AssetHarvester: A Static Analysis Tool for Detecting Secret-Asset Pairs in Software Artifacts

Setu Kumar Basak*, K. Virgil English†, Ken Ogura‡, Vitesh Kambara§, Bradley Reaves¶ and Laurie Williams‖
North Carolina State University, USA
*sbasak4@ncsu.edu, †kvenglis@ncsu.edu, ‡kogura@ncsu.edu, §vkkambar@ncsu.edu, ¶bgreaves@ncsu.edu, ‖lawilli3@ncsu.edu

***Abstract*—GitGuardian monitored secrets exposure in public GitHub repositories and reported that developers leaked over 12 million secrets (database and other credentials) in 2023, indicating a 113% surge from 2021. Despite the availability of secret detection tools, developers ignore the tools' reported warnings because of false positives (25%-99%). However, each secret protects assets of different values accessible through asset identifiers (a DNS name and a public or private IP address). The asset information for a secret can aid developers in filtering false positives and prioritizing secret removal from the source code. However, existing secret detection tools do not provide the asset information, thus presenting difficulty to developers in filtering secrets only by looking at the secret value or finding the assets manually for each reported secret. *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.* We present 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 extracted from 188 public GitHub repositories 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.**

I. Introduction

In March 2024, GitGuardian reported a 113% surge in developers leaking over 12 million secrets in public GitHub repositories in 2023 compared to 2021 [1]. They 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 and API keys, are essential for integrating with external services such as customer databases and payment systems. However, developers keep hard-coded secrets in application packages and version control systems (VCS), exposing sensitive information to attackers [2], [3]. 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 [4].

```
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

Fig. 1: A secret can protect both real and non-sensitive assets, such as a public IP address (real) and localhost (non-sensitive).

At present, many open-source and proprietary secret detection tools, such as TruffleHog [5] and GGShield [6], are available to prevent leaking secrets. However, Basak et al. [7] 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" [8] 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 1a shows a customer 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 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 IV)
- **RQ2:** What performance can be achieved in detecting assets protected by secrets via static analysis in terms of precision, recall, and F1-score? (Section VI)

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. 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 [9]. 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 our paper is structured as follows: Section II 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 V and VI, followed by the implications of our work. We discuss the ethics and limitations of our study in Sections VIII and IX, respectively. We discuss the related work in Section X and conclude in Section XI.

```
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')
```

Fig. 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.

II. 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 [1] 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 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 [10], MySQL [11], SQLite [12], MongoDB [13], and SQL Server [14], according to the Stack Overflow Developer Survey 2023 [15]. 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.

III. AssetBench

To create a dataset of secret-asset pairs, we started with SecretBench [16], 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 [17] (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 [5] and Gitleaks [18]. The dataset contains 97,479 labeled plaintext secrets, manually labeled as true or false by two authors

TABLE I: 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%

of the SecretBench. In addition, the dataset provides additional metadata such as repository name, 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 [19]. 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 II) 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 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 [20] score of 0.96 between two authors, which indicates a "near perfect agreement" according to Landis and Koch's interpretation [21]. 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 I, 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 [22], 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 [23] and selected 490 secret-asset pairs. Next, we randomly selected 100 secret-asset pairs to avoid selection bias [24] 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-332204.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.

IV. Secret-Asset Co-location Patterns

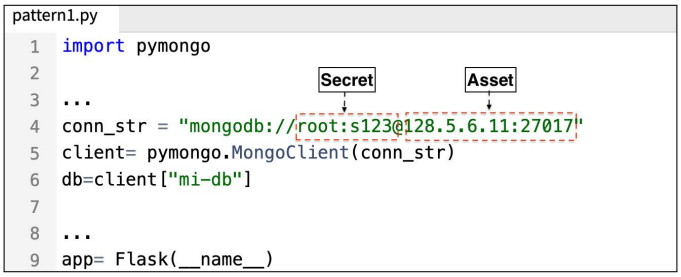
To answer RQ1, the first and second authors independently inspected a random sample of 100 secret-asset pairs from the AssetBench. Both authors observed the location pattern of a secret and the corresponding asset and identified four mutually exclusive secret-asset co-location patterns. We utilized the co-location patterns for programmatically identifying secret-asset pairs in the construction of AssetHarvester, as described in Section V. 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 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.

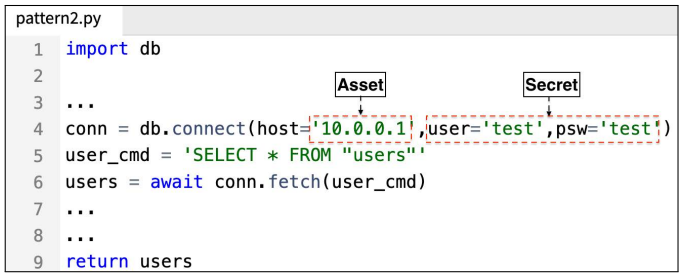
Pattern 2 (Separate Strings, Same Line, Same File) (20): The secret and the corresponding asset identifier can be present in the same line of the file but defined separately in multiple strings. For example, the database server address (`"10.0.0.1"`), the username (`"test"`), and the password (`"test"`)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

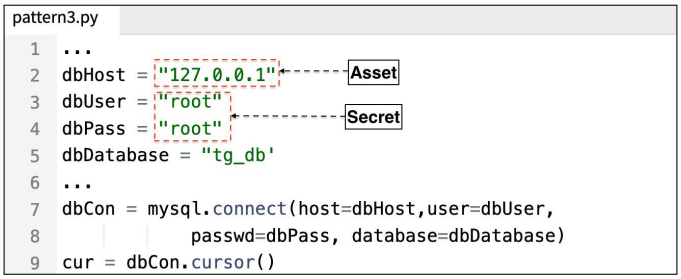
Pattern 4 (Separate Strings, Separate Lines, Separate Files) (7): The secret and the corresponding asset identifier can be present in separate files of the repository. For example, as shown in Figure 3d, the username (`"root"`) and password (`"123456"`) of the database are defined in lines 2 and 3 of



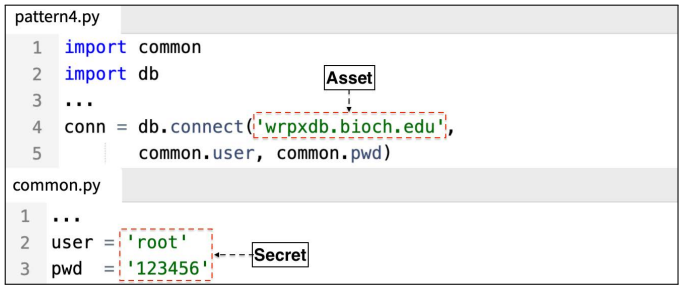
(a) Pattern 1 (Same String, Same Line, Same File)



(b) Pattern 2 (Separate Strings, Same Line, Same File)



(c) Pattern 3 (Separate Strings, Separate Lines, Same File)



(d) Pattern 4 (Separate Strings, Separate Lines, Separate Files)

Fig. 3: We identified four types of secret-asset co-location patterns in the source code.

`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 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.

V. AssetHarvester

We utilized the identified secret-asset co-location patterns (Section IV) 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 IV 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 corresponding secrets.

Step 1.1 Formulating Regex: In this step, we manually inspected the documentation for each database type and identified the database connection string format. Then, we categorized the connection string formats into three groups and formulated the regex for each group, as shown in Table II. We now discuss how we categorized the database connection string formats and formulated regex for each group.

Group 1: (MySQL, PostgreSQL & MongoDB): We observed that according to MySQL [25], PostgreSQL [26], and MongoDB [27] documentations, these three database types have similar connection string formats. The common format is *"[scheme://][user[:[password]]@]host[:port][/db]"*. The `scheme` refers to the transport protocol, `user:password` refers to the credentials, `host:port` refers to the server address, and `db` is the database name in the connection. We observed that only the scheme type differs in the three database connection strings. For example, MySQL uses `"mysql"` or `"mysqlx"` whereas PostgreSQL uses `"postgresql"` or `"postgres"` as the scheme types. Hence, we formulated a common regex with the different scheme types for Group 1.

Group 2: (ODBC & OLE-DB): The Open Database Connectivity (ODBC) [28] and the Object Linking and Embedding Database (OLE-DB) [29] are two standard APIs that provide support for accessing and interacting with different databases. ODBC and OLE-DB support the selected four databases in our study. We noticed that ODBC and OLE-DB have similar connection string formats consisting of key-value pairs separated by semicolons. For example, the connection string format for ODBC is *"Driver=Driver_Name; Server=address; Database=dbname; Uid=username; Pwd=password;"*. The key `Driver` refers to the ODBC database driver to be used such as "SQL Server". The key `Server` refers to the address of 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) [30] is a standard API that allows Java applications to interact with different databases. In

TABLE II: List of regexes categorized into three groups for identifying database connection string

	Type	Connection String Format	Example	Regex
Group 1	MySQL	[mysql|mysqlx|mysql+srv://][user[:[password]]@]host[:port][/db]	mysql://root:root@10.0.0.1:3306/test	(?P<dbms>mysql|mysqlx|mysql+srv|postgresql|postgres|mongodb|mongodb+srv):\/\/(?P<credentials>[^:@\s]*(?::[^@\s]*)?@)?(?P<server>[^\/\?\s`'\";]+)
	PostgreSQL	[postgresql|postgres://][user[:[password]]@]host[:port][/db]	postgresql://test:test@localhost/mydb	
	MongoDB	[mongodb|mongodb+srv://][user[:[password]]@]host[:port][/db]	mongodb://root:test@10.1.1.0:27017	
Group 2	ODBC	Driver={Driver_Name};Server=address;Database=dbname; Uid=username;Pwd=password;	Driver={SQL Server}; Server= 192.168.1.0;Database =test_db;Uid=sa; Pwd=sa	(?:(Provider|Driver)=[^;]*);[\s]*(?:(?:Data Source| Server)=(?P<server>[^;]+);)(?:(?:Initial Catalog| Database)=(?P<database>[^;]+);)?(?:(?:User Id| UID)=(?P<user>[^;]+);)?(?:(?:Password|PWD=)(?P<password>[^;]+);)?
	OLE-DB	Provider={Provider_Name};Data Source=address;Initial Catalog=dbname;User Id=username;Password=password;	Provider={SQL Server}; Data Source= 192.168.1.0; Initial Catalog=test_db;User Id=sa;Password=sa	
Group 3	JDBC	jdbc:[scheme://][user[:[password]]@]host[:port][/db]	jdbc:sqlserver://root:root@localhost:1433	(?P<dbms>mysql|postgresql|mongodb|sqlserver):[/]{2,3}(?P<credentials>[^:@\s]*(?::[^@\s]*)?@)?(?P<server>[^\?\s`'\";]+)\?user=(?P<user>[^\s&;<>]+)(?:&)?(?:\&?password=(?P<password>[^\s&;\]<>]+))?
		jdbc:[scheme://]host[:port][/db]?user=usr&password= pass	jdbc:sqlserver://localhost?user=root& password=root	

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` [31] are available that allow the connection of a database using the JDBC connection string from the Python source code. As shown in Table II, 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 [32] feature of regex. The capturing group allows us to capture a specific part of the match. For example, as shown in Table II, 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` [33] 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` [34], 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 IV, 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 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 [35] to detect the flow of secrets and assets into the database driver functions. Previous research [36] 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 [35], 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 [37]. From the search result, we identified 12 database drivers grouped in 7 categories, which are presented in Table III. We observed that in addition to identifying database drivers for the four databases, ODBC and JDBC, we identified two drivers, peewee [38] and SQLAlchemy [39] for the Object Relational Mapper (ORM) framework [40]. 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) [41], which act as the sinks for database username, password, and server address. The columns "Positional Argument" and "Keyword Argument" of Table III indicate which argument type each driver supports. We now discuss the two argument types as

TABLE III: List of Python database drivers with their supported arguments for secret-asset pairs

Category	Driver Name	Positional Argument	Keyword Argument
MySQL	aiomysql [42]		✓
	mysql-connector [43]		✓
	PyMySQL [44]	✓	✓
PostgreSQL	aiopg [45]	✓	✓
	asyncpg [46]	✓	✓
	psycopg2 [47]	✓	✓
MongoDB	pymongo [48]		✓
SQL Server	pymssql [49]		✓
ODBC	pyodbc [50]	✓	
JDBC	JayDeBeApi [31]	✓	
ORM	peewee [38]	✓	✓
	SQLAlchemy [39]		✓

```
1 import asyncpg
2
3 ...                                  Username        Database Name
4 conn = await asyncpg.connect('root', 'root','test_db', '10.0.0.1')
5 ...                                  Password                Host
```

Fig. 4: The database credentials and server address are passed in a specific order in the database driver function.

sinks for database secrets and assets.

1. Positional Argument: A positional argument [41] 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 4, the username, password, database name, and host address of the database are passed in a specific order in the connect 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 [41] (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 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 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.

Step 2.2 Identifying Secret-Asset Pairs Using CodeQL: For Data Flow analysis, we used Version `2.15.1` of CodeQL [51], an open-source source code analysis framework developed by GitHub. CodeQL treats source code as data and creates databases containing a hierarchical representation of the code, such as the abstract syntax tree, the data flow graph, and the control flow graph. Developers can query the database

```
1 ...
2 ...                                  Host        Database Name
3 conn = await.asyncpg.connect(host='10.0.0.1',database='test_db'
4                                  user='root',password='root')
5 ...                              Username        Password
```

(a) Keyword arguments passed as separate parameters

```
1 ...
2 dbconfig = {
3     "host": '10.0.0.1',
4     "database": 'test_db',
5     "user": 'root',
6     "password": 'root'
7 }
8 conn = await asyncpg.connect(**dbconfig)
```

(b) Keyword arguments defined in a dictionary

Fig. 5: The database credentials and server address are passed as keyword arguments in the database driver functions.

using QL [52], a query language optimized for efficiently analyzing databases representing software artifacts [52]. First, we queried the abstract syntax tree to identify the database driver sinks (functions and respective arguments). Next, we queried the data flow graph to find the sources that fall into the identified sinks and find the value of the secret-asset pairs for the corresponding sources using the abstract syntax tree. Finally, we queried the control flow graph to find the location (file path and line number) of secret-asset pairs. Since the database drivers are external libraries, we utilized the API Graphs [53] of CodeQL to compute the data flow graph. API Graphs are a uniform interface for referring to functions, classes, and methods defined in external libraries. In our study, we used the `semmle.python.ApiGraphs` module for accessing the external library functions.

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 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,

TABLE IV: 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%)

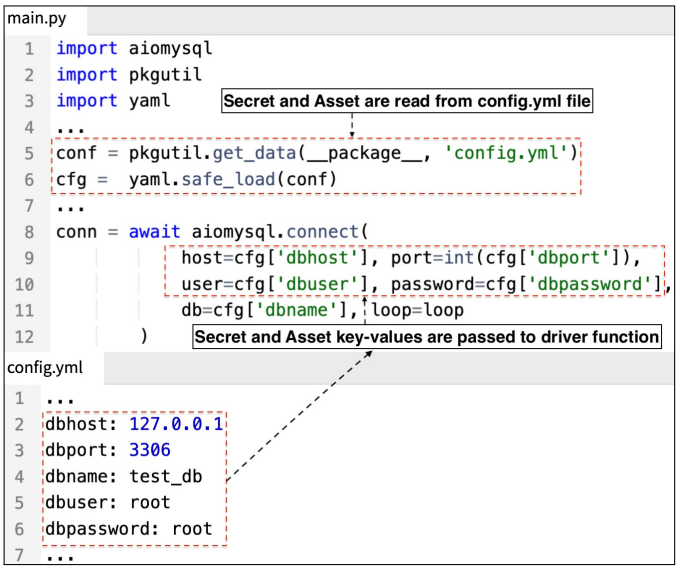


Fig. 6: The config.yml file contains the database secret-asset pair that is read in the main.py file. The secret-asset values are accessed by the key names and passed to the driver function.

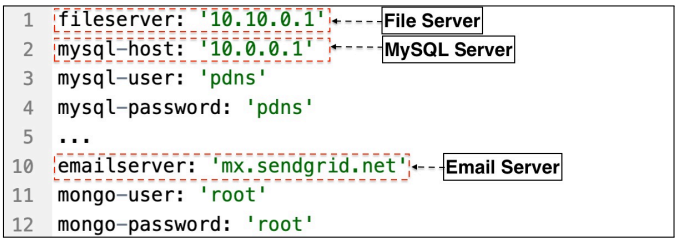


Fig. 7: Multiple or zero corresponding assets can be present in the neighboring lines of a secret.

we used the `PyYAML` [54], `json` [55] and `xmltodict` [56] 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 IV, 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 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 line 17 and line 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 [5] and Gitleaks [18]. The authors of SecretBench [16] have used the same two tools to curate the benchmark dataset as discussed in Section III. 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).

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` [57] 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 (`\b(?:\d{1,3}\.){3}\d{1,3}\b`) and DNS names (`\b[A-Za-z0-9][A-Za-z0-9-.]*\.\D{2,4}\b`) 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, 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, 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

TABLE V: 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

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 calculate the similarity score, we used Jaro-Winkler Similarity [58], 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` [59] package in Python to compute the similarity score and identify the secret-asset pairs.

VI. Performance of AssetHarvester

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

Precision, Recall and F1-Score: Table V 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 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 [60]. 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.

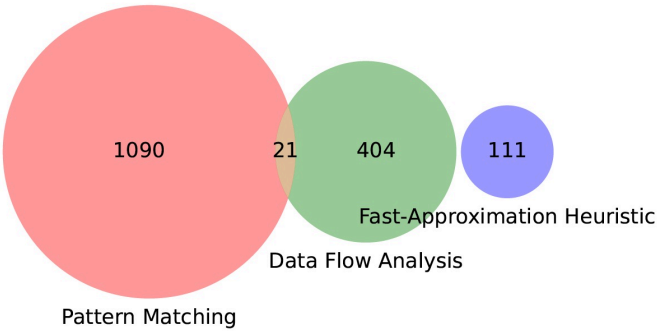


Fig. 8: The number of unique secret-asset pairs found by Pattern Matching, Data Flow Analysis, and Fast-Approximation Heuristic approaches.

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 V). For example, "URL" and "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 IV, 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. [7] 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 III. Table VI 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 [61]. 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 [62].

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 V). 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 V). In addition, we identified 86 secrets that are not present in SecretBench using data flow analysis. As discussed in Section III, 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.

TABLE VI: Comparison of AssetHarvester with 9 Baseline Secret Detection Tools on Secret Detection

Tool	Precision (TP, FP)	Recall (TP, FN)
git-secrets [63]	0.04 (1460, 65)	0.01 (31, 2083)
Gitleaks [18]	0.21 (220, 45)	0.02 (37, 2077)
Repo-supervisor [64]	0.31 (1270, 391)	0.17 (364, 1750)
TruffleHog [5]	0.19 (5666, 1068)	0.40 (851, 1263)
Whispers [65]	0.06 (2203, 147)	0.05 (112, 2002)
Commercial X	0.35 (7260, 2511)	0.28 (589, 1525)
ggshield [6]	0.49 (1972, 969)	0.16 (330, 1784)
GitHub-scanner [66]	0.00 (0, 0)	0.00 (0, 0)
Spectralops [67]	0.15 (574, 88)	0.29 (609, 1505)
AssetHarvester	0.92 (2298, 89)	0.90 (1892, 222)

Developer Survey: Since we leveraged a random sample of the dataset to construct AssetHarvester, 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. [16] 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 [16]. To mitigate bias, we selected a random sample of 15 repositories from 88,252 candidate repositories (excluding 818 repositories of SecretBench) after applying Criteria 1 and Criteria 2 as discussed in Section III. We identified 42 secret-asset pairs (18 unique secret-asset pairs) present in repositories' commit history using AssetHarvester. Then, we conducted a developer survey with the committer of secret-asset pairs to evaluate AssetHarvester's performance, and 18 responses were received. Fifteen respondents agreed with our identified secret-asset pairs, and three respondents termed the secret-asset pairs as false positives.

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 IV). 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 [16]. 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 con-

structed 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 VI).

VII. 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 a credential 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 VI). 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 [16]. 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 V). 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 [1]. 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 II), 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 `smtplib` [68] 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 V.

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.

VIII. Ethics and Data Protection

Since our dataset contains sensitive information, 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. 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. Additionally, we are notifying every developer in our dataset to remove the secret-asset pairs from their VCS.

IX. Threats to Validity

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

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. [16] 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 VI). Thus, SecretBench may have more missing secrets, impacting the results discussed in Section VI.

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.

X. Related Work

Previous studies [2], [69]–[73] 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. [2] found over 100K hard-coded secrets by studying a 13% snapshot of public GitHub repositories. Rahman et al. [70] investigated 5,232 Infrastructure as Code (IaC) scripts extracted from 293 open-source repositories and observed 7 "Security Smells". Among the 7 security smells, hard-coded secrets were found to be the most frequent, with 1,326 occurrences. Within GitHub Gists, which developers use for sharing code snippets, Rayhanur et al. [69] found 689 instances of hard-coded secrets by investigating 5,822 Python Gists in GitHub. These previous works indicate that hard-coded secrets have been leaking in various forms within software artifacts.

Researchers [73]–[75] have recommended that developers follow secure practices for secret management to avoid exposure of secrets in software artifacts. In 2022, Basak et al. [74] identified 24 developer and organization practices by conducting a grey literature review of Internet artifacts such as blog posts. To avoid the accidental commit of secrets, they suggested using VCS scan tools. In another study, Basak et al. [75] investigated the challenges developers face for checked-in secrets in Stack Exchange (SE) and the solutions SE users suggest to mitigate the challenge. They found that to avoid accidentally committing secrets, SE users also suggested using VCS scan tools. However, Basak et al. [7] compared 5 open-source and 4 proprietary VCS scan tools against SecretBench [16] and observed that tools output a lot of false positives. In addition, tools failed to detect all the secrets present in a repository. Recent research [76]–[79] has employed Machine Learning (ML) algorithms to reduce the false positives. However, among the 9 VCS scan tools investigated by Basak et al. [7], two tools (Commercial X (anonymized) and SpectralOps [67]) employed ML algorithms to detect secrets showed lower precision scores of 25% and 1%, respectively. Rayhanur et al. [80] conducted a developer survey in an XTech company (anonymized) and found that developers ignore secrets due to many secrets outputted by VCS scan tools and time pressure. However, if the information about the asset protected by the secret was provided, developers could have prioritized secret eradication. However, existing VCS scan tools do not provide the asset information for a secret. In this study, we concentrated our research efforts on identifying the assets protected by secrets to aid developers in prioritizing secrets removal efforts.

XI. 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. 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.

Acknowledgment

This work was supported by the National Science Foundation 2055554 grant.

References

[1] "The State of Secrets Sprawl 2024," https://www.gitguardian.com/state-of-secrets-sprawl-report-2024, [Online; accessed March 17, 2024].

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

[3] Cybernews Team, "Thousands of Android apps leak hard-coded secrets, research shows," https://cybernews.com/security/android-apps-leak-hardcoded-secrets, 2022, [Online; accessed March 12, 2024].

[4] M. Jackson, "Uber Breach 2022 – Everything You Need to Know," https://blog.gitguardian.com/uber-breach-2022, [Online; accessed March 10, 2024].

[5] "TruffleHog," https://github.com/trufflesecurity/truffleHog, [Online; accessed February 23, 2024].

[6] "GGShield," https://github.com/GitGuardian/ggshield, [Online; accessed March 13, 2024].

[7] 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.

[8] Hadjy, Paul, "What Is Alert Fatigue? 4 Ways to Mitigate It and Prevent Burnout," https://learn.g2.com/alert-fatigue, [Online; accessed March 12, 2024].

[9] "AssetHarvester Artifacts," https://github.com/setu1421/AssetHarvester, [Online; accessed November 20, 2024].

[10] "PostgreSQL," https://www.postgresql.org, [Online; accessed February 14, 2024].

[11] "MySQL," https://www.mysql.com, [Online; accessed March 14, 2024].

[12] "SQLite," https://www.sqlite.org, [Online; accessed February 14, 2024].

[13] "MongoDB," https://www.mongodb.com, [Online; accessed February 14, 2024].

[14] "Microsoft SQL Server," https://www.microsoft.com/en-us/sql-server/sql-server-2022, [Online; accessed February 14, 2024].

[15] "Stack Overflow Developer Survey, 2023," https://survey.stackoverflow.co/2023/#most-popular-technologies-database, [Online; accessed February 14, 2024].

[16] 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.

[17] "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 February 14, 2024].

[18] "Gitleaks," https://github.com/gitleaks/gitleaks, [Online; accessed February 18, 2024].

[19] "The State of Secrets Sprawl 2023," https://www.gitguardian.com/state-of-secrets-sprawl-report-2023, [Online; accessed July 17, 2024].

[20] 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/10.1177/001316446002000010

[21] 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

[22] Spencer EA, Brassey J, Mahtani K, "Recall Bias," https://www.catalogueofbiases.org/biases/recall-bias, [Online; accessed March 11, 2024].

[23] "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 March 11, 2024].

[24] Nunan D, Bankhead C, Aronson JK, "Selection Bias," https://catalogofbias.org/biases/selection-bias, [Online; accessed March 11, 2024].

[25] "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].

[26] "PostgreSQL Connection String," https://www.postgresql.org/docs/current/libpq-connect.html#LIBPQ-CONNSTRING, [Online; accessed February 16, 2024].

[27] "MongoDB Connection String," https://www.mongodb.com/docs/manual/reference/connection-string, [Online; accessed February 16, 2024].

[28] "Microsoft Open Database Connectivity," https://learn.microsoft.com/en-us/sql/odbc/microsoft-open-database-connectivity-odbc, [Online; accessed February 16, 2024].

[29] 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.

[30] "Java JDBC API," https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc, [Online; accessed February 16, 2024].

[31] "JayDeBeApi," https://pypi.org/project/JayDeBeApi, [Online; accessed February 16, 2024].

[32] "Named Capturing Group," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Regular_expressions/Named_capturing_group, [Online; accessed February 16, 2024].

[33] "re - Regular Expression Operations," https://docs.python.org/3/library/re.html, [Online; accessed February 16, 2024].

[34] "GitPython," https://github.com/gitpython-developers/GitPython, [Online; accessed February 16, 2024].

[35] U. Khedker, A. Sanyal, and B. Sathe, *Data flow analysis: theory and practice*. CRC Press, 2017.

[36] 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.

[37] 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

[38] "peewee," https://docs.peewee-orm.com/en/latest, [Online; accessed February 18, 2024].

[39] "SQLAlchemy," https://docs.sqlalchemy.org/en/20, [Online; accessed February 18, 2024].

[40] "Object Relational Mapping (ORM)," https://www.theserverside.com/definition/object-relational-mapping-ORM, [Online; accessed February 19, 2024].

[41] "Positional and Keyword Arguments," https://problemsolvingwithpython.com/07-Functions-and-Modules/07.07-Positional-and-Keyword-Arguments, [Online; accessed February 19, 2024].

[42] "aiomysql," https://aiomysql.readthedocs.io/en/stable, [Online; accessed February 18, 2024].

[43] "mysql-connector," https://dev.mysql.com/doc/connector-python/en, [Online; accessed February 18, 2024].

[44] "PyMySQL," https://pymysql.readthedocs.io/en/latest, [Online; accessed February 18, 2024].

[45] "aiopg," https://aiopg.readthedocs.io/en/stable, [Online; accessed February 18, 2024].

[46] "asyncpg," https://magicstack.github.io/asyncpg/current, [Online; accessed February 18, 2024].

[47] "psycopg2," https://pypi.org/project/psycopg2, [Online; accessed February 18, 2024].

[48] "pymongo," https://pymongo.readthedocs.io/en/stable, [Online; accessed March 4, 2024].

[49] "pymssql," https://www.pymssql.org, [Online; accessed March 2, 2024].

[50] "pyodbc," https://pypi.org/project/pyodbc, [Online; accessed February 18, 2024].

[51] "CodeQL," https://codeql.github.com, [Online; accessed March 4, 2024].

[52] 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.

[53] "Using API graphs in Python," https://codeql.github.com/docs/codeql-language-guides/using-api-graphs-in-python, [Online; accessed February 19, 2024].

[54] "PyYAML," https://pyyaml.org/wiki/PyYAMLDocumentation, [Online; accessed February 19, 2024].

[55] "json - JSON encoder and decoder," https://docs.python.org/3/library/json.html, [Online; accessed February 19, 2024].

[56] "xmltodict," https://pypi.org/project/xmltodict, [Online; accessed February 19, 2024].

[57] "linecache — Random access to text lines," https://docs.python.org/3/library/linecache.html, [Online; accessed February 20, 2024].

[58] W. E. Winkler, "String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage." 1990.

[59] "Python jellyfish package," https://pypi.org/project/jellyfish, [Online; accessed February 21, 2024].

[60] "CodeQL Hardcoded Credentials," https://github.com/github/codeql/blob/main/go/ql/src/Security/CWE-798/HardcodedCredentials.ql, [Online; accessed July 23, 2024].
[61] "GitHub Secret Scanning Patterns," https://docs.github.com/en/code-security/secret-scanning/secret-scanning-patterns, [Online; accessed July 23, 2024].
[62] "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].
[63] "git-secret," https://github.com/sobolevn/git-secret, [Online; accessed February 23, 2022].
[64] "Repo-supervisor," https://github.com/auth0/repo-supervisor, [Online; accessed July 23, 2024].
[65] "Whispers," https://github.com/Skyscanner/whispers, [Online; accessed July 13, 2024].
[66] "Github Secret Scanner," https://docs.github.com/en/code-security/secret-scanning, [Online; accessed July 23, 2024].
[67] "SpectralOps," https://spectralops.io, [Online; accessed March 3, 2024].
[68] "smtplib — SMTP protocol client," https://docs.python.org/3/library/smtplib.html, [Online; accessed March 11, 2024].
[69] 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.
[70] 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.
[71] 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)*. Boston, MA: USENIX Association, Aug. 2022, pp. 2747–2763. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/koishybayev
[72] A. Rahman and L. Williams, "Different kind of smells: Security smells in infrastructure as code scripts," *IEEE Security & Privacy*, vol. 19, no. 3, pp. 33–41, 2021.
[73] A. Krause, J. H. Klemmer, N. Huaman, D. Wermke, Y. Acar, and S. Fahl, "Pushed by accident: A {Mixed-Methods} study on strategies of handling secret information in source code repositories," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 2527–2544.
[74] S. K. Basak, L. Neil, B. Reaves, and L. Williams, "What are the practices for secret management in software artifacts?" in *2022 IEEE Secure Development Conference (SecDev)*, 2022, pp. 69–76.
[75] ——, "What challenges do developers face about checked-in secrets in software artifacts?" in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1635–1647.
[76] 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.
[77] 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.
[78] 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.
[79] 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.
[80] 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.