Secrets in Source Code: Reducing False Positives Using Machine Learning

Aakanksha Saha
Software Engineer
Microsoft
Redmond, USA
aasaha@microsoft.com

Tamara Denning
School of Computing
University of Utah
Salt Lake City, USA
tdenning@cs.utah.edu

Vivek Srikumar
School of Computing
University of Utah
Salt Lake City, USA
svivek@cs.utah.edu

Sneha Kumar Kasera
School of Computing
University of Utah
Salt Lake City, Utah
kasera@cs.utah.edu

***Abstract*—Private and public git repositories often contain unintentional sensitive information in the source code. Many tools have been developed to scan repositories looking for potential secrets and credentials committed in the code base, inadvertently or intentionally, for taking corrective action once these secrets and credentials are found. However, most of these existing works either target a specific type of secret or generate a large number of false positives. Our research aims to create a generalized framework to detect all kinds of secrets – which includes API keys, asymmetric private keys, client secrets, generic passwords – using an extensive regular expression list. We then apply machine learning models to intelligently distinguish between a real secret from a false positive. The combination of regular expression based approach and machine learning allows for the identification of different types of secrets, specifically generic passwords which are ignored by existing works, and subsequent reduction of possible false positives. We also evaluate our machine learning model using a precision-recall curve that can be used by an operator to find the optimal trade-off between the number of false positives and false negatives depending on their specific application. Using a Voting Classifier (combination of Logistic Regression, Naïve Bayes and SVM) we are able to reduce the number of false positives considerably.**

***Index Terms*—Automated software tool, hard-coded secrets, source code, security**

I. Introduction

Secrets, that include sensitive information such as credentials, private keys and passwords, have continually grown in number; as more services are being deployed that require authentication, more secrets are being generated. Worrisomely, developers are taking the insecure shortcut of hard-coding secrets in source code and configuration files. Moreover, once a secret is checked into a source control system, such as GitHub or BitBucket, it is nearly impossible to delete it entirely [1]. Unauthorized developers and malicious users can steal this sensitive information and cause severe financial and personal damage. In 2014, an Uber employee accidentally uploaded his credentials to GitHub, which resulted in the Uber database hack of 50,000 Uber drivers in 2014 [2]. Similarly, Amazon found 10,000 AWS keys accidentally left in the source code by Amazon developers and uploaded to GitHub [3]. These secret access keys were misused by malicious impersonators to run intensive compute jobs, which were billed to the victims. Thus, publishing secrets on public or private repositories can have severe repercussions, and should be avoided. However, this idealized goal has been hard to achieve. Instead, many tools have been developed to scan repositories looking for potential secrets and credentials committed in the code base, inadvertently or intentionally, for taking corrective action once these secrets and credentials are found.

Most of these tools [4]–[7] use regular expression search, entropy checks or a combination of both to identify potential secrets. There are a few existing works [8], [9] that have achieved 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 included 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 [8], [10]. 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.

We develop a generalized framework using regular expressions to detect different types of secret, including generic passwords and then apply machine learning to reduce false positives. Specifically, we first develop a generic tool to detect different types of secrets such as API keys, AWS keys, OAuth client secrets, and generic passwords. Next, we design 24 relevant features for potential secrets extracted from 300 different repositories ($\approx$ 28 million files), and create a labeled dataset containing 5000 examples. Then, we use a Voting Classifier, an ensemble of Logistic Regression, Naïve Bayes and SVM, to significantly reduce false positives while keeping the secret detection rate high. We test our Voting Classifier on a test dataset consisting of 2180 examples and achieve a precision of 84%. We also evaluate our model using a precision-recall curve that allows a user to tune to the optimal balance between the number of false positives and false negatives depending on their application needs. To the best of our knowledge, no existing work combines regular expressions with machine learning to identify different types of secrets (API keys, RSA private keys, generic passwords, connection strings) found on public forums and subsequently, reduces false positives.

978-1-7281-3187-0/20/$31.00 ©2020 IEEE

II. Problem Setting

In this section, we formally define secrets, along with their matching regular expressions or *regex*. We also discuss the problem of false positives in detecting secrets using regular expressions. Finally, we tabulate some of the popular open-source tools, their search criteria in finding confidential information and their adopted methods for reducing false positives.

A. Secrets and their matching regex

Secrets, by definition, should be known only by few. In this work, secrets refer to all the credentials, private keys, configuration files or passwords that a user needs to deploy software across environments. The following is a list of common types of secrets found in the source code and the configuration files of the git repository.

API Keys: Application Programming Interface keys provide project authorization by identifying the app or the project that's making a call to the API [11]. For example, AWS API keys are the access keys used to make programmatic calls to Amazon web services. A possible matching regular expression we use to identify these is as follows: 'AKIA[0-9][A-Z]{16}'.

OAuth Client Secret and Client ID: These secrets are generated when the developer registers to an application. The client_id is a public identifier for apps. For example: GitHub Client ID would look similar to 6779ef20e75817b79602. The client_secret is a secret known only to the application and the authorization server. For each registered application, one needs to store the public client_id and the private client_secret also known as OAuth secret [12].

Access token: An access token is an opaque string that identifies a user, an app, or a Page and can be used by the app to make API calls. These tokens provide temporary, secure access to APIs [13].

SSH/SSL/RSA Private Keys: These private keys are generally used in case of SSL certificates for authentication to a remote server. The generic regular expression that matches these private keys is: '—–BEGIN RSA PRIVATE KEY—–*.'

Generic passwords: These are the common username password pairs that grant access to applications and databases. One possible regular expression match we use for passwords is as follows: '.*[P][A][S][S][W][O][R][D].*[:=](.*)'.

Entropy-based search Apart from regular expression-based secret detection tools, [4] and [5] employ an entropy-based search. In particular, Shannon entropy is a good estimator of randomness in a string [14], [8], [15]. Machine generated secrets having high randomness, such as API keys and private keys, have high entropy while user-created generic passwords tend to have low entropy.

False positives The naive method of regular expressions and the entropy based approach result in high numbers of false positives as strings detected by these methods are not guaranteed to be a secret [8]. **We scan 300 GitHub source code repositories and manually analyze the output to observe the following common types of false positives** (only three of these have been observed in [10]):

1) File path or an environment variable (Ex: secret = $HOME/path/)
2) Function calls (Ex: password =: getPassword())
3) Variables (Ex: password = ui.password)
4) CSS selectors (button[value="test_password"])
5) Sample/Test files, containing test secrets
6) Example keys in Readme (password = "test_12345")
7) Placeholders of Passwords (//proxy_user:proxy_password@)
8) Example keys in commented code section (# apikey:'xxxxxxxxxx')
9) Variable initialization with word 'password' in it (rpcpassword = retypePassword)

It is crucial for a secret detector tool to remove false positives as much as possible and report only potential secrets with high accuracy. The list of false positives can possibly be identified as such by a human observer. However, in some cases, it is very hard to know the ground truth, whether the secrets are really meaningful secrets, without actually trying them. We discuss concerns related to ethics and ground truth of a secret in Section VIII.

Numerous open source tools have been created to detect and identify sensitive information on GitHub. In Table I, we assess some of the popular open source tools and existing works in finding confidential information on public forums and their adopted methods of reducing false positives. The existing works mentioned in Table I focus only on secrets that could be discovered with a high probability of validity and sensitivity. These works do not attempt to examine generic passwords as they do not have a distinct structure and it is difficult to detect passwords with high accuracy. Moreover, the open source tools mentioned in Table I are prone to a large number of false positives or require customized manual patterns to skip false positives [6], [5].

Another proprietary tool named Cred Scan, created by Microsoft, uses a regular expression-based approach to scan for sensitive contents within the source code files [16]. GitRob [17] popular among organizations scans for potentially sensitive files pushed to public repositories. Instead of scanning for hard-coded secrets in source code this tool looks for sensitive file extensions, including *.pem*, *.pkcs12*, *.cscfg*.

In contrast to these related works, we aim to collect all kinds of secrets starting from API keys, private keys to generic passwords where entropy, dictionary words or any pattern-based filtering will not prove effective and therefore, we propose to use machine learning to intelligently distinguish a real secret from a false positive. We believe our methodology, which is a combination of regular expression based approach and machine learning, significantly enhances the state-of-the-art. We acknowledge the use of filters such as entropy, words and file types in prior work [4], [5], [8], [9] as features in our labeled dataset for using machine learning.

III. Approach

In this section, we describe the components of our general framework for reducing false positives while finding secrets

TABLE I
ANALYSIS OF RELATED TOOLS

Tools	Features	Regular expressions	Use of Entropy	Mechanism for Excluding false positives	Results
Truffle Hog [4]	Searches through commit history and git branches	Predefined customisable list	High entropy strings greater than 20 characters	None	NA
Repo Supervisor [5]	Detects secrets in pull requests	Measures high entropy strings	Shannon entropy higher than 4	Filters for URLS, file paths, CSS selectors, email addresses, multiple words with spaces, object keys, strings with certain prefixes and strings having dictionary words	NA
Git Secrets [6]	Scans for sensitive information while committing to git	User supplied regex	55	User supplied patterns and regex	NA
Meli et al. [8]	Scans files on GitHub	Regex for private key files and 11 distinctive API key formats	Reject strings with entropy more than 3 standard deviations from the mean of all candidate strings containing valid secrets	Dictionary words filter and Pattern filter	99.29% accuracy on GitHub search
Sinha et al. [9]	Sample set of 84 projects on GitHub	Regex for API keys (Facebook and Amazon)	Entropy filter used by password strength estimator	Program slicing and User ID + secret key proximity	Precision 91% and Recall 100% on Amazon API key leaks

in source code. The first step is to collect URLs of the git repositories and retrieve the candidate files that we use to clone and scan for hard-coded secrets The scanning of secrets (API keys, private keys, RSA keys, passwords) is done using basic pattern and keyword matching and the output is generated in a JSON format. The raw JSON data is then converted to a clean labeled featurized dataset. This dataset is divided into training, test, and development sets. Lastly, different machine learning algorithms are applied and trained on the training dataset, final evaluation is done on the test dataset, and the development dataset is used for error analysis.

A. URL Collection

To test and compare the performance of different machine learning algorithms in identifying a false positive reported by a secret detector tool, the initial step is to build a dataset containing thousands of examples of secrets and false positives. For building this dataset, we first gather the candidate files that are likely to contain secrets. We use the git Rest API [18] to collect a list of URLs with candidate files. The Rest API allows searching for specific terms in the source code. Among the millions of repositories, we are only interested in finding those that have hard-coded secrets/passwords. We write a Python script to perform an API query with search terms such as 'RSA PRIVATE KEY,' 'AWS Credential,' 'password,' 'secret,' 'API key' to retrieve a list of git URLs. Example query: *"https://api.github.com/search/code?q="Search terms"&page=5"*. Our script returns a list of around 300 git URLs for further cloning and scanning.

There are a few restrictions on how searches are performed on the GitAPI: Only the default branch is considered, and only files smaller than 384 KB are searchable. These restrictions are irrelevant to us as we want to collect different types of secrets hard-coded in the source code repository and are not interested in any particular repository or file sizes.

B. Cloning and Scanning

Once the URLs are collected and saved in a text file, the next step is to get data for our training examples by scanning the source code files of the git repository. We first write a multi-threaded Python program to clone the git repositories. This step is required because Git does not allow complex regular expression search on its source code remotely. Our program takes the URL list as an argument, sorts the URLs uniquely and uses GitPython [19] module to download the files in all the branches of the repository to a local system.

After cloning the repositories, another multi-threaded python program, that we write, scans the source code searching for potential secrets. Each line in the file is matched against a list of regular expressions. We save the secret value along with file name, repository name, and the line on which secret is contained in a JSON file. On scanning around 300 repositories ($approx$ million files), we find that 250 of these repositories generate non-empty JSON files containing both real secrets and false positives. Table II shows the list of secrets and their matching regular expressions that are unique to our research to detect generic passwords. Regular expressions in Table II along with the list of regular expressions in [4] are used for our scanning process giving us a total of 32 regular expressions.

C. Feature Designing and Labeling

To use machine learning algorithms, we need a labeled dataset which has inputs that are defined by features to guide the decision-making process. The list of potential secrets obtained in the previous step is further manually analyzed to

TABLE II
LIST OF REGULAR EXPRESSIONS FOR GENERIC SECRETS

Secret	Target Regular Expression
Generic AppSecret	[a | A][p |P][p |P][s |S][e |E][c |C][r |R][e |E][t |T].*[' |"]([0-9a-zA-Z]{32,45})[' |"]
Generic Password	.*[p |P][a |A][s |S][s |S][w |W][o |O][r |R][d |D].*[:=](.*)
Password in URL	//[\\s:]+:[\\s:]+@
Other passwords	(Password |pass |passwd |session_password |login_password |password) \s*[:=](\s*[^\r\n]+)
Tokens	(API_tokens |api_tokens |tokens |Tokens)\s*[:=](\s*[^\r \n][a-f0-9]{16})
Generic Access Token	[a | A][c |C][c |C][s |S][e |E][s |S][s |S][t |T][o |O][k |K][e |E][n |N].*[:=](.*)
Other secrets	(client_secret |access_secret |customer_secret |app_secret)\s*[:=](\s*[^\r\n]+)

assign features. **To define the attributes of the potential secrets returned in the scanning phase, we perform a manual analysis of the source code to understand the context and the properties of the extracted strings.**

Our complete list of features is given in Appendix A.1. In this list, the first 12 features are unique and new to this paper. These features identify if the lines with potential secret extracted are some programming language construct (function calls, variables, comments) or some test/example secrets dumped in the documentation/README files. Other features such as file extensions and entropy are similar to those in previous works. We convert the entropy value (integer) into binary features by dividing it into separate bins (e.g., Is the entropy between 0-1 and so on).

For all the possible secrets the features as mentioned above are assigned "Yes"(1) or "No"(0) values and a final CSV file is generated containing 5000 examples. Once the dataset (CSV file with features) is created, it is labeled manually. If the string extracted is a real secret/password, the string is labeled as 'S' (secret); on the other hand, false positives are labeled as 'N' (Not a secret). Our labeled dataset consisting of 5000 examples with 24 features is ready to be released and we do not include the secret value, the file name and the repository name in our dataset as explained under ethical conduct in Section VIII.

D. Algorithms

Our goal is to ascertain whether a *potential* secret is truly sensitive, or merely a false positive that only accidentally matches a high recall regular expression. For a potential secret s, we can frame this decision as a predicate $IsSensitive(s)$ that takes the value `true` for all s that are sensitive. With such a predicate, we may be able to filter automatically harvested lists of potential secrets. With restricted domains (e.g., only RSA private keys), we may be able to manually define the predicate via simple pattern matching mechanisms. However, as mentioned earlier, this approach does not generalize: for example, we can not enumerate every possible pattern that defines $IsSensitive$ for a potential secret in the source code. In this work, we ask: *Instead of manual enumeration to define the decision rule $IsSensitive$, can we discover it from data?*

Specifically, we focus on the supervised setting using the labeled dataset of potential secrets that are annotated as being false positives or not. Each potential secret is converted to a list of features using as described in Section III-C, which we will refer to as the attribute vector $\phi(s)$ for a potential secret s. At a high level, our learning goal is to use a featurized dataset D consisting of vectors of the form $\phi(s)$ to discover a decision rule $IsSensitive(s)$. Note that different learning algorithms can use the same dataset D to produce different decision rules (i.e., classifiers). In this work, we perform a comparative study of six different learning algorithms and a meta-algorithm that integrates the predictions of other classifiers. We now provide the overview of all the algorithms we use.

Decision Tree: Decision Tree [20] is a classification technique that has a tree-like structure, where each internal node denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node (terminal node) holds a class label. The idea of a decision tree is to divide the dataset into smaller datasets based on the descriptive features until one reaches a small enough set that only contains data points that fall under one label.

Random Forests: Random Forests [21] is another supervised learning algorithm. It builds multiple decision trees and merges them to get a more accurate and stable prediction.

Logistic Regression: Logistic Regression [22] is a statistical machine learning algorithm that classifies the data by considering outcome variables on extreme ends and tries to make a logarithmic line that distinguishes between them.

Naïve Bayes: The Naïve Bayes algorithm [23] makes a prediction by calculating probabilities of the instance belonging to each class and selects the class value with the highest probability.

Support Vector Machines: A Support Vector Machine (SVM) [24] is a discriminative classifier formally defined by a separating hyperplane. In other words, given labeled training data (supervised learning), the algorithm outputs an optimal hyperplane which categorizes new examples. In two dimensional space this hyperplane is a line dividing a plane in two parts where in each class lay in either side.

K-Nearest Neighbor: KNN algorithm [25] is another classification algorithm that is based on feature similarity. While predicting a label we select the 'k' entries in our database which are closest to the new sample and conduct a majority vote among the 'k' entries to decide the classification label for the new entry.

Voting Classifier: Voting Classifier [26] is a meta-classifier for combining similar or conceptually different machine learning classifiers for classification. In this work, the Soft-Voting Classifier is initialized using three exemplary classifiers Lo-

gistic Regression, Naïve Bayes, and SVM with equal weights of [1,1,1].

IV. Implementation

To test the performance of different machine learning algorithms, we use the libraries from *scikit-learn* [24] on our labeled dataset, one algorithm at a time. We perform our cross validation in Python using the Spyder development environment, and we use Pandas [27] to read and convert our labeled dataset to dataframes for running our experiments.

A. Experimentation

To apply the learning framework, we first perform series of tests to investigate the factors that affect the response (prediction of a label) of the experiments. These factors not only include the choice of the learning algorithms, but also the input features and the parameters of the algorithms. The goal here is to find the optimal configuration of factors that give the best performance [28].

For our experiments, we divide our dataset containing 5000 examples with 24 features into three parts: the training, test and development sets. The training set, with 2465 (2100 false positives and 365 true positives) samples, is used for cross-validation to select the best algorithms and the hyperparameters, and also to train the classifiers using the best settings. The test set, containing 2180 examples, is used for the final evaluation reported in Section V. We separate 355 examples as the development set for manual error analysis which we explain in detail in Section VII. We ensure that the training/development/test splits have approximately the same percentage of each target class as the full set of examples using stratified sampling. This is different from random sampling because in our case the label 'S'(1) is in minority class so stratified sampling ensures equal distribution of 'S'(1) in the training and test dataset.

B. Cross validation on training set

Each learning algorithm is associated with a collection of hyperparameters, that must be chosen appropriately to ensure good predictive performance. To obtain an unbiased estimate of the performance for each hyperparameter setting, and thereby find the best hyperparameters for each algorithm, we use K-fold cross-validation. K-fold cross-validation divides a dataset randomly into K equal-sized parts. By holding out one of these K parts and combining the remaining $K-1$ parts, we get a training-validation pair. For a particular choice of hyperparameters, we can train a model on the training subset of this pair and evaluate it using the validation subset. Repeating this process by holding out each of the K parts, and averaging the performance across the different validation subsets, we get an estimate of how good the choice of hyperparameters is for a particular full training set.

In our work, we have a binary classification problem where the we seek to obtain the $IsSensitive$ predicate described earlier. In particular, the learned classifier predicts a simple yes ('secret') or no ('false positive') response. We employed 4-fold cross validation for hyperparameter selection. Specifically, we divide the 2465 training examples into 4-equal sized sets, and a stratified split is performed on each set. We measured the precision, recall, and F_1 and F_β scores averaged over k-trials for different algorithms and hyperparameter combination. These metrics are described in detail in Section V. Table III shows the best performance of each classifier in terms of precision, recall, F_1 score, and F_β score on the training dataset where $\beta = 2$. F_2 ($\beta = 2$) score weighs recall higher than precision (by placing more emphasis on false negatives). In our work, the cost of misclassifying one secret is very high, and therefore, to ensure that we identify all the valid secrets we give more importance to recall by setting $\beta = 2$.

We achieve the optimal performance with a Voting Classifier that integrates the predictions of Logistic Regression (lr), Naïve Bayes(nb) and SVM using a weight of [1,1,1], respectively.

V. Results

In this section, we first describe the evaluation metrics that we use to measure the performance of different classification models in reducing the number of false positives reported by a secret detector tool. Next, we tabulate the initial results we obtain by running the trained Voting Classifier on the test dataset. In our dataset consisting of 2465 training examples, we observe that only 365 instances ($\approx 15\%$) are real secrets. Therefore, in this scenario, accuracy (fraction of predictions our model gets right) is not the best performance measure. To fully evaluate the effectiveness of our model in predicting a secret as a false positive, we measure precision, recall and F_1 score which are described below in detail.

A. Performance Metrics

1) Precision: Precision tells us how precise/accurate a model is; out of those predicted by a model as positive, how many of them are actual positives [29]. Thus,

$$Precision = \frac{TruePositive(tp)}{tp + FalsePositive(fp)}$$

2) Recall: Recall is the fraction of relevant instances that have been retrieved over the total amount of relevant instances. While recall expresses the ability to find all relevant instances in a dataset, precision expresses the proportion of the data points a model says was relevant and were relevant [29].

$$Recall = \frac{TruePositive(tp)}{tp + FalseNegative(fn)}$$

3) F_1 score: The F_1 score is the harmonic mean of precision and recall. F_1 scores are usually lower than accuracy measures as they embed precision and recall into their computation. The F_1 score is needed to seek a balance between precision and recall if there is an uneven distribution of labels [29].

$$F_1 = \frac{2 * precision * recall}{precision + recall}$$

TABLE III
CROSS-VALIDATION RESULTS ON TRAINING SET HAVING 2465 EXAMPLES WITH 365 REAL SECRETS

Classifier	Average Precision	Average Recall	Average F_1 score	Average F_β score ($\beta = 2$)
Decision Tree	0.78	0.75	0.76	0.76
Random Forests	0.78	0.73	0.75	0.74
Nearest Neighbor	0.78	0.78	0.78	0.78
Logistic Regression	0.81	0.75	0.77	0.76
SVM	0.79	0.73	0.75	0.74
Naïve Bayes	0.33	0.94	0.49	0.68
Voting Classifier (lr, nb, svm)	0.739	0.80	0.76	0.785

TABLE IV
CLASSIFICATION REPORT [30] ON RUNNING THE VOTING CLASSIFIER ON THE TEST SET.

Label	Precision	Recall	F_1 score	# support/examples
0 (N)	0.98	0.98	0.98	1894
1 (S)	0.84	0.89	0.87	286
avg/total	0.97	0.96	0.96	2180

4) F_β score: The F_1 score is symmetric with respect to precision and recall. We may assign recall to be β times as important as precision (for some β). This gives us a generalization of the F_1 score, where we can control the relative importance of precision and recall based on our application [24].

$$F_\beta = \frac{(1+\beta) * precision * recall}{\beta * precision + recall}$$

In our case, we assign $\beta = 2$ thus, prioritizing recall over precision.

5) Confusion matrix: It is a tabular representation for binary classification problem having four different outcomes: true positive, false positive, true negative, and false negative [24], [29]. It is useful for quickly calculating the number of mislabeled data points. The two quantities of our interest are the number of false positives, and the number of false negatives. We wish to see both of these quantities close to 0.

B. Results from test dataset

Table IV shows the result of running the Voting Classifier on our test dataset consisting of 2180 examples, of which 286 belongs to the class of real secrets (S), and the rest are false positives. It gives the measure of precision, recall and F_1 score for each target class. Label 0 represents the false positive class and Label 1 represents the real secret class. The last row is the weighted average of all the metrics, where the support values are the number of examples belonging to each label.

A Soft-Voting Classifier with weights of $[1, 1, 1]$ for Logistic Regression, Naïve Bayes and SVM results in an F_β score of 88.1% and an F_1 score of 86.7% (Label 1). Table V tabulates the number of misclassified instances in each target class. Our classifier misclassifies 47 out of 1894 false positives as secrets and 31 out of 286 true positives as false positive.

TABLE V
NUMBER OF MISCLASSIFIED LABELS REPORTED BY VOTING CLASSIFIER

Classifier	Misclassified instances of N (1894)	Misclassified instances of S (286)
Soft Voting Classifier - Logistic Regression, Naïve Bayes, SVM with weights [1,1,1]	47	31

TABLE VI
CLASSIFICATION REPORT [30] ON RUNNING THE VOTING CLASSIFIER ON A DATASET CONTAINING 361 POTENTIAL SECRETS (EXCLUDING GENERIC PASSWORDS)

Label	Precision	Recall	F_1 score	# support/examples
0 (N)	100	75	86	126
1 (S)	88	100	94	235
avg/total	92	91	91	361

C. Comparison with existing work

Table IV shows the result of applying a learned model on a dataset consisting both API/private keys and generic passwords. We keep our model simple from a computational complexity perspective and from the results we can observe that machine learning can help in drastically reducing false positives, while detecting different kinds of secrets. Given that existing works focus on specific type of secrets, we divide our dataset based on the type of secrets. Table VI shows the classification report [30] of running the Soft-Voting Classifier on a dataset containing 361 examples of only API keys, asymmetric private keys, and client secrets. These results are similar and comparable to other existing works [8] and [9]. Table VII shows a detailed comparison between existing works and our machine learning model in terms of precision, recall and the type of secrets detected. We find that a combination of the regular expression approach and machine learning classifiers (Logistic Regression, Naïve Bayes and SVM) can achieve a precision of 84% and a recall of 89%, while detecting different kinds of secrets.

VI. EVALUATION OF OUR CLASSIFICATION MODEL

In this section, we evaluate our classification model using a *precision-recall* curve that summarizes the trade-off between precision and recall for a predictive model using different probability thresholds [24]. The precision-recall plot uses recall on the x-axis and precision on the y-axis, and the

TABLE VII
COMPARISON WITH RELATED WORK

Technique	Secret types detected	Precision	Recall	F_1 score
Voting Classifier	API keys, client secrets, private keys	88	100	94
Voting Classifier	API keys, client secrets, private keys, generic passwords	84	89	87
Sinha et al. [9](Regular expression + Entropy)	Amazon AWS API keys	91	100	NA
Sinha et al. [9](Regular expression + Entropy)	Facebook API keys	73	100	NA
Meli et al. [8] (Regular expression + Entropy, patterns, dictionary words filter)	API keys, client secrets, private keys	99.29	NA	NA

curve is created by connecting all precision-recall points of a classifier at different thresholds.

As the recall increases, the precision decreases. In our case, the range of the probability threshold is 0.008 to 0.97. The extreme right end of the curve with high recall and low precision guarantees an operator to have the least amount of false negatives. In our case, by selecting a low threshold value the user or the operator can reduce the chances of missing any potential secrets. If one wants to reduce false positives significantly, they can increase the threshold value and move to the upper left end of the curve achieving high precision. Therefore, with this precision-recall curve and threshold tuning, the user can pick the appropriate balance between precision and recall for their application by selecting a corresponding threshold depending on the dataset.

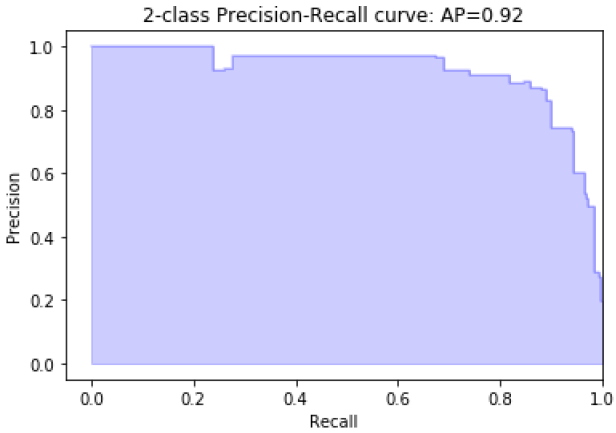

Fig. 1. Precision Recall Curve for target class 1 (Secrets)

Case Study

We now present an interesting case study that we discovered in the course of our research. On tuning the precision-recall curve shown in Figure 1 at a threshold of 0.4 (we classify a 'potential secret' as 'True positive' for a predicted probability > 0.4), we achieve a precision of 59% and a recall of 97%. On analyzing the confusion matrix we find 190 false positives out of 1894 and 10 false negatives out of 286. This implies that 10 passwords are misclassified as false positives. On further analyzing the misclassified passwords, we identify them to be from the pool of generic passwords. Low entropy generic passwords in test files such as 's3cr3t' and low entropy secrets with the English word 'password' in it are misclassified as being not sensitive. This example demonstrates that we need better features to represent generic passwords and more training examples to make our machine learning model robust.

VII. ERROR ANALYSIS

As mentioned in Section IV-A, we keep aside $\approx$ 350 examples as the development set on which we perform error analysis and identify the mislabeled entries. On manually analyzing the mislabeled entries, we make the following observations:

1) Secrets having a $ symbol at the start and having dots are misclassified as false positives. For example: $password\_hash = '\$2y\$13\$nJ1WDlBaGcbCdbNC5.5l4.sgy'$ is misclassified as a false positive.
2) False positives stored in a 'test' file having a high entropy (range of 3 to 4) are wrongly classified as passwords. These are the placeholders in the 'test' file but are not secrets.
3) Test keys having 'BEGIN PRIVATE KEY' tags are misclassified as passwords but are sample keys stored in a test folder.
4) High entropy strings, which are placeholders in a configuration file (*.yml* extension), are misidentified as a real secret. For example: CLIENT_SECRET: {{cf-client-secret-development}}
5) Real passwords having low entropy (less than 3) are misclassified as false positives. For example, String password = 'pencil' or 'root'.
6) Strings such as 'AKIAIOSFODNN7EXAMPLE' which are example keys but have the same format as an AWS API key are misclassified as real secrets.
7) Passwords or secrets placed in a configuration or properties file are correctly identified as a real secret.
8) API keys, hash codes, and RSA keys which have high entropy are correctly identified as true positives.

From our analysis, we observe that false positives which are function calls or variables or example keys in Readme files are correctly classified. Any real secrets placed in configuration, settings or properties file and private keys placed in a file with *.key* or *.pem* extensions are appropriately identified as secrets. However, any placeholders in a configuration file or

test/example file having a high entropy and example API keys with high entropy are almost always misclassified as real secrets. Moreover, any outliers such as hashcodes having $ sign or dots in it, and low entropy generic passwords are incorrectly classified as false positives. Our *potential* error analysis provides future directions in designing better feature set for generic passwords and outliers.

VIII. Ethical Conduct

In our research, we collect approximately 700 secrets by scanning 300 git repositories (≈ 28 million files), but we do not attempt to use any of the exposed secrets to verify the validity of the secret. We understand that some secrets may be non-sensitive, stale, or just invalid. Without actually testing such secrets, it is not possible to know with certainty that a secret is valid or exploitable [8]. We never try to use the leaked secrets on any platform thus, avoiding any ethical issues, such as obtaining any personal or sensitive information. Also, in our dataset, we do not include the secret value, file name, and repository name to avoid further leaking of secrets that can hurt the repository owners and developers.

IX. Conclusion

We built a generalized framework to detect different types of secrets and leveraged machine learning models to reduce false positives generated by secret detector tools. Using a Voting Classifier (combination of Logistic Regression, Naïve Bayes and SVM) we were able to reduce the number of false positives considerably. Users can also fine-tune the machine learning model by setting a different probability threshold to optimize the model based on their application requirements. As future work, we will design more representative features for the group of generic passwords to reduce misclassifications.

References

[1] "Removing sensitive data from a repository - github help," help.github.com/articles/removing-sensitive-data-from-a-repository/.

[2] K. Collins, "*Developers keep leaving secret keys to corporate data out in the open for anyone to take*," qz.com/674520/, May 2016.

[3] S. Knight, "*10,000 AWS secret access keys carelessly left in code uploaded to GitHub*," www.techspot.com/news, March 2014.

[4] D. Ayrey, "*Trufflehog*," github.com/dxa4481/truffleHog, November 2018.

[5] Auth0, "*Repo-Supervisor*," github.com/auth0/repo-supervisor, June 2017.

[6] AWS-Labs, "git-secrets," github.com/awslabs/git-secrets, Dec 2015.

[7] UKHomeOffice, "repo-security-scanner," github.com/UKHomeOffice/repo-security-scanner, Feb 2017.

[8] M. Meli *et al.*, "How bad can it git? characterizing secret leakage in public github repositories," in *NDSS*, 2019.

[9] V. S. Sinha *et al.*, "Detecting and mitigating secret-key leaks in source code repositories," in *12th Working Conference on MSR*, 2015.

[10] radekk, "*Detecting secrets in source code*," auth0.engineering/, June 2017.

[11] G. developers, "*API keys*," cloud.google.com/endpoints/docs/openapi/when-why-api-key, 2019.

[12] Okta, "*The Client ID and Secret*," www.oauth.com/oauth2-servers/client-registration/client-id-secret/, July 2018.

[13] Facebook, "*Access Tokens*," developers.facebook.com/docs/facebook-login/access-tokens/, 2019.

[14] C. E. Shannon, "Prediction and entropy of printed english," *Bell Labs Tech J.*, vol. 30, no. 1, 1951.

[15] A. D. Diego, "Automatic extraction of api keys from android applications," Ph.D. dissertation, UNIVERSITA DEGLI STUDI DI ' ROMA "TOR VERGATA", 2017.

[16] Microsoft, "*Getting started with Credential Scanner (CredScan)*," secdevtools.azurewebsites.net/helpcredscan.html, 2019.

[17] M. Henriksen, "*gitrob*," github.com/michenriksen/gitrob, June 2018.

[18] GitHub-Developer, "*REST API v3*," developer.github.com/v3/search/, 2019.

[19] GitPython, pypi.org/project/GitPython/, 2018.

[20] P. Gupta, "*Decision Trees in Machine Learning*," towardsdatascience.com/decision-trees-in-machine-learning-641b9c4e8052, May 2017.

[21] T. Yiu, "*Understanding Random Forest*," towardsdatascience.com/understanding-random-forest-58381e0602d2, June 2019.

[22] J. Brownlee, "*Logistic Regression for Machine Learning*," machinelearningmastery.com/, April 2016.

[23] D. Soni, "*Introduction to Naive Bayes Classification*," towardsdatascience.com/, May 2018.

[24] F. Pedregosa and et al., "Scikit-learn: Machine learning in Python," *JMLR*, vol. 12, 2011.

[25] A. Bronshtein, "*A Quick Introduction to K-Nearest Neighbors Algorithm*," medium.com/@adi.bronshtein/, April 2017.

[26] S. Raschka, "*EnsembleVoteClassifier*," rasbt.github.io/mlxtend/user_guide/classifier/EnsembleVoteClassifier/, 2014.

[27] Pandas, "*Python Data Analysis Library*," pandas.pydata.org/, July 2019.

[28] E. Alpaydin, *Design and Analysis of Machine Learning Experiments*. MIT Press, 2010.

[29] W. Koehrsen, "*Beyond Accuracy: Precision and Recall*," towardsdatascience.com/, March 2018.

[30] yellowbrick, "*Classification Report yellowbrick 0.9.1 documentation*," www.scikit-yb.org/en/latest/api/classifier/, 2016.

Appendix

A. List of features

We use the following binary features as input to our training and test dataset.

1) Does potential secret have parentheses? (Possible function call)
2) Does potential secret have brackets? (Possible variable declaration)
3) Does potential secret have periods? (Possible function call)
4) Does potential secret begins with a $ sign ? (Possible variable)
5) Does potential secret have the word 'Password' in it? (Possible variable initialization)
6) Does potential secret have spaces? (Possible sentence/loops)
7) Does line with potential secret have HTML tags? (HTML file)
8) Does line with potential secret start with #,*, /*? (Possible comment)
9) Does potential secret have an arrow in it? (Possible pointer variable)
10) Does File or directory path have sub-string "test" or "example" in its name? (Possible test secret)
11) Does potential secret have words like null/nil/undefined/None/true/false in it? (Possible programming language initialization)
12) Is potential secret a numerical value?
13) Entropy bins (To estimate the randomness of the potential secret); Is entropy in range of [0,1) or [1,2) or [2,3) or [3,4) or greater than 4?
14) The type of file in which potential secret is present; Is it in a configuration file or in a configuration folder? (.confg, .cfg, .yml file extension)?
15) Is it in a Settings file (Example: default.settings, settings.c)?
16) Is it in a Properties file (.properties file extension)?
17) Is it in a README file? (.rdoc, .rst, .md extension or README.txt)?
18) Is it in a language file?
19) Does it have ——BEGIN PRIVATE KEY TAG——?
20) Does file contain .pem, .key, .crt extension? (Possible private key files/ certificates)