UNIVERSITÀ DI PISA

ICT Risk Assessment

Experimental evaluation of Trufflehog

Marco Galante

Carlo Bardazzi

A.Y. 2024-2025

Index

1. Overview

1.1 Trufflehog

TruffleHog is a robust and versatile tool designed for discovering, classifying, validating, and analyzing ***secrets***. In this context, "secrets" refer to sensitive credentials or keys that machines use to authenticate and communicate with other systems. These include API keys, database passwords, private encryption keys, and more. The tool aims to mitigate risks associated with exposed secrets by identifying them across various environments, including Git repositories, wikis, logs, and filesystems. While it is available as a free, open-source tool, its enterprise version includes advanced features for enhanced analysis and automation.

TruffleHog is designed to uncover secrets wherever they might reside. It supports scanning diverse environments, ensuring no credential is overlooked:

1. **Version Control Systems**:
   Developers sometimes unintentionally commit sensitive data like API keys or passwords into repositories. TruffleHog's integration with Git-based platforms helps identify such issues early.
2. **Chats and Wikis**:
   Internal collaboration platforms may include credentials shared inadvertently in discussions or documentation.
3. **Object Stores and Filesystems**:
   Both cloud storage (e.g., S3, GCS) and local files can hold secrets, particularly in configuration files or backup data.

1.1.1 Discovery

TruffleHog offers powerful scanning capabilities for identifying secrets across various data sources. It includes sub-commands tailored to each source, enabling targeted scans based on the type of environment being evaluated. Among the supported sources:

- **Git**: Examines repositories (local or remote) for secrets in commit history, branches, and diffs.
- **GitHub and GitLab**: Supports scanning public repositories.
- **Docker**: Inspects images for embedded secrets or sensitive configurations.
- **Amazon S3 and Google Cloud Storage (GCS)**: Analyses buckets for exposed secrets with manual configuration of access credentials.
- **Filesystem**: Searches local files and directories.

1.1.2 Classification

The classification phase uses predefined patterns and regular expressions to identify potential secrets. In the free version, this includes:

- **Regex Matching**: Scans for patterns such as API keys, encryption keys, or database credentials.
- **Known Secret Types**: Detects common secrets like:
  - API keys for AWS, Google Cloud, Azure, and others.
  - GitHub or GitLab personal access tokens.
  - SSL private keys and JWT tokens.

However, the classification in the free version is limited to pattern recognition. It does not offer:

- **Contextual Linking**: Associating secrets with specific users, services, or accounts.

- **Metadata Enrichment**: Additional details like ownership or permissions.

1.1.3 Validation

TruffleHog differentiates between **verified** and unverified secrets through a validation process. Verified secrets are those that, during the scan, are determined to be active or usable. This is achieved through automated checks that attempt to determine whether, for example, an API key is still valid or if a token has expired.

The main difference between the two is:

- **Verified Secrets** – These are active and usable at the time of the scan.
- **Unverified Secrets** – These might be expired, revoked, or invalid, but this cannot be confirmed without further manual or advanced checks.

In the free version we use, validation is limited to basic activity checks, such as verifying AWS or GitHub credentials, without interacting with external APIs to gather additional details. As a result, some secrets may be flagged as **unverified** because the tool cannot definitively determine whether they are still valid or have been revoked.

1.2 Project

1.2.1 Structure

The project is available in our repository and is organized into five directories, each serving a specific role in the analysis and evaluation workflow. Below is a detailed overview of the project's structure:

1. **TruffleOutput:** This directory contains the results generated by TruffleHog for the various datasets found in the *Data* folder. The files in this directory represent the raw output of the tool, which has subsequently been processed for analysis.
2. **checkSecretsResults:** This folder includes the results produced by the `checkSecrets.py` script. The files gathered here are critical for our analysis of TruffleHog, as they compare the detected secrets with those actually present in the datasets.
3. **Data:** This directory holds all the datasets used during the TruffleHog analysis. The datasets were created using dedicated scripts and include both valid and fake secrets. Additionally, this folder contains an Excel file that served as the basis for dataset generation:
   - *FakeSecrets1*: A dataset containing one fake secret for each detector listed in the Excel file.
   - *FakeSecrets10*: A dataset containing 10 fake secrets for each detector listed in the Excel file.
   - *Secrets1*: A dataset containing one valid secret for each detector listed in the Excel file.
   - Secrets10: A dataset containing 10 valid secrets for each detector listed in the Excel file.
   - *Secret Regular Expression FILTERED.xlsx*: An Excel file containing the regex used to generate secrets during the analysis.
   - *Secrets1.txt*: A text file containing all secrets from *Secrets1*.
   - *Secrets10.txt*: A text file containing all secrets from *Secrets10*.
4. **Plot:** This directory includes the charts generated by the `ConfusionMatrix.py` script. These charts are used to visualize and interpret TruffleHog's performance, providing a quantitative analysis of the results described in the *Results* chapter.

5. **Script**: This folder contains all the scripts used for data generation and analysis. Each script plays a specific role in the project's workflow:
    - *adjustXLSX.py*: Modifies and adapts the original Excel file for dataset creation.
    - *checkSecret.py*: Validates TruffleHog's results by comparing them with the expected detectors.
    - *confusionMatrix.py*: Generates confusion matrices to evaluate detection performance.
    - *fixDockerResults.py*: Resolves discrepancies in results obtained through Docker executions.
    - *generate_secrets.py*: Creates a dataset of valid secrets.
    - *generate_FAKE_secrets.py*: Generates datasets of fake secrets.
    - *subcommand.py:* Handles TruffleHog subcommands for automated execution.
    - *unicFile.py*: Merges secrets into a single file for additional testing.

1.2.2 Environment

The execution environment for TruffleHog was configured to ensure maximum control over files and the tool's functionality. For this reason, we opted for a local installation rather than using the official containers provided by the developers. Specifically, we installed **version 3.84.1**, and generated a local executable. This approach allowed us to:

- Gain direct access to the tool's files and internal configurations.
- Analyze the various detectors available in the open-source version.
- Examine and compare the regex patterns used by TruffleHog to identify secrets.

During the data generation phase, the regex patterns describing the structure of secrets detected by each TruffleHog detector were compared with those defined in the previously mentioned Excel file. This comparison allowed for refining the dataset creation process, enhancing the accuracy and reliability of the data used for testing.

1.2.3 Aim

The main objective of this project is to evaluate TruffleHog's performance in identifying valid and invalid secrets using datasets that also include noise elements.

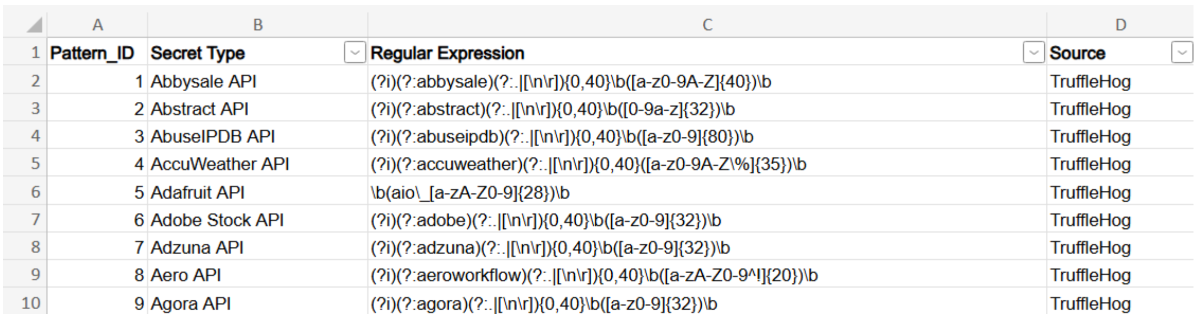
Key aspects of the analysis include:

- Analyzing the **tool's performance** using datasets of secrets generated based on the regex patterns employed by TruffleHog to identify sensitive patterns.
- Validating the **consistency** and effectiveness of TruffleHog's various **subcommands** by comparing results obtained through different scanning modes.
- Assessing TruffleHog's **reliability** on datasets with varying levels of **noise**, measuring its ability to distinguish between genuine secrets and false positives.

The project aims to provide an in-depth overview of TruffleHog's effectiveness in detecting secrets, identifying potential limitations, and proposing possible improvements to the detection process.

2. Data Generation

To generate the data for our analysis, we started by examining the ***"Secret Regular Expression.xlsx"*** file found in this repository. This file was used by the research group to create a dataset of fake secrets based on regex patterns collected for three different tools, including *TruffleHog*.



	A	B	C	D
1	**Pattern_ID**	**Secret Type**	**Regular Expression**	**Source**
2	1	Abbysale API	(?i)(?:abbysale)(?:.|[\n\r]){0,40}\b([a-z0-9A-Z]{40})\b	TruffleHog
3	2	Abstract API	(?i)(?:abstract)(?:.|[\n\r]){0,40}\b([0-9a-z]{32})\b	TruffleHog
4	3	AbuseIPDB API	(?i)(?:abuseipdb)(?:.|[\n\r]){0,40}\b([a-z0-9]{80})\b	TruffleHog
5	4	AccuWeather API	(?i)(?:accuweather)(?:.|[\n\r]){0,40}([a-z0-9A-Z\%]{35})\b	TruffleHog
6	5	Adafruit API	\b(aio_[a-zA-Z0-9]{28})\b	TruffleHog
7	6	Adobe Stock API	(?i)(?:adobe)(?:.|[\n\r]){0,40}\b([a-z0-9]{32})\b	TruffleHog
8	7	Adzuna API	(?i)(?:adzuna)(?:.|[\n\r]){0,40}\b([a-z0-9]{32})\b	TruffleHog
9	8	Aero API	(?i)(?:aeroworkflow)(?:.|[\n\r]){0,40}\b([a-zA-Z0-9^!]{20})\b	TruffleHog
10	9	Agora API	(?i)(?:agora)(?:.|[\n\r]){0,40}\b([a-z0-9]{32})\b	TruffleHog

Figure 1. Structure of Secret Regular Expression.xlsx

Starting with the Excel file containing **761 regex** patterns, we used the *`adjustXLSX.py`* script to:

1. **Remove regex** patterns for secrets intended to be recognized by tools other than TruffleHog (where the *Source* field is not *TruffleHog*).
2. **Remove regex** patterns for generating secrets for detectors unsupported by the free version of TruffleHog (those without a corresponding folder named after the detector in `pkg/detectors`).
3. **Add regex patterns for incomplete templates** to generate secrets for specific detectors, leveraging `adjustXLSX.py`. For example, the Amadeus detector requires a secret formatted as "API + Secret" to be recognized. Below, we highlight in red the field we added to correctly generate the secret for the Amadeus detector:

Amadeus API	(?i)(?:amadeus)(?:.|[\n\r]){0,40}\b([0-9A-Za-z]{32})\b
Amadeus SECRET	**\b([0-9A-Za-z]{16})\b**

To address **points 2** and **3**, we analyzed the contents of the **`pkg/detectors`** directory in version **3.84.1** of TruffleHog. Inside the `detectors` folder, there are subfolders named after each detector supported by the tool. Within each of these subfolders, we identified two files of particular interest:

1. **`DetectorName.go`**: This file contains the correct patterns that secrets must follow to be properly recognized by the detector. It was fundamental in adding missing regex patterns to the earlier mentioned Excel file.
2. **`DetectorName_test.go`**: This file contains valid secret patterns used to test TruffleHog's ability to recognize a secret for that specific detector. It served as a useful cross-check to verify the correctness of the data generated during the analysis of false negatives.

2.1 Script

Next, we developed **`generate_secrets.py`**, a script that processes each regex in the *Secret Regular Expression.xlsx* file to create secrets for each detector. The script is structured around three main functions:

1. **`generate_secret(regex)`**
2. **`expand_charset(charset)`**
3. **`generate_random_string(charset, length)`**

The *`generate_secret`* function returns a secret by identifying key components in the input regex that are fundamental to the structure of a secret:

- Prefix
- Suffix
- Separators
- Charsets

Generally, a secret for a specific detector is composed of:

Prefix + a random alphanumeric string of specified length separated by possible delimiters + Suffix

To generate the random string, the function utilizes *`generate_random_string`*, which creates a random string after determining:

- All valid characters allowed by the charset (identified in the regex) through *`expand_charset`*.
- The acceptable length of the random string to be generated.

Finally, once the secret is generated, it is added to an output file named **`DetectorName_secrets.txt`**.

2.2 Dataset

`generate_secrets.py` can produce an arbitrary number of secrets, saving each one in a file named **`DetectorName_secrets.txt`**. For our analysis, we created two datasets:

- **One secret per detector** – A total of 665 secrets available for detection.
- **Ten secrets per detector** – A total of 6650 generated secrets.

The first dataset was primarily used for debugging purposes, ensuring the script operated correctly and testing all subcommands offered by TruffleHog. The second dataset, with its larger volume of secrets, allowed for a more in-depth analysis, enabling the evaluation of the tool's behavior with a significantly higher amount of data.

2.3 Dataset Fake

We also developed a modified and intentionally **flawed** version of the *`generate_secrets.py`* script, named **`generate_FAKE_secrets.py`**. This variant is designed to generate **1** or **10 fake secrets** for each detector, resulting in a **total of 665 or 6650 secrets**. The script deliberately manipulates the data extracted from the regex patterns, keeping only the prefix and suffix unchanged. The outcome is a dataset composed of invalid secrets, characterized by:

- Charsets not permitted for the type of secret.
- **Non-compliant** string lengths.
- Introduction of noise in the generated files.

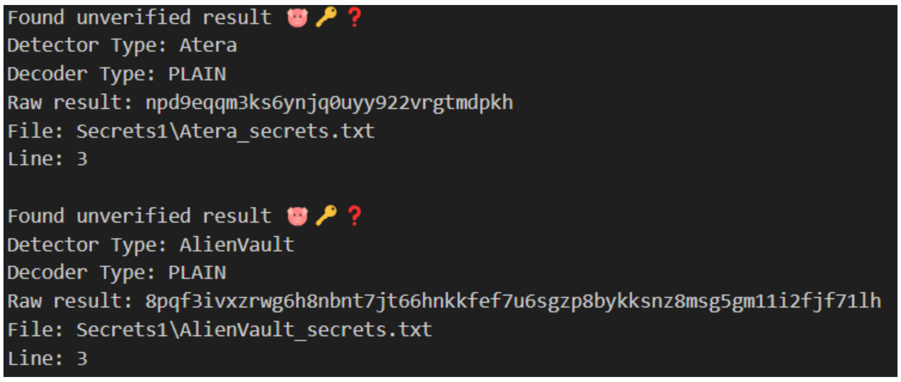
The purpose of this approach was to deepen the analysis of TruffleHog's behavior, evaluating its ability to distinguish between valid and fake secrets, and identifying any anomalies in its detection processes.

3. Analysis

3.1 checkSecrets.py

To evaluate the effectiveness of the TruffleHog tool, in addition to the tool's raw output (stored in the `TruffleOutput` folder in our repository) that indicates the number of detected secrets (both verified and unverified), we developed a script named **`checkSecrets.py`**. This script verifies whether the number of detectors TruffleHog should have used for each secret matches the detectors generated in the datasets. The script operates by comparing:

- **Expected detectors**: The detectors for which secrets were specifically generated in the datasets.
- **Detectors found**: The detectors TruffleHog actually used to identify the secrets.



```
Found unverified result 🐷🔑❓
Detector Type: Atera
Decoder Type: PLAIN
Raw result: npd9eqqm3ks6ynjq0uyy922vrgtmdpkh
File: Secrets1\Atera_secrets.txt
Line: 3

Found unverified result 🐷🔑❓
Detector Type: AlienVault
Decoder Type: PLAIN
Raw result: 8pqf3ivxzrwg6h8nbnt7jt66hnkkfef7u6sgzp8bykksnz8msg5gm11i2fjf71lh
File: Secrets1\AlienVault_secrets.txt
Line: 3
```

Figure 1. Examples of TruffleHog Output, contained in the TruffleOutput folder.

After collecting data from `TruffleOutput`, the script compares the detectors used by TruffleHog with those for which secrets were generated, for each dataset. This process helps identify:

- **True and false positives**: When TruffleHog correctly or incorrectly detects a secret.
- **True and false negatives**: When TruffleHog fails to detect a secret, either correctly or mistakenly.

Specifically, we define the following concepts in our analysis to evaluate the results:

- **True Positive (TP)**: A secret correctly detected by TruffleHog for the specific detector it was generated for. *Example*: In the *Secrets1* dataset, where one valid secret is generated for each detector, the ideal outcome would be **665 true positives** if all secrets are correctly detected.
- **False Positive (FP)**: A secret detected by TruffleHog that does not correspond to a valid secret generated for that detector. *Example*: In *Secrets1*, if each detector has only one valid secret and TruffleHog detects more than one secret for the same detector, the additional secrets are considered false positives.
- **True Negative (TN)**: A secret that is not detected by TruffleHog because no secret was generated for that detector. *Example*: In the *Secrets1* dataset, since each detector has one valid secret to recognize, **there should be no true negatives**.

- **False Negative (FN)**: A secret that TruffleHog fails to detect even though it was generated for a specific detector. *Example*: In *Secrets1*, if one of the secrets generated for the 665 detectors is not detected, it is classified as a **false negative**.

This categorization enables us to precisely evaluate TruffleHog's reliability and accuracy, highlighting not only its successes but also potential shortcomings in its detection process.

3.2 Subcomands.py

As described in the official repository, TruffleHog supports detecting secrets from various input sources. The main available options include:

- git
- github
- gitlab
- Docker
- s3
- filesystem (files and directories)
- gcs (Google Cloud Storage)
- postman
- jenkins

It was not possible to test all the available subcommands. Some, such as **gcs** and **s3**, require uploading the dataset to platforms with paid plans, while **jenkins** is not fully supported in the open-source version.

The **`subcommand.py`** script includes the commands for the five subcommands we were able to test. It automates the execution of TruffleHog on different sources, using the ***Secrets1*** dataset and generating an output file for each execution. The generated files follow the naming convention **`TruffleOutput/prova_DATASET_SUBCOMMANDNAME.txt`**, documenting the analysis results for each specific subcommand.

A combined analysis of all output files, conducted using the **`checkSecrets.py`** script, confirmed our initial hypothesis: the results are **consistent** and **unchanged** when using the same type of input files or directories, regardless of the subcommand used. For this reason:

- The *Secrets1* dataset was tested with all available subcommands.
- The *Secrets10*, *FakeSecrets1*, and *FakeSecrets10* datasets were analyzed exclusively using the **filesystem** subcommand.

Below is a brief summary of all the tested subcommands.

3.2.1 git

The **git** subcommand is used to analyze a remote Git repository. To test this subcommand, a test repository was created, and the dataset containing 665 secrets was uploaded to it. This command temporarily clones the repository and examines its history to search for secrets.

```
./trufflehog-3.84.1/trufflehog.exe git https://github.com/marckgt7/ProvaTruffle >
provaGitHub.txt
```

3.2.2 gitlab

The **git** subcommand also supports scanning a repository hosted on GitLab. To test this functionality, a test repository was created and uploaded to the platform for analysis.

```
./trufflehog-3.84.1/trufflehog.exe git
https://gitlab.com/provatrufflehog/ProvaTrufflehog
```

3.2.3 github

The github subcommand is similar to git but is specifically designed to scan a repository hosted on GitHub. To test this subcommand, the same test repository used for git was utilized. This command enables the analysis of the repository's content to detect secrets.

```
./trufflehog-3.84.1/trufflehog.exe github --
repo=https://github.com/marckgt7/ProvaTruffle > provaGitHub.txt
```

3.2.4 Docker

The docker subcommand allows for analyzing Docker images to detect secrets. To test this subcommand, a Dockerfile was created, and a Docker image was built from it and pushed to a local registry.

Dockerfile:

```
FROM python:3.10-slim
WORKDIR /app
COPY Secrets1 /app/Secrets1
RUN chmod -R 700 /app/Secrets1
CMD ["ls", "-l", "/app/Secrets"]
```

The commands for the generation of the immagine and the test with trugglehog :

```
docker build -t ict_project_image .
docker tag ict_project_image localhost:5000/ict_project_image
docker run -d -p 5000:5000 --restart=always --name registry registry:2
docker push localhost:5000/ict_project_image
./trufflehog-3.84.1/trufflehog.exe docker --image localhost:5000/ict_project_image >
provaDocker.txt
```

3.2.5 filesystem (files and directories)

The filesystem subcommand is used to scan local files and directories for secrets. It was employed to analyze the locally generated dataset.

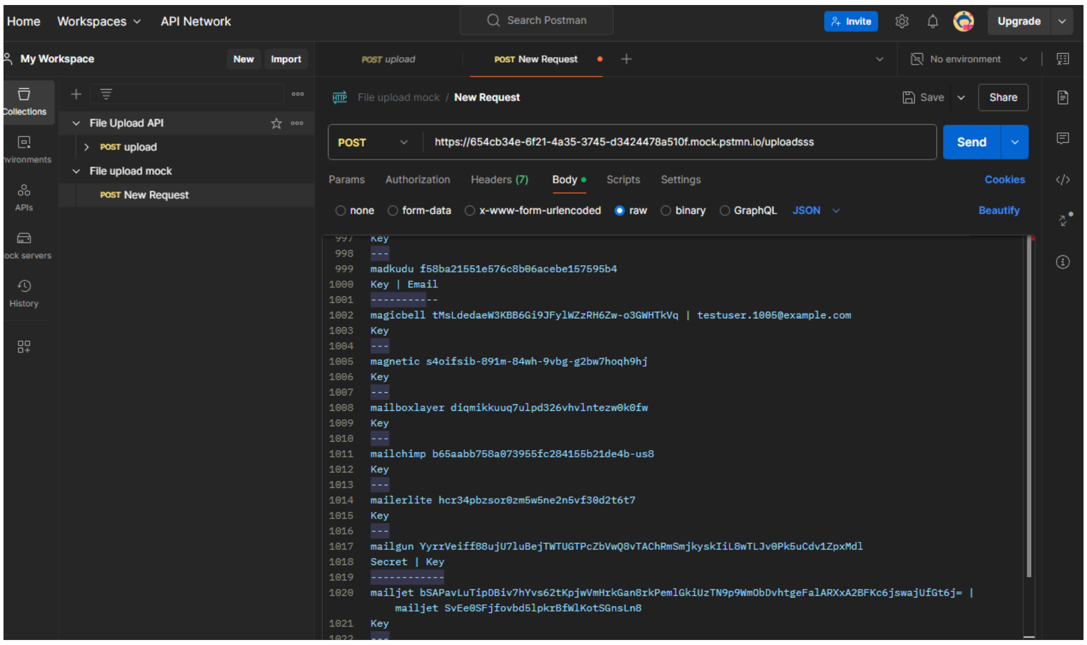
```
./trufflehog-3.84.1/trufflehog.exe filesystem generated_dataset/ > prova.txt
```

3.2.6 Postman

To test the **Postman** subcommand of TruffleHog, an analysis environment was simulated using a valid access token and a Postman workspace containing structured requests and datasets.

```
./trufflehog-3.84.1/trufflehog.exe postman --token=PMAK-6773b476df32b80001cecf50-
b2c92fe098a7cd0453643feba17c43c380ebb3 --workspace-id=e2d7eb70-568c-4b1f-8500-
87ddea278ca4 > prova.txt
```

Unlike the other subcommands, testing the **Postman** subcommand did not allow for separate analysis of individual input files. Instead, all files were combined into a single file, which was then included as the **raw** content in the **body** of the request.



```
POST https://654cb34e-6f21-4a35-3745-d3424478a510f.mock.pstmn.io/uploadsss
```

```
Key
---
madkudu f58ba21551e576c8b06acebe157595b4
Key | Email
------------
magicbell tMsLdedaeW3KBB6Gi9JFylWZzRH6Zw-o3GWHTkVq | testuser.1005@example.com
Key
---
magnetic s4oifsib-891m-84wh-9vbg-g2bw7hoqh9hj
Key
---
mailboxlayer diqmikkuuq7ulpd326vhvlntezw0k0fw
Key
---
mailchimp b65aabb758a073955fc284155b21de4b-us8
Key
---
mailerlite hcr34pbzsor0zm5w5ne2n5vf30d2t6t7
Key
---
mailgun YyrrVeiff88ujU7luBejTWTUGTPcZbVwQ8vTAChRmSmjkyskIiL8wTLJv0Pk5uCdv1ZpxMdl
Secret | Key
------------
mailjet bSAPavLuTipDBiv7hYvs62tKpjwVmHrkGan8rkPemlGkiUzTN9p9WmObDvhtgeFalARXxA2BFKc6jswajUfGt6j= |
    mailjet SvEe0SFjfovbd5lpkrBfWlKotSGnsLn8
Key
---
```

Figure 3. Screenshot of the postman workspace

3.2.7 Jenkins

```
./trufflehog-3.84.1/trufflehog.exe jenkins --url http://localhost:8080  --username
admin --password dd5acf1342fb4578a968f995e3cffc25
```

To test the **Jenkins** subcommand of TruffleHog, various configurations were executed to evaluate its ability to detect secrets. The tests were conducted on different Jenkins project setups, including predefined datasets such as *Secrets1* and *Secrets10*, as well as text strings containing secrets embedded in various job contexts.

In the first configuration, parameterized projects were created where secrets were distributed in separate files or combined into a single file. The files, in JSON, YAML, and TXT formats, were uploaded as inputs to build steps through configuration parameters. Although Jenkins successfully read the files during the build process, TruffleHog failed to detect any secrets.

Next, **string parameters** were used in Jenkins projects, with secrets directly added as values for build parameters. During job execution, TruffleHog was configured to analyze the provided parameters. However, even in this case, no secrets were detected, despite the values being visible in the job logs.

Secrets were then tested by being directly embedded in the ***Description*** field of Jenkins projects using plain text. TruffleHog was configured to analyze the project's metadata but failed to identify any secrets, revealing further limitations in its integration with Jenkins.

Another configuration involved linking Jenkins projects to GitHub repositories containing secret datasets. During the build, the repositories were cloned, and TruffleHog was executed as part of the build process. Yet again, despite the files being correctly downloaded and accessible, the tool did not detect any secrets.

Additionally, files containing secrets and secret strings were directly added to the Jenkins build environment as environment variables. Even though these files and variables were readable within the Jenkins workspace context, TruffleHog was unable to detect any secrets.

Finally, TruffleHog was run directly as a command in one of the job's **build steps**. During this compilation step, the tool was launched to analyze the files present in the Jenkins workspace. However, the results did not show any detected secrets.

Despite the variety of configurations tested, the **Jenkins** subcommand of TruffleHog failed to detect secrets in any of the scenarios. This suggests that the subcommand, in its open-source version, may not fully support data analysis within Jenkins jobs or may require more advanced and specific configurations to enable effective scanning.

3.3 confusionMatrix.py

This script automates the process of analyzing and evaluating the performance of secret detectors by working with files generated during testing and stored in the `./checkSecretsResults` directory. The goal is to identify and quantify **True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN)** for each detector, generating a confusion matrix and graphical representations that clearly and effectively visualize the results.

A key aspect of the process is extracting the expected number of secrets directly from the file name (e.g., *Secrets10* indicates 10 secrets). This number guides the analysis of results, which varies depending on the context. For files based on "fake" data, detected secrets are treated with specific rules: the secrets found are considered false positives, while missing detectors are classified as true negatives. In standard files, detected and missing secrets follow traditional classification logic, with TP, FP, FN, and TN calculated based on observed results.

The script automatically identifies missing detectors, classifying them as false negatives. For each analyzed file, key metrics such as **precision** and recall are calculated, providing a detailed assessment of each detector's detection capabilities. Accuracy is deliberately excluded from the metrics, as true negatives are always zero in our data: all identified secrets are valid. Similarly, the **F1-score** is not used, as its formula considers **true negatives**, making it irrelevant in our context.

4. Results

This section provides a detailed analysis of the results generated by the `checkSecrets.py` script, organized by each dataset examined. The collected metrics will be presented, comparing the expected results with the observed outcomes, and discrepancies will be analyzed in relation to the concepts of true and false positives/negatives, as described in the dedicated checksecrets.py paragraph.

4.1 ValidSecrets

4.1.1 Secrets1

For this dataset, **665 secrets** were generated, one for each of the **665 detectors**. The output of TruffleHog, processed by subcommands.py, produced the following results. These findings align with the observations from `subcommands.py`, confirming that TruffleHog's analysis remains consistent across most storage sources where the dataset is located, except for Postman. A comparison between Postman and Filesystem will be addressed in the next section:

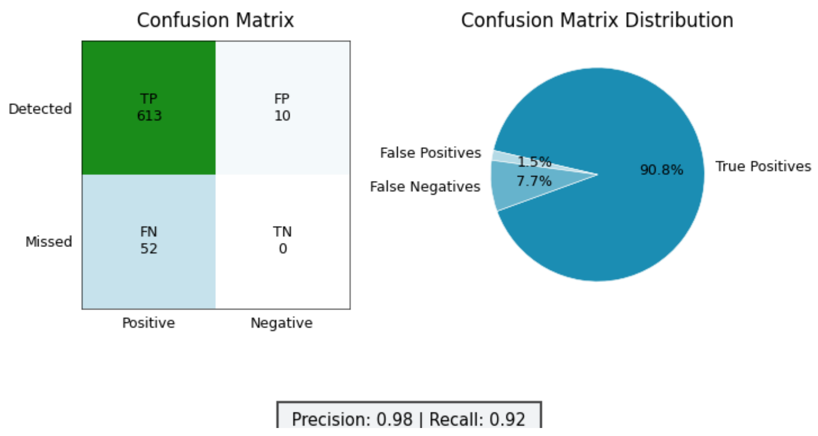


*Figure 2. Subcommand **filesystem***

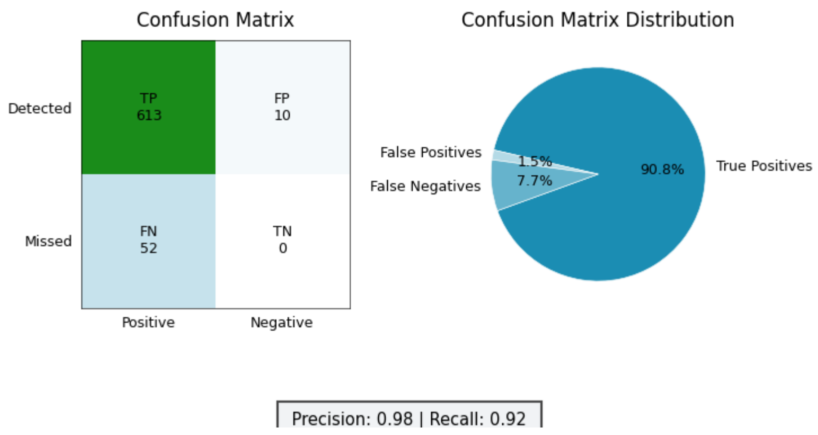


*Figure 3. Subcommand **docker***

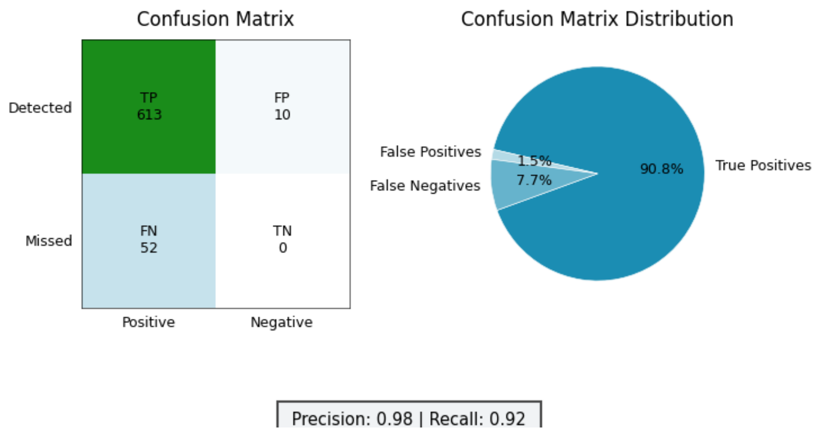


*Figure 4. Subcommand **git***

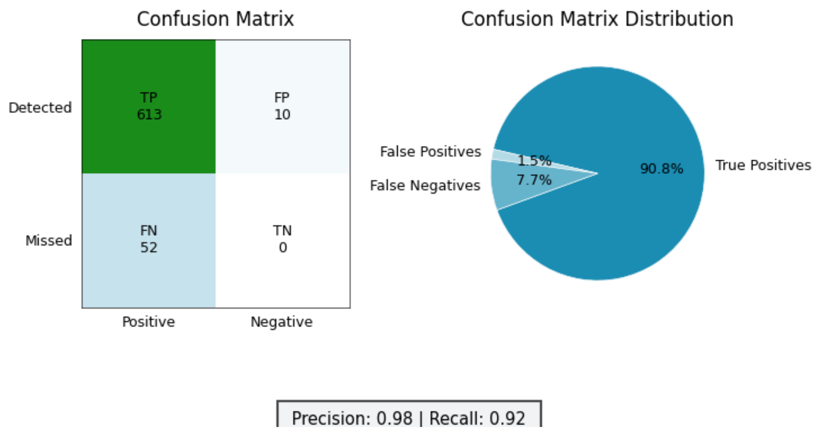


*Figure 5. Subcommand **github***

The analysis revealed significant and consistent results for each subcommand. Below is a detailed summary with observations on the outcomes:

- **613 secrets correctly detected** for the detectors for which specific secrets were generated.
- **10 incorrectly detected secrets**, **attributable to various causes**:
    - Detection does not fully adhere to the patterns defined in the `.go` files of TruffleHog's detectors.
    - Detectors unrelated to the generated secrets were erroneously activated, detecting unintended secrets.
- **52 secrets** present in the dataset but not detected by the tool.
- **0 false negatives**, a result consistent with expectations, as the dataset exclusively consists of valid secrets.

The accuracy of these results is supported by the correspondence between the sum of **detected secrets** (613) and **undetected** ones (52), totaling exactly the **665** secrets in the dataset. However, the **10 additional detections** require further analysis to identify the specific causes.

False positivies

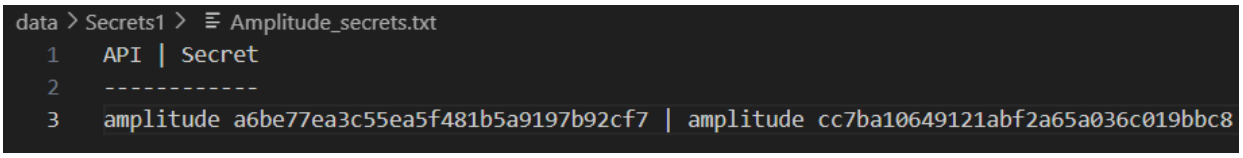
The files in the `checkSecretsResult` folder contain a comprehensive list of the detectors responsible for identifying unintended secrets. The primary causes of these false positives are:

1. **Partial detection of secrets**: TruffleHog identifies individual portions of the secret string as separate secrets.
2. **String combination**: TruffleHog combines different sections of the secret, treating them as distinct secrets.
3. **Unexpected detectors**: TruffleHog detects secrets using detectors for which no secrets were generated.

However, **option 3** can be excluded for the ***Secrets1*** and ***Secrets10*** datasets, as the number of unexpected detectors in both cases is **zero**.

Example

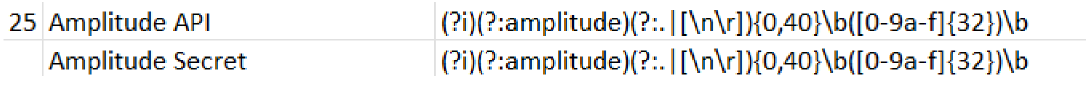
For the **Amplitude** detector, a secret was generated with the following structure:



```
API | Secret
------------
amplitude a6be77ea3c55ea5f481b5a9197b92cf7 | amplitude cc7ba10649121abf2a65a036c019bbc8
```

Figure 6. Amplitude_secrets.txt

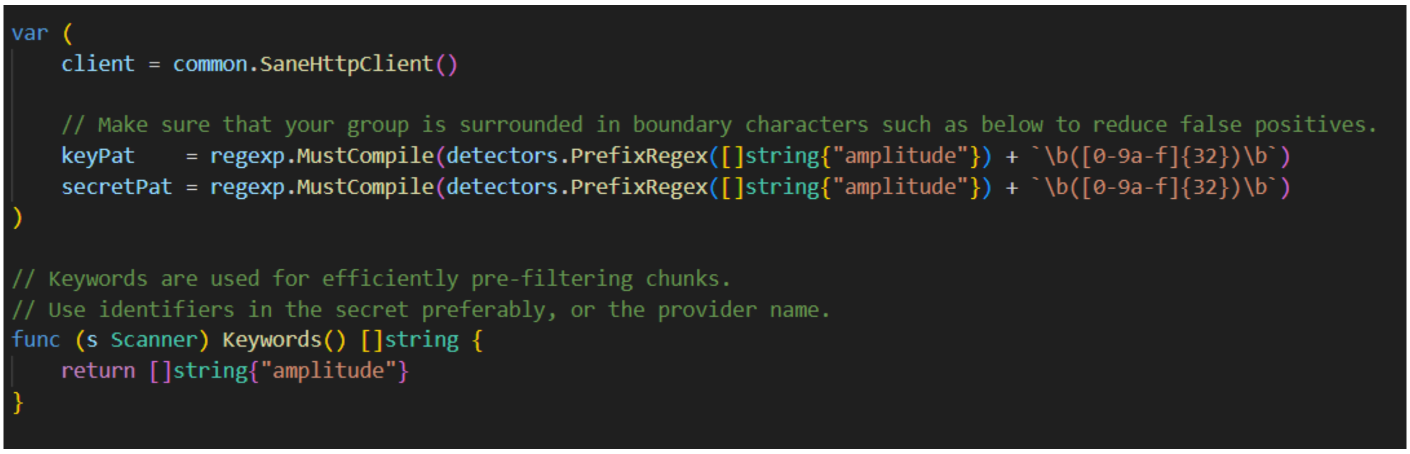
Given the regex of the Excel file:



25	Amplitude API	(?i)(?:amplitude)(?:.|[\n\r]){0,40}\b([0-9a-f]{32})\b
	Amplitude Secret	(?i)(?:amplitude)(?:.|[\n\r]){0,40}\b([0-9a-f]{32})\b

Figure 7. Secret Regular Expression FILTERED.xlsx

The file **amplitudeapikey.go** of the detector of Trufflehog defines the regex to generate *Amplitude* segrets:



```go
var (
	client = common.SaneHttpClient()

	// Make sure that your group is surrounded in boundary characters such as below to reduce false positives.
	keyPat    = regexp.MustCompile(detectors.PrefixRegex([]string{"amplitude"}) + `\b([0-9a-f]{32})\b`)
	secretPat = regexp.MustCompile(detectors.PrefixRegex([]string{"amplitude"}) + `\b([0-9a-f]{32})\b`)
)

// Keywords are used for efficiently pre-filtering chunks.
// Use identifiers in the secret preferably, or the provider name.
func (s Scanner) Keywords() []string {
	return []string{"amplitude"}
}
```

Figure 8. Amplitudeapikey.go

While the **amplitudeapikey_text.go** file defines the validation pattern for the strings keypat e secretpat, needed to identify an Amplitude secret:

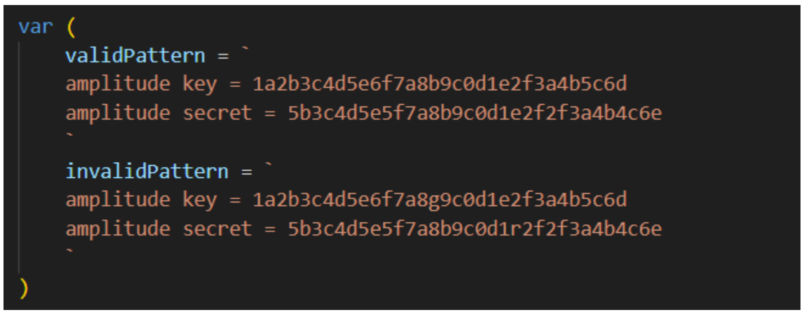
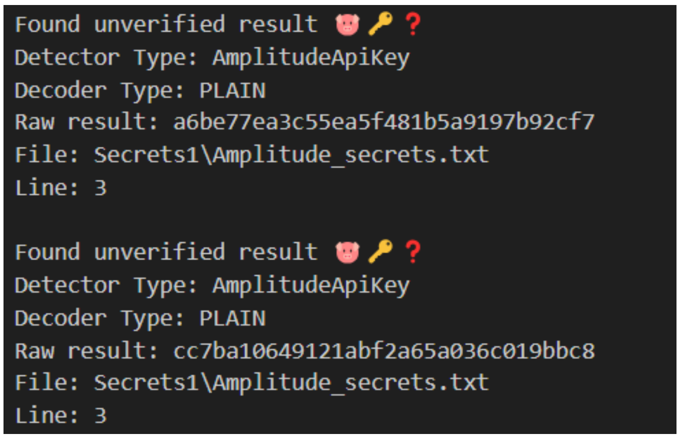

```
var (
	validPattern = `
	amplitude key = 1a2b3c4d5e6f7a8b9c0d1e2f3a4b5c6d
	amplitude secret = 5b3c4d5e5f7a8b9c0d1e2f2f3a4b4c6e
	`
	invalidPattern = `
	amplitude key = 1a2b3c4d5e6f7a8g9c0d1e2f3a4b5c6d
	amplitude secret = 5b3c4d5e5f7a8b9c0d1r2f2f3a4b4c6e
	`
)
```

Figure 9. Amplitudeapikey_text.go

However, in the ***Secrets1*** dataset, two secrets were detected instead of one. Specifically, the output of TruffleHog identified:


```
Found unverified result 🐷🔑❓
Detector Type: AmplitudeApiKey
Decoder Type: PLAIN
Raw result: a6be77ea3c55ea5f481b5a9197b92cf7
File: Secrets1\Amplitude_secrets.txt
Line: 3

Found unverified result 🐷🔑❓
Detector Type: AmplitudeApiKey
Decoder Type: PLAIN
Raw result: cc7ba10649121abf2a65a036c019bbc8
File: Secrets1\Amplitude_secrets.txt
Line: 3
```

Figure 10. Amplitudeapikey TruffleHog's output

In reality, only the string **`cc7ba10649121abf2a65a036c019bbc8`** should have been recognized as a **secret**, while the other string is merely a structural part of the secret!

This tendency to generate false positives was systematically observed with other detectors as well, suggesting that TruffleHog might **automatically combine** all possible substrings, thereby increasing the number of incorrect detections.

False negatives

For the **52 false negatives**, we conducted a thorough verification to identify potential issues in the generation of the secrets.

- Each undetected secret was analyzed to confirm that its structure conformed to the expected patterns.
- The results from **`checkSecrets`** were saved in **`checkSecretsResults`**, including a list of missing detectors.
- Each secret was compared against the regex in the `.go` file of the corresponding detector within the **TruffleHog** repository.

An interesting finding was that the valid test patterns in the `_test.go` files of the missing detectors were not detected. Specifically, when test files were created with secrets suggested by the valid patterns, none of these were recognized by the tool. This suggests that, although the generated secrets are formally correct, the tool struggles to recognize secrets for these detectors.

This allows us to confirm that the **52 undetected secrets** are genuine false negatives, which have a direct impact on the performance metrics.

Evaluation metrics

Despite the presence of some false positives, the evaluation metrics for the tool are very satisfactory, highlighting its high effectiveness in detecting secrets:

- **Precision**: 0.98
- **Recall**: 0.92

These values emphasize TruffleHog's robustness in secret detection, with high precision indicating a low number of false positives. The balance between recall and precision ensures that few valid secrets are overlooked, while the number of incorrect detections remains low, confirming the tool's overall reliability.

4.1.2 Secrets1.txt

As previously mentioned, for the analysis using the **Postman** subcommand, all inputs were consolidated into a single file (*Secrets1.txt*) and provided to TruffleHog. To ensure a fair comparison with the **Filesystem** subcommand, the latter was also tested using the same unified file.

The results show that the performance of the two subcommands is nearly identical, consistent with previous comparisons between Filesystem and other subcommands. Consequently, the analysis of subsequent datasets will be conducted exclusively using the **Filesystem** subcommand.

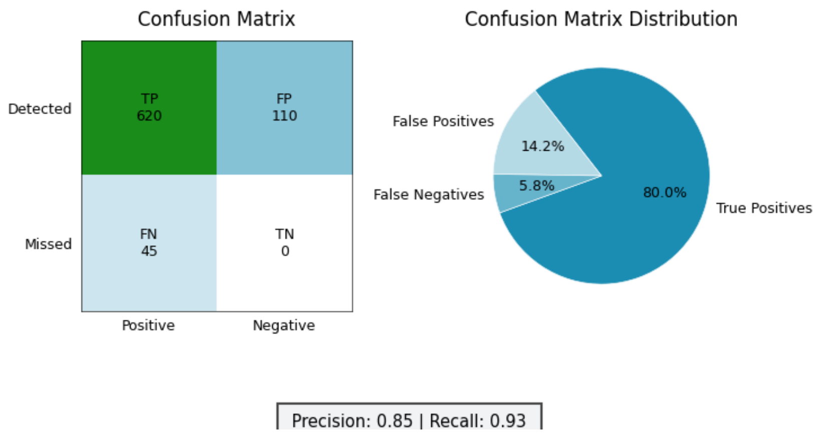


*Figure 11. Subcommand **postman***

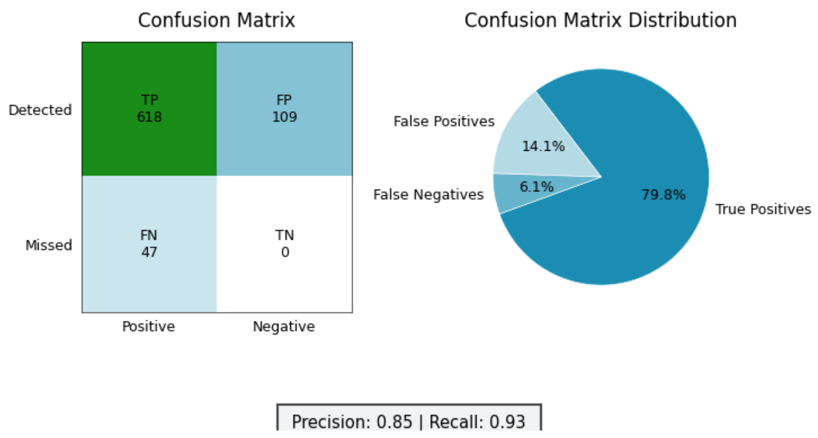


*Figure 12. Subcommand **filesystem***

Despite a slightly lower number of missing secrets compared to previous tests (**45** with Postman and **47** with Filesystem, versus **52** detected by the Filesystem subcommand with separate files), the overall performance of TruffleHog has significantly worsened. In particular, there was a notable increase in false positives: **109** with Filesystem and **110** with Postman, compared to only **10** when using separate files for each detector.

This degradation is attributed to consolidating all secrets into a single file, which leads TruffleHog to interpret them as potential detections for the same detector. The metrics for both subcommands reflect this impact:

- **Precision**: 0.85
- **Recall**: 0.93

These results highlight the importance of separating secrets into individual files to improve detection precision. The coexistence of multiple secrets in a single file increases the risk of false positives, underscoring the need for further optimization in TruffleHog's detection algorithms. The presence of numerous secrets in one file appears to confuse the tool, causing it to associate unrelated secrets with the same detector as valid detections. Consequently, **segmenting** files by detector remains a crucial strategy to optimize performance and reduce errors.

4.1.3 Secrets10

For the Secrets10 dataset, we generated a total of **6650 secrets** to detect, with **10 segrets for each detector**. Here we shows the obtained results:

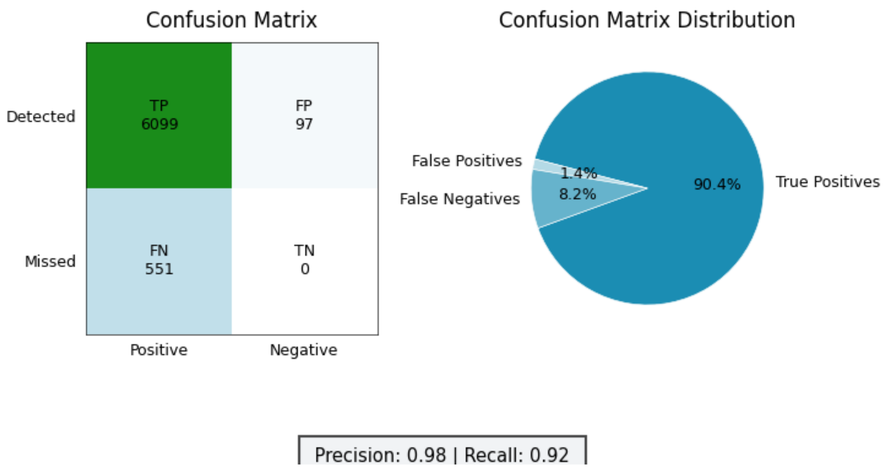


*Figure 134. Subcommand **filesystem***

The tool's performance was impacted by the increased number of secrets in the dataset. However, the results remained consistent with those observed for *Secrets1*:

- **6099 secrets** were correctly detected by the corresponding detectors for which secrets were generated.
- **97 secrets** were erroneously detected, for the same reasons highlighted in the analysis of *Secrets1*.
- **551 secrets** present in the dataset were not detected.
- **0 false negatives** were recorded, as expected, since the dataset consists exclusively of valid secrets.

The analysis of the *Secrets10* results does not show significant differences compared to *Secrets1*, but some new insights, both **positive** and **negative**, emerged regarding false negatives:

- The number of **missing detectors** decreased to **47**, compared to **52** in *Secrets1*.
- Although there are 47 missing detectors, the expected number of false negatives should have been **470**. However, a total of **551 false negatives** were recorded, indicating an **additional 81 false negatives**. This suggests a **decline in secret detection for some detectors**, which failed to detect all 10 expected secrets. Among these detectors are the 5 that are no longer missing:
  - **bombbomb**: 8 secrets detected
  - **contentful**: 1 secret detected
  - **graphcms**: 9 secrets detected
  - **paperform**: 9 secrets detected
  - **storychief**: 9 secrets detected

The metrics collected still indicate excellent detection capabilities by the tool:

- **Precision**: 0.98
- **Recall**: 0.92

4.1.4 Secrets10.txt

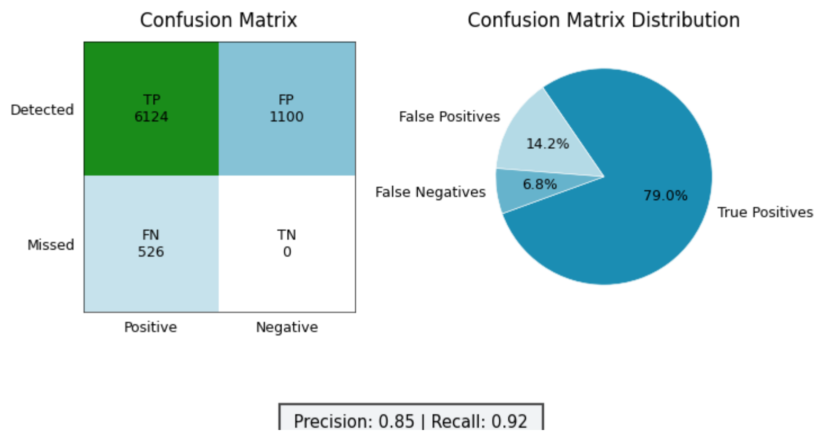


Figure 145. subcomand ***filesystem***

Analogamente a quanto osservato con Secrets1.txt, le performance del tool peggiorano significativamente quando TruffleHog analizza un unico file contenente **6650** segreti (in questo caso), rispetto a **665 file** con **10 segreti ciascuno**.

In questo scenario, i falsi negativi diminuiscono leggermente da **551 a 526**, ma si verifica un'**esplosione di falsi positivi**, che passano da **97 a 1100**. Questo drastico aumento compromette gravemente le performance del tool, persino rispetto ai risultati ottenuti con **Secrets1**:

- **Precision:** 0.85
- **Recall:** 0.92

Similarly to what was observed with Secrets1.txt, the tool's performance deteriorates significantly when TruffleHog analyzes a single file containing **6650** secrets (as in this case) compared to **665 files** with **10 secrets each**.

In this scenario the **False negatives** decrease slightly from **551 to 526.** However, there is a massive increase in **false positives**, rising from **97 to 1100.** This drastic increase severely impacts the tool's performance, even compared to the results obtained with **Secrets1**:

- **Precision**: 0.85
- **Recall**: 0.92

4.2 FakeSecrets

Given the absence of **true negative secrets**, we conducted a specific analysis to evaluate TruffleHog's effectiveness when faced with **structurally incorrect** secrets containing added noise. The goal was to understand the tool's susceptibility to false positives and assess its robustness.

For this analysis, we developed the **`generate_FAKE_secrets.py`** script, derived from `generate_secrets.py`. The key modification involves intentionally altering the generation of the random string that forms the secret. This alteration includes changes to the **allowable characters** and **string length** while keeping the prefix, recognized by TruffleHog, unchanged. This approach aims to trick the tool into detecting invalid secrets as valid.

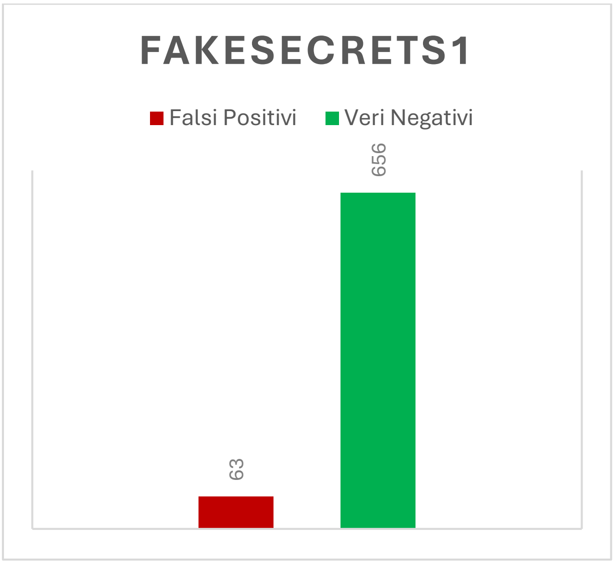
Additionally, to increase the complexity of the analysis and simulate realistic scenarios, noise was introduced between the components of the generated secrets. This method allows us to test the detector's resilience under conditions more akin to real-world contexts.

Following the methodology used in testing valid secrets, we initially generated one fake secret per detector, then increased it to ten secrets. Given the nature of the test—focused exclusively on true negatives and false positives—the analysis of TruffleHog was conducted using **specificity** and the **False Positive Rate (FPR)** as evaluation metrics.

The results from unique test files were not included in the following analysis, as no secrets were detected, resulting in 100% specificity. While noteworthy, this aspect does not significantly contribute to the objectives of our study.

4.2.2 FakeSecrets1

The detected secrets were classified as **false positives**, while the undetected ones were classified as **true negatives**. The results, while very good, were not outstanding. TruffleHog demonstrated a solid ability to identify invalid secrets but generated a notable number of false positives:



$$Specificity = \frac{656}{656 + 63} = 0.912$$

$$FPR = \frac{63}{656 + 63} = 0.088$$

TruffleHog achieved a specificity of **91.2%**, demonstrating a decent ability to correctly identify invalid secrets. However, the detection of **63 false positives** corresponds to a **false positive rate (FPR) of 8.8%**, which reduces its reliability in distinguishing between valid and fake secrets.

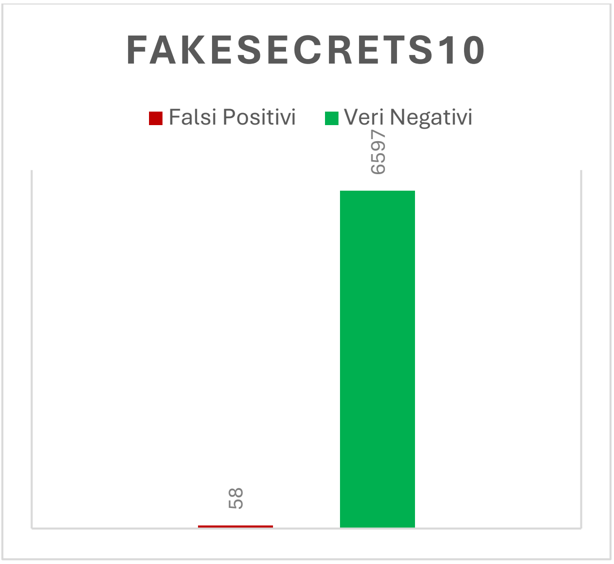
Additionally, data from `checkSecretsResults` reveals that these **63 false positives** were detected by only 9 detectors:

- **Aero**: 1 secret
- **caflou**: 8 secrets
- **currents**: 8 secrets
- **d7network**: 5 secrets
- **formbucket**: 2 secrets
- **hereapi**: 9 secrets
- **lexigram**: 9 secrets
- **nethunt**: 15 secrets
- **simplynoted**: 6 secrets

These results highlight that some detectors are more prone to false positives than others. In particular, **nethunt**, **hereapi**, and **lexigram** accounted for the highest number of false detections, significantly contributing to the overall false positive rate.

4.2.3 FakeSecrets10

The situation gets better increasing the number of secrets to detect:



$$Specificity = \frac{6597}{6597 + 58} = 0.99$$

$$FPR = \frac{58}{6597 + 58} = 0.008$$

With a specificity of **99%**, TruffleHog demonstrated a strong ability to correctly identify invalid secrets, achieving a **false positive rate of just 0.8%**. Compared to the *FakeSecrets1* test, the tool showed significant improvement, reflecting increased robustness when analyzing a larger number of secrets.

Additionally, the **58 false positives** detected were attributed to **8 of the 9 detectors** previously identified:

- **caflou**: 7 secrets
- **currents**: 7 secrets
- **d7network**: 5 secrets
- **formbucket**: 2 secrets
- **hereapi**: 8 secrets
- **lexigram**: 8 secrets
- **nethunt**: 15 secrets
- **simplynoted**: 6 secrets

The consistent presence of these detectors in false positives across both the *FakeSecrets1* and *FakeSecrets10* tests suggests a specific vulnerability. This could indicate that these detectors are more likely to be fooled by invalid secrets, likely due to:

- **Overly permissive detection patterns** – The detectors may classify non-secrets as valid due to overly broad recognition criteria.
- **Excessive sensitivity to prefixes** – The presence of valid prefixes appears to prompt detectors to classify strings as secrets, even when the rest of the string does not conform to the correct format.

4.3 Verified Secrets

Finally, we provide a summary of all the secrets that TruffleHog **verified**, considering them **active** and **currently available**:

	Verified	Unverified Secrets	Detected Secrets
Secrets1	2	621	623
Secrets1.txt	2	728	730
Secrets10	20	6176	6196
Secrets10.txt	20	7204	7224
FakeSecrets1	0	63	63
FakeSecrets10	0	58	58

Specifically, **all** the validated secrets came from **two** specific detectors:

- **IPQuality**
- **Squareup**

During the creation of the fake secrets datasets, we did not expect any of them to be verified, given their simulated nature. This result leads to two key considerations:

1. The **validation process** for the IPQuality and Squareup detectors could be **improved**.
2. The generation of a large number of API keys for these detectors likely produced random strings that were not unique. As a result, some of these strings were mistakenly recognized as valid. This suggests that the **regex patterns used by these detectors may need an updates** to target more complex strings in terms of charset and length.

Finally, given the limitations of the free version of TruffleHog, adopting the **enterprise version** could provide broader coverage and more in-depth analysis. This would help identify similar cases, ensuring more accurate secret management and reducing the risks associated with exposed credentials.

5. Conclusions

The analysis conducted on TruffleHog 3.84.1 highlighted its robust capability in detecting secrets, with overall satisfactory results, particularly in distinguishing between valid and invalid secrets. However, the experiment also revealed some critical aspects that deserve attention to further optimize the tool's performance.

Strengths:

- **High precision, recall, and specificity values:** The observed metrics confirm TruffleHog as a reliable tool for identifying exposed secrets.
- **Versatility in analysis:** The tool demonstrated its ability to detect secrets across various platforms and environments, ensuring broad coverage in both development and production contexts.

Limitations Identified:

- **False positives:** TruffleHog tends to generate false positives, especially when secrets are aggregated in a single file. This phenomenon was observed predominantly in tests using unified datasets.
- **False negatives:** Some detectors failed to identify secrets, including those provided as examples by the tool itself. This highlights a vulnerability that necessitates a review or enhancement of the associated regex patterns.
- **Difficulty with counterfeit secrets:** Tests involving datasets containing fake secrets showed that TruffleHog can be deceived by invalid secrets that only partially match detection regex patterns.

Improvement Suggestions:

- **Regex optimization:** Reviewing detection patterns, particularly focusing on detectors prone to false positives, could significantly enhance the tool's overall accuracy.
- **Advanced validation and machine learning:** In real-world scenarios, minimizing false positives is essential to avoid excessive irrelevant alerts. Incorporating multi-layered validation mechanisms or adopting machine learning models could substantially improve the tool's discrimination capability.
- **More in-depth structural analysis:** The reliance on patterns and keywords makes TruffleHog vulnerable to counterfeit secrets. Enhancing detection logic through a more detailed analysis of the secret's overall structure could mitigate this issue.

It is worth noting that the limitations and improvement proposals outlined here pertain exclusively to the free version of TruffleHog. The enterprise version may not only address additional issues but also reduce false positives and enhance detection capabilities, making TruffleHog more effective and reliable than the results achieved.

Despite these considerations, TruffleHog remains a valuable tool for safeguarding credentials in development and deployment processes. Adopting the enterprise version or implementing the suggested improvements could further enhance its effectiveness, helping to strengthen IT infrastructure security and prevent the accidental disclosure of secrets.