scope of our study, we can state that the unclassified leak rate is negligible. Given 100 potential leaks, we manually classified 80 of them as non critical (i.e., false positive), and 20 of them as actual leaks (i.e., true positive). If we project the results of the manual assessment to the public dataset, the false positives ideally represent about 2 million discoveries. Although this false positive rate seems important, we can point out that it represents less than 15% of the discoveries, and that without the models (so considering only the scanner), the false positive rate is 95%, proving that our approach provides an efficient false positive filtering.

8.2 Leak Classifier evaluation

The main bottleneck in Leak Classifier evaluation was the amount of labeled data, which makes it really hard to assess efficiently the models for secret detection. Thus, it was decided to build our open-source project where secrets were injected and it is possible to generate a ground truth. The ground truth contains all discoveries that should be found in the poisoned repository and a label that tells if the discoveries are false positive or potential leaks. Once a ground truth is generated it can be compared with the results of the Leak Classifier in order to compute evaluation metrics such as recall, precision and f1-score.

8.2.1 Poisoned repository generation

For the creation of the poisoned repositories three repositories which contain no secrets were selected (the choice of the repository does not matter, its only purpose is to provide commit messages and filenames). After that, the repositories were cloned and the secrets, coming from the patterns in Table B.1, generated as described in Section 7.2.1, were injected in every commit. The ground truth was then computed and the poisoned repositories were pushed on GitHub ready to be scanned. Two different situations were analyzed:

- **Situation 1:** secrets are equally distributed between true secrets and false positives secrets but inserted just in files that are not included in the one detected by Path Model.
- **Situation 2:** secrets are equally distributed between true secrets and false positives secrets and inserted in every kind of files.

8.2.2 Evaluation

The three repositories were scanned in both Situation 1 and Situation 2 and a manual review is applied for the 0.1% of the discoveries. Of course, since the training dataset of the Snippet Model and the secret inserted in the project are sampled from the same source, we should expect to have good results. However, we want to see if the model can generalize to unseen patterns. The training dataset of the Snippet model was generated from 50 % of the pattern, to see how it impacts the ability of the model to predict unseen patterns.

After that the results are compared with the ground truth generated before, the metrics computed and then averaged for the three different repositories.

Situation 1	Precision	Recall	F1-score
Half pattern	74.16%	100%	85.16%
Half pattern + manual rewiew	74.52%	100%	85.40%
Full pattern	89.33%	100%	94.36%
Full pattern + manual review	89.69%	100%	94.56%

Table 8.3: Situation 1 evaluation metrics.

Situation 2	Precision	Recall	F1-score
Half pattern	82.80%	100%	90.59%
Half pattern + manual rewiew	83.33%	100%	90.91%
Full pattern	95.81%	100%	97.86%
Full pattern + manual review	96.19%	100%	98.06%

Table 8.4: Situation 2 evaluation metrics.

8.2.3 Discussion

As it is possible to see from both Tables 8.3 and 8.4, if the number of the pattern of data used for training decreases, also the precision of the model decreases but not significantly. That means that the model is capable of generalizing to unseen patterns. This is an important characteristic of the model because it means if

a discovery that has never been seen before is provided in input, the model still manages to extract the key and the value and classify the leak with good precision. So even with a small number of patterns, the model can give good results. On the other hand, if we decide to include more complex secrets into the training data, we might improve the accuracy for those secrets, but it will impacts the score of simpler secrets; indeed, if we increase the possible classification labels (as a word position in a sentence), we will underperform in binary classification. A universal model will struggle to achieve satisfying precision on both complex secrets (in a highly typed language, or object-oriented language) and simpler secrets (environment variable).

The performances are better in 8.4 because discoveries are also injected in files that are easy to recognize by Path Model. A separate discussion has to be made for the Path Model because its performances are related on the keyword it is trying to detect in the path.

8.3 Comparison with other tools

Since the problem of leak detection in public open-source projects is not new, tools have been developed to tackle it. To our knowledge, there is no open-source tool that scans GitHub repositories using machine learning. Still, we can compare SAP Credentials Code Scanner with several tools we selected:

- *TruffleHog* : very popular (4.3k stars on GitHub) and open-source scanning tool. You have to provide your own set of regular expressions to the tool to detect secrets. it 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 such as git-secrets.

- *GitGuardian*: Company founded in 2016 and specialized in the detection of secrets in open-source places. Alongside their commercial offer, they provide free services to scan their own GitHub repositories. They advertise their tool as “machine learning powered” and they can identify “more than 200 API Keys patterns” according to their website.

TruffleHog and its variants aim to be a strong baseline for particular enhancements. Meli et. al offers improvements in their paper to TruffleHog’s algorithm. Various heuristics might be implemented to improve the accuracy of the tools, such as an entropy check: if a string has high entropy, meaning that it consists of seemingly random characters, the probability that this string is an API Key is high.

Several tests were performed on the GitGuardian platform on various API Keys patterns and on plaintext passwords. It was discovered that the platform was not able to detect plaintext credentials and detect only a reduced sample of API Keys, excluding big API Keys providers such as Facebook or Paypal. We solely tested the free version of GitGuardian and it might be possible that the full abilities of the platform is only enabled in the commercial offer. Another commercial tool called Nightfall AI (formerly known as Watchtower) offers the same services, but no API is available to test the tool.

Available tools were compared, on several criteria. First, on their scanning process, which techniques they use and if they are performing any false positives reduction. The open-source solutions do not perform false positive reduction (since they are not scanning the password for most of them), as opposed to commercial offers, where their key argument is the false positive reduction. The user experience is also a key point, how easy and intuitive the tool to be used efficiently. The price could represent an important barrier for small companies that want to protect themselves, so they tend to use free tools. Commercial products provide a user interface, making the tool more accessible to developers, and even non-technical papers. Finally, the presence of an important community leads to better support, scalability or constant tool improvements.

Category	Tool	Scanning process						User experience		Adoption			
		Regular expressions	Entropy check	Heuristics	False positive reduction	Mathematical models	Dig in every commit	Free	User interface	Open-source	Developper community	Scalable	Regular updates
Known algorithms	TruffleHog	●	◐	-	-	-	●	●	-	●	●	●	●
	Git-secrets	●	-	◐	◐	-	◐	●	-	●	◐	●	◐
	Meli et. al	●	●	●	◐	-	-	-	-	-	-	-	-
Commercial offers	GitGuardian	●	-	●	●	●	●	◐	●	-	◐	●	●
	Nighfall AI	●	-	●	●	●	●	-	●	-	-	●	-
Our tool	SCCS	●	-	●	●	●	●	●	●	●	-	●	●

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

Table 8.5: Comparison of available tools

Chapter 9

Conclusion

Secret detection in open-source projects is a highly important issue for companies and developers to secure their code and to protect themselves against malicious credentials miners. The development of this tool can have a great impact on the protection of personal information left on public source code repositories. A developer, using this tool can scan his repositories and remove all personal information left hardcoded with the awareness that his is code secure.

The study, starting from the analysis of existing works and tools on this topic, focused on how to improve the current state of the art. At first a scanner was developed, faster than the existing ones and able to dig inside the repositories, then we moved on mathematical models to reduce false positives. Looking at the result of the scanner we came up with the idea of two Machine Learning models, one for the path and one for the code snippet, capable to label not relevant data as false positive. The main problem encountered was the complete absence of labeled data where to train and test the model. The problem has been overcome with the usage of a synthetic dataset and the experiments for testing the tool have been designed to make it clear that even the use of synthetic data can give good performance in real situations.

Following the objectives set at the beginning, it was built an open-source tool that outperforms the currently available open-source tools, with the detection of both API keys and plaintext credentials, a low false positive rate as well as user-friendly interface to improve the tool with new data. The filtering process has a good precision as showed in the experiment done in Section 8.2 and from the study conducted in Section 8.1 it has been proved that a lot of false positives (82% in the case of public GitHub and 60% in proprietary GitHub) were filtered.

This tool certainly improves performance compared to the tools currently available, but it definitely has room for improvement.

9.1 Further improvements

This version of the tool can be considered as a first release, but for sure it can be enhanced. These are some of the possible improvements:

- *speed of the tool:* it is sufficient to think that the 1000 repositories selected for the experiment in Section 8.1 were scanned in more than 3 months. The main bottleneck of the tool is the Scanner because the more commits and branches there are, the more the tool takes to complete the scan of the repositories. Concurrency was not implemented yet and it could be a way to get some improvements;
- *support not only for GitHub:* there are other public source code repositories that are not supported by the tool (i.e. GitLab or BitBucket).

Research work can also continue for improving the current results especially with regard to:

- *precision of the model:* it can be always improved. One of the main problems in Snippet models is the chosen pattern for model training. If more complex patterns are included, the tool loses precision in recognizing simple patterns, and, on the other hand, if more simple patterns are included, the tool loses precision in recognizing complex patterns. This is because the used patterns are generic, but it can be considered to adapt them to the repository we choose to scan;
- *model retraining:* users can help with their manual review of discoveries in a better precision of the model. Their review can be collected and used for model retraining. Decentralized techniques such as Distributed Machine Learning or Federated Learning seem to be promising directions.

Appendix A

Regular expressions

<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>amzn\.mws\.[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}</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>This paper</td></tr>
<tr><td>Key word</td><td>sshpass -p.*['|\"]</td><td>This paper</td></tr>
<tr><td>Password</td><td>((root|admin|private_key_id|client_email|client_id|token_uri)
\s*(?:=|:|->|<-|=>|<=|==|«))</td><td>This paper</td></tr>
<tr><td>Password</td><td>((password|new_pasword|username
\s*(?:=|:|->|<-|=>|<=|==|«))</td><td>This paper</td></tr>
<tr><td>Password</td><td>((user|email|User|Pwd|UserName|user_name
\s*(?:=|:|->|<-|=>|<=|==|«))</td><td>This paper</td></tr>
<tr><td>Password</td><td>((access_token|access_token_secret|consumer_key |consumer_secret
\s*(?:=|:|->|<-|=>|<=|==|«))</td><td>This paper</td></tr>
<tr><td>Password</td><td>((FACEBOOK_APP_ID|ANDROID_GOOGLE_CLIENT_ID)
\s*(?:=|:|->|<-|=>|<=|==|«))</td><td>This paper</td></tr>
</table>

<table>
<tr><td>Password</td><td>((authTokenToken|oauthToken|CODECOV_TOKEN
\s*(?:=|:|->|<-|=>|<=|==|«))</td><td>This paper</td></tr>
<tr><td>Password</td><td>((IOS_GOOGLE_CLIENT_ID
\s*(?:=|:|->|<-|=>|<=|==|«))</td><td>This paper</td></tr>
<tr><td>Password</td><td>((sk_live|rk_live
\s*(?:=|:|->|<-|=>|<=|==|«))</td><td>This paper</td></tr>
</table>

Table A.1: List of regular expressions used by the Scanner

Appendix B

Patterns

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

Table B.1: List of pattern used in Snippet Model

Bibliography

[1] T. Mitchell, "Machine learning", McGraw Hill, 1997
[2] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space", 2013
[3] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation", Empirical Methods in Natural Language Processing (EMNLP), 2014, pp. 1532–1543
[4] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information", arXiv preprint arXiv:1607.04606, 2016
[5] T. Chen and C. Guestrin, "Xgboost", Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16, 2016, DOI 10.1145/2939672.2939785
[6] A. Conneau, H. Schwenk, L. Barrault, and Y. Lecun, "Very deep convolutional networks for text classification", 2016
[7] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification", arXiv preprint arXiv:1607.01759, 2016
[8] "Alexa Top Software Sites." https://www.alexa.com/topsites/category/Top/Computers/Software
[9] E. Phneah, "Github search shuts down after users' private keys exposed", 2013, https://www.zdnet.com/article/github-search-shuts-down-after-users-private-keys-exposed/
[10] "Removing sensitive data from a repository." https://help.github.com/en/github/authenticating-to-github/removing-sensitive-data-from-a-repository
[11] M. Meli, M. R. McNiece, and B. Reaves, "How bad can it git? characterizing secret leakage in public github repositories.", Proceedings of the 26th Network and Distributed System Security Symposium (NDSS), 2019
[12] "Git-secrets." https://github.com/awslabs/git-secrets
[13] "Trufflehog." https://github.com/dxa4481/truffleHog
[14] "Scumblr." https://github.com/Netflix-Skunkworks/Scumblr
[15] "Repo supervisor." https://github.com/auth0/repo-supervisor
[16] "Gittyleaks." https://github.com/kootenpv/gittyleaks

[17] “Git hound.” https://github.com/ezekg/git-hound
[18] “Semmle + github.” https://blog.semmle.com/secure-software-github-semmle/
[19] “Github security.” https://github.com/features/security
[20] “Gitguardian.” https://www.gitguardian.com/developer
[21] “Gitguardian blog.” https://blog.gitguardian.com/one-year-and-a-half-scanning-github-for-sensitive-data/
[22] “Nightfallai.” https://www.nightfall.ai/
[23] “Nightfallai githhub marketplace.” https://github.com/marketplace/watchtower-radar
[24] “Nginx.” https://www.nginx.com/
[25] “https://www.postgresql.org/.” https://www.postgresql.org/
[26] “Flask.” https://palletsprojects.com/p/flask/
[27] “Hyperscan.” https://www.hyperscan.io/
[28] D. S. S. Grunert, “A comparison of regex engines”, 2017
[29] “Hyperscan for python.” https://python-hyperscan.readthedocs.io/en/latest//
[30] “Gitpython.” https://gitpython.readthedocs.io/en/stable/index.html
[31] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “Codesearchnet challenge: Evaluating the state of semantic code search”, 2019