

ABSTRACT

BASAK, SETU KUMAR. Towards Risk-based Secret Management in Software Artifacts. (Under the direction of Dr. Laurie Williams).

Since 2020, GitGuardian has been detecting checked-in hard-coded secrets in GitHub repositories. During 2020-2023, GitGuardian has observed an upward annual trend and a four-fold increase in hard-coded secrets, with 12.8 million exposed in 2023. However, removing all the secrets from software artifacts is not feasible due to time constraints and technical challenges. Additionally, the security risks of the secrets are not equal, protecting assets of different risks accessible through asset identifiers (a DNS name and a public or private IP address). The value of the protected asset can vary from a database with mock data to medical data whose breach can incur fines. In addition, the ease of accessing the asset can vary since an attacker may need to be on the same network or have physical access to the host server to access the asset. Thus, secret removal should be prioritized by security risk reduction, which existing secret detection tools do not support.

Thesis Statement: Detecting assets protected by secrets and providing the security risk for secret-asset pairs can enable the prioritization of secret removal in software artifacts.

To this end, we conducted six original studies. In the first study, we extracted 779 questions related to checked-in secrets on Stack Exchange and applied qualitative analysis to determine the challenges and the solutions posed by others for each of the challenges. We identify 27 challenges and 13 solutions. We observe an increasing trend in questions about checked-in secrets lacking accepted answers.

In the second study, we conducted a grey literature review of 54 Internet artifacts and identified 24 secret management practices grouped into six categories. Our findings indicate that using secret detection tools and employing short-lived secrets are the most recommended practices to avoid accidentally committing secrets and limit secret exposure.

In the third study, we curated SecretBench, the first publicly available labeled dataset of source codes containing 97,479 secrets (of which 15,084 are true secrets) of 7 secret types extracted from 818 public GitHub repositories.

In the fourth study, we evaluated five open-source and four proprietary secret detection tools against SecretBench. We observed that tools generate a lot of false positives (25%-

99%) and miss secrets (14%-99%). Our manual analysis of reported secrets reveals that false positives are due to employing generic regular expressions and ineffective entropy calculation. In contrast, false negatives are due to faulty regular expressions, skipping specific file types, and insufficient rulesets.

In the fifth study, we presented AssetHarvester, a static analysis tool to detect secret-asset pairs in a repository. Since the location of the asset can be distant from where the secret is defined, we investigated secret-asset co-location patterns and found four patterns. To identify the secret-asset pairs of the four patterns, we utilized three approaches (pattern matching, data flow analysis, and fast-approximation heuristics). We curated a benchmark of 1,791 secret-asset pairs of four database types extended from SecretBench to evaluate the performance of AssetHarvester. AssetHarvester demonstrates precision of (97%), recall (90%), and F1-score (94%) in detecting secret-asset pairs. Our findings indicate that data flow analysis employed in AssetHarvester detects secret-asset pairs with 0% false positives and aids in improving the recall of secret detection tools. Additionally, AssetHarvester shows 43% increase in precision for database secret detection compared to existing detection tools through the detection of assets, thus reducing developer's alert fatigue.

In the final study, we presented RiskHarvester, a risk-based tool to compute a security risk score based on the value of the asset and ease of attack on a database. We calculated the value of asset by identifying the sensitive data categories present in a database from the database keywords in the source code. We utilized data flow analysis, SQL, and Object Relational Mapper (ORM) parsing to identify the database keywords. To calculate the ease of attack, we utilized passive network analysis to retrieve the database host information. To evaluate RiskHarvester, we curated RiskBench, a benchmark of 1,791 database secret-asset pairs with sensitive data categories and host information manually retrieved from 188 GitHub repositories. RiskHarvester demonstrates precision of (95%) and recall (90%) in detecting database keywords for the value of asset and precision of (96%) and recall (94%) in detecting valid hosts for ease of attack. Finally, we conducted a survey (52 respondents) to understand whether developers prioritize secret removal based on security risk score. We found that 86% of the developers prioritized the secrets for removal with descending security risk scores.

© Copyright 2025 by Setu Kumar Basak

All Rights Reserved

Towards Risk-based Secret Management in Software Artifacts

by
Setu Kumar Basak

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2025

APPROVED BY:

Dr. Alexandros Kapravelos

Dr. Bradley Reaves

Dr. Dominik Wermke

Dr. Laurie Williams
Chair of Advisory Committee

DEDICATION

To my parents, whose selfless sacrifices and steadfast encouragement have been my greatest inspiration.

BIOGRAPHY

Setu Kumar Basak received a B.Sc. degree in Computer Science from Khulna University of Engineering & Technology, Khulna, Bangladesh, in 2016. He then joined as a software engineer at Enosis Solutions, Bangladesh, where his domain was dental practice management software systems and opportunity model analysis. After spending five and a half years in the industry, in August 2021, he started his Ph.D. in Computer Science at North Carolina State University, USA, under the guidance of Dr. Laurie Williams. His doctoral research interests mainly include software engineering and software security. He loves to play cricket, table tennis, and volleyball.

ACKNOWLEDGEMENTS

There are many without whom this work would not have been possible, and I would like to express my gratitude to them. First and foremost, I would like to express my gratitude and appreciation to my advisor, Professor Laurie Williams, for her valuable guidance and encouragement throughout the research. I can never thank her enough for the support and encouragement that I received during my studies. It was a great honor for me to know and work with her. I also would like to thank Professor Bradley Reaves, Professor Alexandros Kapravelos, Professor Dominik Wermke, and all other faculty members for their precious time reviewing my thesis, attending my exams, and co-authoring papers.

Furthermore, I am so thankful for meeting and working with these great colleagues here at North Carolina State University: Dr. Sarah Elder, Dr. Rezvan Mahdavi-Hezaveh, Dr. Nasif Imtiaz, Dr. Rayhanur Rahman, Nusrat Zahan, Mahzabin Tamanna, Imranur Rahman, Sivana Hamer, and Jill Marley. I also want to thank Dr. Lorenzo Neil, Tanmay Paredeshi, Jamison Cox, K. Virgil English, Ken Ogura, and Vitesh Kambara for helping in my research.

I want to thank North Carolina State University for offering me graduate student support through out my PhD. In addition, I want to thank the National Science Foundation (NSF) for funding my Ph.D. work.

I would also like to thank my parents for their endless support and love. Their encouragement and patience inspired me to come this far and be where I am now. In addition, I want to thank my friends of the “Sharing is Caring” group, who have been a constant source of joy and companionship throughout my PhD journey. Last but not least, no thanks can be enough for Munmun Basak, my beautiful wife, who has always been there for me with her support, sacrifice, encouragement, and wisdom.

TABLE OF CONTENTS

List of Tables	ix
List of Figures	xi
Chapter 1 INTRODUCTION	1
Chapter 2 Key Concepts and Related Work	5
2.1 Key concepts	5
2.1.1 Secrets in Software Artifacts	5
2.1.2 Assets in Software Artifacts	6
2.1.3 Version Control Systems	6
2.1.4 Secret Detection Tools	7
2.1.5 Grey Literature	7
2.2 Related Work	7
2.2.1 Hard-coded Secrets	8
2.2.2 Secret Management Practices	9
2.2.3 Developer Challenges	9
2.2.4 Benchmark Dataset of Secrets	9
2.2.5 Machine Learning for Reducing False Positives of Secrets	10
2.2.6 Stack Overflow Survey	10
Chapter 3 Study I: What Challenges Do Developers Face About Checked-in Secrets in Software Artifacts?	11
3.1 Methodology	12
3.1.1 Step 1: Q&A Site Selection	12
3.1.2 Step 2: Content Collection	13
3.1.3 Step 3: Identifying Question and Answer Categories	15
3.1.4 Step 4: Analysis	16
3.2 Results	18
3.2.1 Answer to RQ1: What are the technical challenges faced by developers related to checked-in secrets?	19
3.2.2 Answer to RQ2: What solutions do developers get for mitigating checked-in secrets?	27
3.3 Discussion and Recommendations	30
3.4 Ethics	33
3.5 Threats to Validity	33
3.6 Conclusion	34
Chapter 4 Study II: What are the Practices for Secret Management in Software Artifacts?	36
4.1 Methodology	37

4.1.1	Search Internet Artifacts	38
4.1.2	Apply Inclusion Criteria	39
4.1.3	Find Secret Management Practices	39
4.2	Results	40
4.2.1	Practices for Keeping Secrets Out of Source Code (OSC)	41
4.2.2	Practices for Securely Storing Secrets (SSC)	42
4.2.3	Practices to Limit Secrets Exposure (LSE)	43
4.2.4	Practices for Avoiding Accidental Secrets Commit (ASC)	45
4.2.5	Practices for Managing Secrets in Deployment (MSD)	46
4.2.6	Organizational Practices to Enforce Policies for Secrets Protection (OEP)	47
4.3	Discussion	49
4.4	Conclusion	50
Chapter 5 Study III: SecretBench: A Dataset of Software Secrets		51
5.1	Data Extraction	52
5.2	Data Description	56
5.2.1	Curated and Derived Fields	56
5.2.2	Data Characteristics	56
5.2.3	Data Storage	58
5.3	Originality of SecretBench	58
5.4	Research Opportunities	59
5.5	Ethics and Data Protection	59
5.6	Threats to Validity	60
5.7	Conclusion	60
Chapter 6 Study IV: A Comparative Study of Software Secrets Reporting by Secret Detection Tools		61
6.1	Benchmark Dataset	62
6.2	Secret Detection Tools	64
6.2.1	Selection of Secret Detection Tools	64
6.2.2	Tools Description	65
6.2.3	Machine Configuration	67
6.3	Analyzing Tool Results	67
6.3.1	Secret Metadata	67
6.3.2	Filter and Compare Tool Alerts	68
6.3.3	Tool Metric	71
6.4	Results	72
6.4.1	RQ1: How do the secret detection tools perform in detecting secrets, in terms of precision and recall?	72
6.4.2	RQ2: What features are offered by the secret detection tools to aid in preventing secrets exposure?	79
6.5	Discussion and Recommendations	80

6.6	Ethics and Data Protection	81
6.7	Threats to Validity	82
6.8	Conclusion	82
Chapter 7 Study V: AssetHarvester: A Static Analysis Tool for Detecting Secret-Asset Pairs in Software Artifacts		84
7.1	Asset Type Selection	87
7.2	AssetBench	87
7.3	Secret-Asset Co-location Patterns	90
7.4	AssetHarvester	91
7.5	Performance of AssetHarvester	100
7.6	Discussion	105
7.7	Ethics and Data Protection	107
7.8	Threats to Validity	107
7.9	Conclusion	108
Chapter 8 Study VI: How Can We Mitigate 12.8 Million Checked-in Secrets in Software Artifacts?		109
8.1	Research Methodology	112
8.1.1	RiskBench Curation	112
8.1.2	Value of Asset and Ease of Attack Patterns	114
8.1.3	Developer Survey	118
8.2	RiskHarvester Construction	120
8.2.1	Step 1: Identifying Secret-Asset Pairs	120
8.2.2	Step 2: Identifying Value of Asset	122
8.2.3	Step 3: Identifying Ease of Attack	125
8.2.4	Step 4: Calculating Security Risk Score	127
8.3	Results	129
8.3.1	Performance of RiskHarvester	129
8.3.2	Developer Survey	131
8.4	Discussion	132
8.5	Threats to Validity	134
8.6	Conclusion	135
8.7	Ethical Considerations	136
8.8	Open Science Policy Compliance	136
Chapter 9 Conclusion and Future Work		137
9.1	Conclusion	137
9.2	Future Work	138
Bibliography		141
APPENDICES		161
Appendix A	Study I Supplemental Information	162

A.1 Mapping of Question Category to Answer Category	162
Appendix B Study VI Supplemental Information	165

LIST OF TABLES

Table 1.1	Security Risk for Each Secret-Asset Pair	2
Table 3.1	Basic statistics of Stack Overflow (SO), Information Security (IS) and Software Engineering (SE) sites	13
Table 3.2	List of Tags and Keywords used to extract questions from Stack Exchange sites	14
Table 3.3	Question Count Per Year for Stack Overflow (SO), Information Security (IS) and Software Engineering (SE) sites	15
Table 3.4	27 question categories. References to all the examples and developer quotes are available online [1]	20
Table 3.4	27 question categories (Continued). References to all the examples and developer quotes are available online [1]	21
Table 3.4	27 question categories (Continued). References to all the examples and developer quotes are available online [1]	22
Table 3.4	27 question categories (Continued). References to all the examples and developer quotes are available online [1]	23
Table 3.5	Summary of identified question categories, sorted by decreasing question proportion (QC)	25
Table 3.6	Ranked Order of Question Categories Based on Popularity (PQ) and Unsatisfactory Answer Percentage (UNC)	26
Table 5.1	Overview of the SecretBench Dataset	57
Table 5.2	The categories of secrets in SecretBench	58
Table 6.1	The eight categories of secrets in SecretBench.	63
Table 6.2	SecretBench’s top five file types on true secrets.	64
Table 6.3	Precision, Recall, F1-Score, Scan Time (ST), and Popularity Score (PS) of each tool.	73
Table 6.4	Recall of each tool for eight secrets categories.	74
Table 6.5	Seven categories of features and additional secrets metadata provided by each tool.	80
Table 7.1	Count of Secret-Asset pairs for four databases	89
Table 7.2	List of regexes categorized into three groups for identifying database connection string	92
Table 7.3	List of Python database drivers with their supported arguments for secret-asset pairs	95
Table 7.4	Statistics of the presence of database assets in the neighboring lines of the secrets of the same file in AssetBench	97
Table 7.5	Precision, Recall and F1-score of AssetHarvester for each database type100	

Table 7.6	Comparison of AssetHarvester with 9 Baseline Secret Detection Tools on Secret Detection	104
Table 8.1	Security Risk for Each Secret-Asset Pair	110
Table 8.2	Count of Secret-Asset pairs in RiskBench	113
Table 8.3	Examples of a data category with the corresponding sensitivity level for seven domains provided by Google Cloud DLP. The full list can be found online [2].	114
Table 8.4	The Developer Survey Questions. The full questionnaire of the online survey can be found online [2].	119
Table 8.5	List of Python database drivers and ORM frameworks with their supported arguments for secret-asset pairs	121
Table 8.6	Precision, Recall, and F1-Score of RiskHarvester in identifying the database name, table, and column names	129
Table A.1	The mapping of answers to each question category	162
Table A.1	The mapping of answers to each question category	163
Table A.1	The mapping of answers to each question category	164
Table B.1	Regexes categorized into three groups based on connection string format similarity for identifying secret-asset pairs	165
Table B.2	System and User role prompt for detecting placeholder/dummy DNS names.	166

LIST OF FIGURES

Figure 2.1	An AWS access key is present in the source code	6
Figure 2.2	An API URL and Database Server Address are present in the source code	6
Figure 3.1	Trend of Unsatisfactory Answer Per Year	24
Figure 3.2	Temporal Trend of each identified question category. The month of the x-axis is shown in three-year interval. The zero value of Temporal Trend indicates no question is posted on the specific month for a category.	27
Figure 4.1	An overview of our grey literature review methodology.	38
Figure 4.2	Application of inclusion criteria on our grey literature dataset to collect the set of 54 Internet artifacts for our study. Grey literature dataset is available online [3].	40
Figure 6.1	Different outputs of the same secret by three tools.	70
Figure 6.2	Overlap ratio of secrets reported by each tool.	75
Figure 6.3	Venn diagram for overlap of unique true positive secrets among top three tools based on recall. Subfigure (a) depicts the overlap of Gitleaks, TruffleHog, and ggshield. Subfigure (b) depicts the overlap of Gitleaks, SpectralOps, and TruffleHog.	76
Figure 7.1	A secret can protect both real and non-sensitive assets, such as a public IP address (real) and localhost (non-sensitive).	85
Figure 7.2	The database asset identifier has three parts (host, port, and database name) that are defined separately in the same line, in separate lines, or together in the same string.	88
Figure 7.3	We identified four types of secret-asset co-location patterns in the source code.	90
Figure 7.4	The database credentials and server address are passed in a specific order in the database driver function.	95
Figure 7.5	The database credentials and server address are passed as keyword arguments in the database driver functions.	96
Figure 7.6	The config.yml file contains the database secret-asset pair that is read in the main.py file. The secret-asset values are accessed by the key names and passed to the driver function.	98
Figure 7.7	Multiple or zero corresponding assets can be present in the neighboring lines of a secret.	99
Figure 7.8	The number of unique secret-asset pairs found by Pattern Matching, Data Flow Analysis, and Fast-Approximation Heuristic approaches. .	102

Figure 8.1 Asset’s value can be inferred from the database name, table names, and column names from the source code. 116

Figure 8.2 We identified three patterns to locate database, table, and column names for each secret-asset pair in the source code. 117

Figure 8.3 A flow diagram for assigning ease of attack category for an asset identified in the source code. 128

CHAPTER

1

INTRODUCTION

In March 2024, GitGuardian reported an increasing trend in the past four years on secrets exposure in GitHub repositories [4]. In 2023 alone, 12.8 million new secrets were exposed, a four-fold increase compared with 2020. They also found that 1.7 million authors leaked secrets out of 14.9 million who pushed code to GitHub in 2023. Secrets, such as database credentials, cryptographic keys, API keys, and authentication tokens, are essential for integrating with external services such as customer databases, location services, and payment systems. During software development, these secrets may need to be shared by the developers working on a team and, after deployment, may need to be distributed to applications. However, developers either inadvertently or deliberately expose hard-coded secrets in application packages and version control systems (VCS), leaking sensitive information to malicious actors [5, 6]. For example, an attacker leveraged hard-coded credentials present in Uber’s PowerShell script and launched an account takeover of their internal tools and productivity applications in September 2022 [7].

However, removing all the secrets from software artifacts is not feasible since developers ignore secret detection tool alerts due to false positives, time pressure, and technical challenges [8]. Additionally, the security risks of all the exposed secrets are not equal. The value of the asset protected by the secret can vary from a database with mock data to a med-

Table 1.1: Security Risk for Each Secret-Asset Pair

Secret-Asset Pair	Value of Asset	Ease of Attack	Security Risk
Pair 1	1	100	100
Pair 2	40	1	40
Pair 3	40	100	4000

ical database containing millions of patients’ data, a breach of which can cost a company millions of dollars in fines. Similar to the value of an asset, the ease of accessing an asset varies based on different factors. For example, an asset with a public IP address can be easy for an attacker to access. In contrast, an asset with a private IP address or Localhost will require an attacker to be in the same network or have physical access to the host machine. Thus, secret removal should be prioritized by security risk reduction, which existing secret detection tools do not support.

The National Institute of Standards and Technology (NIST) defines the security risk of an entity as a function of the impact of the adverse event and the likelihood of the event by a threat source [9]. For a secret, the security risk can be defined as the function of the value of asset and the attacker’s ease of accessing the asset protected by the secret. Protection Poker [10], a threat modeling game, employs the product of relative measures of “value points” and “ease points” for security risk quantification, such as one requirement being five times easier to attack than another. Similarly, for a secret, the security risk can be defined as the *product of value of asset and ease of attack*. This security risk computation is based upon the hypothesis that attackers are more likely to succeed in attacking assets of high value and that are easier to attack. Table 1.1 provides an example of security risk computation for three secret-asset pairs. A secret-asset pair consists of a secret, such as a database password, and a protected asset by the secret, such as the database server. Pair 3 is deemed to have the highest security risk because the value of the asset is 40 times more valuable than Pair 1 and 100 times easier to attack than Pair 2. Thus, removing the Pair 3 secret from the source code is of primary importance. We hypothesize that providing a security risk score for each secret-asset pair can aid developers in prioritizing the secret removal efforts.

Thesis statement: Detecting assets protected by secrets and providing the security risk for secret-asset pairs can enable the prioritization of secret removal in software artifacts.

We demonstrate our thesis statement through six original studies, each of which inves-

tigates a series of research questions:

1. **Study I:** We performed an empirical investigation of the challenges developers faced about checked-in secrets in software artifacts and the solutions posed by other developers to mitigate the challenges. We investigated the questions and the corresponding solutions posted by developers on Stack Exchange and identified 27 challenges and 13 solutions. This work has been published at the 45th International Conference on Software Engineering (ICSE), 2023.
2. **Study II:** We performed a grey literature review of Internet artifacts to identify secret management practices in software artifacts. We identified 24 practices grouped into six categories, comprising developer and organization practices. Using secret detection tools and employing short-lived secrets are the most recommended practices to avoid accidentally committing secrets and limit secret exposure. This work has been published at the IEEE Secure Development Conference (SecDev), 2022.
3. **Study III:** To evaluate and improve the secret detection tools, we curated SecretBench, the first publicly available benchmark dataset of software secrets. We used the dataset in Study IV, V, and VI. This work has been published at the 20th International Conference on Mining Software Repositories (MSR), 2023.
4. **Study IV:** We evaluated 9 secret detection tools (5 open-source and 4 proprietary) against the benchmark dataset. We observed that the tools output a lot of false positives (25%-99%). We performed a qualitative analysis of the rules triggering the false positives and false negatives. In addition, we categorized the features provided by the secret detection tools to aid in preventing secret exposure. This work has been published at the 17th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2023.
5. **Study V:** We investigated the secret-asset co-location patterns in the source code and detected the assets protected by the secrets by constructing a static analysis tool (AssetHarvester). To identify the secret-asset pairs, we utilized three approaches (pattern matching, data flow analysis, and fast-approximation heuristics). This work has been published at the 47th International Conference on Software Engineering (ICSE), 2025.
6. **Study VI:** We presented RiskHarvester, a risk-based tool to compute the security risk score of secret-asset pairs in a repository. For RiskHarvester construction, we

utilized data flow analysis with SQL and ORM parsing and passive network analysis. Additionally, we conducted an online developer survey on the effectiveness of security risk scores for prioritizing secrets removal. This work has been submitted at the 34th USENIX Security Symposium, 2025.

Overall, by conducting the six studies, we contribute the following:

- A set of challenges faced by the developers about checked-in secrets and a set of solutions or suggestions posed by other developers to mitigate the challenges (Chapter 3);
- A set of practices that practitioners can follow to avoid exposure of secrets in software artifacts (Chapter 4);
- A publicly-available benchmark database of software secrets for future researchers and tool developers (Chapter 5);
- A first comparative study of the existing open-source and proprietary secret detection tools and a qualitative analysis of the reports generated by the tools (Chapter 6);
- A categorization of the features provided by the secret detection tools to aid in preventing secrets exposure (Chapter 6);
- A set of secret-asset co-location patterns generic to secret-asset types, formulating ideas for automated identification of secret-asset pairs within source code (Chapter 7);
- A static analysis tool (AssetHarvester) to detect secret-asset pairs in source code (Chapter 7);
- A first risk-based analysis tool (RiskHarvester) for prioritizing the removal of the secrets from source code (Chapter 8);
- A dataset of false positive secrets reported by the tools (Chapter 6); and
- A dataset of secret-asset pairs with value of asset and ease of attack information that can be extended and utilized by researchers and tool developers for future research and tool development (Chapter 7 and 8).

The rest of the dissertation is structured as follows: We discuss the key concepts and related work to our studies in Chapter 2. Chapters 3, 4, 5, 6, 7, and 8 discuss the methodology and findings of our six completed studies. We conclude the thesis by summarizing all the studies in Chapter 9.

CHAPTER

2

KEY CONCEPTS AND RELATED WORK

In this chapter, we explain key concepts relevant to our work and review the existing related work.

2.1 Key concepts

In this section, we briefly explain the key concepts related to our work.

2.1.1 Secrets in Software Artifacts

In software development, “secrets” refer to API keys, cryptography keys, database credentials, SSL certificates, private keys, usernames and passwords, and other credentials. These secrets are required to perform authentication with external services such as payment systems and online storage. In addition, these secrets should be kept confidential and protected from unauthorized access. For example, as shown in Figure 2.1, the secret is the AWS access key present in line 4.

```

upload.js
1 exports.getAllFiles = getAllFiles;
2 const s3 = new aws_sdk_1.S3({
3     accessKeyId: "AKIA4MTWL5J2RABD7QDF",
4     secretAccessKey: "eI\SWgvKiFNFDPCkbhfc1017R1XFnnwLot03qIqH",
5 });

```

Figure 2.1: An AWS access key is present in the source code

2.1.2 Assets in Software Artifacts

In software development, a secret is used to access an asset such as the database and API service. The asset is present as asset identifiers in the source code. A URL, DSN name, or IP address are the assets' identifiers. For example, as shown in Figure 2.2, the OpenWeatherMap API URL (line 3) is the asset identifier for the API key (line 1), whereas the database server address (line 7) is the asset identifier for the database username and password.

```

1 api_key = "12abc3d45ef6789012345g6789h0ij12"
2 city_id = "5128581"
3 base_url = "https://api.openweathermap.org/data/2.5/weather?"
4 final_url = base_url + "appid=" + api_key
5
6
7 connection_string = "server=10.10.0.1;username=test;password=test"

```

Figure 2.2: An API URL and Database Server Address are present in the source code

2.1.3 Version Control Systems

Version Control Systems (VCS) such as GitHub [11] and GitLab [12] are used by developers to track the progress of source code across the software development lifecycle. For example, GitHub [11] is one of the most popular platforms for hosting open-source software development projects [13]. As of March 2024, GitHub has over 100 million developers and more than 420 million repositories [13]. VCS maintains a record of every source code change, complete with authorship, timestamp, and other details. Like source code, developers can

push hardcoded secrets to VCS that are accessible to malicious actors. Even if developers remove the secrets from the source code, the secrets can remain buried in the commit history of VCS.

2.1.4 Secret Detection Tools

Secret detection tools are software tools or services designed to detect secrets within source code that attackers can exploit. These tools typically use regular expressions, entropy calculation, and machine learning algorithms to detect secrets in the source code. Currently, many open-source and proprietary secret detection tools such as TruffleHog [14] and SpectralOps [15] are present. Via a pre-commit hook, secret detection tools can reject any commit containing secrets that manual searches and reviews will miss. These tools can also find secrets buried in logs and histories. In addition, these tools can be integrated into continuous integration or continuous deployment (CI/CD) pipelines to actively break build/deploy when secrets are found in source code.

2.1.5 Grey Literature

Grey literature is defined as “... literature that is not formally published in sources, such as books or journal articles” [16]. Software practitioners may share their experiences as grey literature, which can be considered a valuable resource for researchers and other practitioners [17]. Academic publications reflect the state-of-the-art, and grey literature provides insight into the state-of-the-practice in any research area. In practical research areas such as software engineering and software security, combining state-of-the-art and state-of-the-practice is essential to provide valuable results [18]. In the area of software secret management, a large number of grey literature artifacts exist, but only a small number of peer-reviewed papers have been published. For example, we used grey literature review in Chapter 3 and Chapter 4.

2.2 Related Work

In this section, we describe the academic literature related to our thesis.

2.2.1 Hard-coded Secrets

Previous studies [5, 19, 20, 21, 22, 23] have investigated the underlying causes of the exposure of secrets in software artifacts. Researchers have found that keeping hard-coded secrets in software artifacts is the most prevalent insecure practice that developers adopt, causing secret leakage.

In 2019, Meli et al. [5] conducted a large-scale and longitudinal analysis of secret leakage on GitHub. They examined a 13% snapshot of public GitHub repositories and found over 200K hard-coded secrets. They focused on private key files and 11 high-impact platforms having distinctive API key formats. In addition, they observed that secret leakage is not only pervasive but also thousands of new and unique secrets are leaked every day.

Sinha et al. [19] outlined three methods for detecting API keys present in the source code. The three methods are pattern-based search, heuristic-driven filtering, and source-based program slicing. They evaluated the effectiveness of the methods using a sample set of 84 GitHub repositories. In addition, they enumerated the mechanisms developers could use to manually prevent secret leaks from repositories.

Rahman et al. [20] investigated 5,232 Infrastructure as Code (IaC) such as Puppet scripts extracted from 293 open-source repositories and observed seven “Security Smells”. Security smells indicate potential security flaws that can lead to security breaches. Among the seven security smells, hard-coded secrets were found to be the most frequent, with 1,326 occurrences. In another study, Rahman et al. [21] conducted qualitative analysis with 1,956 Ansible and Chef scripts. They constructed a static analysis tool called Security Linter for Ansible and Chef scripts (SLAC) to automatically identify security smells in 50,323 scripts collected from 813 open-source software repositories. By applying SLAC, they identified 46,600 occurrences of security smells, including 7,849 hard-coded secrets.

Hard-coded secrets have also been found in GitHub Gists, which are used to share code snippets among developers. Rayhanur et al. [22] investigated 5,822 publicly available Python Gists. They found 689 instances of hard-coded secrets in the code snippets.

Koishybayev et al. [23] analyzed the CI/CD workflow files, such as YAML files from GitHub. After investigating 447,238 YAML files from 213,854 GitHub repositories, they found 333 instances of hard-coded secrets present. They raised the issue to the 333 repositories about the hard-coded secret leaks. All of these prior works suggest that hard-coded secrets have leaked in different forms in software artifacts.

2.2.2 Secret Management Practices

Rahman et al. [24] conducted a grey literature review of 38 Internet artifacts and identified 12 secret management practices related to IaC. They identified practices that are applicable for all IaC languages, such as prioritized encryption, and language-specific practices, such as state separation for Terraform. We took motivation from this study and concentrated our research efforts on finding secret management practices for software artifacts in general, as discussed in Chapter 4.

2.2.3 Developer Challenges

Rayhanur et al. [8] conducted a developer survey in an XTech company (anonymized) to investigate the challenges developers face with secret detection tools. They identified that developers face challenges using secret detection tools due to the overwhelming volume of alerts with high false positives. In addition, they revealed that the effectiveness of the tools was limited due to various factors, such as a lack of contextual understanding of the developers and challenges to integrating the secret detection tool in the existing workflow.

In a mixed-methods study, Krause et al. [25] surveyed 109 developers for Upwork and GitHub and conducted 14 in-depth semi-structured interviews with developers who experienced secret leakage in the past. They found that 30.3% of their participants have encountered secret leakage in the past. These developers are facing several challenges with secret leakage prevention and remediation, such as estimating the risks of leaked secrets. They strongly recommended that developers take preventative measures and be aware of the risks of leaking secret information.

2.2.4 Benchmark Dataset of Secrets

Previous studies [5, 19, 26] have extracted secrets from the GitHub repositories, but none made their dataset public for future research purposes. Saha et al. [26] created a labeled dataset of 5000 secrets (700 true secrets) from 300 GitHub repositories using 32 regex patterns. With the dataset, they applied machine learning algorithms to distinguish true secrets. However, the repositories matched by regex patterns are not filtered for demo and inactive projects, and no information is provided on the files and languages covered. Sinha et al. [19] created a dataset of 84 GitHub repositories and identified pattern-based search and heuristics-driven filtering approaches to reduce the false positive detection of secrets.

However, their dataset is small and contains only AWS credentials. In this thesis, we created a publicly-available benchmark dataset of software secrets, as discussed in Chapter 5.

2.2.5 Machine Learning for Reducing False Positives of Secrets

Recent research [26, 27, 28] has employed Machine Learning (ML) algorithms to reduce the false positives of software secrets. Saha et al. [26] employed a Voting Classifier (a combination of Logistic Regression, Naive Bayes, and SVM) to distinguish real secrets from false positives. Feng et al. [27] applied deep neural networks to uncover the intrinsic characteristics of textual passwords and detect real passwords by reducing false positives. Konygin et al. [28] used a bigram-based approach to detect secrets with low false positive and false negative rates. With the help of bigrams, they transformed the string into a vector and used the CatBoost algorithm to build a binary classifier.

2.2.6 Stack Overflow Survey

To understand more clearly the challenges that developers face, researchers have performed qualitative research into investigating what questions developers are asking on Stack Overflow (SO) [29, 30, 31, 32, 33] as developers constantly search in SO for guidance on solving a challenge during development. Rahman et al. [33] analyzed 2758 Puppet-related questions on SO to investigate the challenges developers face in using Configuration as Code (CaC) tools. They found that developers' most common questions are fixing syntax errors, provisioning instances, and evaluating Puppet's feasibility for specific tasks. Tahir et al. [34] looked through 4000 posts from three Stack Exchange sites to see what developers were discussing about code smells and anti-patterns. They observed that developers frequently post questions on Stack Exchange to check the presence of smell in their code, effectively using Q&A sites as an informal code smell and anti-pattern detector. We took motivation from these studies and concentrated our research efforts on finding difficulties faced by developers for checked-in secrets by analyzing the questions posted in SO, as discussed in Chapter 3.

CHAPTER

3

STUDY I: WHAT CHALLENGES DO DEVELOPERS FACE ABOUT CHECKED-IN SECRETS IN SOFTWARE ARTIFACTS?

While the checked-in secrets issue is well-known through prior works [5, 19, 22, 20], little is known about developers' technical challenges in preventing secrets from being stored in software artifacts. Developers query online forums, such as a developer who posted a question on how to keep secrets out of VCS repositories [35]. Systematically analyzing questions asked by developers and solutions posed by others can reveal the technical challenges and practices adopted by the developers to protect the secrets.

The goal of our study is to aid researchers and tool developers in understanding and prioritizing opportunities for future research and tool automation for mitigating checked-in secrets through an empirical investigation of challenges and solutions related to checked-in secrets.

In this study, we analyze developers' questions and related solutions about checked-in secrets and provide answers to the following research questions:

- **RQ1:** What are the technical challenges faced by developers related to checked-in secrets?
- **RQ2:** What solutions do developers get for mitigating checked-in secrets?

Users can post questions describing a particular technical challenge for which they need support on Stack Exchange [36], a major question and answer (Q&A) site. An answer is a suggestion or solution to a technical challenge. Users can pose multiple answers to a question, but either zero or one answer is accepted. The answer approved by the user who posted the question is termed as the *accepted answer*. We refer to a question lacking an accepted answer or having no answers as a *question with unsatisfactory answer*.

We extracted 779 questions related to checked-in secrets from Stack Exchange spanning from September 2008 to December 2021. From these questions, we conducted a qualitative analytical approach called card sorting [37] to determine the question categories and related answer categories. We also perform quantitative analysis of question categories, which will help researchers and tool developers prioritize further study and tool development. In addition, the answer categories we presented give insights into which practices developers may have adopted. Following is a summary of this study’s contributions:

- A set of challenges faced by the developers about checked-in secrets; and
- A set of solutions or suggestions posed by other developers to mitigate the checked-in secret challenges

The rest of the chapter is structured as follows: The methodology used in our work is described in Section 3.1. We discuss our findings and recommendations in Sections 3.2 and 3.3, respectively. The ethics and limitations of our study are discussed in Sections 3.4 and 3.5, respectively. Finally, Section 3.6 draws the conclusion of our study.

3.1 Methodology

We provide our four-step process for data collection and question and answer analysis as follows:

3.1.1 Step 1: Q&A Site Selection

For collecting questions related to checked-in secrets, we selected Stack Exchange [36] which has been extensively used to gain insights from developers’ questions to align future

research and guide tools providers [34, 33]. Stack Exchange consists of 179 Q&A sites [36]. We extract the name and description of all the sites and manually read them. Then, we select sites that allow questions related to software development, software engineering, and software security. For example, the site “Software Engineering” can feature queries from developers, according to the site description “Q&A for professionals, academics, and students working within the systems development life cycle”. The first author selected three Q&A sites: “Stack Overflow” [38], “Information Security” [39] and “Software Engineering” [40]. The basic statistics of the three sites are shown in Table 3.1. In Step 2, we use these sites for question collection.

Table 3.1: Basic statistics of Stack Overflow (SO), Information Security (IS) and Software Engineering (SE) sites¹

Site	#Questions	#Answers	#Users	#Questions/Day
SO	23m	34m	18m	5.5k
IS	66k	114k	228k	9.6
SE	61k	173k	352k	5.5

3.1.2 Step 2: Content Collection

Start with initial tags and keywords for title and body: To increase the likelihood of speedy response and aid in automated search, each question can be given one or more tags [41]. Tags allow the extraction of questions that are specific to a given technology. For example, the tag “secret-key” can be used for identifying questions related to checked-in secrets according to the tag description “Use this tag for questions related to the creation, storage and usage of secret keys”. Initially, we select “secret-key” and “access-keys” tags. Users can also post questions without giving tags. To avoid missing candidate questions, we use secrets-related keywords, such as “expose”, “protect”, and “sensitive”, to search in the body and title of the questions to extract relevant questions.

Extract questions from Stack Exchange data explorer: The Stack Exchange dataset is accessible publicly via data dumps [42] and the Stack Exchange data explorer [41]. The data dumps are released quarterly, whereas the online Stack Exchange data explorer provides the

¹Based on data retrieved from the Stack Exchange Data Explorer [36] on June 2022

most recent data. We use the tags and keywords in a SQL query and extract data from the Stack Exchange data explorer instead of data dumps. We collect the ID, title, body, accepted answer, view count, score, creation date, closed date, and tags of each extracted question from the three sites identified in Step 1. We collected 6022, 2591, and 1415 questions from Stack Overflow, Information Security, and Software Engineering sites, respectively.

Identify relevant questions: We manually inspected each question’s title and body and accepted questions with a discussion related to checked-in secrets while rejecting all others.

Find new relevant tags and keywords: We use snowball sampling [43] which is a non-probability sample selection technique to locate hidden populations by relying on the characteristics of initial sample. Since a question can have multiple tags, we find new relevant tags by looking at all the tags present in the questions. For example, the question “Where to keep static information securely in Android app?” [44] can be found by “secret-key” tag. The question also has tags “access-token” and “security” which we can add to our list of tags for finding more questions. Similarly, add new keywords by reading the title and body of the question. Altogether, we used 59 tags and 42 keywords which can be found in Table 3.2.

Table 3.2: List of Tags and Keywords used to extract questions from Stack Exchange sites

Tags	Keywords
secret-key, access-keys, access-token, security, credentials, passwords, api-key, private-key, app-secret, connection-string, sensitive-data, environment-variables, config-files, certificate, configuration, google-api, amazon-s3, oauth, youtube-api, stripe-api, square, paypal, braintree, amazon-mws, gmail-api, twilio-api, mailgun, mailchimp, google-drive-api, key-management, development-process, coding-style, password-protection, source-code-protection, code-security, source-code, secure-coding, open-source, azure-key-vault, password-storage, password-management, key-exchange, confidentiality, sensitive-data-exposure, web-development, git, gitignore, version-control, github, svn, tfs, gitlab, repository, bitbucket, launchpad, mercurial, git-rewrite-history, git-history, git-filter-branch	expose, exposing, protect, protecting, sensitive, remove, removing, commit, committing, share, sharing, keep, keeping, manage, managing, delete, deleting, clear, clearing, ignore, ignoring, secure, securing, store, storing, hide, hiding, avoid, avoiding, push, pushing, host, hosting, security, connection string, secret, password, credential, private key, token, api key, access key

Table 3.3: Question Count Per Year for Stack Overflow (SO), Information Security (IS) and Software Engineering (SE) sites

Year	SO ^a	SO ^b	IS ^a	IS ^b	SE ^a	SE ^b	Total ^a	Total ^b
2008	23	4	0	0	0	0	23	4
2009	136	22	0	0	0	0	136	22
2010	212	30	5	0	28	1	245	31
2011	284	43	73	0	163	5	520	48
2012	370	44	129	2	170	8	669	54
2013	447	48	160	6	146	7	753	61
2014	485	41	257	5	136	3	878	49
2015	481	47	361	5	152	4	994	56
2016	581	51	340	6	126	2	1047	59
2017	581	76	323	5	110	4	1014	85
2018	538	63	268	5	107	4	913	72
2019	518	54	235	1	102	3	855	58
2020	722	88	226	1	90	2	1038	51
2021	644	83	214	4	85	2	943	89

^a Total number of questions before filtering

^b Total number of questions after filtering

Repeat and stop criteria: We repeat the previous step until we no longer found new tags and keywords in each set of extracted questions.

Finally, we identified 694 questions in Stack Overflow, 40 questions in Information Security, and 45 questions in Software Engineering. In total, we identified 779 questions from the three sites spanning from September 2008 to December 2021 which are available online [45]. The count of questions from each year before and after filtering is shown in Table 3.3.

3.1.3 Step 3: Identifying Question and Answer Categories

From the 779 checked-in secrets-related questions, two authors independently apply card sorting [37], a qualitative analysis technique, to identify the question and answer categories. Card sorting is a qualitative technique for classifying textual items into categories [37]. Card sorting aids in creating informative categories and is commonly used in research [33]. The following three phases of card sorting are implemented in accordance with Zimmerman et

al. [37]'s recommendations.

Preparation: Each question's ID, title, body, and accepted answer are collected.

Execution: The first and second authors perform card sorting by giving labels to each question and the corresponding answer and sort into categories. The body and title of the questions are used to derive question categories, whereas the accepted answers are used to derive answer categories.

Analysis: The obtained question and answer categories are cross-checked by both authors after the first and second authors finish their card sorting analysis individually. We use a negotiated agreement [46] to resolve the disagreed-upon categories. A negotiated agreement is an approach to discuss the disagreements among the raters to resolve disagreements when two or more raters code the same artifacts [46]. We resolve disagreements by discarding categories inappropriate for checked-in secrets or combining similar categories into one category. The first author determines 32 unique question categories and 16 unique answer categories. The second author determines 30 unique question categories and 14 unique answer categories. The first and second authors finalize 27 question and 13 answer categories by resolving the disagreements presented in Table 3.4 and Section 3.2, respectively.

3.1.4 Step 4: Analysis

We use the identified question and answer categories from Step 3 to answer our research questions.

RQ1: What are the technical challenges faced by developers related to checked-in secrets?

We break down RQ1 into four sub-research questions as follows:

- **RQ1.1** What are the questions developers ask about checked-in secrets?
- **RQ1.2** Which questions related to checked-in secrets exhibit more unsatisfactory answers?
- **RQ1.3** Which questions are the most popular among developers related to checked-in secrets?
- **RQ1.4** How do question categories related to checked-in secrets trend over time?

We investigate the four sub-research questions as following:

RQ1.1: What are the questions developers ask about checked-in secrets? We first provide the set of question categories to answer RQ1.1 along with a description and an example of each category which we determine from Step 3. Next, we compute the proportion of questions for each category x , $QC(x)$.

RQ1.2: Which questions related to checked-in secrets exhibit more unsatisfactory answers? A question with no accepted answer could indicate that the developer who asked the question was dissatisfied with the responses. Lacking accepted answers or having no answers may suggest an important category that needs assistance. We answer RQ1.2 by quantifying which of the checked-in secrets-related question categories has more questions with unsatisfactory answers. We compute the proportion of questions with unsatisfactory answers for question category x , $UNC(x)$.

Furthermore, we compute the proportion of questions with unsatisfactory answers for each year y , $TUN(y)$ to see how the proportion of unsatisfactory answers related to checked-in secrets has changed over time.

RQ1.3: Which questions are the most popular among developers related to checked-in secrets? Developers can view a question and corresponding answers without becoming registered users on Stack Exchange. The number of total visits for a question by registered and non-registered users of the website is used to calculate the View Count of a question [41]. The View Count can help us observe which questions are most popular among the developers. Registered users can also vote up or down on questions. Upvotes indicate that users find the question helpful, well-researched, or thought-provoking. Downvotes indicate that users believe the question lacks real explanation, contains misleading information, or is poorly researched. A question's Score on Stack Exchange is calculated by subtracting the number of downvotes from the number of upvotes [47]. Rather than selecting a single metric, we use both View Count and the Score of the question as a better approximation for question popularity. Previous studies use a similar a popularity metric [34].

We use Spearman's rho ρ [48] to verify the rank correlation between View Count and Score. View Count is found to have a significant correlation with Score ($\rho = 0.72$, $\alpha < 0.001$). We use Feature Scaling [49] to normalize the View Count and Score values of each question by Equation 3.1 since the range of both the metrics are different.

$$X_{nor} = \frac{X - X_{min}}{X_{max} - X_{min}} \quad (3.1)$$

where X denotes the original value, X_{min} denotes the range's minimum value, X_{max}

denotes the range's maximum value and X_{nor} denotes the normalized value.

To determine how popular a question is, we use the average of normalized View Count and Score values. Next, we calculate the popularity of each category x , $PQ(x)$ using Equation 3.2. A question category x with a high popularity score means developers need support to mitigate the specific challenge.

$$PQ(x) = \frac{\text{sum of popularity score of questions in category } x}{\text{total questions in category } x} \quad (3.2)$$

RQ1.4: How do question categories related to checked-in secrets trend over time?

We examine temporal trends, similar to previous studies [33, 29], to see how the number of questions relevant to the identified question categories changes over time. We first use Equation 3.3 to compute the temporal trend of category x for each month m .

$$TT(x, m) = \frac{\text{number of questions of category } x \text{ in month } m}{\text{number of questions in month } m} \quad (3.3)$$

Then, to see whether the observed trend is significantly increasing or decreasing, we use the Cox-Stuart test [50], a statistical method that compares earlier data points in a time series to later data points to evaluate the trend. To assess which question categories have increasing or decreasing trends, we apply a 95% statistical confidence level ($p < 0.05$). We term the temporal trend to be “Consistent” if we can not determine whether the trend is increasing or decreasing.

RQ2: What solutions do developers get for mitigating checked-in secrets?

To answer RQ2, we first provide the answer categories to mitigate the challenges related to checked-in secrets, which we determine from Step 3. Then, we provide a mapping of answer categories to each of the question categories. From the question-answer category mapping, we can understand the solutions posed by developers to mitigate a specific technical challenge.

3.2 Results

In this section, we discuss our findings and answer our research questions.

3.2.1 Answer to RQ1: What are the technical challenges faced by developers related to checked-in secrets?

We answer the four sub-research questions of RQ1 in the following sub sections.

Answer to RQ1.1: What are the questions developers ask about checked-in secrets?

We identify 27 unique question categories of 9 domains, which we present in Table 3.4 sorted based on the number of questions in a domain. The domain name, question category name, a description of the question category, and a representative example are provided for all the question categories. The number of questions in each category is indicated in parenthesis in the “Category” column.

The proportion of questions in each identified question category and the other four metrics mentioned in Section 3.1 are presented in Table 3.5. The proportion of questions, percentage of unsatisfactory answers, popularity score, Cox-Stuart test value of temporal trend of questions, and the identified trend of questions in each question category are represented in the columns “QC(%)”, “UNC(%) (Count)”, “PQ”, “Cox Stuart, p-value” and “Trend” respectively. According to Table 3.5, the top four question categories based on QC metric are “(Deployment) Store/Version”, “(Secrets) Store/Version”, “(Secrets) Ignore/Hide”, and “(VCS Feature) History Sanitize”. These four categories constitute 56.1% of all questions.

Answer to RQ1.2: Which questions related to checked-in secrets exhibit more unsatisfactory answers?

Table 3.5 shows that UNC scores of more than 40% are found in 16 of the 27 identified question categories. Our finding indicates that 44.3% of questions within our dataset have unsatisfactory answers. The top four question categories, “(Deployment) Store/Version”, “(Secrets) Store/Version”, “(Secrets) Ignore/Hide” and “(VCS Feature) History Sanitize” have UNC scores of 43.0%, 47.1%, 34.1% and 45.7% respectively.

Figure 3.1 presents the trend of unsatisfactory answers for each year between 2008 and 2021. We observe that the percentage of unsatisfactory answers shows an increasing trend. More than 50% of questions have unsatisfactory answers since 2017, thus indicating that the developers are not getting desired answers to mitigate the challenges of checked-in secrets.

Table 3.4: 27 question categories. References to all the examples and developer quotes are available online [1]

Domain	Category (Count)	Description	Example
Secrets	Q1: Store/Version (121)	We observe that the same questions of knowing the best way to store secrets have been asked for different technologies, such as ASP.NET and Python. We also observe developers asking about versioning the secrets for environments, such as development and production environments, where they do not know the consequences of storing secrets in VCS repositories.	<i>How should I store a password used by a service written in .NET?</i>
	Q2: Ignore/Hide (85)	We observe that developers are aware of the consequences of secrets presence in the source code and want to hide the secrets. As one developer stated: <i>"The credentials are hard-coded at the moment, but they should not be. What is the proper way of hiding them?"</i> . Developers also question about challenges faced in avoiding secrets from being committed to the VCS repository.	<i>Hide API keys from github public repo?</i>
	Q3: Exploitability (30)	Developers do not know whether storing a secret such as a Google API key or testing credentials in source code or a VCS repository can be exploited. For example, one developer stated: <i>"I'm making use of google API for location. Can the key be hardcoded? ... If it's sensitive, why is it sensitive and how can attackers exploit this?"</i> .	<i>Is having sensitive data in a PHP script secure?</i>
	Q4: Distribute (11)	Developers ask questions about sharing secrets with other developers so that they can run the project successfully in their environment. As one developer stated: <i>"How can I keep my API key secret, but have my project still be functional if someone clones the repo?"</i> . We observe that developers are unsure how to share secrets with specific developers without exposing them.	<i>Push to GitHub that project is still functional when the repo cloned?</i>
	Q5: Restriction (2)	We observe questions posted for restricting a specific group of developers from having access to secrets. For example, <i>"What happens if a malicious developer decides to steal the secret and use it for malicious purposes? Is there a way to store secrets such that a backend developer doesn't have direct access to the API Key?"</i> .	<i>What are ways to manage secrets in a big organisation?</i>
Deployment	Q6: Store/Version (149)	Platform as a service (PaaS), such as Heroku [51] and Google App Engine [52], are commonly used to manage applications. During deployment, the code is fetched from the repositories. We observe developers asking questions about where to store the secrets needed for deployment since secrets are not pushed in the repository. Developers want to know the secure way of versioning secrets for deployment environments. This question category is the most frequently asked.	<i>Where to store sensitive files for heroku platform?</i>
	Q7: Improper Configuration (34)	As the configuration files are ignored in the repository and source code is fetched from the repository for deployment, developers are getting exceptions due to improper configuration in the deployment server. We observe developers asking for help resolving the build and deploy-related exceptions. We observe that most of the exceptions are during Django application deployment.	<i>Azure Django App has SECRET_KEY Exception</i>

Table 3.4: 27 question categories (Continued). References to all the examples and developer quotes are available online [1]

Domain	Category (Count)	Description	Example
Deployment	Q8: Ignore/Hide (15)	During the build and deployment of an application, developers use the secrets present in the continuous integration and continuous deployment (CI/CD) scripts or the VCS repository. We observe developers asking to know the best practice of hiding the secrets from CI/CD scripts or repositories and perform successful build and deploy.	<i>Docker-Compose with Gitlab CI managing sensitive data</i>
	Q9: Dot File (3)	Developers deploy directly from VCS repositories using Git tools. They push sensitive dot files such as .git and .gitignore files that can be accessed at the website's root location. Previous research [5] has found secrets in the .gitignore file, even though the .gitignore file is designed to restrict unintended source files committing into VCS. We observe developers facing challenges restricting the dot files' access from the website's root.	<i>How to make .gitignore safe?</i>
VCS Feature	Q10: History Sanitize (81)	Developers accidentally or knowingly push sensitive information into the VCS repository. One developer stated: <i>"I am using a shared github repository to collaborate on a project ... I committed and pushed a script file containing a password which I don't want to share"</i> . The sensitive information remains in the VCS history even when removed in another commit. Developers ask questions about sanitizing the VCS history using different tools but could not use the tools properly. Rahman et al. [8] also observed developers bypassing secret scanning tools warning because of facing technical challenges of eliminating secrets completely from the VCS history.	<i>How to remove sensitive data from a file in github history?</i>
	Q11: Ignore Already Committed (14)	Knowing the exploitability of secrets present in source code, developers want to commit a default file without secrets. However, they want to untrack further local changes of the file from VCS repositories to avoid accidentally committing the local changes, and VCS does not support the functionality [53]. As a result, we observe developers ask questions about ignoring an already-committed file from VCS tracking.	<i>Stop tracking file in Git after a first commit?</i>
	Q12: Line Level Security (11)	<i>"Do any version control systems allow you to specify line level security restrictions rather than file level?"</i> stated by one developer. VCS, such as Git, only supports file-level restrictions. We observe developers wanting to mark specific lines in a file that contains secrets and tell the VCS to secure the lines to avoid exposing the secrets.	<i>hide or change value a line at git commit but not locally</i>
	Q13: Encrypt File (1)	We observe developers asking questions about if there is a way to encrypt a secrets-containing file before committing to VCS repositories.	<i>Encrypting files added to repos</i>
Configuration File	Q14: Store/Version (56)	Config files contain secrets. We observe developers face challenges storing the config files in the VCS repository since it would expose the secrets. For example, one developer stated: <i>"I'd like to version control the whole project, including config file, but I don't want to share my passwords"</i> .	<i>Preferred way to store application configurations?</i>

Table 3.4: 27 question categories (Continued). References to all the examples and developer quotes are available online [1]

Domain	Category (Count)	Description	Example
Configur- ation File	Q15: Ig- nore/Hide (32)	We observe developers asking questions about ignoring or hiding sensitive secrets-containing config files such as the web.config and database.yml files from the VCS repository. Developers also complain about the lack of documentation or suggestions the specific technology provides on ignoring config files.	<i>Protecting the sensitive files from pushing to version control?</i>
	Q16: Dis- tribute (9)	Developers face challenges sharing secrets-containing config files with other team members without exposing them publicly. For example, one developer stated: “Should I add these 2 files to versioning or do I have to distribute these files manually to other team members?”.	<i>Managing project config files in repository?</i>
	Q17: Ex- ploitabil- ity (3)	Developers place environment variables replacing secrets in the config files and want to confirm the exploitability from outside. We also observe developers placing secrets in PHP .ini files and asking about the exploitability of the secrets. For example, one developer stated: “Is better to hide somewhere .ini file and deny access via .htaccess?”.	<i>Storing sensitive info. inside .ini file is good or bad approach?</i>
	Q18: Ac- cessibility (3)	To avoid exposing secrets, developers load secrets dynamically by referencing external files in config files but get an undefined error. An example includes loading an external database settings file into a web.config file. We observe developers facing challenges in avoiding the undefined error and could not find the proper documentation.	<i>How to securely use credentials outside web.config?</i>
Pre-open Source	Q19: Cross- check (52)	We observe developers asking questions before open-sourcing their projects. The questions include should developers clean VCS history and what checklists should they run to avoid exposing secrets.	<i>OpenAuth & Open Source Projects?</i>
Client- Side Applicat- ion	Q20: Store (28)	Developers work on client-side applications without a server-side implementation and store secrets on the client-side, such as in Javascript and Android applications. Developers face challenges in storing the secrets securely as secrets can easily be exposed from the developer console or by decompiling the binary packages.	<i>Securely storing secret data in a client-side web application?</i>
	Q21: Hide (14)	One developer stated: “Using Javascript however, I don’t feel comfortable that the client secret is exposed in my code ... because if someone looks at my source they have the client_id and client_secret which makes it possible to authenticate themselves with my code”. We observe developers looking for ways to hide client-side application secrets.	<i>How do I hide API key in create-react-app?</i>
	Q22: Ex- ploitabil- ity (5)	Developers ask questions to confirm whether the implementation of keeping secrets in the client-side application code is exploitable or not. “Could I sleep at night knowing that I won’t see “Super Cool Web App Hacked, change your passwords!” all over HN and Reddit ... as a result of this implementation.” stated by one developer.	<i>In iOS, is there leak risk if I write the secret key in the code?</i>

Table 3.4: 27 question categories (Continued). References to all the examples and developer quotes are available online [1]

Domain	Category (Count)	Description	Example
Secure-ness	Q23: Private Repository (13)	One developer stated: <i>“Is it safe for me to store my Amazon S3 keys/secrets in a private Github repo? I know that it is not safe for a public repo but I am wondering if a private repo is safe?”</i> . We observe developers asking about the safety of secrets present in a private repository.	<i>Storing Amazon S3 keys in private repo</i>
	Q24: Un-pushed Branch (1)	We observe developers ask questions about the security of secrets if they do not push the secrets-containing branch to a public repository. For example, one developer stated: <i>“Is there any chance my sensitive data could end up in the remote repository index somehow?”</i> .	<i>Commit password to branch that never pushed?</i>
External Secret Management	Q25: Setup (3)	We observe developers moving secrets to external secret management services, such as HashiCorp Vault [54] and Azure Key Vault [55]. However, developers face challenges in properly setting up these hardware security modules. Examples of such questions include where to store the vault key, the feasibility of using vaults, and how to store the database connection strings in the vault.	<i>Storing DB Connection Strings in Azure Key Vault</i>
Others	Q26: Importance (2)	We observe developers asking questions about why they should keep secrets out of the VCS repository. For example, one developer stated: <i>“It seems like common knowledge that it’s a good practice to keep secrets files ... checked out of your git repository ... Why?”</i> .	<i>Why should you keep secrets out of your repository?</i>
	Q27: Decision (1)	One developer stated: <i>“Today I found what looked to be my supervisor’s password in some code in version control ... How should I handle this situation?”</i> . We observe developers being hesitant about making decisions when they find secrets in the VCS repository.	<i>What should I do when I find sensitive info in VCS?</i>

Answer to RQ1.3: Which questions are the most popular among developers related to checked-in secrets?

The popularity of each question category is presented in the “PQ” column of Table 3.5. In our study, the popularity score varies between 0.005 and 0.030. For example, a question with Score 0 and View Count 17 has a PQ score of 0.005, whereas a question with Score 12 and View Count 17847 has a PQ score of 0.030. The top three most popular question categories are “(Client-Side Application) Store”, “(Secrets) Store/Version” and “(Client-Side Application) Hide”. In Table 3.6, we also provide the question categories in descending order, sorted by PQ and UNC(%). Further observations are aided by the ranking of the 27 question categories:

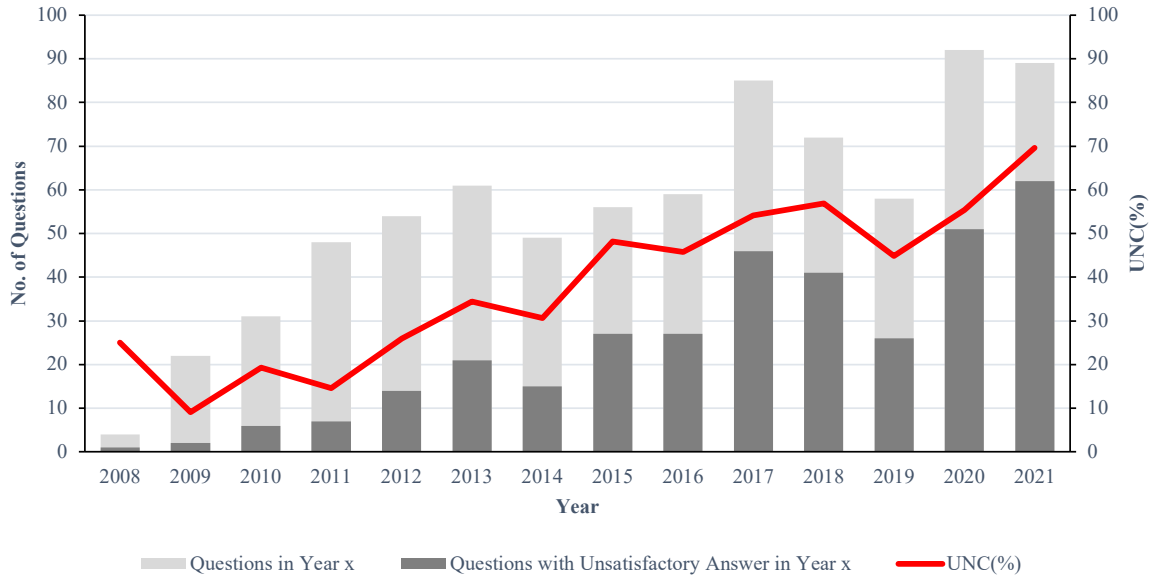


Figure 3.1: Trend of Unsatisfactory Answer Per Year

- “(Client-Side Application) Store” and “(Client-Side Application) Hide” rank first and third based on the popularity score (PQ) and have a UNC score of 60.7% and 35.7%, respectively. The observation indicates that the questions related to storing and hiding secrets in client-side applications are most popular among developers but do not receive satisfactory answers. Therefore, future research is needed on the client-side frameworks for securely managing secrets.
- “(Secrets) Store/Version” ranks second based on the popularity score (PQ) and has a UNC score of 47.1%. Our observation indicates that developers are showing more interest in the question of securely storing secrets for different technology frameworks such as ASP.NET, Ruby on Rails and Python. But, developers could not implement properly because of lacking proper documentation.
- “(Secrets) Distribute” and “(Deployment) Improper Configuration” question categories rank fourth and sixth for unsatisfactory answers, respectively. However, these question categories rank 22nd and 19th based on popularity score. Though the popularity score is low, developers are not receiving satisfactory answers for distributing secrets and fixing improper configuration errors during deployment. Therefore, future research can address secure secret distribution, and respective technology providers can provide proper documentation to fix improper configuration errors during de-

Table 3.5: Summary of identified question categories, sorted by decreasing question proportion (QC)

(Domain) Question Category	QC (%)	UNC (%) (Count)	PQ	Cox Stuart, p-value	Trend
(Deployment) Store/Version	19.2	43.0 (64)	0.020	↑, 0.11	Consistent
(Secrets) Store/Version	15.6	47.1 (57)	0.023	↑, 0.003	Increasing
(Secrets) Ignore/Hide	10.9	34.1 (29)	0.015	↑, 0.11	Consistent
(VCS Feature) History Sanitize	10.4	45.7 (37)	0.018	↑, < 0.001	Increasing
(Configuration File) Store/Version	7.2	39.3 (22)	0.022	↑, 0.5	Consistent
(Pre-open Source) Cross-check	6.7	40.4 (21)	0.010	↓, 0.3	Consistent
(Deployment) Improper Configuration	4.4	58.8 (20)	0.008	↑, < 0.001	Increasing
(Configuration File) Ignore/Hide	4.1	40.6 (13)	0.008	↓, 0.34	Consistent
(Secrets) Exploitability	3.9	56.7 (17)	0.014	↓, 0.59	Consistent
(Client-Side Application) Store	3.6	60.7 (17)	0.030	↑, 0.002	Increasing
(Deployment) Ignore/Hide	1.9	46.7 (7)	0.010	↑, 0.09	Consistent
(VCS Feature) Ignore Already Committed	1.8	28.6 (4)	0.007	↑, 0.29	Consistent
(Client-Side Application) Hide	1.8	35.7 (5)	0.022	↑, 0.13	Consistent
(Secureness) Private Repository	1.7	46.2 (6)	0.014	↑, 0.27	Consistent
(Secrets) Distribute	1.4	63.6 (7)	0.007	↑, 0.11	Consistent
(VCS Feature) Line Level Security	1.4	36.4 (4)	0.007	↓, 0.5	Consistent
(Configuration File) Distribute	1.2	66.7 (6)	0.007	↑, 0.14	Consistent
(Client-Side Application) Exploitability	0.6	40.0 (2)	0.008	↑, 0.5	Consistent
(Configuration File) Exploitability	0.4	0.0 (0)	0.012	↑, 0.5	Consistent
(Configuration File) Accessibility	0.4	33.3 (1)	0.008	↑, 0.13	Consistent
(Deployment) Dot File	0.4	33.3 (1)	0.007	↑, 0.5	Consistent
(External Secret Management) Setup	0.4	66.7 (2)	0.015	↑, 0.13	Consistent
(Others) Importance	0.3	100.0 (2)	0.005	↑, 0.25	Consistent
(Secrets) Restriction	0.3	50.0 (1)	0.007	↓, 0.75	Consistent
(VCS Feature) Encrypt File	0.1	0.0 (0)	0.008	↓, 0.5	Consistent
(Secureness) Unpushed Branch	0.1	0.0 (0)	0.005	↓, 0.5	Consistent
(Others) Decision	0.1	0.0 (0)	0.008	↓, 0.5	Consistent

ployment.

- We observe developers searching for VCS features to ignore the tracking of already-committed files to avoid local changes being accidentally committed in the VCS repository. An option exists to delete the file from remote repository and then ignore the file by placing the file name in the `.gitignore` file. However, developers do not want to delete and want a copy of the file in the remote repository, which VCS does not support [56]. Developers are also looking for line-level restrictions in VCS to hide secrets in particular lines of the source code. Though VCS has a feature called `git smudge-clean` [57] which can be used to replace a secret with a dummy value during commits, developers face difficulties in implementing the process. Despite “(VCS

Table 3.6: Ranked Order of Question Categories Based on Popularity (PQ) and Unsatisfactory Answer Percentage (UNC)

Metric	(Domain) Question Category (Sorted in decreasing order of metric)
PQ	(Client-Side Application) Store, (Secrets) Store/Version, (Client-Side Application) Hide, (Configuration File) Store/Version, (Deployment) Store/Version, (VCS Feature) History Sanitize, (Secrets) Ignore/Hide, (External Secret Management) Setup, (Secureness) Private Repository, (Secrets) Exploitability, (Configuration File) Exploitability, (Pre-open Source) Cross-check, (Deployment) Ignore/Hide, (VCS Feature) Encrypt File, (Configuration File) Accessibility, (Others) Decision, (Configuration File) Ignore/Hide, (Client-Side Application) Exploitability, (Deployment) Improper Configuration, (Deployment) Dot File, (Secrets) Restriction, (Secrets) Distribute, (Configuration File) Distribute, (VCS Feature) Line Level Security, (VCS Feature) Ignore Already Committed, (Others) Importance, (Secureness) Unpushed Branch
UNC (%)	(Others) Importance, (Configuration File) Distribute, (External Secret Management) Setup, (Secrets) Distribute, (Client-Side Application) Store, (Deployment) Improper Configuration, (Secrets) Exploitability, (Secrets) Restriction, (Secrets) Store/Version, (Deployment) Ignore/Hide, (Secureness) Private Repository, (VCS Feature) History Sanitize, (Deployment) Store/Version, (Configuration File) Ignore/Hide, (Pre-open Source) Cross-check, (Client-Side Application) Exploitability, (Configuration File) Store/Version, (VCS Feature) Line Level Security, (Client-Side Application) Hide, (Secret) Ignore/Hide, (Configuration File) Accessibility, (Deployment) Dot File, (VCS Feature) Ignore Already Committed, (Configuration File) Exploitability, (Others) Decision, (Secureness) Unpushed Branch, (VCS Feature) Encrypt File

Feature) Ignore Already Committed” and “(VCS Feature) Line Level Security” ranking 25th and 24th, respectively, based on popularity score, the two question categories consist of 25 questions where developers are seeking the new VCS feature.

Answer to RQ1.4: How do question categories related to checked-in secrets trend over time?

Figure 3.2 depicts the temporal trend of 15 question categories that have at least 10 questions. For each category, the figure provides a scatter plot with a smoothing plot with the trends highlighted. We can understand whether the trend of each question category is increasing, decreasing, or consistent from the “Cox Stuart”, “p-value” and “Trend” columns of Table 3.5. Table 3.5 highlights the question categories with a p-value less than 0.05 in grey.

From Table 3.5, we observe an increasing trend in four question categories. While only four question categories showed increases, the trend is across 13 years of the data. We also observe that developers are posting more questions in “(Secrets) Store/Version”, “(VCS Feature) History Sanitize”, “(Deployment) Improper Configuration”, and “(Client-Side Application) Store” categories, but their questions are not well-answered. The four question categories have a UNC score of more than 45%, and three out of four question categories

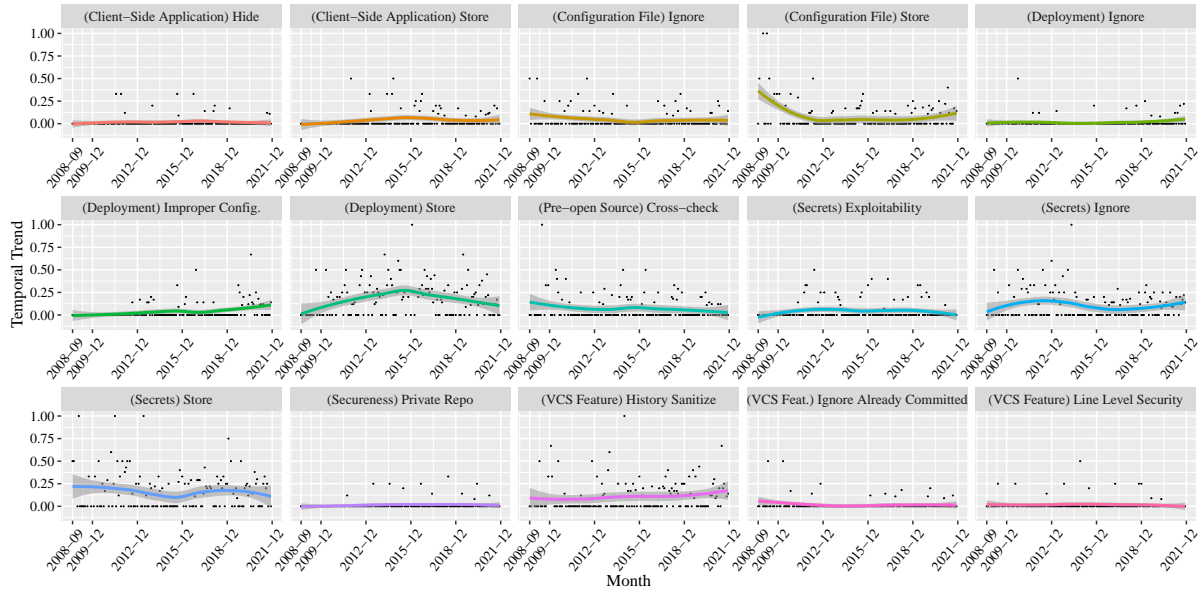


Figure 3.2: Temporal Trend of each identified question category. The month of the x-axis is shown in three-year interval. The zero value of Temporal Trend indicates no question is posted on the specific month for a category.

are also in the top six categories based on the popularity score (PQ). The increasing trend of these four question categories substantiates the absence of proper documentation on managing secrets during the deployment and the need for future research on client-side frameworks. In addition, the increasing trend also substantiates the need to improve existing VCS history sanitizing tools to make integration easier for developers.

3.2.2 Answer to RQ2: What solutions do developers get for mitigating checked-in secrets?

We identify 13 answer categories from our analysis, which we present below based on the descending order of the number of questions in which StackExchange users suggest the specific answer category. For example, 179 answers to the 779 questions suggest the ‘A1: Move Secrets out of Source Code/Version Control and Use Template Config File’ category. We do not declare all the answer categories as best practices. Indeed, below we highlight the shortcomings of these answer categories as appropriate.

A1: Move Secrets out of Source Code/Version Control and Use Template Config File (179): Developers may put secrets, such as database credentials, in a file where the code

for database functionalities are present. As a result, developers face challenges in hiding the credentials from VCS repositories. In such cases, developers are suggested to move the secrets to a config file. Then, the config file with original secrets should be ignored from the VCS repository, and a template config file should be committed to the repository. Template config files, such as `database.sample.yml` file of Ruby on Rails, contain the minimum configurations with dummy secrets to avoid build failure. Developers will replace the dummy secrets in their development environment. Furthermore, a `.gitignore` file should be included with all repositories to ignore the secrets-containing files. GitHub has published a collection of `.gitignore` templates [58] for different technologies.

A2: Secret Management in Deployment (78): We observe that developers mostly face challenges storing or versioning secrets for multiple environments during deployment. Configuration management systems, such as Ansible-Vault [59] and Chef-Vault [60], provide support for secret management. Developers are advised to use deployment variables, such as Heroku Config Vars [61], which create environment variables for respective environments. Developers are also suggested to keep the dot files such as `.git` and `.hg` files out of the root directory during deployment to avoid exposing secrets.

A3: Use Local Environment Variables (56): An environment variable is a dynamic object which is set outside of the application and used to avoid the storage of secrets in code or local config files. Developers are suggested to use environment variables to load the secrets at runtime. The benefits of using environment variables are switching secrets between deployed versions without modifying any code and making it less likely that secrets get checked into the repository. However, environment variables can leak secrets as they are passed down to child processes, which allows for unintended access [62].

A4: Rewrite VCS History (48): Secrets will not be removed entirely by removing in another commit as secrets will remain in the VCS history. Developers suggest removing secrets using `git-filter-repo` [63], `git-filter-branch` [64], and BFG repo cleaner [65]. Though official GitHub documentation [66] suggests using BFG repo cleaner instead of `git-filter-repo` and `git-filter-branch`, we have seen Stack Exchange users mostly suggest using the latter. GitHub has also suggested contacting them with the repository name to clear the secrets from their cache and advised to tell the project collaborators to do `git rebase` instead of `git merge` [66] though no Stack Exchange users' solutions suggested these actions.

A5: Store Encrypted/Obfuscated Secrets (39): Storing secrets as encrypted, encoded, or obfuscated is one of the solutions suggested by Stack Exchange users. Different encryption algorithms, such as AES and RSA, are suggested. In some cases, developers are suggested to encode secrets using Base64 encoding in Android applications. Another suggestion is to

split the secrets into multiple parts and keep them in the source code. The number of parts should be high, so the attacker will have to check for more than a billion permutations. Tools such as git-secret [67] and git-crypt [68], are available for encrypting secrets-containing files. The disadvantage of encryption is to deal with the encryption keys securely.

A6: Use of External Secret Management Service (26): Developers are recommended to implement external secret management services, such as HashiCorp Vault [54] and AWS KMS [69]. These hardware security modules can safely store secrets with tightly-controlled access. However, because they are challenging to set up and maintain, these solutions may be unsuitable in some situations. In addition, they need a significant investment of time and money.

A7: Load Externally and Use Secondary Private Repository (23): Since developers want to avoid committing secrets into VCS that are needed for the application's functioning, developers are advised to load secrets externally using AWS S3 or a secondary private repository. Since AWS S3 needs access keys to retrieve stored files, the same problem of storing the access keys may occur. A secondary private repository can be used to store secrets and loaded dynamically using git submodules [70]. However, private repositories are not free from exploitation by attackers [71].

A8: Revocation and Rotation (16): The first step to stop secrets sprawl is to revoke the secrets immediately. One developer suggested: *"The important bit: Consider your credentials compromised. Change them. No matter what you do at this point, they are no longer secure"* [1]. A good practice is to rotate the secrets periodically. Short-lived secrets prevent previously-undetected data breaches from posing a threat, as access will be cut off even if the breach is not identified.

A9: Server-Side Implementation (16): To avoid keeping secrets in client-side applications for fetching data from web services, developers are recommended to implement web service functionality on the server side. Then, the server will use the secrets and fetch data for the client side, thus removing the necessity to keep secrets in client-side applications.

A10: VCS Feature (Git Hooks and Flags) (10): To avoid secrets from pushing in VCS repositories, developers are suggested to implement git hooks [72] and git flags [73, 74]. The pre-commit and post-commit hooks can be used to filter and smudge before commit or after pull, respectively [57]. However, developers are warned as implementing git hooks properly is difficult. Developers are also suggested to use the git flags such as `-skip-worktree` [74] and `-assume-unchanged` [73] to prevent changes from being committed to existing files.

A11: Add Files to the Staging Area Explicitly (3): A simple strategy to avoid exposing secrets accidentally is to add files explicitly in the VCS staging area. Developers are suggested

to avoid using wildcards (`git add -A` or `git add *`) for adding files, thus having complete control and visibility over what files are committed.

A12: Restrict API Access and Permissions (3): Since attackers frequently use secrets within their scope, detecting when they are doing so maliciously might be challenging. However, damage and lateral movement can be limited by restricting access and permissions of the secrets. For example, GitHub IP white-listing [75] can be employed to prevent any untrusted sources from accessing the GitHub repositories.

A13: VCS Scan Tools (1): Developers are advised to run VCS scan tools, such as TruffleHog [14] and Gitrob [76], before any commit or in an existing repository to find out the presence of secrets. The tools can find secrets buried in histories that manual searches and reviews will miss. However, tools may return a significant number of false positives [8].

The mapping of answers to each question category is provided in Appendix A.1. We observe that the same answer category has been mentioned to mitigate challenges of multiple question categories. For example, 'A1: Move Secrets out of Source Code/Version Control and Use Template Config File', 'A3: Use Local Environment Variables' and 'A2: Secret Management in Deployment' have been mentioned as part of a solution in 20, 12, and 10 out of 27 question categories, respectively.

3.3 Discussion and Recommendations

In this section, we discuss our findings and make recommendations. In our discussion, we trace the questions and answers by their identifiers assigned in Table 3.4 and Section 3.2.2, respectively.

Tool enhancement. We find that developers face difficulty with properly sanitizing VCS history (Q10). Developers commonly use `git-filter-branch` [64] and `git-filter-repo` [63] to sanitize VCS history. However, both the tools have safety and usability issues which can easily corrupt the repository's history [77]. For example, these tools can easily mix up the old and new history of the repository. In addition, coming up with the correct shell script is difficult as developers find out if the sanitizing code script is right or wrong by trying the script out. Even worse, broken filters often result in silent incorrect rewrites without proper output. Even if the developers sanitize the VCS history properly using the tools, the tools can not clear the cache in the respective version control systems, such as GitHub, as the sensitive information can appear again from the cache, according to GitHub's official documentation[3]. As of now, clearing from the cache is a manual process that can be

automated.

In addition, we observe that developers are suggested to use VCS scan tools (A13) to avoid accidentally committing secrets, but developers seem to bypass scan tool warnings due to high false positives [8]. There are currently many open-source and proprietary VCS scan tools [78], but developers find it challenging to choose one tool out of many. Researchers and tool developers can work on comparing the effectiveness and efficiency of the VCS scan tools and improving the tools by reducing false positive warnings.

We also found that developers want new VCS features, such as line-level security, where developers can quickly point to the specific lines to which they want to restrict visibility in the VCS (Q12). In addition, we found that developers want to ignore local changes of already-committed files from VCS tracking without removing the file from the repository (Q11). Though Stack Exchange users suggested using `--assume-unchanged` [73] and `--skip-worktree` [74] flags to ignore local changes of already-committed files from VCS tracking (A10), the official Git documentation suggests these flags not be used [53].

Recommendation 1: We recommend improving the existing tools, such as making the integration of VCS history sanitizing tools easy for the developers and reducing VCS scan tool false positives. We also recommend developing new tools for line-level security and ignoring local changes of already-committed files.

Documentation. We find that developers face challenges in securely managing secrets while developing with different technologies due to the absence of proper documentation (Q1, Q6-Q8). For example, Foursquare API documentation [79] suggests developers use a client secret in useless or server-side authentication. However, a developer did not understand the documentation and asked in Stack Exchange whether the secret could be used in the client-side authentication [80]. Developers also seem to query to understand the safest approach when multiple approaches are suggested in the same documentation [81]. For example, ASP.NET Core documentation suggests using local environment variables and secret manager tools to store secrets securely but does not specify which one will be the safest approach in specific use cases [82]. However, we agree that no solution will be perfectly secure, but the documentation should be clear and detailed so that developers understand which use cases are appropriate for each approach. Furthermore, we observe that developers want reference links on how to implement a specific approach suggested in the documentation. For example, Google API provides documentation of the best practices for securely using API keys [83]. However, developers could not figure out how to implement

these suggestions as reference links to the specific suggestions are not given [84]. We also observe that documentation does not explicitly mention whether the particular suggestion, such as setting up continuous deployment in Azure Function, is for the development or production environment [85]. As a result, developers may implement a suggestion in the production environment that was intended for use in the development environment [86], thus exposing secrets to the attackers.

Recommendation 2: We recommend that each technology improve the technical documentation for managing secrets by i) clearly explaining the suggested approach's use cases and restrictions; ii) mentioning which approach will be safest for specific use cases when multiple approaches are suggested; iii) providing reference links to implement the suggested approaches; and iv) explicitly mentioning whether the particular approach is for development, production, or both environments.

Client-side applications. Often, developers architect applications with only a client-side implementation and only later realize they must securely embed a secret in the code they distribute. As a result, questions about client-side secret storage (Q20), were the most popular among all topics we studied, as seen in Table 3.5. One solution is for the developer to operate an API for their app that wraps the third-party API and keeps the secret server-side. Instead, novice developers embed third-party API calls in the client because it seems easier, cheaper (no infrastructure costs), and functions as expected. Unfortunately, secrets in the client-side application can not be protected against even a basic adversary with access to a debugger or decompiler. Inspired by popular DRM schemes such as Apple's FairPlay Streaming [87], we posit that privileged system elements, such as virtual machines, runtimes, browsers, or kernels could provide an interface for secure secret management.

Recommendation 3: We recommend that kernels and privileged runtimes develop frameworks to provide secure secret management for client-only applications.

Guidelines. From the identified challenges in Table 3.4, we observe that developers have a knowledge gap about whether a secret is exploitable or not (Q3), why they should keep secrets out of VCS (Q26), and what to do if they find secrets in the source code (Q27). We also found that some solutions are insecure for managing secrets by analyzing the solutions posed by Stack Exchange users. For example, storing secrets as Base64 encoded in the source code can be exploitable as secrets can be decoded easily (A5). Furthermore, storing secrets in a private repository is not a safe approach (A7) as private repositories are not

free from exploitation by attackers or insider threats [71, 88]. Therefore, a guideline to train developers on securely managing secrets can eliminate the knowledge gap, and developers can make correct decisions during development. The National Institute of Standards and Technology (NIST) [89] provides a framework SP 800-218 [90] to mitigate the risk of software vulnerabilities but does not have practices specific to securely managing secrets.

Recommendation 4: We recommend that NIST update the SP 800-218 framework by including practices specific to securely managing secrets to train developers.

3.4 Ethics

The contents of all the Stack Exchange sites are under Creative Commons (CC BY-SA 3.0) license [91] with the following requirements: “You are free to: *Share* - copy and redistribute the material in any medium or format, *Adapt* - remix, transform, and build upon the material for any purpose, even commercially” [91]. Stack Exchange inspires academics to utilize the data in research articles [92] and requires researchers to give attribution to posts using a direct link [93]. As a result, we include hyperlinks to connect our quotes to the original posts, which are available online [1].

3.5 Threats to Validity

In this section, we discuss the limitations of our study.

Q&A Site Selection: We did not collect questions from other Q&A sites, such as CodeProject [94] and Coderanch [95]. We accounted for this limitation by considering three Q&A sites of Stack Exchange instead of only using Stack Overflow.

Manual Analysis Bias: Caused by multiple interpretations and oversight, the manual analysis may induce bias. For example, the identified question and answer categories are susceptible to bias. We mitigated this bias by cross-checking the obtained question and answer categories and adding question and answer categories that both participants agreed on.

Closed Questions: The nature of inquiries about checked-in secrets in software artifacts may be broad, and Stack Exchange moderators do not like such questions. As a result, the moderators may decide to close some of the important questions. However, only 52 questions were flagged as closed, accounting for less than 7% of the 779 questions in our

dataset. We also observed that the closed questions had a high View Count (as high as 49471) and high Score (as high as 126) [96]. As a result, we claim that the closed questions of our dataset have remained active after being closed, proving the significance of the topics under discussion.

Popularity Metric: We measured the popularity metric of a question by taking the question's View Count and Score values into account. On the other hand, this metric may be biased because it ignores the time span of the views. Therefore, a new question with a low View Count and Score value may be regarded as unpopular. Also, Stack Exchange does not provide the temporal View Count of a question. As a result, a significant percentage of the View Count may accumulate when the question is initially posted or may have recently increased. Unfortunately, we have not yet arrived at a suitable treatment for this threat.

Counting Questions: We counted questions of a category posted by developers over time to find if a particular question category trends. There can be questions in that specific category that have been answered before, but developers are still posting new questions. It implies that the particular category continues to be a problem despite the ongoing effort. We agree that there can be a trend of decreasing questions of a category, but the problem may not be solved till today. However, we are not claiming those categories as of less importance. Instead, we are highlighting the recent ongoing problematic topics to the research community so that researchers can prioritize the challenges and work on resolving them.

Accepted Answer: We termed a question lacking an accepted answer as a *question with unsatisfactory answer*. However, a developer who posted the question may be satisfied with the suggested solution posted by Stack Exchange users. Nevertheless, the developer may forget or not know how to mark the suggested solution as accepted in Stack Exchange. Unfortunately, we have not yet arrived at a suitable treatment for this threat.

3.6 Conclusion

Software relies heavily on the use of secrets for authentication and authorization, and the exposure of secrets is increasing each day. By analyzing the questions developers ask, we can understand the challenges developers face regarding checked-in secrets. In our empirical study, we studied 779 questions posted on Stack Exchange to investigate the challenges faced by developers and the corresponding solutions posed by others to mitigate the challenges. We identified 27 challenges and 13 solutions. The four most common

challenges, in ranked order, are: (i) store/version of secrets during deployment (Q6); (ii) store/version of secrets in source code (Q1); (iii) ignore/hide of secrets in source code (Q2); and (iv) sanitize VCS history (Q10). The three most common solutions, in ranked order, are: (i) move secrets out of source code/version control and use template config file (A1); (ii) secret management in deployment (A2); and (iii) use local environment variables (A3). In addition, we observe that the same solution has been mentioned to mitigate multiple challenges. We also observe an increasing trend in questions lacking accepted answers. Our findings will benefit researchers and tool developers who can investigate how the secret management process can be enhanced to facilitate secure development.

CHAPTER

4

STUDY II: WHAT ARE THE PRACTICES FOR SECRET MANAGEMENT IN SOFTWARE ARTIFACTS?

The presence of software secrets in VCS repositories necessitates the integration of adequate secret management practices for secure development. However, such integration may be difficult due to the lack of a comprehensive set of practices related to managing secrets. For example, developers seem to query online forums to find the best practices for storing secrets [97]. Secret management practices can be derived systematically to help practitioners in limiting the exposure of secrets. In addition, practitioners can utilize the derived set of practices as a comparison point for their existing secret management practices.

Analyzing Internet artifacts, such as blog articles and online forum question and answer (Q&A) posts, is one way to derive secret management practices in software artifacts. In previous studies [24, 98], the importance of Internet artifacts has also been recognized in determining security practices.

The goal of our study is to aid practitioners in avoiding the exposure of secrets by identi-

fying secret management practices in software artifacts through a systematic derivation of practices disseminated in Internet artifacts.

We answer the following research question: **RQ: What are the practices used by practitioners for secret management in software artifacts?**

We conducted a grey literature review [99] and collected 54 Internet artifacts, such as blog articles and Q&A posts. From the collected Internet artifacts, we conducted a qualitative analysis approach called open coding [100] and determined practices that are specific to secret management in software artifacts.

Our contribution is a set of practices that practitioners can follow to avoid exposure of secrets in software artifacts.

The rest of the chapter is structured as follows: The methodology used in our work is described in Section 4.1. We provide our findings in Section 4.2. The implications and limitations of our study are addressed in Section 4.3. Finally, Section 4.4 draws the conclusion of our study.

4.1 Methodology

To identify practices used by practitioners for secret management in software artifacts, two authors conduct a grey literature review independently. Grey literature is defined as “... literature that is not formally published in sources, such as books or journal articles” [99]. A grey literature review differs from a systematic literature review (SLR) or a systematic mapping study (SMS) as researchers leverage peer-reviewed literature indexed in scholar databases in the case of SLR or SMS. Grey literature review, on the other hand, makes use of non peer-reviewed artifacts such as online videos, blog articles, and Q&A posts that are available on the Internet [17]. Grey literature provides better coverage of emerging research topics [18, 24]. We are inspired by academics who have analyzed Internet artifacts to determine security practices that can be used in the software development process [24, 98]. We hypothesize that by collecting and analyzing Internet artifacts systematically, we can find practices for secret management in software artifacts.

Figure 4.1 shows a summary of our grey literature review methodology. The following is a breakdown of each stage in our methodology.

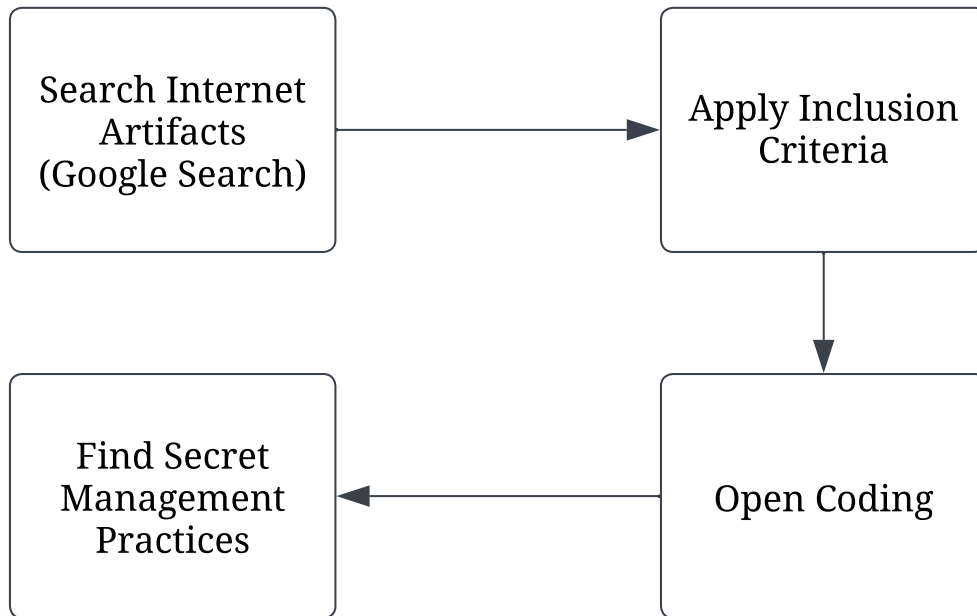


Figure 4.1: An overview of our grey literature review methodology.

4.1.1 Search Internet Artifacts

Using a set of search strings, we collect Internet artifacts. As our research study focuses on the practices for managing secrets in software artifacts, we begin with the search string “practice for managing secrets in source code”. Next, we choose the top 100 results determined by Google search engine’s page rank algorithm as a search stopping criteria [18] and collect the results. We observe practitioners referring to “secrets” as “credentials”, “passwords” and “sensitive information” based on a manual examination of the 100 search results. Based on the observations above, we include these keywords as part of the search construction process and conduct our search procedure using four search strings, which are stated as follows:

- practice for managing secrets in source code
- practice for managing credentials in source code
- practice for managing passwords in source code
- practice for managing sensitive information in source code

Altogether, we collect 400 Internet artifacts, 100 for each of the four search strings. To avoid a conflict with the authors' browsing history, we search in incognito mode of the Google Chrome browser.

4.1.2 Apply Inclusion Criteria

To find Internet artifacts relevant to our research study, we use the following inclusion criteria:

- The artifact is not a duplicate of another artifact;
- The artifact is available for reading;
- The artifact is written in English; and
- The artifact discusses at least one practice for secret management in software artifacts

We determine 54 Internet artifacts after applying the inclusion criteria. Three Q&A posts and 51 blog articles comprise our collection of 54 Internet artifacts. Figure 4.2 depicts a detailed breakdown of our filtering procedure.

4.1.3 Find Secret Management Practices

We apply open coding [100] to our collected grey literature artifacts. Open coding is a qualitative analysis technique that can reveal the underlying theme from unstructured textual information [100]. Open coding is widely utilized to identify patterns from Internet artifacts [24, 101]. The first and second authors review each Internet artifact and extract the stated practices as part of the open coding process. After the first and second authors finish their open coding individually, the identified practices are cross-checked by both authors. We use a negotiated agreement [46] to resolve the disagreed-upon practices. Negotiated agreement is an approach to discuss the disagreements among the raters in an effort to resolve disagreements when two or more raters code the same artifacts [46]. We resolve disagreements either by discarding practices that are not suitable for managing secrets or combining similar practices into one practice. We group each identified practice into a category that solves a specific issue of secret management, such as developer practices for avoiding accidental secrets commit or organizational practices to enforce policies for secrets protection.

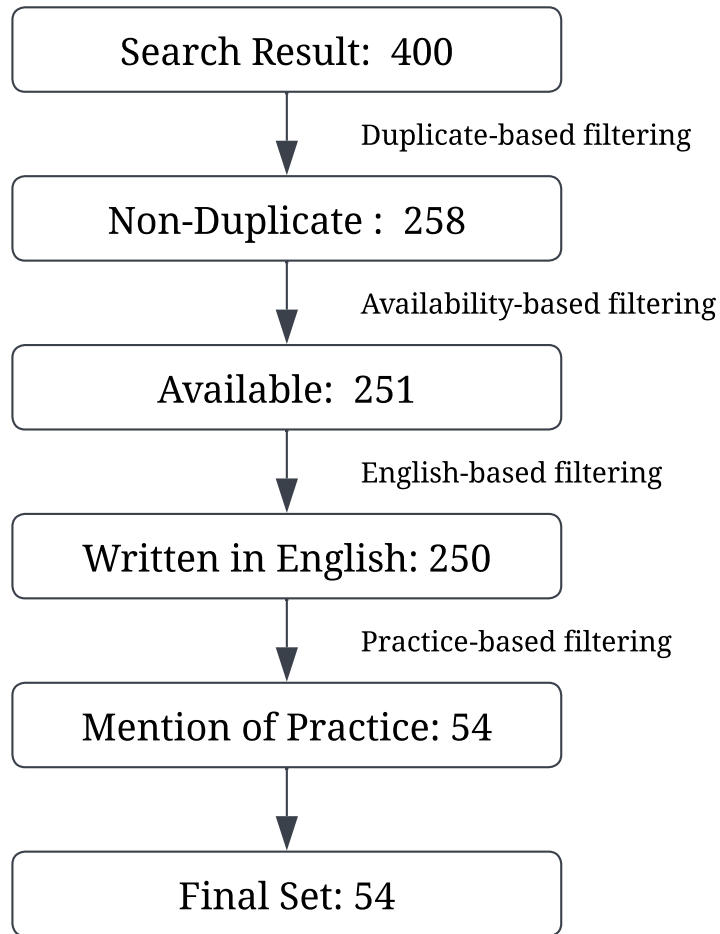


Figure 4.2: Application of inclusion criteria on our grey literature dataset to collect the set of 54 Internet artifacts for our study. Grey literature dataset is available online [3].

4.2 Results

Based on our findings, we identified 24 practices classified into six categories. In the following subsections, we provide the details of the identified practices of each of the six categories and the number of Internet artifacts that discuss the practices. For example, the ‘Practices for Keeping Secrets Out of Source Code (OSC)’ category has four practices, and 18 of the 54 Internet artifacts mention the ‘OSC-1: Use Local Environment Variables’ practice.

4.2.1 Practices for Keeping Secrets Out of Source Code (OSC)

Beyond allowing developers to set permissions on their accounts, VCS does not ensure the security of secrets to remain in a secured and controlled environment. Practitioners recommend the below four practices to keep secrets out of source code or VCS repositories.

OSC-1: Use Local Environment Variables (18)

Local environment variables, which are dynamic objects defined outside of the application and used to avoid storing secrets in VCS or configuration (config) files, are recommended by practitioners. The Twelve-Factor App methodology [102] is a set of 12 language-agnostic guidelines for building software-as-a-service applications with portability and resilience. The third factor, *Config principle* of the Twelve-Factor App methodology, also states that config information should be kept as environment variables and injected into the application at runtime [102]. Libraries, such as `dotenv` [103], can be used to load variables into the running process. Without modifying any code, environment variables can easily be changed between released versions. On the other hand, practitioners also advise avoiding local environment variables for client-side applications as secrets can be extracted using the browser dev tools [104].

OSC-2: Move Secrets to Configuration File (15)

Practitioners recommend moving secrets to external config files such as `web.config` and `config.yaml` files. Config files are environment-specific and can be updated at any time without redeploying the application, as the lifecycle is independent of the application. Instead of checking the original config file into VCS, developers are suggested to add a template config file. The template config files such as `database.sample.yaml` file of Ruby on Rails and `web.default.config` file of ASP.NET will contain minimum configurations with dummy values which developers will replace in their development environment. Using template config files reduces the chance of secrets being checked into VCS, thus preventing potential secret exposure.

OSC-3: Ignore Sensitive Files (11)

Practitioners recommend to avoid committing sensitive files, such as `.env` and `.config` files, into VCS repositories. Even a `.DS_Store` (a hidden file present in every folder on an OS X system) can leak the names of the files and folders present on a web server. A search on

GitHub for `.DS_Store` returns more than 800K results [105]. To avoid committing sensitive files, all repositories should include a `.gitignore` file. GitHub has published a collection of useful `.gitignore` templates for different technologies [58].

OSC-4: Add Server-Side Implementation for Client-Side Applications (2)

Secrets present in client-side applications, such as Javascript and Android applications, can be exposed by the developer console or by decompiling the binary files (APK or iPA files) [106]. To avoid keeping secrets in client-side applications for fetching data from different web services, practitioners recommend implementing the web service functionality on the server-side. The server will use the appropriate secrets and fetch data for the client-side, thus removing the necessity to keep secrets in client-side applications.

4.2.2 Practices for Securely Storing Secrets (SSC)

Developers can store secrets insecurely in source code or VCS repositories. Practitioners recommend the following three practices to store secrets securely.

SSC-1: Use External Secret Management Systems (28)

As emails can be forwarded and messaging applications can be hacked, practitioners recommend to avoid sending secrets through emails or any messaging applications, such as Microsoft Teams or Slack. Only one compromised account is enough to expose sensitive data. When secrets are exchanged through internal networks, bad actors can use secrets to migrate laterally between services. Instead, practitioners recommend to use external secret management systems, such as HashiCorp Vault [54], AWS KMS [69] and Knox [107]. These hardware security modules can safely store secrets with tightly-controlled access. Developers can be assigned to groups based on the teams they work on, and secrets can be shared with the groups by granting proper permissions. If any developer switches teams or leaves the company, the secrets used by the developer can easily be invalidated using external secret management systems. The ability to set up dynamic secrets, lease-based secret management (limiting access for a fixed period before automatic revocation), and audit trails, which allow administrators to check for any breaches, are other important features. The ability to rotate secrets over time by giving specific states for secret versions is a unique feature of Knox that is not found in other systems. A secret version can be tagged as 'primary' to denote that the secret is the current recommended, 'active' to denote that

the secret is still usable, or 'inactive' to denote that the secret is disabled. Administrators can use this mechanism to roll secrets across machines without impacting the service. External secret management systems minimize human involvement in creating, distributing, and maintaining secrets. Practitioners have recommended SSC-1 practice the most among all the practices for managing secrets, though a significant investment of time and money is needed.

SSC-2: Store Encrypted Secrets (14)

Practitioners recommend avoiding Base64 encoding of secrets as encoded secrets can be decoded easily. Instead, for a project having a single developer or a small team, practitioners suggest encrypting secrets-containing files in VCS. Several tools such as `git-crypt` [68] and `git-secret` [67] which use GPG to encrypt content are available for encrypting sensitive files containing secrets. Technologies, such as Ruby on Rails, starting with Version 5.1, have included built-in capabilities to encrypt secrets with VCS [108]. Though developers have to manage encryption keys securely (keep out of VCS) and no role-based access control of secrets is present, the benefit of using encryption tools is that the implementation does not need additional infrastructure.

SSC-3: Private Repositories Are Not Safe (2)

One practitioner stated: “*A secret in a private repo is like a password written on a \$20 bill, you might trust the person you gave it to, but that bill can end up in hundreds of peoples hands as a part of multiple transactions and within multiple cash registers*” [109]. Since repositories can be forked into new projects and cloned onto new machines, secrets present in the history of a repository will be propagated to the forked and cloned repositories. Only one compromised developer account or a misconfiguration will be enough to get access to all secrets present in private repositories. For example, in 2021, a repository misconfiguration of setting the default username and password combination of `admin/admin` resulted in Nissan source code being exposed online [88].

4.2.3 Practices to Limit Secrets Exposure (LSE)

Practitioners recommend below four practices to limit the exposure of secrets.

LSE-1: Use Short-lived Secrets (15)

Short-lived secrets, according to practitioners, prevent previously-undetected data breaches from becoming a threat by terminating access even if the breach is not identified. If a validity period cannot be assigned to secrets, practitioners advise revoking and redistributing the secrets periodically. Practitioners also suggest rotating and redistributing the secrets correctly to avoid any failure. For example, in 2021, Microsoft Azure experienced a 14-hour downtime due to an error in secret rotation used for authentication [110].

LSE-2: Restrict API Access and Permissions (8)

Because attackers frequently use secrets within their scope, detecting when they are doing so maliciously may be challenging. Practitioners suggest that damage and lateral movement can be limited by restricting access and permissions to secrets. For example, a leaked AWS S3 key, which had the permission to spin up AWS EC2 instances, resulted in a \$6000 bill overnight as an attacker spun up 140 instances [111]. IP white-listing adds another degree of protection against attackers who try to exploit APIs maliciously. For example, GitHub IP white-listing [75] can be employed to restrict insecure sources from accessing the repositories. External service users can set API key usage restrictions by making API keys accessible from specific URLs. The key will be useless to attackers if attackers cannot invoke the service from the allowed URLs. A daily limit on API key usage should also be set to avoid bill spikes.

LSE-3: Revoke Secrets and Sanitize VCS History (6)

Secrets will not be removed entirely by removing them in another commit, as secrets will remain in the VCS history. Practitioners advise sanitizing VCS history in two steps. The first step is to revoke the secrets present in the code. The second step is to purge and rewrite the VCS history using tools such as `git-filter-branch` [64], `BFG repo cleaner` [65], or `git-filter-repo` [63]. GitHub documentation [66] suggests using `BFG repo cleaner` instead of other tools. To avoid anomalies, the best practice is to close all pull requests before scanning VCS history using the tools. GitHub suggests contacting them with the repository name to clear the secrets from their cache and advised to tell the project collaborators to do `git rebase` instead of `git merge` as merge can introduce some of the tainted history [66].

LSE-4: Audit All Code Uploaded to VCS and Review VCS Audit Logs for Suspicious Activity (4)

Practitioners recommend auditing all code uploaded to VCS on a regular basis. For example, legacy code may be used as part of an organization's new software. The problem with integrating legacy code is that what was once secure might not be anymore, as secrets may be present in legacy code. Therefore, auditing any code uploaded to VCS will be advantageous for the software's long-term integrity, even if the procedure is time-consuming. The administrator of an organization can also review the activities of other team members using the audit log feature of VCS. Suspicious activities can be flagged and tracked by constructing a trace profile based on the user's activity, the action's location, and the time of the event. For example, GitLab [112] provide the archive of audit logs where the admin can search for events between any period or any specific user action.

4.2.4 Practices for Avoiding Accidental Secrets Commit (ASC)

Developers can accidentally push secrets into VCS repositories. Practitioners recommend the below three practices to avoid accidental committing of secrets.

ASC-1: Use VCS Scan Tools (16)

Though code reviews can detect logical flaws and maintain coding practices, practitioners do not recommend relying on code reviews to detect secrets. If secrets are added in one commit and removed in another commit, the net difference in code changes will be zero. The reviewer only sees the net difference, but secrets will remain in the VCS history, thus allowing an attacker to find secrets from the prior revisions. Practitioners recommend running VCS scan tools such as TruffleHog [14], Gitrob [76] and `git-all-secrets` [113] in the VCS repository to find out the presence of secrets. Via a `pre-commit` hook, VCS scan tools can reject any commit containing secrets that manual searches and reviews will miss. VCS scan tools can also find secrets buried in logs and histories. VCS scan tools are also recommended to use with continuous integration or continuous deployment (CI/CD) pipelines to actively break build/deploy when secrets are found in source code. Practitioners also mention that VCS scan tools will return a lot of false positives, which developers will need to filter manually [8].

ASC-2: Add Files to the Staging Area Explicitly (5)

One simple practice suggested by practitioners to avoid exposing secrets accidentally is to add files explicitly in the VCS staging area. Developers should avoid using wildcard commands (`git add -A`, `git add .` and `git add *`) when adding files to have full control over what is committed. One practitioner also suggested: “*Committing early and committing often will not only help navigate file history and break up otherwise large tasks, in addition it will reduce the temptation to use wildcard commands.*” [109].

ASC-3: Use VCS Hooks to Check Files Prior to Committing (5)

To prevent secrets from pushing into VCS repositories, practitioners advise implementing VCS hooks [72] which allow scripts to be executed before or after a specific action in the VCS repository. The `pre-commit` and `post-commit` hooks can be used to filter and smudge secrets before commit or after pull, respectively [57]. Each contributor to the VCS repository needs to set up VCS hooks individually. According to practitioners, VCS hook scripts need extra effort to write properly since putting all of the secret behavior in the script is challenging.

4.2.5 Practices for Managing Secrets in Deployment (MSD)

Developers can expose secrets during deployment. Practitioners recommend the following four practices to manage secrets in deployment securely.

MSD-1: Use Secret Variables in CI/CD (6)

Practitioners recommend removing hard-coded secrets from CI/CD scripts, and use the secret variables of the build/deploy systems, such as Heroku [61] and Azure Pipeline [114]. VCS such as GitHub [115] and GitLab [116] have also secret variables which can be used in the CI/CD pipeline. The secret variables are set as environment variables in the deployment environment and hidden from any logs. Practitioners also suggest keeping secret variables settings disabled for pull requests to avoid inadvertently passing secrets during builds for pull requests.

MSD-2: Use Configuration Management Systems (4)

The configurations of different machines are coordinated by Configuration Management System (CMS) tools from a centralized location. Practitioners recommend using secret management systems supported by CMS tools, such as Chef-Vault [60] and Ansible-Vault [59]. Using shared secrets, these CMS tools keep secrets out of revision history and from other machines. Secrets can be distributed to specific machines using the same mechanism which ensures each machine receives the correct configuration.

MSD-3: Use Different Secrets for Each Environment (3)

Practitioners recommend to avoid using the same secrets for multiple environments so that exposure to the secrets of one environment does not compromise other environments. The secrets of production environments should be different from development or pre-production environments. Practitioners also recommend keeping production environment secrets limited to a small set of owners to avoid the risk of failure.

MSD-4: Keep Dot Files out of Root Directory (2)

During deployment, practitioners recommend keeping dot files, such as `.git`, `.gitignore` and `.env` files, out of the root directory. Proper access restrictions should be applied to dot files on production servers to avoid exposing secrets [117]. If the `.git` folder is not kept out of the root directory, then the whole history of committed changes will be exposed to the attacker. Previous research [5] has also found secrets in the `.gitignore` file despite the `.gitignore` file is designed to restrict unintended source files committing into VCS.

4.2.6 Organizational Practices to Enforce Policies for Secrets Protection (OEP)

Organizations can adopt general practices to enforce policies in VCS for developers. These general practices can minimize vulnerabilities which in turn helps in avoiding exposure of secrets. Practitioners recommend the below six practices for enforcing policies.

OEP-1: Tightly Manage Developer Permissions (6)

According to practitioners, organizations should follow the principle of least privilege. Organizations should not give developers more permissions than the required scope, such as

changing repository visibility and adding external contributors. If the repository contains secrets, the more developers who have permission to change the visibility of the repository, the higher the risk of failure. For example, GitHub has organization-level settings to restrict the ability to change the visibility of the repository to anyone with admin access or organization owners [118]. One practitioner stated: “*The higher the turnover of external contributors, the higher the security risks*” [119]. External contributors can be strictly managed to reduce the number of redundant developers and their access to the repositories. Limiting access and permission-granting privileges to organization owners is one way to handle external contributors. For example, GitHub has organization-level settings for allowing the permission to add external contributors to the organization owners only [120].

OEP-2: Enforce Two-Factor Authentication (6)

To prevent source code leakage via insecure developer accounts, practitioners recommend enforcing two-factor authentication (2FA). When logging into VCS, such as GitLab and GitHub, 2FA provides an added layer of security which can be enforced through organization-level settings [121, 122].

OEP-3: Require Commit Signing (3)

A malicious user can push exploitable code into VCS by pretending to be someone else and remain untraceable by altering the username and email address in the `git config`. For verification and traceability of code merge, commit signing [123], which is a cryptographic code-signing technique, can be used. Commit signing is done through GPG, and the signed commit gets a ‘verified’ badge. A malicious user’s commit can easily be traced as the commit will not have a ‘verified’ badge.

OEP-4: Add a security.md File (3)

Practitioners recommend adding a `security.md` file [124] in a VCS repository to officially document security-related processes and procedures, such as token accessibility, authentication requirements, and vulnerability reporting. The `security.md` file can serve as a helpful reference for developers as well as a centralized space for the organization’s security expectations.

OEP-5: Implement Single Sign-On (2)

SAML single sign-on (SSO) [125] is a VCS feature practitioners recommend. Access to VCS resources, such as specific repositories and pull requests, can be managed by explicitly providing permissions to resources by leveraging SAML SSO. SAML SSO also allows to set up of approved identity providers, which enables the organization to force the developers to sign in using the organization's accounts instead of privately-owned VCS accounts [125].

OEP-6: Disable Forking (1)

A Git feature called forking allows a developer to duplicate a repository and is useful for testing and sandboxing. However, practitioners recommend to disable forking. A fork can reveal secrets to the public though the repository is originally private. With each fork, the risk grows exponentially, resulting in a chain of security vulnerabilities. For example, GitHub has organization-level settings to disallow forking of repositories [126].

4.3 Discussion

In this section, we discuss the implications and limitations of our study.

Implication for Practitioners: Our derivation indicates how secrets should be kept out of source code or VCS to avoid being exposed. Insufficient application of practices related to secret management can result in unfavorable outcomes. The identified practices can help practitioners manage secrets in software artifacts and act as a comparison point for practitioners with their existing secret management practices.

Implications for Researchers and Tool Developers: Further research can be conducted in the field of secret management by leveraging our findings. For example, researchers can look into how many of the practices specified in Section 4.2 are followed in commercial and open-source software. We hypothesize that systematic follow of secret management practices is not prevalent based on the presence of secrets in software artifacts. If empirical studies support our hypothesis, researchers can look into the contributing factors to ignoring secret management practices. Our findings will also assist tool developers in determining whether to create new tools or improve existing ones to help practitioners manage secrets more effectively. The false-positive rate of VCS scan tools, for example, can be improved.

Limitations: Since the identified practices are bound to the Internet artifacts which we analyzed in our study, our findings are subject to external validity and may not generalize

to another collection of Internet artifacts. We account for the constraint by employing four search strings to collect and filter Internet artifacts in a systematic way. Manual analysis may induce bias while identifying practices. We account for this bias by adding a second rater who identified practices from the same set of Internet artifacts independently. Finally, the sets of practices of each rater are cross-checked to mitigate bias.

4.4 Conclusion

In addition to users, software also relies heavily on the use of secrets for authentication and authorization, and the exposure of secrets is increasing each day. The presence of secrets in software artifacts substantiates the practitioner's lack of knowledge on securely managing secrets. A set of secret management practices can help practitioners avoid exposing secrets in software artifacts. To identify practices for secret management in software artifacts, we conducted a grey literature review of 54 Internet artifacts. Our analysis identified 24 practices grouped into six categories and comprised of developer and organizational practices. According to our findings, the most recommended practices for moving secrets out of source code and securely storing secrets include using local environment variables (OSC-1) and external secret management services (SSC-1). We also observed that using VCS scan tools (ASC-1) and employing short-lived secrets (LSE-1) are the most recommended practices to avoid accidentally committing secrets and limit secrets exposure. Our findings can also be beneficial for researchers and tool developers who can investigate how the secret management process can be enhanced to facilitate secure development.

CHAPTER

5

STUDY III: SECRETBENCH: A DATASET OF SOFTWARE SECRETS

To avoid exposing secrets in VCS, several open-source and proprietary secret detection tools [78], such as TruffleHog [14] and Microsoft CredScan [127], are available. However, these tools have been shown to produce false positive warnings [8]. In previous studies [26, 19], researchers have worked on reducing false positives. However, their curated datasets are not large and varied and are unavailable for future research and evaluation purposes. In addition, developers face challenges in choosing one tool out of many, and no publicly-available dataset is available for comparing the effectiveness of the tools.

The goal of our study is to aid researchers and tool developers in evaluating and improving secret detection tools by curating a benchmark dataset of secrets through a systematic collection of secrets from open-source repositories.

We present SecretBench, a labeled dataset of source codes consisting of 97,479 secrets extracted from 818 public GitHub repositories using two secret detection tools. We manually inspected each secret and labeled 15,084 secrets as true secrets. The dataset encompasses 49 programming languages and 311 file types. The dataset is hosted in Google BigQuery [128] and Cloud Storage [129] and designed to be amenable to expansion by the community. Our

dataset will aid in expediting the research to evaluate and improve secret detection tools.

The rest of the chapter is structured as follows: The methodology used to curate the dataset in our study is described in Section 5.1. We provide the description of the dataset in Section 5.2. We discuss the originality and research opportunities of our dataset in Sections 5.3 and 5.4, respectively. The ethics and limitations of our study are discussed in Section 5.5 and 5.6, respectively. Finally, Section 5.7 draws the conclusion of our study.

5.1 Data Extraction

We provide our eight-step process for data collection of SecretBench as follows:

Step 1: Open Source Software Repository Platform Selection: We choose GitHub [11] to select candidate repositories containing secrets for our study. GitHub is the most popular platform for hosting open-source software development projects [13]. As of December 2022, GitHub has over 94 million developers and more than 330 million repositories [13], including at least 36 million public repositories [130].

Step 2: Build Regular Expression (Regex) Pattern Set: We build a regex pattern set for different types of secrets to identify the candidate repositories containing secrets for our study. For example, the regex pattern for a Slack token is “(xoxb|xoxp|xapp|xoxa|xoxr)-[0-9]{10,13}[a-zA-Z0-9]*”. TruffleHog [14], a popular open-source secret-scanning tool, has a package of secret detectors [131]. We extracted 751 regex patterns from the source code of the detector package and included those in our pattern set. In addition, we included 10 regex patterns from Meli et al. [5] to find the presence of secrets in GitHub repositories that are not present in the TruffleHog detector package. In total, we used 761 regex patterns in our pattern set, which is available online [132].

Step 3: Identify Candidate Software Repositories: To identify the candidate software repositories, we used the Google BigQuery Public Dataset of GitHub [128] (Dataset ID: *bigquery-public-data.github_repos*), which was released in 2016 by Google in collaboration with GitHub. The source code of over 2 billion files from more than 2.9 million open-source licensed repositories can be accessed with SQL queries [128]. We used the most recent snapshot available at the start of this project (September 20, 2022). We wrote an SQL script with all the 761 regex patterns to search for secrets in the source code files and executed the script in Google BigQuery. The SQL script took almost 22 hours to complete, as every file is checked with all the regex patterns. The returned result is a table of two columns: “repo_name” and “matches”. The “repo_name” column represents the repository name,

and the “matches” column represents the list of regex patterns matched with the specific repository. In total, we have found 2,234,618 repositories with at least one regex pattern match.

Step 4: Apply Selection Criteria on Candidate Repositories: As suggested by prior research [133], GitHub repositories need to be curated by removing inactive, beginner, and tech-demo projects. To curate the repositories collected in Step 3, we collected fork information, contributor counts, and commit counts using the GitHub Rest API [134]. We applied the following selection criteria to curate the collected repositories. The number in parenthesis with the criteria name indicates the number of filtered repositories after applying that specific criteria.

- **Availability (2,013,913):** The repository is available to download.
- **Uniqueness (1,735,864):** The repository is not a forked repository. This criteria is applied to avoid near duplicates of the same repository.
- **Collaboration (889,984):** The repository contributor count must be at or above the dataset median of 2. This criteria is applied to avoid personal or hobby projects.
- **Development History (622,719):** The repository commit count must be at or above the dataset median of 20 commits.
- **Recent Activity (93,958):** The repository must have at least one commit in the last one year. This criteria is applied to avoid inactive projects.

In addition, we observed some repositories with different “repo_name” fields point to the same repository. For example, repositories “Jasig/cas” and “apereo/cas” are the same repository though having different repository names in the dataset. This duplication happened because the repository owner changed the repository name at some point, but the Google BigQuery dataset kept both names. However, GitHub stores the actual repository name of the duplicate repository. We collected the actual repository name of each repository using the GitHub Rest API and filtered the duplicate repositories. After all selection criteria, we passed 89,070 unique repositories to Step 5.

Step 5: Find Multiset-Multicover Repositories of Regex Patterns: In this step, we further select repositories so that we get a sample of multiple secrets for each secret type while minimizing the overall repository count of the dataset. In later steps, we manually determine if identified secrets were actually secret or not. However, identifying and manually

labeling secrets from the 89,070 repositories remaining in Step 4 is impractical. Our goal of identifying the smallest selection of repositories that altogether include a specified count of each identified secret pattern is actually an instance of the *multiset-multicover* problem, so we applied the multiset-multicover algorithm described in Algorithm 1. This algorithm is an extension of the Minimum Set Cover algorithm [135] to select a minimal set of repositories covering all the regex patterns with a certain number of repositories for each pattern.

Before applying the multiset-multicover algorithm, we observe that 390 out of 761 regex patterns found no match in any repository. The median regex pattern matched 10 repositories, with 186 regex patterns matching 10 or more repositories. We term these patterns “upper tail” regex patterns. An additional 120 regex patterns matched between 1 and 9 repositories; we will refer to these as “lower tail” regex patterns. The median lower tail regex pattern matched 2 repositories.

For a comprehensive dataset, we seek a balance between examples of common and uncommon secret types, so we applied the multiset-multicover algorithm in two phases. In Phase 1, we ran the multiset-multicover algorithm for the 186 upper tail regex patterns to find a set of repositories where each regex pattern matches at least 10 repositories. We identified 649 repositories among the upper tail regex patterns. For Phase 2, we ran the multiset-multicover algorithm for the 120 lower tail regex patterns to find a set of repositories where each regex pattern should match at least 2 repositories and identified 190 repositories. Then, we merged the repositories of Phase 1 with Phase 2 and removed duplicate repositories. Altogether, we identified 818 repositories for SecretBench to collect candidate secrets.

Step 6: Find Candidate Secrets: We wrote a Python program to clone the repositories. We used GitPython [136] to download all the branches of a repository and saved the files into a Google Cloud VM Instance [137] (OS: Ubuntu 18.04 LTS, RAM: 16 GB, Persistent Disk: 500 GB). Next, we ran two secret detection tools, TruffleHog [14] and Gitleaks [138], to identify candidate secrets from the repositories. Both tools are widely used for secret detection and can identify secrets buried in the repository’s history and logs. We used these tools since manually inspecting each file of a repository to find secrets is infeasible and would be error-prone. The tools provide a JSON output for each repository. The JSON output contains the candidate secrets with additional metadata such as the commit id, commit date, committer email, file path, start line, end line, start column, and end column of the file where secrets are matched. Next, we wrote another Python program to read each report generated by the tools and extract the candidate secrets along with the metadata. Altogether, we identified 97,479 candidate secrets present in different commits of 818 repositories, of

Algorithm 1 Multiset-Multicover Algorithm

Require: *PatternsToCover, U*
Require: *InstanceSize, K*

- 1: $R_a \leftarrow \text{ReadAllRepos}()$
- 2: $\text{CoveredRepos}, C_r \leftarrow \emptyset$
- 3: $\text{CoveredPatterns}, C_p \leftarrow \emptyset$
- 4: **while** $C_p \neq U$ **do**
- 5: $M \leftarrow \text{FindRepoWithMostUncoveredPatterns}(R_a, C_p, U)$
- 6: $C_p \leftarrow C_p \cup \text{FindMatchedPatternsForRepo}(M, R_a)$
- 7: $C_r \leftarrow C_r \cup M$
- 8: **end while**
- 9: $R_{cc} \leftarrow \text{FindRepoCountPerPatternInInitialCover}(C_r, R_a, U)$
- 10: $U_p \leftarrow \text{FindPatternsLessThanKInstance}(R_{cc})$
- 11: **while** $\text{len}(U_p) \neq 0$ **do**
- 12: $M \leftarrow \text{FindRepoWithMostUncoveredPatterns}(R_a, C_p, U_p)$
- 13: $R_p \leftarrow \text{FindMatchedPatternsForRepo}(M, R_a)$
- 14: $C_p \leftarrow C_p \cup R_p$
- 15: **for** e **in** R_p **do**
- 16: $R_{cc}[e] \leftarrow R_{cc}[e] + 1$
- 17: **end for**
- 18: $U_p \leftarrow \text{FindPatternsLessThanKRepoInstance}(R_{cc})$
- 19: $R_a \leftarrow \text{RemoveSelectedRepoFromList}(M, R_a)$
- 20: $C_r \leftarrow C_r \cup M$
- 21: **end while**
- 22: $C_r \leftarrow \text{RemoveDuplicateRepos}(C_r)$
- 23: **return** C_r

which 27,336 secrets are unique.

Step 7: Label Candidate Secrets: The first and second authors manually inspected each candidate secret independently using the metadata collected in Step 6. A candidate secret is labeled as “True” if the secret is a true secret, otherwise labeled as “False”. We observed the agreement of the labeling of secrets with a Cohen’s Kappa [139] score of 0.86 between two raters, which indicates a “near perfect agreement” according to Landis and Koch’s interpretation [140]. The disagreements were resolved after a discussion between the two raters. In our dataset, we identified 15,084 true secrets, of which 4,014 secrets are unique.

Step 8: Developer Survey: We conducted a developer survey to evaluate whether the committer of the secrets agrees with our label. First, we selected unique secrets committed between 2021 and 2022 to avoid recall bias [141] from the developers and identified 7,617 secrets. Since GitHub allows the developers to use a no-reply email address

(`user-name@users.noreply.github.com`) as the commit email address [142], we filtered those secrets and identified 2,115 secrets. Then, we selected 200 secrets (randomly selected to avoid selection bias [143]) and emailed the developers to know if they agreed with our labeling of the secret and the reason they disagreed. In the email, we provided the repository name, commit id with the commit GitHub link, file path, start line, end line, and a screenshot of the code where the secret is found. We received 56 responses, a 28.0% response rate. Altogether, 44 (78.6%) respondents fully agreed with our label, while 6 (10.7%) respondents disagreed. The remaining 6 (10.7%) respondents were not sure.

5.2 Data Description

In this section, we provide brief details of our dataset.

5.2.1 Curated and Derived Fields

We collected the metadata related to the secret such as repository name, commit id, commit date, committer email, file path, start line and end line. To further enrich the dataset, we have augmented the mined data with additional features that are computed or derived from the source code files and secrets. Example of computed and derived fields are “file_type”, “is_template”, “in_url”, “entropy”, “character_set” and “has_words”. An overview of our SecretBench dataset is presented in Table 5.1.

5.2.2 Data Characteristics

Our SecretBench dataset is diverse in terms of different project characteristics. The dataset consists of 97,479 secrets in 818 repositories, and some repositories use multiple programming languages. For example, the repository “paradite/hn-ratio” [145] consists of two programming languages: JavaScript and Shell. Altogether, our dataset repositories used 49 programming languages. The top 5 programming languages based on the number of repositories are Shell (459), JavaScript (414), Python (312), Java (180), and Ruby (172). The number in parenthesis denotes the number of repositories containing the specific language. In addition, our dataset consists of secrets present in 311 file types. The top 5 file types based on the number of secrets in those files are js (10,412), nix (8,623), json (8,132), txt (7,737), and xml (6,429). Additionally, the top 5 file types based on the number of true

Table 5.1: Overview of the SecretBench Dataset

Field Name	Description	Data Type
id	Unique identifier of the secret.	Integer
secret	Candidate secret string.	String
repo_name	Name of the repository.	String
domain	Domain of the repository such as GitHub.	String
commit_id	Commit hash where the secret is added.	String
file_path	File path where the secret is included.	String
file_type	Type of the file such as .py and .config.	String
start_line	Start line no. where the secret is present.	Integer
end_line	End line no. where the secret is present.	Integer
start_column	Start index of the secret in the start line.	Integer
end_column	End index of the secret in the end line.	Integer
committer_email	Email address of the committer.	String
commit_date	The timestamp of the commit.	TimeStamp
label	The ground truth label of the secret.	Boolean
is_template	Flag to indicate if the secret is a placeholder such as “MY_PASSWORD”.	Boolean
in_url	Flag to indicate if the secret is part of URL such as “http://user:pwd@site.com”.	Boolean
entropy	Shannon entropy of the secret.	Float
character_set	Characters used in the secret (NumberOnly, CharOnly, Any).	String
has_words	Flag to indicate if any common English word [144] of at least length of 4 is present within the secret.	Boolean
length	Length of the secret.	Integer
is_multiline	Flag to indicate if the secret is present in multiple lines.	Boolean
category	The category of the secret.	String
file_identifier	Unique identifier of the file to check the secret from local system.	String
repo_identifier	Unique identifier of the repository to check the secret from local system.	String

secrets are txt (2,935), toml (1,985), js (1,583), html (1,337), and pem (813). The number in parenthesis denotes the number of secrets in the specific file type.

The secrets in our dataset are categorized into eight categories and presented in Table 5.2, sorted based on the number of true secrets. More details of our dataset is presented in our GitHub repository [146].

Table 5.2: The categories of secrets in SecretBench

Category	True Secrets	Total Secrets
Private Key	5,789	8,584
API Key and Secret	4,529	5,162
Authentication Key and Token	3,569	5,833
Other	524	66,690
Generic Secret	334	439
Database and Server URL	162	9,970
Password	150	705
Username	27	96

5.2.3 Data Storage

Our dataset is stored as relation structured data in Google BigQuery (Dataset ID: *dev-range-332204.secretbench.secrets*). Users can run SQL queries to access and expand the dataset. In addition, we stored the downloaded 818 repositories and the secret-containing individual source code files in Google Cloud Storage. When downloaded into the local system, the “repo_identifier” and “file_identifier” mentioned in Table 5.1 can be used to locate the repository and specific source code file related to the secret, respectively.

Since our dataset is sensitive, Google BigQuery and Cloud Storage enable us to give access to the dataset to only selected groups, such as fellow researchers and tool developers. To get access to our dataset, researchers and tool developers need to contact us through email.

5.3 Originality of SecretBench

Previous studies [19, 26, 5] have extracted secrets from the GitHub repositories, but none made their dataset public for future research purposes. Saha et al. [26] created a labeled dataset of 5000 secrets (700 true secrets) from 300 GitHub repositories using 32 regex patterns. With the dataset, they applied machine learning algorithms to distinguish true secrets. However, the repositories matched by regex patterns are not filtered for demo and inactive projects, and no information is provided on the files and languages covered. Sinha et al. [19] created a dataset of 84 GitHub repositories and identified pattern-based search and heuristics-driven filtering approaches to reduce the false positive detection of secrets. However, their dataset is small and contains only AWS credentials.

On the other hand, our dataset presented herein is large and diverse. We applied 761 regex patterns of different types of secrets and selected 818 GitHub repositories encompassing 49 programming languages. Our dataset consists of 97,479 labeled secrets, including 15,084 true secrets present in 311 different file types. We also provided different features related to the secret, such as whether the secret is a template or present in a URL. In addition, we made our dataset available for future researchers and tool developers.

5.4 Research Opportunities

To prevent exposing secrets in VCS, there are several open-source and proprietary secret detection tools [78]. However, these tools are known to generate false positive warnings [26, 8]. Researchers and tool developers can identify different rules and patterns from false positive secrets to reduce false positive warnings. However, mining data from open-source and building ground-truth datasets is challenging and time-consuming. In this case, our SecretBench dataset can be used to circumvent the challenge and speed up the research and tool evaluation on reducing false positives. In addition, since several secret detection tools exist, developers face difficulty choosing one tool out of many. Future research is needed to aid developers in making informed choices about using different secret detection tools through an analysis of the effectiveness of the tools. In this case, our SecretBench dataset can act as a benchmark for comparing the effectiveness of the secret detection tools.

Dataset Enhancement: Our dataset can be further improved by including repositories from other VCS services such as GitLab and Bitbucket. In addition, we can add more features regarding secrets to help in secret detection automatically using machine learning algorithms. Example features include whether the secrets have parentheses (possible function call), begin with a \$ sign (possible variable), and have context words such as “dummy” and “fake” in the surrounding code of the secret. We released these additional features online [147].

5.5 Ethics and Data Protection

Since our dataset contains sensitive information such as true secrets and the committer’s email addresses, we will distribute our dataset selectively. Researchers and tool developers who want to use our dataset will sign a data protection agreement with us to avoid any

unethical use. After that, we will give access to our dataset from Google BigQuery and Cloud Storage using their email addresses. In addition, at no point we did not attempt to use the secrets to verify the validity of the secrets. Instead, we labeled the secrets only by inspecting the secrets and the source code context of the secrets.

To validate our labeling, we only contacted the developers who committed the secrets. We did not reveal the identity of the developers to any managers or higher officials where they work. In addition, we are notifying every developer in our dataset to remove the secrets from their VCS.

5.6 Threats to Validity

In this section, we briefly discuss the limitations of our study. **VCS Selection:** We did not consider other VCS services such as GitLab [12] and Bitbucket [148]. In the future, we plan to expand our dataset by including repositories of other VCS services. **Manual Analysis Bias:** The labeling of the secrets in our dataset is susceptible to bias. To mitigate the bias, a second rater labeled the secrets independently, and we resolved the disagreements. **Recall Bias:** For the developer survey, though we have selected secrets that are committed in 2021 and 2022, the responses could have recall bias. We provided the developers with a screenshot of the secret-containing source code and additional metadata to mitigate the bias.

5.7 Conclusion

We provide the SecretBench dataset consisting of 97,479 labeled secrets extracted from 818 GitHub repositories encompassing 49 programming languages and 311 file types. Our dataset will aid in evaluating and improving secret detection tools, thus preventing secret leakage in VCS and application packages. By adding new projects and features, we aim to expand our dataset. We invite the research community to join our effort to expand and enrich the dataset to create novel software secret management research opportunities.

CHAPTER

6

STUDY IV: A COMPARATIVE STUDY OF SOFTWARE SECRETS REPORTING BY SECRET DETECTION TOOLS

To prevent secrets from leaking in VCS, several open-source and proprietary tools such as Gitleaks and SpectralOps are available. However, these tools generate many false positives. Chess and McGraw [149] state that a high percentage of false positives may lead to 100 percent false negatives because people stop using the tool. This phenomenon is called *alert fatigue* [150]. In addition, a tool will be unsound if it allows false negatives to escape to reduce false positives. As a result, developers face challenges in selecting secret detection tools. To our knowledge, no research has been conducted yet evaluating and comparing existing secret detection tools.

The goal of our study is to aid developers in choosing a secret detection tool to reduce the exposure of secrets through an empirical investigation of existing secret detection tools.

In this study, we analyzed existing open-source and proprietary secret detection tools and provided answers to the following research questions:

- **RQ1:** How do the secret detection tools perform in detecting secrets in terms of precision and recall?
- **RQ2:** What features are offered by the secret detection tools to aid in preventing secrets exposure?

We selected five open-source and four proprietary tools and compared the tools against a benchmark dataset of 818 repositories. We analyzed the tools report and evaluated how tools perform in detecting secrets. In addition, we analyzed the features offered by the tools in preventing the exposure of secrets and identified future research needs for secure software secret management. We have also made a dataset of the false positive secrets reported by the tools publicly-available for future researchers to aid in expediting research on the accuracy of the tools [151]. We summarize our contributions as follows:

- A first comparative study of the existing open-source and proprietary secret detection tools and a qualitative analysis of the reports generated by the tools;
- A categorization of the features provided by the secret detection tools to aid in preventing secrets exposure; and
- A dataset of false positive secrets reported by the tools.

The rest of the chapter is structured as follows: Section 6.1, 6.2 and 6.3 introduce the benchmark dataset, selection process of tools, and the methodology to compare and evaluate the tools result, respectively. We discuss the findings and implications of our work in Sections 6.4 and 6.5, respectively. Section 6.6 discusses the ethics, followed by the limitation of our study. We draw the conclusion of our study in Section 6.8.

6.1 Benchmark Dataset

To compare the secret detection tools, we selected *SecretBench* [152], a publicly-available benchmark dataset of software secrets. We accessed the dataset using Google Cloud Storage (Bucket Name: *secretbench*) [129] and Google BigQuery (Dataset ID: *dev-range-332204.secretbench.secrets*) [128]. A detailed description of the dataset is provided as follows:

Repositories: The dataset has been curated from the Google BigQuery Public Dataset of GitHub [128] using 761 regular expression patterns of different types of secrets. The dataset consists of 818 public GitHub repositories.

Secrets: The dataset consists of 97,479 labeled plain-text secrets (labeled as true and false) extracted from 818 repositories. The secrets were manually labeled by the two authors of SecretBench [152]. Among the 97,479 candidate secrets, 15,084 are true secrets. In addition, among the true secrets, 4,457 are unique since the same secret can have multiple instances in a repository (multiple commits and files).

Categories: The secrets of the dataset are categorized into eight categories. The number of total candidate secrets and true secrets of the eight categories are presented in Table 6.1. The top three categories based on the number of true secrets are: “Private Key”, “API Key and Secret” and “Authentication Key and Token”. The candidate secrets of the “Other” category are random strings and non-exploitable IDs such as GitHub commit IDs which are mostly false positives (99.29%).

Table 6.1: The eight categories of secrets in SecretBench.

Category	True Secrets	Total Secrets
Private Key	5,789	8,584
API Key and Secret	4,529	5,162
Authentication Key and Token	3,569	5,833
Other	524	66,690
Generic Secret	334	439
Database and Server URL	162	9,970
Password	150	705
Username	27	96

Programming Languages: The dataset repositories comprised source codes of 49 programming languages. The top five programming languages based on the number of repositories are Shell (459), JavaScript (414), Python (312), Java (180), and Ruby (172). The number in the parenthesis denotes the number of repositories that used the specific language.

File Types: The dataset consists of 311 file types in which secrets have been found. All the 311 file types and the number of true secrets present in these file types can be found in the GitHub repository of SecretBench [153]. The top five file types based on the number of true secrets are presented in Table 6.2.

Secrets Metadata: The dataset provides secrets metadata, such as repository name, file path, commit ID and start line of where the secrets are matched. We used the metadata to compare the tool-reported secrets, as discussed in Section 6.3.

Table 6.2: SecretBench’s top five file types on true secrets.

File Type	Description	True Secrets
txt	Text File	2,935
toml	Configuration File	1,985
js	Javascript file	1,583
html	Hypertext Markup Language File	1,337
pem	Privacy Enhanced Mail Format File	813

6.2 Secret Detection Tools

In this section, we explain the selection process of secret detection tools; provide a brief description of each tool; how we installed each tool; and how we scanned the benchmark repositories using each tool.

6.2.1 Selection of Secret Detection Tools

To find the existing open-source and proprietary secret detection tools, we searched both the web and academic literature. We constructed a set of the following search strings: *(secret OR credential OR password) AND (detection OR scanning OR digger) AND (tool OR utility)*. For web search, we used the Google Search Engine and selected the top 100 results for each search string according to the Google Search Engine’s Page Rank algorithm. The stopping criteria of 100 for each search string has been set based on the guideline of grey literature search in prior works [18]. Similarly, for academic literature search, we searched the top five scholar databases recommended in the computing science domain [154, 155, 156, 157, 158]. We identified 20 tools from the search result and applied the following selection criteria to choose the secret detection tools for our study.

1. **Accessible:** The tool can be installed into a local system or accessed via subscription from the tool vendors.
2. **Scans Git Repositories:** The tool can scan Git repositories since our dataset contains Git repositories.
3. **Active:** The tool’s repository has shown activity for the last two years. We checked the last commit date in the repository of the open-source tools.

4. **Flags Secrets:** The tool flags individual secrets instead of flagging only secret-containing suspicious file names.
5. **Reports Plain Text Secret:** The tool reports secrets in plain text as we must compare the secrets with our benchmark dataset.

Based on the above selection criteria, we excluded 11 tools. After each tool, we provide in parenthesis the criteria we used to exclude a tool using the enumerated criteria listed above: Credential-Digger [159] (1), Credscan [160] (1), Cocode [161] (1), detect-secrets [162] (5), git-all-secrets [113] (3), git-hound [163] (5), gitrob [76] (3), Gittyleaks [164] (3), repo-security-scanner [165] (4), SecretHunter [166] (1) and Saha et al. Tool [26] (1). Ultimately, we selected 9 secret detection tools, of which 5 tools are open-source and 4 tools are proprietary.

6.2.2 Tools Description

For the selected secret detection tools, we provide a) a brief description of the tool, b) how we installed the tool, and c) the scanning technique employed for finding secrets in benchmark repositories. Since each tool provides configuration options for detecting secrets, we installed and ran the tools with recommended configurations by contacting the tool vendors or by obtaining suggested configurations in the product documentation to get higher accuracy.

git-secrets: git-secrets [167] developed by AWS-Labs [168] is an open-source tool. We installed Version 1.3.0 of the tool using HomeBrew. In addition, as a pre-requisite to scan for secrets in the repositories, we installed two git hooks (`git secrets -install` and `git secrets -register-aws`) separately for each repository. We used the `-scan-history` flag (`git secrets -scan-history &> report.txt`) to scan the entire Git history and outputted the secrets in a text file.

Gitleaks: Gitleaks [138] is an open-source tool written in Go. We installed Version 8.2.7 of the tool using HomeBrew and scanned the repositories using the `detect` command (`gitleaks detect -v -source=repo_dir -report-path=report.json`). The verbose flag (`-v`) has been used to retrieve metadata information of the matched secret, and we extracted the secrets in JSON files.

Repo-supervisor: Repo-supervisor [169] is an open-source tool written in JavaScript. We downloaded the binary release (Version 3.2.0) and installed Node Package Manager (NPM) dependencies (`npm ci && npm run build`). The tool operates in two separate modes. The first mode allows to scan GitHub pull requests through webhooks, and the second mode

works from the command line, where it scans local repository directories. We performed the latter by executing the `cli.js` file (`JSON_OUTPUT=1 node ./dist/cli.js repo_dir`) and extracted the output in JSON file.

TruffleHog: TruffleHog [14] is an open-source tool developed by Truffle Security [170] and written in Go. We installed Version 3.18.0 of the tool using HomeBrew. We scanned the repositories with `-regex` and `-entropy` flags enabled (`trufflehog git -regex -entropy file://repo_dir`) and downloaded the JSON report.

Whispers: Whispers [171] is an open-source tool written in Python. The tool supports different formats for structured text parsing, such as YAML and XML. The tool parses the source code in key-value pairs, where the key is the field name and the value is the potentially hard-coded secret assigned to the given key. We installed Version 2.1.5 of the tool using pip3. To scan the repositories, we executed the `whispers repo_dir > report.json` command and extracted the output in JSON files.

Commercial X: Since the proprietary tool vendor would not allow their identity to be disclosed in the study, we refer to them as “Commercial X”. In addition to scanning GitHub repositories, the tool can find secrets in images and non-searchable PDFs. The tool can be integrated with Slack, JIRA, and Google Drive to find any secrets exposure. We contacted their team and provided the snapshot of 818 repositories of our benchmark. They ran their tool on those repositories and provided us with the scan report. We parsed the scan report and outputted the secrets with the metadata in a CSV file.

ggshield: ggshield [172] has been developed by GitGuardian [173]. We installed the tool (Version 1.14.3) using HomeBrew. Though the tool is open-sourced in GitHub, the tool requires an API key for scanning a repository since ggshield internally uses GitGuardian’s public API [174] through `py-gitguardian` [175] client to scan and detect secrets. We contacted GitGuardian to get an API key (API Quota Limit: 8 Million) and set the key in the local environment variable to scan all the benchmark repositories. We executed the `scan repo` command “`ggshield secret scan repo repo_dir --show-secrets --json -v -o report.json`” for searching secrets in each repository. The `--show-secrets` flag has been used to extract the secrets in non-redacted form, and the found secrets are outputted in a JSON file.

GitHub Secret Scanner: GitHub has an integrated secret scanner [176] to scan for secrets in the repositories. The “Secret Scanner” settings can be enabled from the “Code security and analysis” option in GitHub. To scan the repositories of the benchmark dataset, we forked each repository into the first author’s GitHub account. We enabled the “Secret Scanner” settings for each repository. As soon as we enabled the settings, the scanner was triggered and displayed the detected secrets under the “Security/Secret scanning alerts” tab of the

specific repository. We wrote a Python script to extract each repository’s secrets in a CSV file using GitHub Rest API [134].

SpectralOps: SpectralOps [15] is a proprietary tool. To scan repositories in a local environment, we created a Spectral account and contacted the Spectral support team to gain access for seven days. We received a Spectral Data Source Name (DSN) key and saved it in the local environment. The tool provides three scanning modes: “Developer”, “Security” and “Audit” based on different precision and recall rates. The Spectral team recommended using the “Security” mode for better precision and recall. We ran the scan command (“spectral scan –all –forensic –ok –show-match –include-tags base,audit –with-branches –json report.json”) and outputted secrets in JSON files. The base and audit tags are used for “Security” scan mode, and –forensic flag retrieves the secret’s metadata.

6.2.3 Machine Configuration

We installed eight tools in two Mac instances except for the GitHub Secret Scanner and Commercial X. The configuration of the instances are as follows: Instance 1 (OS: Monterey version 12.3.1, RAM: 64 GB, Persistent Disk: 1 TB) and Instance 2 (OS: Monterey version 12.6.2, RAM: 32 GB, Persistent Disk: 1 TB). We used two Mac instances to speed up the scanning process since the benchmark dataset contains large repositories with a large commit count. After scanning with each tool, we wrote Python scripts to extract the secret with additional metadata from the JSON and text files and outputted in CSV files. The extracted results are used for analysis and comparison, as discussed in Section 6.3.

6.3 Analyzing Tool Results

In this section, we explain the secret and tool metadata we analyzed and how we filtered and compared the tool results to answer our research questions.

6.3.1 Secret Metadata

In this section, we discuss the metadata information related to secrets we processed to answer our research questions.

Commit ID: A commit ID in Git is a unique SHA-1 hash created whenever a new commit is recorded. The commit ID helps to identify the exact commit reference where the secrets have been found during comparison.

File Path: The file path is the file's location in the repository where the secret has been found. We normalized the file path as it contained either the computer root folder location where the tool has been installed or the repository directory. For example: Repopervisor outputs the file path as “<Repo_dir>/conf/file.py” while Spectralops outputs as “/Users/<User_name>/<Repo_dir>/conf/file.py”. We extracted the file path as “conf/file.py” for comparison.

Line Number: The line number denotes the line in the file where the secret has been matched, which helps to identify if the same secret is present in multiple places of the same file.

Plain Text Secret: The plain text secret is the tool-reported hard-coded secret in the source code. However, some tools report secrets along with the source code context. For example, git-secrets outputs the function or variable declaration where the secret is used (`bitly_token <- bitly_auth(key = "xxxxxx")`). The “xxxxxx” is the secret where `bitly_auth` and `bitly_token` are the function and variable name, respectively. As a result, matching reported secrets with the benchmark through automation is difficult. In addition, manual inspection is impractical due to the large number of reported secrets by the tools. However, we observed patterns such as “key=”, “token=” and “id:” in the reported secret text. We removed non-alphanumeric characters, such as brackets and space, from the string and extracted the secret by only taking the string part after the pattern. We used these normalized secrets for comparison.

Alert Count: The alert count is the total number of alerts reported by each tool which indicates the amount of audit effort required by the practitioners. Tools such as SpectralOps and ggshield provide the number of alerts in the respective reports. For tools that do not provide the number of alerts in the report, we calculated the total number of alerts using a Python script by iterating through each report.

6.3.2 Filter and Compare Tool Alerts

We observe that tools provide non-secret alerts, such as alerts for suspicious files and dangerous functions. For example, Whispers flags suspicious files, such as `database.sql` file, and dangerous functions, such as `exec` and `eval`. In the output, the tool provides a rule identifier for different types of alerts, such as `secret` and `api-key` for secrets; `file-known` for suspicious files; and `system` for dangerous functions. We filtered the non-secret alerts using the rule identifiers. We also filtered secrets committed after November 25, 2022, since the benchmark dataset contains secrets introduced before that date. For example, the

GitHub secret scanner scans the repository’s latest snapshot (February 25, 2023) since the tool can not scan a local repository. We retrieved the commit date of each commit using GitHub Rest API [134]. We filtered any secrets introduced after November 25, 2022, for a fair comparison of the tools with the benchmark.

Next, we compare the secret of each repository reported by the tool with the secrets of the same repository in the benchmark. We mark the secret reported by the tool as true positive (TP) if the secret is matched. Otherwise, we mark the secret as a false positive (FP). However, we are unable to match different types of secrets with exact string comparison for all the tools though we normalized the secrets. We now discuss the different scenarios of the secret match and how we calculated the match for each.

Jaro-Winkler Similarity: After normalizing the secrets for source code context, we observe that additional source code as a suffix can be present. For example, git-secrets outputs secrets with additional source code context (`"analytics_configuration": {key: "xxxxxxxxxxxxx", type: "Traffic"}`). The secret is “xxxxxxxxxxxxx” and after normalizing, we got “xxxxxxxxxxxxxtypeTraffic” where the string part “typeTraffic” is not part of the secret. As a result, we cannot perform an exact match of the secret with the benchmark. To address this scenario, we used Jaro-Winkler Similarity [177] for string comparison, a variant of the Jaro Distance metric [178]. The Jaro–Winkler similarity employs a prefix scale that rewards strings that match from the beginning with high scores [177]. The Jaro–Winkler algorithm provides a similarity score between [0,1] where 0 represents two entirely dissimilar strings and 1 represents identical strings. We used the `jaro_winkler_similarity` function of `jellyfish` [179] package in Python to calculate the similarity. We found the similarity score of “xxxxxxxxxxxxx” and “xxxxxxxxxxxxxtypeTraffic” is 0.82. We termed two secrets a match if the similarity score equals or exceeds 0.7. We set the cut-off similarity score of 0.7 by randomly sampling secrets and observing the score with the benchmark.

Gestalt Pattern Match: We observe that a secret can contain additional context in the middle, especially for multi-line secrets. For example, private keys are generally present as multi-line in the source code. Tools output these private keys differently, making it difficult to perform an exact match with the benchmark. Figure 6.1 shows three different outputs of the same secret. Tool A outputs the “Proc-Type” and “DEK-Info” properties along with carriage return (“\r”) and line feed (“\n”), which is the same as the benchmark. However, Tool B excludes the “Proc-Type” and “DEK-Info” properties in the output, and Tool C includes the properties but outputs the secrets in a single-line instead of a multi-line without “\r\n”. To address this scenario, we used the Gestalt pattern matching algorithm [180] after

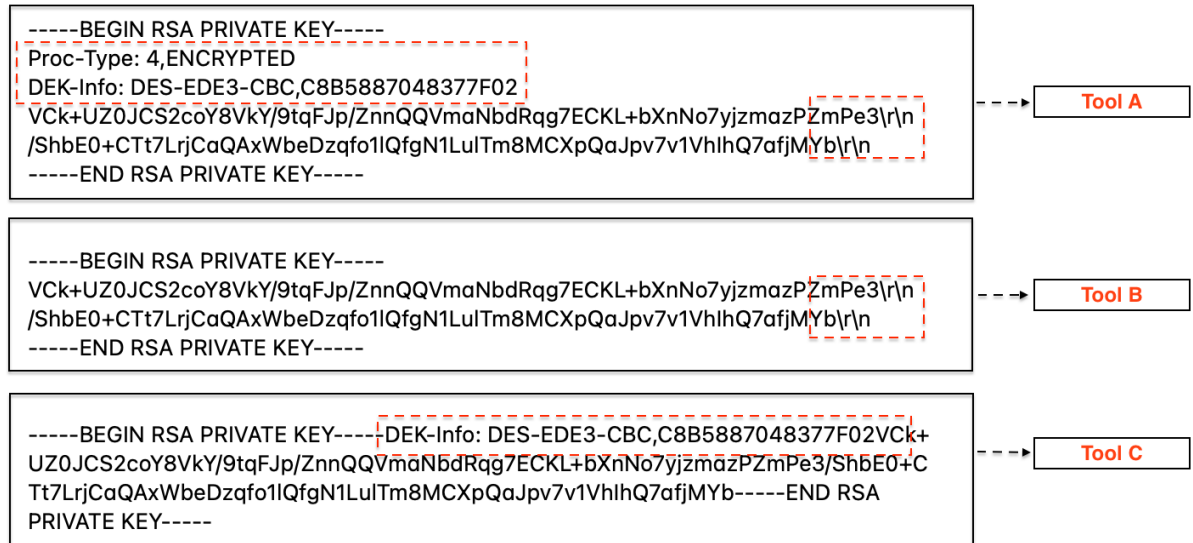


Figure 6.1: Different outputs of the same secret by three tools.

removing non-alphanumeric characters from the secret and making the secret single-line. The algorithm calculates the similarity score by finding the longest common substring and then recursively finding the number of matching characters in the non-matching regions on both sides of the longest common substring [180]. As a result, we could match a secret even if the secret does not contain the middle context (the properties of the private key). We used the `SequenceMatcher` function of `difflib` [181] package in Python to calculate the Gestalt similarity score. We termed two secrets a match if the similarity score equals or exceeds 0.6. Similar to the Jaro-Winkler similarity, we set the cut-off similarity score of 0.6 by randomly sampling secrets and observing the score with benchmark secrets.

We marked a secret reported by a tool as TP if the secret equals or exceeds the cut-off similarity score of either the Jaro-Winkler or Gestalt algorithm. To check whether the combination of algorithms correctly matches tool-reported secrets with benchmark and label automatically, we randomly selected 100 unique reported secrets from each tool and manually inspected the label calculated by the algorithms. The combination of both algorithms correctly labeled 97% of the secrets.

Recall Cases: We observe that the same secret can be present in multiple commits, multiple files, and different lines of the same file of a repository. As a result, finding and removing all instances of a secret from the source code is necessary. However, every tool does not provide all the metadata related to secrets, such as the commit id, file path, and line number, as shown in Table 6.5. As a result, we calculated the recall of each tool in

two cases to have a fair comparison. Case 1 of recall denotes when the secrets of the benchmark are found exactly in the same commit, file, and line number of the tools report, and Case 2 denotes that the secrets of the benchmark are found at least in the repository, irrespective of the metadata. For Case 1, we matched each tool’s reported secrets with all the benchmark secrets for a repository. If a secret of the benchmark matches the tool-reported secret but does not match the metadata, then we mark the secret as a false negative (“FN”). However, for Case 2, we matched the unique secrets of the benchmark for a repository with the reported secrets of the tools. If a secret of the benchmark matches the tool-reported secret but does not match the metadata, we still mark the secret as true positive (“TP”) since the secret is at least found in the repository. We could not calculate Case 1 for Repopervisor and SpectralOps as these tools do not provide either commit id or line number, thus calculating F1-score using precision and Case 2 of recall.

6.3.3 Tool Metric

In this section, we discuss the tool metric we calculated to answer our research questions.

Scan Time: Scan time helps to understand how quickly secrets will be identified to remediate any secrets exposure. Running each tool multiple times on all 818 benchmark repositories is impractical since scanning takes a long time. Hence, we calculated the scan time on a sample set of repositories of our benchmark to calculate the efficiency of the tools. First, we curated the sample set of 15 repositories as follows:

- **Repository Size:** The largest, smallest and median size of a repository in the benchmark is 5,658.22 MB, 0.04 MB, and 37.42 MB, respectively. We selected a random sample of 6 repositories based on the repository size: 4 repositories with repository sizes greater than the median and 2 repositories less than the median.
- **Commit Count:** Since a repository of a larger size can have a low number of commit counts, and vice-versa, we also included repositories in the sample set based on the commit count. The benchmark repository’s highest, lowest and median commit count is 425,699, 22, and 1,200, respectively. We selected a random sample of 6 repositories based on the commit count: 4 repositories with a commit count greater than the median and 2 repositories less than the median.
- **Programming Language:** The sample set should have at least 1 repository for each of the top 5 programming languages of the benchmark (see, Section 6.1). We randomly

selected 3 additional repositories since 2 languages were already present in the above-selected 12 repositories.

Next, we ran each tool 5 times on each of the 15 repositories, calculated the total scan time using the `time` [182] package of Python, and calculated the average scan time.

Popularity: Since the open-source tools publish their source code in a public repository, we can measure the tool’s popularity among the developers. Developers can *fork* the open-source tools repository in GitHub. The fork count of a repository indicates a higher chance of attracting potential contributors to the project. Developers can also *star* a repository when they want to appreciate the project and *watch* when they want to be notified of all the activities (bug fixes, new features) of the project. We used each open-source tools repository’s fork, star, and watch count as a proxy to calculate the tool’s popularity instead of considering a single metric. Previous studies [183, 184] have also used these metrics to calculate the popularity of a repository. To verify the rank correlation among fork, star, and watch count, we calculated the Spearman’s rho (ρ) [48] using Kaggle’s GitHub repository dataset [185]. We observed a significant correlation between star and fork ($\rho = 0.71$), watch and fork ($\rho = 0.60$), and watch and star ($\rho = 0.55$) counts. To calculate the popularity score for each tool, we normalized the fork, star, and watch counts using min-max normalization [49] and calculated the average of the counts.

6.4 Results

In this section, we discuss our findings and answer our research questions.

6.4.1 RQ1: How do the secret detection tools perform in detecting secrets, in terms of precision and recall?

We now discuss a) the precision, recall, and F1-score of each tool; b) the overlap of secrets reporting by the tools; c) a comparison of the scan time and popularity of the tools; and d) an analysis of the false positives and false negatives reported by the tools.

Precision, Recall and F1-score: Table 6.3 presents the precision, recall and F1-score of each tool. The column “Precision (Total Alerts, TP)” denotes the precision of each tool in detecting secrets. The numbers in parenthesis denote the total number of alerts reported by the tool and the count of true positives detected by the tool, respectively. The columns “Recall - Case X (TP, FN)” present the recall of each tool, where X denotes the two cases as

discussed in Section 6.3.2. The numbers in parentheses denote the number of true positives and false negatives found by the specific tool, respectively. Low precision indicates more false positives causing the tool to be unusable and low recall indicates more false negatives causing a missed opportunity to be alerted of a secret. The column “F1 Score” denotes the F1-score of each tool, the harmonic mean of precision and recall (Case 2) as discussed in Section 6.3.2. We now discuss our observations related to precision, recall, and F1-score.

Table 6.3: Precision, Recall, F1-Score, Scan Time (ST), and Popularity Score (PS) of each tool.

Tool	Precision	Recall - Case 1	Recall - Case 2	F1 Score	ST	PS
	(Total Alerts, TP)	(TP, FN)			(min.)	
git-secrets	0.05 (94491,4907)	0.04 (671,14413)	0.21 (956,3501)	0.08	6.71	0.92
Gitleaks	0.46 (45932,21047)	0.86 (12954,2130)	0.88 (3901,556)	0.60	46.29	0.85
Repo-supervisor	0.02 (181310,3652)	X	0.17 (751,3706)	0.04	0.32	0.04
TruffleHog	0.06 (90982,5426)	0.31 (4736,10348)	0.52 (2323,2134)	0.11	8.52	0.87
Whispers	0.01 (416516,2448)	0.01 (122,14962)	0.38 (1707,2750)	0.02	0.91	0.00
Commercial X	0.25 (86607,21674)	0.22 (3255,11829)	0.48 (2151,2306)	0.32	X	X
ggshield	0.19 (167046,32277)	0.23 (3536,11548)	0.46 (2068,2389)	0.26	228.94	0.06
GitHub-scanner	0.75 (1721,1292)	0.03 (408,14676)	0.36 (1606,2851)	0.48	54.48	X
Spectralops	0.01 (1547994,4777)	X	0.67 (2979,1478)	0.02	50.03	X
	Highest	Second Highest	Third Highest			

- We observe that based on the precision, the top three tools are GitHub Secret Scanner (75%), Gitleaks (46%), and Commercial X (25%), respectively. Among the nine tools, five tools have a precision score of less than 7%.
- Based on recall, we observe that Gitleaks is the top tool in both cases (Case 1: 86% and Case 2: 88%) and the second-best based on precision. In addition, TruffleHog has the second-best recall in Case 1 (31%) and third-best in Case 2 (52%) though the precision is only 6%.
- We observe that based on F1-score, the top three tools are Gitleaks (60%), GitHub Secret Scanner (48%), and Commercial X (32%).
- Though GitHub Secret Scanner is the top tool based on precision, the recall score is

low (6%), indicating the tool misses many secrets. In contrast, SpectralOps is the third-best based on recall (68%), with a precision score of only 1%. Thus, our findings indicate that no current tool has the coveted high precision and high recall scores.

- Recent research [27, 26] utilizes machine learning (ML) to reduce false positives. However, Commercial X and SpectralOps, which employ ML to detect secrets, have lower precision scores 25% and 1%, respectively.

Table 6.4: Recall of each tool for eight secrets categories.

Category	Case	git-secrets	Gitleaks	Repo-supervisor	TruffleHog	Whispers	Commercial X	ggshield	GitHub-scanner	Spectralops
Private Key	Case 1	0.00 (4,5785)	0.96 (5585,204)	X	0.39 (2284,3505)	0.00 (28,5761)	0.37 (2133,3656)	0.38 (2227,3562)	0.00 (22,5767)	X
	Case 2	0.28 (782,2018)	0.99 (2759,41)	0.16 (441,2359)	0.59 (1648,1152)	0.55 (1546,1254)	0.57 (1606,1194)	0.53 (1470,1330)	0.33 (914,1886)	0.77 (2166,634)
API Key and Secret	Case 1	0.05 (233,4296)	0.86 (3917,612)	X	0.18 (802,3727)	0.01 (26,4503)	0.12 (547,3982)	0.14 (624,3905)	0.05 (205,4324)	X
	Case 2	0.09 (55,586)	0.75 (478,163)	0.17 (111,530)	0.33 (211,430)	0.11 (68,573)	0.34 (220,421)	0.44 (284,357)	0.38 (241,400)	0.55 (352,289)
Auth Key and Token	Case 1	0.09 (338,3231)	0.71 (2539,1030)	X	0.36 (1274,2295)	0.01 (46,3523)	0.08 (286,3283)	0.10 (378,3191)	0.04 (125,3444)	X
	Case 2	0.14 (74,463)	0.56 (299,238)	0.24 (127,410)	0.58 (308,229)	0.09 (49,488)	0.27 (143,394)	0.33 (176,361)	0.48 (258,279)	0.43 (233,314)
Generic Secret	Case 1	0.17 (67,277)	0.96 (321,13)	X	0.09 (29,305)	0.04 (12,322)	0.31 (105,229)	0.44 (148,186)	0.01 (4,330)	X
	Case 2	0.14 (18,114)	0.94 (124,8)	0.11 (15,117)	0.14 (18,114)	0.08 (10,122)	0.42 (56,76)	0.58 (76,56)	0.42 (56,76)	0.61 (81,51)
DB and Server URL	Case 1	0.00 (0,162)	0.34 (55,107)	X	0.93 (150,12)	0.01 (2,160)	0.43 (69,93)	0.51 (83,79)	0.26 (42,120)	X
	Case 2	0.08 (5,61)	0.41 (27,39)	0.29 (19,47)	0.98 (65,1)	0.11 (7,59)	0.60 (40,26)	0.59 (39,27)	0.67 (44,22)	0.53 (35,31)
Password	Case 1	0.07 (11,139)	0.70 (105,45)	X	0.32 (48,102)	0.04 (6,144)	0.38 (57,93)	0.26 (39,111)	0.00 (0,150)	X
	Case 2	0.08 (5,55)	0.85 (51,9)	0.15 (9,51)	0.17 (10,50)	0.18 (11,49)	0.57 (34,26)	0.15 (9,51)	0.23 (14,46)	0.55 (33,27)
Username	Case 1	0.85 (23,4)	0.85 (23,4)	X	0.85 (23,4)	0.00 (0,27)	0.00 (0,27)	0.26 (7,20)	0.00 (0,27)	X
	Case 2	1.00 (2,0)	1.00 (2,0)	0.00 (0,2)	1.00 (2,0)	0.00 (0,2)	0.00 (0,2)	0.5 (1,1)	0.00 (0,2)	0.00 (0,2)
Other	Case 1	0.01 (5,519)	0.78 (409,115)	X	0.24 (126,398)	0.00 (2,522)	0.11 (58,466)	0.06 (30,494)	0.02 (10,514)	X
	Case 2	0.07 (15,204)	0.74 (161,58)	0.13 (29,190)	0.28 (61,158)	0.07 (16,203)	0.24 (52,167)	0.06 (13,206)	0.36 (79,140)	0.41 (89,130)
								Highest	Second Highest	Third Highest

Since the secrets of our benchmark dataset are categorized into eight categories, such as “Private Key” and “API Key and Secret”, we calculated the recall score per category for each tool. As a result, we identified which tool performs best in which category of secrets to aid developers in choosing tools based on the category of secrets present in their code. Table 6.4 presents the recall score of each tool for the eight categories in two cases (Case 1 and Case 2). The numbers in parentheses denote the number of true positives and false negatives found by the tool for a specific category. We observed that Gitleaks and TruffleHog are the top two tools in most categories. However, SpectralOps has the second-best recall score for categories such as “Private Key” (Case 2) and “Generic Secret” (Case 2), whereas ggshield has the second-best recall for “Username” (Both cases). In addition, SpectralOps has the second-best recall score for categories such as “API Key and Secret” (Case 2), whereas GitHub Secret Scanner has the second-best recall for “Database and Server URL” (Case 2).

Tool Overlap: We measured how much unique true positive (TP) secrets one tool reported overlap with another to identify which tools output similar secrets. The heatmap of Figure 6.2 depicts the overlap ratio between each pair of tools. For a pair of tools (A, B),

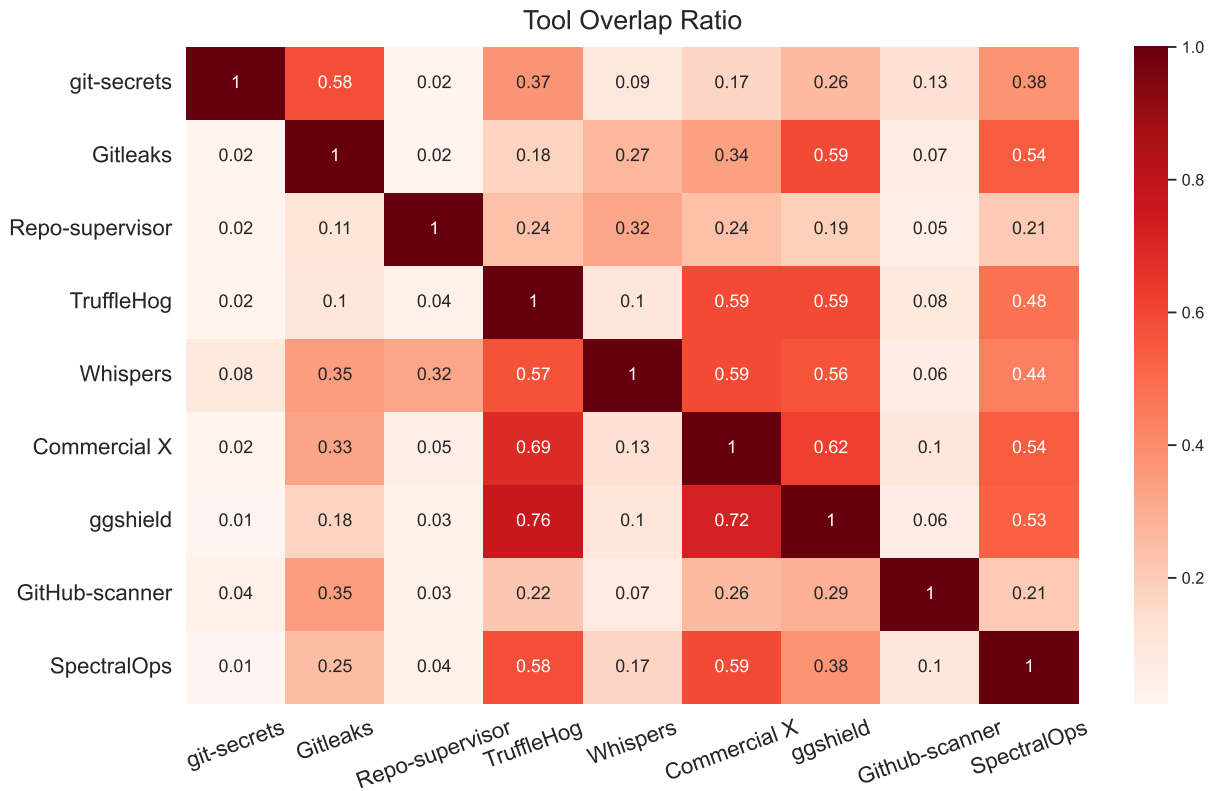


Figure 6.2: Overlap ratio of secrets reported by each tool.

the heatmap shows how many unique TP secrets reported by tool A are also reported by tool B. For example, 76% of the unique TP secrets reported by ggshield are also reported by TruffleHog. However, only 18% of the unique TP secrets reported by ggshield are reported by Gitleaks. The Venn diagrams in Figure 6.3 show the non-overlap unique TP secrets among Gitleaks, TruffleHog, and ggshield (*Top three tools based on recall (Case 1)*) and among Gitleaks, SpectralOps, and TruffleHog (*Top three tools based on recall (Case 2)*). Figure 6.3a shows that Gitleaks and TruffleHog outputs 1533 and 438 non-overlap unique TP secrets, respectively. Similarly, as shown in Figure 6.3b, we observed that Gitleaks and TruffleHog outputs 632 and 334 non-overlap unique TP secrets, respectively. As a result, our findings substantiate the necessity of not relying on a single tool to identify all the secrets present in a repository.

Scan Time: The column “ST” of Table 6.3 shows the time taken by each tool in minutes to scan the sample set of repositories. We could not calculate the scan time of Commercial X as the tool vendor has conducted the scanning, and the report does not contain any scan time. The top three tools based on scan time are Repo-supervisor, Whispers, and git-secrets,

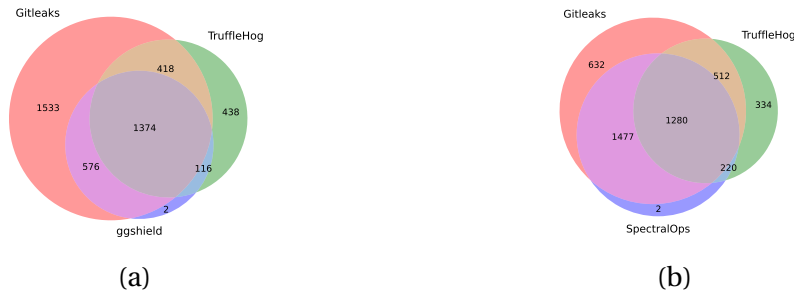


Figure 6.3: Venn diagram for overlap of unique true positive secrets among top three tools based on recall. Subfigure (a) depicts the overlap of Gitleaks, TruffleHog, and ggshield. Subfigure (b) depicts the overlap of Gitleaks, SpectralOps, and TruffleHog.

which took 0.32, 0.91, and 6.71 minutes, respectively. However, these tools have relatively low precision and recall scores indicating that tools did not scan the source code thoroughly. In contrast, the top two tools based on precision - GitHub Secret Scanner and Gitleaks took 54.48 and 46.29 minutes, respectively. However, we observe that tools having higher scan times do not always yield high precision and recall scores. For example, ggshield took the highest amount of time (4.8 hours) among all the tools, but the precision and recall were relatively low. We identified that Gitleaks, GitHub Secret Scanner, and SpectralOps showed a balance between scanning time and either high precision or recall.

Tool Popularity: The column “PS” of Table 6.3 presents the popularity score of each tool. We could only calculate the popularity score of the five open-source tools and one proprietary tool, ggshield. The source code of ggshield is open-sourced in GitHub, except for their proprietary scanning API implementations. Based on the PS score, the top three tools are git-secrets (0.92), Gitleaks (0.87), and TruffleHog (0.85), respectively. Though git-secrets is the most popular among the developers, the precision and recall are relatively low. In contrast, Gitleaks and TruffleHog are popular among developers having relatively high precision or recall scores.

Analysis of False Positives: Since we observed a high false positive rate by the tools, we inspected a random sample of 50 false positives from each tool to identify the types of false positive secrets. We now discuss our observations related to the false positive secrets and the detection rules triggering the false positives.

1. Generic Regular Expressions (regex): Tools use generic regex to detect secrets that trigger false positives. We now discuss the generic regex for different types of secrets.

1.1 API Keys and Tokens: Tools, such as Whispers, employ generic regex (`. * [A-Za-z0-9_]+(key|token)$`) for finding API keys and tokens. The regex treats any string

having a “key” or “token” at the end as an API key or token. As a result, placeholder API keys or tokens such as “testkey” and “sampletoken” are output as secrets. However, tools such as Gitleaks and GitHub Secret Scanner identify API keys and tokens by applying regex for specific API keys and tokens. For example, the regex employed for the Stripe API key is `(?i)(sk|pk)_(test|live)_[0-9a-z]{10,32}`. However, the regex matches “sk_live_111111111111”, a dummy API key, and outputs as a Stripe API key.

1.2 Password: To detect passwords, generic regex such as `(passwords?|passwd|pass|pwd)_?[0-9]*$` is used. As a result, strings such as “testpassword” or a UNIX command (“pwd”) are detected as passwords.

1.3 Cryptographic Key: According to a study by Meli et al. [5], cryptographic keys are the most exposed secrets in the source code. However, tools employ generic regex `(.*[-]{3,}BEGIN(RSA|DSA|EC|OPENSSSH)??(PRIVATE)?KEY[-]{3,}.*)` to identify cryptographic keys, thus reporting false positives. For example, a template string such as “--BEGIN RSA KEY--” with no following RSA key characters matches as a secret.

2. Ineffective Entropy Calculation: We observe that tools employ Shannon entropy [186] to identify possible secrets. Though the core Shannon entropy algorithm is correct, differentiating secrets from false positives is not always effective. For example, TruffleHog computes the entropy of “2b95710rD1e6287e69Z8f2E24373449d879b70c7601B3x9” and “ThisIsAReallyLongString” as 4.08 and 4.11 respectively, thus having higher entropy score for the latter [187]. As a result, the dummy string is termed as a secret. We also observed substantial instances of GitHub commit ids, such as “0e2b3d4e3dec5f38ae95f62519eb2736f73c0b”, outputted as secrets because of ineffective entropy calculation.

3. Insufficient Filters/Prefix Regex: We observe that tools apply filters for HTML attributes and CSS selectors. For example, Repo-supervisor applies regex to prevent false positives such as “input[val=‘test’]” and “button[value=‘submit’]” [188]. However, the filters are insufficient as we observed strings such as “shape=rect;rounded=1” and “child{margin-bottom:10px;}” are still marked as secrets. In addition, tools apply prefix regex to ensure that at least one of the specified keywords related to the API key and token are within some characters (e.g., 40 characters) of the capturing group. For example, if a Strava API key is found by regex `[0-9a-z]{40}`, then the specified prefix regex checks whether the keyword “strava” is present within 40 characters of the capturing group [189]. However, checking with prefix keywords does not always prevent false positives. For example, TruffleHog applies regex `((?:g1pat|)[a-zA-Z0-9=]{20,22})` with “gitlab” as prefix keyword to identify GitLab tokens. However, for a string such as “https://docs.gitlab.com/gitlab-basics/add-file.html#add-a-file-using-the-command”, TruffleHog treats “add-a-file-using-the-” as a

token since the string matches the regex and the prefix keyword is present within 40 characters.

Analysis of False Negatives: Since we observed a low recall score by the tools, we inspected a random sample of 50 false negatives from each tool. We now discuss the reasons behind the low recall score.

1. Faulty Regex: We observe that tools miss secrets because of employing faulty regular expressions. For example, Whispers employ regex `(.[A-Za-z0-9_]+(key|token)$)`, which expects a secret will have a “key” or “token” word at the end. However, the “key” or “token” word can be present at the start of the context of the secret (`api_key="xxxx"`) or even not present at all, thus unable to capture secrets.

2. Skip Specific File Types: We observe tools skip specific file types while scanning. For example, ggshield does not scan HTML files to prevent false positives [190]. However, we observed that secrets are present in the HTML files either inside the HTML tags or in the JavaScript code embedded in the HTML files in a `<script></script>` tag. In addition, the HTML file type is in the top five file types containing secrets in the benchmark dataset (Table 6.2).

3. Insufficient Ruleset: We observe that tools do not have sufficient rulesets for all secret types. For example, TruffleHog does not have detectors for IGDB [191] and Mashape API [192] keys. As a result, since TruffleHog matches prefix keywords for a specific key, these API keys are not captured. We also observe that tools do not periodically add/update rules for detecting secrets. For example, the rules of the tools such as Whispers were last updated on August 25, 2021.

False Positive Secrets Dataset: We created a dataset of the false positives reported by the tools to expedite the research on improving the accuracy of the tools. The dataset is stored as a relational structured data in Google BigQuery (Dataset ID: `dev-range-332204.fp_secretbench`), and users can run SQL queries to access the dataset. However, the dataset may contain sensitive information, such as mislabeled true positives since the applied string-matching algorithms may mislabel the tool-reported secrets (Section 6.3.2). As a result, we will distribute only to fellow researchers and tool developers who should email the authors to access the dataset [151].

6.4.2 RQ2: What features are offered by the secret detection tools to aid in preventing secrets exposure?

Tools provide features to aid developers in preventing the exposure of secrets. We categorized the features into seven categories. Table 6.5 presents the features offered by each tool, which we discuss as follows.

F1: Pre-commit Hook Integration: Pre-commit hook is a VCS mechanism that can be used for any validation before a commit is pushed. Secret detection tools can be integrated into a pre-commit hook to prevent leaking secrets. The tools will scan the source code of the current commit and reject the commit if any secret is found. Developers can employ this feature in accordance with “shifting left” on security [193].

F2: CI/CD Integration: Secret detection tools offer integration with continuous integration and continuous deployment (CI/CD) pipelines such as GitHub Actions [194], Travis [195], and CircleCI [196]. As a result, if a secret is found in the deployment package, the deployment can be rejected.

F3: Custom Rule: Tools support adding custom rules, thus allowing developers to devise rules to detect known secrets. Tools allow adding custom detectors using regex or keywords for scanning secrets. In addition, tools support custom rules for ignoring secrets. If a dummy secret is knowingly committed in the source code, developers can devise rules to ignore that secret to reduce the false positive warnings from the tools.

F4: Secret Verification: If any potential secret is detected, the tool verifies the validity of the secret by calling the endpoint provided by the respective API vendor to reduce false positives. For example, TruffleHog’s AWS credential detector [197] performs a “GetCallerIdentity” API call against the AWS API to verify if the credential is active. In addition, if the secret is validated, GitHub Secret Scanner notifies the repository administrators and owners through email.

F5: Remediation Steps: Tools provide remediation workflows when a secret is detected to revoke and rotate the secret quickly. Tools assign the detected secret to the developer who leaked the secret. The developer can resolve the secret alert either by revoking the secret or marking it as a false positive. Tools also use developer feedback to improve their algorithm to reduce false positives. In addition, tools also provide suggestions, such as removing the secret from Git history and reviewing access logs to nullify the threat completely.

F6: Infrastructure as Code (IaC) Script Scan: Scanning for secrets in IaC script is essential as Rahman et al. [20] identified hard-coded secret is the most occurring security smell within IaC scripts. SpectralOps and ggshield provide support for scanning secrets in IaC

scripts.

F7: Non-source Code Scan: Developers can expose secrets in screenshots added as images in a repository and non-searchable PDFs shared for tutorials. These secrets can not be captured using regular regex matches. However, Commercial X employs Object Character Recognition (OCR) to detect secrets in images and non-searchable PDFs.

Table 6.5: Seven categories of features and additional secrets metadata provided by each tool.

Tool	Tools Feature							Secrets Metadata		
	Pre-commit Hook	CI/CD Integration	Custom Rule	Secret Verification	Remediation Steps	laC Script Scan	Non-source Code Scan	Commit ID	File Path	Line No.
git-secrets	✓		✓					✓	✓	✓
Gitleaks	✓		✓					✓	✓	✓
Repo-supervisor	✓	✓							✓	
TruffleHog	✓	✓	✓	✓	✓			✓	✓	✓
Whispers			✓					✓	✓	✓
Commercial X		✓	✓		✓		✓	✓	✓	✓
ggshield	✓	✓	✓		✓	✓		✓	✓	✓
GitHub-scanner			✓	✓	✓			✓	✓	✓
Spectralops	✓	✓	✓	✓	✓	✓		✓	✓	✓

6.5 Discussion and Recommendations

In this section, we discuss our findings and make recommendations.

Developers should employ tools based on the type of secrets present in the project.

Table 6.3 shows that tools miss secrets as the recall (Case 2) varies between 17% and 88%. However, if developers know the secret types present in the project, selecting tools based on secret types can yield higher recall. For example, for “Database Server and URLs” category, the recall (Case 2) score of TruffleHog is 98% (Table 6.4), whereas the overall recall (Case 2) score is 52% (Table 6.3).

Tool vendors should update detection rules periodically. According to the State of APIs Report from Rapid [198], API types are expanding, and API adoption is on the surge, with 63% of developers relying more on APIs in 2022. However, we observe that tools do not update the detection rules for API keys and tokens. For example, the rules of Whispers were last updated on August 25, 2021. We recommend tool vendors to update detection rules periodically to prevent missing secrets.

Tool vendors should correctly employ secret verification by collaborating with API vendors. We find that tools verify the found secrets with the API endpoints (F4). As a result,

tools show relatively higher precision by reducing false positives. For example, before the verification option was enabled (`-only-verified`), TruffleHog’s precision was 6%, outputting almost 100K alerts for our benchmark. In contrast, the precision changed to 90% when the verification was enabled and outputted only 611 secrets. However, verification methods are not 100% correct as we observe 10% false positives. For example, the tool tagged dummy server URLs such as “`http://dyn.example.com:password@dyn.dns.he.net`” as secrets. In addition, TruffleHog does not report an active secret if the API endpoint is unreachable [199]. We also find that GitHub has a secret scanning partner program [200] where API vendors can join in scanning their API keys and tokens in GitHub repositories and receiving notifications for quick remediation. However, only 66 API vendors have joined the program [201]. Therefore, we recommend that API vendors collaborate with tool vendors in correctly employing secret verification to prevent the exposure of secrets.

Tool vendors should develop automated technology to revoke and rotate secrets as remediation steps quickly. We find that tools provide remediation workflows when a secret is detected (F5). However, currently, the workflow is a manual process where the leaked secret is assigned to the developer to revoke and rotate the secret. In addition, developers have to sanitize the Git history by themselves using history sanitizing tools such as BFG repo-cleaner [65]. However, recent research [202] shows that malicious actors take only one minute to start making calls with the leaked API keys. Therefore, we suggest that tool vendors develop an automated workflow that the organization can employ in their system. The organization can mark the used secrets, and if a secret is reported that are among the used secrets, the workflow will automatically revoke and rotate the secrets. In addition, the workflow will sanitize the Git history without developers’ manual effort, deploy new artifacts if needed, and review access logs to find any breaches.

6.6 Ethics and Data Protection

Since the dataset of false positive secrets may contain mislabeled true positives, we will distribute the dataset selectively. To prevent unethical use, researchers and tool developers will sign a data protection agreement with us. Following that, we will use their email addresses to grant them access to our dataset from Google BigQuery. In addition, we have redacted/obfuscated example secrets presented in the chapter.

6.7 Threats to Validity

In this section, we discuss the limitations of our study.

Tool Selection: Our study’s list of tools is not exhaustive. Though we have chosen the tools based on the selection criteria mentioned in Section 6.2.1, we could not access proprietary tools such as Cycode [161] and CredScan [160]. As a result, we do not claim the findings we have in Section 6.4 to be generalizable for all tools.

Benchmark Dataset: Our selection of benchmark dataset is susceptible to bias. Basak et al. [152] curated SecretBench using open-source tools Gitleaks and TruffleHog, which also poses bias to the result of these two tools. However, they manually inspected and labeled each extracted secret using the tools. Out of 97,479 reported secrets of these two tools, 15,084 are true secrets. We used the true secrets to compare the tools of our study. SecretBench also has the drawback of only extracting secrets from GitHub repositories rather than from other VCSs, such as GitLab and BitBucket. Since SecretBench is the only publicly-available dataset, we could not compare the tools with another benchmark dataset to mitigate the potential bias.

Secrets Matching: We employed two string matching algorithms, Jaro-Winkler Similarity, and Gestalt Pattern Match, to match a secret with the benchmark for some tools. The similarity cut-off scores for both the algorithm we chose poses a threat to internal validity. However, we randomly selected 100 unique reported secrets from each tool and found that the combination of both algorithms’ cut-off scores correctly labeled 97% of the secrets.

Precision for Each Secret Category: We have the category of a secret and the number of secrets in a category of benchmark dataset. As a result, we could calculate the recall of each category by checking if the secrets of the specific category of the benchmark are present in the tool-reported secrets. However, we could not calculate the precision for a category since the tool can output false positives, which requires manual inspection for categorization.

6.8 Conclusion

We investigated five open-source and four proprietary secret detection tools against a benchmark dataset containing 818 GitHub repositories. We found that the top three tools based on precision are: GitHub Secret Scanner (75%), Gitleaks (46%), and Commercial X (25%), and based on recall are: Gitleaks (88%), SpectralOps (67%) and TruffleHog (52%). We also provided tools performance based on secret type to aid developers select the best tools for their use cases. Our manual analysis of the reported false positives indicates that generic

regex and ineffective entropy calculation are the reasons for high false positives. We also analyzed the false negatives and found that faulty regex, skipping file types, and insufficient rulesets for secret detection are the reasons for low recall. In addition, we provided a dataset of false positives to expedite the research in secret detection. We also categorized the features offered by the secret detection tools to aid in preventing the exposure of secrets. We recommend developers choose tools based on secret types present in the project to prevent missing secrets. In addition, we recommend future research on developing an automated technology for quick remediation of the exposed secret.

CHAPTER

7

STUDY V: ASSETHARVESTER: A STATIC ANALYSIS TOOL FOR DETECTING SECRET-ASSET PAIRS IN SOFTWARE ARTIFACTS

At present, many open-source and proprietary secret detection tools, such as TruffleHog [14] and GGShield [172], are available to prevent leaking secrets. However, Basak et al. [203] investigated five open-source and four proprietary secret detection tools and found that five of these nine tools demonstrate a precision of less than 7%. The tool with the highest precision (75%) among the nine tools misses many secrets, having only 3% recall. Thus, developers may develop “alert-fatigue” [150] and start to ignore the warnings reported by the tools.

A secret in a software artifact protects an asset (a database or an API service) accessible through asset identifiers (a URL, a DNS name, or an IP address). However, a secret may look like a false positive that protects a real asset. For example, Figure 7.1a shows a customer

```

1  return await aiomysql.connect(
2      host='120.77.222.216', -- Public IP Address
3      db='customer',
4      port=3306, user='root'
5      password='123456')

```

(a) A public IP address protected by a secret

```

1  db = pymysql.connect(host='localhost', -- Localhost
2      db='test',
3      user='root',
4      passwd='332315Yuan@',
5      port=3306)

```

(b) A localhost protected by a secret

Figure 7.1: A secret can protect both real and non-sensitive assets, such as a public IP address (real) and localhost (non-sensitive).

database with a public IP address (“120.77.222.216”) protected by the password “123456”. On the contrary, a secret may look like a true positive but protects a non-sensitive asset. For example, Figure 7.1b shows the password “332315Yuan@” protects a test database of a “localhost” that is typically not vulnerable to outside attackers. However, existing secret detection tools do not provide the asset information corresponding to a secret. As a result, developers manually filter alerts based on the secret value without the asset information and may ignore a secret protecting a valuable asset. In addition, developers may lose their development time while manually identifying the asset for each secret reported by the tools. Thus, programmatically identifying the assets protected by the secrets can aid in reducing the manual effort of developers to filter false positives and identify secrets that protect valuable assets. Additionally, developers can prioritize efforts to remove secrets based on the asset context.

The goal of our study is to aid software practitioners in prioritizing secrets removal by providing the assets information protected by the secrets through our novel static analysis tool.

However, identifying the assets protected by the secrets is not straightforward since

the asset identifiers can have multiple parts defined separately in the source code. For example, a database server address contains a host, port, and database name. In addition, multiple assets can be present in the same file (such as a configuration file). Additionally, the asset can be distant from where the secret is defined in the source code. For example, a secret is disclosed in one file, and the corresponding asset is disclosed in another file of the repository. Thus, even if we identify the asset, mapping the asset to the correct secret is challenging. In this study, we investigate how we can programmatically identify both the secret and the asset protected by the secret and provide answers to our research questions:

- **RQ1:** What are the secret-asset co-location patterns present in software artifacts? (Section 7.3)
- **RQ2:** What performance can be achieved in detecting assets protected by secrets via static analysis in terms of precision, recall, and F1-score? (Section 7.5)

We curated AssetBench, a benchmark of 1,791 secret-asset pairs of four database types extracted from 188 public GitHub repositories. To answer RQ1, we investigated and categorized the secret-asset co-location patterns in the source code. To answer RQ2, we constructed AssetHarvester, a static analysis tool to identify the database secret-asset pairs. In constructing AssetHarvester, we utilized pattern matching, data flow analysis, and fast-approximation heuristics to detect the secret-asset pairs. We evaluated the performance of AssetHarvester against AssetBench in terms of precision, recall, and F1-score. We provide a summary of our contributions as follows:

- We constructed AssetHarvester, a static analysis tool to detect the assets protected by secrets to aid developers in prioritizing secrets removal. Information on the asset protected by the secret aids in the prioritization. Additionally, AssetHarvester has shown 43% and 50% increase in precision and recall, respectively, for database secret detection compared to existing detection tools through the detection of assets.
- We have made the implementation of AssetHarvester publicly available [204]. Additionally, we provided AssetBench, a dataset of secret-asset pairs that can be extended and utilized by researchers and tool developers for future research and tool development.

The rest of the chapter is structured as follows: Section 7.1 introduces the selection process of asset types, followed by the benchmark dataset curation and the secret-asset

co-location patterns. We discuss the AssetHarvester construction and evaluation results of AssetHarvester against AssetBench in Sections 7.4 and 7.5, followed by the implications of our work. We discuss the ethics and limitations of our study in Sections 7.7 and 7.8, respectively. We draw the conclusion of our study in Section 7.9.

7.1 Asset Type Selection

A software artifact may contain different types of assets, such as database server addresses and API URLs, which are protected by secrets. However, the 2024 GitGuardian report [4] reveals that out of 12 million exposed secrets in GitHub, the top secret type is database providers. Additionally, database assets can be challenging to detect due to multiple asset identifier formats, among other asset types. For example, Figure 7.2 shows a database asset identifier can have multiple parts (host, port, and database name) defined separately in the same line (line 2) or different lines (lines 6-8). The database asset can also be in the same string (line 13). Thus, we selected database secret-asset pairs to be detected in this study. Since multiple database providers are present, we need to narrow our scope to maintain our study’s feasibility. We observed that the top five databases developers use are PostgreSQL [205], MySQL [206], SQLite [207], MongoDB [208], and SQL Server [209], according to the Stack Overflow Developer Survey 2023 [210]. However, we excluded SQLite from our study since SQLite is a file-based database requiring no authentication. Finally, we selected these four databases for our study.

7.2 AssetBench

To create a dataset of secret-asset pairs, we started with SecretBench [152], a publicly available benchmark dataset of software secrets. We accessed the dataset through Google Cloud Storage (Bucket Name: *secretbench*) and Google BigQuery (Dataset ID: *dev-range-332204.secretbench.secrets*). The authors curated 818 repositories from the September 2022 snapshot of Google BigQuery Public Dataset of GitHub [128] (Dataset ID: *bigquery-public-data.github_repos*). The repositories in the dataset comprise source codes of 49 programming languages. The secrets present in the repositories are extracted using two open-source secret detection tools, TruffleHog [14] and Gitleaks [138]. The dataset contains 97,479 labeled plain-text secrets, manually labeled as true or false by two authors of the SecretBench. In addition, the dataset provides additional metadata such as repository name,

```

1 # Database asset defined separately in the same line
2 pymysql.connect(host='10.0.0.1', port=3306, db='test',
3                 user='sa', password='root')
4
5 # Database asset defined separately in different lines
6 pymysql.connect(host='10.0.0.1',
7                 port=3306,
8                 db='test',
9                 user='sa',
10                password='root')
11
12 # Database asset defined in the same string.
13 pymysql.connect('mysql://sa:root@10.0.0.1:3306/test')

```

Figure 7.2: The database asset identifier has three parts (host, port, and database name) that are defined separately in the same line, in separate lines, or together in the same string.

commit ID, file path, and line number where the secrets have been found. However, the dataset does not provide information regarding the assets protected by the secrets. Hence, we extended the dataset as *AssetBench* by identifying assets for each secret in our study.

Filtering Dataset: Before identifying assets, we applied the following selection criteria to filter SecretBench.

Criteria 1 (Programming Language): The 2023 GitGuardian report indicates that developers most frequently leaked secrets in repositories written in Python programming language [211]. Additionally, we observed that Python is third among 49 programming languages containing secrets in SecretBench. Thus, we chose repositories containing Python source code for our study. We selected 188 repositories from 818 repositories and 34,569 secrets from 97,479 secrets of SecretBench.

Criteria 2 (Database Type): A repository can contain different types of secrets, such as API keys and database credentials. We filtered the secrets of the selected four databases (Section 7.1) and selected 2,114 secrets from 34,569 secrets.

Identifying Assets: Next, the first and third authors manually inspected each secret independently using the repository name, commit ID, file path, and line number provided by the dataset and identified the asset protected by the secret. However, the asset may not

Table 7.1: Count of Secret-Asset pairs for four databases

Database Type	# Secret-Asset Pair	% of Pair
MySQL	777	43.4%
PostgreSQL	679	37.9%
MongoDB	310	17.3%
SQL Server	25	1.4%

be present in the same file where the secret is located. In such cases, both authors inspected the candidate asset-containing files in the repository. The asset’s value for each secret with additional metadata (the file path and line number where the asset is found) is collected. We observed the agreement of finding the secret-asset pairs with a Cohen’s Kappa [139] score of 0.96 between two authors, which indicates a “near perfect agreement” according to Landis and Koch’s interpretation [140]. The disagreements were resolved after a discussion between the two authors. However, neither author found corresponding assets for 323 secrets. We removed those secrets and selected 1,791 secret-asset pairs. In Table 8.2, we presented the database type and the number of secret-asset pairs with percentages for each type. AssetBench contains 25 secret-asset pairs for SQL Server, representing 1.4% of the total pairs. The relatively lower percentage might stem from SQL Server’s proprietary nature, leading to lesser adoption in open-source projects than available open-source databases.

Developer Survey: To evaluate whether the committer of the secret agrees with our identified asset for the secret, we conducted a developer survey. First, to avoid recall bias [141], we selected secret-asset pairs committed between 2021 and 2022 and identified 683 secret-asset pairs. Next, we filtered out secret-asset pairs that have a noreply (`xxx@users.noreply.github.com`) or GitHub Actions bot (`action@github.com`) commit email address [142] and selected 490 secret-asset pairs. Next, we randomly selected 100 secret-asset pairs to avoid selection bias [143] and emailed the developers to know their agreement with our labeling and the reason for any disagreements. In the email, we provided the secret-asset pair information with a screenshot of the code where the secret-asset pair is found. We received 27 responses out of 100, and all respondents agreed with our label.

Dataset Storage: Our curated dataset, AssetBench, is stored in Google BigQuery (Dataset ID: `dev-range-351901.assetbench.assets`) as a relational structured data. Users can access the dataset through SQL queries. However, due to the sensitive nature of the dataset, we

will provide access to the dataset to only selected researchers and tool developers. Those who require access need to contact us through email.

7.3 Secret-Asset Co-location Patterns

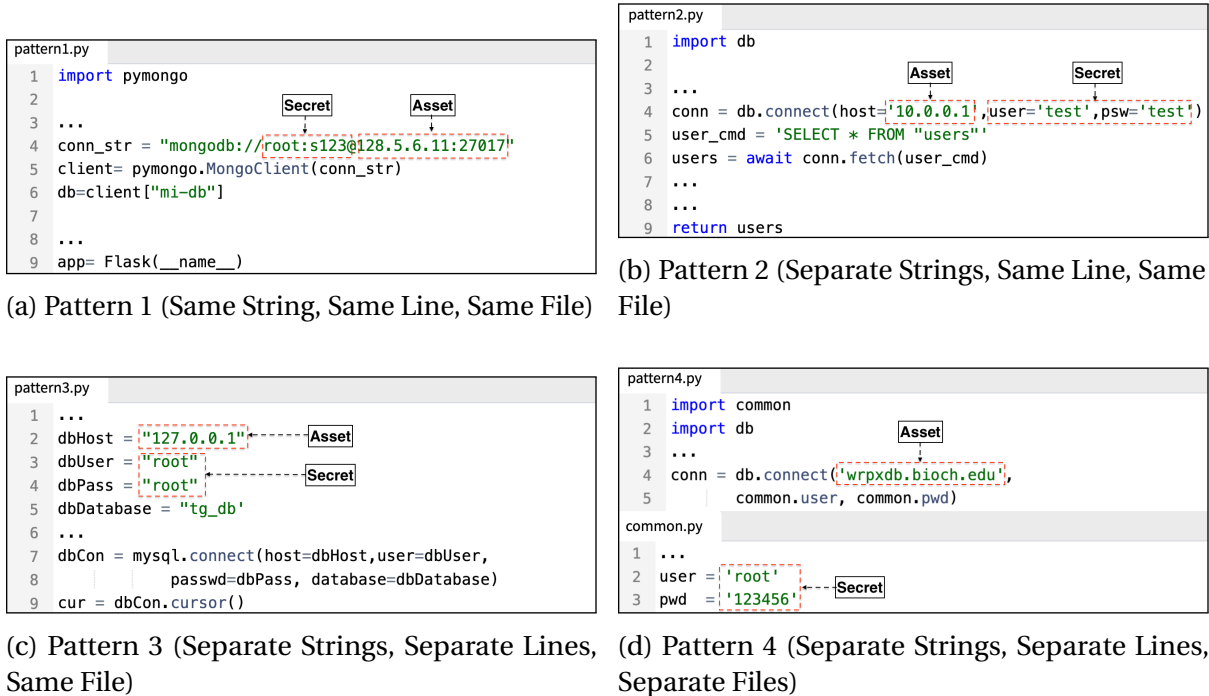


Figure 7.3: We identified four types of secret-asset co-location patterns in the source code.

To answer RQ1, the first and second authors independently inspected a random sample of 100 secret-asset pairs from the AssetBench. Both authors observed the location pattern of a secret and the corresponding asset and identified four mutually exclusive secret-asset co-location patterns. We utilized the co-location patterns for programmatically identifying secret-asset pairs in the construction of AssetHarvester, as described in Section 7.4. We now describe the four secret-asset co-location patterns. The number in parenthesis denotes the occurrences of each pattern found out of 100 secret-asset pairs.

Pattern 1 (Same String, Same Line, Same File) (54): The secret and the corresponding asset identifier can be present in the same string and the same line of a file, such as in a database connection string. For example, Figure 7.3a shows a MongoDB connection string

("mongodb://root:s123@128.5.6.11:27017") defined in line 4, where "root" and "s123" are the username and password of the database, and "128.5.6.11:27017" is the database server address.

Pattern 2 (Separate Strings, Same Line, Same File) (20): The secret and the corresponding asset identifier can be present in the same line of the file but defined separately in multiple strings. For example, the database server address ("10.0.0.1"), the username ("test"), and the password ("test") are defined and passed as separate arguments to `db.connect` method in line 4, as shown in Figure 7.3b.

Pattern 3 (Separate Strings, Separate Lines, Same File) (19): The secret and the corresponding asset identifier can be present in the same file of the repository but defined in separate strings and separate lines. For example, as shown in Figure 7.3c, the username ("root") and password ("root") are defined in lines 3 and 4, respectively, whereas the database server address ("127.0.0.1") is defined in line 2 of the same file.

Pattern 4 (Separate Strings, Separate Lines, Separate Files) (7): The secret and the corresponding asset identifier can be present in separate files of the repository. For example, as shown in Figure 7.3d, the username ("root") and password ("123456") of the database are defined in lines 2 and 3 of `common.py` file, respectively. The values of the `common.py` file are imported in line 1 of the `pattern4.py` file, where the database server address ("wrpxdb.bioch.edu") is defined in line 4. However, the secret and the asset may not always be present in the same file types. For example, both the files in Figure 7.3d have `.py` extension. However, the secret or asset can be defined in configuration files such as `config.yml` file, that can be read in a `.py` file.

7.4 AssetHarvester

We utilized the identified secret-asset co-location patterns (Section 7.3) and constructed AssetHarvester using pattern matching (Step 1), data flow analysis (Step 2), and fast-approximation heuristic (Step 3). We now discuss the three-step process of constructing AssetHarvester.

Step 1: Asset Finding Using Pattern Matching: We observed from Pattern 1 in Section 7.3 that the secret and the corresponding database asset are present in a database connection string. Since a database connection string follows a specific format, we can formulate regular expressions (regex) to identify the secret and the corresponding asset. We now discuss our approach to formulating the regex and identifying the assets protected by the

database server, and keys `Uid` and `Pwd` refer to the credentials in the connection. However, for OLE-DB, the key for the database server is `Data Source` instead of `Server`. Hence, we formulated a common regex with all the ODBC and OLE-DB key-value pairs for Group 2.

Group 3: (JDBC): Similar to ODBC and OLE-DB, Java Database Connectivity (JDBC) [217] is a standard API that allows Java applications to interact with different databases. In our study, though we selected repositories containing Python programming language, repositories can have Java source code containing JDBC connection strings. In addition, Packages such as `JayDeBeApi` [218] are available that allow the connection of a database using the JDBC connection string from the Python source code. As shown in Table 7.2, the JDBC connection string starts with "`jdbc`" prefix followed by the scheme type, server address, and database name. We observed that the username and password can be present in two forms, either before the server address or separately in the query parameters. We combined the two forms and formulated a common regex for Group 3.

To separate the secret and asset from the database connection string, we used the capturing group [219] feature of regex. The capturing group allows us to capture a specific part of the match. For example, as shown in Table 7.2, we implemented three capturing groups in the MySQL regex. The capturing group `<dbms>` captures the database type, `<credentials>` captures the username and password, and `server` captures the server address of the database.

Step 1.2 Identifying Secret-Asset Pairs Using Regex: In this step, we executed the regexes formulated in Step 1.1 to identify the database connection strings. We used the `re` [220] library of Python to execute the regexes. Since the database connection strings can be present in the Git commit history of a repository, we used `GitPython` [136], a Python library for traversing the commit history. In addition to the commit ID, file path, and line number of a match, we extracted the secret and the corresponding database asset from the connection string using the capturing group of regex.

Step 2: Asset Finding Using Data Flow Analysis: Among the four patterns described in Section 7.3, except Pattern 1, we observed that the secret and the corresponding database asset are not present in the same string. Instead, the secret and the corresponding database asset are defined separately and passed into a database driver function defined in the same or separate source file from where the secret and asset are present. For example, as shown in Figure 7.3c (Pattern 3), the database username, password, and the server address present in lines 3, 4, and 2, respectively, are passed into `mysql.connect` driver function defined in line 7.

For `AssetHarvester`, we utilized Data Flow analysis [221] to detect the flow of secrets

and assets into the database driver functions. Previous research [222] has used Data Flow analysis for security weakness propagation in the source code, such as the use of weak cryptographic algorithms. In a Data Flow analysis [221], the data flow among program elements of the entire source code is modeled through a Data Flow Graph (DFG). A DFG is a directed graph that consists of a set of nodes and a set of edges. The nodes in the DFG represent the semantic elements that carry values at runtime, whereas edges represent the way data flows between program elements. In a program, a node representing the origin of data is called the *Source*, whereas a node representing the destination of the data is called the *Sink*. In our study, a database secret and the corresponding asset are the *Sources*, and the database driver functions are the *Sinks*. We now describe the process of identifying the Python database drivers for our study. Additionally, we discuss the ways sources can flow into the database driver sinks and the process of identifying the secret-asset pairs from the sources and sinks.

Step 2.1 Identifying Database Drivers: To identify the Python database drivers, we constructed a set of search strings: *(MySQL OR PostgreSQL OR MongoDB OR SQL Server) AND (driver for Python)*. We selected the top 100 results from Google Search Engine for each search string. The stopping criteria for choosing the top 100 results are based on the grey literature search guideline in prior studies [18]. From the search result, we identified 12 database drivers grouped in 7 categories, which are presented in Table 8.5. We observed that in addition to identifying database drivers for the four databases, ODBC and JDBC, we identified two drivers, peewee [223] and SQLAlchemy [224] for the Object Relational Mapper (ORM) framework [225]. ORM is different than other drivers since ORM abstracts the database access with objects instead of directly managing the database access with SQL queries. The identified drivers have a function such as `connect` or `create_pool` to connect with the database. We observe that a driver function can have two different argument types (Positional and Keyword) [226], which act as the sinks for database username, password, and server address. The columns “Positional Argument” and “Keyword Argument” of Table 8.5 indicate which argument type each driver supports. We now discuss the two argument types as sinks for database secrets and assets.

1. Positional Argument: A positional argument [226] is passed to a function based on the position in the argument list without explicitly specifying the parameter name. Since the order of the position of the arguments is fixed, we know which positions will act as the database credentials (username and password) and asset (host, port, and database name) sinks. For example, as shown in Figure 7.4, the username, password, database name, and host address of the database are passed in a specific order in the `connect` function

Table 7.3: List of Python database drivers with their supported arguments for secret-asset pairs

Category	Driver Name	Positional Argument	Keyword Argument
MySQL	aiomysql [227]		✓
	mysql-connector [228]		✓
	PyMySQL [229]	✓	✓
PostgreSQL	aiopg [230]	✓	✓
	asyncpg [231]	✓	✓
	psycopg2 [232]	✓	✓
MongoDB	pymongo [233]		✓
SQL Server	pymssql [234]		✓
ODBC	pyodbc [235]	✓	
JDBC	JayDeBeApi [218]	✓	
ORM	peewee [223]	✓	✓
	SQLAlchemy [224]		✓

```

1 import asyncpg
2
3 ...
4 conn = await asyncpg.connect('root', 'root', 'test_db', '10.0.0.1')
5 ...

```

Figure 7.4: The database credentials and server address are passed in a specific order in the database driver function.

of `asyncpg`. Hence, we identified the sources that flow into each ordered position of the driver function for the database secrets and assets.

2. Keyword Argument: A keyword argument [226] (also called Named argument) is passed to a function by specifying the parameter name with the corresponding value. Unlike positional argument, the order of keyword argument is not fixed in a function. We observe that keyword arguments can be passed in separate parameter names and dictionary objects. As shown in Figure 7.5a, the username, password, database name, and host address of the database are passed in separate named arguments without fixed order, whereas defined in a dictionary object and passed in the function as shown in Figure 7.5b. Since we know the argument names, we can identify the sources flowing into the relevant arguments of the driver function for the database secrets and assets.

Table 7.4: Statistics of the presence of database assets in the neighboring lines of the secrets of the same file in AssetBench

Secret	Absolute Difference Between Secret and Asset Line Number (Number of Secret-Asset Pairs)				
	0	1	2	3	>=4
Database Password	407 (31.9%)	340 (26.7%)	349 (27.4%)	124 (9.7%)	54 (4.2%)

Step 2.3 Identifying Secret-Asset Pairs Using CodeQL and File Parsing: We observed that the database secret and the corresponding asset can be present in a configuration (config) file such as YAML, JSON, and XML files. The config file is read as a dictionary object, and the values of the dictionary object are accessed in the driver function. For example, as shown in Figure 7.6, the secret and the corresponding asset of MySQL database are present in the `config.yml` file, which is read in a dictionary object `cfg` of the `main.py` file (lines 5 and 6). The values of dictionary object `cfg` are accessed in the `aiomysql.connect` driver function (lines 8-11) using key names such as `dbhost` and `dbuser`. However, CodeQL does not support data flow analysis of source codes across multiple programming languages. As a result, the flow of secrets and assets from the `config.yml` file into the driver function of the `main.py` file can not be captured.

However, we observed that by utilizing the data flow analysis of CodeQL, we can find the config file name and the key names that flow into the driver function. Since we identified the config file name and associated the key names, we parsed the config file and retrieved the values for each key name. For retrieving the values from the YAML, JSON, and XML files, we used the `PyYAML` [239], `json` [240] and `xmltodict` [241] packages of Python, respectively.

Step 3: Asset Finding Using Fast-Approximation Heuristic: We observed that developers may have accidentally or intentionally kept the secret and the corresponding asset as commented lines in the source code. However, commented lines are ignored during data flow analysis. Additionally, capturing the data flow may not always be possible if the source code has dynamic behavior, such as extensive use of reflection. Thus, we can not identify the assets protected by secrets in those cases using data flow analysis. However, when the secret-asset pair is present in the same file, we observed from AssetBench that the database asset may be present in the neighbor lines of the corresponding secret. As shown in Table 7.4, the percentage of database assets present within three neighboring lines of the corresponding database password in the same file is 95.8%. Thus, we can check the neighboring lines of the secret line to identify the corresponding asset. In our study, we

```

main.py
1  import aiomysql
2  import pkgutil
3  import yaml
4  ...
5  conf = pkgutil.get_data(__package__, 'config.yml')
6  cfg = yaml.safe_load(conf)
7  ...
8  conn = await aiomysql.connect(
9      host=cfg['dbhost'], port=int(cfg['dbport']),
10     user=cfg['dbuser'], password=cfg['dbpassword'],
11     db=cfg['dbname'], loop=loop
12 )

config.yml
1  ...
2  dbhost: 127.0.0.1
3  dbport: 3306
4  dbname: test_db
5  dbuser: root
6  dbpassword: root
7  ...

```

Secret and Asset are read from config.yml file

Secret and Asset key-values are passed to driver function

Figure 7.6: The config.yml file contains the database secret-asset pair that is read in the main.py file. The secret-asset values are accessed by the key names and passed to the driver function.

define three neighboring lines as three lines above and three lines below the secret line. For example, if a secret is present in line 20, the asset can be present between lines 17 and 23. We now discuss the approach of identifying the secret-asset pairs using neighboring lines.

Step 3.1 Identifying and Filtering Secrets: First, we identified the secrets in the repositories using two open-source secret detection tools, TruffleHog [14] and Gitleaks [138]. The authors of SecretBench [152] have used the same two tools to curate the benchmark dataset as discussed in Section 7.2. Since the two tools can overlap outputting the same database secret in a repository, we merged the unique secrets. Next, we filtered the unique secrets for which we already found assets using Regex (Step 1) and Data Flow Analysis (Step 2).

```
1 fileserver: '10.10.0.1' ←-- File Server
2 mysql-host: '10.0.0.1' ←-- MySQL Server
3 mysql-user: 'pdns'
4 mysql-password: 'pdns'
5 ...
10 emailserver: 'mx.sendgrid.net' ←-- Email Server
11 mongo-user: 'root'
12 mongo-password: 'root'
```

Figure 7.7: Multiple or zero corresponding assets can be present in the neighboring lines of a secret.

Step 3.2 Identifying Secret-Asset Pairs Using Neighboring Lines: In this step, to identify the neighboring lines for each secret, we used the `linecache` [242] library of Python that provides random access to source code lines. We observe that a database asset identifier can be present as an IP address (10.0.0.1) or a DNS name (`wrpxdb.bioch.edu`), as shown in Pattern 2 and 4, respectively. Hence, we formulated regexes for capturing the IP addresses and DNS names in the neighboring lines. However, the neighboring lines may contain multiple IP addresses and DNS names. Among those assets, one asset can be the corresponding asset for a secret. For example, as shown in Figure 7.7, a file server and a MySQL database address are present in lines 1 and 2, respectively. The correct asset for the MySQL database username and password present in lines 3 and 4 is the MySQL database address. In addition, the asset protected by the secret may not be present in the source code. For example, a DNS name for an email server is present in line 10. However, the email server is not the asset protected by the MongoDB database username and password present in lines 11 and 12, respectively.

We observe that a specific group's secret and corresponding asset can have the same prefix in the variable or key names. For example, as shown in Figure 7.7, the key names of username (`mysql-user`), password (`mysql-password`) and server address (`mysql-host`) of MySQL database have the same prefix (`mysql`). However, the key name of the file server does not start with the same prefix as the key names of the MySQL database. Hence, we can apply a string-matching algorithm to calculate similarity scores between the secret line and the candidate asset lines and choose the asset with the highest similarity score. In addition, we discard the asset if the similarity score with the secret line is less than a threshold. To

Table 7.5: Precision, Recall and F1-score of AssetHarvester for each database type

Database Type	Precision (TP, FP)	Recall (TP, FN)	F1 Score
MySQL	0.98 (712, 13)	0.91 (712, 65)	0.94
PostgreSQL	0.98 (620, 10)	0.91 (620, 51)	0.94
MongoDB	0.96 (286, 11)	0.92 (286, 24)	0.94
SQL Server	1.00 (8, 0)	0.32 (8, 17)	0.48
Overall	0.97 (1626, 34)	0.90 (1626, 165)	0.94

calculate the similarity score, we used Jaro-Winkler Similarity [177], a string-matching algorithm that uses a prefix scale by giving a high similarity score to strings that match from the beginning. The Jaro-Winkler algorithm provides a similarity score between 0 and 1, and we chose 0.5 as the threshold similarity score. We utilized the `jaro_winkler_similarity` function from the `jellyfish` [179] package in Python to compute the similarity score and identify the secret-asset pairs.

7.5 Performance of AssetHarvester

In this section, we answer RQ2 by evaluating the performance of AssetHarvester against AssetBench.

Precision, Recall and F1-Score: Table 7.5 presents the precision, recall and F1-score of AssetHarvester for each database type. The column “Precision (TP, FP)” denotes the precision for each database type. The number in parenthesis denotes the number of true positive and false positive secret-asset pairs outputted by AssetHarvester. The column “Recall (TP, FN)” denotes the recall for each database type. The number in parenthesis denotes the number of true positive and false negative secret-asset pairs outputted by AssetHarvester. The column “F1 Score” denotes each database type’s F1-score (the harmonic mean of precision and recall). We now discuss our observations related to precision, recall, and F1-score.

- We observed that AssetHarvester demonstrated overall 97% precision, indicating high precise detection of assets protected by secrets with low false positives. The count of false positives (34) indicates that the tool incorrectly outputted 34 assets out of 1,791 secret-asset pairs.

- The overall recall score of AssetHarvester is 90%, indicating a strong ability to identify instances of assets for secrets, supported by an F1-score of 94%. The count of false negatives (165) indicates that the tool failed to detect 165 instances of secret-asset pairs.
- Among the four database types, the recall score of SQL Server is low (32%) though the precision score is 100%. The tool could not detect 17 instances of SQL Server assets out of 25 secret-asset pairs. We discussed the reason for false negatives later in this section.

Performance of Pattern Matching, Data Flow Analysis, and Fast-Approximation Heuristic: Figure 7.8 depicts that AssetHarvester detected unique secret-asset pairs using the three approaches, thus indicating the importance of the three approaches. Out of 1,626 secret-asset pairs, using pattern matching (regex) and data flow analysis (CodeQL), we found 1,090 and 404 unique secret-asset pairs, respectively. In addition, we found 111 unique secret-asset pairs using the fast-approximation heuristic (Neighboring Lines). However, we observed that 21 instances of secret-asset pairs were detected by both pattern matching and data flow analysis. The overlap happened because of `dsn` keyword argument of driver functions, which takes a connection string that is also matched by the regex of database types. However, the overlap is low since all the connection strings found by the regex are not passed to Python database driver functions. Instead, the connection strings are either passed to non-python such as Java or .NET database driver functions or not passed to any functions. Thus, those connection strings could not be captured by data flow analysis. Additionally, we observed that among the three approaches, AssetHarvester incorrectly detected 9 and 25 secret-asset pairs out of 34 false positives using pattern matching and fast-approximation heuristics, respectively. However, AssetHarvester did not detect any false positives using data flow analysis since we used specific sinks for database secret-asset pairs from the documentation compared to generic sinks implemented by GitHub CodeQL [243]. Additionally, we filtered sources flowing into sinks that do not point to primitive values (string or integer). We also filtered values with only colons or slashes that flowed into the sinks as part of the asset URL from string concatenation to avoid false positives. We now discuss our observations on the rules triggering the false positives and false negatives.

Analysis of False Positives: We observed that the false positives are mostly triggered by the neighboring lines rule (73.5% of the 34 false positives reported by AssetHarvester). We noticed that the key names of the secret and corresponding asset do not always follow the specific pattern of having similar prefixes (Step 3.2, Section 7.4). For example, “URL” and

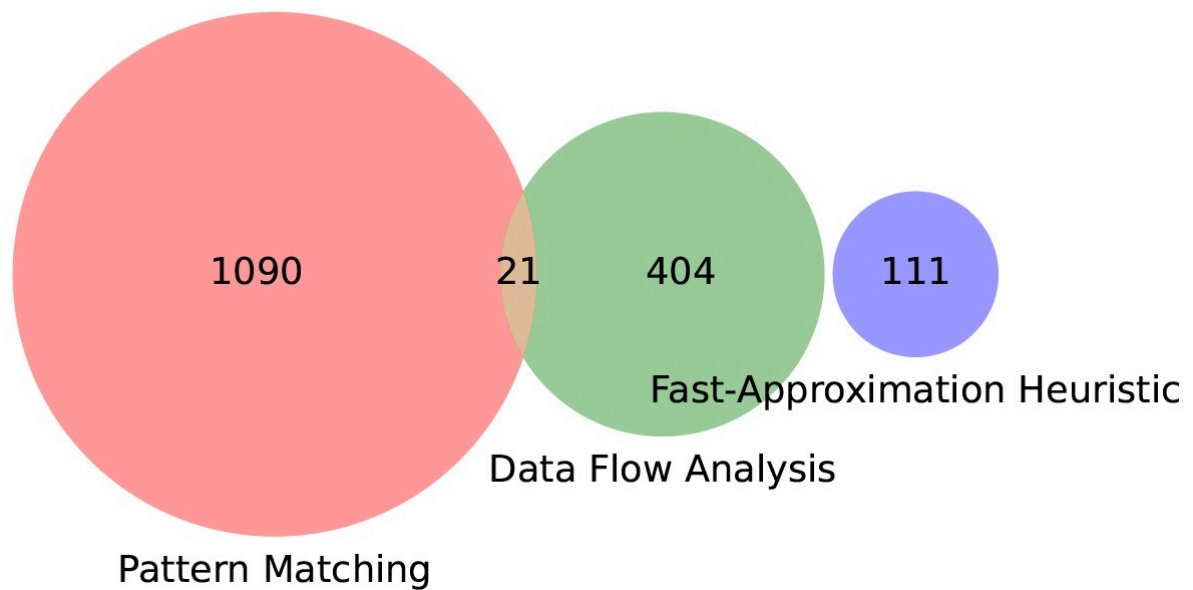


Figure 7.8: The number of unique secret-asset pairs found by Pattern Matching, Data Flow Analysis, and Fast-Approximation Heuristic approaches.

“password” are the key names of a database server address and password but do not have the same prefixes. As a result, when multiple IP addresses or DNS names were present in the neighboring lines, and the similarity score met the threshold, AssetHarvester could not detect the correct asset for a secret.

Analysis of False Negatives: We observed that AssetHarvester failed to detect secret-asset pairs when the asset is not present within three neighboring lines of the secret. As shown in Table 7.4, 54 (4.2%) instances of secret-asset pairs in AssetBench do not fall within three neighboring lines. In our study, the repositories also contain non-Python source codes, such as Java and .NET, where the secret-asset pairs are passed to Java and .NET database driver functions. However, AssetHarvester did not detect those secret-asset pairs since we only executed data flow analysis for Python source codes in our study. Thus, AssetHarvester shows a lower recall (32%) for SQL Server among other database types since the SQL Server assets are typically passed to .NET driver functions. Additionally, AssetHarvester could not detect assets present as variables in the connection strings not passed to Python driver functions. For example, the connection string "jdbc:postgresql://\${databaseServer}" contains the variable databaseServer defined separately with the actual value.

Comparison with Baseline Secret Detection Tools: Existing secret detection tools do

not detect the assets protected by secrets. Thus, we could only compare AssetHarvester’s performance on secret detection with the existing tools. Basak et al. [203] compared five open-source and four proprietary secret detection tools against SecretBench. We selected these nine baseline tools and evaluated them against 188 repositories containing 2,114 secrets of four databases, as curated in Section 7.2. Table 7.6 presents each tool’s precision and recall. We observed that the nine baseline tools show lower precision (less than 50%) and recall (less than 41%) scores than AssetHarvester (precision 92% and recall 90%). Additionally, we noticed that GitHub-scanner did not output any database secrets since the supported secret patterns lack database secret patterns [244]. However, GitHub introduced generic secret scanning for unstructured secrets such as database passwords, which we could not compare since the tool is in the beta phase and restricted to enterprise accounts only [245].

We observed that the lower precision of the tools is due to employing generic regex. For example, TruffleHog detects the database connection string but does not check if the connection string contains a password, thus outputting the connection string without a password as a secret. We resolved these false positives using the capturing group of regex for AssetHarvester (Step 1.2, Section 7.4). Additionally, the lower recall is due to employing insufficient rulesets and ineffective entropy calculation. For example, tools reject a secret based on entropy score. However, we found instances of secrets having lower entropy scores protecting real assets. We improved recall by identifying secrets flowing into the respective database driver functions using data flow analysis without considering the entropy score for AssetHarvester (Step 2, Section 7.4). In addition, we identified 86 secrets that are not present in SecretBench using data flow analysis. As discussed in Section 7.2, the authors of SecretBench used two open-source tools, TruffleHog and Gitleaks, to curate the benchmark dataset. These tools leverage regex and entropy scores to identify secrets. Thus, secrets that are not matched by the regex and entropy scores are missed.

Developer Survey: Since we leveraged a random sample of the dataset to construct AssetHarvester (Section 7.3), AssetHarvester’s evaluation against AssetBench is susceptible to bias. However, no other publicly-available benchmark dataset containing secret-asset pairs is present. Basak et al. [152] initially curated 89,070 candidate GitHub repositories for SecretBench. Since identifying and manually labeling secrets from 89,070 repositories was impractical, they finally selected 818 repositories using a multiset-multicover algorithm [152]. To mitigate bias, we selected a random sample of 15 repositories from 88,252 repositories (excluding 818 repositories of SecretBench) and identified 42 secret-asset pairs using AssetHarvester. Then, we conducted a developer survey with the committer of secret-

Table 7.6: Comparison of AssetHarvester with 9 Baseline Secret Detection Tools on Secret Detection

Tool	Precision (TP, FP)	Recall (TP, FN)
git-secrets [67]	0.04 (1460, 65)	0.01 (31, 2083)
Gitleaks [138]	0.21 (220, 45)	0.02 (37, 2077)
Repo-supervisor [169]	0.31 (1270, 391)	0.17 (364, 1750)
TruffleHog [14]	0.19 (5666, 1068)	0.40 (851, 1263)
Whispers [171]	0.06 (2203, 147)	0.05 (112, 2002)
Commercial X	0.35 (7260, 2511)	0.28 (589, 1525)
ggshield [172]	0.49 (1972, 969)	0.16 (330, 1784)
GitHub-scanner [176]	0.00 (0, 0)	0.00 (0, 0)
Spectralops [15]	0.15 (574, 88)	0.29 (609, 1505)
AssetHarvester	0.92 (2298, 89)	0.90 (1892, 222)

asset pairs to evaluate AssetHarvester’s performance. We received 18 responses, and 15 respondents agreed with our identified secret-asset pairs.

AssetHarvester’s Effectiveness Beyond Benchmark: We identified the secret-asset co-location patterns from a random sample of AssetBench containing database secret-asset pairs (Section 7.3). However, we selected a random sample of 10 secret-asset pairs for each of 7 non-database secret types, such as API keys, tokens, and private keys from SecretBench [152]. We found that the co-location of all the selected 70 secret-asset pairs matches our identified four co-location patterns, thus indicating the generality of the identified patterns. For example, as depicted in Pattern 1 (Same String, Same Line, Same File), the secret-asset pair can be present in the same string in a URL. Thus, we can identify the API key and corresponding server endpoint from the API URL, similar to a database connection string. Additionally, we did not limit AssetHarvester to the database drivers found in the random sample. Instead, we identified the database drivers from the database provider documentation. We found instances of only 3 database drivers in the random sample. However, we constructed AssetHarvester with 12 database drivers and detected secret-asset pairs from 9 database drivers in AssetBench. We also identified secret-asset pairs from one new database driver while evaluating AssetHarvester’s performance with 15 new repositories (Section 7.5).

7.6 Discussion

In this section, we discuss the implications of AssetHarvester from the findings of our study.

Data Flow Analysis aids in detecting all parts of a credential and the corresponding asset as one group. A credential can have multiple parts required to access the protected asset. For example, an access key ID and a secret access key are required to access an AWS resource, whereas for accessing a database, both a username and a password are required. Similar to credentials, assets can have multiple parts as well. For example, a database asset consists of a host, port, and database name. However, existing secret detection tools can not detect all parts of an asset if present separately in the source code. In addition, the tools output each part of a credential in separate alerts instead of outputting as one group. Thus, developers need to manually identify the related alerts out of all the alerts reported by the tools. However, AssetHarvester leverages data flow analysis to detect all parts of a credential and the asset and output as one alert to the developers. In our study, we detected the database credential (username and password) and asset (host, port, database name) flowing into the same database driver functions using data flow analysis. Additionally, we provided the information on the call location as an additional context for the developers to prioritize secret and asset eradication from the source code.

AssetHarvester can be extended to detect secret-asset pairs in other programming languages and non-database secret types. In our study, we detected secret-asset pairs of four database providers in Python programming language. However, the found secret-asset co-location patterns are generic and can be applied to other programming languages and non-database secret-asset pairs (Section 7.5). We now discuss the effort needed and challenges to extend AssetHarvester for other programming languages and secret types.

Programming Languages: Among the three techniques (pattern matching, data flow analysis, and fast approximation heuristics) we leveraged, pattern matching and fast approximation heuristics are programming language-agnostic. Hence, these two techniques can be applied to find secret-asset pairs in other programming languages without additional effort. On the contrary, the data flow analysis is dependent on the programming language. However, the abstract syntax tree, control flow, and data flow graph of the source code for each language in a repository can be computed separately, which CodeQL supports. Then, the queries to identify secret-asset pairs can be run on each of the computed graphs, thus extending for each programming language with minimal effort.

Non-Database Secret Types: We observed a random sample of 10 secrets for seven secret types such as private key, API key, and authentication token from SecretBench [152]. Next,

we categorized the secret-asset pairs into two categories. We now describe how we can extend AssetHarvester to detect secret-asset pairs except for the database provider.

1. Cloud Providers: For authentication and authorization with cloud providers, such as Google Cloud, API keys and tokens are used that can be present in an API URL or separately passed to a function. Since each cloud provider follows a specific format for API URLs, a regex can be formulated for each API and added to the regex list (Step 1.1, Section 7.4). Additionally, when the secret is not present in the API URL, the secret and the API URL can be detected by data flow analysis since these values are passed in a common HTTP request client (`get` and `post` methods). We observed that cloud secrets are the second most exposed secrets on GitHub, according to the 2024 GitGuardian report [4]. Additionally, they found a 1212-fold increase in leaked OpenAI API keys since 2022, driven by the rising use of large language models (LLMs). We will prioritize extending AssetHarvester to identify cloud provider secrets for future work.

2. Non-Database Servers: A secret can protect non-database servers, such as Mail and FTP servers. Similar to database servers (Group 1, Table 7.2), non-database servers have specific formats containing a scheme type (`scheme://user:password@host:port`). For example, the “smtp” or “pop3” are the Mail server’s scheme types, whereas “ftp” is the scheme type for the FTP server. We can add the scheme types in Group 1 of the regex list to capture the non-database server secret-asset pairs. In addition, the secret and the corresponding server URL are used by functions such as “login” and “SMTP” functions of `smtpplib` [246] module of Python for sending email. Web and Mail servers also use private keys to enable secure connections. These keys are stored in a file separately from the asset location and read from another function in the source code. Thus, we can leverage data flow analysis to retrieve the file name from the function and parse the file to identify the private key, as shown in Step 2.3 of Section 7.4.

Our list of regexes and sinks for AssetHarvester is configurable and requires no source code change to detect non-database secret types. Though identifying the regex and sinks for each secret type from the documentation requires manual effort, the process can be automated in the future. For example, LLMs can aid in identifying regex and sinks from source code patterns and vendor documentation for each secret type. We can leverage the knowledge gained from the manual analysis of our study and generate prompts for LLMs.

7.7 Ethics and Data Protection

Since our dataset contains sensitive information, such as the secret-asset pairs and the committer’s email addresses, we will distribute the dataset selectively. Researchers and tool developers who want to use our dataset will sign a data protection agreement with us to ensure ethical use. Once completed, access to our dataset will be granted via Google BigQuery using their email addresses. In addition, we did not attempt to use the secret-asset pairs to verify their validity. We only contacted the developers who committed the secret-asset pairs to validate our labeling. We did not disclose developers’ identities to management or higher officials within their organizations. Additionally, we are notifying every developer in our dataset to remove the secret-asset pairs from their VCS.

7.8 Threats to Validity

In this section, we discuss the limitations of our study.

Manual Analysis: Manual analysis can introduce bias due to the multiple interpretations and oversights. For example, identifying the assets protected by secrets while curating AssetBench is susceptible to bias. We mitigated the bias by cross-checking the identified secret-asset pairs with two raters.

Benchmark Dataset: Our selection of benchmark dataset for secrets is susceptible to bias. Basak et al. [152] utilized two open-source tools (TruffleHog and Gitleaks) and curated SecretBench, which we extended as AssetBench by identifying the protected asset for each secret. However, we observed that AssetHarvester identified 86 secrets not present in SecretBench (Section 7.5). Thus, SecretBench may have more missing secrets, impacting the results discussed in Section 7.5. Since SecretBench is the only publicly-available dataset and we constructed AssetHarvester from the random sample of AssetBench (Section 7.3), we evaluated AssetHarvester’s performance on 15 new GitHub repositories to mitigate the bias.

Developer Survey: For the developer survey of AssetBench, we selected the secret-asset pairs committed between 2021 and 2022. However, the developer’s responses could have recall bias. To mitigate the bias, we provided screenshots of the secret and asset-containing source code with metadata (commit ID, file path, and line number) to the developers.

Data Flow Analysis: In our study, we used CodeQL for data flow analysis in the latest snapshot of the repositories. CodeQL can only model the data flow with the provided snapshot of the source code. However, developers can push secret-asset pairs in one commit

and remove them in another commit. Secret-asset pairs can still be present in the old snapshot, that can not be detected by executing data flow analysis on the latest snapshot. However, we can compute data flow analysis for each repository snapshot to identify the secret-asset pairs from Git history, which will be impractical and time-consuming.

Neighboring Lines: Our selection of three neighboring lines for identifying the assets for a corresponding secret poses a threat to internal validity since the three-line range is selected from AssetBench containing only database secret-asset pairs. However, we selected a random sample of 50 non-database secrets from SecretBench and identified the corresponding assets. We found that the percentage of assets present within three neighboring lines of the corresponding secret is 96.3%, thus indicating the rule’s generalizability to other secret types.

7.9 Conclusion

We constructed AssetHarvester, a static analysis tool to detect the assets protected by the corresponding secrets in a repository by investigating the secret-asset co-location patterns. We utilized pattern matching, data flow analysis, and fast-approximation heuristics to construct AssetHarvester. To evaluate AssetHarvester, we curated AssetBench, a benchmark dataset of 1,791 secret-asset pairs comprising four database types (PostgreSQL, MySQL, MongoDB, and SQL Server). The secret-asset pairs are extracted from 188 public GitHub repositories. We found that AssetHarvester demonstrates precision of (97%), recall (90%), and F1-score (94%) in detecting secret-asset pairs. Our findings indicate that data flow analysis employed in AssetHarvester detects secret-asset pairs with 0% false positives and also aids in improving the recall of secret detection tools. In addition, though fast-approximation heuristics introduce relatively more false positives, this approach improves recall by detecting assets that cannot be detected using other approaches.

CHAPTER

8

STUDY VI: HOW CAN WE MITIGATE 12.8 MILLION CHECKED-IN SECRETS IN SOFTWARE ARTIFACTS?

Since 2020, GitGuardian has been detecting checked-in hard-coded secrets in GitHub repositories [4]. During 2020-2023, GitGuardian has observed an upward annual trend and a four-fold increase in hard-coded secrets, with 12.8 million exposed in 2023. However, removing all the secrets from software artifacts is not feasible. Rayhanur et al. [8] found developers ignoring secret detection tool alerts due to false positives, time pressure, and technical challenges. Additionally, the security risks of the secrets are not equal, protecting assets ranging from obsolete databases to sensitive medical data. Thus, secret removal should be prioritized by security risk reduction, which existing secret detection tools do not support. Developers may stop using the tools due to “alert fatigue” [150] if the alerts are not efficiently prioritized for secret removal.

Existing secret detection tools prioritize secret removal based on “severity”, a rating tied to the secret type [247]. For example, the same severity rating is assigned to any database

Table 8.1: Security Risk for Each Secret-Asset Pair

Secret-Asset Pair	Value of Asset	Ease of Attack	Security Risk
Pair 1	1	100	100
Pair 2	40	1	40
Pair 3	40	100	4000

secret without considering the protected asset information. However, the value of the asset protected by the secret can vary from a database with mock data to medical data whose breach can incur fines. Similarly, the ease of accessing an asset varies. For example, an asset with a public IP address can be easy for attackers to access. In contrast, an asset with a private IP address or localhost will require attackers to be on the same network or have physical access to the host machine.

The National Institute of Standards and Technology (NIST) defines the security risk of an entity as a function of the impact of the adverse event and the likelihood of the event by a threat source [9]. For a secret, the security risk can be defined as the function of the value of asset and the attacker’s ease of accessing the asset protected by the secret. Protection Poker [10], a threat modeling game, employs the product of relative measures of “value points” and “ease points” for security risk quantification, such as one requirement being five times easier to attack than another. Similarly, for a secret, the security risk can be defined as the *product of value of asset and ease of attack*. This security risk computation is based upon the hypothesis that attackers are more likely to succeed in attacking assets of high value and that are easier to attack. Table 8.1 provides an example of security risk computation for three secret-asset pairs. A secret-asset pair consists of a secret, such as a database password, and a protected asset by the secret, such as the database server. Pair 3 is deemed to have the highest security risk because the value of the asset is 40 times more valuable than Pair 1 and 100 times easier to attack than Pair 2. Thus, removing the Pair 3 secret from the source code is of primary importance. We hypothesize that providing a security risk score for each secret can aid developers in prioritizing the secret removal efforts. *The goal of our study is to aid software practitioners in prioritizing secrets removal efforts through our security risk-based tool.*

In this research, we studied how we can programmatically calculate the security risk score by identifying the value of asset and ease of attack for each secret-asset pair and provided answers to our research questions:

RQ1: What performance can be achieved in automatically identifying the value of

asset and ease of attack for secret-asset pairs in terms of precision, recall, and F1 score? (Section 8.3.1)

RQ2: Do developers prioritize secret removal based on the descending security risk scores? (Section 8.3.2)

We constructed RiskHarvester, a risk-based tool to provide security risk scores of database secret-asset pairs based on the value of asset and ease of attack. We calculated the value of asset by identifying the categories of sensitive data, such as personal information present in the database, from the database keywords (database, table, and column names). We utilized data flow analysis, SQL, and Object Relational Mapper (ORM) parsing to detect the database keywords from the source code. To calculate the ease of attack, we used passive network analysis to retrieve the database host information.

To answer RQ1, we constructed RiskBench, a benchmark of 1,791 database secret-asset pairs from 188 GitHub repositories. We manually inspected each secret-asset pair and included the database keywords, corresponding sensitive data categories, and valid host information in RiskBench. We evaluated RiskHarvester against RiskBench in identifying the database keywords, sensitive data categories, and valid hosts for each secret-asset pair. To answer RQ2, we conducted an online developer survey to understand whether developers use the descending security risk scores to prioritize secret removal from software artifacts. We hypothesize that developers will prioritize secret removal ranked by descending risk scores. We provided a summary of our contributions as follows:

- We automatically computed relative security risk scores of checked-in secrets to aid developers in prioritizing secret removal, which existing secret detection tools do not support. Additionally, we reported the developer study findings on the effectiveness of the security risk score in the alerts for secret removal prioritization.
- We made the implementation of RiskHarvester publicly available [2]. We also provided RiskBench, a dataset of secret-asset pairs to aid researchers and developers, available via a data protection agreement.

Our paper is organized as follows: Section 8.1 depicts our research methodology. We discuss the RiskHarvester construction and evaluation results against RiskBench in Sections 8.2 and 8.3, respectively, followed by the implications and limitations of our work. We conclude in Section 8.6, followed by ethics considerations and open science policy.

8.1 Research Methodology

In this section, we explain the process of RiskBench curation, identifying the value of asset and ease of attack patterns for calculating security risk score, and the developer survey.

8.1.1 RiskBench Curation

To select a dataset of secret-asset pairs for calculating security risk score, we started with AssetBench [248], a publicly-available dataset of secret-asset pairs. We accessed the dataset through Google Cloud (ID: dev-range-332204.assetbench). The authors of AssetBench curated 818 repositories from the September 2022 snapshot of GitHub’s Google BigQuery Dataset (ID: bigquery-public-data.github_repos) [128]. The dataset contains 97,479 manually labeled secrets (as true or false), extracted using two open-source secret detection tools, TruffleHog [14] and Gitleaks [138]. In addition, two authors of AssetBench manually inspected candidate asset-containing files to identify the assets protected by the corresponding secrets. The dataset also provides metadata such as repository name, commit ID, file path, and the line number where the secret-asset pairs have been identified. However, the dataset does not contain the database keywords (database, table, and column names) and corresponding sensitive data categories for each secret-asset pair. Additionally, the dataset lacks information on whether the asset identifier is a placeholder. In our study, we have utilized the database keywords, data categories, and asset identifiers for calculating the value of asset and ease of attack (Section 8.2). Thus, to evaluate the performance of RiskHarvester in identifying the value of asset and ease of attack for each secret-asset pair (RQ1), we extended the dataset as *RiskBench* by including the additional information.

Filtering Dataset: Before identifying the value of asset and ease of attack information for each secret-asset pair, we applied the following selection criteria to filter AssetBench.

Criteria 1 (Programming Language): According to the 2023 GitGuardian report [211], developers most frequently exposed secrets in source code written in Python in GitHub. Thus, we selected repositories containing Python source code in our study. We selected 188 repositories from 818 repositories and 34,569 secrets from 97,479 secrets of AssetBench.

Criteria 2 (Secret Type): The 2024 GitGuardian report [4] reveals that out of 12.8 million exposed secrets in public GitHub repositories, the top secret type is database secrets. Thus, we selected database secret-asset pairs to calculate the security risk score in our study. However, we need to narrow the scope to maintain our study’s feasibility since multiple database providers are present. We observed that according to the Stack Overflow Devel-

Table 8.2: Count of Secret-Asset pairs in RiskBench

Database Type	# Secret-Asset Pair	% of Pair
MySQL	777	43.4%
PostgreSQL	679	37.9%
MongoDB	310	17.3%
SQL Server	25	1.4%

oper Survey 2024 [249], the top five databases used by developers are PostgreSQL [205], MySQL [206], SQLite [207], SQL Server [209], and MongoDB [208]. However, we excluded SQLite since SQLite is a file-based database requiring no authentication. Finally, we filtered the secret-asset pairs of the four databases and selected 1,791 secret-asset pairs from 34,569 secrets.

Table 8.2 shows the number of secret-asset pairs of the four database types with the percentage of each type. We observed that only 25 secret-asset pairs (1.4% of the total) are present for SQL Server. The lower percentage may be due to SQL Server’s proprietary nature, limiting the adoption in open-source projects compared to open-source databases.

Identifying Database Keywords, Sensitive Data Categories, and Asset Identifier Information: To identify the database keywords, the first and second authors of the paper manually inspected each secret-asset pair using the repository name, commit ID, file path, and line number provided by the dataset. Since the keywords, such as table and column names, may not be present in the same file where the asset is located, both authors inspected the candidate database keywords containing files in the repository. Finally, the database name, corresponding table, and column names are collected.

Next, to assign a sensitive data category for each database keyword, we utilized the data categories provided by Google Cloud Data Loss Prevention (DLP) [250]. The Google Cloud DLP is a service that helps organizations discover and classify sensitive data to comply with GDPR, HIPAA, and PCI-DSS regulations [250]. The Google Cloud DLP provides 192 sensitive data categories grouped into 7 domains. These 7 domains include Personally Identifiable Information (PII), such as a name, and Sensitive Personally Identifiable Information (SPII), such as a passport number. In addition, each data category is assigned a sensitivity level (“HIGH”, “MODERATE” and “LOW”). Table 8.3 presents an example of a data category and the corresponding sensitivity level in each domain. However, we observed that data categories contain similar information. For example, data categories such as “CANADA_PASSPORT” and “US_PASSPORT” contain passport information, and these data categories

Table 8.3: Examples of a data category with the corresponding sensitivity level for seven domains provided by Google Cloud DLP. The full list can be found online [2].

Domain	Data Category	Sensitivity
PII	PERSON_NAME	MODERATE
SPII	PASSPORT	HIGH
DEMOGRAPHIC	GENDER	MODERATE
CREDENTIAL	AUTH_TOKEN	HIGH
GOVERNMENT_ID	VAT_NUMBER	HIGH
DOCUMENT	RESUME	MODERATE
CONTEXTUAL_INFORMATION	ORGANIZATION_NAME	LOW

have the same sensitivity level. Since we will compute the similarity score of a database keyword with the data category (Step 2.2, Section 8.2) to assign the correct data category, reducing the number of comparisons will improve the mapping performance. Thus, we manually inspected each data category and merged similar categories into a generic data category such as “PASSPORT”. Finally, we identified 113 data categories. Both authors independently assigned a data category to each database keyword.

Next, both authors inspected the asset identifier for a secret and labeled whether the database host is a placeholder considering the asset source code context. The agreement of finding the database keywords, corresponding data categories, and if the host is a placeholder with a Cohen’s Kappa [139] score of 0.88, 0.93, and 0.85, respectively, between the two authors. These scores indicate a “near perfect agreement” according to Landis and Koch’s interpretation [140]. The disagreements were resolved after a discussion between the two authors.

8.1.2 Value of Asset and Ease of Attack Patterns

For each secret-asset pair, we calculated the security risk score as the product of the value of asset and ease of attack, as defined in Equation 8.1. This security risk computation is based upon the hypothesis that attackers are more likely to succeed in attacking assets of high value and that are easier to attack.

$$\text{Security Risk} = (\text{Value of Asset}) \times (\text{Ease of Attack}) \tag{8.1}$$

The first and second authors independently inspected a random sample of 50 secret-asset pairs from RiskBench and developed patterns for programmatically calculating the

value of asset and ease of attack for each secret-asset pair. Now, we describe the observed patterns, which form the basis of RiskHarvester construction in calculating security risk score (Section 8.2). The source code snippets of Figure 8.1 and 8.2 are taken from RiskBench repositories of the 50 secret-asset pairs.

Value of Asset Patterns: A database asset identifier has three parts (host, port, and database name) [251]. We observed that the asset's value can be inferred from the database name since the name reveals the type of data the database contains. Figure 8.1a shows that a patient ("db_patient") and a test log ("log_test") database is passed in the "db" arguments where the value of patient database will be relatively higher than that of a test log database. However, we may not always be able to infer the database's data from the database name. Figure 8.1b shows that the database name is "my-db" (line 5), posing difficulty in inferring the value of the asset. However, we observed that the database asset has been configured in line 7, and a SQL query is executed to access the user table containing the email column. Thus, we can retrieve the type of data of the database from the table and column names which we used to calculate the value of asset (Step 2.2, Section 8.2).

We now describe the three mutually exclusive patterns observed in the database, table, and column name locations for an asset identifier. The numbers in parentheses denote occurrences of each pattern in 50 secret-asset pairs.

V-Pattern 1 (SQL Database Driver Calls) (25): We observed that the database name and the corresponding table and column names of relational databases can be found in the SQL database drivers, such as MySQL drivers [229]. These SQL database drivers provide functions to connect to databases, execute queries, manage transactions, and fetch results. We observed that the database name is passed in the same function where the database secret and server address information is also passed to set up the connection. For example, Figure 8.2a shows that the database name ("db_patient") is passed in `pymysql.connect` function (line 4). Additionally, the corresponding database table and column names are present in raw SQL queries passed in query functions of SQL database drivers such as the "execute" function (line 7, Figure 8.2a).

V-Pattern 2 (NoSQL Database Driver Calls) (16): We observed that the database name and the corresponding table and column names of non-relational databases can be found in the NoSQL database drivers, such as MongoDB drivers [252]. Unlike relational databases, non-relational databases are document or key-value pair databases without a structured schema. The table and column names are referred to as collection and field names, respectively. Figure 8.2b shows that the database name and collection name are passed as a dictionary key to the NoSQL driver client and db instance in lines 2 and 3, respectively.

```
1 pool = await aiomysql.create_pool(  
2     user='root', password='123456',  
3     host='localhost', port='3333',  
4     db='db_patient'})  
5  
6 db = pymysql.connect(  
7     host='192.168.1.2', port='3306',  
8     db='log_test',  
9     user='test', password='test')
```

(a) The value of asset can be inferred from the database name

```
1 mysqlconfig = {  
2     "host": "sh1.cirray.cn",  
3     "user": "milton",  
4     "password": "potr0987",  
5     "db": "my-db"}  
6  
7 conn = await aiomysql.connect(**mysqlconfig)  
8 cur = await conn.cursor()  
9 cur.execute(f"SELECT * FROM user WHERE email='{email}'")  
10 res = await cur.fetchone()
```

(b) The value of asset can be inferred from table and column names

Figure 8.1: Asset’s value can be inferred from the database name, table names, and column names from the source code.

However, the field names are passed as key-value pairs in a dictionary object instead of as a raw SQL string to the driver query function (lines 7-8), where each key is the field name of the corresponding collection.

V-Pattern 3 (ORM Framework Calls) (9): We observed that the database name and the corresponding table and column names of relation databases can be found in Object Relational Mapper (ORM) framework calls [225], such as SQLAlchemy [224]. Unlike other drivers, ORM abstracts database access through objects rather than directly managing the

```

1 db = pymysql.connect(
2     user='root', password='root',
3     host='192.168.1.2', port='3306',
4     db='db_patient')
5
6 cur = await db.cursor()
7 cur.execute(f"SELECT patient_name, blood_type
8             FROM patient WHERE pid={pid}")
9 ...

```

(a) V-Pattern 1 (SQL Database Driver Calls)

```

1 client = MongoClient('localhost', 27017)
2 db = client['tint-data']
3 user_collection = db['user']
4
5 def update_user(username, token):
6     res = user_collection.update(
7         {'username': username,
8          'oauth_token': token})
9 ...

```

(b) V-Pattern 2 (NoSQL Database Driver Calls)

```

1 from flask.ext.sqlalchemy import SQLAlchemy
2 ...
3 app = Flask(__name__)
4 app.config['SQLALCHEMY_DATABASE_URI'] =
5     'mysql://root:admin@10.10.0.1/portfolio'
6
7 db = SQLAlchemy(app)
8 ...
9
10 class User(db.Model):
11     __tablename__ = 'user'
12
13     id = db.Column(db.Integer, primary_key=True)
14     username = db.Column(db.String(), unique=True)
15     password = db.Column(db.String())
16
17     def __init__(self, username, password):
18         self.username = username
19         self.password = password

```

(c) V-Pattern 3 (ORM Framework Calls)

Figure 8.2: We identified three patterns to locate database, table, and column names for each secret-asset pair in the source code.

access with raw SQL queries. Figure 8.2c shows that the database name ("portfolio") is defined in a connection string along with secret and server address and passed to ORM configuration (line 4). Since ORM maps tables in a relational database to classes and rows to instances of those classes, the table name and column names can be found in these classes. The "__tablename__" attribute defines the table name (line 11), and the other attributes define the column names, such as username and password (lines 14-15), as shown in Figure 8.2c.

Ease of Attack Patterns: Similar to the value of an asset, ease of accessing the asset can vary based on multiple factors. For example, attackers can more easily access an asset with a public IP address, whereas attackers can not directly access an asset on a localhost. Since the server address is defined as a combination of host and port, we can infer the ease of accessing the asset from these parts of the asset identifier.

Now, we describe the observed four patterns in the asset identifiers in the source code. The numbers in parentheses denote occurrences of each pattern in 50 secret-asset pairs.

E-Pattern 1 (DNS Name) (27): We observed that developers put a DNS name, such as ("sh1.cirray.cn"), in the host part of the asset identifier as a database server address in the source code. However, not all the DNS names are resolved to IP addresses due to non-existent domains, misconfigured DNS records, or expired domains. In addition,

we observed invalid DNS names that are not valid according to DNS name format or a placeholder/dummy such as "your-project-name.com". Thus, the ease of attack of a secret-asset pair differs if DNS is resolvable (Steps 3.1.1 and 3.1.2, Section 8.2).

E-Pattern 2 (IP Address) (23): Developers put IP addresses such as ("185.60.21.35") instead of DNS name as database server addresses in the host part of the asset identifier. However, not all the IP addresses are routable addresses. For example, localhost address ("127.0.0.1") or private IP addresses ("192.168.1.1") pose more difficulty to external attackers to access the asset than public IP addresses. We also observed invalid or placeholder IP addresses such as "x.x.x.x" or "0.0.0.0" that attackers can not leverage to access the asset. Thus, the ease of attack of a secret-asset pair differs if the IP address is routable (Steps 3.1.3 and 3.1.4, Section 8.2).

E-Pattern 3 (Scannable) (7): Not all the public IP addresses that are either present directly in the host or resolved from the DNS names are scannable since IP addresses can be present behind firewalls or other security measures. Scannable means the discovery of active services through network scans. The IP addresses that are scannable are easier to attack than those of non-scannable IP addresses (Step 3.1.5, Section 8.2).

E-Pattern 4 (Port Open) (4): Developers include the database server's port number in the asset identifier, such as port 3306 for MySQL database. If the IP address is publicly accessible and the database port is open, the attackers can more easily access the database. In contrast, access is relatively difficult if the database port is closed or restricted (Step 3.1.6, Section 8.2). We observed four ports open in the 50 secret-asset pairs that we inspected using Censys [253], an online service that provides the port information of a host.

8.1.3 Developer Survey

To answer RQ2, we conducted a survey to understand whether developers prioritize the removal of secrets consistent with our hypothesis (based on the descending security risk scores).

Participant Selection: To find survey participants, we selected the developers who committed the secret-asset pairs in RiskBench. We selected these developers since they have experience with software secrets. We observed that the same developer committed multiple secret-asset pairs in a repository. Thus, we identified unique 1,478 committers from the 1,791 secret-asset pairs of RiskBench. In addition, we filtered out committers having a noreply (xxx@noreply.github.com) or GitHub Actions bot email address [142] and selected 1,282 committers. Finally, we randomly selected 500 committers to avoid selection

Table 8.4: The Developer Survey Questions. The full questionnaire of the online survey can be found online [2].

Secret Alert Information			Question Description
Secret A	Secret B	Secret C	
Secret: "Fm)4dj" Severity: CRITICAL Other Info: Repo and File Location	Secret: "123456" Severity: CRITICAL Other Info: Repo and File Location	Secret: "123456" Severity: CRITICAL Other Info: Repo and File Location	Q1: Based on the alert information, in what order would you prioritize the removal of the secrets? Did the "Severity" info help?
Asset: "127.0.0.1" Other Info: Asset Location	Asset: "111.230.140.27" Other Info: Asset Location	Asset: "120.77.222.217" Other Info: Asset Location	Q2: Given the additional asset information, would you change your prioritization order to remove the secrets? Why?
Security Risk Score: 100 Value of Asset: HIGH (Blockchain data) Ease of Attack: VERY_DIFFICULT (Localhost)	Security Risk Score: 40 Value of Asset: LOW (Video URL and TimeStamp data) Ease of Attack: DIFFICULT (Public IP but not reachable)	Security Risk Score: 320 Value of Asset: MODERATE (Phone and Email Address data) Ease of Attack: DIFFICULT (Public IP reachable but database port 3306 is not open)	Q3: Given the additional security risk score (value of asset and ease of attack) information, would you change your prioritization order to remove the secrets? Why?
The value of the asset is calculated based on the database table and column names from the source code. The ease of attack is calculated based on the accessibility of the database server. Next, we calculated the security risk score by multiplying the value of an asset and the ease of attack.			Q4: (Optional) Do you agree with our means of calculating the security risk score? What suggestions do you have for improving the security risk score?

bias [143] to participate in the online survey.

Survey Design: Table 8.4 presents the four questions of the developer survey. We provided three alerts of database secrets of GitHub repositories detected by TruffleHog [14]. For each alert, we provided the secret, the “Severity” level, and the repository and file location provided by TruffleHog. Based on the alert information, we asked developers in which order they would prioritize secret removal and why. We hypothesize that developers will choose Secret A first without considering the asset information since Secret A looks like an actual password. Next, we provided the asset identifier (IP address) protected by the database secret with file location in each alert. Then, we asked if developers would change the priority of secrets removal based on the asset identifier and why. We hypothesize that developers will change their priority and choose B and C since these secrets point to public IP addresses. Next, we provided the security risk score, value of asset, and ease of attack information, such as the sensitive data categories and passive network information from RiskHarvester (Section 8.2). Then, we asked if developers would change the priority of secret removal based on the security risk score and why. We hypothesize that developers will change their order to Secret C, A, and B. We also asked an optional question on developers’ suggestions for improving the security risk score calculation. In the survey, all the questions are kept open-ended to avoid bias from predefined options and explore diverse perspectives.

Conducting Survey: For conducting the survey, we leveraged the Qualtrics [254], a popular online survey host. However, before conducting the main survey, we conducted a pilot survey with five security researchers from the anonymized lab. In the pilot survey, we provided an additional question for suggestions on survey improvement, including any unclear, irrelevant, or overly detailed aspects. We conducted the main survey in November and December 2024. We offered a \$20 Amazon gift card to five randomly selected partici-

pants if they wished to participate in the lottery. We discussed the IRB approval and other ethical considerations in Section 8.7.

8.2 RiskHarvester Construction

We calculated the security risk score as the product of value of asset and ease of attack for each secret-asset pair (Equation 8.1). We utilized the identified value of asset and ease of attack patterns (Section 8.1.2) and constructed RiskHarvester to calculate the security risk score. We now discuss the four-step process of constructing RiskHarvester.

8.2.1 Step 1: Identifying Secret-Asset Pairs

Before we identify the value of asset and ease of attack of secret-asset pairs, we used the implementation source code of AssetHarvester [248], an open-source static analysis tool, to detect secret-asset pairs in a repository (Steps 1.1-1.3). AssetHarvester demonstrates precision of (97%), recall (90%), and F1-score (94%) in detecting secret-asset pairs in RiskBench. We extended AssetHarvester as RiskHarvester to calculate the security risk score for each secret-asset pair (Steps 2-4).

Step 1.1 Pattern Matching: In the source code, a secret and the corresponding asset can be present in database connection strings that follow a specific format for different database types. For example, MySQL, PostgreSQL, and MongoDB follow the same connection string format ([scheme://][user:password@]host:port/db). Thus, regular expressions (regex) are formulated to identify the connection strings by manually analyzing the database documentation. In addition, the capturing group [219] feature of the regex is utilized to isolate the secret and the corresponding asset (host, port, and db name) from the connection string. Table B.1 in the Appendix presents the regexes, which are grouped into three groups based on the connection string format similarity.

Step 1.2 Data Flow Analysis: A secret and the corresponding asset can be defined separately in variables and passed to database driver functions instead of defined in a connection string. For example, Figure 8.2a shows that the database secret and the corresponding asset are passed to the driver functions (lines 1-3). The secret-asset pair is passed to the driver function as positional or keyword arguments [226]. A positional argument is passed based on the position in the argument list without specifying the parameter name, whereas a keyword argument is passed by explicitly specifying the parameter name, such as “password” or “host”, without fixed order in the function. Table 8.5 presents the list of

Table 8.5: List of Python database drivers and ORM frameworks with their supported arguments for secret-asset pairs

Category		Driver Name	Pos. Arg.	Key. Arg.
SQL Driver	MySQL	aiomysql [227]		✓
		PyMySQL [229]	✓	✓
	PostgreSQL	aiopg [230]	✓	✓
		asyncpg [231]	✓	✓
		psycopg2 [232]	✓	✓
	SQL Server	pymssql [234]		✓
	ODBC	pyodbc [235]	✓	
JDBC	JayDeBeApi [218]	✓		
NoSQL Driver	MongoDB	pymongo [233]		✓
		Flask-PyMongo [255]		✓
ORM Framework		peewee [223]	✓	✓
		SQLAlchemy [224]	✓	✓
		Django [256]		✓

Python database drivers and ORM frameworks with the supported argument type. Since the argument positions and names for a secret-asset pair are known in the driver functions, data flow analysis [221] is leveraged to identify the secret-asset pair by analyzing the data flow graph (DFG). DFG is a directed graph where the secret-asset pair is the source that flows into the driver function arguments, which act as sinks. CodeQL [236], an open-source source code analysis framework that provides the data flow graph computed from the repository source code, is used for data flow analysis to identify the secret-asset pair.

Step 1.3 Fast-Approximation Heuristics: The data flow analysis may not always be captured when source code has dynamic behavior, such as extensive use of reflection. In such cases, the secret-asset pair can be identified from the neighboring lines in the source code. Secrets are first extracted using two open-source secret detection tools, TruffleHog [14] and Gitleaks [138]. Next, an IP address or DNS name is searched in the three neighboring lines of the secret. Since multiple assets can be present in the neighboring lines, the prefixes of both the secret and asset variables are matched to find the correct asset. For example, “mysql” is the prefix of MySQL database secret (“mysql-password”) and server (“mysql-host”) variables.

8.2.2 Step 2: Identifying Value of Asset

In this section, we described the process of extracting database keywords from the source code (Step 2.1) and mapped these identified keywords to sensitive data categories (Step 2.2).

Step 2.1 Extracting Database Keywords: We extracted the database keywords (database, table, and column names) from database drivers and ORM frameworks. Our study included eight SQL and two NoSQL database drivers and three ORM frameworks for extracting database keywords (Table 8.5).

SQL Database Driver Calls: We observed that the database name and corresponding table and column names are passed to SQL database driver functions (V-Pattern 1). The database name is passed as a positional or keyword argument based on SQL driver types along with the secret, host, and port in the same driver function, such as in the `pymysql.connect` function (lines 1-4, Figure 8.2a). Thus, we included the database name argument in the data flow analysis of Step 1.2 and identified the database name along with the host and port.

Additionally, we observed that raw SQL queries are passed in query functions other than the “connect” function where the secret-asset pair is passed. Figure 8.2a shows a SELECT SQL query is directly passed in the “execute” function (lines 7-8) for retrieving the patient information. However, SQL queries can also be defined in separate files such as `.sql` and `.ddl` files, which are mostly used for database migration and executed from the source code. However, CodeQL does not support data flow between source codes of multiple file types. Thus, the flow of raw SQL present in the `.sql` file can not be captured into the Python database driver’s “execute” function. As a result, we first identified the SQL file name from the file open functions [257] using data flow analysis. Finally, to parse the table and column names from the raw SQL, we used the `sql_metadata` [258] package of Python that provides a parser for retrieving table and column names from raw SQLs.

NoSQL Database Driver Calls: We observed that the database name, corresponding table, and column names are passed in the NoSQL database drivers (V-Pattern 2) for non-relational databases. However, unlike SQL database drivers, database and table names are passed as dictionary keys to the connection client and corresponding database instance. Thus, we first located the data flow node in the DFG for the connection client instance (sink) initialized with the secret-asset pair and traced the source that flows into the specific sink to extract the database name. Using the identified database name, we located the data flow node of the corresponding database instance (sink) and repeated the process to

identify the table name that flows in the database instance sink. Since the column names are passed as key-value pairs in a dictionary in the driver query function, we first traced the data flow node of the dictionary that flows into the table instance sink. Next, to find the column names, we extracted the keys from the key-value pairs of the dictionary. Finally, the database, table, and column names of non-relational databases are extracted.

ORM Framework Calls: From V-Pattern 3, we observed that developers employ ORM frameworks to access relational databases. For ORM framework calls, we found that the database name is passed to the ORM configuration functions as a part of the connection string. Thus, we located the configuration function sink in the DFG and extracted the database name by tracing the flow of the connection string into the sink (similar to Step 1.2). However, ORM abstracts database access through objects instead of raw SQL queries or key-value pairs. The database tables are mapped to model classes, and the columns are mapped to the attributes of the classes. Thus, we first located the ORM class that uses the ORM database instance in the DFG. Then, we identified the class source code from the abstract syntax tree and extracted the attribute names of the class. To extract the attribute names, we used Python's `py_models_parser` [259] package, which can parse the model classes and table definitions. Finally, we separated the table and column names from the corresponding attribute names of the ORM class as database keywords.

Step 2.2 Mapping Database Keywords to Sensitive Data Categories: Since each database keyword can have different sensitivity, we mapped each identified keyword to a data category of 113 categories provided by Google Cloud DLP (Section 8.1.1). We observed that the Google Cloud DLP provides API to assign a data category to specific instances of the data. For example, instead of the database keyword "passport", the API takes a country-specific passport number as input and outputs the mapped data category. However, in our study, we only have the identified database keywords from the source code for secret-asset pairs (Step 2.1). In addition, we observed that the database keywords will not always match the data category name exactly. For example, the database keyword is "NID_NUMBER", which should be assigned to the "NATIONAL_ID_NUMBER" category. We now discuss the lexical and semantic string similarity algorithms we used to map each database keyword to a data category.

Prefix Match: We observed that database keywords match from the start of a data category. For example, the database keyword is "FINANCIAL_ACC", and the corresponding data category is "FINANCIAL_ACCOUNT_NUMBER". To measure the similarity between these strings, we utilized the Jaro-Winkler algorithm [177] that emphasizes prefix similarity by assigning higher scores to strings that share common prefixes. The algorithm generates

a similarity score between 0 and 1, where 0 indicates entirely dissimilar strings, and 1 indicates identical strings. We set a cut-off score of 0.7. To employ the Jaro-Winkler algorithm, we leveraged the `jaro_winkler_similarity` function of Python's `jellyfish` [179] package.

Substring Match: We observed that database keywords may not have a longer common prefix with a data category. For example, the database keyword “NID_NUMBER” should match the “NATIONAL_ID_NUMBER” category, though the middle characters are missing in the keyword. To address the scenario, we used the Gestalt pattern matching algorithm [180], which calculates a similarity by identifying the common substring and recursively comparing characters in the unmatched regions on both sides of the longest common string. Thus, we could match the database keyword with the correct category even if the database keyword is incomplete or has missing segments. Like the Jaro-Winkler algorithm, Gestalt provides a similarity score between 0 and 1, and we set a cut-off score of 0.7. We implemented the algorithm using the `SequenceMatcher` function of Python's `difflib` [181] package.

Semantic Match: We observed that database keywords differ lexically from the correct data category but have the same meaning. For example, the database keyword “CELL_NO” should map to the “PHONE_NO” category due to the same meaning. Thus, we need to calculate semantic similarity between the strings instead of lexical similarity (Prefix and Substring match). For semantic similarity between words, we leveraged `fastText` [260], a natural language processing (NLP) model for generating word embeddings by capturing semantic information. In addition, we observed that the subwords in the database keyword can be the same as the subwords of the data category but present in different orders. For example, despite the subword's order, the database keyword “DATE_OF_BIRTH” should match the “BIRTH_DATE” category. We chose `fastText` over other NLP models, such as `Word2Vec` [261] and `GloVe` [262], since `fastText` supports out-of-vocabulary word embeddings and is trained with character n-grams. As a result, `fastText` can be used to capture the similarity of the words with different subword orders. In our study, we used the pre-trained `fastText` model `cc.en.300.bin`, trained on Common Crawl and Wikipedia with 5-character n-grams, a window size of 5, and 10 negatives. We used the `fasttext` [263] package of Python to access the model and calculate semantic similarity. We set a cut-off similarity of 0.65.

Non-English & Transliterated Word Match: We observed that non-English or transliterated words are present in the source code as database keywords. A transliterated word is a word from one language written in another language's alphabet by representing the

pronunciation. For example, the Chinese word “性别” or the corresponding transliterated word “Xingbie” is present in the SQL queries. As a result, we first translated the non-English and transliterated words to English words and then computed the lexical and semantic similarity. In our study, we leveraged the Google Cloud’s Translation API [264] using the `google-cloud-translate` package [265].

The cut-off similarity scores are chosen by randomly sampling database keywords and observing the score. We assigned a sensitivity level of “UNSPECIFIED” when no data category is matched, such as the database keyword “test”.

8.2.3 Step 3: Identifying Ease of Attack

In this section, we described the process of identifying ease of attack information (Step 3.1) and assigning ease of attack categories based on the identified information (Step 3.2).

Step 3.1 Finding Ease of Attack Information: We identify different ease of attack information from the host and port part of the asset identifier after each step (Steps 3.1.1-3.1.6).

Step 3.1.1 Valid DNS Name: From E-Pattern 1, we observed that developers put a DNS name in the host part of the asset identifier as a database server address. However, the DNS name can be invalid according to the DNS name format set by the Internet Engineering Task Force (IETF) [266] through Request for Comments (RFCs) [267]. For example, each segment between dots in the DNS name can have up to 63 characters and should not start or end with a hyphen. In our study, we utilized the `domain` function of Python’s `validators` [268] package to validate the DNS name format.

Additionally, developers can put a placeholder DNS name in the source code, such as `"www.example.com"`. However, detecting placeholder DNS names is challenging because the placeholder DNS names can conform to the DNS name format, and no universal registry exists to identify them. We can apply a rule-based approach by analyzing the common placeholder keywords to detect placeholder DNS names. However, the rule-based approach has limitations, such as a lack of adaptability due to a fixed set of keywords to arbitrary DNS names. Additionally, the rule-based approach cannot interpret the meaning behind names. However, we can apply Large Language Models (LLMs) to detect the placeholder DNS names since LLMs excel in semantic understanding and recognizing contextual clues that differentiate actual DNS names from placeholders [269, 270]. While other Generative pre-trained transformer (GPT) style LLMs exist, we leveraged ChatGPT due to ChatGPT’s performance in Zero-shot Learning (ZSL) through Chain-of-Thought (CoT) prompting [270, 271, 272]. The ZSL enables models to address unseen tasks without

prior training examples, while CoT prompting guides the models through a structured, step-by-step reasoning process to arrive at more accurate answers. In our study, we employed `gpt-4o-2024-08-06` [273] model of ChatGPT with temperature 0.2 to make the model more deterministic and confident. As shown in Table B.2 of the Appendix, we provided one example of a placeholder and one example of actual DNS names with the context source code in the CoT system prompt. In the user prompt, we provided the DNS name to be classified as a placeholder or not, along with two neighboring lines of source code for context. Finally, we identified the valid DNS names from the prompt answer, which we passed on to the next step.

Step 3.1.2 Resolvable DNS Name: We observed that all valid DNS names may not resolve to IP addresses due to non-existent domains or misconfigured DNS records (E-Pattern 1). Thus, we checked whether the DNS names from Step 3.1.1 are resolvable by querying the DNS servers. We leveraged `nslookup` [274], a BIND name server software member that obtains the mapping between a domain name and IP address. However, we observed that `nslookup` can return a Canonical Name (CNAME) record when queried for a DNS name. The DNS system allows aliases using CNAME records to simplify domain management, enabling a single canonical domain to represent multiple aliases. Thus, to identify the IP address from the A (IPv4) or AAAA (IPv6) record for the DNS name, we recursively queried using the canonical domain name. In our study, we used Python's `nslookup` [275] package.

Step 3.1.3 Valid IP Address: From E-Pattern 2, we observed that invalid or placeholder IP addresses are present in the source code. To check the validity of the IP address directly present in the host part or the resolved IP address from the DNS name (Step 3.1.2), we used the `ip_address` function of `validators` [268] package of Python.

Step 3.1.4 Routable IP Address: Since assets with public IP addresses are easier to access by attackers than non-routable addresses such as localhost or private IP addresses (E-Pattern 2), we checked whether the IP addresses from Step 3.1.3 are routable. We used Python's `ipaddress` [276] package that provides functions for detecting the routable addresses.

Step 3.1.5 Scannable IP Address: We observed that not all the public IP addresses identified from the source code are scannable (E-Pattern 3). To detect if the IP addresses from Step 3.1.4 are scannable, we did not use `ping` command since `ping` uses Internet Control Message Protocol (ICMP) packets that are typically blocked by servers through firewalls. In addition, `ping` does not provide information on the active services running on the server. Thus, we used Censys Search API [277], which uses TCP and UDP packets in the network scan and maintains a database of publicly available information on the active services of a

server. Finally, we filtered the scannable IP addresses and detected corresponding active service ports.

Step 3.1.6 Port Open: Developers put the database port number in the asset identifier (E-Pattern 4). Thus, we checked whether the port is open for the scannable IP address using the open ports for the scannable IP address found in Step 3.1.5.

Step 3.2 Assigning Ease of Attack Category: From Step 3.1, we observed that at each step, we find new information regarding the ease of attack of the identified asset. Thus, we need to assign an ease of attack category based on the asset information similar to the value of asset to calculate the security risk score (Step 4). To systematically assign an ease of attack category to an asset, we started with a value of 0 for ease of attack. Next, when we retrieve new information after each step, such as if the DNS name is valid (Step 3.1.1), we increment the value for ease of attack by 1. Similarly, if the valid DNS name is resolvable (Step 3.1.2), we increment the value again by 1. In our study, for ease of attack, we assigned four categories (VERY_DIFFICULT, DIFFICULT, MODERATE, and EASY). The first and second authors of the paper independently inspected the calculated value for ease of attack and assigned a category based on the asset information. The paper's third author, who has over 15 years of experience in network security, resolved the disagreements related to the assigned categories between the first and second authors. Figure 8.3 depicts the final categories assigned for ease of attack on an asset at different steps. For example, the ease of attack for an asset is MODERATE if the IP address is scannable, whereas EASY if the database port is open. Finally, we integrated the ease of attack mappings based on host information into RiskHarvester, eliminating manual effort for tool users.

8.2.4 Step 4: Calculating Security Risk Score

We identified the value of asset (Step 2) and ease of attack (Step 3) as ordinal categories. To calculate the security risk score (Equation 8.1), we need to perform ordinal scaling [278], which assigns numerical values to the categories while preserving their inherent order. Thus, to assign numerical values, we leveraged Protection Poker [10], a threat modeling game for security risk quantification. We conducted a Protection Poker session with the first, second, and third authors of the paper. We leveraged the nine values from 1, 2, 3, 5, 8, 13, 20, 40, and 100 used by Protection Poker for estimating the “value of asset” and “ease of attack”. We assigned numerical values to the categories of value of asset and ease of attack after two Protection Poker rounds. For value of asset, the assigned values are HIGH (100), MODERATE (40), LOW (5), and UNSPECIFIED (1). For ease of attack, the assigned

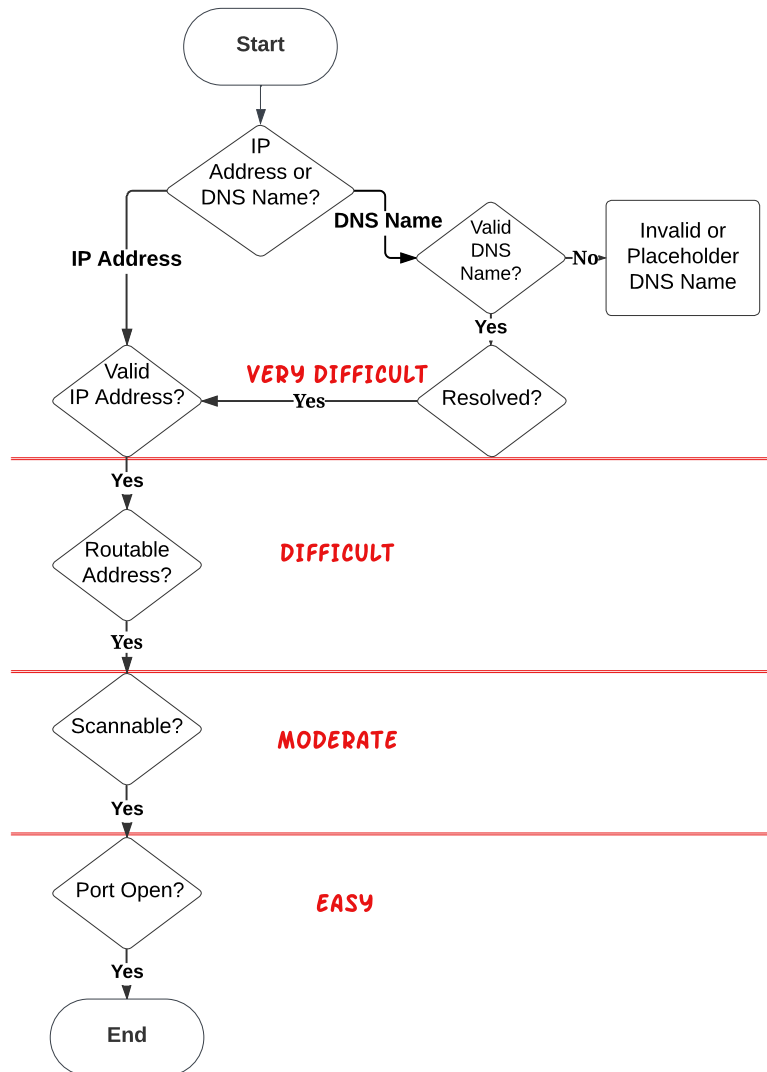


Figure 8.3: A flow diagram for assigning ease of attack category for an asset identified in the source code.

values are VERY_DIFFICULT (1), DIFFICULT (8), MODERATE (40), and EASY (100). This mapping is integrated into RiskHarvester to automatically calculate the security risk score. Finally, we multiplied the value of asset and ease of attack to calculate the security risk score (Equation 8.1). For example, if the value of asset is HIGH (100) and the ease of attack is DIFFICULT (8), the security risk score is 800.

Table 8.6: Precision, Recall, and F1-Score of RiskHarvester in identifying the database name, table, and column names

Database Type	Database Name			Table Name			Column Name		
	Precision (TP, FP)	Recall (TP, FN)	F1	Precision (TP, FP)	Recall (TP, FN)	F1	Precision (TP, FP)	Recall (TP, FN)	F1
MySQL	0.98 (707, 13)	0.96 (707, 29)	0.97	0.97 (320, 10)	0.90 (320, 36)	0.93	0.95 (813, 46)	0.85 (813, 133)	0.89
PostgreSQL	0.96 (598, 20)	0.95 (598, 33)	0.95	0.96 (327, 13)	0.94 (327, 20)	0.95	0.90 (310, 33)	0.87 (310, 47)	0.88
MongoDB	0.90 (95, 10)	0.82 (95, 21)	0.86	0.92 (25, 2)	0.76 (25, 8)	0.83	0.77 (87, 25)	0.88 (87, 11)	0.82
SQL Server	1.00 (5, 0)	0.62 (5, 3)	0.76	1.00 (5, 0)	0.33 (5, 10)	0.51	0.85 (12, 2)	0.40 (12, 18)	0.54
Overall	0.97 (1405, 43)	0.94 (1405, 86)	0.95	0.96 (677, 25)	0.90 (677, 74)	0.93	0.92 (1222, 106)	0.85 (1222, 209)	0.88

8.3 Results

In this section, we answer RQ1 by evaluating the performance of RiskHarvester against RiskBench and RQ2 by assessing whether developers prioritize secret removal from software artifacts ranked by descending security risk scores.

8.3.1 Performance of RiskHarvester

Performance of Finding Database Keywords: Table 8.6 presents RiskHarvester’s precision, recall, and F1-score in identifying the database, table, and column names for each database type. The column “Precision (TP, FP)” denotes the precision score with the number of true positive and false positive database keywords outputted by RiskHarvester. The column “Recall (TP, FN)” denotes the recall score with the number of outputted true positive and false negative database keywords. The column “F1” denotes the F1-score (the harmonic mean of precision and recall).

We observed that RiskHarvester demonstrated overall precision of 97%, 96%, and 92% in identifying the database, table, and column names, respectively, indicating high precise detection of database keywords. The count of false positives indicates that the tool incorrectly outputted 43 database names, 24 table names, and 106 column names out of 3,304 database keywords. In addition, RiskHarvester demonstrated an overall recall of 94% and 90%, indicating a strong ability to identify database and table names, respectively, supported by F1-scores of 95% and 93%. However, the recall of identifying column names is 85%, which is relatively lower than that of database and table names. The count of false negatives indicates that the tool failed to detect 86 database names, 74 table names, and 209 column names. We also observed that among the four database types, the recall of database, table, and column names of SQL Server is low (62%, 33%, and 40%, respectively), though

the precision is 100%. In addition, the recall for table names in MongoDB is relatively low (76%) compared to MySQL and PostgreSQL. We now discuss our observations on the false positives and false negatives.

Analysis of False Positives: Since the SQL drivers use raw SQL queries (V-Pattern 1), we observed that the false positives on table and column names are mostly caused by the dynamically constructed queries (61% of the false positives). We identified all the string parts flowing in the driver function sink for a SQL query using data flow analysis and reconstructed the query using the source code line and column information (Step 2.1). However, we could not reconstruct the complete query due to the presence of conditional statements and dynamically fetched values from the environment variables or config files. As a result, we identified incorrect table and column names while parsing the SQL query. Similar to dynamic raw SQL query, we observed that 24% of the total incorrect column names are from dynamically constructed dictionary objects for column names passed in the NoSQL drivers (V-Pattern 2). Additionally, the false positives of database names are mostly triggered by the neighboring lines rule (Step 1.3), comprising 71.5% of the 43 false positives. We observed that the prefix match of the neighboring key names met the threshold, though the key name is not the correct asset of the corresponding secret containing the database name.

Analysis of False Negatives: We observed that the repositories of RiskBench also contain non-Python source codes such as C# and Java. While we detected secret-asset pairs in non-Python code using regex (Step 1.1), we could not identify table and column names since data flow analysis was only applied to Python code (Step 2.1). For example, the SQL server shows a relatively lower recall (33% and 40% for table and column names) since SQL Server table and column names are typically passed to .NET driver functions. In addition, we reconstructed the raw SQL query from the query parts flowing into the driver sinks. However, similar to false positives, we missed table and column names due to improperly reconstructing the original query for having dynamic behavior. Additionally, we observed that 54 (4.2%) instances of secret-asset pairs in RiskBench do not fall within three neighboring lines. Thus, we failed to detect the database name when the asset identifier containing the database name was not present in the three neighboring lines of the secret (Step 1.3).

Performance of Sensitive Data Category Mapping: We applied lexical and semantic matching to map the identified database keywords to the corresponding sensitive data categories (Step 2.2). We observed a precision of 85% among the 3,673 database keywords of RiskBench. We manually inspected a random sample of 50 false positive mappings. We

noticed that 27 false positives are due to the Jaro-Winkler similarity, which employs prefix bias. For example, “credit_limit” and “tax_rate” keywords are wrongly mapped to “CREDIT_CARD_NUMBER” and “TAX_ID” data categories, respectively. In addition, we observed that 16 false positives were due to semantic matching. For example, “transaction_code” is mapped to “CREDIT_CARD_NUMBER” since both terms appear in financial contexts, leading to a semantic link.

Performance of Detecting Placeholder Host: We observed that the precision of identifying the placeholder host is 96%. All the 11 false positives are DNS names outputted by the ChatGPT model (Step 3.1.1). For example, “gg-is-awesome-246.mongodb.net” is termed a placeholder due to the “is-awesome” substring in the DNS name. In addition, our tool shows a recall of 94% for detecting the placeholder host out of 317 placeholder hosts in RiskBench. Similar to false positives, all the missing placeholder hosts are DNS names.

8.3.2 Developer Survey

We received 52 responses (10.4%) out of 500 developers. We now discuss our observations from the responses.

Q1: Secret and Severity Information: We observed that 41 developers (78% of respondents) wanted to remove Secret A first, terming Secret B and C as placeholder/dummy, supporting our hypothesis. For example, <P23> stated “*A first, then B or C. A appears to have an actual password, whereas B and C are just placeholders.*” In addition, we observed that the severity information did not help the developers. <P13> stated that “*Severity info did not help, needed to look at the secret to determine that B and C are likely fake secret values.*” Additionally, 5 developers were unsure about the order of secret removal due to missing secret contexts, such as the asset information. For example, <P45> stated that “*I have no idea in what order to prioritize. To effectively prioritize, I need to know the context for what these secrets grant access.*” However, 6 developers considered the asset information into account by inspecting the source code that we did not provide.

Q2: Additional Asset Information: We observed that 38 of 41 developers who selected Secret A in Q1 changed their priority after considering the asset identifier information. These developers changed their priority to Secrets B and C even though the secrets looked like placeholders, supporting our hypothesis. <P23> stated that “*Since Secret A coming from localhost, we might not access it directly. But the other two seem on the internet and should be our top priority to address.*” However, 3 developers did not change the priority without providing any reason. Additionally, all the 5 developers who were unsure about

which secrets to prioritize in Q1 have used the asset information to make decisions. <P45> stated that *“Based on the added information of ip address of the system secret is used to access, I would deprioritize Secret A compared to the other two, as localhost is more likely to be hardened against outside access.”* Since 6 developers in Q1 already considered the asset information, their priority stayed the same. <P5> stated that *“No, and I detailed in my previous explanation since I already took the IPs into account.”*

Q3: Additional Security Risk Score Information: We observed that 86% (45 out of 52) of the respondents changed their priority to Secret C, A, and B based on the descending security risk score, thus supporting our hypothesis. <P9> stated that *“I would make changes to my prioritization (Secret C, A, B). Secret C has high risk score, personal data exposure, and reachable IP make it most critical to address. While Secret A has lower risk (100), high value of blockchain data means cannot be ignored, even though protected by localhost.”* Additionally, developers pointed out that they had not considered the value of asset information before. <P10> stated that *“I didn’t check the Value of Asset. More security is needed for valuable assets.”* However, 4 developers did not change their priority without specifying any reason, and 3 developers wanted more context on the value of asset and ease of attack.

Q4: Feedback on Security Risk Score: Developers provided feedback on the security score calculation, such as <P3> stating that *“This simple calculation makes sense and is easy to understand.”* Another developer <P11> stated that *“It aligns with some of the ways we do it in my sector (cloud security) at least.”* However, developers also suggested improvements to the security score calculation based on active network analysis. For example, <P40> suggested that *“It would also be important to include deployment information as passive network analysis might not give the real picture.”* In addition, developers suggested accounting for whether the data is encrypted in the value of asset. <P30> stated that *“If it is encrypted, it should be scored less than non-encrypted data.”* Developers also suggested including the attack vectors, such as privileges required and lateral movement, in the ease of attack calculation. <P7> stated that *“attach attack vectors if possible such as privileges required. In general, attack vector score would change the prioritization.”*

8.4 Discussion

In this section, we discuss the implications of RiskHarvester based on our study findings.

Only the asset identifier is not enough to aid developers in prioritizing the software secret removal. Basak et al. [248] constructed AssetHarvester for detecting the corresponding

asset identifier by the secret. From our developer survey, we observed that 73% of developers changed their priority based on asset identifiers, and 10% of developers were unsure of their decision and wanted more asset context. However, when we provided the security risk scores with the value of asset and ease of attack information, 86% of the developers changed their priority in the descending order of security risk score. In addition, developers pointed out that the value of asset and ease of attack information helped them to make informed decisions to tackle the secret removal efforts.

The risk-based analysis for secrets should be integrated into the secret detection tools. We developed RiskHarvester to automatically calculate the security risk for the checked-in secrets. Our approach eliminates the need for developers to manually analyze each secret detection tool alert and calculate the security risk. To integrate our approach, the input for RiskHarvester will be the repository source code, and RiskHarvester will output the secrets ranked by descending security risk score. Thus, developers can focus their mitigation efforts on the most critical security risks.

RiskHarvester can be extended to calculate the security risk score of secrets in other programming languages and secret types. In our study, we calculated the security risk score of four database providers in Python. We now discuss the effort needed and challenges to extend RiskHarvester for other programming languages and secret types.

Programming Language: We identified the secret-asset pairs using pattern matching, data flow analysis, and fast-approximation heuristics. We parsed the database names from the identified asset identifiers. Since pattern matching and fast-approximation heuristics are programming-language agnostic, we can apply the techniques in other programming languages without additional effort. Additionally, we leveraged data flow analysis to detect the secret-asset pair instance that flows into query functions and then extracted database keywords from raw SQL and ORM classes (Step 2.1). Though data flow analysis is programming language dependent, we can compute the abstract syntax tree, control flow, and data flow graph for each programming language in a repository separately using CodeQL. Next, we can identify the secret-asset pair sources and sinks from the computed graphs with minimal effort. Additionally, SQL parsing is programming-language agnostic, enabling the parsing of database keywords in other languages.

Non-database Secret Types: From SecretBench [152], we inspected five random samples of secrets of seven secret types, such as API keys, private keys, and authentication tokens. We now discuss extending RiskHarvester to identify non-database secret-asset pairs and corresponding asset keywords.

1. Secret-Asset Pairs: The 2024 GitGuardian report [4] reveals that cloud secrets such as

API keys and tokens are the second most exposed in GitHub. Since cloud providers have specific formats, we can identify the secret-asset pair using the regex (Step 1.1). Additionally, we can identify the functions of frameworks such as .NET and Spring, where the secret-asset pairs are passed similarly to database drivers and employ data flow analysis (Step 1.2). The list of functions will not be huge since most non-database secret-asset pairs are passed in HTTP clients such as get and post functions.

2. Non-Database Keywords: We inferred the asset's value from raw SQL and ORM parsing for database secrets. However, for non-database secrets, we can infer the data category from the request body and response of HTTP requests that use the secret-asset pair. Thus, using data flow analysis (Step 2), we can identify the request body (sinks) and parse the request body parameters (sources) to infer the data categories. However, the responses of HTTP requests (typically in JSON or XML format) are serialized into classes. Thus, we can employ data flow analysis to detect the response class and identify the data categories from the class attributes. In our study, we used the `py-models-parser` [259] package to parse ORM models (Step 2.1), which also supports parsing any data class.

The security risk score can be improved by employing active network analysis in RiskHarvester. Our study focused on passive network information for estimating the ease of attack. However, developers wanted the deployment-related security information, such as the level of security controls present on the asset and network vulnerabilities (Section 8.3.2). We will extend RiskHarvester by leveraging the information provided by the active network scanning and vulnerability assessment tools such as Nessus [279] and Nmap [280]. Additionally, we will improve the value of asset by retrieving the metadata information of a database, such as whether the data is encrypted or hashed, using database auditing tools such as Datadog [281]. We will deploy RiskHarvester in the organization where these scanning and auditing tools are deployed.

8.5 Threats to Validity

In this section, we discuss the limitations of our study.

Manual Analysis: We identified database keywords and the data types for each secret-asset pair of AssetBench by manually inspecting the source code (Section 8.1.1). However, manual analysis is prone to bias due to differing interpretations and oversights. Two authors cross-checked the identified database keywords and data types to mitigate the bias.

Benchmark Dataset: Our benchmark dataset selection is susceptible to bias. Basak et

al. [248] identified the secret-asset pairs of AssetBench using two open-source tools, TruffleHog and Gitleaks, from GitHub repositories. However, these two tools may miss secrets from the repositories. Additionally, AssetBench does not contain repositories from other VCSs, such as BitBucket [148]. We could not mitigate the potential bias since AssetBench is the only publicly available dataset.

Developer Survey: Our survey findings are susceptible to external validity, as the participant pool might not accurately represent the broader developer population. To mitigate the limitation, we randomly sampled developers with prior experience in software secrets. The survey results may be influenced by how we presented the problem to the developers. To ensure clarity in the survey questions, we conducted a pilot survey with security researchers and refined the questions based on their feedback. Additionally, we provided open-ended questions to mitigate the bias from predefined options [282].

Data Flow Analysis: We used CodeQL for data flow analysis on the latest repository snapshot. However, CodeQL only models data flow for the current snapshot, and secret-asset pairs may still exist in previous commits. While we could analyze each snapshot from Git history to identify these pairs, the approach would be impractical and time-consuming.

8.6 Conclusion

We present RiskHarvester, a risk-based tool to compute a security risk score based on the value of the asset and ease of attack on a database. We calculated the value of asset by identifying the sensitive data categories present in a database from the database keywords. We utilized data flow analysis, SQL, and Object Relational Mapper (ORM) parsing to identify the database keywords. To calculate the ease of attack, we utilized passive network analysis to retrieve the database host information. To evaluate RiskHarvester, we curated RiskBench, a benchmark of 1,791 database secret-asset pairs with sensitive data categories and host information manually retrieved from 188 GitHub repositories. RiskHarvester demonstrates precision of (95%) and recall (90%) in detecting database keywords for the value of asset and precision of (96%) and recall (94%) in detecting valid hosts for ease of attack. Finally, we conducted an online survey to understand whether developers prioritize secret removal based on security risk score. We found that 86% of the developers prioritized the secrets for removal with descending security risk scores.

8.7 Ethical Considerations

In our study, we followed the USENIX Ethics Guidelines [283]. Since RiskBench contains sensitive information, the dataset will be selectively shared with researchers and tool developers under a data protection agreement to ensure ethical use. Additionally, we obtained IRB approval from our institution (Uni IRB blinded) before conducting the survey. No personally identifiable information was collected from participants apart from the email addresses of the participants who wished to participate in the lottery. Additionally, a consent form was included in the email stating that participants should not attempt to use the secret-asset pairs to verify their validity. We have also stated in the consent form that participation was voluntary and participants could withdraw at any time.

8.8 Open Science Policy Compliance

Our curated dataset, RiskBench, is stored in Google BigQuery (Dataset ID: *dev-range-411201.riskbench*) as a relational structured data. Researchers and tool developers can utilize and extend the dataset for future research using SQL queries in Google BigQuery. Additionally, we have made the implementation of RiskHarvester publicly available [2].

CHAPTER

9

CONCLUSION AND FUTURE WORK

9.1 Conclusion

In this thesis, we conducted six original studies. In the first study, we extracted 779 questions related to checked-in secrets on Stack Exchange and applied qualitative analysis to determine the challenges and the solutions posed by others for each of the challenges. We identify 27 challenges and 13 solutions. We observe an increasing trend in questions lacking accepted answers.

In the second study, we conducted a grey literature review of 54 Internet artifacts and identified 24 secret management practices grouped into six categories. Our findings indicate that using secret detection tools and employing short-lived secrets are the most recommended practices to avoid accidentally committing secrets and limit secret exposure.

In the third study, we curated SecretBench, a publicly available labeled dataset of source codes containing 97,479 secrets (of which 15,084 are true secrets) of 7 secret types extracted from 818 public GitHub repositories.

In the fourth study, we evaluated five open-source and four proprietary secret detection tools against SecretBench. We observed that tools generate a lot of false positives (25%-

99%) and miss secrets (14%-99%). Our manual analysis of reported secrets reveals that false positives are due to employing generic regular expressions and ineffective entropy calculation. In contrast, false negatives are due to faulty regular expressions, skipping specific file types, and insufficient rulesets.

In the fifth study, we presented AssetHarvester, a static analysis tool to detect secret-asset pairs in a repository. Since the location of the asset can be distant from where the secret is defined, we investigated secret-asset co-location patterns and found four patterns. To identify the secret-asset pairs of the four patterns, we utilized three approaches (pattern matching, data flow analysis, and fast-approximation heuristics). We curated a benchmark of 1,791 secret-asset pairs of four database types extended from SecretBench to evaluate the performance of AssetHarvester. AssetHarvester demonstrates precision of (97%), recall (90%), and F1-score (94%) in detecting secret-asset pairs. Our findings indicate that data flow analysis employed in AssetHarvester detects secret-asset pairs with 0% false positives and aids in improving the recall of secret detection tools. Additionally, AssetHarvester shows 43% increase in precision for database secret detection compared to existing detection tools through the detection of assets, thus reducing developer's alert fatigue.

In the final study, we presented RiskHarvester, a risk-based tool to compute a security risk score based on the value of the asset and ease of attack on a database. We calculated the value of asset by identifying the sensitive data categories present in a database from the database keywords in the source code. We utilized data flow analysis, SQL, and Object Relational Mapper (ORM) parsing to identify the database keywords. To calculate the ease of attack, we utilized passive network analysis to retrieve the database host information. To evaluate RiskHarvester, we curated RiskBench, a benchmark of 1,791 database secret-asset pairs with sensitive data categories and host information manually retrieved from 188 GitHub repositories. RiskHarvester demonstrates precision of (95%) and recall (90%) in detecting database keywords for the value of asset and precision of (96%) and recall (94%) in detecting valid hosts for ease of attack. Finally, we conducted a survey to understand whether developers prioritize secret removal based on security risk score. We found that 86% of the developers prioritized the secrets for removal with descending security risks.

9.2 Future Work

We now discuss the future works based on the studies performed in this thesis.

Improvement of VCS History Sanitizing Tools: In Study I, we observed that developers

faced difficulty with properly sanitizing VCS history due to the limitations of existing VCS history sanitizing tools. These tools have safety and usability issues that can easily corrupt the repository's history. In addition, coming up with the correct shell script is difficult as developers find out if the sanitizing code script is right or wrong by trying the script out. Even worse, broken filters often result in silent incorrect rewrites without proper output. A future direction is to conduct a study on improving the easier integration of VCS history sanitizing tools for removing secrets from the source code history.

User Study on Developer and Organizational Practices: In Study II, we identified 24 developer and organizational practices for secure software secret management. However, we did not conduct any study with developers and organizations on whether they follow all the practices. A future direction is to conduct a survey with developers and organizations to identify which practices are being followed and the technical difficulties of implementing the practices.

SecretBench Extension: In Study III, we curated the SecretBench dataset by extracting secrets from GitHub repositories. However, we did not consider other VCS services such as GitLab and BitBucket. We will extend the SecretBench by extracting secrets from other VCS services in the future.

Automating the Secret Remediation Process: In Study IV, we evaluated open-source and proprietary secret detection tools, which provide remediation workflows when a secret is detected. However, currently, the workflow is a manual process where the leaked secret is assigned to the developer to revoke and rotate the secret. In addition, developers have to sanitize the Git history by themselves using history sanitizing tools such as BFG repo-cleaner. However, recent research [202] shows that malicious actors take only one minute to start making calls with the leaked API keys. Therefore, a future direction is to develop an automated workflow that the organizations can employ in their systems. The organization can mark the used secrets, and if a reported secret is among the used secrets, the workflow will automatically revoke and rotate the secrets. In addition, the workflow will sanitize the Git history without developers' manual effort, deploy new artifacts if needed, and review access logs to find any breaches.

AssetHarvester Extension: In Study V, we constructed AssetHarvester to detect secret-asset pairs of database types in a repository. However, we manually inspected the vendor documentation to formulate the regex patterns and identify the database driver functions (sinks). The future direction is to study how we can automate the process. LLMs can aid in identifying regex and sinks from source code patterns and vendor documentation for each secret type. We can leverage the knowledge gained from the manual analysis of our study

and generate prompts for LLMs.

RiskHarvester Extension: In Study VI, we calculated the ease of attack from the passive network information. However, developers wanted the deployment-related security information, such as the level of security controls present on the asset and network vulnerabilities (Section 8.3.2). The future direction is to extend RiskHarvester by leveraging the information provided by the active network scanning and vulnerability assessment tools such as Nessus [279] and Nmap [280]. Additionally, we will improve the value of asset by retrieving the metadata information of a database, such as whether the data is encrypted or hashed, using database auditing tools such as Datadog [281]. We will deploy RiskHarvester in the organization where these scanning and auditing tools are deployed.

BIBLIOGRAPHY

- [1] “Example Links to Questions and Developer Quotes,” <https://figshare.com/s/edebdcb73def3bdb7cfb>, [Online; accessed June 24, 2022].
- [2] “RiskHarvester Artifacts,” <https://figshare.com/s/cb458af04d2254e1956a>, [Online; accessed January 10, 2025].
- [3] “Grey literature review dataset,” <https://github.com/setu1421/Secret-Management-Practices-Grey-Literature-Review-Dataset>, [Online; accessed August 23, 2022].
- [4] “The State of Secrets Sprawl 2024,” <https://www.gitguardian.com/state-of-secrets-sprawl-report-2024>, [Online; accessed March 17, 2024].
- [5] M. Meli, M. R. McNiece, and B. Reaves, “How bad can it get? characterizing secret leakage in public github repositories.” in *NDSS*, 2019.
- [6] S. Nichols, “Popular mobile apps leaking AWS keys, exposing user data,” <https://www.techtarget.com/searchsecurity/news/252500361/Popular-mobile-apps-leaking-AWS-keys-exposing-user-data>, 2021, [Online; accessed December 25, 2021].
- [7] M. Jackson, “Uber Breach 2022 – Everything You Need to Know,” <https://blog.gitguardian.com/uber-breach-2022>, [Online; accessed March 10, 2024].
- [8] M. R. Rahman, N. Imtiaz, M.-A. Storey, and L. Williams, “Why secret detection tools are not enough: It’s not just about false positives-an industrial case study,” *Empirical Software Engineering*, vol. 27, no. 3, pp. 1–29, 2022.
- [9] “Security Risk,” <https://csrc.nist.gov/glossary/term/risk>, [Online; accessed April 5, 2024].
- [10] L. Williams, A. Meneely, and G. Shipley, “Protection poker: The new software security game”, *IEEE Security & Privacy*, vol. 8, no. 3, pp. 14–20, 2010.
- [11] “GitHub,” <https://github.com>, [Online; accessed March 3, 2022].
- [12] “GitLab,” <https://gitlab.com>, [Online; accessed March 3, 2022].
- [13] “About GitHub,” <https://github.com/about>, [Online; accessed Jan 3, 2023].
- [14] “TruffleHog,” <https://github.com/trufflesecurity/truffleHog>, [Online; accessed February 23, 2022].
- [15] “SpectralOps,” <https://spectralops.io>, [Online; accessed April 13, 2023].
- [16] C. Lefebvre, E. Manheimer, and J. Glanville, “Searching for studies,” *Cochrane handbook for systematic reviews of interventions: Cochrane book series*, pp. 95–150, 2008.

- [17] V. Garousi, M. Felderer, M. V. Mäntylä, and A. Rainer, *Benefitting from the Grey Literature in Software Engineering Research*. Cham: Springer International Publishing, 2020, pp. 385–413. [Online]. Available: https://doi.org/10.1007/978-3-030-32489-6_14
- [18] V. Garousi, M. Felderer, and M. V. Mäntylä, “Guidelines for including grey literature and conducting multivocal literature reviews in software engineering,” *Information and Software Technology*, vol. 106, pp. 101–121, 2019. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0950584918301939>
- [19] V. S. Sinha, D. Saha, P. Dhoolia, R. Padhye, and S. Mani, “Detecting and mitigating secret-key leaks in source code repositories,” in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, pp. 396–400.
- [20] A. Rahman, C. Parnin, and L. Williams, “The seven sins: Security smells in infrastructure as code scripts,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 164–175.
- [21] A. Rahman, M. R. Rahman, C. Parnin, and L. Williams, “Security smells in ansible and chef scripts: A replication study,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 1, pp. 1–31, 2021.
- [22] M. R. Rahman, A. Rahman, and L. Williams, “Share, but be aware: Security smells in python gists,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 536–540.
- [23] I. Koishybayev, A. Nahapetyan, R. Zachariah, S. Muralee, B. Reaves, A. Kapravelos, and A. Machiry, “Characterizing the security of github {CI} workflows,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2747–2763.
- [24] A. Rahman, F. L. Barsha, and P. Morrison, “Shhh!: 12 practices for secret management in infrastructure as code,” in *2021 IEEE Secure Development Conference (SecDev)*, 2021, pp. 56–62.
- [25] A. Krause, J. H. Klemmer, N. Huaman, D. Wermke, Y. Acar, and S. Fahl, “Committed by accident: Studying prevention and remediation strategies against secret leakage in source code repositories,” 2022.
- [26] A. Saha, T. Denning, V. Srikumar, and S. K. Kasera, “Secrets in source code: Reducing false positives using machine learning,” in *2020 International Conference on Communication Systems & NETWORKS (COMSNETS)*. IEEE, 2020, pp. 168–175.
- [27] R. Feng, Z. Yan, S. Peng, and Y. Zhang, “Automated detection of password leakage from public github repositories,” in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 175–186.

- [28] A. V. Konygin, A. V. Kopnin, I. P. Mezentsev, and A. A. Pankratov, "Using bigrams to detect leaked secrets in source code." in *ENASE*, 2023, pp. 589–596.
- [29] K. Bajaj, K. Pattabiraman, and A. Mesbah, "Mining questions asked by web developers," in *Proceedings of the 11th Working conference on mining software repositories*, 2014, pp. 112–121.
- [30] A. Rahman, E. Farhana, and N. Imtiaz, "Snakes in paradise?: Insecure python-related coding practices in stack overflow," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 200–204.
- [31] C. Rosen and E. Shihab, "What are mobile developers asking about? a large scale study using stack overflow," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1192–1223, 2016.
- [32] A. Barua, S. W. Thomas, and A. E. Hassan, "What are developers talking about? an analysis of topics and trends in stack overflow," *Empirical Software Engineering*, vol. 19, no. 3, pp. 619–654, 2014.
- [33] A. Rahman, A. Partho, P. Morrison, and L. Williams, "What questions do programmers ask about configuration as code?" in *Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering*, ser. RCoSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 16–22. [Online]. Available: <https://doi.org/10.1145/3194760.3194769>
- [34] A. Tahir, J. Dietrich, S. Counsell, S. Licorish, and A. Yamashita, "A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites," *Information and Software Technology*, vol. 125, p. 106333, 2020. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0950584920300926>
- [35] "How to keep secret key information out of Git repository?" <https://stackoverflow.com/questions/52293453>, [Online; accessed February 27, 2022].
- [36] "Stack exchange sites," <https://stackexchange.com/sites>, [Online; accessed December 23, 2021].
- [37] T. Zimmermann, "Card-sorting: From text to themes," in *Perspectives on Data Science for Software Engineering*, T. Menzies, L. Williams, and T. Zimmermann, Eds. Boston: Morgan Kaufmann, 2016, pp. 137–141. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/B9780128042069000271>
- [38] "Stack Overflow," <https://stackoverflow.com>, [Online; accessed January 3, 2022].
- [39] "Information Security," <https://security.stackexchange.com>, [Online; accessed January 3, 2022].

- [40] “Software Engineering,” <https://softwareengineering.stackexchange.com>, [Online; accessed January 3, 2022].
- [41] “Stack exchange data explorer,” <https://data.stackexchange.com>, [Online; accessed December 23, 2021].
- [42] “Stack exchange data dump,” <https://archive.org/details/stackexchange>, [Online; accessed December 23, 2021].
- [43] T. P. Johnson, “Snowball sampling: introduction,” 2014. [Online]. Available: <https://doi.org/10.1002/9781118445112.stat05720>
- [44] “Where to keep static information securely in Android app?” <https://stackoverflow.com/questions/61724202>, [Online; accessed June 15, 2022].
- [45] “GitHub Repository,” <https://github.com/setu1421/ICSE-2023-Artifacts>, [Online; accessed January 28, 2023].
- [46] J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen, “Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement,” *Sociological Methods & Research*, vol. 42, no. 3, pp. 294–320, 2013.
- [47] Y. Yao, H. Tong, T. Xie, L. Akoglu, F. Xu, and J. Lu, “Want a good answer? ask a good question first!” 2013.
- [48] J. H. Zar, *Spearman Rank Correlation*. John Wiley & Sons, Ltd, 2005.
- [49] “Feature Scaling,” https://en.wikipedia.org/w/index.php?title=Feature_scaling&oldid=1075231919, [Online; accessed April 18, 2023].
- [50] D. R. Cox and A. Stuart, “Some quick sign tests for trend in location and dispersion,” *Biometrika*, vol. 42, no. 1/2, pp. 80–95, 1955. [Online]. Available: <http://www.jstor.org/stable/2333424>
- [51] “Heroku,” <https://www.heroku.com>, [Online; accessed March 7, 2022].
- [52] “Google App Engine,” <https://cloud.google.com/appengine>, [Online; accessed March 7, 2022].
- [53] “Git Update Index (Notes),” https://git-scm.com/docs/git-update-index#_notes, [Online; accessed June 15, 2022].
- [54] “Manage Secrets & Protect Sensitive Data,” <https://www.vaultproject.io>, [Online; accessed March 3, 2022].
- [55] “Azure Key Vault,” <https://azure.microsoft.com/en-us/services/key-vault>, [Online; accessed March 3, 2022].

- [56] “Ignoring a previously committed file,” <https://www.atlassian.com/git/tutorials/saving-changes/gitignore>, [Online; accessed March 12, 2022].
- [57] “Smudge and clean your git working directory,” <https://www.atlassian.com/git/tutorials/saving-changes/gitignore>, [Online; accessed March 12, 2022].
- [58] “Gitignore templates,” <https://github.com/github/gitignore>, [Online; accessed April 13, 2022].
- [59] Michael DeHaan, “Ansible Vault,” <https://docs.ansible.com/ansible/latest/cli/ansible-vault.html>, [Online; accessed March 2, 2022].
- [60] “Chef Vault,” https://docs.chef.io/workstation/chef_vault, [Online; accessed March 2, 2022].
- [61] “Heroku Config Vars,” <https://devcenter.heroku.com/articles/config-vars>, [Online; accessed March 2, 2022].
- [62] Monica, Diogo, “Why you shouldn’t use ENV variables for secret data,” <https://diogomonica.com/2017/03/27/why-you-shouldnt-use-env-variables-for-secret-data>, 2017, [Online; accessed February 15, 2022].
- [63] “Git Filter Repo,” <https://github.com/newren/git-filter-repo>, [Online; accessed March 29, 2022].
- [64] “Git Filter Branch,” <https://git-scm.com/docs/git-filter-branch>, [Online; accessed March 7, 2022].
- [65] “BFG Repo Cleaner,” <https://rtyley.github.io/bfg-repo-cleaner>, [Online; accessed March 7, 2022].
- [66] “Removing sensitive data from a repository,” <https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/removing-sensitive-data-from-a-repository>, [Online; accessed February 13, 2022].
- [67] “git-secret,” <https://github.com/sobolevn/git-secret>, [Online; accessed February 23, 2022].
- [68] “git-crypt,” <https://github.com/AGWA/git-crypt>, [Online; accessed February 23, 2022].
- [69] “AWS Key Management Service,” <https://aws.amazon.com/kms>, [Online; accessed March 3, 2022].
- [70] “Git Submodules,” <https://git-scm.com/book/en/v2/Git-Tools-Submodules>, [Online; accessed February 15, 2022].

- [71] Cimpanu, Catalin, “Hacker gains access to a small number of Microsoft’s private GitHub repos,” <https://www.zdnet.com/article/hacker-gains-access-to-a-small-number-of-microsofts-private-github-repos>, 2020, [Online; accessed February 15, 2022].
- [72] “Git Hooks,” <https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks>, [Online; accessed February 23, 2022].
- [73] “Git Flag (Assume-unchanged),” <https://git-scm.com/docs/git-update-index#Documentation/git-update-index.txt---no-assume-unchanged>, [Online; accessed February 23, 2022].
- [74] “Git Flag (Skip-worktree),” <https://git-scm.com/docs/git-update-index#Documentation/git-update-index.txt---no-skip-worktree>, [Online; accessed February 23, 2022].
- [75] “Keeping your organization secure,” <https://docs.github.com/en/enterprise-cloud/latest/organizations/keeping-your-organization-secure>, [Online; accessed March 3, 2022].
- [76] “Gitrob,” <https://github.com/michenriksen/gitrob>, [Online; accessed February 23, 2022].
- [77] “Git Filter Branch Safety,” <https://git-scm.com/docs/git-filter-branch#SAFETY>, [Online; accessed January 27, 2023].
- [78] “Nine DevSecOps secret scanning tools to keep the bad guys at bay,” <https://www.cybersecasia.net/tips/nine-devsecops-scanning-tools-to-keep-the-bad-guys-at-bay>, [Online; accessed Jan 7, 2023].
- [79] “Foursquare API (Search for Venues),” <https://developer.foursquare.com/reference/v2-venues-search>, [Online; accessed April 15, 2022].
- [80] “Foursquare API exposing secret in javascript,” <https://stackoverflow.com/questions/32559855>, [Online; accessed June 15, 2022].
- [81] “What is the safest way to store user secrets in a .NET Core application?” <https://stackoverflow.com/questions/47316330>, [Online; accessed June 15, 2022].
- [82] Rick Anderson and Kirk Larkin, “Safe storage of app secrets in development in ASP.NET Core,” <https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets?tabs=windows&view=aspnetcore-6.0>, [Online; accessed June 15, 2022].
- [83] “Best practices for securely using API keys,” <https://support.google.com/googleapi/answer/6310037>, [Online; accessed June 14, 2022].
- [84] “How do I securely use Google API Keys,” <https://stackoverflow.com/questions/39625587>, [Online; accessed June 14, 2022].

- [85] “Continuous deployment for Azure Functions,” <https://docs.microsoft.com/en-us/azure/azure-functions/functions-continuous-deployment>, [Online; accessed June 11, 2022].
- [86] “How to configure Connection string in continuous deployment on Azure functions,” <https://stackoverflow.com/a/46790625/4299527>, [Online; accessed June 14, 2022].
- [87] “Apple FairPlay Streaming,” <https://developer.apple.com/streaming/fps/FairPlayStreamingOverview.pdf>, [Online; accessed June 29, 2022].
- [88] Cimpanu, Catalin, “Nissan source code leaked online after Git repo misconfiguration,” <https://www.zdnet.com/article/nissan-source-code-leaked-online-after-git-repo-misconfiguration>, 2021, [Online; accessed April 12, 2022].
- [89] “The National Institute of Standards and Technology (NIST),” <https://www.nist.gov/>, [Online; accessed June 14, 2022].
- [90] K. S. Murugiah Souppaya and D. Dodson, “Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities,” <https://csrc.nist.gov/publications/detail/sp/800-218/final>, [Online; accessed June 10, 2022].
- [91] “Stack Overflow Creative Commons Data Dump,” <https://stackoverflow.blog/2009/06/04/stack-overflow-creative-commons-data-dump>, [Online; accessed June 11, 2022].
- [92] “Academic Papers Using Stack Overflow Data,” <https://stackoverflow.blog/2010/05/31/academic-papers-using-stack-overflow-data>, [Online; accessed June 11, 2022].
- [93] “Attribution Required,” <https://stackoverflow.blog/2009/06/25/attribution-required>, [Online; accessed June 11, 2022].
- [94] “CodeProject,” <https://www.codeproject.com>, [Online; accessed March 16, 2022].
- [95] “Coderanch,” <https://coderanch.com>, [Online; accessed March 16, 2022].
- [96] “Is it completely safe to publish an ssh public key?” <https://security.stackexchange.com/questions/150540>, [Online; accessed June 15, 2022].
- [97] “Best practice for storing sensitive connection data when connecting to a db,” <https://stackoverflow.com/questions/66070209>, [Online; accessed May 16, 2022].
- [98] A. A. U. Rahman and L. Williams, “Software security in devops: Synthesizing practitioners’ perceptions and practices,” in *2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED)*, 2016, pp. 70–76.

- [99] C. Lefebvre, E. Manheimer, and J. Glanville, *Searching for Studies*. John Wiley & Sons, Ltd, 2008, ch. 6, pp. 95–150. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470712184.ch6>
- [100] J. Saldaña, *The coding manual for qualitative researchers*, 2015.
- [101] M. M. Hasan, F. A. Bhuiyan, and A. Rahman, “Testing practices for infrastructure as code,” ser. LANGETI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 7–12. [Online]. Available: <https://doi.org/10.1145/3416504.3424334>
- [102] “The Twelve-Factor App,” <https://12factor.net/config>, [Online; accessed April 22, 2022].
- [103] “dotenv,” <https://www.npmjs.com/package/dotenv>, [Online; accessed April 22, 2022].
- [104] P. Adajah, “Best practices for managing and storing secrets in frontend development,” <https://blog.logrocket.com/best-practices-for-managing-and-storing-secrets-in-frontend-development>, [Online; accessed April 22, 2022].
- [105] “Scanning the alexa top 1m for .ds_store files,” <https://en.internetwache.org/scanning-the-alexa-top-1m-for-ds-store-files-12-03-2018>, [Online; accessed April 20, 2022].
- [106] S. Nichols, “Popular mobile apps leaking aws keys, exposing user data,” <https://www.techtarget.com/searchsecurity/news/252500361/Popular-mobile-apps-leaking-AWS-keys-exposing-user-data>, [Online; accessed March 25, 2022].
- [107] “Pinterest Knox,” <https://github.com/pinterest/knox>, [Online; accessed March 3, 2022].
- [108] “Ruby on rails 5.1 release notes,” https://edgeguides.rubyonrails.org/5_1_release_notes.html, [Online; accessed April 21, 2022].
- [109] M. Jackson, *Best practices for managing and storing secrets including API keys and other credentials*, <https://blog.gitguardian.com/secrets-api-management>.
- [110] A. Hernandez, “What do we know about the microsoft azure outage?” <https://www.venafi.com/blog/what-do-we-know-about-microsoft-azure-outage>, [Online; accessed April 20, 2022].
- [111] S. Gooding, “Ryan hellyer’s aws nightmare: Leaked access keys result in a \$6,000 bill overnight,” <https://wptavern.com/ryan-hellyers-aws-nightmare-leaked-access-keys-result-in-a-6000-bill-overnight>, [Online; accessed March 19, 2022].
- [112] “Audit events,” https://docs.gitlab.com/ee/administration/audit_events.html, [Online; accessed May 13, 2022].

- [113] “git-all-secrets,” <https://github.com/anshumanbh/git-all-secrets>, [Online; accessed June 26, 2023].
- [114] “Azure Pipeline Variables,” <https://docs.microsoft.com/en-us/azure/devops/pipelines/process/variables>, [Online; accessed March 17, 2022].
- [115] “GitHub Encrypted secrets,” <https://docs.github.com/en/actions/security-guides/encrypted-secrets>, [Online; accessed March 17, 2022].
- [116] “GitLab CI/CD variables,” <https://docs.microsoft.com/en-us/azure/devops/pipelines/process/variables>, [Online; accessed March 17, 2022].
- [117] “Source code disclosure via exposed .git folder,” <https://pentester.land/tutorials/2018/10/25/source-code-disclosure-via-exposed-git-folder.html>, [Online; accessed March 10, 2022].
- [118] “Restricting repository visibility changes in your organization,” <https://docs.github.com/en/organizations/managing-organization-settings/restricting-repository-visibility-changes-in-your-organization>, [Online; accessed April 08, 2022].
- [119] “Security best practices for GitHub,” <https://spectralops.io/resources/how-to-choose-a-secret-scanning-solution-to-protect-credentials-in-your-code>, [Online; accessed March 23, 2022].
- [120] “Setting permissions for adding outside collaborators,” <https://docs.github.com/en/enterprise-cloud@latest/organizations/managing-organization-settings/setting-permissions-for-adding-outside-collaborators>, [Online; accessed May 08, 2022].
- [121] “Requiring two-factor authentication in your organization,” <https://docs.github.com/en/organizations/keeping-your-organization-secure/managing-two-factor-authentication-for-your-organization/requiring-two-factor-authentication-in-your-organization>, [Online; accessed May 04, 2022].
- [122] “Enforce two-factor authentication,” https://docs.gitlab.com/ee/security/two_factor_authentication.html, [Online; accessed May 04, 2022].
- [123] “Managing commit signature verification,” <https://docs.github.com/en/authentication/managing-commit-signature-verification>, [Online; accessed May 14, 2022].
- [124] “Adding a security policy to your repository,” <https://docs.github.com/en/code-security/getting-started/adding-a-security-policy-to-your-repository>, [Online; accessed May 03, 2022].
- [125] “About authentication with SAML single sign-on,” <https://docs.github.com/en/enterprise-cloud@latest/authentication/authenticating-with-saml-single-sign-on/about-authentication-with-saml-single-sign-on>, [Online; accessed May 05, 2022].

- [126] “Managing the forking policy for your organization,” <https://docs.github.com/en/organizations/managing-organization-settings/managing-the-forking-policy-for-your-organization>, [Online; accessed April 19, 2022].
- [127] “Getting started with Credential Scanner (CredScan),” <https://secdevtools.azurewebsites.net/helpcredscan.html>, [Online; accessed Jan 2, 2023].
- [128] “GitHub on BigQuery: Analyze all the open source code,” <https://cloud.google.com/blog/topics/public-datasets/github-on-bigquery-analyze-all-the-open-source-code>, [Online; accessed Jan 5, 2023].
- [129] “Google Cloud Storage,” <https://cloud.google.com/storage>, [Online; accessed Dec 24, 2022].
- [130] “GitHub Public Repositories,” <https://github.com/search?q=is:public>, [Online; accessed Jan 3, 2023].
- [131] “TruffleHog,” <https://github.com/trufflesecurity/trufflehog/tree/main/pkg/detectors>, [Online; accessed Jan 4, 2023].
- [132] “SecretBench Regular Expressions,” <https://doi.org/10.5281/zenodo.7555981>, [Online; accessed Jan 22, 2023].
- [133] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, “The promises and perils of mining github,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 92–101. [Online]. Available: <https://doi.org/10.1145/2597073.2597074>
- [134] “Getting started with the REST API,” <https://docs.github.com/en/rest/guides/getting-started-with-the-rest-api?apiVersion=2022-11-28>, [Online; accessed Jan 2, 2023].
- [135] N. E. Young, *Greedy Set-Cover Algorithms*. Boston, MA: Springer US, 2008, pp. 379–381. [Online]. Available: https://doi.org/10.1007/978-0-387-30162-4_175
- [136] “GitPython,” <https://github.com/gitpython-developers/GitPython>, [Online; accessed Dec 24, 2022].
- [137] “Google Cloud Compute Engine,” <https://cloud.google.com/compute>, [Online; accessed Dec 26, 2022].
- [138] “Gitleaks,” <https://gitleaks.io>, [Online; accessed Dec 26, 2022].
- [139] J. Cohen, “A coefficient of agreement for nominal scales,” *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960. [Online]. Available: <https://doi.org/10.1177/001316446002000104>

- [140] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977. [Online]. Available: <http://www.jstor.org/stable/2529310>
- [141] Spencer EA, Brassey J, Mahtani K, "Recall bias," <https://www.catalogueofbiases.org/biases/recall-bias>, [Online; accessed Jan 3, 2023].
- [142] "Setting your commit email address," <https://docs.github.com/en/account-and-profile/setting-up-and-managing-your-personal-account-on-github/managing-email-preferences/setting-your-commit-email-address>, [Online; accessed Dec 22, 2022].
- [143] Nunan D, Bankhead C, Aronson JK, "Selection bias," <https://catalogofbias.org/biases/selection-bias>, [Online; accessed Jan 3, 2023].
- [144] "Google Common English Words," <https://github.com/first20hours/google-10000-english>, [Online; accessed Dec 21, 2022].
- [145] "paradite/hn-ratio," <https://github.com/paradite/hn-ratio>, [Online; accessed Jan 17, 2023].
- [146] "SecretBench GitHub Repository," <https://github.com/setu1421/SecretBench>, [Online; accessed Jan 22, 2023].
- [147] "SecretBench Additional Features," <https://doi.org/10.5281/zenodo.7555981>, [Online; accessed Jan 22, 2023].
- [148] "Bitbucket," <https://bitbucket.org>, [Online; accessed Jan 8, 2023].
- [149] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security & Privacy*, vol. 2, no. 6, pp. 76–79, 2004.
- [150] Hadjy, Paul, "What Is Alert Fatigue? 4 Ways to Mitigate It and Prevent Burnout," <https://learn.g2.com/alert-fatigue>, [Online; accessed April 12, 2023].
- [151] "False Positive Secret Dataset," <https://github.com/setu1421/FPSecretBench>, [Online; accessed July 02, 2023].
- [152] S. K. Basak, L. Neil, B. Reaves, and L. Williams, "Secretbench: A dataset of software secrets," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023, pp. 347–351.
- [153] "SecretBench File Types," <https://github.com/setu1421/SecretBench/tree/main/Metadata>, [Online; accessed June 26, 2023].
- [154] "ACM Digital Library," <https://dl.acm.org>, [Online; accessed April 11, 2023].
- [155] "SpringerLink," <https://link.springer.com>, [Online; accessed April 11, 2023].

- [156] “IEEE Xplore,” <https://ieeexplore.ieee.org/Xplore/home.jsp>, [Online; accessed April 14, 2023].
- [157] “DBLP,” <https://dblp.org>, [Online; accessed April 13, 2023].
- [158] “ScienceDirect,” <https://www.sciencedirect.com>, [Online; accessed April 11, 2023].
- [159] S. Lounici, M. Rosa, C. Negri, S. Trabelsi, and M. Önen, “Optimizing leak detection in open-source platforms with machine learning techniques,” 01 2021, pp. 145–159.
- [160] “CredScan,” <https://secdevtools.azurewebsites.net/helpcredscan.html>, [Online; accessed April 10, 2023].
- [161] “Cycode: The Application Security Platform,” <https://cycode.com>, [Online; accessed April 10, 2023].
- [162] “detect-secrets,” <https://github.com/Yelp/detect-secrets>, [Online; accessed April 14, 2023].
- [163] “git-hound,” <https://github.com/tillson/git-hound>, [Online; accessed June 26, 2023].
- [164] “Gittyleaks,” <https://github.com/kootenpv/gittyleaks>, [Online; accessed June 26, 2023].
- [165] “repo-security-scanner,” <https://github.com/techjacker/repo-security-scanner>, [Online; accessed April 14, 2023].
- [166] E. Wen, J. Wang, and J. Dietrich, “Secrethunter: A large-scale secret scanner for public git repositories,” in *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2022, pp. 123–130.
- [167] “git-secrets,” <https://github.com/awslabs/git-secrets>, [Online; accessed April 15, 2023].
- [168] “Amazon Web Services - Labs,” <https://github.com/awslabs>, [Online; accessed April 15, 2023].
- [169] “Repo-supervisor,” <https://github.com/auth0/repo-supervisor>, [Online; accessed April 13, 2023].
- [170] “Truffle Security,” <https://trufflesecurity.com>, [Online; accessed April 14, 2023].
- [171] “Whispers,” <https://github.com/Skyscanner/whispers>, [Online; accessed April 13, 2023].
- [172] “ggshield,” <https://github.com/GitGuardian/ggshield>, [Online; accessed April 13, 2023].

- [173] “The State of Secrets Sprawl 2022,” <https://blog.gitguardian.com/the-state-of-secrets-sprawl-2022>, [Online; accessed March 16, 2022].
- [174] “GitGuardian API,” <https://api.gitguardian.com/docs>, [Online; accessed April 13, 2023].
- [175] “py-gitguardian: GitGuardian API Client,” <https://github.com/GitGuardian/py-gitguardian>, [Online; accessed April 13, 2023].
- [176] “Github Secret Scanner,” <https://docs.github.com/en/code-security/secret-scanning>, [Online; accessed April 16, 2023].
- [177] W. E. Winkler, “String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage.” 1990.
- [178] M. A. Jaro, “Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida,” *Journal of the American Statistical Association*, vol. 84, no. 406, pp. 414–420, 1989.
- [179] “Python jellyfish package,” <https://pypi.org/project/jellyfish>, [Online; accessed March 28, 2023].
- [180] P. E. Black, “Ratcliff/obershelp pattern recognition,” *Dictionary of algorithms and data structures*, vol. 17, 2004.
- [181] “SequenceMatcher of difflib package,” <https://docs.python.org/3/library/difflib.html#module-difflib>, [Online; accessed March 28, 2023].
- [182] “Python time package,” <https://docs.python.org/3/library/time.html>, [Online; accessed March 28, 2023].
- [183] J. Zhu, M. Zhou, and A. Mockus, “Patterns of folder use and project popularity: A case study of github repositories,” in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM ’14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: <https://doi.org/10.1145/2652524.2652564>
- [184] H. Borges and M. Tulio Valente, “What’s in a github star? understanding repository starring practices in a social coding platform,” *Journal of Systems and Software*, vol. 146, pp. 112–129, 2018. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0164121218301961>
- [185] “Kaggle’s GitHub Repository Dataset,” <https://www.kaggle.com/code/pelmers/explore-github-repository-metadata>, [Online; accessed March 12, 2023].
- [186] C. E. Shannon, “A mathematical theory of communication,” *The Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

- [187] “Improving TruffleHog’s Entropy Calculation,” <https://github.com/trufflesecurity/trufflehog/issues/168>, [Online; accessed April 24, 2023].
- [188] “Repo-supervisor: CSS Filters,” <https://github.com/auth0/repo-supervisor/blob/3d6252571318a24c1ffbaba48e024d79e44f9ac0/src/filters/entropy.meter/pre.filters/css.selectors.js>, [Online; accessed April 24, 2023].
- [189] “TruffleHog: Prefix Regex,” <https://github.com/trufflesecurity/trufflehog/blob/main/pkg/detectors/strava/strava.go>, [Online; accessed April 13, 2023].
- [190] “GitGuardian: Secrets Detection Engine,” https://docs.gitguardian.com/secrets-detection/quick_start#how-it-works, [Online; accessed April 10, 2023].
- [191] “IGDB API Docs: Getting Started,” <https://api-docs.igdb.com/#getting-started>, [Online; accessed April 10, 2023].
- [192] “Mashape API Documentation,” <https://rapidapi.com/rokity/api/mashape>, [Online; accessed April 10, 2023].
- [193] “DevOps tech: Shifting left on security,” <https://cloud.google.com/architecture/devops/devops-tech-shifting-left-on-security>, [Online; accessed April 23, 2023].
- [194] “GitHub Actions,” <https://github.com/features/actions>, [Online; accessed April 20, 2023].
- [195] “Travis CI,” <https://www.travis-ci.com>, [Online; accessed April 12, 2023].
- [196] “CircleCI,” <https://circleci.com>, [Online; accessed April 15, 2023].
- [197] “TruffleHog AWS Detector,” <https://github.com/trufflesecurity/trufflehog/blob/main/pkg/detectors/aws/aws.go>, [Online; accessed April 15, 2023].
- [198] “2022 State of APIs,” <https://stateofapis.com/>, [Online; accessed April 25, 2023].
- [199] “TruffleHog: Verified secrets - unreachable website,” <https://github.com/trufflesecurity/trufflehog/issues/1112>, [Online; accessed April 10, 2023].
- [200] “GitHub Secret scanning partner program,” <https://docs.github.com/en/code-security/secret-scanning/secret-scanning-partner-program>, [Online; accessed April 10, 2023].
- [201] “GitHub Secret scanning patterns,” <https://docs.github.com/en/code-security/secret-scanning/secret-scanning-patterns#supported-secrets>, [Online; accessed April 10, 2023].
- [202] Hercz, Tibor, “What happens when you leak AWS credentials and how AWS minimizes the damage,” <https://xebia.com/blog/what-happens-when-you-leak-aws-credentials-and-how-aws-minimizes-the-damage>, [Online; accessed April 10, 2023].

- [203] S. K. Basak, J. Cox, B. Reaves, and L. Williams, "A comparative study of software secrets reporting by secret detection tools," in *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2023, pp. 1–12.
- [204] "AssetBench and AssetHarvester Artifacts," <https://figshare.com/s/c8bf9140d5a4fde44a87>, [Online; accessed March 20, 2024].
- [205] "PostgreSQL," <https://www.postgresql.org>, [Online; accessed February 14, 2024].
- [206] "MySQL," <https://www.mysql.com>, [Online; accessed March 14, 2024].
- [207] "SQLite," <https://www.sqlite.org>, [Online; accessed February 14, 2024].
- [208] "MongoDB," <https://www.mongodb.com>, [Online; accessed February 14, 2024].
- [209] "Microsoft SQL Server," <https://www.microsoft.com/en-us/sql-server/sql-server-2022>, [Online; accessed February 14, 2024].
- [210] "Stack Overflow Developer Survey, 2023," <https://survey.stackoverflow.co/2023/#most-popular-technologies-database>, [Online; accessed February 14, 2024].
- [211] "The State of Secrets Sprawl 2023," <https://www.gitguardian.com/state-of-secrets-sprawl-report-2023>, [Online; accessed July 17, 2024].
- [212] "MySQL Connection String," <https://dev.mysql.com/doc/refman/8.0/en/connecting-using-uri-or-key-value-pairs.html>, [Online; accessed February 16, 2024].
- [213] "PostgreSQL Connection String," <https://www.postgresql.org/docs/current/libpq-connect.html#LIBPQ-CONNSTRING>, [Online; accessed February 16, 2024].
- [214] "MongoDB Connection String," <https://www.mongodb.com/docs/manual/reference/connection-string>, [Online; accessed February 16, 2024].
- [215] "Microsoft Open Database Connectivity," <https://learn.microsoft.com/en-us/sql/odbc/microsoft-open-database-connectivity-odbc>, [Online; accessed February 16, 2024].
- [216] J. A. Blakeley, "Ole db: a component dbms architecture," in *Proceedings of the twelfth international conference on data engineering*. IEEE Computer Society, 1996, pp. 203–203.
- [217] "Java JDBC API," <https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc>, [Online; accessed February 16, 2024].
- [218] "JayDeBeApi," <https://pypi.org/project/JayDeBeApi>, [Online; accessed February 16, 2024].

- [219] “Named Capturing Group,” https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Regular_expressions/Named_capturing_group, [Online; accessed February 16, 2024].
- [220] “re - Regular Expression Operations,” <https://docs.python.org/3/library/re.html>, [Online; accessed February 16, 2024].
- [221] U. Khedker, A. Sanyal, and B. Sathe, *Data flow analysis: theory and practice*. CRC Press, 2017.
- [222] A. Rahman and C. Parnin, “Detecting and characterizing propagation of security weaknesses in puppet-based infrastructure management,” *IEEE Transactions on Software Engineering*, vol. 49, no. 06, pp. 3536–3553, jun 2023.
- [223] “peewee,” <https://docs.peewee-orm.com/en/latest>, [Online; accessed February 18, 2024].
- [224] “SQLAlchemy,” <https://docs.sqlalchemy.org/en/20>, [Online; accessed February 18, 2024].
- [225] “Object Relational Mapping (ORM),” <https://www.theserverside.com/definition/object-relational-mapping-ORM>, [Online; accessed February 19, 2024].
- [226] “Positional and Keyword Arguments,” <https://problemsolvingwithpython.com/07-Functions-and-Modules/07.07-Positional-and-Keyword-Arguments>, [Online; accessed February 19, 2024].
- [227] “aiomysql,” <https://aiomysql.readthedocs.io/en/stable>, [Online; accessed February 18, 2024].
- [228] “mysql-connector,” <https://dev.mysql.com/doc/connector-python/en>, [Online; accessed February 18, 2024].
- [229] “PyMySQL,” <https://pymysql.readthedocs.io/en/latest>, [Online; accessed February 18, 2024].
- [230] “aiopg,” <https://aiopg.readthedocs.io/en/stable>, [Online; accessed February 18, 2024].
- [231] “asyncpg,” <https://magicstack.github.io/asyncpg/current>, [Online; accessed February 18, 2024].
- [232] “psycopg2,” <https://pypi.org/project/psycopg2>, [Online; accessed February 18, 2024].
- [233] “pymongo,” <https://pymongo.readthedocs.io/en/stable>, [Online; accessed March 4, 2024].

- [234] “pymssql,” <https://www.pymssql.org>, [Online; accessed March 2, 2024].
- [235] “pyodbc,” <https://pypi.org/project/pyodbc>, [Online; accessed February 18, 2024].
- [236] “CodeQL,” <https://codeql.github.com>, [Online; accessed March 4, 2024].
- [237] O. d. Moor, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, D. Sereni, and J. Tibble, “Keynote address: .ql for source code analysis,” in *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, 2007, pp. 3–16.
- [238] “Using API graphs in Python,” <https://codeql.github.com/docs/codeql-language-guides/using-api-graphs-in-python>, [Online; accessed February 19, 2024].
- [239] “PyYAML,” <https://pyyaml.org/wiki/PyYAMLDocumentation>, [Online; accessed February 19, 2024].
- [240] “json - JSON encoder and decoder,” <https://docs.python.org/3/library/json.html>, [Online; accessed February 19, 2024].
- [241] “xmldict,” <https://pypi.org/project/xmldict>, [Online; accessed February 19, 2024].
- [242] “linecache — Random access to text lines,” <https://docs.python.org/3/library/linecache.html>, [Online; accessed February 20, 2024].
- [243] “CodeQL Hardcoded Credentials,” <https://github.com/github/codeql/blob/main/go/ql/src/Security/CWE-798/HardcodedCredentials.ql>, [Online; accessed July 23, 2024].
- [244] “GitHub Secret Scanning Patterns,” <https://docs.github.com/en/code-security/secret-scanning/secret-scanning-patterns>, [Online; accessed July 23, 2024].
- [245] “Detection of generic secrets with secret scanning,” <https://docs.github.com/en/code-security/secret-scanning/about-the-detection-of-generic-secrets-with-secret-scanning>, [Online; accessed July 23, 2024].
- [246] “smtplib — SMTP protocol client,” <https://docs.python.org/3/library/smtplib.html>, [Online; accessed March 11, 2024].
- [247] “Automated severity scoring comes to the GitGuardian secrets detection platform,” <https://blog.gitguardian.com/automated-severity-scoring-comes-to-the-gitguardian-secrets-detection-platform>, [Online; accessed April 5, 2024].
- [248] S. K. Basak, K. V. English, K. Ogura, V. Kambara, B. Reaves, and L. Williams, “Assetharvester: A static analysis tool for detecting secret-asset pairs in software artifacts,” 2024. [Online]. Available: <https://arxiv.org/abs/2403.19072>

- [249] “Stack Overflow Developer Survey, 2024,” <https://survey.stackoverflow.co/2024/technology#1-databases>, [Online; accessed December 14, 2024].
- [250] “Google Cloud Sensitive Data Protection (Cloud DLP),” <https://cloud.google.com/security/products/sensitive-data-protection>, [Online; accessed December 4, 2024].
- [251] “PostgreSQL Connection String,” <https://www.postgresql.org/docs/current/libpq-connect.html#LIBPQ-CONNSTRING>, [Online; accessed December 16, 2024].
- [252] “MongoDB Drivers,” <https://www.mongodb.com/docs/drivers>, [Online; accessed December 19, 2024].
- [253] “Censys Search Python Library,” <https://support.censys.io/hc/en-us/articles/360056141971-Censys-Search-Python-Library>, [Online; accessed December 26, 2024].
- [254] “Qualtrics XM: The Leading Experience Management Software,” <https://www.qualtrics.com>, [Online; accessed December 13, 2024].
- [255] “Flask-PyMongo,” <https://flask-pymongo.readthedocs.io/en/latest>, [Online; accessed December 18, 2024].
- [256] “Django ORM,” <https://docs.djangoproject.com/en/5.1/ref/databases>, [Online; accessed December 18, 2024].
- [257] “Python Built-in File Open Function,” <https://docs.python.org/3/library/functions.html#open>, [Online; accessed January 5, 2025].
- [258] “sql-metadata,” https://pypi.org/project/sql_metadata, [Online; accessed December 4, 2024].
- [259] “py-models-parser,” <https://github.com/xnuinside/py-models-parser>, [Online; accessed December 4, 2024].
- [260] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, “Enriching word vectors with subword information,” *Transactions of the association for computational linguistics*, vol. 5, pp. 135–146, 2017.
- [261] T. Mikolov, “Efficient estimation of word representations in vector space,” *arXiv preprint arXiv:1301.3781*, vol. 3781, 2013.
- [262] 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.
- [263] “fastText,” <https://pypi.org/project/fasttext>, [Online; accessed December 28, 2024].
- [264] “Google Cloud Translation API,” <https://cloud.google.com/translate/docs/reference/rest>, [Online; accessed December 28, 2024].

- [265] “google-cloud-translate,” <https://pypi.org/project/google-cloud-translate>, [Online; accessed December 28, 2024].
- [266] “Internet Engineering Task Force (IETF),” <https://www.ietf.org>, [Online; accessed December 28, 2024].
- [267] “Request for Comments (RFCs),” <https://www.ietf.org/rfc>, [Online; accessed December 28, 2024].
- [268] “validators - Python Data Validation for Humans,” <https://yozachar.github.io/pyvalidators/stable>, [Online; accessed December 28, 2024].
- [269] Y. Liu, T. Han, S. Ma, J. Zhang, Y. Yang, J. Tian, H. He, A. Li, M. He, Z. Liu *et al.*, “Summary of chatgpt-related research and perspective towards the future of large language models,” *Meta-Radiology*, p. 100017, 2023.
- [270] J. Yang, H. Jin, R. Tang, X. Han, Q. Feng, H. Jiang, S. Zhong, B. Yin, and X. Hu, “Harnessing the power of llms in practice: A survey on chatgpt and beyond,” *ACM Transactions on Knowledge Discovery from Data*, vol. 18, no. 6, pp. 1–32, 2024.
- [271] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, “Chain-of-thought prompting elicits reasoning in large language models,” *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.
- [272] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [273] “GPT Models,” <https://platform.openai.com/docs/models/gp#gpt-4o>, [Online; accessed December 28, 2024].
- [274] “nslookup - Linux man page,” <https://linux.die.net/man/1/nslookup>, [Online; accessed December 27, 2024].
- [275] “Python’s nslookup Package,” <https://pypi.org/project/nslookup>, [Online; accessed December 27, 2024].
- [276] “ipaddress — IPv4/IPv6 manipulation library,” <https://docs.python.org/3/library/ipaddress.html>, [Online; accessed December 26, 2024].
- [277] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J. A. Halderman, “A search engine backed by internet-wide scanning,” in *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, 2015, pp. 542–553.
- [278] A. Agresti, *Categorical data analysis*. John Wiley & Sons, 2012, vol. 792.
- [279] “Nessus Vulnerability Scanner: Network Security Solution,” <https://www.tenable.com/products/nessus>, [Online; accessed December 17, 2024].

- [280] “Nmap: the Network Mapper - Free Security Scanner,” <https://nmap.org>, [Online; accessed December 17, 2024].
- [281] “Datadog: Cloud Monitoring as a Service,” <https://www.datadoghq.com>, [Online; accessed December 17, 2024].
- [282] R. Tourangeau, “The psychology of survey response,” *University of Cambridge*, 2000.
- [283] “USENIX Security ’25 Ethics Guidelines,” <https://www.usenix.org/conference/usenixsecurity25/ethics-guidelines>, [Online; accessed December 17, 2024].

APPENDICES

APPENDIX

A

STUDY I SUPPLEMENTAL INFORMATION

A.1 Mapping of Question Category to Answer Category

In Table A.1, we provided the mapping of answer categories to each question category (Chapter 3).

Table A.1: The mapping of answers to each question category

Domain	Question Category	Answer Category
Secrets	Q1: Store/Version	Revocation and Rotation, Move Secrets out of Source Code/Version Control and Use Template Config File, Secret Management in Deployment (Deployment Variables, Config Transformations), Use of External Secret Management Service (HashiCorp Vault and Azure key Vault), Use Local Environment Variables, Restrict API Access and Permissions, Load Externally and Use Secondary Private Repository, Server-Side Implementation

Table A.1: The mapping of answers to each question category

Domain	Question Category	Answer Category
Secrets	Q2: Ignore/Hide	Revocation and Rotation, Rewrite VCS History (Use git-filter-branch, git-filter-repo and BFG repo cleaner), Move Secrets out of Source Code/Version Control and Use Template Config File, VCS Feature (Git Hooks: Git attribute custom driver, pre-commit and post-commit hooks), Store Encrypted/Obfuscated Secrets, Secret Management in Deployment (Deployment Variables, Config Transformations, Shared Directory), Use of External Secret Management Service (HashiCorp Vault and Azure key Vault), Use Local Environment Variables, Load Externally and Use Secondary Private Repository, Server-Side Implementation
	Q3: Exploitability	Revocation and Rotation, Rewrite VCS History (Use of git-filter-branch), Move Secrets out of Source Code/Version Control and Use Template Config File
	Q4: Distribute	Move Secrets out of Source Code/Version Control and Use Template Config File, Secret Management in Deployment, Load Externally and Use Secondary Private Repository
	Q5: Restriction	Restrict API Access and Permissions
Deployment	Q6: Store/Version	Move Secrets out of Source Code/Version Control and Use Template Config File, VCS Feature (Git attribute custom driver), Store Encrypted/Obfuscated Secrets, Secret Management in Deployment (Config Transformation, Deployment Variables, Configuration Management Systems (Puppet, Ansible-Vault)), Use of External Secret Management Service (HashiCorp Vault, Azure Key Vault), Use Local Environment Variables, Restrict API Access and Permissions, Load Externally (S3) and Use Secondary Private Repository, Server-Side Implementation
	Q7: Improper Configuration	Move Secrets out of Source Code/Version Control and Use Template Config File, Secret Management in Deployment (Deployment Tools: GitHub Actions, Deployment Variables), External Secret Management Service (AWS Secret Manager), Use Local Environment Variables
	Q8: Ignore/Hide	Move Secrets out of Source Code/Version Control and Use Template Config File, Store Encrypted/Obfuscated Secrets (Mask Passwords Plugin), Secret Management in Deployment (Deployment Variables, Keep dot file out of root directory), Use of External Secret Management Service (Azure key Vault), Use Local Environment Variables
	Q9: Dot File	Secret Management in Deployment (Keep dot file out of root directory)
VCS Feature	Q10: History Sanitize	Revocation and Rotation, Rewrite VCS History (Use git-filter-branch, git-filter-repo, BFG repo cleaner, git reset –hard, git rebase), Move Secrets out of Source Code/Version Control and Use Template Config File, VCS Scan Tools (Truffle Hog, Gitrob), Load Externally and Use Secondary Private Repository
	Q11: Ignore Already Committed	Rewrite VCS History, Move Secrets out of Source Code/Version Control and Use Template Config File, VCS Feature (Git Flags: git –skip-worktree, git update-index –assume-unchanged), Use Local Environment Variable
	Q12: Line Level Security	Move Secrets out of Source Code/Version Control and Use Template Config File, VCS Feature (Git Hooks: Git attribute custom driver, pre-commit and post-update hook), Add Files to the Staging Area Explicitly, Store Encrypted/Obfuscated Secrets (Rails Secrets), Use Local Environment Variables, Load Externally and Use Secondary Private Repository
	Q13: Encrypt File	VCS Feature (Git Hooks: pre-commit and post-commit)

Table A.1: The mapping of answers to each question category

Domain	Question Category	Answer Category
Configur- ation File	Q14: Store/Version	Move Secrets out of Source Code/Version Control and Use Template Config File, Store Encrypted/Obfuscated Secrets, Secret Management in Deployment (Deployment Variables, Config Transformations, Shared Directory), Use of External Secret Management Service, Use Local Environment Variables, Load Externally and Use Secondary Private Repository
	Q15: Ignore/Hide	Move Secrets out of Source Code/Version Control and Use Template Config File, Add Files to the Staging Area Explicitly, Store Encrypted/Obfuscated Secrets, Secret Management in Deployment, Use Local Environment Variables, Load Externally and Use Secondary Private Repository
	Q16: Distribute	Move Secrets out of Source Code/Version Control and Use Template Config File
	Q17: Exploitability	Use Local Environment Variables, Use of External Secret Management Service, Move Secrets out of Source Code/Version Control and Use Template Config File
	Q18: Accessibility	Secret Management in Deployment (Config Transformation)
Pre-open Source	Q19: Cross-check	Rewrite VCS History (Use of git-filter-branch, git-filter-repo, BFG repo cleaner), Move Secrets out of Source Code/Version Control and Use Template Config File, Store Encrypted/Obfuscated Secrets, Secret Management in Deployment (Deployment Variables, Mask Secrets in log), Use of External Secret Management Service (Cloud KMS), Use Local Environment Variables, Load Externally and Use Secondary Private Repository, Server-Side Implementation
Client-Side Application	Q20: Store	Revokation and Rotation, Store Encrypted/Obfuscated Secrets, Use Local Environment Variables, Move Secrets out of Source Code/Version Control and Use Template Config File, Server-Side Implementation
	Q21: Hide	Revokation and Rotation, Use Local Environment Variables, Move Secrets out of Source Code/Version Control and Use Template Config File, Server-Side Implementation
	Q22: Exploitability	Move Secrets out of Source Code/Version Control and Use Template Config File
Secure-ness	Q23: Private Repository	Use of External Secret Management Service, Move Secrets out of Source Code/Version Control and Use Template Config File, Load Externally and Use Secondary Private Repository
	Q24: Unpushed Branch	Move Secrets out of Source Code/Version Control and Use Template Config File
Ext. Sec. Mng..	Q25: Setup	Load Externally and Use Secondary Private Repository
Others	Q26: Importance	No Accepted Answer
	Q27: Decision	Revokation and Rotation

Table B.2: System and User role prompt for detecting placeholder/dummy DNS names.

Type	Chain-of-Thought Prompting
System	<p>In source code, developers sometimes use placeholder/dummy DNS names instead of actual DNS names. For example, in the code snippet below, "www.example.com" is a placeholder/dummy DNS name.</p> <p>– Start of Code – mysqlconfig = { "host": "www.example.com", "user": "hamilton", "password": "poiou987", "db": "test" } – End of Code –</p> <p>On the other hand, in the code snippet below, "kraken.shore.mbari.org" is an actual DNS name.</p> <p>– Start of Code – export DATABASE_URL=postgis://everyone:guest@kraken.shore.mbari.org:5433/stoqs – End of Code –</p> <p>Given a code snippet containing a DNS name, your task is to determine whether the DNS name is a placeholder/dummy name. Output "YES" if the address is dummy else "NO".</p>
User	<p>Is the DNS name "{dns}" in the below code a placeholder/dummy DNS? Take the context of the given source code into consideration.</p> <p>{source_code}</p>