A Credential Usage Study: Flow-Aware Leakage Detection in Open-Source Projects

Ruidong Han, Huihui Gong, Siqi Ma, *Member, IEEE*, Juanru Li, *Member, IEEE*, Chang Xu, Elisa Bertino, *Fellow, IEEE*, Surya Nepal, *Senior Member, IEEE*, Zhuo Ma, *Member, IEEE*, and JianFeng Ma, *Member, IEEE*

***Abstract*—Authentication and cryptography are critical security functions and, thus, are very often included as part of code. These functions require using credentials, such as passwords, security tokens, and cryptographic keys. However, developers often incorrectly implement/use credentials in their code because of a lack of secure coding skills. This paper analyzes open-source projects concerning the correct use of security credentials. We developed a semantic-rich, language-independent analysis approach for analyzing many projects automatically. We implemented a detection tool, SEAGULL, to automatically check open-source projects based on string literal and code structure information. Instead of analyzing the entire project code, which might result in path explosion when constructing data and control dependencies, SEAGULL pinpoints all literal constants to identify credential candidates and then analyzes the code snippets correlated to these candidates. SEAGULL accurately identifies the leaked credentials by obtaining semantic and syntax information about the code. We applied SEAGULL to 377 open-source projects. SEAGULL successfully reported 19 real-world credential leakages out of those projects. Our analysis shows that some developers protected or erased the credentials in the current project versions, but previously used credentials can still be extracted from the project's historical versions. Although the implementations of credential leakages seem to be fixed in the current projects, attackers could successfully log into accounts if developers keep using the same credentials as before. Additionally, we found that such credential leakages still affect some projects. By exploiting leaked credentials, attackers can log into particular accounts.**

***Index Terms*—credential leakage, bug detection, static code analysis**

I. Introduction

CREDENTIALS, such as authorization passwords, security tokens, and cryptographic keys, are crucial for securing accounts, protecting data stored in databases, and message transmission. Therefore, it is essential to protect against leakage. However, most developers only recognize the importance of building secure authentication schemes (e.g., multifactor authentication) or transmission protocols (e.g., TCP/IP) but ignore the protection of credentials, which are often hardcoded in plaintext in source code. Most research works [1], [2] focus mainly on exploiting protection schemes instead of analyzing whether the involved credentials are declared and used correctly. Recent password guessing studies [3], [4], [5], [6] indicate that attackers can exploit these leaked credentials to guess a user's other software/application credentials within an acceptable number of attempts. Furthermore, in the context of open-source or reused programs, other developers may refer to the design and implementation of the code, unknowingly exposing their credentials.

This work was supported by the National Natural Science Foundation of China(Key Program 62232013); Major Research plan of the National Natural Science Foundation of China (Grant No. 92267204, 92167203); Natural Science Basis Research Plan in Shaanxi Province of China (Grant No. 2022JM-338); Fundamental Research Funds for the Central Universities (XJSJ23185).

Therefore, we aim to conduct a security analysis to assess whether security credentials are protected and used correctly in project source code. Such an analysis is critical because it can help developers strengthen secure code practices with guidelines concerning the correct use of security credentials and reinforce the need to adopt these practices.

Concerning credential usage, we focus on widely used passwords and security tokens in this work. In this paper, the passwords consist of manual passwords, random passwords, and seed keys; the tokens consist of cryptographic keys and verification keys. Leakages of secret keys have been investigated in previous work [7], [8], [9], [10], which emphasizes the need for adopting good security practices when using credentials. Similarly, it is essential to note the correct and incorrect ways of using different credentials. For example, passwords used for identity verification should always be strongly protected and stored, along with the corresponding usernames, in local/remote databases. Unlike passwords, security tokens utilized for data transmission protection should be generated by cryptographically secure pseudo-random generators [11] and transmitted to the user/application via secure communication protocols.

Previous work [12], [13], [14], [15], [16] has addressed the problem of detecting credential leakage issues in open-source projects. Nonetheless, these approaches mainly rely on manually summarized rules and patterns to identify leakages in code or detect vulnerable patterns in credential usage. For example, Sonar-Java [15] has 600+ rules to assist in the detection of credential issues in Java code, and Bandit [16] uses AST trees to detect credential problems in Python code. Nonetheless, these approaches are typically limited to a specific programming language. Therefore they are not generally applicable to mixed open-source projects that implement different portions of code with different programming languages (e.g., C/C++, Python, and Java). The reason for using different programming languages is that a single language is often not the best fit for implementing certain code functions [17]. Therefore, recent research [18], [19], [20] has focused on designing tools for analyzeing open-source projects with code written in multiple languages. To address the challenges arising from differences in code logic, these approaches consider the entire source code as text due to the use of different languages. Then they use natural language processing to extract semantic and syntactical information from the source code. Although the application scope of these approaches can be extended, handling the entire source code as readable text leads to large numbers of inaccuracies in the analysis because the text obtained from the code may include nonhuman language words or combinations of words such as `def`, `abs` and `getSrcFrom`.

To address the lack of accuracy of the existing approaches, we implement a novel dual-model machine learning enhanced classification and detection approach that is programming language independent. Our approach consists of two major components: a credential classifier and a flow context classifier. In our approach, we first transform the project's source code under analysis into a semantic-rich, language-independent form consisting of string literal and code structure information. We use the credential classifier to classify string literals into credential candidates and ordinary strings to detect credentials. As a simple text analysis might result in high numbers of false-positives and false-negatives, we apply the flow context classifier to analyze the data and control flows correlated to the credential candidates to identify the sensitive activities. Sensitive activities use credentials for remote connection, authentication, and encryption. We then check whether the credentials are correctly used in these sensitive activities.

We implement SEAGULL, a credential leakage detection tool, and apply it to open-source projects. SEAGULL successfully detects 19 credential leakages out of 377 projects written in different programming languages and various historical versions. We also compare SEAGULL with the state-of-the-art language-independent analysis tools PassFinder, Detect-Secrets, and Gitleak. The detection results show that SEAGULL has the highest F1-score, 87.96%, and effectively identifies credential leakages in code written in various programming languages. Interestingly, SEAGULL identified two credential leakages from the historical versions of two projects, which have been fixed in the latest projects. We emailed project developers to confirm whether the credentials in the historical versions were still valid. One of the developer's comments was that the credential was still valid and then the developer disabled it. This result emphasizes that patching credential leakages in open-source projects requires ensuring that patches are also applied to the historical versions if these versions are accessible or abandoning credential usage forever.

Contributions:

- We design a novel cross-language approach for detecting credential leakages from diverse open-source projects. Unlike other cross-language analyses, we extract literal constants and code structure information and leverage them to understand the code's character-level string semantics and flow structures.
- We propose SEAGULL that combines natural language processing and flow analysis to more accurately and automatically locate credential leakage in the source code.
- We apply SEAGULL to real-world projects and found that credentials are incorrectly used in many projects. Although some developers were aware of the credential leakage issues and fixed them, the fixes were incorrect.
- We have open sourced SEAGULL at https://github.com/BlackJocker1995/Seagull.

II. PROBLEM AND SOLUTION

A. Credential Leakage

Listing 1 displays an open-source project on GitHub with 124 stars, that leaks an authentication token. Specifically, in Line 10, this code stores the secret (credential) in plaintext. It is leveraged for authorization (line 42), posing a security risk as it exposes the source credential within the source code. This indicates that the credential has been used or distributed by at least 124 developers, which increases the likelihood of its exposure to a wider audience.

Listing 1 Example with credential leakage

```
8   // client id/secret
9   public static final String CLIENT_ID =
    ↪  "5b074*****c16627***4";
10  public static final String CLIENT_SECRET =
    ↪  "a2c*******cd861c****c5c86e5****baf4c96a5";
    ...
40  CreateAuthorization createAuthorization = new
    ↪  CreateAuthorization();
41  createAuthorization.client_id = CLIENT_ID;
42  createAuthorization.client_secret = CLIENT_SECRET;
43  accountService.createAuthorization(createAuthorization)
```

In addition, some developers include credentials directly in the source code for code testing convenience. For instance, in many programs that utilize older versions of the jhipster framework [1], there is a test file (i.e., `TokenProviderTest.java`) that stores credentials in hard-coded plaintext. While this implementation is helpful for initial testing and is not strictly a leak (test credential), as the program matures or becomes open-source, such an implementation may inadvertently encourage other developers to apply similar credentials, leading to a "real leak". Although newer versions of jhipster have eliminated this file, other programs that rely on older versions remain vulnerable to this issue.

The proper way to use credentials is to avoid plaintext. We consider the example of using a password for authentication; some development kits (e.g., Android and WPF) apply password interfaces such as `passwordView`, and `JPasswordField`), which store the user input into a dynamic variable. Hardware-based methods [21], [22] consider storing the password in a hardware device (e.g., TPM and SGX) to prevent leakage. Another type of commonly used credential is tokens, namely, sequences of pseudo-random strings generated dynamically to verify data integrity and confidentiality [23], [24], [25]. These credentials should be stored in a file with the appropriate access rights.

B. Weaknesses of Existing Approaches

Although researchers have proposed many approaches for exploring leaked credentials, the following drawbacks are manifested:

[1]jhipster framework: https://github.com/jhipster

Weakness I: Converting source code into a textual representation. Many credential analysis approaches [12], [16], [18] consider source code textual pieces and pinpoint credentials using natural language processing. Although credentials are textual strings, it is inaccurate to convert all source code into a textual representation when assessing the security of credentials because the syntax and semantics of the source code are lost. Without understanding the details of credential implementations and credential handling processes (e.g., how a credential is stored as a variable of a specific type, and how it is used as a parameter of sensitive APIs), it is challenging to locate credentials accurately and determine whether a credential is used correctly. For example, a 16-byte string consisting of characters such as digits, letters, and symbols can be used either as a secret key or a public identifier.

Weakness II: Detecting credentials with language-dependent heuristics. Traditional approaches [13], [15] rely on prebuilt rules or patterns of leakage activities to detect credential leakages. Although pattern-based identification works well for specific credentials, such as security tokens, it does not suit a broader range of credentials. A methodology aiming to achieve accurate detection requires domain knowledge of the code (the used programming languages, credential-related APIs, etc.), often involving great manual effort. Moreover, the formats of credentials might not be similar across different projects. In these cases, heuristic-based approaches [26], [19], [27] will suffer from either many false-positive or many false-negative results.

Another issue is that increasingly many projects leverage more than one programming language. Considering the Android app ecosystem as an example, many Android apps are developed using both Java and C languages, and credentials are often shared between the Java and C modules [17]. In these cases, heuristic rules can only be applied to a specific object written in a single programming language [28], [29], but fail to find all credentials and detect all their leakages.

Weakness III: Simply applying machine learning to literal constants. To address open-source projects with millions of lines of code and diverss coding styles, recent approaches [30], [31] utilize machine learning (ML) to "recognize" credentials. Such approaches are promising, but their accuracy is still low compared to the accuracy of methods using fine-grained program analysis techniques. The shortcoming of most existing ML-assisted approaches is that they use context-independent classifications of literal constants. The classification model does not consider the code context where a literal constant is used. As a result, such approaches only achieve relatively coarse-grained identification.

C. Solutions

We combine semantic understanding with static code analysis to address the drawbacks of the existing credential analysis approaches. Specifically, our approach transforms the source code into a programming language-independent form. It then applies dual-model ML-enhanced, context-aware classification to detect credential leakages.

- Instead of directly searching for credentials in source code text, we leverage code querying techniques to find credentials. In this way, we handle source code in a unified way and fully use the intrinsic information of the code.
- Instead of only manually collating data to identify credentials, we use a semiautomated approach with Generative Adversarial Networks (GAN) to create many supplementary credential datasets in different forms and train a credential classification model with not only manually labeled datasets but also the generated datasets.
- Instead of merely using one ML model for string/symbol classification in source code, we train two models against literal constants and the code structure (i.e., code graph). The insight here is that a single-string classification model cannot achieve sufficiently accurate credential identification. We need to check the related codes of the literal constants to improve the identification accuracy. Therefore, in our approach, the source code is transformed into a **flow context**, a form containing the code of a credential and the data and control flows that are directly/indirectly dependent on it. We further develop a model for determining whether a flow context could trigger insecure credential usage. Credential leakage is confirmed with high confidence if both classifiers label the credential.

III. SEAGULL DESIGN

To assess the usage of credentials in software, we design and implement SEAGULL, which consists of three components (see Figure 1): *Code Description Retrieval* - it creates a codebase description composed of all variables, data, and control flows; *Credential Candidate Selection* - it explores the constants that might be credentialed with a character-level analysis. *Flow Context Classifier* - it explores insecure, sensitive activities for credentials and determines whether any credential is involved in a remote connection, authentication, and encryption.

Specifically, SEAGULL first assesses all source code for each project. Through code scanning, SEAGULL creates a codebase description, which contains the declared variables, the correlated data, and the control flows of the code (①②). Then it extracts all literal constants assigned to variables or used as function input arguments. Taking each literal constant as input, a *credential classifier* labels each constant that might be a credential as *credential candidate* (③ to ⑥). Given each credential candidate, SEAGULL tracks the data and control flows that are directly/indirectly dependent on the candidate (⑦ to ⑨). Finally, relying on a *flow context classifier*, SEAGULL detects flows that involve sensitive activities and labels such flows as *leakage* (⑩⑪).

A. Code Description Retrieval

To detect potential credential leakages in code, SEAGULL examines the code to pinpoint all the declared variables to which strings are assigned and string arguments in the function call. Based on these labeled strings, SEAGULL then tracks the execution flows that are directly/indirectly dependent on them. Specifically, SEAGULL initially establishes a codebase description recording the above information (i.e., constant

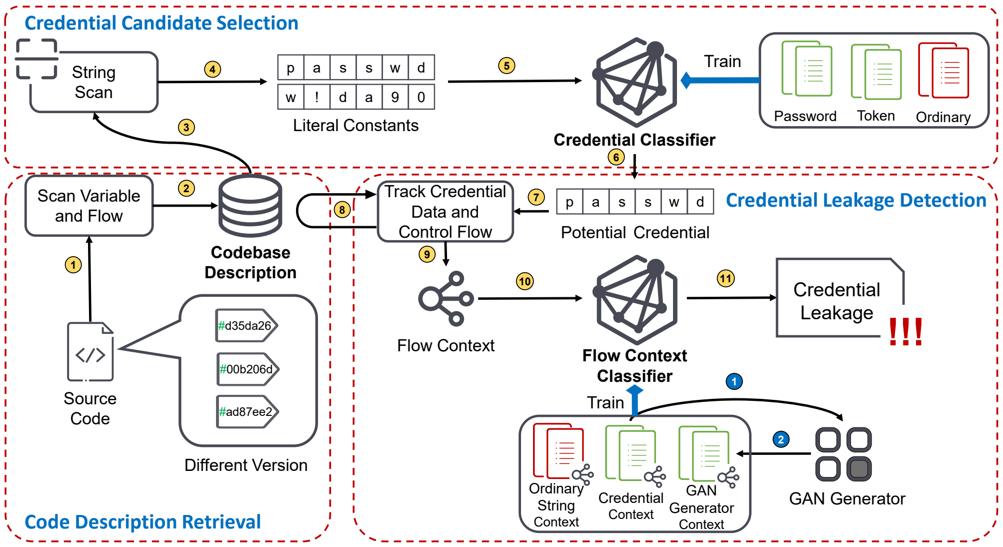


Fig. 1: Architecture of SEAGULL.

string value, variable, and flow) by applying CodeQL [32], an open-source semantic code analysis engine. CodeQL can convert code into a database-like form, capturing comprehensive information such as variables and call flow within the code. This transformed representation contains nearly all the relevant details necessary for analysis. In SEAGULL, we leverage rich information (database) to formulate various analysis policies designed explicitly for cross-language recognition of variables, strings, and flow.

In this paper, our primary focus is on analyzing code written in the most commonly used programming languages: C, C++, Java, C#, Python, and JavaScript. These languages are prevalent in open-source projects. Table I provides an overview of the environmental requirements for CodeQL to analyze code in these programming languages. Nevertheless, it is worth noting that SEAGULL can also be readily applied to projects written in other programming languages [2].

TABLE I: Environment requirement for different language.

	Language	Requirement
Compiled	C/C++ Java C#	gcc/g++ or Make or CMake Maven or Gradle .Net Core
Interpreted	Python Javascript	Python Runtime Javascript Runtime

1) Strings and Variables: Due to the variation across programming languages, we employ distinct schemes in SEAGULL to capture strings and variables.

Compiled languages (i.e., C, Java, and C#). They require static variable types to be declared in the source code, and SEAGULL identifies the variables' initialized statements and function call statements. Then, it selects the variables and function arguments of type `char*` or `String`.

[2]Supported languages and frameworks: https://codeql.github.com/docs/codeql-overview/supported-languages-and-frameworks/

Interpreted languages (i.e., Python and JavaScript). They are dynamically typed; SEAGULL determines the variable types by searching for the assignment statements and function arguments. Given an assignment expression, SEAGULL regards a value as a literal constant if listed in quotes or double quotes. The assigned variable on the left is labeled a *string*. A variable will be labeled a *string* if assigned by a *string* variable. Similarly, SEAGULL examines function invocation to determine whether any value in quotes is utilized as an input argument. If so, it identifies the position of the argument and locates the function declaration to obtain the corresponding variable.

2) Code Flow: Based on the *string* variable, SEAGULL obtains data and control flows directly/indirectly dependent on these variables via CodeQL. Specifically, we set the *string* variable as the source and other functions, variables, and arguments as the sinks. CodeQL infers the execution path from the source to each sink within the same file. All the information, i.e., *string* variables, the corresponding data, and flow dependencies, is collected to establish a *Codebase Description*.

B. Credential Candidate Selection

Since an open-source project could include many literal constants, manually filtering out the credentials is time-consuming. Hence, we build a string checker, *Credential Classifier*, to automatically classify literal constants into two categories, *credential* and *ordinary*.

1) Credential Pattern Analysis: By inspecting the credential strings, we observe that credentials differ slightly from ordinary strings in form, semantics, and length. For instance, credentials usually consist of a mix of alphanumeric and special characters, such as `!abc123000`, `a09acf0233558f34`; however, ordinary strings are common sentences or special identifiers with semantic meanings, e.g., `Process is finished!`, `SwithOn`.

Therefore, SEAGULL builds a credential classifier to study patterns of credentials. To achieve fine-grained classification, three groups of training data are taken as input, i.e., *password*, *token*, and *ordinary*. Specifically, a *password* is a literal constant consisting of several random characters used by sensitive functions (e.g., authentication and data encryption/decryption). A *token* is a literal constant used as a secret access key for integrity and confidentiality validation or a hash token for user authentication. All the other strings belong to the *ordinary* group.

2) Classifier Construction: As the character distribution in credentials differs from the distribution in ordinary strings, SEAGULL adopts a non-linear learning algorithm to build the credential classifier. SEAGULL first converts the strings into feature vectors. Since each string's characters have no semantic meaning, SEAGULL uses a character tokenizer for vector transformation. Specifically, it adopts the N-gram method [33] to split each string into substrings of a length shorter than N. For instance, when splitting the string `abc` with $N = 3$, the N-gram result is {`a`, `b`, `c`, `ab`, `bc`, `abc` }. Then, SEAGULL merges all results into a word list and generates an encoding dictionary based on the word appearance frequency. By leveraging this dictionary, SEAGULL converts all strings into vectors and pads them to length 512 with zeros. Afterward, SEAGULL uses CNN [34], a string-level Convolution Neural Network model for extracting local character features and building a credential classifier. The CNN model consists of an embedding layer, two convolution layers, two max-pooling layers, a dropout layer (with a 0.2 dropout rate), and a dense layer with ReLU activation.

By searching for the string values in the codebase description, the credential classifier outputs a probability score for each label (i.e., *password*, *token*, and *ordinary*) to which the string might correspond. SEAGULL then selects the label with the highest probability score for the string. All strings with *password* and *token* labels are regarded as credential candidates for further credential leakage checking.

C. Credential Leakage Detection

Sensitive activities using credentials usually involve similar function calls; thus, SEAGULL constructs a flow context classifier to study patterns of sensitive activities and further explores leaked credentials. This paper considers sensitive the most common activities requiring credentials, i.e., data/connection encryption, user authentication, and cryptographic algorithm setup.

1) Flow Pattern Analysis: Starting from a literal constant, we construct flow contexts by discovering all the data and control flows directly/indirectly dependent on this constant in the codebase description. Therefore, following the execution order, each flow context includes all the traceable information. The involved variables are saved in the `Variable Name` format. Additionally, SEAGULL record the invoked functions based on the type of function call (i.e., regular and internal). If the invoked function is regular (i.e., `call(args)`, SEAGULL records the function name and arguments in the format of [`Function Name`, `Argument Names`]. For an internal function (e.g., `object.call(args)`), SEAGULL records the name of the external object as well, in the format of [`Object Name`, `Function Name`, `Argument Names`].

To avoid overfitting caused by the imbalanced training dataset, we apply distribution-aware synthesis [35] based on the labeled credentials and ordinary flows. The synthesis relies on the distributional differences between credential flows and ordinary flows. We leverage a Generative Adversarial Network (GAN) [36] to augment the set of credential flows. A GAN consists of two competitive models, i.e., a generator and a discriminator. The former generates additional credential flows with similar credential use patterns, whereas the latter discriminates between real and fake flows. The generator and discriminator are trained simultaneously in multiple rounds. Finally, the generator create synthetic credential flows that are difficult to distinguish from the original flows.

2) Classifier Construction: The system accepts flow contexts containing both credentials and ordinary strings as input. It employs the deep learning algorithm TextCNN [37] to construct the flow context classifier. We initially convert each flow context into a feature vector to accomplish this. Since the variable names and function names commonly include natural language words and abbreviations, we segment all the names in the flow contexts (i.e., function, variable, and argument names) into meaningful words and abbreviations bu camel-cased and underline splitting. Then, we leverage the encoding dictionary to transform each flow context into a vector, which records the appearance frequency of each word to generate a frequency dictionary. The frequency dictionary is applied to convert all strings into vectors (padding to 256 with zero values). Unlike string values, the flow contexts contain variable names and function names, similar to natural language words containing obvious word boundaries. Therefore, we adopt TextCNN [37], a classification method commonly used for natural language processing. In particular, we use GloVe(6B-100) [38] as the embedding layer to reduce the impact of the data distribution. The model consists of one GloVe embedding layer, two convolution layers, two max-pooling layers, a dropout layer (with a 0.2 dropout rate), and a dense layer with ReLU activation.

3) Leakage Detection: To assess whether a credential is leaked, SEAGULL first searches for literal constants and establishes a codebase description for the project. Through the *Credential Classifier*, SEAGULL identifies the credential candidates labeled *credential*. It then constructs the flow context of each credential candidate by tracking and recording the data and control flows directly/indirectly dependent on the credential candidate.

SEAGULL transforms each flow context into a feature vector using the encoding dictionary. The feature vector is then given as input to the *Flow Context Classifier*, which labels it either *ordinary* or *leakage*. When a leakage is reported, SEAGULL then pinpoints the leaked credential in the codebase description and conducts backward slicing to locate the code snippets leaking the credential. If a literal constant's text is classified as a credential (by *Credential Classifier*), but it is not involved in any flow context (i.e., it is not used), SEAGULL will generate a warning.

SEAGULL addresses false-positives via two models. The first is the *Credential Classifier*, which is used to quickly screen potential strings, effectively reducing the analysis required by the second model. The second is the *Flow Context Classifier*, which uses a more precise method to analyze the potential credentials. The combined use of these two models enhances the time efficiency while at the same time ensuring detection accuracy.

IV. EVALUATION

We evaluate the performance of SEAGULL by answering the following research questions (**RQs**):

- **RQ1: Classifier Performance.** How well does each classifier label credentials and the corresponding leakages?
- **RQ2: SEAGULL Accuracy.** Does the dual-model improve the performance of SEAGULL in detecting credential leakages, and how well does SEAGULL detect leakages?
- **RQ3: Practical Application.** How well does SEAGULL perform in detecting projects in practice?

A. Experimental Setup

This section describes our experimental environment and how the experimental data are collected.

1) Experimental Program: We develop a Python-based SEAGULL demo. All experiments and training procedures are conducted on a server with the following specifications: an AMD Ryzen 3970X processor, 64 GB of RAM, an RTX 3090 graphics card, and the Ubuntu 20.04 operating system.

2) Credential Data Collection: To train the classifiers, we collected data from the following sources. To ensure data diversity, the duplicated data were removed.

- **GitHub.** We randomly select $13,422$ open-source projects published on GitHub, where their analyzed database comes from LGTM [3]. LGTM is an online analysis platform offered by CodeQL, that houses a collection of CodeQL analysis databases for various known projects on GitHub. Based on their database, SEAGULL exported all literal constants and established corresponding codebase descriptions. The credential pattern is a meaningless string comprising random numbers, letters, and symbols. We filtered out string values violating this pattern and marked them as *ordinary*, such as complete sentences, storage paths, description logs, and binary texts. For reset constants, we find their initial variable and check whether their variable names contain sensitive keywords such as `password`, `passwd`, `pwd`, `secret`, `auth`, `token`, `security`, `seed` and `key`, and further manually labeled the literal constants *password* or *token*. We generally labeled $1,643$ credentials (i.e., $1,050$ passwords and 593 tokens) and $97,271$ ordinary strings. Notably, we only considered the unique text here, not its flow.
- **RockYou2021 [39].** We included strings summarized in RockYou, a password leakage compilation of 8.4 billion unique password strings. By eliminating the non-ASCII passwords (i.e., non-English strings), we randomly choose $100,000$ strings as *password*.
- **Password and Token Generator.** Since some passwords and tokens might be generated by random number generators instead of humans, we leveraged a widely used password manager software, OnePass [40] to generate $100,000$ passwords comprised of 6-20 digits, letters, or symbols and $200,000$ security tokens by following the token formats of the mainstream internet service providers (e.g., Google [25] and AWS [24]).

3) Credential Flow Collection: For literal constants marked as credentials, we checked the flow contexts to explore whether any sensitive activities (i.e., TLS/SSL connection, authentication, and cryptography) are launched. If a function achieves a sensitive activity, we label the flow context a *credential flow*. Otherwise, it is labeled an **ordinary flow**. Flow contexts extracted from *ordinary* strings are treated as *ordinary flow*.

B. RQ1: Classifier Performance.

As SEAGULL relies on the performance of the credential and flow context classifiers, we evaluate each classifier separately. To assess the performance of each classifier, we calculated precision, recall, and F1-score as the evaluation metrics.

Credential Classifier. In total, the test experiment includes $201,050$ passwords, $200,593$ security tokens, and $97,271$ ordinary strings, of which 10% (i.e., $20,102$ passwords, $20,060$ security tokens, and $9,727$ ordinary strings) are used to construct the test dataset. By utilizing the training data, the classification results for the test dataset are obtained, as presented in Table II. On average, the credential classifier successfully identifies $39,669$ out of $40,162$ credentials, achieving a precision of 95.85%. It identifies $19,611$ passwords and $20,058$ tokens out of $20,102$ passwords and $20,060$ tokens, respectively. The average F1-score when identifying passwords, security tokens, and ordinary strings is 94.16%. It has a high false-positive rate when detecting ordinary strings because some ordinary strings follow credential patterns such as `ChangeSpeed` and `Surprise!`. Thus the classifier mislabels them as "password". Additionally, passwords provided by RockYou are simple, affecting precision and recall when distinguishing passwords from ordinary strings.

TABLE II: Performance of credential classifier.

	Password	Token	Ordinary
Reported	21,205	20,183	8,502
Confirmed	19,611	20,058	8,009
Precision	92.48%	99.38%	94.20%
Recall	97.56%	99.99%	82.33%
F1-Score	94.95%	99.68%	87.87%

Flow Context Classifier. Similarly, given the labeled flow contexts (i.e., $2,793$ credential flows and $97,271$ ordinary flows), we iteratively utilized 90% of them to train the flow context classifier and the remaining 10% to test performance. Besides, we assessed the diversity improvements contributed by GAN. Hence, we executed GAN to generate $10,000$

[3]CodeQL online platform: https://LGTM.com, which was merged into the GitHub at Nov,2022 https://docs.github.com/code-security

credential flows and then ran the classifier on the dataset with-/without the generated credential flows, respectively. Figure 2 shows a two-dimensional representation of the flow context distribution, where the blue dots represent ordinary flows, the orange dots represent authentic credential flows, and the green dots represent the GAN-generated credential flows. The green dots are distributed similarly to the orange dots; the GAN-generated credential flows are similar to the authentic credential flows.

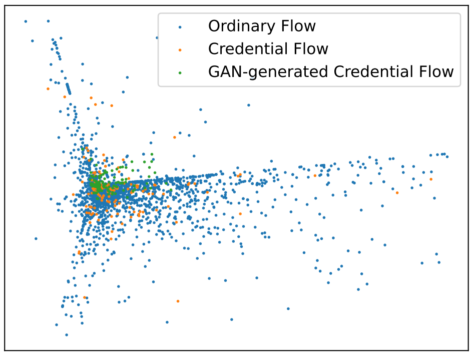


Fig. 2: Flow context distribution in a two-dimensional space.

Table III shows the results of the flow context classifier with/without using GAN. When using the additional generated credential flow contexts, SEAGULL successfully identifies 264 credential leakages out of 265, achieving an F1-score of 94.29%. However, SEAGULL misses 12 credential leakages when not using the generated data. The results show that the GAN-generated dataset increases the programming logic diversity and effectively improves the detection performance of the flow context classifier.

TABLE III: Performance of the flow context classifier with and without GAN.

	Credential Context		**Ordinary Context**	
	W	**W/O**	**W**	**W/O**
Reported	295	281	9,697	9,711
Confirmed	264	253	9,696	9,699
Precision	99.99%	99.98%	89.49%	90.04%
Recall	99.68%	99.71%	99.62%	95.47%
F1-Score	94.29%	92.67%	99.84%	99.79%

W means model trained with GAN. **W/O** means model trained without GAN.

Answer to RQ1: Two classifiers are well-trained to explore credentials and credential leakages, ensuring a general application scope of SEAGULL.

C. *RQ2:* SEAGULL *Accuracy*

After assessing the performance of each classifier, we evaluate SEAGULL from three aspects: 1) the necessity of the dual-model; 2) comparison with state-of-the-art works; 3) diversity of programming languages.

For the detection assessment, a sample of 300 real-world GitHub projects was randomly collected from LGTM, along with their corresponding databases. Figure 3 shows their GitHub star distribution. Approximately 52.35% of the projects have fewer than ten stars, 12.42% have 11-50 stars, 6.38% have 50-100 stars, 19.13% have 101-1000 stars, and 9.73% have over 1,000 stars. Then, SEAGULL applied the database to create a codebase description. Based on a codebase description analysis, 20,801 literal constants were identified. Subsequently, following the variable rules, 748 literal constants contain keywords `password`, `passwd`, `pwd`, `secret`, `auth`, `token`, `security`, `seed` and `key` and were marked as potential credentials. By manually checking the usage of these potential credentials, we finally confirmed 219 credential leakages, consisting of 169 password leakages and 50 security token leakages in 80 projects. Note that a project might use the same credentials to conduct different sensitive activities. For example, a credential might be used for authentication and encryption.

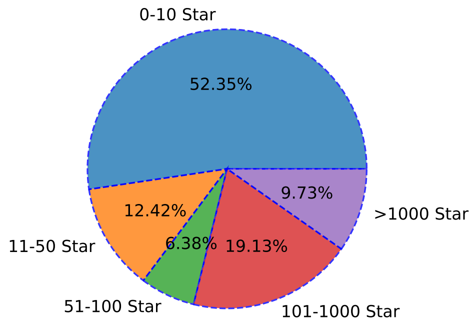


Fig. 3: Distribution of stars for the projects.

1) Necessity of the dual-model: To assess the necessity of the dual-model approach, we conducted an experiment in which we executed only one model at a time (either the credential or the flow context classifier) to evaluate the detection accuracy. Any literal constant reported as a credential is labeled credential leakage in the credential classifier-only scheme. Without applying the filtration of the credential classifier, the flow context classifier-only scheme identifies literal constants and constructs flow contexts to make judgments. Table IV reports the experimental results obtained using only one classifier.

TABLE IV: Detection results of the dual-model.

	Credential Classifier	**Flow Context Classifier**
Reported	319	241
Correct	201	192
Precision	63.01%	79.67%
Recall	91.78%	87.67%
F1-Score	74.72%	83.48%

Credential Classifier. When only analyzing literal constants without considering the usage of those constants, 319 credential leakages were reported, of which 201 are actual leakages, achieving a precision of 63.01%. We manually inspected the results and found that some ordinary strings are short and only include single words (e.g., `HalfSpeed`, and `Dodge!`). Thus, the credential classifier regarded these constants as credentials.

Furthermore, the credential classifier also missed 18 credential leakages. Some of them utilizes non-standard forms of tokens, such as `&^gHUIBHFJNFgdfgdJHGFHJ&^%` with symbol characters; others use longer private keys (e.g., RSA with length 2048), that are beyond the size of our dataset.

Flow Context Classifier. When relying only on the flow context analysis for exploitation, 241 leakages were reported, achieving an F1-score of 83.48%. Compared with the credential classifier, the flow context classifier is more accurate in detecting credential leakages with fewer false-positives. However, it is challenging to correctly identify sensitive activities that use credentials because the flow of using the credentials could be very different. Consequently, the flow context classifier might neglect vulnerable flows if the sensitive activities are not precisely defined.

In conclusion, none of the classifiers can balance the false-positives and the false-negatives; thus, the dual-model can eliminate the limitations of each classifier and achieve a better detection result (shown in Sec. IV-C2).

2) Comparison with State-of-the-Art Works: To evaluate the detection effectiveness of SEAGULL, we compared it with three state-of-the-art detection tools, PassFinder [18], Detect-Secrets [19], and Gitleaks [41]. Specifically, PassFinder applies a simple deep learning approach to recognize credential strings and identify credential leakages using natural language processing to check the statements surrounding the credentials. Detect-Secrets and Gitleaks detect credential leakages via regex-based approaches. Such approaches rely on heuristic rules driving the search for a wide range of secrets in code.

TABLE V: Detection results of SEAGULL, PassFinder, Detect-Secrets and Gitleaks.

	SEAGULL	**PassFinder**	**Detect Secrets**	**Gitleaks**
Reported	188	369	950	467
Correct	179	156	68	31
Precision	95.21%	42.28%	6.95%	6.64%
Recall	81.74%	71.23%	30.14%	13.72%
F1-Score	87.96%	53.06%	11.29%	8.95%

The comparison results are listed in Table V. In total, SEAGULL, PassFinder, Detect-Secrets, and Gitleaks successfully detected 179, 156, 66, and 31 credential leakages, respectively. The results demonstrate that SEAGULL is more effective than PassFinder, Detect-Secrets, and Gitleaks, achieving an F1-score of 87.96%. Generally, the regex-based approaches (i.e., Detect-Secretes and Gitleak) performed the worst. Although they reported many leakages (i.e., 950 and 467 reported leakages, respectively), most of these reports are false-positives. In detail, Gitleaks cannot distinguish password usage effectively. Among the 31 credential leakages that are correctly reported by Gitleaks, only 2 password leakages are reported. Detect-Secrets successfully identified 52 password leakages and 14 security token leakages.

We manually inspected the cases that were reported incorrectly and found that Detect-Secrets and Gitleak: 1) mistakenly labeled the strings (e.g., UUID and regular expression for testing) in the annotations and comments as credentials and further regarded these credentials as leakages; 2) identified the credentials declared in the dead code, that is, no sensitive functions use the declared credentials. Both tools detect credential leakage without considering the semantics of the source code, which triggers false-positives by reporting unused credentials, tokens in comments, format log strings, and hash strings.

TABLE VI: F1-Score comparison between the credential classifiers of SEAGULL and PassFinder.

	Password	**Token**	**Ordinary**
SEAGULL	94.95%	99.68%	87.87%
PassFinder	93.37%	99.50%	84.12%

PassFinder (a deep learning approach), which performs better by achieving a higher precision of 53.06%, misses 63 credential leakages, causing many false-negatives. An additional analysis was performed exclusively between our credential classifier and PassFinder's. We utilized the identical training and test dataset as SEAGULL, as described in Sec. IV-C1, and evaluated their respective performances. Table VI presents the F1-Scores for different classes. Our classifier demonstrates superior performance under the same testing conditions. The distinctive characteristic of credentials is their composition of letter-number combinations (e.g., `abc123`), setting them apart from regular words. As a result, correlations exist not only within these character (`a`, `b`, ..., `3`) but also between combinations (`abc` and `123`). Consequently, the additional processing of N-gram takes into consideration these correlations between combinations as well. The more critical difference is that the two have different ways of extracting context. PassFinder relies on analyzing the code block invoked six lines before and after the statement where the credential is operated. However, in real projects, the declaration and use of credentials are not always close, as demonstrated in Listing 1 (Line 10 and Line 43). The issue with PassFinder is that its extracted context does not effectively describe the relationship between a string and its usage when it is far from the declaration. Instead, SEAGULL gives more attention to the flow context, which is more reasonable and accurate. Compared to the code block, the flow context enables the description of credential usage scenarios across functions and user-defined functions (e.g., encryption and authentication). In addition, the performance of SEAGULL depends on the suitability of the training dataset. Hence, some customized credential formats and names cannot be distinguished because our dataset does not include the corresponding naming pattern or credential formats. For example, we found an authentication named `userDump`, whose natural semantics are treated as irrelevant to sensitive activities.

3) Language Diversity: As SEAGULL analyzes projects written in multiple languages, we specifically analyzed whether SEAGULL could exploit each type of programming language effectively. Furthermore, we compared SEAGULL with a language-dependent tool, CodeQL-Hard-coded, which is an official script provided by CodeQL for identifying credentials in Java.

Table VII lists the credential leakages written in different programming languages. On average, SEAGULL achieved a recall higher than 80% for most programming languages, especially Python projects, with the highest recall of 91.67%. As SEAGULL reported more false-positives when analyzing projects written in C#, we manually analyzed these projects. Consequently, we found that 1) some tokens are defined in customized formats, which cannot be recognized via the standard forms provided in the training dataset; 2) some customized cryptographic functions have complicated names (e.g., `WXBizMsgCpt`), which cannot be detected by SEAGULL, even though the utilized credential is identified. Nonetheless, the above issues are not specific to C#, which demonstrates that SEAGULL supports cross-language analysis as long as the credential formats and function naming principles follow the general standards.

Additionally, we ran the language-dependent tool CodeQL-Hard-coded to explore credentials in Java projects and compared its detection results with the results of Java projects reported by SEAGULL (shown in Table VIII). The results show that SEAGULL achieves a high precision and recall of 93.70% and 80.40%, respectively. However, CodeQL-Hard-Coded only achieves a precision of 0.59%. These results indicate that SEAGULL achieves high accuracy for cross-language analyses and when analyzing code written in a single programming language.

TABLE VII: Detection accuracy of SEAGULL for different languages.

	Java	**C#**	**Python**	**Cpp**	**JS**
Total Leakage	148	26	24	11	10
Detected	119	20	22	10	8
Recall	80.4%	76.9%	91.6%	90.9%	80.0%

TABLE VIII: Detection results of SEAGULL and CodeQL for the Java language.

	SEAGULL in Java	**CodeQL-Hard-coded**
Reported	127	16,546
Correct	119	97
Precision	93.70%	0.59%
Recall	80.40%	55.75%

Answer to RQ2: SEAGULL is highly effective in detecting credential leakages across different programming languages. Through the credential classifier, SEAGULL avoids many non-relevant explorations, which addresses the path explosion issue. Besides, the flow context classifier ensures that SEAGULL is suitable for various programming styles of different developers.

D. RQ3: Practical Application

To verify whether SEAGULL can be applied to various open-source projects, we randomly downloaded 377 open-source projects including 101 C/C++, 76 C#, 26 Java, 86 JavaScript, and 88 Python projects, and applied SEAGULL to detect credential leakages. SEAGULL considered the last ten commits for each project for analysis, taking approximately 20-70 seconds to establish codebase descriptions locally for each project. They contain 2,706 literal constants, and detailed information, including text, file path, and project information, are listed on the website[4]. If a literal constant appears in the same path across multiple commit versions, its records will be merged into the latest commit. In total, SEAGULL reported 22 credential leakages. After analyzing these credentials manually, we confirmed that 19 out of 22 (86%) credentials are leaked by 19 projects, including 2 projects with C/C++, 4 with C#, 10 with Java, 1 with JavaScript and 2 with Python. Table IX reports credential leakage examples from the selected real-world projects. Although two leakages are located in the historical versions of the projects, leaving the credentials without any extra protection still rendered them vulnerable because humans tend to reuse easy-to-remember passwords; thus, developers could reuse the same credentials as input for sensitive activities.

Interestingly, we observed password leakage in one of the historical versions of ua***gv, which is a configuration leakage detection project (Listing 2). The password `EGHUP**ZHLNLS` and email sender `m133***30@163.com` and receiver `han78***98@ilve.com` are declared as a string literal in the source code (Lines 9-11). The password is used to log into the sender's email (Line 19) and send messages to the receiver (line 20). The context of `EGHUP*****ZHLNL`, which includes `[password, smtp, login, sender]`, is deemed by SEAGULL to be an instance of credential leakage. The leakage is present in a historical version and then fixed in the current version by removing the email addresses of the sender and receiver (patch #54ed**a). Unfortunately, such a patch is insufficient to address the leakage because attackers can easily launch a man-in-the-middle attack to obtain the sender's email. Similarly, jum**llan/andr*****-api removes the entire file `Config.java` in #4cdb**8, but the leaked token is still present in Git commits, which also represents a vulnerability attackers can exploit to launch a man-in-the-middle attack.

Listing 2 Historical credential leakage in an open-source project.

```
9   - sender = "m133***30@163.com"
    + sender = ""
10  - recver = "han78***98@ilve.com"
    + recver = ""
11  password = "EGHUP*****ZHLNLS"
12  message = MIMEText(content, "plain", "utf-8")
    ...
18  smtp = smtplib.SMTP_SSL("smtp.163.com", 994)
19  smtp.login(sender, password)
20  smtp.sendmail(sender, [recver], message.as_string())
```

Answer to RQ3: SEAGULL successfully analyzes various language projects to detect credential leakage in both current and historical versions.

[4]https://github.com/BlackJocker1995/Seagull/blob/master/real_test/real_project.csv

TABLE IX: Credential leakages in selected real-world projects.

Project	Language	Location	GitHub Star
4***0n/r***e	Java	src/main/java/com/rarchives/*****/ripper/rippers/TumblrRipper.java	903
jum**llan/andr*****-api (#4cdb**8)	Java	app/src/main/java/com/androidstudy/mpesa/Config.java	135
chri***an1741/Twit*****Analysis	Python	src/StreamingTwitter.py	18
Blac****5/ua***gv (#54ed**a)	Python	Cptool/mailtool.py	12
nm***-repo/nm*****uncer	C#	client/csharp/SampleBouncerApiClient/BouncerClient.cs	1

The names of the projects are partially anonymized.

E. Usage of Credentials

By analyzing the collected credential flows, we studied the general application scenarios of credentials to understand what sensitive activities can be affected. Specifically, we summarized the frequency of words appearing in credential flows. Figure 4 shows the top 20 words that appear frequently. Meaningful words (e.g., `authorization` `service`, `assert`, `sql`) indicating remote services and data management are included.

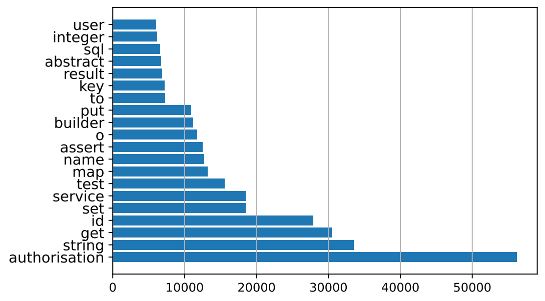


Fig. 4: Top-20 words distribution in context flow of credential.

By analyzing the vulnerable code, we saw that the credential leakages even affect the authentication services of mainstream platforms such as Amazon, Tumblr, and GitHub. For instance, r**me (Lstlisting 3), an album ripper plugin for quickly downloading all images in online albums with over 900 stars on GitHub, initializes a default literal constant API key `DEFAULT_API_KEY` in Line 37. The flow context containing `[TUMBLR_AUTH_CONFIG_KEYAUTH_CONFIG_KEY, Utils, getConfigString]`, is deemed as an instance of credential leakage. Through static code analysis, we found that the API key is a default API key for communicating with `tumber.com` (Line 40). Therefore, the default API key will always be utilized to configure the remote connection with tumblr. As a functional plugin, other developers might invoke this vulnerable plugin as an external package to build their projects, which might widely spread the vulnerability. Such a vulnerable yet widely used plugin is a notable example of a lack of security in software supply chains.

We also identified a misapplied credential used for access control. BullN*****ugin (Listing 4), an official Plugin of `BullNexRP`, embeds a literal string password as `R4D****mbg` in the source code. Using that username and password, an attacker can access the database at the remote server of `jdbc:mysql` and carry out malicious activities (i.e., stealing/deleting data). The flow context containing `[passwd, getConnection, url, user]`; is deemed an instance of credential leakage.

Listing 3 Tumblr authentication with credential leakage

```
22  private static final String DOMAIN = "tumblr.com",
23                              HOST   = "tumblr";
    ...
37  private static final String DEFAULT_API_KEY  =
    ↪  "JFNLu3Cb****RdUvZib****pSEVYYt****6o8Y****ZIoKyuN";
38
39  private static final String API_KEY;
40  API_KEY = Utils.getConfigString(TUMBLR_AUTH_CONFIG_KEY,
    ↪  DEFAULT_API_KEY);
    ...
83  checkURL += "/info?api_key=" + API_KEY;
84  JSONObject json = Http.url(checkURL).getJSON();
```

Listing 4 Remote access control with credential leakage

```
static String url = "jdbc:mysql:/xxxxxx.com:3306/xxxxx";
static String user = "dnlc****ZQ";
static String passwd = "R4D****mbg";
...
Connection conn = getConnection(url, user, passwd);
```

V. Related Work

This section discusses related work on vulnerability detection and sensitive information leakage in source code.

A. Automated Vulnerability Detection

Traditional vulnerability detection methods, such as [42], [43], [44], [45], rely on rule-based analyses but often suffer from high false-positives rates. More recent approaches [28], [46], [47], [48], [49], [29] rely on deep learning methods to reduce false-positives. Russell et al. [28] implemented an automated vulnerability detection tool for source code using deep machine learning. Their approach is based on a custom C++ lexer for capturing the content meaning of critical tokens and on a token vocabulary dictionary, which transforms all software repositories into standardized vectors. Then, the tool combines a convolution model with a random forest classifier to detect leakages. A similar approach was subsequently proposed by Li et al. [46]. They proposed a framework that transforms the sample source code into a minimal intermediate representation obtained by code-dependent analysis and code slicing. The framework uses three concatenated convolutional neural networks to extract high-level features from representations. Then, a classifier is trained to detect vulnerability features.

The Funded tool by Wang et al. [47] is based on a novel graph-based learning method for capturing program dependencies, thereby leveraging recent advances in neural graph networks (CNNs). Wang et al. created a dataset from the wealth of historical information available from open-source projects; they manually marked vulnerable source code snippets in historically committed patches. Based on this dataset, they trained a deep learning model to detect vulnerable code snippets. Alon et al. [49] proposed a general path-based representation method that takes ASTs of code snippets as input ,applies a deep neural network model to detect code semantics and, based on these semantics, catch vulnerable code snippets. The NLP-EYE tool by Wang et al. [29] is a source code-based security analysis system. It leverages natural language processing (NLP) to detect memory corruptions in source code. Our work differs from those approaches in that we focus on detecting credential leakages, whereas those focus on other vulnerabilities, such as memorable ones.

There are also approaches [50], [51], [52], [53] that use code similarity to detect vulnerabilities. For example, the tool called Vulpecker, developed by Li et al. [52], automatically detects vulnerabilities in source code. It does this by using characterization patch features and a code-similarity algorithm. Rather than source code, the VulHunter tool by Guo et al. [53] uses the intermediate representation output bytecode as input to a neural networks. VulHunter uses a bidirectional LSTM to build a neural network to analyze the similarity of two input bytecodes. It determines whether a code is vulnerable by calculating the similarity between the target program and a set of vulnerability templates. These methods are not applicable to the problem of credential leakage detection because of the enormous diversity of credentials in source code.

B. Sensitive Information Leakage

As developers may inadvertently leave sensitive information in public source code, previous works have begun to focus on this issue and propose solutions. The differences between them are shown in Table X. PassFinder by Feng et al. [18] identifies potential password leakage using neural networks. PassFinder relies on a password dataset manually constructed by analyzing $64,050$ GitHub projects. PassFinder then takes the password dataset and relevant code contexts as input to train password and context models, respectively. The models are utilized to identify password leakages. Although PassFinder detects password leakage from GitHub projects effectively, substantial manual efforts are needed to check and label variable values. Labyrinth by Pistoia et al. [54] is a privacy enforcement system for the mobile environment. The main idea is to introduce a man-in-the-middle proxy to capture all the data exchanged with servers. Via a value-similarity analysis, it detects whether sensitive data (such as passwords) are leaked. JTaint by Xie et al. [55] is a dynamic taint analysis system. It uses JalangiEX to discover potential privacy leaks in Chrome extensions by monitoring the process taint propagation, where excessive permissions and operation behaviors from extensions pose privacy leakage risks. Peng et al. [56] built a crawler for retrieving phishing kits and measure credential sharing in phishing sites. The crawler creates fake credentials and monitors the network traffic to explore where sensitive information is leaked. A significant drawback of the tools by Peng et al. and Xie et al. is that these tools require configuring the runtime environment and are only for specific languages. In contrast, our method can execute cross-language analyses without using the runtime environment. Meli et al. [12] built a tool for exploring secret leakages of private critical files. The device uses regular expressions to scan the candidate text. It leverages an entropy filter to evaluate the text information entropy, a word filter to check the keywords, and a pattern filter to check the secret format. The tool then determines whether the candidate text has a secret leakage. As in the case of static analysis, such a regular expression-based approach results in a high false-positive rate.

VI. Conclusions

Security credential leakage makes user accounts and sensitive data vulnerable. In this paper, we propose SEAGULL, a tool that combines static analysis and deep learning for carrying out large-scale leakage searches for credential leakages in open-source projects. SEAGULL leverages CodeQL to generate a static description for project source code that includes variable properties and data and control flows. Based on these descriptions, SEAGULL trains and uses a credential and flow context classifier to accurately detect security credential leakages in code written in different languages. The experimental evaluation results show that SEAGULL achieves high precision. The results also show that compared with three state-of-the-art tools, SEAGULL is more accurate in credential detection, especially with respect to the false-positive rate. This result is important because low false-positive rates are critical to reducing the amount of manual analysis required to verify whether leakages are present in the code.

References

[1] M. Khodaei, H. Jin, and P. Papadimitratos, "Secmace: Scalable and robust identity and credential management infrastructure in vehicular communication systems," IEEE Transactions on Intelligent Transportation Systems, vol. 19, no. 5, pp. 1430–1444, 2018.

[2] C. Meshram, R. W. Ibrahim, L. Deng, S. W. Shende, S. G. Meshram, and S. K. Barve, "A robust smart card and remote user password-based authentication protocol using extended chaotic maps under smart cities environment," Soft Computing, vol. 25, no. 15, pp. 10 037–10 051, 2021.

[3] Q. Wang and D. Wang, "Understanding failures in security proofs of multi-factor authentication for mobile devices," IEEE Transactions on Information Forensics and Security, vol. 18, pp. 597–612, 2022.

[4] D. Wang, Y. Zou, Z. Zhang, and K. Xiu, "Password guessing using random forest," in Proceedings of the 32nd USENIX Security Symposium (USENIX Security), 2023, pp. 1–18.

[5] D. Wang, X. Shan, Q. Dong, Y. Shen, and C. Jia, "No single silver bullet: Measuring the accuracy of password strength meters," in Proceedings of the 32nd USENIX Security Symposium (USENIX Security), 2023, pp. 1–28.

[6] D. Wang, Y. Zou, Q. Dong, Y. Song, and X. Huang, "How to attack and generate honeywords," in Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP). IEEE, 2022, pp. 966–983.

[7] Q. Wang, J. Li, Y. Zhang, H. Wang, Y. Hu, B. Li, and D. Gu, "Nativespeaker: Identifying crypto misuses in android native code libraries," in Proceedings the 20th International Conference on Information Security and Cryptology (ICISC). Springer, 2017, pp. 301–320.

[8] L. Piccolboni, G. Di Guglielmo, L. P. Carloni, and S. Sethumadhavan, "Crylogger: Detecting crypto misuses dynamically," in Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP). IEEE, 2021, pp. 1972–1989.

[9] S. Ma, E. Bertino, S. Nepal, J. Li, D. Ostry, R. H. Deng, and S. Jha, "Finding flaws from password authentication code in android apps," in Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS). Springer, 2019, pp. 619–637.

TABLE X: Comparison with related works.

	Analysis	Method	Target Platform	Focus
SEAGULL	Static	Machine Learning	Source Code	String Text + Flow Context
Passfinder [18]	Static	Machine Learning	Source Code	String Text + Code Block
[12]	Static	Regular Expressions	Source Code	String Text
Labyrinth [54]	Dynamic	Similarity Detection	Android & IOS	String Text
JTaint [55]	Dynamic	Taint analysis	Web Browser	Flow Propagation
[56]	Dynamic	Fake Request	Web Browser	Request Feedback

[10] J. Gao, P. Kong, L. Li, T. F. Bissyandé, and J. Klein, "Negative results on mining crypto-api usage rules in android apps," in Proceedings of the 16th International Conference on Mining Software Repositories (MSR). IEEE, 2019, pp. 388–398.

[11] S. Ma, J. Li, H. Kim, E. Bertino, S. Nepal, D. Ostry, and C. Sun, "Fine with "1234"? an analysis of sms one-time password randomness in android apps," in Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021, pp. 1671–1682.

[12] M. Meli, M. R. McNiece, and B. Reaves, "How bad can it git? characterizing secret leakage in public github repositories." in Proceedings of the 27th the Network and Distributed System Security (NDSS), 2019.

[13] "Git-secrets - prevents you from committing passwords and other sensitive information to a git repository," 2022, https://github.com/awslabs/git-secrets.

[14] D. Wang, P. Wang, D. He, and Y. Tian, "Birthday, name and bifacial-security: Understanding passwords of chinese web users," in Proceedings of the 28th USENIX Security Symposium (USENIX Security), 2019, pp. 1537–1555.

[15] "Code quality and security for java," 2022, https://github.com/SonarSource/sonar-java.

[16] "A tool designed to find common security issues in python code." 2022, https://github.com/pyCQA/bandit.

[17] Y. Zhang, S. Ma, J. Li, D. Gu, and E. Bertino, "Kingfisher: Unveiling insecurely used credentials in iot-to-mobile communications," in Proceedings the 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2022.

[18] F. Runhan, Y. Ziyan, P. Shiyan, and Z. Yuanyuan, "Automated detection of password leakage from public github repositories," in Proceedings the 44th ACM International Conference of Software Engineering (ICSE). ACM, 2022.

[19] "Detect-secrets - an aptly named module for (surprise) detecting secrets within a code base," 2022, https://github.com/Yelp/detect-secrets.

[20] "Shhgit - helps secure forward-thinking development, operations, and security teams by finding secrets across their code before it leads to a security breach," 2022, https://github.com/eth0izzle/shhgit.

[21] L. Zhou and Z. Zhang, "Trusted channels with password-based authentication and tpm-based attestation," in Proceedings of the 2010 International Conference on Communications and Mobile Computing, vol. 1. IEEE, 2010, pp. 223–227.

[22] R. Sundararajan, "State space classification of markov password–an alphanumeric password authentication scheme for secure communication in cloud computing," International Journal of Pervasive Computing and Communications, vol. 17, no. 1, pp. 121–134, 2021.

[23] "Github personal access tokens," 2022, https://github.com/settings/tokens.

[24] "Aws access tokens - use to grant your user access to ddd, change, or delete user attributes," 2022, https://docs.aws.amazon.com/cognito/latest/developerguide/amazon-cognito-user-pools-using-the-access-token.html.

[25] "Using oauth 2.0 to access google apis," 2022, https://developers.google.com/identity/protocols/oauth2.

[26] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019, pp. 164–175.

[27] M. R. Rahman, A. Rahman, and L. Williams, "Share, but be aware: Security smells in python gists," in Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2019, pp. 536–540.

[28] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in Proceedings the 17th IEEE International Conference on Machine Learning and Applications (ICMLA). IEEE, 2018, pp. 757–762.

[29] J. Wang, S. Ma, Y. Zhang, J. Li, Z. Ma, L. Mai, T. Chen, and D. Gu, "NLP-EYE: Detecting memory corruptions via Semantic-Aware memory operation function identification," in Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID), 2019, pp. 309–321.

[30] F. N. Sibai, A. Shehhi, S. Shehhi, B. Shehhi, and N. Salami, "Secure password detection with artificial neural networks," in Proceedings of the 2008 International Conference on Innovations in Information Technology (IIT). IEEE, 2008, pp. 628–632.

[31] K. H. Hong and B. M. Lee, "A deep learning-based password security evaluation model," Applied Sciences, vol. 12, no. 5, p. 2404, 2022.

[32] G. Team, "Codeql: the libraries and queries that power security researchers around the world," 2022, https://codeql.github.com/.

[33] W. B. Cavnar, J. M. Trenkle et al., "N-gram-based text categorization," in Proceedings of the 3rd Annual Symposium on Document Analysis and Information Retrieval (ASDAIR), vol. 161175. Citeseer, 1994.

[34] H.-C. Shin, H. R. Roth, M. Gao, L. Lu, Z. Xu, I. Nogues, J. Yao, D. Mollura, and R. M. Summers, "Deep convolutional neural networks for computer-aided detection: Cnn architectures, dataset characteristics and transfer learning," IEEE Transactions on Medical Imaging, vol. 35, no. 5, pp. 1285–1298, 2016.

[35] S. T. Jan, Q. Hao, T. Hu, J. Pu, S. Oswal, G. Wang, and B. Viswanath, "Throwing darts in the dark? detecting bots with limited data using neural data augmentation," in Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P). IEEE, 2020, pp. 1190–1206.

[36] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in Advances in Neural Information Processing Systems, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, Eds., vol. 27. Curran Associates, Inc., 2014.

[37] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," Advances in Neural Information Processing Systems, vol. 28, 2015.

[38] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2014, pp. 1532–1543.

[39] E. Mikalauskas, "Rockyou2021: largest password compilation of all time leaked online with 8.4 billion entries," 2021, https://github.com/ohmybahgosh/RockYou2021.txt/.

[40] "Onepassword - the easiest way to store and use strong passwords. log in to sites and fill forms securely with a single click." 2022, https://1password.com/.

[41] "Gitleaks - a sast tool for detecting and preventing hardcoded secrets like passwords, api keys, and tokens in git repos." 2022, https://github.com/zricethezav/gitleaks.

[42] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," ACM SIGOPS Operating Systems Review, vol. 35, no. 5, pp. 57–72, 2001.

[43] O. Ferschke, I. Gurevych, and M. Rittberger, "Flawfinder: A modular system for predicting quality flaws in wikipedia." in Proceedings of the 2012 Online Working Notes/Labs/Workshop (CLEF), 2012, pp. 1–10.

[44] A. Kaur and R. Nayyar, "A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code," Procedia Computer Science, vol. 171, pp. 2023–2029, 2020.

[45] B. Chinthanet, S. E. Ponta, H. Plate, A. Sabetta, R. G. Kula, T. Ishio, and K. Matsumoto, "Code-based vulnerability detection in node.js applications: How far are we?" in Proceedings of the 35th IEEE/ACM

International Conference on Automated Software Engineering (ASE), 2020, pp. 1199–1203.

[46] X. Li, L. Wang, Y. Xin, Y. Yang, and Y. Chen, "Automated vulnerability detection in source code using minimum intermediate representation learning," Applied Sciences, vol. 10, no. 5, p. 1692, 2020.

[47] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang, "Combining graph-based learning with automated data collection for code vulnerability detection," IEEE Transactions on Information Forensics and Security, vol. 16, pp. 1943–1958, 2020.

[48] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: A survey," Proceedings of the IEEE, vol. 108, no. 10, pp. 1825–1848, 2020.

[49] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," ACM SIGPLAN Notices, vol. 53, no. 4, pp. 404–419, 2018.

[50] J. Jang, A. Agrawal, and D. Brumley, "Redebug: Finding unpatched code clones in entire os distributions," in Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P). IEEE, 2012, pp. 48–62.

[51] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in Proceedings of the 2017 IEEE Symposium on Security and Privacy (S&P). IEEE, 2017, pp. 595–614.

[52] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "Vulpecker: An automated vulnerability detection system based on code similarity analysis," in Proceedings the 32nd Annual Conference on Computer Security Applications (ACSAC), 2016, pp. 201–213.

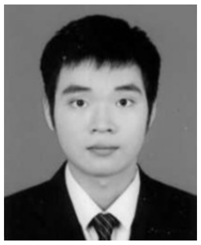
[53] N. Guo, X. Li, H. Yin, and Y. Gao, "Vulhunter: An automated vulnerability detection system based on deep learning and bytecode," in Proceedings of the 2019 International Conference on Information and Communications Security (CCS). Springer, 2019, pp. 199–218.

[54] M. Pistoia, O. Tripp, P. Centonze, and J. W. Ligman, "Labyrinth: Visually configurable data-leakage detection in mobile applications," in Proceedings of the 16th IEEE International Conference on Mobile Data Management (MDM), vol. 1. IEEE, 2015, pp. 279–286.

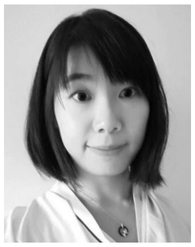
[55] M. Xie, J. Fu, J. He, C. Luo, and G. Peng, "Jtaint: Finding privacy-leakage in chrome extensions," in Proceedings of the Australasian Conference on Information Security and Privacy (ACISP). Springer, 2020, pp. 563–583.

[56] P. Peng, C. Xu, L. Quinn, H. Hu, B. Viswanath, and G. Wang, "What happens after you leak your password: Understanding credential sharing on phishing sites," in Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (AsiaCCS), 2019, pp. 181–192.

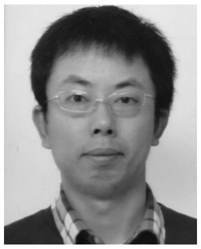
Ruidong Han received his B.Eng. degree in Computer Science from Northwest A&F University in 2018, China. He is currently a Ph.D. candidate in the School of Cyber Engineering, at Xidian University. His research interests include IoT security, trusted computing, and intelligent system security.



Gong, Huihui received the B.E. degree in Science of Intelligence with the School of Information Science and Technology, Sun Yat-sen University, Guangzhou, China, in 2018. He is currently pursuing a Ph.D. degree in Computer Science at the School of Computer Science, Faculty of Engineering, The University of Sydney, Sydney, Australia. His current research interests include adversarial machine learning and vulnerability detection on software.



Siqi Ma received her Ph.D. degree in information system from Singapore Management University in 2018. She is currently a lecturer in the School of Engineering and Information Technology at the University of New South Wales. Her research interests are mobile security, web security, and IoT security.



Juanru Li is the director with the Group of Software Security In Progress (G.O.S.S.I.P), Shanghai Jiao Tong University. His research interests cover a wide array of systems and software security problems, with an emphasis on designing practical solutions and techniques that help computer systems stay secure.

Chang Xu is ARC Future Fellow and Senior Lecturer at the School of Computer Science, University of Sydney. He received the University of Sydney Vice-Chancellor's Award for Outstanding Early Career Research. His research interests lie in machine learning algorithms and related applications in computer vision. He has published over 100 papers in prestigious journals and top-tier conferences.

Elisa Bertino (Fellow, IEEE) is a Samuel D. Conte professor of computer science with Purdue University. She serves as director of the Purdue Cyberspace Security Lab (Cyber2Slab). In her role as director of Cyber2SLab, she leads multi-disciplinary research in data security and privacy. Prior to joining Purdue in 2004, she was a professor and department head with the Department of Computer Science and Communication, University of Milan. She is a fellow member of the ACM and AAAS.

Surya Nepal is a senior principal research scientist with CSIRO's Data61. He currently leads the Distributed Systems Security Group comprising more than 30 research staff and more than 50 postgraduate students. His main research focus is in the development and implementation of technologies in the area of cybersecurity and privacy, and AI and cybersecurity. He is a member of the editorial boards of the IEEE Transactions on Service Computing, ACM Transactions on Internet Technology, IEEE Transactions on Dependable and Secure Computing.

Zhuo Ma received the Ph.D. degree in computer architecture from Xidian University, Xi'an, China, in 2010. Now, he is an associate professor with the School of Cyber Engineering, at Xidian University. His research interests include cryptography, machine learning in cyber security, and Internet of Things security.

Jianfeng Ma received a B.S. degree in computer science from Shaanxi Normal University in 1982, an M.S. degree in computer science from Xidian University in 1992, and a Ph.D. degree in computer science from Xidian University in 1995. Currently, he is a Professor at the School of Computer Science and Technology, at Xidian University. He has published over 150 journal and conference papers. His research interests include information security, cryptography, and network security.