# Using Bigrams to Detect Leaked Secrets in Source Code

Anton V. Konygin[1,2,a], Andrey V. Kopnin[1], Ilya P. Mezentsev[1] and Alexandr A. Pankratov[1]
[1]*SKB Kontur, Russia*
[2]*N.N. Krasovskii Institute of Mathematics and Mechanics, Russia*

Keywords: Leaked Secrets, Source Code, Machine Learning, Security.

Abstract: Leaked secrets in source code lead to information security problems. It is important to find sensitive information in the repository as early as possible and neutralize it. By now, there are many different approaches to leaked secret detection without human intervention. Often, these are heuristic algorithms using regular expressions. Recently, more and more approaches based on machine learning have appeared. Nevertheless, the problem of detecting secrets in the code remains relevant since the available approaches often give a large number of false positives. In this paper, we propose an approach to leaked secret detection in source code based on machine learning using bigrams. This approach significantly reduces the number of false positives. The model showed a false positive rate of 2.4% and false negative rate of 1.9% on test dataset.

## 1 INTRODUCTION

A secret in source code is any sensitive information, such as passwords, API keys, certificate files, and configuration files. However, the term "secret" is often used in a narrower sense, excluding configuration or certification files (see, e.g. (Sinha et al., 2015), (Lounici et al., 2021), (Ding et al., 2020)). In this work, we follow this rule, and only passwords and API keys are called secrets.

The security of software systems relies heavily on the use of secrets. This is especially true for code hosted in public repositories, e.g., on GitHub[1]. In 2014, an Uber employee accidentally uploaded his credentials to GitHub, which resulted in the Uber database hack of 50,000 Uber drivers (Collins, 2016). Similarly, Amazon found 10,000 AWS keys accidentally left in the source code by Amazon developers and uploaded to GitHub (Knight, 2016). In 2016, researchers found over 1,500 Slack API tokens in public GitHub repositories belonging to major companies (Marlow, 2019). These tokens might provide access to exfiltrate internal communications or mine for other shared credentials, such as server or database access. Leaks such as these can have widespread effects beyond the individual service to which the leaked credential applied.

In (Meli et al., 2019) a large-scale analysis of secret leakage on GitHub is carried out. The authors examine billions of files collected using two complementary approaches: a nearly six-month scan of real-time public GitHub commits and a public snapshot covering 13% of open-source repositories. The research focuses on private key files and 11 high-impact platforms with distinctive API key formats. This focus allows to develop conservative detection techniques that the authors manually and automatically evaluate to ensure accurate results. They found that secret leakage not only affects more than 100,000 repositories, but also that thousands of new unique secrets are being leaked every day. The work shows that the leakage of secret data on public repository platforms is widespread and far from being solved, exposing developers and services to constant risk of compromise and abuse.

At the end of 2022, GitHub made it possible[2] to automatically detect leaked tokens (more than 200 patterns). Password detection remains a more difficult task due to the greater password variability.

One way to minimize the risks is to detect the secret as quickly as possible and start a procedure of neutralizing it. Thus, we get the task of detecting leaked secrets: by having code fragments as input, it is necessary to detect secrets if they are present.

The classic approach to solving this problem is to

[a] https://orcid.org/0000-0002-0037-2352
[1] https://www.github.com
[2] https://github.blog/2022-12-15-leaked-a-secret-check-your-github-alerts-for-free/

use regular expressions. This approach works well with API keys, especially when their format is known, but gives a lot of false positives for passwords. In general, this is to be expected, especially when the password can be arbitrary. Also, entropy analysis is often used to detect secrets. This method is based on the hypothesis that secrets have a higher degree of uncertainty than the rest of the code. The method performs better when secrets consist of random sequences of characters.

One of the approaches to improving the quality of secret detection is the use of machine learning (Saha et al., 2020). This approach allows you to customize the detector for a specific situation (domain-specific) without the need to write an explicit detection algorithm. Along with all of the advantages, machine learning methods have well-known drawbacks.

Since the data is sensitive (it contains passwords and API keys), the collected training datasets cannot be too large. At the moment there are approaches to solving this problem, e.g., using federated learning (Kall and Trabelsi, 2021), which is a machine learning paradigm allowing models to be trained on local data without the need for its owners to share it.

The problem with data is not the only issue. The main problem at the moment is a large number of false positives (see (Rahman et al., 2022)). This applies to both approaches based on machine learning and approaches which use classical methods. Despite the relevance of the task of detecting secrets in the code and the efforts made, a significant false positive rate hinders the implementation of effective open-source solutions (Lounici et al., 2021).

Thus, any new advances in solving the problem are welcomed, as they can lead to a reduction in the amount of sensitive information in the code and an increase in the security level of software engineering.

In this paper, we propose a bigram-based approach that allows us to detect secrets with low false positive and false negative rates. With the help of bigrams, we transform the string into a vector. Next, we use the CatBoost algorithm (Dorogush et al., 2018) to build a binary classifier. One of the features of our approach is that we use data from different distributions for training and testing. For training, we build a dataset based on public and automatically generated data. For testing, we use a private local dataset of passwords and API keys (it contains sensitive information and we cannot make it public). Thus, the model is tested in more complex conditions — the domain area is changed. However, the low percentage of errors during testing suggests that the solution obtained is stable and reliable. In the paper, we describe in detail the procedure for collecting a dataset for training.

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

- We present an approach to detecting leaked secrets in the code. The approach is based on bigrams and gradient boosting on decision trees.
- The proposed approach significantly reduces the number of false positives (false positive rate of 2.4% and false negative rate of 1.9% on test dataset) and does not require additional manual data labeling (public and automatically generated data are used).

The rest of this paper is structured as follows. Section 2 presents some related work. Section 3 describes the proposed approach. Section 4 presents the evaluation of the model. Finally, we make conclusions in Section 5.

## 2 RELATED WORK

Secret detection approaches can be divided into two types: based on classical approaches (e.g., regular expressions, entropy analysis) and based on machine learning.

The most straightforward approach is to use regular expressions. Such tools as `trufflehog`[3], `repo-supervisor`[4], `git-secrets`[5], `repo-secutiry-scanner`[6] use regular expression search, entropy checks or a combination of both to identify potential secrets. Recent studies (Lounici et al., 2021) show that the classic regular expression approaches generate a high false positive rate of about 82%. A different approach is used in (Sinha et al., 2015). It discusses the advantages and disadvantages of the following methods for detecting leaks of API keys: sample selection by using keyword search, pattern-based search, heuristics-driven filtering, and source-based program slicing. In addition, a possible solution that combines these techniques is proposed. To reduce false positives a standard password strength estimator `zxcvbn`[7] is used. The estimator penalizes strings with repeating characters or those that contain dictionary words while accepting strings that have a high amount of apparent randomness (entropy). The false positives that remained were auto-generated key-like strings that were used for different purposes,

[3]https://github.com/trufflesecurity/trufflehog

[4]https://github.com/auth0/repo-supervisor

[5]https://github.com/awslabs/git-secrets

[6]https://github.com/UKHomeOffice/repo-security-scanner

[7]https://github.com/dropbox/zxcvbn

such as identifiers of serialized objects. Both pattern based (Roesch, 1999) and data-flow based (Klieber et al., 2014), (Zhou et al., 2011) techniques have been applied for detecting deliberate leakage of user credentials by malicious software. Similar techniques (Viennot et al., 2014) have also been applied for detecting accidental leakage of credentials in innocuous client-side software.

In (Meli et al., 2019) the authors use regular expressions and a system of filters (entropy filter, words filter, pattern filter) to detect API keys. They write that they did not attempt to examine passwords as they can be virtually any string in any given file type, meaning they do not conform to distinct structures and making them very hard to detect with high accuracy. According to the article (Saha et al., 2020) most of existing works either target a specific type of secret or generate a large number of false positives. This makes it possible to achieve high accuracy in detecting secret keys in public repositories. However, these works are specifically aimed towards private keys (e.g., an RSA private key) and API keys, and none of these include generic passwords in their scan. Another serious limitation of the existing tools is that they generate a high number of false positives because loose regular expressions, used in the tools, match invalid entries (Meli et al., 2019). These false positives reduce the reliance on the existing tools resulting in extensive manual, time-consuming, effort to actually identify the false positives. Therefore, there is a strong need for developing generic approaches to include all types of secrets and not just private or API keys, and that significantly reduce the false positives in identification of the secrets. To tackle the issue in (Saha et al., 2020) a voting classifier (combination of logistic regression, naive bayes and SVM) is used with an extensive regular expression list. Authors test the voting classifier on a test dataset consisting of 2,180 examples and achieve a precision of 84%.

Similar approach is proposed in (Lounici et al., 2021). The paper presents an approach to detecting data leaks in open-source projects with a low false positive rate. In addition to commonly used regular expression scanners, authors propose several machine learning models (LCBOW model, Q-learning) targeting the false positives. The approach, while producing a negligible false negative rate, decreases the false positive rate to, at most, 6% of the output data. However, this approach assumes that in addition to the secret itself, its context is available. This additional requirement is not always met. In addition, this requirement significantly limits the portability of the approach to other programming languages. Also, this requirement limits the possibility of data augmentation when building datasets. Almost all public datasets with secrets that can be used for training do not contain the context of secrets. In particular, it becomes difficult to repeat the results, since our datasets do not contain contexts: only a string literal and a label. Our approach is char-based (Bag-of-Words in the case of a classifier of (Lounici et al., 2021)) and does not use a secret context.

It should be noted that, along with the described approaches, approaches based on knowledge of all the secrets can be useful in some cases. In (Ding et al., 2020) authors propose to use known production secrets as a source of ground truth for sniffing secret leaks in codebases.

## 3 PROPOSED APPROACH

Regular expressions are often used to find secrets in code. In this case, we must explicitly formulate the search rules. The effectiveness of this approach essentially depends on the ability to describe all of the secrets. For example, if the secrets are API keys in a known format (e.g., UUID), then such a solution may well provide the required quality. The quality will be low if we try to look for passwords (in the code) for which the format is not fixed. Even if it is known that the password meets certain requirements (e.g., the length and symbols used), then there will be many false positives. However, it may be inefficient to completely abandon regular expressions since they are effective and allow you to verify requirements for secrets.

The approach described in this paper consists of two stages (see Fig. 1). The first stage uses regular expressions. Having information about the format of the API keys and the requirements for the complexity of the password, regular expressions allow us to find strings that look like secrets (candidate strings). Since there are many non-secrets among such strings, additional filtering is required. Depending on the scenario, up to 82% (Lounici et al., 2021) of the strings are false positives (in our case we observed up to 70% false positives).

Thus, a second stage is required at which additional filtration takes place. In our case filtering was done manually. Despite the labor costs, secrets must be detected. Therefore, to automate the process of detecting secrets completely, a solution with a low false positives rate is needed. Currently, manual verification has been replaced by automatic — the proposed method is used that allows you to detect secrets with high quality. Since the classifier is not based on the

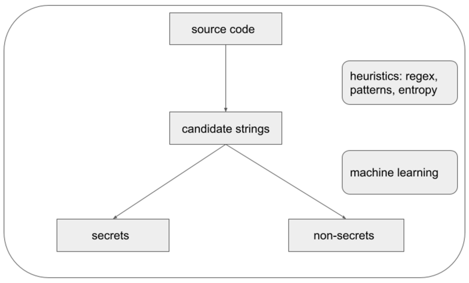

Figure 1: Overview of the proposed approach. In general, the approach consists of two stages. At the first stage, candidate strings are extracted from the code fragment. It is important to pay attention to the speed and scalability of the methods used at this stage since the codebases may be huge. Extraction can be done with regular expressions or code splitting based on whitespaces. At the second (main) stage, a decision is made as to which of the classes each candidate string belongs to. Here we can use a large class of methods because amount of candidates is small in comparison to the entire codebase. In our case, the decision is made with the use of machine learning.

heuristics used at the first stage, the classifier is of independent interest, and this work is devoted to its description. It is important to note that in this work the model does not use the context of the string being verified (tokens before and tokens after). In our opinion the use of such information can improve the quality of the model. Building a context model for detecting leaked secrets in the code is a promising direction for future work.

Since the approach uses machine learning, and machine learning, in turn, is working with data, first we will talk about the data, and then about the model (secret classifier).

## 3.1 Data

To train a machine learning model we need a dataset. We have labeled data from the target distribution. For each string from the data there is a label, whether it is a secret or not. However, these data have a number of specialities. First, data contains 7,382 samples. That is not a lot and it imposes restrictions on the choice of method. Especially considering that some of the data needs to be held out for evaluation. Also, if we evaluate a model using a small number of samples, the results become statistically unreliable. Besides, the distribution of the labeled data is significantly different since both classes (secrets and non-secrets) went through the initial regular expression filtering. Considering all this, all 7,382 samples were left for evaluation. To train the model we collected a separate dataset based on public and automatically generated data.

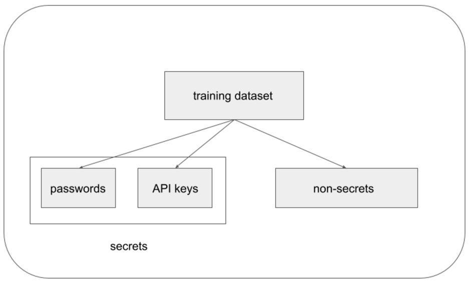

Figure 2: The training dataset consists of two classes: secrets that must be protected and cannot be stored in a codebase and non-secrets (regular code snippets). The data consists of three parts: passwords from public repositories, automatically generated API keys, and code strings from a public dataset containing code snippets. The first two parts (passwords and API keys) form the first class — *secrets*, and the third part (code strings) — the second class, *non-secrets*.

Building a new dataset has obvious advantages (you can build a large dataset), but it also has some drawbacks. A priori, it is possible that the model will show high quality on the train dataset, but low on the test one (since they are from different domains). However, in our case everything went smoothly. The model showed the ability to cross-domain transfer. Also, note that since no sensitive data is used for training the model, such a model can be safely made public.

In order to collect a large and diverse dataset, we use various sources (Fig. 2). In general, we need to collect three sets of data: passwords, API keys, and non-secrets.

### 3.1.1 Passwords

Currently, there are many different datasets available containing leaked user passwords. To build the dataset with passwords we use three sources: the `rock_you`[8] dataset and two datasets from the `SecLists`[9] repository. These datasets are filtered (too simple passwords are skipped) and deduplicated.

Authorization systems often do not allow you to specify an arbitrary sequence of characters as a password — the password must provide minimal complexity. The complexity of the password is regulated by its length, the use of numbers, special characters. Therefore, when collecting passwords for our dataset, we use filtering so that the most “simple” passwords do not get into the dataset. Note that often public datasets with passwords contain additional information (password popularity or complexity) that can be used to judge its complexity. However, we do not use

[8]https://www.tensorflow.org/datasets/catalog/rock_you
[9]https://github.com/danielmiessler/SecLists

this information due to the fact that different datasets use different approaches to determine the complexity. Our dataset did not include passwords that consisted only of numbers. In addition, a password was not included in our dataset if it consisted of only lowercase or only uppercase letters. Additionally, we used one more technical limitation: we skipped passwords longer than 128 characters (note that there were very few such long passwords, no more than ten items for the entire dataset). Filtering simple passwords is optional and not a mandatory step. Note that the exclusion of simple passwords from the training sample does not mean that they will automatically cease to be determined as passwords (in our approach, the decision is made on the basis of bigrams).

As a result of such filtering, the number of passwords decreased from 14,344,391 to 249,447 for the `rock_you` source. The million most popular passwords list (Miessler, 2021b) from the `SecList` repository reduced to 481,621 items. Before filtering `000webhost` list (Miessler, 2021a) contained 720,302 passwords, and after filtering — 718,568 passwords. After filtering we deduplicated the data since some passwords were present in several sources at the same time. For example, the size of the intersection of `rock_you` and `000webhost` list was equal to 34,642. As a result, we received a dataset containing 1,337,342 passwords.

Note that it was possible not to perform the filtering and data deduplication procedures. Then we would have received a lower quality dataset, since it contained duplicates and more simple passwords. Nevertheless, by performing filtering and deduplication, we significantly reduced the size of the dataset.

#### 3.1.2 API Keys

To obtain API keys we use the `keygen` utility, which allows you to generate keys in a given format. We found it convenient to use the appropriate Python package[10]. The following arguments were used for generation:

- `keygen -s h -l 8`
- `keygen -s h -l 16`
- `keygen -s uldp -l 8`
- `keygen -s uldp -l 16`
- `keygen -e uuid`

As a result, we received 1,337,340 keys (approximately equal to the number of passwords in our dataset).

[10]https://pypi.org/project/keygen/

#### 3.1.3 Non-Secrets

Passwords and API keys make up the class of secrets in the training dataset (see Fig. 2). In addition, we need a non-secret class. To do this we take a part of the dataset GitHub Code Snippets[11] with code snippets in different languages. This dataset contains 4,850,000 code snippets. Each snippet is a piece of code. Thus, the number of string literals that are not secrets is even greater. To ensure the class balance for the downstream binary classification task we extracted only 2,674,682 string literals from the snippets (according to the size of the secret class consisting of passwords and API keys). The code from the snippets has been split into string literals based on whitespaces. If the string literal was longer than 128 characters, then it was skipped.

Note that in this case again we have a big difference in domains. This complicates the original task, but in the case of success, a more reliable solution can be obtained. For example, we got another distribution of programming languages. In the dataset, the most popular languages are: C, JavaScript, Go, Java, and C++, while the main target language for the evaluation is C#.

Thus, the dataset for training the model contains about 5 million strings. Each string is labeled as secret or non-secret. The number of secrets is roughly equal to the number of non-secrets. Half of the secrets are passwords and the other half are API-keys.

### 3.2 Models

The aim of this research is to obtain a secret detector with a small number of false positives. In general, we follow the following plan (Fig. 3):

1. transform the string into a vector;
2. classify the vector (secret, non-secret).

This data transformation is standard for machine learning based approaches. First, we vectorize an object: the values of explicit or implicit features are calculated. Then it becomes possible to classify objects using machine learning methods. At the output we have the probability that object belongs to certain class.

The choice of features at the stage of vectorization is very important. Obviously, if the features do not contain information useful for separating classes, then it will not work well to classify objects. Here we use two approaches. First, we follow the assumption

[11]https://www.kaggle.com/datasets/simiotic/github-code-snippets

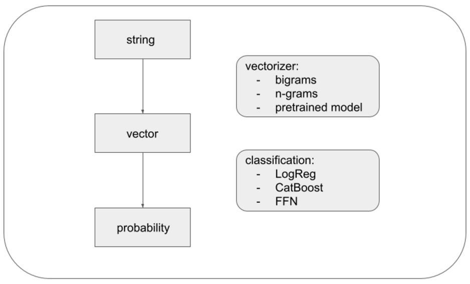

Figure 3: Data pipeline. At the input we have a candidate string, at the output — the probability that object belongs to certain class. First, the input string is vectorized. For example, using bigrams each string is transformed into a vector from $\mathbb{R}^{1024}$ via count vectorizer with the dictionary of size 1024. Next, the vector is sent to the classifier (CatBoost, LogReg or feedforward neural network) to determine if it is secret or not using confidence threshold.

that the code (and hence its fragments, string literals) has much in common with the natural language (Ray et al., 2016). This means that the methods that have proven themselves in NLP are of interest. Secondly, we don't use hand-crafted features, but use bigrams (Manning et al., 2008) and pretrained code models. A bigram is a sequence of two adjacent elements from a string of tokens, which are typically letters, syllables, or words. In our case bigram is two consecutive characters. As usual with this approach, a vocabulary of bigrams is built on the basis of the training dataset.

We assume that the entropy of non-secrets is lower than that of secrets (some approaches for detecting secrets are based on this (Meli et al., 2019)). Thereofore we use only non-secrets (approximately 2.5 million) to build a vocabulary of bigrams. Using secret we will get a huge vocabulary in which any arbitrary bigram can be found with approximately the same probability. If we use only strings from non-secret class, which are mostly code fragments, then the distribution of bigrams is very different from uniform. Moreover, due to the large number of bigrams, it is not advisable to use all of them as features: firstly, the vectors will have large dimension, and, secondly, they will be very sparse. If the bigram is uncommon, then the corresponding feature will almost always be zero. (For this reason, we did not use bigrams from secret class. If you use some of them, then in most cases they will not be found among non-secrets or other secrets. If you use all, then there will be too many of them.) Thus, to calculate the features, we used only a limited number of the most frequent bigrams. In our case the dimension of the vectors was 1024. In the general case, the choice of the number of features depends on the computing resources and the ability of downstream classifiers to work with sparse vectors.

In this bigram-based approach, the coordinates of the vectors (features) express the statistical properties of *n*-grams. In other words, the resulting vectors contain the syntactic features of the strings. This means that, e.g., the vectors corresponding to the strings "password" and "secret" are very different. Despite the close semantics, regarding bigrams these strings are completely diverse. This is not a problem when the training dataset contains all possible variations, but can lead to a significant reduction in quality otherwise. One way to mitigate this issue is to use semantic features. Hence we have another approach, that uses pretrained model as a universal feature extractor for getting semantic embeddings. The proposed approach uses GraphCodeBERT (Guo et al., 2021) representation for a string candidate. It is a pretrained model for a programming language that considers the inherent structure of code. GraphCodeBERT is semantic contextual multilingual representation based on the multi-layer bidirectional Transformer (Vaswani et al., 2017). The model follows BERT (Devlin et al., 2019) and RoBERTa (Liu et al., 2019). The resulting 768-dimensional vector represents a string.

After the string is transformed into a vector (of length 1024 or 768, depending on the selected representation) the classification is started. As classification algorithms we used two well-known standard algorithms: LogReg(McCullagh and Nelder, 1989) and CatBoost(Dorogush et al., 2018).

## 4 RESULTS

At the moment, there are a large number of different methods for solving the problem of binary classification. In this study, we used two: logistic regression (one of the simples algorithms) and CatBoost (an algorithm that has proven itself well for many classification tasks).

LogReg (logistic regression) is a statistical model that models the probability of one event (out of two alternatives) taking place by having the log-odds (the logarithm of the odds) for the event be a linear combination of one or more independent variables ("predictors"). Here we uses `sklearn`[12] implementation of the algorithm.

CatBoost is an algorithm for gradient boosting on decision trees. CatBoost builds symmetric (balanced) trees, unlike XGBoost and LightGBM. In every step, leaves from the previous tree are split using the same condition. The feature-split pair that accounts for the lowest loss is selected and used for all the level's

[12]https://scikit-learn.org

Table 1: GPU usage.

| | CPU | GPU |
|---|---|---|
| representation: bigrams | ✓ | |
| representation: pretrained | | ✓ |
| classification: LogReg | ✓ | |
| classification: CatBoost | | ✓ |

nodes. This balanced tree architecture aids in efficient CPU implementation, decreases prediction time, makes swift model appliers, and controls overfitting as the structure serves as regularization.

Depending on string literal representation and which algorithm is used for the classification task, we have the following approaches.

1. bigrams+LogReg (bigrams and logistic regression),
2. bigrams+CatBoost (bigrams and CatBoost),
3. pretrained+CatBoost (GraphCodeBERT and CatBoost).

We did not explore the pretrained+LogReg approach because by this point we already had the results of the pretrained+CatBoost approach. Based on the results of pretrained+CatBoost, it became clear that you should not count on fewer errors in pretrained+LogReg than it was in bigrams+CatBoost.

### 4.1 Training

For training the dataset described above was used, consisting of public and automatically generated data. The GPU was used to speed up the calculations during model training (see Table 1).

As follows from Table 1 we used a GPU to train the CatBoost classifier. Due to the large size of the dataset and the large dimensionality of the vectors, the algorithm builds a large tree. This requires computational resources and a large number of iterations.

We used the implementation of the CatBoost[13] algorithm for the Python language. Iteration limit is 50,000; coefficient at the L2 regularization term of the cost function is 2; loss function is `logloss`; learning rate is 0.05; bagging temperature is 1; random strength is 1; leaf estimation method is `Newton`; early stopping rounds parameter is 100. Model training took less than an hour on a GeForce GTX 1080 Ti graphic card.

To build semantic embeddings we used a pretrained GraphCodeBERT[14] model. This model was used for vectorization without additional fine-tuning (see (Abnar et al., 2021)).

[13]https://catboost.ai

[14]https://huggingface.co/microsoft/graphcodebert-base

Table 2: The results of the three experiments. For each experiment, FPR, FNR, and F1 score are given.

| | FPR, % | FNR, % | F1 |
|---|---|---|---|
| bigrams+LogReg | 5.92 | 16.67 | 0.78 |
| **bigrams+CatBoost** | **2.4** | **1.87** | **0.96** |
| pretrained+CatBoost | 7.94 | 7.52 | 0.79 |

Table 3: The confusion matrix for bigrams+CatBoost model.

| | predicted secret | predicted non-secret |
|---|---|---|
| secret | 1946 | 37 |
| non-secret | 130 | 5269 |

### 4.2 Evaluation

Models were evaluated on a separate private dataset. This dataset is the most consistent with the target domain. It contains 7,382 samples. All these samples correspond to the real situation and have been manually labeled into two classes: secrets and non-secrets. As final metrics, we used the FPR (false positive rate) and FNR (false negative rate) since such metrics most accurately reflect the functionality of the models.

In Table 2 we present the test results. It is expected that the model based on LogReg showed the worst result. The best result was shown by a model based on bigrams and CatBoost. The confusion matrix for this experiment can be found in Table 3.

The model based on the pretrained model showed an average result, lower than expected. However, we believe that this result can be better if string context is used, that is, characters before and characters after are taken into account. In such case, contextual embeddings may contain information useful for classification.

## 5 CONCLUSION

In this research we investigate the problem of detecting leaked secrets in the code. As a result, we propose an approach to leaked secret detection in source code based on machine learning using bigrams. This approach significantly reduces the number of false positives. It showed a false positive rate of 2.4% and false negative rate of 1.9%. The model was trained on public and automatically generated data, and the evaluation took place on target domain, which allows us to conclude that the resulting solution is stable and reliable.

We believe that a promising direction for future research is the use of pretrained code models. Apparently, the application of such models to build semantic embeddings using *the context of the secret* (previous and subsequent tokens) will give better quality, although it will require more computing resources. The pretrained code models contain the programming language model (MLM, masked-language modeling task). This makes it possible to understand, based on the context, what is happening in a given fragment of code. Whether it's sensitive information or not. A similar situation occurs in the code completion task or in the variable name prediction task, where pretrained code models are great (Guo et al., 2022).

## REFERENCES

Abnar, S., Dehghani, M., Neyshabur, B., and Sedghi, H. (2021). Exploring the Limits of Large Scale Pretraining. *ArXiv*, abs/2110.02095.

Collins, K. (2016). Developers keep leaving secret keys to corporate data out in the open for anyone to take. https://qz.com/674520.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*, pages 4171–4186, Minneapolis, Minnesota.

Ding, Z. Y., Khakshoor, B., Paglierani, J., and Rajpal, M. (2020). Sniffing for Codebase Secret Leaks with Known Production Secrets in Industry. *CoRR*, abs/2008.05997.

Dorogush, A. V., Ershov, V., and Gulin, A. (2018). Catboost: gradient boosting with categorical features support. *ArXiv*, abs/1810.11363.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S. K., Clement, C. B., Drain, D., Sundaresan, N., Yin, J., Jiang, D., and Zhou, M. (2021). GraphCodeBERT: Pre-training code representations with data flow. In *ICLR 2021*.

Guo, D., Svyatkovskiy, A., Yin, J., Duan, N., Brockschmidt, M., and Allamanis, M. (2022). Learning to Complete Code with Sketches. *ArXiv*, abs/2106.10158.

Kall, S. and Trabelsi, S. (2021). An Asynchronous Federated Learning Approach for a Security Source Code Scanner. In *ICISSP*, pages 572–579.

Klieber, W., Flynn, L., Bhosale, A., Jia, L., and Bauer, L. (2014). Android taint flow analysis for app sets. In *SOAP*.

Knight, S. (2016). 10,000 AWS secret access keys carelessly left in code uploaded to GitHub. https://www.techspot.com/news.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. (2019). RoBERTa: A robustly optimized bert pretraining approach. *arXiv e-prints*, page arXiv:1907.11692.

Lounici, S., Rosa, M., Negri, C. M., Trabelsi, S., and Önen, M. (2021). Optimizing Leak Detection in Open-source Platforms with Machine Learning Techniques. In *ICISSP*, pages 145–159.

Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press, USA.

Marlow, P. (2019). Finding Secrets in Source Code the DevOps Way. In *SANS Institute*.

McCullagh, P. and Nelder, J. A. (1989). Generalized linear model. In *CRC press*, volume 37.

Meli, M., McNiece, M. R., and Reaves, B. (2019). How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories. *Proceedings 2019 Network and Distributed System Security Symposium*.

Miessler, D. (2021a). 000webhost. https://github.com/danielmiessler/SecLists/blob/master/Passwords/Leaked-Databases/000webhost.txt.

Miessler, D. (2021b). 10 million password list top 1000000. https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/10-million-password-list-top-1000000.txt.

Rahman, M. R., Imtiaz, N., Storey, M.-A., and Williams, L. (2022). Why secret detection tools are not enough: It's not just about false positives — An industrial case study. *Empirical Software Engineering*, 27(3):1–29.

Ray, B., Hellendoorn, V., Godhane, S., Tu, Z., Bacchelli, A., and Devanbu, P. (2016). On the "naturalness" of buggy code. ICSE '16, pages 428–439.

Roesch, M. (1999). Snort — Lightweight Intrusion Detection for Networks. In *LISA*.

Saha, A., Denning, T., Srikumar, V., and Kasera, S. K. (2020). Secrets in Source Code: Reducing False Positives using Machine Learning. In *2020 International Conference on COMmunication Systems NETworkS (COMSNETS)*, pages 168–175.

Sinha, V. S., Saha, D., Dhoolia, P., Padhye, R., and Mani, S. (2015). Detecting and Mitigating Secret-Key Leaks in Source Code Repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 396–400. IEEE Press.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. NIPS'17, pages 6000–6010.

Viennot, N., Garcia, E., and Nieh, J. (2014). A measurement study of Google Play. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems, ser. SIGMETRICS*, pages 221–233.

Zhou, Y., Zhang, X., Jiang, X., and Freeh, V. W. (2011). Taming informationstealing smartphone applications (on android). In *TRUST*.