# Improving the Accuracy of Detecting Hardcoded Credentials with Deep Learning

***Abstract*—Hardcoding credentials (e.g., account passwords, usernames, API access keys) refers to the dangerous practice of embedding plain sensitive text into the source code. Having used hardcoded credentials (HCs) for testing and debugging, developers often forget to remove them before releasing their code, thus inadvertently introducing a security vulnerability, exploitable by malicious parties to steal secret data and interfere with the system's behavior. Recognized as a serious security threat, HCs have been the target of prior mitigation efforts. Unfortunately, these prior approaches suffer from low accuracy, yielding excessive false positives that must be reviewed by hand. As a result, detecting HCs in large codebases is laborious and intellectually tiresome, thus escalating the attendant software validation and maintenance costs. In this paper, we investigate how the accuracy of detecting HCs can be improved with Deep Learning (DL). By applying our DL-based approach to a set of representative Java applications, we were able to achieve better precision and recall than existing approaches. Based on our evaluation results, we see the potential of using our approach in combination with existing techniques to improve the overall accuracy and reduce the required manual effort in identifying the presence of HCs in large codebases. Supplementing existing HC detection techniques with DL can improve software security and eventually become part of continuous integration.**

***Keywords*-Hardcoded Credentials, Machine Learning, Deep Learning, Software Maintenance**

## I. Introduction

Software development companies have increasingly adopted continuous integration (CI) and continuous deployment (CD) to verify that their source code meets the project's quality requirements. The practice of CI entails automating the process of building, packaging, and testing applications under development. Following a consistent integration process has been shown to lead to more commits, better team collaboration, and higher software quality [1]–[4].

One of the main benefits of CI is that it detects errors, bugs, and security vulnerabilities early in the development process, thus lowering future maintenance costs [5]. It has been shown that delaying the fixing of bugs and removing security vulnerabilities to later software development phases can increase the maintenance costs by orders of magnitude [6]–[8]. Therefore, companies actively look for ways to effectively and efficiently identify and fix defects as early as possible.

When it comes to security, the code under development can often afford to follow looser security standards than the same code that has been deployed. As software systems are being developed and fine-tuned, it typically works with mock data whose leakage would not cause any harmful consequences. The practices followed during these phases prioritize the convenience of developers in completing their tasks as easily and quickly as possible. One of the common practices that facilitates the testing and debugging of password-protected software modules is to hardcode credentials. Rather than requesting an external user to enter their credentials into the system, programmers tend to hardcode this information in the source code, so it contains hardcoded credentials or HCs for short. This practice greatly facilitates debugging and testing.

```
private String userName = "admin";
private String password = "root";
```

Fig. 1. Hardcoded credentials

HCs frequently contain account information, including passwords, SSH keys, and other sensitive information. Figure 1 shows an example of HCs in a Java code snippet. Despite their usefulness in the fine-tuning and testing phases, HCs present a serious security vulnerability if they have not been removed from the source code before deployment. In fact, the incidence of HCs in the production code can be exploited to steal secret data and subvert the system's behavior [9]. Hence, the last phase of CI must identify and sanitize HCs before the code is deployed to prevent this dangerous vulnerability from seeping into production. Unfortunately, the current state-of-the-practice approaches, techniques, and tools for detecting HCs are not reliable, as they are unable to accurately differentiate between real HCs and false positives, occurrences of literal strings that contain no sensitive information [10], [11]. Such string literals can occur quite frequently in large systems, storing information that ranges from simple logging messages to public authentication keys.

Having searched for given targets, any detection technique can return *true positives* and *false positives*, while failing to return *false negatives*. False positives are misidentified targets. False negatives are genuine targets that are mistakenly left undetected. Consider how the presence of false positives and negatives in the detection techniques for vulnerabilities impact *security* and *human effort*. False positives do not jeopardize security per se, but their high incidence incurs a high human effort, required to carefully examine each instance by hand. If this effort becomes prohibitively expensive, security would also be compromised. In contrast, false negatives immediately hurt security, as vulnerabilities remain undetected. However, in the absence of any false negatives, human analysts have nothing to analyze, so no additional effort and attendant costs are incurred. Existing approaches to detecting HCs are

designed to minimize the number of false negatives, while remaining oblivious to the number of false positives, which tend to be quite high [10]. This high number, however, increases the human effort required to check each instance of misidentified HCs. Besides, the additional time required for such checking may violate the guiding principles of CI, which encourages rapid validation cycles. Hence, developers often find it impossible to validate their projects on time, so production code with undetected HCs in it ends up shipped to the customer.

*Use Case: Developing and Testing a Login Component*

Consider developing a module that among other functionalities allows authorized users to log in to an online banking application. Once all the required functionalities are in place, developers start writing tests to verify that the module works according to the specification. Specifically, these tests would need to cover use cases, security protection, and exceptional conditions. To eliminate the need to manually enter any usernames and passwords, developers can automatically generate and hardcode a credential, configuring the system to allow all access for this credential, including all critical services of the online banking application (e.g., bank account, credit history). Since the HC was automatically generated, it would unlikely to contain any commonly observed telltales. For instance, substrings and variations of "password", "secret", "admin", or "root."

In the case of budgetary restrictions or/and a rushed schedule, developers may end up not removing the HC from the test suite. Notice that due to the absence of telltales, the HC may have slipped past code reviews, making its way to production. Once in production, the software module now contains an exploitable vulnerability: if a malicious party makes use of the HC, the resulting damage can prove catastrophic both to the bank and its customers.

*Approach and Contributions*

This paper reports on the results of an investigation we conducted to explore whether the accuracy of detecting HCs can be improved with Deep Learning. To that end, we collected a large, representative dataset of open-source Java projects hosted on GitHub. From those, we extracted all string literals, which were filtered with Shannon's code entropy [10], [12], pattern matching, and other heuristics. We manually labeled the remaining strings as to whether they were HCs. The labeled HCs were then further analyzed to identify the presence of cross-correlations. This dataset was then used to train our DNN model. We then evaluated this trained model by applying it to unrelated projects, comparing its precision and recall with those of state-of-the-practice HC detectors.

Our approach for HC detection shows better precision and recall than existing approaches. We concretely realized our approach as MAGISTER, an HC detection tool for Java projects, available as a standalone module or as a SonarQube plugin. Additionally, we curated a list of passwords, passphrases, and substrings that may help security analysts during code reviews to identify hardcoded credentials. Lastly, we empirically analyzed open-source Java projects for the existence of HCs, from which we extracted a comprehensive dataset with more than 6.4 million strings for use by fellow researchers and practitioners.

The remainder of the paper is structured as follows: in Section II, we provide the background and related work; in Section III we describe the data collection methodology; in Section IV, we analyze the dataset and present the results of interviewing security specialists; in Section V, we use the collected data and its insights to train a Deep Neural Network (DNN) to identify hardcoded credentials in Java source code. Finally, in Section VII, we discuss our conclusions and future work directions.

## II. BACKGROUND & RELATED WORK

In this section, we describe the background necessary to understand our contributions. Additionally, we present how the state of the practice compares to our approach.

### A. Hardcoded Secrets

In this work, we refer to credentials as the sensitive information that allows users to access or deploy software. These credentials refer to secrets such as usernames, passwords, private keys, configuration files, etc. The following list describes the types of credentials commonly found in the source code [11]:

- API Keys: application programming interface keys authorize projects by identifying the app or the process invoking the API [13] (e.g., AWS keys allow access to Amazon web services).
- OAuth Client Secret and Client ID: secrets generated when the developer registers a service with an application. A client id is a public identifier for apps, while the client secret is only known to the app and the authorization server. For each registered application, one has to store its public and private keys [14].
- Access token: an opaque string that identifies a user, an app, or a page. It can be used by the app to make API calls. These tokens provide temporary secure access to APIs [15].
- SSH/SSL/RSA Private Keys: keys generally used to authenticate SSL certificates with a remote server.
- Generic passwords: common (username, password) pairs that grant access to applications and databases.

Despite the importance of credentials for security, developers often unintentionally leak them, leading to data breaches and expensive maintenance [10]. Popular resources, such as Docker and AWS VM instances, can also contain security vulnerabilities that threaten their users [16], as their instances often contain leftover secrets that attackers [17] can easily exploit. This issue is aggravated by the GitHub's history recovery feature, as new commits do not remove secrets from the repository history, so malicious parties can still access and exploit old secrets. Differently from previous work [10], we do not rely on Google BigQuery [18] or GitHub's key search API for the selection of the repositories for our data collection.

We follow a procedure similar to the one described by [11] for generating their dataset (Section III). In fact, we collected a considerably larger number of repositories, having relied on the list of popular open-source repositories of [19].

### B. Tools for Scanning Credentials

To identify potential secrets, the state of the practice relies on regular expression search, entropy checks, or a combination of the two [14], [20]–[22]. In particular, Shannon's code entropy [10], [12] is often used for identifying HCs, as Machine-generated credentials have higher entropy (e.g., API keys and private keys) than user-created generic passwords, which tend to be composed of natural language. Existing works [10], [23] have achieved high accuracy for the detection of machine-generated private keys (e.g., an RSA private key, API key) without considering user-generated passwords on their scans. CredScan and GitRob [24], [25] use pattern-matching to scan source code and potentially sensitive files. Existing tools have a high number of false positives, which is caused by regular expressions that match invalid entries [10], [13], [26]. These false positives reduce the effectiveness of the existing tools by necessitating the exertion of extensive manual, time-consuming, and error-prone effort to verify the results. This outcome is a direct consequence of their design objective to minimize the number of *false negatives*.

Our approach to detecting HCs leverages entropy analysis, dictionaries, and pattern-matching as the first step in the classification process. We use the attributes from the target strings to train a deep-learning model to classify whether a given candidate is a credential. Our methodology advances the state of the art as follows. While prior works use filters, such as entropy, words, and file types [10], [20], [23], [27] to identify HCs, our approach differs by relying on deep learning for classifying text. There is prior work that uses machine learning to reduce the number of false positives [11], by voting if a candidate is likely to be a credential. Our work differs by using the attributes of the candidates to better steer the DNN model in its classification. It does not rely on classic ML techniques for that end. Also, this study differs from [11] by collecting a comprehensive dataset of real-world open-source projects, which is used to train our DNN model. We leverage a DNN model informed by prior research that has demonstrated how Machine Learning (ML) techniques can accurately classify large data vectors if trained with a high-quality dataset [11], [28].

### C. Deep Learning & Text Classification

Deep Learning (DL) [29]–[32] is an artificial intelligence method inspired by the structure of neurons in the human brain. DL leverages artificial neural networks (ANN) for processing data and creating patterns for making decisions. DL has been applied to the fields of image and video recognition, natural language processing (NLP), and many other areas [28], [33]–[41]. DNNs combine the advantages of DL and neural networks to solve nonlinear problems while delivering better performance and accuracy than conventional ML algorithms [33], [42].

Text classification is an important task of Natural Language Processing with many applications, such as web search, information retrieval, ranking, and document classification [43], [44]. Recently, models based on neural networks have become increasingly popular [45]–[47]. While these models achieve very good performance in practice, they tend to be relatively slow both during training and testing, limiting their applicability to very large datasets. In the bag-of-word method, words are treated as independent features out of context. Our approach uses a DL model to analyze source code in a bag-of-words fashion. Our approach uses DL and text classification to determine if the detected strings are HCs. It does not rely on natural language processing for classifying text, but rather on attributes extracted from the strings, similar to the method used in [10].

## III. Data Collection Methodology

In this section, we present our data collection's motivation, system overview, implementation, methodology, and results. The data collected in this section is analyzed in Section IV and used to train MAGISTER in Section V.

### A. Objectives and Procedure

We downloaded open-sourced Maven projects hosted on GitHub. We have chosen Maven projects due to their large numbers and also for the abundance of resources available for parsing Java source code. For the search query "Java parsing," Google Scholar returns 136,000 results. *Our study's objective is to determine what are the properties of hard-coded credentials in open-source projects*. In particular, we obtain the metrics of Shannon's code entropy, use regular expressions to match known credential patterns, and look for tokens commonly found in credentials. Other security vulnerabilities (e.g., SQL injection, code sanitization), privacy, and code smells are considered as out of scope in this study.

Figure 2 shows the workflow for our data generation. We use the resulting dataset to train our credential scanning model in Section V. To that end, we used the dataset information from [19] to download open-source Maven projects hosted in GitHub. (**1**) We used JavaParser [48] to generate the abstract syntax tree (AST) of all Java files. (**2**) We extracted all string literals from the AST. (**3** and **4**) We reduced the number of candidates in the dataset by using pattern matching [10] to remove non-ASCII strings and Shannon's code entropy [49] to remove low-complexity strings, respectively. (**5**) We used the Levenshtein string distance to cluster the filtered candidates into similar groups [50]. The purpose of step 5 is to facilitate the process of manually labeling the dataset. (**6**) Lastly, manually label the resulting clusters as to whether or not they contain credentials.

### B. System Implementation

We implemented our system in three modules; a GitHub crawler, a project analyzer, and a dataset labeling tool. We

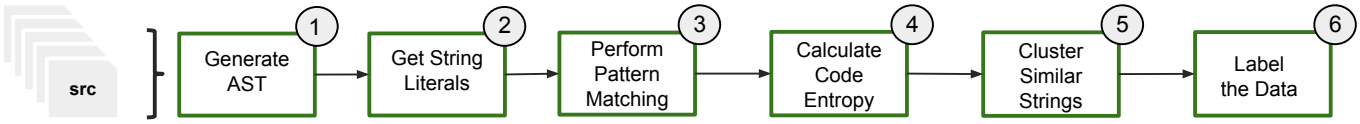


Fig. 2. Experimental workflow on the edge.

TABLE I
DOWNLOADED PROJECTS INSIGHTS & STATS

| Metrics | Values | |
|---|---|---|
| **Avg. Forks** | 5.41 | |
| **Avg. Labels** | 7.30 | |
| **Avg. Open Issues** | 1.85 | |
| **Avg. Watchers** | 12.61 | |
| **Avg. # Commits** | 2.22 | |
| **Avg. # Topics** | 1.12 | |
| **Most Forks** | android-stickyheaderswipelistview | 1253 |
| **Most Open Issues** | java-client-api | 2824 |
| **Most Subscribers** | rides-java-sdk | 2338 |
| **Most Commits** | Kotlin | 38741 |
| **Most Branches** | Kotlin | 865 |

used Python 3.7.3, the GitPython library, and the list of Maven repositories from [19] to implement the GitHub crawler. We used Java 1.8, JavaParser [48], Shannon's code entropy [49], and regular expressions that define code secrets to identify patterns string literals could be hard-coded credentials. Lastly, we used python 3.7.3 and the Distance python module to cluster the information of the dataset. All the source code we used for data collection available in the availability Section VIII.

### C. Results

In total, we selected 23,898 non-empty, public projects from GitHub to compose our dataset. From these, we excluded all non-Java files. The resulting dataset comprises approximately 1.7 million Java source files (26 Gb of data). From these source files, we extracted around 6.4 million string candidates. We further filter these candidates by means of pattern matching and code entropy. In total, we used 29 regular expressions that were curated from [10], [11], [20]. The resulting dataset is comprised of approximately 600k candidates. We clustered these candidates into 101,000 clusters for the labeling step. The resulting dataset and the source code used for the data collection are available in our Availability package VIII. Table I shows the average number of forks, labels, open issues, watchers, commits, and topics of the downloaded projects. Additionally, it shows the downloaded projects with the most forks, open issues, subscribers, commits, and branches.

### D. Specialist Interviews

We contacted security experts at a large multinational software development company. These specialists helped us with accurately defining which credentials and their attributes should be the target of our DNN model. Additionally, we used their input to guide the labeling process of the dataset (Figure 2 step 6). We further analyze the data collected herein in section IV and use it to train our model in section V.

In total, we contacted four specialists, requesting their guidance in defining our process of data labeling. From these, three were software engineers with more than 15 years of experience in security-sensitive software development, code reviews, maintenance, and verification of large enterprise software. Both had more than 5 years hands-on experience with addressing the security challenges in the development of enterprise software. One specialist was a principal security researcher with more than 30 years of experience in the development of computer systems.

```
private String filePath =
 "/response/tags/sample_get_list_user.json";
private String publicKey =
 "PublicKey=ywU4dD6IQdHrjYXAPcfvCP2zzX73lY%3D";
private String template1 =
 "grant_type=username=%s&password=%s";
private String template2 =
 "jdbc://%s/UID=%s/PWD=%s/charset=%s/";
```

Fig. 3. Code snippets with examples of the non-HC strings

As per their guidance, the most pertinent goal in detecting HCs is to be able to identify strings that may carry an unintelligible value (e.g., hashes, phrases, UUIDs). In contrast, they defined "the low hanging fruit" (e.g., strings named password, admin, or secret) as having a lower priority than the ones carrying unintelligible values. Human analysts should be able to easily spot such literals during code reviews. Additionally, public keys, emails, and user-identifiable data should be ignored as, even though they are considered code smells, they are not credentials. Figure 3 shows additional examples of strings that are **not** considered as credentials as per the feedback we received. These include file locations, templates for queries, and strings holding personal information (e.g., telephone numbers, names, addresses). Finally, our ultimate goal is to not overwhelm human analysts with multiple validation targets, but instead to facilitate the search for true credentials. Specifically, developers are concerned with reducing the number of false positives to a quantity manageable as a manual verification step.

### E. Data Labeling & Labeling Results

We manually labeled all the clusters following the guidelines provided by the specialists. Upon completing the cluster labeling step, we unraveled their contents and removed candidates with Shannon's entropy lower than 3.0, as we target strings with high degree of entropy [10]. Finally, we manually rechecked the remaining candidates to confirm the dataset labels.

In total, we labeled 101,592 clusters, from which 84,263 were labeled as *false* and 17,329 as *true*. Upon unraveling the contents of the clusters and performing the second pass over the classified data, we arrived at 17,279 strings labeled as *true* and 407,301 as *false*. We also curated a list with 60 strings, which are *easily identifiable HCs*. This list contains strings such as *K3yStor3p@ssw0rd* and *StrongP@ss!12*. We do not use any of the detected secrets to verify its validity. Developers may use the list to check their code. The full list is available in Section VIII.

### F. Ethics Considerations

Finally, we understand that some secrets may be non-sensitive, stale, or just invalid. We did not test or try any of the perceived secrets to determine whether or not they impact. Therefore, some of them may be invalid or no longer exploitable [10]. Furthermore, we did not use the secrets to obtaining any personal or sensitive information.

### G. Threats to Validity

As with any study, our evaluation also carries threats to validity. We identified the sources of validity threats. First, we manually labeled all the collected strings and clusters. We may have mislabeled some of the candidates, due to stress or fatigue. We attempted to mitigate this threat by performing additional passes over the complete dataset. Another potential threat to validity is the set of projects and procedures we used to extract the target candidates. We chose the list of projects from the study [19], different projects and procedures are likely to lead to the collection of different target strings. Another threat to validity are the specialist-provided guidelines, as their input has certainly impacted our labeling results. Finally, our system implementation strategies could have affected the results, with other implementations possibly yielding different results. Nevertheless, our implementation strategies followed widely acceptable principles by using known open-source solutions.

### H. Lessons Learned

Unfortunately, there are no tools or approaches for automatically classifying source code that will result in 100% precision and recall for detecting credentials. Therefore, we had to rely on manually analyzing the dataset. This process was extremely costly, as even though we reduced the number of candidates and added the clustering step, it still took more than 300 man-hours to perform the data labeling steps described in Section III. We caution practitioners that decide to reproduce their study to look for ways to optimize the data labeling step.

## IV. Data Analysis

This section describes motivating questions and methodology we used to analyse the labeled data gathered in Section III.

### A. Motivation and Research Questions

Hardcoded credential detection is a non-trivial process, as any string can store a credential. For example, a developer may choose to hardcode the following username "Smart.Cat" and password "ThisIsCool" to access an app's backend database. Notice that both strings do not have any telltales that suggest that they are credentials. Also, both could have been assigned to variables with random names (e.g., `temp1` and `temp2`), so they could be non-credential strings. However, to code reviewers, both strings should be marked as suspicious.

One of the hardest system design issues we face is to translate the experience of the analysis to our classification model. To make inroads in tackling this issue, we analyze the data collected in Section III for the properties that the labeled strings carried. Specifically, we analyze their code entropy, matched patterns, presence of known password substrings (e.g., pass, word, admin), thus seeking answers to the following questions: **MQ: Which string attributes are most likely to contribute to the labeled result?** The rationale behind MQ is that the information gathered from this data analysis to be used to train MAGISTER in Section V. Hence, the answer to MQ will provide valuable insight on how to train and optimize MAGISTER.

### B. Methodology

To answer **MQ**, we apply polynomial regression models to identify whether the collected performance metrics are correlated with the label. Each model uses a different set of given input attributes. We used the t-statistic data obtained from regression analysis to determine if the dependent variables were influential in describing the variation of the dependent variable. We used coefficient determination ($R^2$), F-test, and root mean squared error (RMSE) to determine the accuracy of the regression model. $R^2$ is a statistical measure of how close the data are to the regression line. F-test checks if the variances of two populations are the same by comparing the ratio of the variances. An F-test of 1 indicates that two populations are the same. The RMSE represents the differences between estimated and observed values. Additionally, it measures the model's accuracy. We used the measurements gathered from the empirical study (Section III) for the regression models. In total, we used 424,580 data points from the data collection (Section III). Refer to the Availability section VIII for the complete dataset. For the regression model, we divided the data into two sets, a training set with 80% of the data points (339,664) and the remaining 20% (84,916) for testing.

### C. MQ: On Attributes and Labels

*a) Pattern Matching:* We observed the following accuracy results with the polynomial regression models, for correlation between pattern matching and the label: **entropy** to **label**, 5th degree polynomial regression model has accuracy of 0.16 ($R^2$) and mean square error of 0.05. **Entropy** and **Pattern Matching**, in 6th degree polynomial regression. We observed a accuracy of 0.22 $R^2$ and mean square error of 0.05. **Entropy**, **Patterns** and **Number of Patterns**, 5th degree

polynomial regression model has accuracy of 0.27 ($R^2$) and mean square error of 0.04. **Entropy**, **Patterns**, **Number of Patterns**, and String length, 5th degree polynomial regression model has accuracy of -3.14 ($R^2$) and mean square error of 0.24. **Entropy**, **Patterns**, **Number of Patterns**, and **Substring weights**, 5th degree polynomial regression model has accuracy of 0.30 ($R^2$) and mean square error of 0.04.

*b) Heuristics:* We observed the following accuracy results with the polynomial regression models, for correlation between heuristics and the label: **entropy** and **heuristics**, in 6th degree polynomial regression. We observed a accuracy of 0.18 $R^2$ and mean square error of 0.05. **entropy**, **heuristics** and **number of heuristics**, 5th degree polynomial regression model has accuracy of 0.18 ($R^2$) and mean square error of 0.05. **Entropy**, **heuristics**, **number of heuristics**, and string length, 4th degree polynomial regression model has accuracy of -70.19 ($R^2$) and mean square error of 4.25. **Entropy**, **heuristics**, **number of heuristics**, and **substring weights**, 5th degree polynomial regression model has accuracy of 0.18 ($R^2$) and mean square error of 0.05.

*c) Pattern Matching & Heuristics:* We observed the following accuracy results with the polynomial regression models, for correlation between heuristics and the label: **entropy**, **pattern matching**, **number of patterns**, **heuristics**, **number of heuristics**, and **string weights**, in 5th degree polynomial regression. We observed accuracy of 0.31 $R^2$ and mean square error of 0.04.

### D. Summary of the Results & Discussion

We developed 10 regression models by means of leveraging the t-statistic data obtained from regression analysis, it was determined if the dependent variables are in describing the variation of the dependent variable. The accuracy of the regression models was determined using coefficient determination ($R^2$), F-test, and root mean squared error (RMSE). The developed regression models for the label and the variables of entropy, heuristics, patterns, number of heuristics, number of patterns, and substring weights are available in the availability package (Section VIII).

The following box summarizes the answers to our motivating question:

- 31% of the variation of the test set can be explained by the variation of **entropy**, **pattern matching**, number of patterns, **heuristics**, number of heuristics, and **string weights** in a 5th degree polynomial regression model with ($R^2$) 0.31 and mean square error of 0.05.
- We observed in our regressions models that **String length** is not correlated to the label value.

## V. MAGISTER

MAGISTER is a DNN that classifies candidate strings as credentials. While traditional hardcoded credential scanning approaches do not use DL, MAGISTER uses a DL model in conjunction with the target's string properties for its classification. Developers provide target strings for analysis, a set of properties extracted from the target strings, and MAGISTER outputs the probability of whether or not the candidate is a credential.

### A. Training

TABLE II
LAYERS OF THE NEURAL NETWORK

| Layer | Layer Size | Activation Function |
|---|---|---|
| **input layer** | 12 neurons | Rectified Linear Unit |
| **hidden layer #1** | 24 neurons | Scaled Exponential Linear Unit |
| **hidden layer #2** | 36 neurons | Rectified Linear Unit |
| **hidden layer #3** | 96 neurons | Exponential Linear Unit |
| **hidden layer #4** | 96 neurons | Scaled Exponential Linear Unit |
| **hidden layer #5** | 32 neurons | Rectified Linear Unit |
| **hidden layer #6** | 24 neurons | Scaled Exponential Linear Unit |
| **hidden layer #7** | 1 neurons | Sigmoid |

We trained the Keras Sequential model [51] by means of binary cross-entropy[1] between the predicted and ground truth on CPUs. The stochastic gradient descent variant Adam was applied as the optimizer [52]. We use 431,699 data vectors, totaling 43.6MB of data to train the DNN for the classification of string targets. The training set was collected by our data collection study (Section III). We used the training/validation split of 80%/20%. When updating global models, we selected the following hyper-parameters by optimizing them throughout the empirical analysis: 1000 for batch size and 450 for epochs. Table II shows the architecture of the DNN. We optimized the activation methods through trial and error, systematically experimenting with different configurations, activation methods, and numbers of neurons per layer to maximize the prediction accuracy for all metrics. Figure 4 shows the architecture of our DNN as well as its inputs and output.

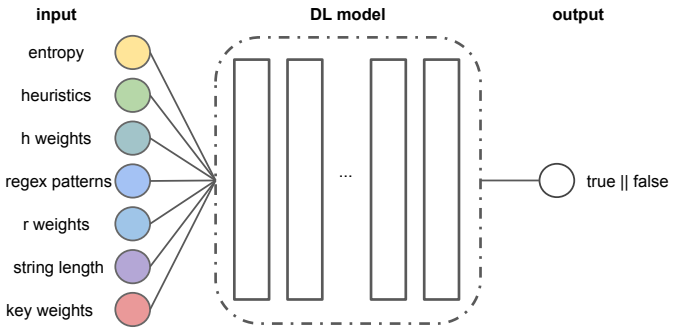


Fig. 4. MAGISTER Architecture

We used the insights we gained from Section IV to train the DNN. Specifically, to classify the string candidates, we use as input **entropy**, **pattern matching**, **number of patterns**, **heuristics**, **number of heuristics**, **string weights**, and **string length**. We converted these attributes to float values. Specifically, we assigned unique values to each of the regular

[1]https://keras.io/api/losses/probabilistic_losses/#binarycrossentropy-class

expressions, heuristic, and substrings. We use the same regular expressions used in Section III-C to perform the pattern matching.

We refer to heuristics as a combination of substring search for literal and variable's name. Specifically, we check if the following keywords are present in the variable's name and in its contents: *api*, *key*, *user*, *uname*, *pw*, *password*, *pass*, *mail*, *login*, *secret*, *emph*, *com*, *aws*, *metadata*, *json*, *github*, and */home/*. Candidates that do not match any of these keywords receive the value *none*. We assigned unique values to each heuristic, adding up the values if more than one heuristic is detected. We also search literal values that match for the strings that substring that we curated from the list of *easily identifiable HCs* (Section IV).

We chose to rely on the extracted attributes from the input strings, as current DNN models cannot receive string literals as *direct* input. Rather, a set of tokens, other values that describe the bag-of-words that form the string.

### B. Results

To classify the label of the target string, we use a training/-validation split of 80% (345,360 data points) / 20% (86,339 data points), resulting in: **label** accuracy values are 0.65 ($R^2$), mean square error of 0.09, and mean absolute error 0.97.

### C. Results Discussion & Sample Utilization

From operating MAGISTER, we derived the following insights. First, we added the string length parameter to the input list of MAGISTER, as we observed an increase in the model's accuracy. Even though, as observed in Section IV, the label value is not correlated to string length. Second, as we identified in Section IV, it is impossible to accurately estimate the label with only the string attributes. However, our DNN shows promising accuracy results, as 65% of our test set variation is explained by the training set.

Figure 5 shows MAGISTER's workflow. It follows similar steps as for the ones used in Section III for the data collection. Specifically, in step **1**, MAGISTER uses JavaParser [48] to generate the abstract syntax tree (AST) of all java files of the target project. In step **2**, it extracts all string literals from the AST. In **3**, it uses regular expressions to match the known credential patterns with the string candidates. In **4**, it calculates Shannon's code entropy of the string candidates. In step **5**, it checks for the existence of known easily spotted credentials and their substrings (e.g., "password", "admin", etc.) in the candidates. Lastly, in step **6**, the DNN model receives as input all the candidate's attributes and outputs the probability of the candidate being a credential. For our analysis, we consider any string with 60% or more probability as a credential.

Developers may deploy MAGISTER as part of their CI pipeline. Developers are responsible for fine-tuning MAGISTER's sensitivity. As previously mentioned, the DNN's output is a probability of the candidate being a credential. Therefore developers should take that into account and specify how likely is the MAGISTER to flag candidates. In other words, how likely they want to manually vet MAGISTER's output for false positives. Our availability package (Section VIII) includes a standalone version of MAGISTER, which is packaged as a jar file. Also, we make it available as a SonarQube plugin.

### D. Lessons Learned

It has been observed that increasing the size of the passwords is not likely to increase the security of a password. We observed this phenomenon in the dataset, as according to our regression models (Section IV) data length is not correlated to the label results. We cannot conclude if this finding is representative of the state-of-practice. It is likely to be explained by our choice of terms for analysis. Different datasets may show a correlation between the length of string and if it is a credential.

## VI. EVALUATION

This section describes the motivating questions, methodology, and experiments we conducted to evaluate MAGISTER. Note that this study is intended to inform practitioners about the deployment and utilization of MAGISTER.

### A. Empirical Study

Our evaluation objectives are to evaluate the differences in terms of *precision* and *recall* of MAGISTERand state-of-the-practice HCs detection tools. *precision* is the fraction of actual credentials among all the marked candidates. *recall* is the fraction of the total amount of credentials that were actually retrieved. The $F_1$ score is the harmonic mean of precision and recall, used to find a balance between precision and recall if there is an uneven distribution.

Therefore, our evaluation is driven by the following motivating questions:

- **MQ1**: How does MAGISTER compare to state-of-the-practice approaches for detecting HCs in terms of accuracy?
- **MQ2**: How does a combination of MAGISTER with state-of-the-practice approaches for HC detection compare with the state-of-practice?

The purpose of MQ1 is to determine how the accuracy (i.e., precision and recall) of state-of-the-practice credential scanning tools compare to MAGISTER. The purpose of MQ2 is to determine if MAGISTER can help improve the accuracy of state-of-the-practice credential scanning tools.

### B. Methodology

To answer MQ1, we selected 11 open-source Gradle Android projects, hosted on GitHub. We used the following search queries: "messenger + Android" and "edge + distributed + android" to select the projects. We chose Android projects that were not covered in our data collection (Section III). We chose the following state-of-the-practice credential scanning tools for our evaluation, SonarQube [53], TruffleHog [20], GitSecrets [21], Gittyleaks [54], and SpotBugs [55]. To answer MQ2, we used the aggregated analysis results from TruffleHog and GittyLeaks for each of the 11 projects used in MQ1 as

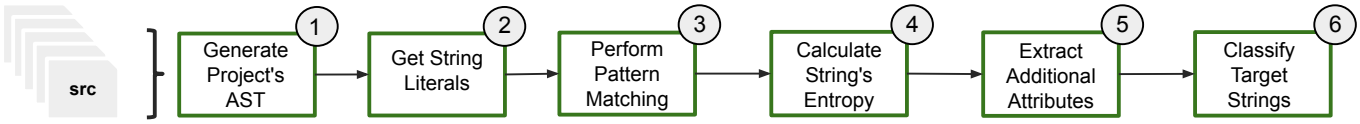


Fig. 5. Workflow of MAGISTER.

input for MAGISTER. We report the precision and recall of the combined version.

To measure precision, recall, and the F1 score, we manually curated a goldset of all strings of the source files from the downloaded projects. To that end, we used the guidelines provided by the specialists in Section III-D. We classified each string as either *credential* or *not a credential*. We used this information to compare with the execution results of selected tools, thus determining their accuracy.

### C. Results

Table III shows the number of false positives, true positives, and false negatives that MAGISTER detected while analyzing the selected repositories. These values show that depending on the configuration settings of MAGISTER and the contents of the projects, it can achieve very high accuracy. Specifically, for the project "CollEdge", it was able to achieve 80% precision and recall. We observed that for the projects "deltachat-android", "AndroidIM", and "react-native-fcm" MAGISTERshowed precision and recall of 0%, which means that for those projects, it only detected false positives. We observed that for the projects "SlyceMessaging" and "Mesibo Messenger" MAGISTER did not select any strings due to their low entropy.

Table IV shows the precision and recall of the selected code scanning tool under analysis. Cells with "*" indicate that the credential scanning tool did not detect any HCs. Regarding SonarQube's analysis results, we did not consider any marks that were not from SonarQube's built-in Java rule[2] for HCs detection. SonarQube has detected code smells, security hotspots, and other issues in the selected projects, but we considered those results outside the scope of our evaluation. Regarding GitSecrets and SpotBugs, both tools did not detect any HC in selected repositories. "MA+TH+GL" refers to the combined TruffleHog, GittyLeaks, and MAGISTER.

We caution that for different projects, configurations, and definitions of credentials, MAGISTER would likely achieve different accuracy results.

*MQ1 – On* MAGISTER *vs. State-of-Practice:* We observed that MAGISTER showed better accuracy than the state of the practice for most of the analyzed projects (i.e., RICCi, CallEdge, CaloFood, and Aveon). However, low accuracy values can still plague the process of code review.

*MQ2 – On complementing state-of-the-practice with* MAGISTER*:* We observed that using MAGISTER may increase the precision of the output from TruffleHog and GittyLeaks. Specifically, we observed that for the CaloFood project combined version achieved 100%, but at a lower recall (Table IV). Also, we observed that MAGISTER can be used to filter out false positives, as it correctly classified as not credentials the output from TruffleHog's and GittyLeak's for the following projects: SlyceMessaging, deltachat-android, react-native-fcm, and Mesibo Messenger.

[2]https://rules.sonarsource.com/java/RSPEC-2068

### D. Use Case Revisited

Recall the Login Component motivating scenario in Section I. Due to budgetary constraints and a rushed schedule, developers inadvertently pushed credentials to the production phase. Assume that MAGISTER has been added to the CI toolchain. MAGISTER can point out targets for code review, with better accuracy than other credential scanning tools. Depending on the MAGISTER's configuration, it should be possible to fine-tune the number of targets that analysts need to manually go through during code reviews.

To come up with a widely applicable and comprehensive model for classifying the string targets as credentials, one would have to study numerous combinations of attributes and additional code repositories. Our study is only the first step in this process. Our hope is that our findings can bootstrap many additional studies in this important area. By collating the results of such studies, one can define actionable information that would become a valuable asset for developers in the important tasks of verifying and validating their code.

### E. Lessons Learned

For scanning credentials, the cost of missing a credential (a false negative) is very high. By default HC scanning tools have low precision (too many false positives), so the preferred tool would have reduced the number of false positives to facilitate code reviews. Therefore, in the domain of credential scanning, recall is more critical than precision, which is the case in other domains [56], [57]. We pose that low precision also negatively affects the perceived trustworthiness of HC scanning tools, and this question warrants further investigation.

## VII. CONCLUSIONS AND FUTURE WORK

In this work, we put forward MAGISTER, a DNN that provides the likelihood of string targets being credentials. To arrive at the classification model, we performed an empirical study, in which we downloaded 23,898 open-source projects from GitHub. From those, we extracted around 600,000 strings via pattern matching and entropy analysis. We use specialists-provided guidelines to manually label all the targets as to whether or not they are credentials. Additionally, we used regression techniques to identify correlations between the

TABLE III
ANALYZED PROJECTS AND THEIR GOLDSET INFORMATION

| Project | Number of Java Files | LoC | True Positives | False Positives | False Negatives | Total | Precision | Recall | F1 |
|---|---|---|---|---|---|---|---|---|---|
| **RICCi** | 89 | 4839 | 3 | 10 | 18 | 32 | 23% | 10% | 0.139 |
| **CollEdge** | 175 | 14759 | 4 | 1 | 1 | 8 | 80% | 80% | 0.8 |
| **CaloFood** | 44 | 3800 | 7 | 7 | 4 | 14 | 50% | 64% | 0.561 |
| **Aveon** | 199 | 13264 | 1 | 8 | 9 | 9 | 11% | 10% | 0.10 |
| **Hue-Edge** | 34 | 3746 | 0 | 11 | 0 | 11 | 0% | 0% | 0 |
| **AndroidIM** | 18 | 1783 | 0 | 6 | 0 | 6 | 0% | 0% | 0 |
| **SlyceMessaging** | 55 | 2259 | 0 | 0 | 0 | 0 | * | * | * |
| **AppLozic-Android-SDK** | 337 | 52675 | 24 | 382 | 13 | 430 | 6% | 64% | 0.11 |
| **deltachat-android** | 384 | 42358 | 0 | 296 | 0 | 0 | 0% | 0% | 0 |
| **react-native-fcm** | 13 | 1271 | 0 | 12 | 0 | 12 | 0% | 0% | 0 |
| **Mesibo Messenger** | 23 | 3695 | 0 | 0 | 0 | 0 | * | * | * |

TABLE IV
PRECISION AND RECALL OF CREDENTIALS SCANNING TOOLS

| Project | Magister | | TruffleHug | | GittyLeaks | | MA+TH+GL | |
|---|---|---|---|---|---|---|---|---|
| | ***Precision*** | ***Recall*** | ***Precision*** | ***Recall*** | ***Precision*** | ***Recall*** | ***Precision*** | ***Recall*** |
| **RICCi** | 23% | 10% | * | * | 0% | 0% | * | * |
| **CallEdge** | 80% | 80% | * | * | * | * | * | * |
| **CaloFood** | 50% | 64% | 55% | 38% | 10% | 18% | 100% | 36% |
| **Aveon** | 11% | 10% | 0% | 0% | 2% | 8% | 0% | 0% |
| **Hue-Edge** | * | * | * | * | * | * | * | * |
| **AndroidIM** | 0% | 0% | * | * | 0% | 0% | * | * |
| **SlyceMessaging** | * | * | 0% | 0% | 0% | 0% | * | * |
| **AppLozic-Android-SDK** | 6% | 64% | 29% | 16% | 45% | 64% | 9% | 45% |
| **deltachat-android** | 0% | 0% | 0% | 0% | * | * | * | * |
| **react-native-fcm** | 0% | 0% | 0% | 0% | 0% | 0% | * | * |
| **Mesibo Messenger** | * | * | 0% | 0% | 0% | 0% | * | * |

attributes of the collected strings and the labels. We used the data analysis insights and the labeled dataset to train a DNN model to strings as credentials or not.

We have identified several possible future work directions. First, we would like to extend our prediction models to support additional languages other than Java. Second, we would like to expand the dataset size to include targets that were filtered out. Third, we would like to deploy MAGISTER to measure its usefulness in enterprise size software development settings. Last, we would like to enhance our approach to leverage the variable's contextual information, such as its control flow and utilization.

## VIII. AVAILABILITY

The project's source code, datasets, and collected metrics are available in the following online repository: https://github.com/mrmagister/magister-availability. THE FULL TRAINING DATASETS WILL BE MADE PUBLICLY AVAILABLE UPON PUBLICATION.

## ACKNOWLEDGEMENTS

OMITTED FOR BLIND REVIEW.

## REFERENCES

[1] M. Nath, J. Muralikrishnan, K. Sundarrajan, and M. Varadarajanna, "Continuous integration, delivery, and deployment: A revolutionary approach in software development," *International Journal of Research and Scientific Innovation (IJRSI)*, vol. 5.

[2] A. Kumbhar, M. Shailaja, and R. Anupindi, "Getting started with continuous integration in software development," 2015.

[3] S.-T. Lai and F.-Y. Leu, "Combining agile with traditional software development for improvement maintenance efficiency and quality," in *International Conference on Broadband and Wireless Computing, Communication and Applications*, pp. 254–264, Springer, 2020.

[4] L. Cruz, R. Abreu, and D. Lo, "To the attention of mobile software developers: guess what, test your app!," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2438–2468, 2019.

[5] A. Gautam, S. Vishwasrao, and F. Servant, "An empirical study of activity, popularity, size, testing, and stability in continuous integration," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 495–498, IEEE, 2017.

[6] H. Femmer, D. M. Fernández, E. Juergens, M. Klose, I. Zimmer, and J. Zimmer, "Rapid requirements checks with requirements smells: two case studies," in *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, pp. 10–19, 2014.

[7] N. Kiyavitskaya, N. Zeni, L. Mich, and D. M. Berry, "Requirements for tools for ambiguity identification and measurement in natural language requirements specifications," *Requirements engineering*, vol. 13, no. 3, pp. 207–239, 2008.

[8] B. Boehm, H. D. Rombach, and M. V. Zelkowitz, *Foundations of empirical software engineering: the legacy of Victor R. Basili*. Springer Science & Business Media, 2005.

[9] R. Singh Verma and B. Chandavarkar, "Hard-coded credentials and web service in iot: Issues and challenges," *International Journal of Computational Intelligence & IoT, Forthcoming*, vol. 2, no. 3, 2019.

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

[11] A. Saha, T. Denning, V. Srikumar, and S. K. Kasera, "Secrets in source code: Reducing false positives using machine learning," in *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pp. 168–175, IEEE, 2020.

[12] C. E. Shannon, "Prediction and entropy of printed english," *Bell system technical journal*, vol. 30, no. 1, pp. 50–64, 1951.

[13] Google, "Cloud endpoints for openapi," *https://cloud.google.com/endpoints/docs/openapi/*, 2020.

[14] OAuth, "The client id and secret," *https://www.oauth.com/oauth2-servers/client-registration/client-id-secret/*, 2020.

[15] Facebook, "Access tokens," *https://developers.facebook.com/docs/facebook-login/access-tokens*, 2020.

[16] J. Wei, X. Zhang, G. Ammons, V. Bala, and P. Ning, "Managing security of virtual machine images in a cloud environment," in *Proceedings of the 2009 ACM workshop on Cloud computing security*, pp. 91–96, 2009.

[17] C. M. Bishop, *Pattern recognition and machine learning*. springer, 2006.

[18] Google, "Google BigQuery Public Datasets," *https://cloud.google.com/bigquery/public-data/*, 2020.

[19] J. C. Davis, L. G. Michael IV, C. A. Coghlan, F. Servant, and D. Lee, "Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 443–454, 2019.

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

[21] M. Dowling, "git-secrets," *https:// github.com/awslabs/git-secrets*, 2015.

[22] U. H. Office, "repo-security-scanner," *https:// github.com/UKHomeOffice/repo-security-scanner*, 2017.

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

[24] Microsoft, "Getting started with credential scanner (credscan)," *https://secdevtools.azurewebsites.net/helpcredscan.html*, 2017.

[25] M. Henriksen, "Gitrob: Putting the open source in OSINT," 2018.

[26] F. Cheirdari and G. Karabatis, "Analyzing false positive source code vulnerabilities using static analysis tools," in *2018 IEEE International Conference on Big Data (Big Data)*, pp. 4782–4788, IEEE, 2018.

[27] https://auth0.com/, "Repo-supervisor," *https:// github.com/auth0/repo-supervisor*, 2020.

[28] B. Dantas Cruz, A. K. Paul, and E. Tilevich, "Stargazer: A deep learning approach for estimating the performance of edge-based clustering applications," *IEEE International Conference on Smart Data Services*, 2020.

[29] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[30] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[31] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[32] R. Johnson and T. Zhang, "Effective use of word order for text categorization with convolutional neural networks," in *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL HLT 2015*, p. 103, 2015.

[33] K. Kowsari, D. E. Brown, M. Heidarysafa, K. J. Meimandi, M. S. Gerber, and L. E. Barnes, "Hdltex: Hierarchical deep learning for text classification," in *2017 16th IEEE international conference on machine learning and applications (ICMLA)*, pp. 364–371, IEEE, 2017.

[34] D. Ciregan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *2012 IEEE conference on computer vision and pattern recognition*, pp. 3642–3649, IEEE, 2012.

[35] L. Deng, G. Hinton, and B. Kingsbury, "New types of deep neural network learning for speech recognition and related applications: An overview," in *2013 IEEE International Conf. on Acoustics, Speech and Signal Processing*.

[36] P. Baldi, P. Sadowski, and D. Whiteson, "Searching for exotic particles in high-energy physics with deep learning," *Nature communications*, vol. 5, p. 4308, 2014.

[37] G. Liu, Y. Xu, Z. He, Y. Rao, J. Xia, and L. Fan, "Deep learning-based channel prediction for edge computing networks toward intelligent connected vehicles," *IEEE Access*, vol. 7, pp. 114487–114495, 2019.

[38] N. Jain, A. Bhatele, M. P. Robson, T. Gamblin, and L. V. Kale, "Predicting application performance using supervised learning on communication features," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 95, ACM, 2013.

[39] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Y. Ng, and C. Potts, "Recursive deep models for semantic compositionality over a sentiment treebank," in *Proceedings of the 2013 conference on empirical methods in natural language processing*, pp. 1631–1642, 2013.

[40] T. Mikolov, M. Karafiát, and L. Burget, "Jan ˇcernocky, and sanjeev khudanpur. 2010. recurrent neural network based language model," in *Eleventh annual conference of the international speech communication association*, pp. 1045–1048, 2010.

[41] I. Sutskever, O. Vinyals, and Q. Le, "Sequence to sequence learning with neural networks," *Advances in NIPS*, 2014.

[42] M. Z. Nezhad, D. Zhu, N. Sadati, and K. Yang, "A predictive approach using deep feature learning for electronic medical records: A comparative study," *arXiv preprint arXiv:1801.02961*, 2018.

[43] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American society for information science*, vol. 41, no. 6, pp. 391–407, 1990.

[44] R. K. Bakshi, N. Kaur, R. Kaur, and G. Kaur, "Opinion mining and sentiment analysis," in *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pp. 452–455, IEEE, 2016.

[45] Y. Kim, "Convolutional neural networks for sentence classification," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1746–1751, 2014.

[46] A. Conneau, H. Schwenk, L. Barrault, and Y. Lecun, "Very deep convolutional networks for natural language processing," *arXiv preprint arXiv:1606.01781*, vol. 2, 2016.

[47] X. Zhang and Y. LeCun, "Text understanding from scratch," *arXiv preprint arXiv:1502.01710*, 2015.

[48] N. Smith, D. van Bruggen, and F. Tomassetti, "Javaparser: visited," *Leanpub, oct. de*, 2017.

[49] H. B. Barlow, T. P. Kaushal, and G. J. Mitchison, "Finding minimum entropy codes," *Neural Computation*, vol. 1, no. 3, pp. 412–423, 1989.

[50] S. Colianni, S. Rosales, and M. Signorotti, "Algorithmic trading of cryptocurrency based on twitter sentiment analysis," *CS229 Project*, pp. 1–5, 2015.

[51] F. Chollet *et al.*, "Keras." https://keras.io, 2015.

[52] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[53] SonarSource, "Code quality and security," *https://www.sonarqube.org/*, 2013.

[54] gittyleaks, "gittyleaks," *https://github.com/kootenpv/gittyleaks*, 2020.

[55] SpotBugs, "Spotbugs eclipse plugin. (2018)," *https://spotbugs.github.io*.

[56] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web.," tech. rep., Stanford InfoLab, 1999.

[57] S. Banerjee and A. Lavie, "Meteor: An automatic metric for mt evaluation with improved correlation with human judgments," in *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pp. 65–72, 2005.