Loose Bits Sink Gits: Unearthing Repository Secrets and Scanning Developer Trends

A technical paper prepared for presentation at SCTE TechExpo24

Golam Kayas
Technical Research and Development Engineer
Comcast Cable,
golam_kayas@gmail.com

Justin Evans
Software Development Engineer
Comcast Cables,
justin_evans@comcast.com

Jayati Dev
Privacy Engineer
Comcast Cables,
jayati_dev@comcast.com

Bahman Rashidi
Cybersecurity & Privacy Research Director
Comcast Cables,
bahman_rashidi@comcast.com

Vaibhav Garg
Executive Director, Cybersecurity Research & Public Policy
Comcast Cables,
vaibhav_garg@comcast.com

Table of Contents

List of Figures

List of Tables

1. Introduction

The rise in popularity of social programming and collaborative projects proliferates the use of code sharing platforms like GitHub and Gitlab. Developers create online projects to collaborate with peers across the globe. These projects get forked, imported and shared all around, ingraining themselves into many other codebases. With over 83 million active users in 2022, GitHub and other code sharing platforms' popularity have been rising over the years [5]. With the availability of code comes the possibility of secrets and other vulnerabilities being accidentally shared. Passwords and credentials are critical components of online security, and their unauthorized disclosure can lead to devastating consequences [31]. In a report published by Verizon in 2022 it was found that over 60% of all breaches were from stolen credentials [30]. Code-sharing platforms like GitHub, Gitlab and Bitbucket, while used for personal projects are also widely used in industry, making them a prime target for cybercriminals looking to exploit vulnerabilities in the software development process.

GitHub, Gitlab and other similar code sharing sites are web-based platforms that allow collaboration on software development projects. They provide a centralized location that allows teams to create and share code in repositories. These platforms include collaboration features like issue tracking, project management, and a built-in wiki [4, 6, 16]. GitHub's popularity stems from its ease of use and its ability to integrate with other tools and services used in software development workflows. It is by far the most popular code sharing platform among developers [8].

The booming popularity of code sharing platforms comes with increased risk of exposed secrets. A study from GitGuardian [13] illustrated that one of every ten authors expose private secrets in the source code stored in the code sharing platforms. Finding vulnerabilities and secrets has been an ongoing process for many years with many companies sponsoring bug bounty projects and creating code scanning tools [20]. As machine learning comes into prominence, advancements in computing have allowed for faster and more accurate secret detection to warn developers of potential breaches [25]. The most popular secret detection plugin is Dependabot [14] with Truffelhog [2] and Git-Secrets [9] also common in the software development industry. We also found a machine learning (ML)-based scanner called xGitGuard that utilizes a random forest model to detect secrets [7].

When looking into the code secrets being leaked onto the above-mentioned platforms there are a few metrics we investigated based on industry standards and frameworks such as the Security Scorecard [23]. For GitHub we queried repositories based on number of stars, number of forks, and project age - comparing the number of secrets found with project age and external engagement (forks and stars).

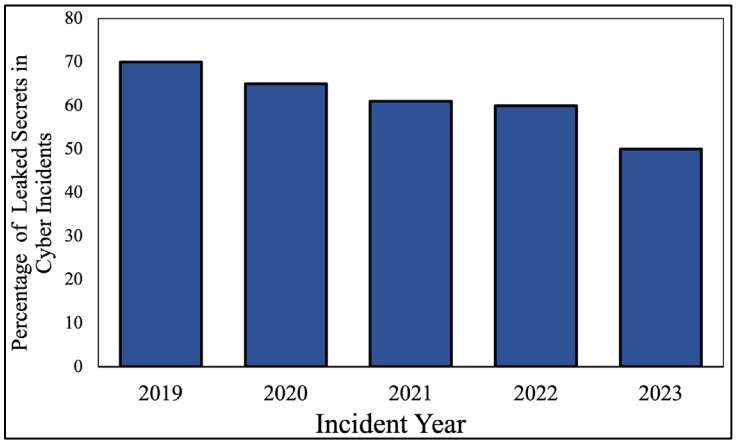

Figure 1 - The percentage of leaked secrets in cyber incidents over the years.

Specifically, we answer three research questions:

RQ1: **How does the underlying architecture of industry grade GitHub scanning tools affect accuracy?** To answer this question, we consider three industry-used secrets scanning tools that are available on GitHub. All three of these tools operate on different mechanisms of secret detection, and our goal was to check which mechanism has a better accuracy in detecting secrets. We found that the ML-based tool had a higher number of true positives and less false negatives than regex-based or entropy-based tools.

RQ2: **What are some patterns of behavior for GitHub secrets?** Using the ML-based tool, xGitGuard, we scanned 1468 repositories for secrets to determine the prevalence of secrets in these repositories. Out of 1468, 596 repositories contained secrets such as Application Programming Interface (API) tokens, keys, etc. We also found there were patterns in the code in terms of how secrets (plaintext, encrypted, hardcoded, comments, etc.) were embedded and their location in code.

RQ3: **What metrics for a repository influence the presence of secrets?** To answer this, we look at the number of secrets in a repository and change over time based on engagement. We consider GitHub repositories for the pattern of secrets and engagement over time.

Through these three experiments, we show that there are patterns in open-source repositories in terms of secret management, which developers can identify and mitigate. Furthermore, we also show that the number of secrets increases with project age and decreases with engagement, respectively. This corroborates Figure 1 from Verizon's annual reports that the percentage of breaches due to leaked secrets has steadily decreased. This could hint that newer code bases are more secure while leaked credentials remain the top contributor to breaches. There are many ways that developers can keep secrets outside of their repos using these code sharing platforms, from using local environment variables to secret vaults. We also discussed the different tactics we observed developers taking and how they correlated to the age and engagement of the repositories [22, 27].

In the following section, we discuss the background literature that inspired this study. We then talk about the experiment methods and process, followed by a discussion of the results.

2. Background & Related Work

In this section of the paper, we aim to provide an overview of the existing research and practices surrounding the detection and mitigation of secrets in code sharing platforms. To analyze the usage of GitHub in the software engineer development life cycle, several studies have been conducted [4, 6, 16]. Other studies have been conducted pertaining to the creation and testing of tools to assist GitHub Users [10, 15]. Google maintains BigQuery [12] which is a snapshot of open-source repositories open to the research community. These repositories allow for larger scale research without having to worry about GitHub's API limits.

Despite the popularity and importance of code sharing platforms, security-sensitive information is often leaked. Even a well-crafted web search can reveal passwords and secret keys [21] to an adversary. Besides, different modern resources such as Docker images or VMs in cloud platforms can all expose secrets of the publishers, customers, and managing counterparts [34, 3]. These resources are often public and can cause severe vulnerabilities [26]. Secrets exposure on GitHub is a known issue by both the security community and developers themselves [19, 11, 22]. One of the primary causes of secret exposure is inadvertent mistakes or oversights made by developers in the development process [36, 22]. In the rush to meet deadlines or due to a lack of awareness, developers may accidentally include sensitive information in their code or configuration files [19, 18].

Exploration of secrets in public code repository is not new. Researchers have used several predefined regular expressions to find secrets with generic formats from the source codes [17]. Machine generated secrets, like API keys, have high randomness and entropy. Specifically, one can use the Shannon entropy calculation to identify high entropy secrets. Considering the patterns and high entropy of these secrets, many tools are developed using regular expressions, entropy calculations or a combination of both to find out secrets in public code repositories.

There are many different tools to scan for leaked secrets in online repositories. TruffleHog [2], a popular tool, searches throughout the commit history and git branches using predefined customizable regular expressions. This tool considers high entropy strings larger than 20 characters. Another tool, Repo Supervisor [1] detects secrets in the pull requests, finding high entropy string values as potential secrets. GitSecrets[9], created by Amazon, looks for secrets while the commit happens. It uses user-defined regular expressions to detect secrets.

In Meli et al's [22] paper they look to answer the question: How prevalent are leaked secrets in code uploaded to GitHub? This work used regex along with entropy, dictionary, patterns, and words filters to find the secrets from their dataset. They further investigate some of the possible root causes for these secrets and how long it takes for developers from leak till the time they fix it. This study centers its attention on recently committed code, alongside a captured snapshot from BigQuery. They found over 200,000 unique leaked secrets in their 6 months of scanning.

Krause et al. [19] surveyed 109 developers to gain insight on how secrets were being leaked and how secret leakage is perceived as a problem by developers. From their surveys they selected 14 developers to conduct in-depth interviews investigating secret remediation techniques and approaches. Based on developer responses they found that 30% of developers were aware of secrets that had leaked in their code. They found that in terms of mitigation techniques most developers utilized GitHub's secret scanner and added sensitive files to the *gitignore*.

Sinha et al. [28] used a sample set of 84 repositories to perform a similar study. They used 7 different regular expressions focusing on Amazon Web Services (AWS) API keys and leveraged entropy filters and a password strength estimator. This work also uses light static code analysis to increase the accuracy of the search.

A big drawback of the aforementioned secret scanners is they generate a high number of false positives. Additionally, all these works are specifically designed to look for private keys such as Rivest, Shamir, Adleman (RSA) encryption keys and API keys (such as AWS access tokens). But the regex and entropy calculation-based methods do not work well with other types of secrets such as generic textual passwords. Saha et al. [24] proposes a regular expression-based method which uses machine learning to reduce the number of false positives. The proposed method can detect both secret keys and textual passwords from the source files. The authors chose 24 relevant features to train their aggregated model, combining different types of classifiers. The users can also pick the trade between false positive and false negative by training the model using a precision-recall curve. Feng et al. [11] developed PassFinder, a deep learning based textual password detection model that investigated password leakage in GitHub with continuous scanning over 75 days. In their study they found over 60,000 repositories in which passwords were leaked. This work benchmarks PassFinder against Regex-based scanning to show the lower false positive and higher detection rates. But their model is only focused on finding generic passwords and cannot detect the secret keys. SecretHunter [35] proposes a reinforcement learning based model that outperforms the regex-based model in secret detection. The authors also improved the bandwidth usages by obtaining the metadata of the source files and using the metadata to do some filtering, such as filename filtering and deduplication, before downloading the source file. xGitGuard [7] can scan GitHub with simultaneous queries running together, and it uses file hashing to avoid duplicate file scanning. This tool uses two sets

of keywords to filter only the targeted files, not the entire repository. Besides, xGitGuard utilizes natural language processing and ML techniques to extract secrets from the targeted source files.

To try and minimize the risk of secrets exposure, many different applications have been developed targeting different vectors of secret exposure. These solutions range from active scanners and secrets vaults to best practice guidelines. While a combination of solutions may work best with the growing number of 3rd party add-ons it can be difficult to integrate or encourage adoption.

3. Study Design

In this section, we discuss our approach to selecting the data and identifying secrets. A secret, in our context, refers to any sensitive access information that, if exposed, could pose significant risks to the developers and the projects they are working on. For these experiments, we have scoped secrets only to API tokens and passwords for simplicity and ease of analysis with regex-based scanners. Our methods which we will briefly introduce here will be discussed in more detail in their pertaining experiment section.

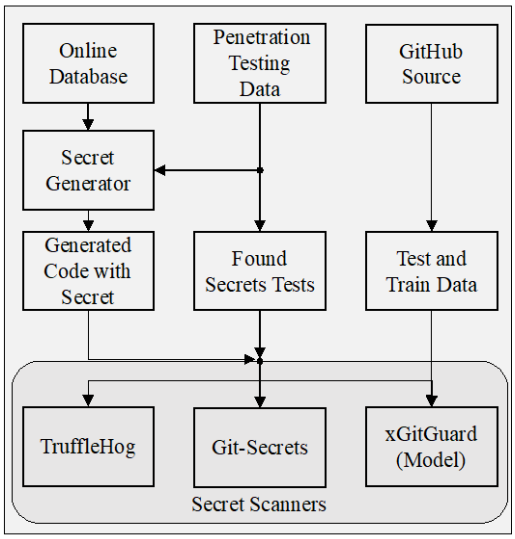

Figure 2 - Data Collection and Secret Detection Overview.

3.1. Data Collection

To gather the dataset required for our experiments, we explored well-known platforms like GitHub and GitLab. Our primary objective was to acquire a large enough sample size that would produce a relatively balanced mixture of secrets. We had strict control over our dataset maintaining knowledge of the secrets the scanners would be scanning for.

For procuring the secrets, we conducted an in-depth search, analyzing API key generators, public GitHub and GitLab repositories and databases that contained API regular expressions. This process ensured that we had a wide range of secrets, some of which are commonly employed in real-world applications.

3.2. Sampling and Data Refinement

To establish a foundation for our experiments, we conducted a sampling of the vast GitHub and GitLab repositories. These platforms house a staggering number of codebases, exceeding 200 million, making it

essential to refine our selection process. To achieve this, we employed specific queries to search repositories created till April 2022 (so that there is at least a year's gap between our data collection and repository creation) with over at least 50 forks.

In April 2023, we executed these queries on GitHub, yielding a collection of repositories suited for our scanning and analysis purposes. This process ensured that we would have the same set of public repositories across Experiment 2 and 3. In addition to this, we also curated a set of secrets for Experiment 1 to provide a controlled list of secrets that were known and check for detection accuracy.

The collection of curated secrets required that we properly de-identify them while ensuring they were true secrets. We tapped into sources such as data from penetration tests, which offered valuable insights into actual secrets that developers have discovered. This data was gathered with the help of industry experts in penetration testing and threat hunting. Additionally, we extracted relevant data from leaked websites like "*Have I Been Pwned*" and various databases that contained instances of compromised secrets. These methods provided samples to create a database to test scanner accuracy. These secrets were masked when extracting output from the scanner. Furthermore, we ensure that these secrets are discarded after the end of the study. These curated secrets were either put directly into a file to be used to test the scanners while the other secrets were used to find patterns to generate like secrets to put into a blank repository to be scanned.

3.3. Scoping Secrets

As previously mentioned, when scanning public repositories, we look for secrets. For this paper's purposes, there are two main secrets: passwords (credentials) and API token types (keys). Passwords are confidential phrases generated by the user to access various systems such as user accounts and databases. Keys, which include API keys and tokens, are often used in request headers to authenticate third party platforms. Other secrets fitting into this category include Secure Socket Shell (SSH) keys and RSA tokens used for secure communication and encrypting/decrypting data.

Secrets can often be found in many different locations within the repositories. These leaks might appear as hardcoded strings, constants, configuration files, or environment variables. In the wrong hands, leaked secrets can cause security breaches (Verizon found that 60% of breaches were related to leaked credentials [30]), ranging from unauthorized access to potential exposure of critical user information. These pose not only a significant risk to the application's security but also to the reputation of the developers and organizations they represent. The aftermath of such a breach can led to financial losses, legal complications, and eroded user trust. As such, discovery, and careful management of these secrets within the codebase is paramount to safeguarding the integrity and security of the entire software ecosystem.

We investigate these specific secret types because the types of secrets have different vectors in which they can be exploited. Passwords not only can compromise the system in which they are designed, but depending on developer tendencies can open access to other platforms. When leaked, API keys and tokens can give malicious actors access to their associated services and expose companies to further data breaches. To limit the scope of this study, we do not consider other secret types which can be a part of future work.

3.4. Scanner Choice

In our research, we identified three prominent scanners that have garnered significant traction within industry and the open-source community. These scanners were identified based on industry reports as well as research into the open-source scanner market. We also chose these scanners because each one (i)

utilizes a different underlying technology, (ii) is available open-source and can be customized, (iii) is an industry-grade scanner that is adopted by the industry and which is scalable for many repositories, and (iv) is actively maintained by a team of experts. We excluded scanners which were not open-source (or a paid version existed) and those which were developed in academia as research proof-of-concept since they did not meet our selection criteria (e.g., active maintenance).

TruffleHog was the first secret detection tool in consideration. This scanner has earned notable recognition, not only for its usage across various research domains, but also as a standard in industry practices. TruffleHog operates on the foundation of entropy calculations with regular expressions and user-defined rules, ensuring that its scope can be tailored to the unique requirements of different applications. However, we noticed that the initial setup of TruffleHog can be intricate, especially for users unfamiliar with the intricacies of regular expressions and rule configuration. This potential complexity can sometimes lead to suboptimal scanning outcomes.

Git-Secrets was the second detection tool in our selection. This scanner's prominence in industry circles solidifies its credibility and influence. Git-Secrets operates similar to TruffleHog, utilizing regular expressions but without entropy and user-defined rules to unveil secrets within code repositories. Its wide recognition and usage make it a desirable choice for our experiments. Like TruffleHog, Git-Secrets may present challenges during the initial setup phase.

xGitGuard, our third chosen scanner, utilized key words and a Random Forest ML algorithm. This tool combined traditional regular expression matching and ML techniques. While the inclusion of ML augments its efficacy, it also introduces a higher degree of complexity. Users will need to familiarize themselves with its intricacies to harness its full potential. We found this was offset by a well written user guide that walks through each of the steps.

Note that we chose only three scanners to have a representative of each *type* of scanner - regex with entropy, regex only, and ML-based respectively. Post selection of these three tools, we set up three experiments as noted below.

Experiment 1: This experiment focused on how well the three chosen scanners performed on our data sets against each other. These datasets had a set number of secrets allowing us to gauge their accuracy, precision, and recall. Each one of the scanners, after properly being set up, scanned the same repositories' results and looked over to gauge their performance. This experiment also assisted in our selection of scanner for Experiment 2.

Experiment 2: After selecting a scanner, we then prepared it to scan the repositories gathered from our query scans. This experiment was designed to answer the question: How prevalent secrets across Git repositories are (both GitHub and GitLab) and are there variables that can help to predict if a repository has secrets? The result of these scans tells us how many secrets we would find in each repository as well as the location of these secrets in the repository that they were found in. Furthermore, we looked at the different strategies developers took to mask secrets in their repositories.

Experiment 3: In our final experiment we investigated the influence of project age and external engagement (measured in terms of number of forks and stars) on the detection of secrets across repositories.

4. Experimental Setup and Implementation

In this section we will go into more detail on how each experiment was set up. We will also cover the challenges we faced and the results from each experiment. To answer each of the three research questions,

we set up three main experiments. Each one of these experiments tackles a different aspect of secret leakage in code repositories. Throughout our three experiments, we leveraged the results and insights obtained from each preceding experiment to inform and guide subsequent experiments. Our initial experiment aimed to evaluate the accuracy and precision of three opensource industry-grade scanners. This experiment and the data collected helped to shape the secret scanner we chose. We utilized the winning scanner from Experiment 1 to conduct Experiment 2 and Experiment 3.

4.1. Experiment 1: Choosing a Scanner

This controlled experiment was conducted to compare three open-source industry-grade scanners. We gathered three leading scanners xGitGuard, Git-Secrets, and TruffleHog. We reviewed the source code and setup instructions for all three scanners. Git-Secrets and TruffleHog were determined to be regex-based detectors. For these regular expressions, patterns were set up in their respective configuration file used in the test when a regular expression was readily available. For setting up xGitGuard, which was an ML-based detector, we found that it was trained using the dataset mentioned in Section 3.1.

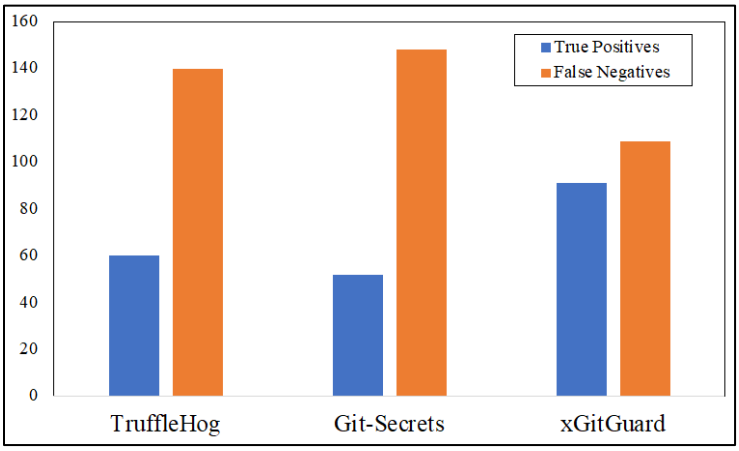

Figure 3 - The performance of the scanner on curated set of leaked secrets.

Following scanner setup, we established five repositories that contained the secrets created using API generators and patterns mentioned before. These contained dummy code with two written in C#, two in Python, and one in React Native. The secrets were deliberately placed within the code repositories

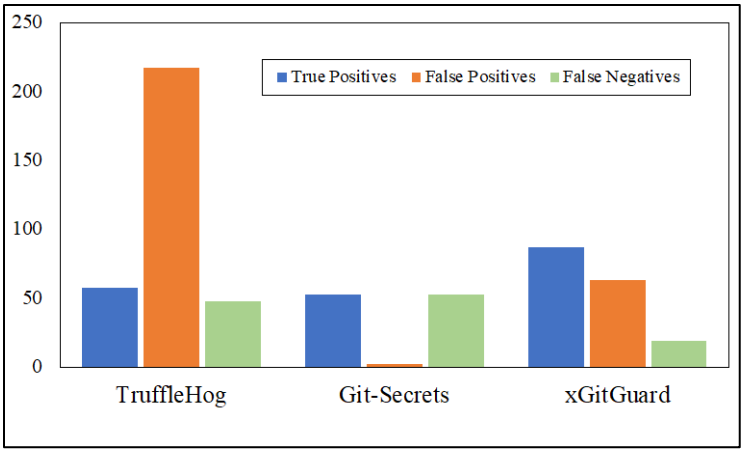

Figure 4 - The performance of the scanners on the repositories with generated secrets embedded within.

mimicking different scenarios in which code could be leaked. These scenarios were based on an initial data analysis of GitHub in which we found secrets hardcoded, in config files, and in comments. We chose 5 repositories to embed the secrets to test scanners effectiveness in scanning multiple repos at once. For the other dataset containing found secrets, all the secrets were compiled into a single text file, with each line representing a different secret embedded in code.

Each one of the scanners took slightly different methods to set up. For xGitGuard we pulled the open-source repository from GitHub and installed the necessary requirements on the computer. We also added the functionality to point xGitGuard to certain repositories rather than utilizing GitHub search. The modification involved reading the code bases and passing them through the scanner, rather than directly passing the code from the GitHub calls. We then gathered a labeled dataset of over 4000 secrets to train xGitGuard which is provided by xGitGuard and was found to be easy to use.

TruffleHog was extremely easy to set up with options available from brew and docker. We compiled from its source and decided to not use only verified keywords in our detections. Furthermore, in our tests we found that TruffleHog performed better utilizing the entropy filter and adding a few of the custom regular expressions that the tool allows. Once this setup was completed, we tested it against the other two scanners.

Git-Secrets, like TruffleHog, is available to be installed via *HomeBrew* (brew). Once it was installed via brew (or GitHub directly) as we did, the user then can easily add hooks to their local repositories. Git-Secrets also allows third-party configurations and add-on scanners which can be set up as an AWS profile. To improve performance, we configured GitSecrets so that the AWS provider as well as pre-configured regex patterns can both be detected.

Once the setup was completed, the scans were performed on the dataset, starting with the code repositories that had secrets manually embedded. Our initial findings indicated that all three of the scanners did well with the APIs with xGitGuard only slightly ahead. When it came to passwords all three struggled but only xGitGuard was able to match passwords reliably without overloading false positives.

Table 1 - API token types (non-exhaustive) used in the curated set of repositories.

Token Type	Number of Tokens Generated
AWS Tokens	20
Google Tokens	20
Artifactory Tokens	12
Docker Tokens	13
Heroku UUID	5
Okta Tokens	4
Postman API	3

Table 2 - Percentage of each secret type (credentials and keys) correctly detected by each scanner.

	Passwords	API Types (Public, Curated)
Git-Secrets	0%	70.2%, 87.0%
TruffleHog	0.59%	76.0%, 98.0%
xGitGuard	30.17%	97.0%, 100.0%

The API types that we used in testing are detailed in Table 1. We chose these due to their importance in industry and the initial repository search we conducted on GitHub to find the common ones, especially AWS and Google tokens. Some of the other common technologies in our search included Docker, Artifactory, Postman, Okta and Heroku. Please note that this is not an exhaustive list - future work may include other API types. We specifically chose this list also because it was easier to find generators for these seven types.

By conducting this experiment in a controlled environment, we aimed to assess the precision and effectiveness of these three leading open-source scanners. This enabled us to evaluate if the secret determined was accurate or not, since they were curated. Figure 3 shows the results obtained from the curated list of secrets. As we see in the figure, in terms of true positives (number of secrets identified correctly) and false negatives (number of secrets marked wrong as non-secrets), ML-based xGitGuard was followed by TruffleHog and Git-Secrets. We conducted the same analysis on a larger set of secrets embedded into code - the results obtained are shown in Figure 4. GitSecrets outperformed TruffleHog and xGitGuard in terms of low false positives (incorrectly identifying non-secrets as secrets). However, in terms of false negatives and true positives, the trend was like the curated list of secrets.

For our sample dataset, Git-Secrets performed less accurately than xGitGuard and TruffleHog. When running against the secrets embedded in repositories it was able to detect 70% of the API types and got 87% of the API types in the curated dataset. For passwords, Git-Secrets had too many false positives to prove valuable. Git-Secrets' password detection regular expressions incorporated common password requirements such as having both numbers and letters, having at least 8 characters and using special characters. Perhaps due to this restricted set of regular expressions added, Git-Secrets was unable to detect any of the passwords.

TruffleHog was second place in terms of detection, as shown in Table 2. In detection of API keys TruffleHog performed better than Git-Secrets, even though there were overlapping API types that both could not detect. However, TruffleHog got 76% of the API types in public repositories and 98% of API types in our curated repository set respectively. The common API types that both TruffleHog and Git-Secrets struggled with were Docker tokens and Heroku universally unique identifiers (UUID) that were generated using their corresponding token generators. This was possibly because since both scanners were regular expressions based, they were unable to detect the change in format for these API token types (Heroku and Docker use dashes and underscores in their respective formats).

In terms of password detections, TruffleHog detected about 0.59% of passwords, only slightly better than Git-Secrets, since it also relied on regular expressions. The slight improvement in detection was perhaps due to an entropy filter. However, it was not able to adequately differentiate the passwords in the embedded code for the public repositories. For the curated dataset, it detected one password successfully. In terms of usability, TruffleHog logs were the easiest to read.

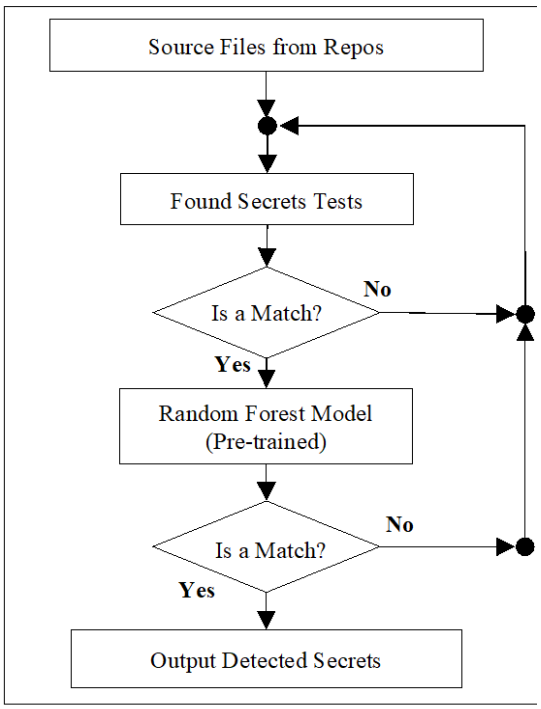

Figure 5 - xGitGuard architecture.

For our dataset, xGitGuard outperformed both Git-Secrets and TruffleHog. In terms of API types, xGitGuard was able to find the largest proportion - 97% of API types in public repositories and 100% of API types in the curated repository set. For the embedded secrets, xGitGuard had 63 false positives. Since ML-based xGitGuard was better in terms of detecting both passwords and secrets, we used xGitGuard in our subsequent experiments for scanning and evaluating repositories. We performed a deep dive into the architecture of xGitGuard to show how it works (Figure 5) using an underlying Random Classifier model that is trained on a pre-determined dataset provided by xGitGuard.

4.2. Experiment 2: Secrets in Public Repositories

In this experiment, the goal was to understand patterns of secrets in open-source repositories. We investigated (a) the prevalence of secrets on open-source repositories like GitHub and GitLab and (b) the location of these secrets.

For a scanned sample size large enough that it would represent the total number of repositories present, we scanned 1465 repositories (987 for GitHub and 478 for GitLab). 306 GitHub (31%) and 290 GitLab (60.67%) repositories returned secrets, showing that prevalence of secrets in GitLab was more than that in GitHub. The breakdown of the total number of secrets is given in Table 3. In the 306 GitHub repositories which contained secrets, there were 4116 secrets. In the 290 GitLab repositories, there were 4093 secrets. For the GitHub sample, the number of credentials and the number of keys were skewed towards keys. For the GitLab sample, 38.48% of secrets were credentials and 61.52% were keys, indicating that more keys instead of credentials are typically leaked.

Table 3 - Total number of secrets (credentials and keys) found in GitHub and GitLab

	Secret Type	Value	Percent
GitHub	Credentials	1378	33.48%
(Total Secrets = 4116)	Keys	2738	66.52%
GitLab	Credentials	1575	38.48%
(Total Secrets = 4093)	Keys	2518	61.52%

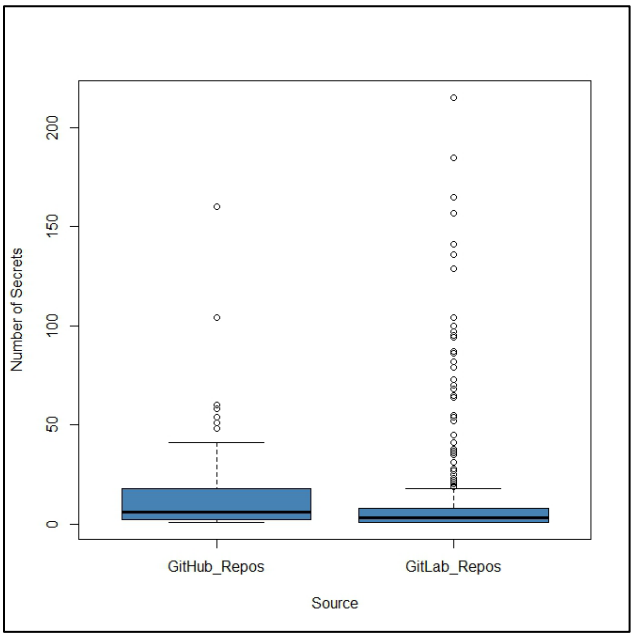

Figure 6 - Distribution of secrets for open-source repositories on GitHub (4116 total secrets) and GitLab (4093 total secrets).

The median number of secrets per repository for GitHub was 13, with 160 being the highest number of secrets present in any of the repositories. For GitLab, the median number of secrets was 3, with 215 being

the highest number of secrets present in a repository. Figure 6 shows the distribution of secrets in GitHub and GitLab, where we can see that median number of secrets in GitHub is higher than that in GitLab.

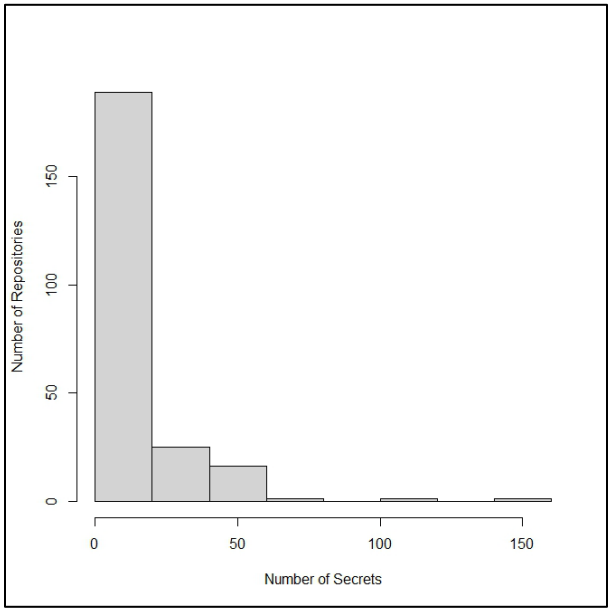

Figure 7 - The distribution of validated secrets for open-source repositories on GitHub

Once the prevalence of secrets across these public repositories was determined, we aimed to answer where were these secrets placed in code. To answer this, we investigate the origin of secrets in our sample of 5554 secrets (1461 on GitHub scans 1-4 and 4093 on GitLab) and performed a random 10% sampling. This gave us a large enough sample set that was representative of our dataset. We first went through all the secrets from the sample to make sure there were no false positives. This was validated by four other researchers to make sure the presence of secrets was identified accurately. 42.1% (234) of the sample were secrets that were correctly identified. Please refer to Figure 7 to see the data.

When looking at the secrets that were leaked, there were a few different sources of leakage that we were able to identify. Many secrets were directly hardcoded into the repository. Hardcoded secrets accounted for 64% of the leaked secrets. Hardcoded secrets have a few different ways to persist into code. Often, they are put in where developers need to authenticate with an API and are trying to get the code to run. Such actions often occur during pre-production or testing phases, and unfortunately, these hardcoded secrets tend to persist in the codebase without being removed or are recognized as nonessential by developers. The main place we saw developers hardcoding secrets was when developers stored the sensitive information in variables for later use. This could account for our earlier findings in which most repositories only had one secret found during scanning. Fortunately, detecting and rectifying these single instances of leakage becomes comparatively easier since there is only one code line that needs to be altered.

The next category of secrets were passwords or API keys that were found in comments. These accounted for 16% of the secrets found. Key vectors of leakage appear to be through comments and configuration files. Our investigation revealed that developers sometimes inadvertently expose sensitive data through comments, often meant as notes to colleagues but mistakenly commit to the repository alongside the code. Similarly, configuration files pose a significant risk when storing passwords and API keys, as keeping them outside the repository prevents direct exposure. However, a potential drawback arises when a developer mistakenly commits the configuration file, inadvertently exposing all the stored secrets.

While 234 secrets were correctly identified, we wanted to see how developers managed the secrets that were identified as false positives. We investigated the code for all false positives for strategies taken by developers to mask these secrets. Some of these methods were:

- Encrypting token and keys into variables (especially for RSA keys): 30%
- Storing secrets in a vault: 21%
- Environment variables to be stored on user end: 13%
- Global parameter calls that a user enters at execution: 19%
- Using configuration files or function calls for authentication: 17%

We can see that secrets leakage is pervasive across both GitHub and GitLab. We did see some successful remediations from developers, but we had an alarming number of secrets that were directly hardcoded. While we did notice some differences in the GitHub and GitLab secrets distribution, we did not notice much difference in the type of secrets found or in how they were leaked in the two platforms.

4.3. Experiment 3: Repository Metrics v.Secrets

For Experiment 3, we wanted to see if repository metrics influenced the presence of secrets. For engagement metrics, we considered those recommended in the Open-Source Security Foundation's Security Scorecard [1] - number of watchers, number of contributors, project age, recent releases, months since last update, number of forks, and number of stars. As discussed in Section 2, we considered only project age, number of forks, and number of stars since these were better metrics of external engagement. We considered 234 repositories from the past twenty years, collecting repository names, project age, number of forks, and number of stars. From the 233 repositories, we removed two outliers' repositories which were the only ones with greater than 100 secrets.

4.3.1. Correlation of Secrets and Repository Age

To see how the number of secrets changed with project age, we first divided project age into four categories based on the quartile values. This gave us four categories for project age range: (i) 0-2 years, (ii) 3-6 years, (iii) 7-12 years, and (iv) 13-22 years with equal number of repositories for each category. We then looked at the total number of secrets per project age category and the number of secrets per repository (see Figure 8).

As seen in Figure 8, the number of secrets increases with the project age category. Thus, older repositories have more secrets than newer ones. This is expected, since code bases usually increase with time, and hence might contain a greater number of secrets. The other potential reason could be that newer repositories have better security practices to mask secrets. Since this had four groups and was non-parametric, we performed a Kruskall-Wallis test (coefficient=10.322, p-value =

[1] https://github.com/ossf/scorecard

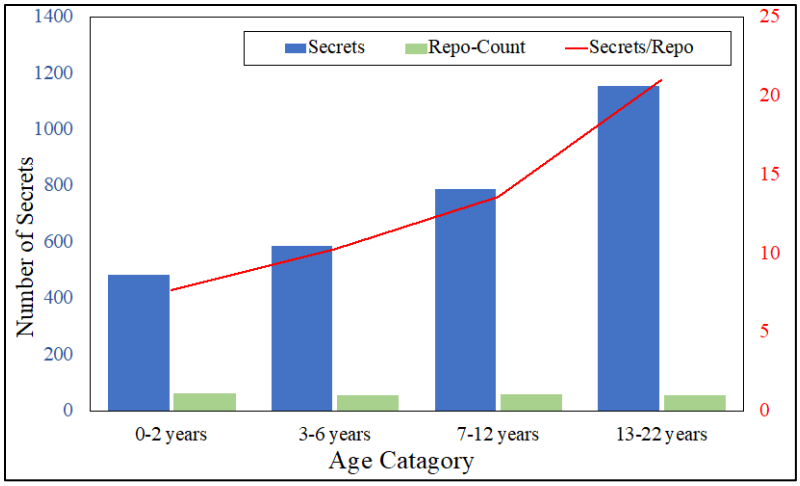

Figure 8 - Number of repositories and secrets per project age category.

0.01602) to check if the differences between the different age groups were significant. Since the p-value in less than 0.05, the differences between groups were significant, especially between the lower two (06 years) and the upper two (7-22 years) respectively (p-value = 0.021 after Bonferroni correction).

Similarly, to see how the number of secrets changed with the number of stars, we divided the number of stars into four categories based on quartile values. This gave us four categories: (i) 01300 stars, (ii) 1301-8300 stars, (iii) 8301-17600 stars, and (iv) 17600-87200 stars with equal number of repositories for each category. We then looked at the total number of secrets per star category and the number of secrets per repository (see Figure 9).

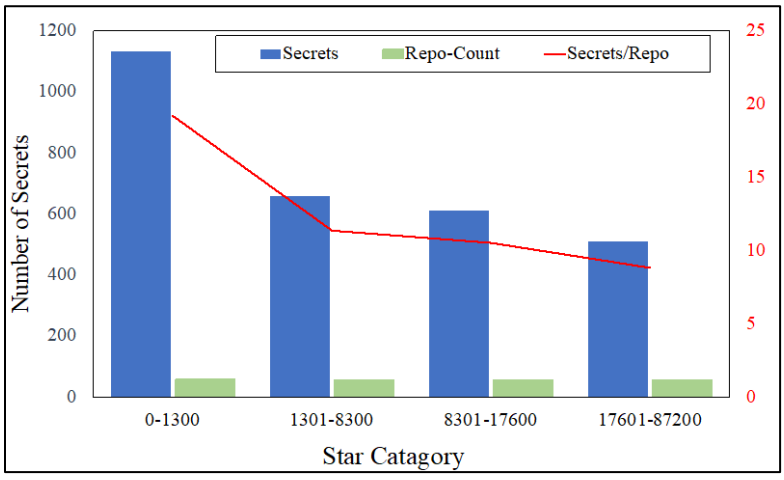

Figure 9 - Number of repositories and secrets versus the number of star categories.

As seen in Figure 9, the number of secrets decreases with the number of stars. Thus, repositories with a higher number of stars (external engagement shown in the form of bookmarks) have fewer secrets.

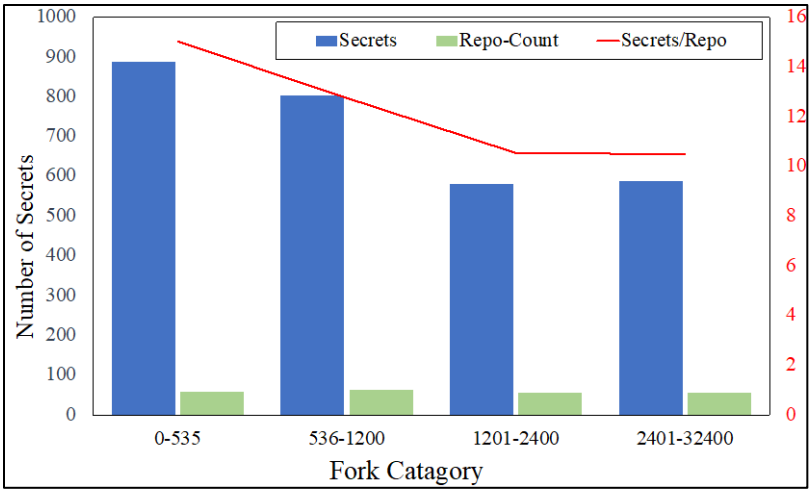

Figure 10 - Number of repositories and secrets versus the number of fork categories.

One potential reason for this could be that repositories with greater number of stars have greater engagement and review by GitHub members who are not part of the internal contribution team. Since this had four groups and was non-parametric, we performed a Kruskall-Wallis test (coefficient=9.5421, p-value = 0.02289) to check if the differences between the different age groups were significant. Since the p-value is less than 0.05, the differences between groups were significant, especially between the lower two (06 years) and the upper two (7-22 years) respectively (p-value = 0.032 after Bonferroni correction).

Finally, to see how the number of secrets changed with the number of forks, we divided the number of forks into four categories based on quartile values. This gave us four categories: (i) 0-535 forks, (ii) 536-1200 forks, (iii) 1201-2400 forks, and (iv) 240132401 forks with equal number of repositories for each category. We then looked at the total number of secrets per star category and the number of secrets per repository (see Figure 10).

As seen in Figure 10, the number of secrets decreases with the number of forks, like the number of stars. Thus, repositories with higher number of forks (external engagement shown in the form use of code base for individual usage) have fewer secrets. One potential reason for this could be that repositories with greater number of forks are reviewed more often by developers who use this code for their own application. Since this had four groups and was non-parametric, we performed a Kruskall-Wallis test (coefficient=6.0134, pvalue = 0.111) to check if the differences between the different age groups were significant. Even though there are differences in number of secrets, the p-value is greater than 0.05, indicating that the differences between groups are not significant. Thus, for our sample size, more developers using a code base did not automatically indicate backward contribution and reduction in number of secrets. This shows that potential scheduled secret reviews are needed for open-source code repositories to ensure that secrets are protected from leakage.

Our scans unveiled a concerning reality, secrets leakage is a pervasive issue that can afflict any repository. As we analyzed the data, a trend emerged, indicating that the age of a repository plays a role in its vulnerability to secret leaks (Figure 8). Older repositories were more susceptible, potentially attributed to a less security-focused culture prevalent during their development. However, this also potentially serves as a testament to the progress we have made over the years, as newer repositories demonstrated better security practices, albeit still suffering from secret leakage. Despite the progress made in the newer repos there is still a long way to go to keep secrets out of developer repositories.

While our study focused on metrics like the number of forks and stars as measures of engagement, there remain promising avenues for future investigations. For instance, examining the commit history of external contributors could shed new light on this aspect. Regrettably, due to time constraints, delving into these additional aspects fell beyond the scope of our current study.

These findings suggest that repositories of all engagement levels in our sample are not immune to secrets leakage. It underscores the importance of scrutinizing all repositories, regardless of their perceived popularity. Considering these revelations, there is an urgent need to emphasize proper security education to developers, ensuring they are equipped with the knowledge and practices to safeguard against secret leaks effectively. Only through a comprehensive and vigilant approach can we enhance the security posture of repositories and bolster the protection of sensitive information within the development ecosystem.

5. Discussion

Through our three experiments, we answered our corresponding research questions. For the first experiment, we found that the order of increasing performance for secret detection scanners was regex-based (Git-Secrets), regex and entropy-based (TruffleHog), and ML-based (xGitGuard). When testing the three scanners we found that all of them were challenged with detecting the passwords, especially when they were curated (emulated real-world secrets found during penetration testing and online databases). While passphrases can be very secure the use of single word passwords with numbers can make it hard for vulnerability scanners to tell them apart from variables or other pieces of code. Having more secure passwords would not only allow vulnerability scanners to better prevent passwords from leaking it would also help stem the possibility of a breach due to brute force attacks. In this regard having more open vulnerability scanners and more options to integrate into repositories would also help to drive developer adoption.

For our second experiment, we detected the prevalence of secrets in GitHub and GitLab repositories and found that many of them contained secrets. These were located both in the code as well as in the comments made during commits. For those that were incorrectly identified as secrets but were not, we saw several patterns of behavior - diverse ways in which developers protected secrets. This ranged from encrypting tokens to storing them in vaults. Most of the secrets that we found in our sample were API tokens and keys. Developers should be encouraged to avoid adding sensitive information as comments and instead utilize dedicated communication channels for sharing notes and insights among the team. Since so many of the secrets we detected were either hard-coded or in comments developer education such as integrating security checks into Integrated Development Environments (IDE)s and pop-up messages could prove to be beneficial. This we believe would also be a good future study to see how pop-up messages and developer education in proper cybersecurity practices diminishes the number of secrets leaked into repositories.

With so many different third-party applications, managing API keys can be tedious. A strategy we recommend for developers is to pull the secret from local environment variables. This approach offers the advantage of seamless integration for anyone who forks and utilizes the codebase, requiring them to only add the needed system environment variables. A popular option we already saw being utilized is the application of vaults and secret files. Vaults offer a heightened level of security for managing and protecting sensitive information. By centralizing secrets in a secure vault, organizations can control access and limit exposure, ensuring that only authorized personnel can access the secrets when necessary. GitHub has its own vault built in making it quite easy to integrate, but even in instances where developers don’t use GitHub Secrets there are many other vaults that one can integrate. We also observed instances where applications prompted users to manually enter API keys during runtime. While this approach

effectively avoids hardcoding secrets in the code, it may lead to potential issues such as hardcoded or bypassed entries when users opt to avoid repeatedly entering API keys. The choice of security measures should be tailored to the specific use case and organizational requirements while respecting proper coding practices and user orientated design. Selecting the appropriate security tools and methods ensures that sensitive information remains protected while providing a seamless experience for developers and end-users alike.

When conducting our third experiment, we did see a decrease in the number of secrets in the newer repositories. This could be either due to smaller code bases or due to the growing trend for better security in the wake of several major breaches. While less secrets are good, many of the secrets we detected could have been caught by integrating Continuous Integration and Continuous Deployment (CI/CD) pipelines with automated security checks. Such pipelines can detect potential leaks during the development process, preventing them from being introduced into the repository in the first place. For those secrets that still make it past scanners, regular security audits and code reviews should be conducted to identify and remediate any hidden vulnerabilities, bolstering the overall security resilience of the codebase and ensuring that sensitive information remains well-protected. To ensure better security practices, it is essential to address these issues systematically, providing proper education and awareness to developers regarding secure coding practices. Since we also saw that repositories with greater engagement had lesser number of secrets, another way of handling secrets which already make their way into code is to have greater engagement with the community so that these secrets are detected early and handled accordingly.

While our study shed light on what type of features are more prominent in repos that contain secrets, we were limited to the repositories that we scanned. One key limitation we faced was the GitHub API restriction limiting us to 5000 searches an hour. Because of this limitation we had to build in waiting periods in the code, significantly slowing down the scanning. GitLab API was also quite restricting, requiring us to have a premium subscription to the site to get the metadata and scanning parameters that we required. While a comprehensive scan of these online repositories was out of our scope, we would like to see other future studies compare how different online code repositories differ in how they manage secrets. We also limited our engagement metrics to looking at forks and stars, while other papers have looked at number of contributors, we believe both to be good measures of engagement for a repo.

6. Conclusion

Through our research we touched on many distinct aspects of secret detection and handling in open-source repositories. We found that secrets have decreased in repositories that are newer but both GitHub and GitLab repositories had similar amounts of secrets. Many such secrets were written directly into the code showing the importance of early secrets management.

We chose our three main scanners to highlight how opensource industry scanners perform and have seen more studies with more scanners. We have yet to see a comprehensive study of scanners on the market as well as comparative study utilizing real world repository scanning. This would be an excellent avenue for future studies as well as utilizing our top scanner xGitGuard to garner further metrics about the secrets in repos.

API keys can be found easily by all the scanners showing that the use of such products can limit the amount of secrets developers accidentally leak. We found thousands of leaked secrets in our scans of GitHub and Gitlab with issues prevailing in all categories of repositories. Developers' secrets can be mitigated utilizing many different methods with education and stronger code review practices being key to preventing code leakage in the future.

Developers' use of vaults and other secret stores kept secrets out of several repositories we investigated. It should be noted that these techniques should be utilized from the beginning to ensure past secrets that could have been hard coded for testing do not make it into production code.

Abbreviations

API	application programming interface
AWS	Amazon Web Services
CI/CD	continuous integration and continuous delivery
IDE	integrated development environment
ML	machine learning
RSA	Rivest–Shamir–Adleman
SCTE	Society of Cable Telecommunications Engineers
SSH	secure socket shell
UUID	universally unique identifier

Bibliography & References

1. Auth0. Repo-supervisor, 2017. Online: https://github. com/auth0/repo-supervisor.

2. Dunning Julian Ayrey Dylan, Decker Dustin. Trufflehog, 2018. Online: https://github.com/trufflesecurity/ trufflehog.

3. Marco Balduzzi, Jonas Zaddach, Davide Balzarotti, Engin Kirda, and Sergio Loureiro. A security analysis of amazon's elastic compute cloud service. In *Proceedings of the 27th annual ACM symposium on applied computing*, pages 1427–1434, 2012.

4. Tegawend´e F Bissyand´e, David Lo, Lingxiao Jiang, Laurent R´eveillere, Jacques Klein, and Yves Le Traon. Got issues? who cares about it? a large scale investigation of issue trackers from github. In *2013 IEEE 24th international symposium on software reliability engineering (ISSRE)*, pages 188–197. IEEE, 2013.

5. Github Blog. 100 million developers and counting, 2023. Online: https://github.blog/2023-01-25-100-million-developers-and-counting/.

6. Jordi Cabot, Javier Luis C´anovas Izquierdo, Valerio Cosentino, and Bel´en Rolandi. Exploring the use of labels to categorize issues in open-source software projects. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 550–554. IEEE, 2015.

7. Comcast. xgitguard, 2022. Online: https://github.com/Comcast/xGitGuard.

8. Valerio Cosentino, Javier Luis, and Jordi Cabot. Findings from github: Methods, datasets and limitations. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 137–141, New York, NY, USA, 2016. Association for Computing Machinery.

9. Michael Dowling. git-secrets: Prevents you from committing passwords and other sensitive information to a git repository., 2015. Online: https://github.com/awslabs/ git-secrets.

10. Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 422–431. IEEE, 2013.

11. Runhan Feng, Ziyang Yan, Shiyan Peng, and Yuanyuan Zhang. Automated detection of password leakage from public github repositories. In *Proceedings of the 44th International Conference on Software Engineering*, pages 175–186, 2022.

12. S´ergio Fernandes and Jorge Bernardino. What is bigquery? In *Proceedings of the 19th International Database Engineering & Applications Symposium*, pages 202–203, 2015.

13. GitGuardian. Voice of practitioners the state of secrets in appsec. Whitepaper, GitGuardian, 2023.

14. Github. Dependabot: Automated dependency updates built into github, 2023. Online: https://github.com/dependabot.

15. Georgios Gousios. The ghtorent dataset and tool suite. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 233–236. IEEE, 2013.

16. Javier Luis C´anovas Izquierdo, Valerio Cosentino, Bel´en Rolandi, Alexandre Bergel, and Jordi Cabot. Gila: Github label analyzer. In *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*, pages 479–483. IEEE, 2015.

17. David Knothe and Frederick Pietschmann. Large-scaleexploit of github repository metadata and preventive measures. *arXiv preprint arXiv:1908.05354*, 2019.

18. Munir Kotadia. Aws urges developers to scrub github of secret keyss, March 2014. [Online; posted Mar 24 2014].

19. Alexander Krause, Jan H Klemmer, Nicolas Huaman, Dominik Wermke, Yasemin Acar, et al. Committed by accident: Studying prevention and remediation strategies against secret leakage in source code repositories. *arXiv preprint arXiv:2211.06213*, 2022.

20. Thomas Maillart, Mingyi Zhao, Jens Grossklags, and John Chuang. Given enough eyeballs, all bugs are shallow? Revisiting Eric Raymond with bug bounty programs. *Journal of Cybersecurity*, 3(2):81–90, 10 2017.

21. Steve Mansfield-Devine. Google hacking 101. *Network Security*, 2009(3):4–6, 2009.

22. Michael Meli, Matthew McNiece, and Bradley Reaves. How bad can it git? characterizing secret leakage in public github repositories. 01 2019.

23. OpenSSF. Openssf scorecard, 2020. Online: https:// github.com/ossf/scorecard.

24. Aakanksha Saha, Tamara Denning, Vivek Srikumar, and Sneha Kumar Kasera. Secrets in source code: Reducing false positives using machine learning. In *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pages 168–175. IEEE, 2020.

25. Kamran Shaukat, Suhuai Luo, Vijay Varadharajan, Ibrahim A. Hameed, Shan Chen, Dongxi Liu, and Jiaming Li. Performance comparison and current challenges of using machine learning techniques in cybersecurity. *Energies*, 13(10), 2020.

26. Rui Shu, Xiaohui Gu, and William Enck. A study of security vulnerabilities on docker hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 269–280, 2017.

27. Vibha Singhal Sinha, Diptikalyan Saha, Pankaj Dhoolia, Rohan Padhye, and Senthil Mani. Detecting and mitigating secret-key leaks in source code repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 396–400, 2015.

28. Vibha Singhal Sinha, Diptikalyan Saha, Pankaj Dhoolia, Rohan Padhye, and Senthil Mani. Detecting and mitigating secret-key leaks in source code repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 396–400. IEEE, 2015.

29. Verizon. 2019 data breach investigations reports, 2019. Online: https://www.verizon.com/business/resources/ reports/dbir/2019/results-and-analysis/.

30. Verizon. 2023 data breach investigations reports, 2023. Online: https://www.verizon.com/business/resources/ reports/dbir/.

31. Verizon. Investigations report- results and analysis: Introduction, 2023. Online: **Error! Hyperlink reference not valid.**//www.verizon.com/business/resources/reports/dbir/ 2023/results-and-analysis-intro/.

32. Verizon2002. 2020 data breach investigations reports, 2020. Online: https://www.verizon.com/business/resources/ reports/dbir/2020/results-and-analysis/.

33. Verizon2021. 2021 data breach investigations reports, 2021.Online: https://www.verizon.com/business/resources/ reports/dbir/2021/masters-guide/summary-of-findings/.

34. Jinpeng Wei, Xiaolan Zhang, Glenn Ammons, Vasanth Bala, and Peng Ning. Managing security of virtual machine images in a cloud environment. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 91–96, 2009.

35. Elliott Wen, Jia Wang, and Jens Dietrich. Secrethunter: A large-scale secret scanner for public git repositories. In *2022 IEEE TrustCom,* pzges 123-130, IEEE,2022

36. Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1296-1310, 2019.

