Hey, Your Secrets Leaked! Detecting and Characterizing Secret Leakage in the Wild

Jiawei Zhou*†§, Zidong Zhang†§, Lingyun Ying†✉, Huajun Chai†, Jiuxin Cao*✉, Haixin Duan‡
*Southeast University, Email: jiawei_zhou@seu.edu.cn, jx.cao@seu.edu.cn
†QI-ANXIN Technology Research Institute, Email: {zhangzidong, yinglingyun, chaihuajun}@qianxin.com
‡Quancheng Lab; Tsinghua University; Tsinghua University-QI-ANXIN Group JCNS, Email: duanhx@tsinghua.edu.cn

***Abstract*—Secrets, whether structured like API keys or unstructured like passwords, are essential for securing applications and services. However, the growing use of open-source projects and rapid development cycles has amplified the risk of secret leakage. Current detection tools suffer from high false positive rates and low recall due to simplistic methods like regular expressions and entropy checks, often missing unstructured secrets or mislabeling non-sensitive data.**

In this paper, we introduce KEYSENTINEL, an advanced automated secret detection tool that addresses these limitations through machine learning, semantic analysis, and prefix matching. To evaluate KEYSENTINEL, we created the first cross-platform benchmark with 11,826 labeled secrets in 1,806,530 files across GitHub, PyPI, and WeChat. We compare KEYSENTINEL with six currently available tools. The results show KEYSENTINEL achieves state-of-the-art performance, with precision (91.18%), recall (81.71%), and an F1 score (0.86), surpassing industry-standard tools and significantly reducing false positives. It also outperforms large language models like GPT-4 and o1 in accuracy and cost-effectiveness.

Besides, we conduct a large-scale measurement study, analyzing 80,330,098 files from GitHub, PyPI, and WeChat. We found that up to 30% of projects are at risk of secret leaks. Furthermore, we also scan the codebase of an IT company to assess real-world secret leakage risks. Our findings underscore the pervasive nature of secret leaks and highlight the urgent need for enhanced secret management practices across platforms.

1. Introduction

In modern software development pipelines, secrets (e.g., API keys, tokens, credentials, and passwords) are employed for access control and authentication across various services. Typically, an application may require dozens or even hundreds of these secrets. The large volume of secrets requires stringent safeguards and secure usage to prevent unauthorized disclosure. However, the widespread use of cloud services and rapid development cycles have significantly heightened the risk of secret leakage, as developers commonly hard-code secrets in files. For instance, Symantec [28] reported that 1,859 popular iOS and Android apps contained hard-coded AWS credentials, with 77% of the credentials being valid access tokens.

Similarly, Legit Security [9] reported that an average of 12 secrets were submitted per 100 repositories each week. Additionally, emerging artificial intelligence (AI) applications also face secret leakage issues. Permiso.io [26] warned that exposed API keys in hosted AI models could create security vulnerabilities, leading to unauthorized access and exploitation.

Thus, secret leakage has become a critical security concern. John Shier [33] reported that compromised secrets were the primary cause of cyber attacks in early 2023, accounting for 50% of root causes. More seriously, the exposure of these secrets usually results in data breaches and financial losses. For example, the Internet Archive [27] experienced a breach due to a publicly accessible GitLab authentication token, which remained exposed for nearly two years and allowed hackers to compromise its user database. Furthermore, IBM [10] estimated that breaches caused by credential compromises cost an average of $4.62 million and took nearly 11 months to address. Thus, protecting secrets from leakage is essential for safeguarding digital assets and reducing security risks.

However, precisely detecting leaked secrets is very challenging due to their diverse forms, such as API keys and passwords, as well as their presence across platforms, file types, and unstructured contexts. Tools like *Gitleaks* [38] and *TruffleHog* [40] rely on basic methods such as regular expressions, entropy checks, and simple whitelists, often leading to severe false positives by misidentifying non-secret strings as secrets. Meanwhile, machine learning approaches also struggle with precision-recall trade-offs in real-world environments due to fixed attributes and predefined classifications. While prior research [74], [68], [62], [72] has made efforts in secret leakage detection, these efforts remain constrained by platform, secret type, and text format, resulting in only modest accuracy improvements. Furthermore, even after leaks are discovered, complete removal is difficult due to secondary storage like caches and mirrors [34], underscoring the urgent need for accurate and timely detection in diverse environments.

§*Both authors contributed equally to this research.*
✉ *Co-corresponding authors.*

Moreover, the absence of high-quality benchmarks makes comprehensive evaluation challenging. The existing dataset [56], primarily derived from pre-filtered GitHub repositories, shows evident platform and tool biases. Its lack of cross-platform diversity restricts the ability to conduct a comprehensive evaluation across different file types and environments.

Our Approach. We introduce KEYSENTINEL, a secret detection tool that leverages advanced extraction and filtering techniques. In the secret extraction phase, we design 910 regular expressions to capture a wide range of secrets, including structured secrets with fixed formats and unstructured secrets requiring prefix-based extraction. Additionally, we use string parsing to filter out incomplete strings and develop several fine filters based on secret characteristics to screen out non-secret strings.

Specifically, we define two main types of secrets: machine-generated secrets are random without semantic meaning, while human-chosen secrets often have recognizable content. For machine-generated secrets, we apply granular semantic analysis to exclude strings with semantic meaning or regular patterns, while for human-chosen secrets, we use machine learning to identify non-secret strings. Furthermore, we adopt heuristic-based prefix analysis to minimize false positives from prefix matching. Eventually, KEYSENTINEL achieves 91.18% precision and an F1 score of 0.86.

Dataset & Benchmark. To measure secret leakage across different platforms, we collected a cross-platform dataset that includes 80,330,098 files from 4,280 GitHub repositories, 668,847 PyPI packages, and 41,719 WeChat mini-programs (MPs). Furthermore, based on this raw dataset, we build a benchmark dataset with 11,826 manually labeled secrets, scattered across 1,806,530 files (608,945 files on GitHub, 634,721 on PyPI, and 562,864 on WeChat).

Evaluation. We compare our tool with six previous tools in terms of secret detection. The results show that our tool performs best, achieving superior precision (91.18%), recall (81.71%), and an F1 score of 0.86. Among the other tools, *Gitleaks* performs best, with an overall F1 score of 0.37 and precision of 26.60%. It performs reasonably well on GitHub (F1 score of 0.59, precision of 61.29%), but struggles on PyPI and WeChat (F1 scores of 0.26 and 0.16). In contrast, KEYSENTINEL consistently achieves F1 scores above 0.8 and precision at around 90% across all platforms, significantly reducing false positives while maintaining strong robustness. Additionally, our ablation experiments demonstrate that each filtering strategy is effective in secret detection. Finally, we compared KEYSENTINEL with powerful large language models (LLMs), specifically GPT-4 and o1. GPT-4 achieved an F1 score of 0.41, while o1 scored 0.63. However, both had higher false positives and longer processing times than KEYSENTINEL.

Measurement. We conduct a large-scale study to assess secret leaks across three major platforms (i.e., GitHub, PyPI, and WeChat) covering a total of 80 million files. Our analysis found that 24.78% of 4,280 GitHub repositories, 7.47% of 668,847 PyPI packages, and 30.08% of 41,719 WeChat MPs exposed secrets. Typically, secret leaks are found in source code files like .py and .js, and private key files such as .key. However, we found that secret leakage is not limited to these file types. Various text and configuration files, including .config, and .log, also exhibit plenty of secret leaks. Notably, we further tracked the handling of secrets and version updates in PyPI packages. The results show that 85.91% of PyPI packages did not remove secrets during version updates, suggesting that the presence of secrets was not detected. Only 4.09% removed secrets, while 8.84% even added more secrets.

Furthermore, by applying KEYSENTINEL to a cooperated IT company's internal codebase, we detected 858 leaked secrets across 256 of 107,625 files in 7 projects and further identified 38 valid secrets out of 104 examined by developer communication. These findings underscore the widespread issue of secret leakage in both open-source and private codebases. We further discuss the main cause of secret leakage and propose some best practices to mitigate these risks.

Responsible Disclosure. We have responsibly disclosed our findings and sent 3,906 disclosure emails. At the time of writing, we have received 205 responses, with 12 confirmations of valid secret exposure.

Contributions. The main contributions of this paper are:

- ***New Benchmark***: We develop **the first cross-platform benchmark** featuring 11,826 label real-world secrets, leveraging a comprehensive dataset of over 18 million raw project files compiled from GitHub, PyPI, and WeChat, which enables more precise evaluations of existing secret detection tools.
- ***A SoTA Tool***: We have designed and developed a novel secret detection tool that **achieves state-of-the-art** performance compared to existing mainstream tools and LLMs (GPT-4 and o1), featuring a precision of 91.18%, a recall of 81.71%, and an F1 score of 0.86, making it suitable for use in various environments.
- ***Large-scale Measurement***: We conduct the first large-scale, cross-platform analysis of secret leakage risk in the wild, examining over 80 million files across three major platforms, revealing varying leakage rates on a per-project basis: GitHub (24.78%), PyPI (7.47%), and WeChat (30.08%). Additionally, we assess secret leakage in a real-world IT company with permission, confirming 38 valid secrets out of 104 investigated, uncovering issues like secrets in log files and third-party template leaks, which lead to specific mitigation recommendations.

Open Source. We will open-source our tool and benchmark on GitHub [53].

Roadmap. The remainder of this paper is organized as follows. Section 2 reviews the current state of secret detection technologies and related work. Section 3 describes our new secret dataset and the corresponding benchmark. Section 4 presents the detailed design of KEYSENTINEL, and Section 5 gives its prototype implementation. Section 6 evaluates

the performance of KEYSENTINEL against other popular tools using our benchmark dataset. Section 7 discusses the results of various measurements, including insights from secret detection in a real IT company. Section 8 discusses the limitations of our work, the lessons learned, and the good practices. Finally, Section 9 concludes this paper.

2. Background & Related Work

In this section, we provide an overview of secret definitions and the related work in secret detection, benchmarking, and leakage measurement.

2.1. Background

Secret. In software development, we categorize all credentials used for authentication, database access, API interactions, and similar functions as *secrets*. These secrets are divided into two main types: machine-generated [72] and human-chosen [67], [75] secrets. Specifically, machine-generated secrets are created with cryptographic algorithms or randomization techniques, making them inherently random and devoid of semantic meaning. In contrast, human-chosen secrets are selected by individuals and often include recognizable or meaningful information.

Additionally, machine-generated secrets can be further categorized as either structured or unstructured based on their format. For instance, Google API keys typically start with `AIza` [74], and private keys are often identified by the marker `-----BEGIN [label]-----` [64], indicating specific patterns. In contrast, some machine-generated secrets, like the fixed 30-character alphanumeric OpenCage geocoding API key [31], are entirely random and lack discernible structure. Besides, we treat human-chosen secrets as unstructured since they are selected by individuals and often contain meaningful information.

Leaked Secret. Leaked secrets, as defined in this study, are hard-coded credentials found in any non-binary file that could pose potential risks. Previous research on secret leakage has primarily focused on source code files or specific types of private keys and certificate files [62], [72], [74], [82]. However, our findings reveal that secrets are not limited to these sources and can also appear in comments, documentation, and other non-binary files. Secrets embedded in binary files are not part of this scope. Plus, we also excluded non-English files.

2.2. Secret Detection

Detection Tools. There are lots of research efforts about secret detection [72], [61], [71], [55]. Rahman et al. conducted an industrial case study on the limitations of secret detection tools. They found that these tools often have high false positive rates, which reduces developer trust and leads to persistent secret leaks [71]. To compare existing tools more effectively, we adopted the criteria established by Basak et al. [55]. Based on tool availability, popularity, and the ability to scan local files, we selected several widely used and reliable tools. These include *Ggshield* [37], *Gitleaks* [38], *TruffleHog* [40], *Repo-supervisor* [35], *Git-secret* [36], and *Whispers* [39]. Furthermore, we summarized the reviews of these tools, which are presented in Table 1. First, different tools have varying input and output formats. For example, *Repo-supervisor* lacks line number support, which makes localization analysis difficult. *Whispers* introduces noise by analyzing file paths and sensitive functions. Second, some tools have become outdated and lack maintenance and updates. For instance, *Repo-supervisor* and *Whispers* have not been updated for over a year. Moreover, most tools rely on basic entropy filtering and whitelisting, which limits their detection effectiveness.

To better understand secret detection, we focus on two main processes: secret extraction and secret filtering, along with related works.

Secret Extraction. Secret extraction involves identifying potential secrets in various files. Most tools use regular expressions and entropy checks to extract secrets [74]. Meli et al. [70] combined regular expressions and filters, achieving over 90% accuracy. However, their approach primarily focused on structured secrets and overlooked unstructured ones. Zuo et al. [82] applied programmatic analysis to source code but struggled with other file types. Tools like *Gitleaks* [24] and *TruffleHog* [25] relied on prefix matching, which often results in high false positive rates.

Secret Filter. As not all extracted strings represent true secrets, secret filtering is necessary to isolate genuine secrets from extracted candidates. Meli et al. [70] used entropy and pattern-based filters but found limited adaptability outside specific conditions. Saha et al. [72] developed a voting classifier combining multiple algorithms to reduce false positives, but their dataset was limited [56], restricting real-world application. Wen et al. [76] used reinforcement learning to assess file sensitivity, focusing on speed over comprehensive benchmarking. In addition, human-generated passwords pose challenges for entropy-based filters because they often lack discernible patterns. Feng et al. [62] used TextCNN models with semantic analysis but faced misclassification issues with diverse file types.

2.3. Secret Benchmark Dataset

Table 2 summarizes current benchmark datasets. The dataset created by Sinha et al. [74] contains 84 GitHub repositories, focusing on AWS credentials, but is limited in both type and volume. Similarly, Saha's dataset [72] includes around 700 API keys, tokens, and passwords from 300 repositories, yet remains small in scale. Neither dataset is publicly available by now. In contrast, SecretBench is the only public dataset [56], which scanned GitHub using *Gitleaks* and *TruffleHog* with a set of 761 regular expressions. However, it exhibits platform and tool biases due to its exclusive focus on GitHub and reliance on specific tools [55].

TABLE 1: Comparison of secret detection tools.

Tool	Line Num.	Multi-factor	Still Maintained	Public Regex (#)	# Stars	Filtering Method	Non-secret Detections
Ggshield	✓	✓	✓	✗	1,500	Not disclosed	Username, port, etc.
TruffleHog	✓	✓	✓	✓ (736)	13,200	Entropy	-
Repo-supervisor	✗	✗	✗	✗	631	Whitelist, Entropy	-
Gitleaks	✓	✗	✓	✓ (166)	14,600	Whitelist, Entropy	-
Whispers	✓	✗	✗	✓ (19)	468	Whitelist, Similarity	File path, threat function
Git-secret	✓	✗	✓	✗	11,800	Not disclosed	-

TABLE 2: Comparison of secret benchmark datasets.

Benchmark	Machine Secret*	Human Secret**	Public Accessible	Manually Labeled	Not Pre-filtered	Cross Platform	Target File	Extraction Tool	# True Secret
Sinha's (2015)	✓	✗	✗	-	-	✗	Source code	-	-
Saha's (2020)	✓	✓	✗	✓	✗	✗	Source code	Saha's customized tool	700
SecretBench (2023)	✓	✓	✓	✓	✗	✗	Non-binary file	*Gitleaks* & *TruffleHog*	4,014
Ours	✓	✓	✓	✓	✓	✓	Non-binary file	All tools from Table 1	7,062

* Machine-generated secret; ** Human-chosen secret.

Despite these efforts, current datasets often suffer from a lack of diversity, limited real-world applicability, and insufficient cross-platform representation. In summary, they exhibit three main limitations:

- **I. Pre-filtering Bias**: Previous dataset samples are biased due to pre-filtering with specific tools or keywords (e.g., *TruffleHog*). Thus, they cannot accurately represent the true distribution of secrets in the wild.
- **II. Tool Bias**: Dataset extraction relies on specific tools like *Gitleaks* and *TruffleHog*. It creates a bias toward the secrets detected by these tools. As a result, the dataset becomes unfair to others [55]. Additionally, the approach limits the coverage of secrets.
- **III. Platform Bias**: Most benchmarks focus on a single platform (e.g., GitHub), lacking cross-platform data. It restricts comprehensive evaluation, as different platforms have unique secret usage (leakage) patterns.

2.4. Secret Leakage Measurement

Secret leakage across various platforms has become a significant issue, prompting increased attention to measurement efforts. Meli et al. [70] conducted the first large-scale measurement of GitHub, creating optimized regular expressions and filters for near real-time detection. Dahlmanns et al. [60] extended Meli et al.'s work to container images, identifying 740 compromised private keys verified in real environments. However, their focus only on API keys and tokens likely under-reported broader leakage issues. Zhang et al. [80] examined WeChat MPs, targeting app secret vulnerabilities with API-verifiable methods, but lacked comprehensive detection techniques or large-scale risk assessment. Furthermore, secret leakage on platforms like PyPI remains largely unexplored.

3. Dataset & Benchmark

To address the limitations of existing benchmarks, we build a more comprehensive secret benchmark. As illustrated in Figure 1, the dataset construction process involves collecting files from major platforms without pre-filtering, which helps reduce bias and supports large-scale cross-platform analysis. After extracting candidate secrets using various tools, we perform iterative manual refinement to ensure the benchmark aligns with real-world detection needs and supports thorough performance evaluations.

3.1. Building the Raw Dataset

We chose GitHub repositories, PyPI packages, and WeChat MPs as primary data sources for their unique characteristics: GitHub for diverse programming practices, PyPI for insights into typical open-source ecosystems, and WeChat for its closed-source environment requiring specialized parsing.

Collecting Files. To address potential sampling biases, we collected files from these platforms using tailored acquisition strategies for each:

- **GitHub**: We selected the top 5,000 projects from Gitstar-ranking (accessed on July 15, 2023) [3], focusing on repositories under 100 MB. This yielded 4,280 repositories with 3,081,425 files across 6,906 file types.
- **PyPI**: Using the PyPI XML-RPC API [63], we collected packages from January 1, 2023, to April 30, 2024, resulting in 668,847 packages with 57,686,181 files across 5,759 file types.
- **WeChat**: Based on prior research [81], we developed a MP crawler, collecting (March–October 2023) 41,719 projects with 19,562,492 files across 71 file types.

In total, we collected 80,330,098 files across 10,961 file types, occupying 2.95 TB of disk space.

3.2. Building the Benchmark Dataset

Given the large-scale secret detection across different platforms, we extract a subset of our raw dataset as a benchmark dataset to evaluate detection tools. To reduce

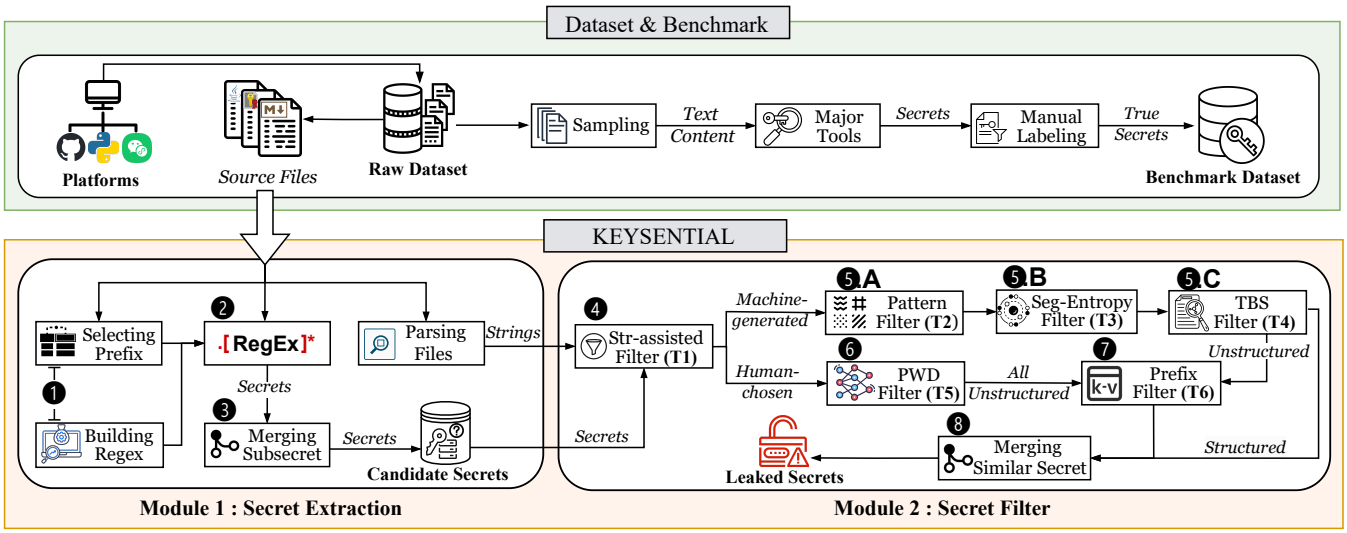

Figure 1: Overall workflow of our work.

the workload of subsequent manual filtering and make the dataset more balanced, we randomly sample projects from our raw dataset and aim for a similar number of files across platforms. It results in 608,945 out of 3,081,425 (19.76%) candidate files from GitHub repositories, 634,721 out of 57,686,181 (1.10%) from PyPI packages, and 562,864 out of 19,562,492 (2.88%) from WeChat MPs. There are two main building processes:

Extracting Candidate Secrets. Manually searching for and labeling secrets in a massive amount of text is impossible. Instead of manually labeling secrets directly from the raw files, we use existing secret detection tools to annotate the extracted secrets, reducing the workload. To expand the coverage as much as possible and reduce the overrepresentation bias of secrets detected by any specific tool, we incorporate all available tools in the extraction process and treat all detected results as candidate secrets. This strategy allows for a fair and inclusive comparison of detected outputs across different tools. Moreover, we standardized the dataset and extraction process by storing files locally and executing each tool as per its guidelines. We developed a Python script to unify tool outputs and filtered out non-secret items, setting placeholders for tools lacking line numbers.

Labeling True Secrets. We obtained a total of 709,125 candidate secrets through the tools. Due to the high false positive rate of existing tools, ensuring the reliability of manual labeling is essential. We adopted the following steps and rules to ensure the reliability of the benchmark:

- **I**: Since many false secrets share similar characteristics or formats, we streamlined the labeling process by identifying recurring patterns and designing scripts to filter them out, reducing the manual labeling workload. For example, common false positives included wxss [45] style elements (e.g., `font-` or `width:`), from which we filtered out 194,948 instances. Table 3 shows tools flagged 498,936 candidate secrets in WeChat MPs, but only 1,029 were confirmed as true.
- **II**: We assigned two experts with over five years of experience in software vulnerability discovery to independently review each case's context and metadata, following a standardized guideline. The labeling process was double-blind, meaning the experts were unaware of each other's decisions and had no knowledge of which tool detected each candidate secret, ensuring an unbiased evaluation.
- **III**: Once the labeling was completed, disagreements were resolved through a secondary consensus review, leading to the final results. The overall reliability of the labeling process was reinforced by independent annotations from two researchers, who achieved a Cohen's Kappa value of 0.91 [59], indicating "near perfect agreement" according to Landis and Koch [66].

Benchmark Properties. Our benchmark includes 11,826 manually-labeled true secrets, covering various types of sensitive information such as API keys, tokens, private keys, database credentials, and passwords. Secret attributes defined in the dataset include file path, secret content, project name, and starting line/column numbers, along with secret category, file type, entropy, and exposure status. Table 3 shows the distribution: GitHub holds 7,659 secrets, PyPI 3,138, and WeChat 1,029, reflecting variations due to project scale, code structure, and development practices.

TABLE 3: Statistics of secret sources across platforms.

Platform	# Files	# Candidate Secrets	# True Secrets
GitHub	608,945	87,427	7,659
PyPI	634,721	122,762	3,138
WeChat	562,864	498,936	1,029
Total	1,806,530	709,125	11,826

4. Design of KEYSENTINEL

4.1. Motivation

Existing secret detection techniques struggle to accurately identify machine-generated secrets due to their semantic complexity. Traditional methods—such as pattern-based filtering, entropy analysis, and word-based filtering—often produce high false positive rates, failing to distinguish true secrets from non-secret strings that exhibit high entropy. Additionally, prior machine learning approaches, like text-CNN models, are primarily designed for password classification and do not generalize well to diverse machine-generated secrets. A fundamental limitation is that simply combining previous methods or tools does not yield optimal results. Many existing techniques operate in isolation and suffer from inherent limitations, failing to leverage complementary strengths. Instead of mere integration, our approach refines and extends these methods to enable more fine-grained analysis and filtering.

4.2. Overview

The overall process of our secret detector KEYSENTINEL is shown in Figure 1. KEYSENTINEL consists of two main modules: secret extraction and secret filtering, incorporating six filters (T1 to T6). The secret extraction module begins by selecting prefixes and building regular expressions (❶) based on prior research and tools. These regular expressions are then used to scan files and extract potential secrets (❷). Afterward, duplicate secrets are merged (❸). Finally, KEYSENTINEL extracts a set of candidate secrets. The secret filtering module begins with the String-assisted (Str-assisted) Filter (T1), which parses strings and matches them with candidate secrets (❹), then further filters based on their category and structure. Machine-generated secrets undergo the Pattern Filter (T2), Segmented Entropy (Seg-Entropy) Filter (T3), and Tokenization-based Semantic (TBS) Filter (T4) for selection (❺). Human-chosen secrets are screened using the Password (PWD) Filter (T5) to exclude non-secrets (❻), while unstructured secrets are evaluated with the Prefix Filter (T6) to reduce false positives (❼). Secrets failing any filter are marked as non-secrets, and the remaining valid candidates are consolidated by similarity (❽).

Challenges. Since the design of KEYSENTINEL still initialize from regex-based methods, we should address the following challenges:

- **Challenge I**: The diverse categories and formats of secrets pose a challenge, leading to limited coverage in existing regular expressions. Additionally, their extraction methods struggle to pinpoint secret content, limiting analysis accuracy.
- **Challenge II**: The complexity of the text environment in which secrets appear often leads to regex matches that result in duplicate or incomplete strings. Leveraging the structural features of the text is essential to ensure the completeness and correctness of candidate secrets.

4.3. Secret Extraction

In this module, we aim to extract a wide range of candidate secrets while ensuring correctness.

Building Extraction Regex. To address Challenge I, we first gather raw regular expressions from open-source tools (e.g., *Gitleaks* and *TruffleHog*) and Meli et al. [70]. We then refine these expressions through the following steps:

- **Secret Prefix Matching**: Unstructured secrets, including some machine-generated and all human-chosen ones, make up 60% of detected secrets. Their lack of standardized formatting makes them difficult to recognize. To better identify these secrets, we use a wider range of prefixes. Through manual analysis of candidate files, we select 11 secret prefixes, such as `key`, `pass`, `password`, and `secret`. These prefixes help capture more potential unstructured secrets for further analysis.
- **Expanding Scanning Range**: Tools like *Gitleaks* [24] and *TruffleHog* [25] typically scan for passwords longer than ten characters, covering only 37.29% [73]. To improve coverage, we reduce the minimum length to five characters, capturing 99.5% of passwords [73]. Additionally, we extend the scan to cover special characters (e.g., &, @, $, #), which helps catch complex secrets often missed by other tools.
- **Targeted Secret Extraction**: For secrets in URI formats like `id:secret@example.com`, we focus on extracting only the `secret` part. We standardize regular expressions to extract only the secret, with prefix-based extractions divided into two groups: one for the prefix (e.g., `password`:examplepass) and one for the secret (e.g., password:`examplepass`). This method improves precision and reduces false positives in filtering.

After these refinement steps, we develop 910 optimized regular expressions covering machine-generated, generic-prefix, and human-chosen secrets, ensuring comprehensive extraction coverage.

Merging Subsecret. Due to using numerous regular expressions, a single text may match multiple patterns. For instance, in `key=abcd&e`, one regex might extract `abcd&e` while another captures `abcd`. Thus, removing substrings is essential to improve accuracy and reduce unnecessary analysis and computation. We identify the longest substring through traversal and merge the secrets accordingly (as detailed in Appendix B). We are also inspired by this to design non-secret entities. If the longest substring matches the format of a non-secret entity, like a certificate, all its substrings are treated as non-secret. Based on this, we design a method to identify non-secret entities, like certificates, often prone to false positives. By leveraging the relationships between secret substrings, we can filter out false-positive secrets and exclude non-secret entities, improving overall accuracy.

By all those mentioned processes, we extract candidate secrets with a lower false positive rate.

Algorithm 1 String-assisted Filter

```
function STRING_ASSISTED_FILTER(file_path, candidate_secret)
    try:
        strings ← parseFile(file_path)        ▷ Parse file based on its type
    except:
        return candidate_secret               ▷ File types are not supported
    filtered_hit ← ∅
    for all hit in candidate_secret do
        if IntegrityVerify(hit, strings_value) then
            skip hit                          ▷ Filter out incomplete strings
        else
            append hit to filtered_hit
        end if
    end for
    return filtered_hit
end function
```

4.4. Secret Filter

To address Challenge II, we need to filter out non-secrets in candidate secrets. Below, we detail the design of our filter for raw strings (T1) and filters for machine-generated (T2, T3, T4) and human-chosen secrets (T5, T6).

T1:String-assisted Filter. As mentioned in Challenge II, to address the issue of incomplete candidate secrets containing non-string characters, we develop a string-assisted filter to validate candidate secrets' integrity (❹). In detail, true secrets often appear as specific strings in structured text, such as code literals or JSON values. However, extracting by regular expressions alone can misidentify incomplete fragments as secrets. To reduce false positives, we use text parsing to verify string positions and compare candidate secrets with parsed text.

TABLE 4: Parsing methods of major file types.

File Type	Parsing Tools	Runtime	Category
`js`, `jsx`	Babel (npm) [6]	Node.js	Code
`ts`	Babel (npm)	Node.js	Code
`py`	AST (PyPI) [7]	Python	Code
`java`	Javalang (PyPI) [8]	Python	Code
`go`	Go/AST [4], Go/Parser [5]	Golang	Code
`json`	JSON (PyPI)	Python	Data
`yaml`	YAML [50], JSON (PyPI)	Python	Data
`xml`	XMLtoDict [51], JSON (PyPI)	Python	Data
`plist`	Plistlib [52], JSON (PyPI)	Python	Data
`csv`	Pandas [49], JSON (PyPI)	Python	Data
`ipynb`	JSON (PyPI)	Python	Data

We categorize the detected files into two types: code and data. As shown in Table 4, we utilize different tools for analysis. Given the limitations of existing tools, we focus on 11 specific file types, which still account for 86% of structured files. For code files, we use abstract syntax trees (ASTs) to extract strings. For data files, we convert them to JSON format and extract values as potential secret strings.

We design the first filter according to the Algorithm 1, ensuring the integrity of candidate secrets by parsing 11 structured file types. After parsing, we compare candidate secrets with parsed string characters for assisted filtering. Certain secret types are excluded due to their inherent length (e.g., private keys) or subcomponent structure (e.g., a secret being a part of a URI scheme). The main principles of our selection strategy are:

- Secrets appearing at the same position in the text are considered valid.
- If the content at the same position is a substring or intersects with the parsed string, it is incomplete and not valid.
- Secrets detected outside parsed positions should be retained, such as those in comments, which may still be valid.

T2:Pattern Filter. Previously, Meli et al. [70] developed a pattern whitelist for continuous sequences such as `aaaa`, `1234`, and `dcba` [70]. We extend this by including special symbols (e.g., `&, @, $`) to exclude them from machine-generated secrets. To achieve this, we create an algorithm that detects repeated, sequential, and reverse patterns across all character types, forming a whitelist through heuristic filtering (❺.A).

T3:Segmented Entropy Filter. To comprehensively reduce randomness issues, we introduce a filter based on Shannon Entropy [46]. Previous research applied a 3-standard deviation threshold to filter out outlier candidate secrets [70]. As mentioned earlier, our study finds that candidate secrets with common prefixes are prevalent but vary in type and length, requiring manual inspection. To enhance randomness, we filter out secrets with below-average entropy. Following statistical guidelines [69], we calculate average entropy for categories with over 30 occurrences, as low-frequency categories may lack reliability. Additionally, around 3% of longer secrets show high entropy but contain localized patterns that need further review.

To address the high entropy of long secrets, we implement a segmented entropy algorithm. Research indicates that secrets under 30 characters are the most common [62]; thus, we divide longer secrets (over 30 characters) into 30-character segments and assess the entropy of each segment. Segments shorter than 30 characters at the end are merged with the second-to-last segment. Through experimentation, we determine effective thresholds for average entropy filtering, applying a threshold of 2.4 for secrets shorter than 30 characters and 3 for each segment of longer secrets. This conservative approach (say ❺.B) ensures that any remaining unfiltered secrets are addressed by subsequent filters, as validated in Section 6.

T4:Tokenization-based Semantic Filter. Although the previous filters (i.e., T2, T3) can filter candidate secrets with random characteristics, this is still insufficient. For example, if the secret string is a sentence (e.g., `YourString_is_not_a_TrueSecret`), its entropy can reach 3.7, exceeding the threshold and thus bypassing the statistical filter (i.e., T3). As machine-generated secrets typically do not contain human-readable or meaningful content, simple entropy or pattern checks are inadequate for filtering such secrets that may cause false positives. It makes semantic analysis essential for accurate identification. Further-

more, current word filtering faces semantic issues due to two factors: limited dictionary coverage and overly simplistic semantic analysis based solely on word presence, leading to false positives. For example, `auth` in `sql_auth_initdb` is meaningful but missed by existing rules. Thus, a finer-grained semantic analysis algorithm is needed.

- **Word Segmentation**: We select a comprehensive wordbook from an open-source GitHub project based on n-gram data from Google [16]. The wordbook contains the top 500,000 mixed words, including nouns, verbs, and technical terms (such as `npm` for Node.js). The broad coverage is crucial because coding contains a mix of everyday words, abbreviations, and technical terms. However, word inclusion alone is insufficient for identifying secrets. Longer secrets may contain short words or substrings (e.g., `App` in `Apple`), leading to false matches. Therefore, specialized tokenization techniques are necessary. While tools like NLTK [21] and TextBlob [22] struggle with concatenated strings, WordNinja [23] can address this issue. However, it may incorrectly split abbreviations (e.g., `auth` into `au` and `th`). Our analysis indicates that most abbreviations follow patterns like `a_b_c` or `a.b.c` and are typically under five characters [20], [15]. To address this issue, we leverage this characteristic and design Algorithm 2 in Appendix, which combines custom regular expressions with WordNinja.
- **Semantic Threshold**: While machine-generated secrets are generally random, longer ones may sometimes resemble real words. To establish a threshold for semantic content, we analyze candidate secrets, categorize them as meaningful or meaningless, and iteratively refine the threshold. This process eventually results in the threshold conditions defined in Algorithm 3 in Appendix (❺.C), which evaluates the semantic significance of words based on frequency and length.

T5:Password Filter. Human-chosen secrets often cause false positives, as placeholders like `yourpass` or `put_your_secret` may be misidentified as actual secrets. Whitelist filtering alone is insufficient to distinguish these from true secrets. For example, prefix-based detection may mistakenly recognize `password: g_bytes_ref`, which is not a secret but a GLib method [41], leading to false positives.

To address this issue, we use a TextCNN-based model with nine layers, consisting of six convolutional and three fully connected layers, for character-level text classification [79]. Using an open-source secret dataset [19], we categorize inputs into three types: human-chosen passwords, machine-generated secrets, and non-secret strings. Human-chosen secrets can sometimes include complex, strong passwords with high randomness, which this model may classify as machine-generated secrets. To maintain effective filtering and avoid false negatives, we consolidate the output from three into two categories: human-chosen passwords and machine-generated secrets as valid, while ordinary strings are categorized as non-secrets (❻).

To evaluate the model, we select 3,601 passwords from our benchmark, of which 1,465 are true and 2,136 are false, for identification and filtering by the model. Although in real scenarios, the precision rate of 62.28% may not reach the outstanding performance described by Feng et al. [62], the model can still filter out a portion of non-secret content and retain a high recall rate. To further enhance the filtering process, we incorporate a prefix filter, which boosts the model's performance to 81.05% precision and an F1 score of 0.85, as Table 5 shows.

TABLE 5: Performance of PWD Filter (T5) with and without Prefix Filter (T6).

Method	Precision	Recall	F1
T5: PWD	62.28%	89.35%	0.77
T5: PWD + T6: Prefix	81.05%	88.74%	0.85

T6:Prefix Filter. Our benchmark indicates that 60% of secrets require generic prefix matching due to their lack of distinctive structures, which increases the risk of false positives. Consequently, we propose a prefix filter to process these candidate secrets further (❼). We identify and propose improvements for two major issues related to false positives associated with prefixes:

- **Prefix Confusion**: Generic regular expressions often match patterns like `api`. To better capture the context surrounding potential secrets and enhance detection coverage, we use a non-capturing group `(?:[0-9a-z\-_\t.]{0,20})` to match 0 to 20 characters around the target. However, this can lead to false matches within words (e.g., `api` in `capital`). To address this, we create a whitelist using English words and terms from open-source projects [16], [17], applying word-splitting techniques for heuristic filtering.
- **Semantic Analysis of Prefix**: Some matched strings with prefixes may not indicate actual secrets. For instance, *Gitleaks* and *TruffleHog* often trigger false positives for strings like `API_username` or `fake_API`, and `publickey`. Table 6 summarizes common false-positive scenarios. To address this, we split prefixes (e.g., `a_key_b` into `a`, `key` and `b`) and create a whitelist based on extensive file scans and known patterns for heuristic filtering.

TABLE 6: Common false-positive scenarios from prefixes and heuristic filtering.

False-positive Scenarios	**Example**
1. Related to files or paths	AuthenticateUser.php => `h4f1***`
2. High randomness like hashes	user_access_sha256 : `a4dv***`
3. Identifiers	client_id : `5v4f***`
4. Negative descriptions	fake_api : `AAAA***`

Similar Secret Merging. Upon completing the secret extraction and filtering process, KEYSENTINEL produces the final set of correct secrets. When multiple secrets are identical or intersect at the same position, merging similar secrets is necessary (see ❽). To facilitate this, we utilize two

established algorithms: the Jaro-Winkler algorithm [78] and the SequenceMatcher algorithm [57]. The former rewards strings with matching prefixes by assigning higher similarity scores. The latter calculates similarity by identifying the longest common subsequence and recursively matching characters in the non-matching regions. These algorithms enable more flexible secret comparisons even though the output secret structure varies slightly. We adopt the conditions and thresholds as previous work [55]: 1) a Jaro-Winkler similarity score ≥ 0.7, or 2) a SequenceMatcher score ≥ 0.6. Two secrets are considered identical if their content is identical or similar at overlapping positions.

5. Implementation

We implemented KEYSENTINEL using 4,269 lines of code (4,095 in Python, 61 in JavaScript, and 113 in Go). Specifically, we used 910 custom regular expressions to extract secrets and designed a merging algorithm to filter non-secrets and reduce duplicates. In addition, to ensure the integrity of the secrets, we developed 11 text-parsing methods for common file formats (see Table 4). For similar secret merging, we use the similarity comparison functions from the `jellyfish` [12] and `difflib` [13] PyPI packages, respectively. Furthermore, the deep learning models used in the PWD Filter (T5) are trained and implemented with PyTorch [1] using an NVIDIA Tesla V100 GPU (32 GB).

To conduct large-scale scanning, we utilize an Apache Spark YARN cluster with 3,000 cores and 6,144 GB RAM. A Docker image was employed for cross-platform scanning in restricted environments, and we skipped files over 1 MB as recommended by previous tools [2].

6. Evaluation

In this section, we evaluate KEYSENTINEL and compare it with six selected tools in our cross-platform benchmark. Additionally, we compare KEYSENTINEL's performance with SoTA LLMs.

6.1. Evaluation of KEYSENTINEL

Experiment Setup. Our experiments are conducted on a server running Ubuntu 20.04.3 LTS with an Intel Xeon Silver 4210 CPU featuring 40 CPU cores and 251 GB of RAM. The experiments are conducted on our benchmark dataset. For the ablation studies, we perform multiple trials, disabling one filter (i.e., T1 through T6) in each trial.

Results. As Table 7 shows, the filters in KEYSENTINEL demonstrate high performance overall, achieving 91.18% precision, 81.71% recall, and an F1 score of 0.86 on the benchmark. Moreover, the result highlights the impact of removing each filter on precision, recall, and F1 score, illustrating the contribution of each component to the overall performance. The TBS Filter (T4) plays a critical role in maintaining accuracy. Its removal leads to a 0.31 drop in the F1 score (from 0.86 to 0.55) and a reduction in precision due to the inclusion of invalid machine-generated secrets. Each filter is essential, as removing any one results in a significant decrease in the precision. The multi-layered filtering strategy ensures KEYSENTINEL's high accuracy and reliability in detecting real-world secret leaks.

TABLE 7: Results of ablation experiment.

Method	Precision	Recall	F1	δF1 ↓	Filter Objects
(w/o) T1: Str-assisted	85.10%	83.45%	0.84	-0.02	Structured file
(w/o) T2: Pattern	86.92%	82.19%	0.84	-0.02	Machine-generated secret
(w/o) T3: Seg-Entropy	57.87%	83.29%	0.68	-0.18	Machine-generated secret
(w/o) T4: TBS	**41.80%**	**81.79%**	**0.55**	**-0.31**	**Machine-generated secret**
(w/o) T5: PWD	81.05%	81.97%	0.82	-0.05	Human-chosen secret
(w/o) T6: Prefix	43.29%	86.02%	0.58	-0.29	Unstructured secret
KEYSENTINEL	91.18%	81.71%	0.86	0.00	All

δ**F1** ↓: Difference in F1 score compared to KEYSENTINEL.

Additionally, we investigate the impact of different filters across platforms and find subtle inconsistencies. For WeChat, T4 is most crucial; its removal decreases the F1 score by 0.68. T3 is also significant, reducing the score by 0.26, likely due to the limited file type diversity and obfuscation in .js files that disrupts semantic content. In contrast, GitHub and PyPI mainly rely on T6 due to the prevalence of unstructured secrets. Detailed data is available in Table 13 in Appendix.

In summary, our approach not only enhances existing methods (T2-T5) but also introduces new techniques (T1, T6) to tackle previously unresolved challenges, significantly improving performance for more comprehensive and accurate secret detection.

6.2. Comparative Evaluation

We evaluate the following detection tools on our benchmark: *Ggshield*, *Gitleaks*, *TruffleHog*, *Repo-supervisor*, *Git-secret*, *Whispers*, and compare their results with those of KEYSENTINEL.

Experiment Setup. To standardize comparisons across tools with varied output structures, we conduct scans using the default commands outlined. Since different tools may report secrets in slightly different formats, we adopt the same approach used for merging similar secrets to determine whether detected secrets are identical. Since the same secret may appear multiple times in different locations, we handle data imbalance by normalizing the outputs and identifying unique secrets based on project names and values.

Results. Table 8 summarizes tool evaluations. While *Ggshield* achieves the second-highest precision of 73.36%, its strict regular expressions cause it to miss many secrets, resulting in a recall rate of only 14.86%. Moreover, *Repo-supervisor* and *Git-secret* perform poorly, with F1 scores near 0. The former flags 638,386 excessive secrets but lacks proper filtering, leading to numerous false positives. The latter detects only 423 out of 11,826 labeled secrets, indicating insufficient regular expression coverage. In contrast, KEYSENTINEL undergoes filtering through a combination of various filters (T1 through T6), incorporating machine learning,

semantic analysis, and prefix matching techniques. KEYSENTINEL ranks first across all metrics, achieving 91.18% precision, 81.71% recall, and an F1 score of 0.86 with raw output. After normalization, KEYSENTINEL reaches 89.89% precision, 78.62% recall, and an F1 score of 0.84, achieved by filtering out duplicates and retaining unique secrets.

Comparison on Different Platforms. KEYSENTINEL demonstrates strong cross-platform performance in our benchmark, achieving F1 scores of 0.86 on GitHub, 0.87 on PyPI, and 0.87 on WeChat, significantly outperforming all other tools. In contrast, other tools show significant performance declines when moving from GitHub to PyPI or WeChat. For example, *Gitleaks*, the next-best performer on GitHub (precision 61.29%, F1 score of 0.59), experiences sharp drops on PyPI (precision 16.69%, F1 0.26) and WeChat (precision 9.06%, F1 0.16). *TruffleHog* has minor improvements on PyPI (F1 0.08) but remains largely ineffective overall. Details are provided in Table 14 in Appendix.

Overall, most tools, except KEYSENTINEL, exhibit significant cross-platform limitations, particularly on PyPI and WeChat. KEYSENTINEL's superior performance underscores its robustness and effectiveness in handling complex secret detection across diverse platforms.

TABLE 8: Different tools' results of comparative evaluation.

Tool	Raw Output			Unique Secret Normalization		
	Precision	Recall	F1	Precision	Recall	F1
Ggshield	73.36%	14.86%	0.25	74.54%	19.10%	0.30
Gitleaks	26.60%	58.46%	0.37	19.18%	65.02%	0.30
TruffleHog	2.04%	11.27%	0.03	2.63%	13.11%	0.04
Repo-supervisor	0.09%	6.15%	≤ 0.01	0.07%	5.32%	≤ 0.01
Git-secret	1.18%	0.04%	≤ 0.01	9.44%	0.48%	≤ 0.01
Whispers	4.08%	16.91%	0.07	6.40%	14.46%	0.09
KEYSENTINEL	**91.18%**	**81.71%**	**0.86**	**89.89%**	**78.62%**	**0.84**

6.3. Comparison with LLMs

With the recent advancements in LLMs for secret detection, we conduct additional experiments to assess KEYSENTINEL's performance against leading LLMs, specifically OpenAI's GPT-4 and the recently introduced OpenAI o1, known for its strong complex reasoning capabilities.

Experiment Setup. Given the high costs of analyzing the full dataset via the OpenAI API [11], large-scale measurements are impractical. Instead, we select a sample of 1,000 files, including 500 with secrets and 500 without. These files represent 70 distinct file types, including code files like .py and .js, document files like .md and .txt, and other files like .pem and .json. We use our prompt for GPT-4 Turbo and o1-preview to make them a secret detector. Our prompt has been referred to the GitGuardian 2024 technical report [18] and show as follow:

```
You are an expert in secret detection for
plaintext files. You will receive plaintext
content, and your task is to identify any
secrets within the text. If a secret is found,
provide your response in the following format:
   Name: [name of the secret]
   Secret: [copy of the secret]
   Description: [short description of the
secret]
   Repeat the format for each detected secret.
If no secrets are detected, respond with: "ALL
CLEAR."
```

TABLE 9: Unique secret normalization vs. LLMs

Tool	Precision	Recall	F1
KEYSENTINEL	**91.47%**	**80.70%**	**0.86**
GPT-4 Turbo	43.76%	38.71%	0.41
o1-preview	62.80%	62.50%	0.63

Results. As shown in Table 9, KEYSENTINEL outperforms both GPT-4 Turbo and o1-preview, achieving the highest accuracy (91.47%), recall (80.70%), and F1 score (0.86). o1-preview ranks second with 62.80% accuracy and an F1 score of 0.63, while GPT-4 Turbo has the lowest scores across all metrics. Despite their reasoning strengths, models like GPT-4 Turbo and o1 still suffer from high false positives and inconsistent results—e.g., o1 may flag the same placeholder differently. Even with prompt tuning and Chain-of-Thought (CoT) techniques, current LLMs face notable limitations in secret detection, especially in terms of accuracy and cost, requiring human oversight. Moreover, KEYSENTINEL is also faster, scanning each file in 0.36 seconds on average, compared to 7.57 seconds for GPT-4 Turbo and 19.61 for o1. Given that 90% of commits involve ≤ 7 changed files [54], KEYSENTINEL achieves practical efficiency with 2 seconds per commit.

To further illustrate the challenges faced by LLMs, we present and analyze two typical false positive cases in the following.

GPT-4 Turbo Case. The code snippet of a false positive case of GPT-4 Turbo is as follows:

```
   ...
   services_jwt = CanvasSecurity::ServicesJwt.
for_user(
   domain,
   @current_user,
   real_user: @real_current_user,
   workflows:,
   context: @context,
   symmetric:
   )
   ...
```

In this case, GPT-4 Turbo mistakenly flagged `services_jwt` as a secret. The variable represents a generated JWT rather than a hard-coded token. The model likely misidentified it due to its association with JWTs, highlighting a limitation in distinguishing between token generation and actual sensitive data.

o1-preview Case. While o1-preview, one of the leading LLMs, resolves the above false positive case from GPT-4 Turbo, it still cannot correctly identify the following code snippet.

```
...
print('Modified this file')
print('Third Modification')
print('changes in second branch')
print('123456 new change in second branch')
...
```

And the CoT and *Response* of o1-preview are as follows:

```
...
Analyzing the code snippet: I'm inspecting a
code snippet with print statements, noting some
hints at potential secrets.
Evaluating the code: Parsing through the
code snippet, '123456' appears to be more of
a placeholder than an actual secret, given its
commonplace usage.
Identifying potential secret: I'm noting
the possibility of '123456' being a hard-coded
password. It's crucial to avoid over-flagging
and ensure accurate identification without
confirming specific policies.
Name: hard-coded Password Secret: 123456
Description: A hard-coded password included in the code.
...
```

In this case, o1-preview mistakenly identified `123456` as a hard-coded password, a false positive since it was part of a print statement used for debugging. This error likely results from over-reliance on detecting numerical patterns without context. It highlights the need for better contextual analysis to avoid misclassifications in secret detection.

7. Measurement

In this section, we measure three platforms—GitHub, PyPI, and WeChat—using the raw dataset from Section 3.1. Additionally, we analyze the internal codebases of an IT company to investigate real-world secret leakage issues.

7.1. Cross Platform Measurement

With the help of KEYSENTINEL, we analyze secret leakage in GitHub repositories, PyPI packages, and WeChat MPs. To ensure the reliability of our measurement, we randomly selected and labeled 300 samples, achieving a precision of 91.33% on GitHub, 91.67% on PyPI, and 96.33% on WeChat MPs. Results reveal a notable prevalence of leaks: 24.78% of 4,280 GitHub repositories, 7.47% of 668,847 PyPI packages, and 30.08% of 41,719 WeChat MPs contain leaked secrets. These findings underscore significant vulnerabilities, particularly in WeChat MPs, and highlight the urgent need for effective platform countermeasures.

File Type Measurement. To explore whether secrets are more prone to leak in specific file types across platforms, we identify the top 5 file types with leaks on GitHub, PyPI, and WeChat, as presented in Table 10.

TABLE 10: Top 5 file types associated with secret leaks on different platforms.

GitHub		PyPI		WeChat	
Type	**Ratio**	**Type**	**Ratio**	**Type**	**Ratio**
.key	59.25%	.rsp	38.45%	.js	0.37%
.pem	25.23%	.pem	32.92%	.json	0.02%
.pm	16.24%	.mhtml	32.74%	.svg	≤0.01%
.http	11.24%	.key	23.57%	.wxml	≤0.01%
.m3u	10.56%	.env	12.62%	.wxss	≤0.01%

- **GitHub**: File types like .key and .pem, used for keys and certificates, show high leakage rates of 59.25% and 25.23%, respectively. Additionally, our analysis identifies secret leaks in some unexpected files like .m3u, which may suggest potential infringements related to unauthorized access or exploitation of IPTV services. A detailed case analysis is provided in Section 7.4.
- **PyPI**: .pem files show a 32.92% leakage rate, and .rsp files have a 38.45% rate due to encryption algorithm testing.The absence of secret detection measures in PyPI makes these issues worse.
- **WeChat**: Since WeChat MPs are powered by JavaScript, .js files account for the majority of leakage instances. However, their proportion of total project files is relatively low (0.37%), likely due to the uniformity of the project file types, with .js files making up the bulk of them. Despite this, static files like .wxml and .wxss [45] also show leaks, likely due to improper handling of sensitive data.

Summary: Secret leaks occur across various file types, with platform-specific high-risk types. Thus, identifying these is crucial to improving detection and prevention.

7.2. Handling of Leaked Secrets

To explore how developers manage leaked secrets, we track both package versions and their update timelines on PyPI. Rather than focusing solely on version changes, we monitor secrets over time to see whether developers remove them in subsequent releases. Combining both version and time-based analysis, this approach provides a clearer picture of how often secrets persist and whether they are addressed in newer updates. It also offers valuable insights into how developers respond to security risks following a leak.

Secrets Change with Packages/Versions. We collect 3,280 PyPI packages with various versions and track the number of unique secrets associated with each. Notably, 85.91% of packages with leaked secrets retain the same number of secrets across version updates, suggesting that many developers are unaware of these leaks. Additionally, 8.84% of the packages exhibit an increase in the number of secrets with newer versions, while 4.09% of confidential information is removed. However, some secrets are only partially deleted, and older versions of these packages remain accessible, presenting significant security risks. In PyPI, outdated versions are seldom removed or deprecated, allowing exposed confidential information to be easily exploited by potential attackers. Thus, even if developers remove sensitive information in newer versions, users relying on older versions may inadvertently introduce security risks.

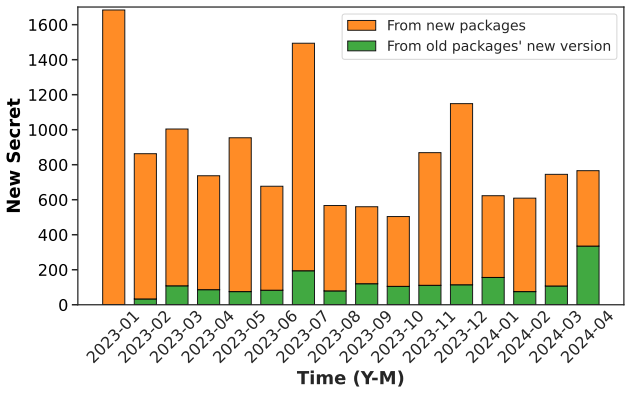

Figure 2: Monthly leaked secret statistics on PyPI.

Temporal Analysis of Secret Leakage. To investigate the temporal trends of secret leakage and the sources of new leaks, we refer to GitGuardian's analysis [14], which applies restrictions on a large volume of alerts to reduce noise. As illustrated in Figure 2, the exposed secrets can be attributed to two primary categories: the introduction of new secrets in updated versions of previously leaked packages and the emergence of newly leaked packages. Specifically, on average, 89.43% of the monthly leaked secrets are linked to newly released packages, which are the primary contributors to secret leakage, while 10.57% relates to new versions of existing leaked packages, which also show an upward trend. For example, in the `***system-server` package, an API key was leaked in the settings.py file during Django deployment; while in the `***client` package, a password was first leaked in the METADATA file in January 2023, with new secrets exposed in the updated version within the administration.py file in February 2023. On average, each new package contains 2.48 leaked secrets, while each existing package version has an average of 0.04 new leaks. These findings indicate that many developers continue to introduce new secrets during updates. Thus, it is crucial to enhance oversight and raise developers' awareness of best practices in secret management.

7.3. Secret Detection in the Real-world Company

To investigate secret leakage in real-world production environments, we have obtained permission to scan seven internal code repositories of an IT company. Although the company has internal regulations explicitly prohibiting the inclusion of hard-coded secrets in its code repositories, we find all (7/7) repositories had leaked secrets.

To further investigate leaked secrets, as shown in Table 12, we classify them into four risk levels based on the potential impact of their exposure, and the proportion of highly sensitive (high-risk and critical-risk) secrets up to 77%. Due to ethical constraints, we cannot directly validate leaked secrets. Instead, to streamline the process, we prioritized critical-risk secrets and designed a disclosure questionnaire covering key aspects such as content, location, validity, and whether it originated from a test environment. Since all detected secrets were found in production, any identified test secrets were confirmed as reused. Moreover, we utilize tools like PwnedPasswords [29] to identify exposed and commonly used secrets and use the offline open-source estimator zxcvbn [77] to evaluate the strength of leaked secrets.

TABLE 11: Valid verification of real-world secrets.

Project	# Total (in Test-Env.)	# Valid (in Test-Env.)	# Strong T	# Strong F	# IsC T	# IsC F
Proj.1	23 (2)	9 (0)	19	4	4	19
Proj.2	11 (10)	6 (5)	9	2	2	9
Proj.3	11 (1)	5 (0)	10	1	1	10
Proj.4	10 (1)	4 (1)	9	1	1	9
Proj.5	25 (13)	6 (1)	19	6	5	20
Proj.6	17 (3)	8 (2)	14	3	2	15
Proj.7	7 (0)	0 (0)	6	1	1	6
All	104 (30)	38 (9)	86	18	16	88

Strong: Secrets rated Level 4 (strongest) by zxcvbn;
IsC: Common secrets in PwnedPasswords [29].

Results & Discussion. As shown in Table 11, feedback from seven repository owners indicates that 36.54% (38/104) of identified secrets are valid. Moreover, we find older versions of those repository leaked secrets as well. Of the valid secrets, 23.68% (9/38) are in test environments, while 76.32% are in production, highlighting that many developers overlook the importance of separating these environments.

Moreover, timely detection and response to newly leaked secrets are crucial, as these secrets may not yet be exploited but could lead to severe consequences if accessed by attackers. Weak and previously exposed secrets should also be addressed because they can be used for brute-force attacks. Our verification demonstrates that plenty of detected weak passwords are valid.

Additionally, ensuring secret security requires vigilance at every step since any weak link can result in vulnerabilities. Even though many developers use strong passwords, embedding these secrets directly in code still undermines their security. Notably, 82.69% of high-risk secrets are complex passwords with high strength, emphasizing that even credentials designed to be difficult to guess or brute-force provide no security once they are exposed.

7.4. Case Studies

This section presents three typical leakage cases from three platforms.

Bypassing IPTV Authentication. IPTV (Internet Protocol Television) [42] streams live and on-demand TV over Internet networks, enabling high-definition, lag-free viewing without traditional cables. However, attackers can sniff packets to obtain service secrets and bypass verification. Table 10 indicates that many .m3u files on GitHub contain potential secret leaks, often from projects designed to bypass IPTV services. Listing 1 provides a code snippet from

TABLE 12: Risk level descriptions and examples.

Risk Level	Ratio	Description	Example

I. Critical Risky	34%	If an attacker obtains this information, it can be directly exploited without context assessment. Examples include database addresses with credentials, SSH credentials, or commonly used public services.	`DBURL: jdbc:mysql://sql.a1***.com:3306/mydb`
			`Username: admin`
			`Password: P@ss****`

II. High Risky	43%	If an attacker obtains this information, it can be indirectly exploited without context assessment. Examples include private key certificates and p12 certificates.	`-----BEGIN CERTIFICATE-----`
			`MIIDXTCCAk.....(truncated)`
			`-----END CERTIFICATE-----`
			`Apple Push Service .p12: ***_cert.p12`

III. Mid Risky	14%	If an attacker obtains this information, a simple assessment of the code context is needed to determine potential further exploitation.	`API Key: XZA12-******`
			`An internal function call:`
			`connect_to_service(api_key)`

IV. Low Risky	9%	If an attacker obtains this information, it is generally difficult or impossible to exploit under normal circumstances.	`# Modify the parameters as needed.`
			`PASSWORD=gra******`
			`LOCAL_IP=0.0.0.0`

the `Tvlist-awesome-m3u-m3u8` repository, which includes .m3u files with secrets from over 8,000 IPTV addresses across 38 countries. This misuse violates providers' content rights, causing financial losses. Leaked secrets may enable attackers to "legally" exploit IPTV keys, leading to copyright infringement and potential DDoS attacks.

```
#Sample: Tvlist-awesome-m3u-m3u8
#EXTINF:-1 ,Ru*****jing
https://live.cig***.tv/live/hd**.m3u8?
    auth_key=52***39b
#EXTINF:-1 ,Ho***an (1080p)
http://5.2**.7*.3*:1***0/C**1/index.m3u8?
    token=kds***Io
```

Listing 1: Content snippet of the IPTV list on GitHub.

Secret in Log Files. In real-world scenarios, we discover secret leaks in database dumps (e.g., .sql files containing log data), confirmed by developers. Investigation shows that a vulnerability platform had scanned other companies for weak passwords and logged results in .sql files, which were overlooked, leading to exposure. Similar issues are found on PyPI, where .debug files contain secret-based authentication logs. For instance, as shown in Listing 2, the `exponent-server-sdk-async` package (versions 2.1.2 and 2.1.3) records push logs in debug.log, potentially exposing secrets due to failed network requests. We reported this issue to the developer, and they removed the secrets and cleared historical versions. Our analysis indicates that file types such as .debug, .error, .log, and .audit present varying degrees of risk, with 2,772 log files flagged as potential threats. It highlights the need to carefully monitor and sanitize log and result files generated during testing and runtime. These files may unknowingly store sensitive data and pose accidental security risks.

Leaked by 3rd Party Templates. With the growing popularity of MPs, many businesses use them for sales and promotion. Youzan [44] provides comprehensive e-commerce solutions, enabling businesses to build online stores, manage sales, and engage with customers efficiently. Merchants can create MPs on platforms like WeChat using Youzan's

```
#Sample: exponent_server_sdk_async/debug.log
2024-04-29 14:34:16.545 |DEBUG|
__main__:<module>:17 - Using Expo Token:
    OLEae8Zpg*****************pqdamTjJZBN-F
2024-04-29 14:39:33.811 |DEBUG|
__main__:<module>:17 - Using Expo Token:
    OLEae8Zpg*****************pqdamTjJZBN-F
```

Listing 2: Content snippet of the leaked log file on PyPI.

backend tools, simplifying template selection, content editing, and publishing. However, as shown in Listing 3, we discovered that Youzan's template MPs may use the same hard-coded client_id and client_secret pairs for backend API authorization. An attacker could exploit these secrets to obtain an access_token from Youzan's authorization API [47] and potentially perform sensitive operations [48], such as checking orders, retrieving customer addresses, and modifying product prices. This breach, stemming from leaked template secrets, not only poses significant security risks but also undermines platform trust. It potentially causes severe financial losses, reputational damage, and privacy violations for affected merchants, resulting from unauthorized orders and price manipulations.

```
"31XQ": function(e, t, n) {
    t.a = {
    clientId: "4d6******c3ed8",
    clientSecret: "1cd******2a",
    common: {
    yzLogo: "https://***.yzcdn.cn/.../.../***.
    png" } };
```

Listing 3: Code snippet of a Youzan template on WeChat.

8. Discussion

8.1. Ethics and Responsible Disclosure

Following ethical guidelines [65], all analyses were conducted locally, and no actual attacks were used to validate detected secrets. Moreover, we tried our best to conduct a responsible disclosure process for identifying secret leaks

in GitHub repositories, PyPI packages, and WeChat MPs. Similar to prior work [63], [81], we used public contact information to reach project owners, sending 3,906 disclosure emails total and receiving 205 responses as the paper writing. Notably, while most projects we examined were over a year old, secrets in 12 projects were still valid and posed active security risks, as confirmed by project owners. This finding underscores the ongoing threat of secret leaks in older projects and the need for continuous monitoring and proactive management. Through this disclosure process, we aim to mitigate immediate risks and encourage developers to adopt stronger secret management practices.

8.2. Lessons Learned

Through active discussions with developers, we identify three main issues contributing to secret leakage:

I. Human Error. Developers' poor practices and limited security awareness often lead to secret leaks. For example, 60.07% of confirmed leaks are due to manually hard-coded secrets, involving sensitive information like API keys and passwords. Additionally, secrets frequently appear in comments, bundled log files, or intermediate files added to projects (as seen in our case studies), leading to potential leakages.

II. Misuse of Production and Test Environments. Production environments hold high-validity secrets (76.32%) that are often inadvertently left in code, while testing environments frequently reuse these production secrets for convenience. Our data shows that 23.68% of leaks occur in testing settings, where secrets are often treated as low-risk and overlooked in detection. Temporary secrets or default credentials, like weak passwords (`12345678`), often persist longer than intended, creating security vulnerabilities.

III. Static Secret Implementation. Many service providers rely on static secret practices for access control, placing the security responsibility on developers. Although providers implement measures such as IP whitelisting, the safety of these static secrets largely hinges on the developers' security practices. If developers lack sufficient security awareness, such dependency could result in critical vulnerabilities. For example, as shown in our WeChat MPs case study, embedding secrets in production code significantly increases the risk of exposure and exploitation.

In conclusion, many developers lack awareness of secret management best practices. Vendors and platforms should enhance secret detection tools and provide clearer security guidance to help mitigate these issues.

8.3. Good Practices

To address secret leakage, we propose some good practices.

I. Avoiding Hard-coded Secrets. Developers should avoid embedding sensitive data in code directly. Pre-commit hooks and CI/CD secret detection can scan for sensitive information before committing and building. Using environment variables or secret management tools like AWS Secrets Manager [30] or HashiCorp Vault [32] ensures secrets remain outside the codebase. Moreover, regular code reviews are essential, especially in large teams.

II. Isolating Testing Environments. To avoid secret leakage, developers should ensure that testing environments use environment-specific secrets rather than production credentials. The automated removal of test secrets, along with the timely clearing of documentation and logs, helps ensure secure testing practices.

III. Better Provider Solutions. Service providers should adopt short-lived, auto-rotating secrets or enforce multi-factor authentication to minimize the implementation of static secrets. Strengthened one-time viewing for secrets and integrated secret management tools can reduce developer burden and improve security. Additionally, providers can implement an incident response system to inform developers of potential leaks on code platforms. For example, OpenAI notifies developers via email when API secrets are detected in public GitHub repositories [43].

8.4. Limitation

Unparsebale Files. Obfuscated secrets fall outside our metrics, as none of the tools we employ perform deobfuscation. As a purely static tool, KEYSENTINEL struggles to detect secrets in encrypted or heavily obfuscated text, code files, and binary files. Building on previous work [58], we use a dataset of 4,264 PowerShell scripts, which includes both obfuscated and their corresponding deobfuscated files, to evaluate KEYSENTINEL's detection capabilities for obfuscated files. In the deobfuscated files, KEYSENTINEL detected an additional 280 secrets. However, in their corresponding original obfuscated files, KEYSENTINEL identified only 48 out of the 280, resulting in a detection rate of approximately 17.14%. Thus, our experiments show that KEYSENTINEL can still effectively detect them when combined with companion deobfuscation tools.

Contextual Analysis. While KEYSENTINEL incorporates advanced extraction and filtering methods, it operates purely statically, without analyzing the broader context of the file where a secret is found. We also tried to use LLM models to detect secrets and found that the effect is also unsatisfactory, highlighting the challenges of accurately interpreting context. Moreover, KEYSENTINEL does not analyze extended contextual information to further assist in secret validation. Determining whether leaks remain active or revoked often requires more than the secret itself, relying on factors like type, usage conditions, and metadata. This process is highly complex and raises ethical concerns. Therefore, automated context-based secret detection remains an area for future exploration.

9. Conclusion

In conclusion, secret leakage remains a significant risk in software development, primarily due to hard-coded secrets like API keys and passwords. Current tools often

miss secrets or generate false positives due to simplistic pattern matching. Thus, we develop KEYSENTINEL, which integrates advanced techniques like semantic analysis to address these issues. It outperforms industry-standard tools, achieving a precision of 91.18%, a recall of 81.71%, and an F1 score of 0.86. We also conduct various, large-scale measurements cross three platforms. We discuss main reasons of secret leakage, such as human error and misusing testing environments, emphasizing the need for better secret management practices. Plus, service providers should adopt secure methods like dynamic secret generation to reduce the reliance on hard-coded secrets. The combined approach is crucial for mitigating secret leakage and enhancing overall security.

Acknowledgement

We thank anonymous reviewers for their valuable comments and suggestions. The authors from Southeast University were supported in part by the National Natural Science Foundation of China (Grant Nos. 62472092, 62172089, 62106045), Natural Science Foundation of Jiangsu Province under Grants No. BK20241751, Jiangsu Provincial Key Laboratory of Computer Networking Technology, Jiangsu Provincial Key Laboratory of Network and Information Security under Grants No. BM2003201, and Key Laboratory of Computer Network and Information Integration of Ministry of Education of China under Grants No. 93K-9, Nanjing Purple Mountain Laboratories. We also thank the Big Data Computing Center of SEU for providing computational resources. The author from Quancheng Lab was supported in part by the Taishan Scholars Program. All of the work was done at the QI-ANXIN Technology Research Institute.

References

[1] "Get started: PyTorch 2.0," https://pytorch.org/get-started/pytorch-2.0/, Accessed: 2023-07-07.

[2] "Gitguardian: content scan," https://api.gitguardian.com/docs#tag/Scan-Methods/operation/content_scan, Accessed: 2023-07-07.

[3] "GitHub: Repositories Ranking," https://gitstar-ranking.com/repositories, Accessed: 2023-07-15.

[4] "Go: ast," https://pkg.go.dev/go/ast, Accessed: 2023-07-18.

[5] "Go: parser," https://pkg.go.dev/go/parser, Accessed: 2023-07-18.

[6] "Npmjs: babel," https://www.npmjs.com/package/@babel/core, Accessed: 2023-07-18.

[7] "PyPI: AST," https://pypi.org/project/AST/, Accessed: 2023-07-18.

[8] "PyPI: javalang," https://pypi.org/project/javalang/, Accessed: 2023-07-18.

[9] "A New Approach to Application Security - Legit Security," https://www.legitsecurity.com/hubfs/Collateral/A%20New%20Approach%20to%20Application%20Security%20-%20Legit%20Security%20-%20v1.pdf, Accessed: 2024-09-23.

[10] "IBM: Cost of a Data Breach Report 2024," https://www.ibm.com/reports/data-breach, Accessed: 2024-09-23.

[11] "OpenAI: Rate limits," https://platform.openai.com/docs/guides/rate-limits, Accessed: 2024-10-07.

[12] "PyPI: jellyfish," https://pypi.org/project/jellyfish/, Accessed: 2024-10-07.

[13] "Python difflib: Helpers for computing deltas," https://docs.python.org/3/library/difflib.html, Accessed: 2024-10-07.

[14] "Remediate secrets incidents: Overview," https://docs.gitguardian.com/secrets-detection/remediate/overview, Accessed: 2024-10-07.

[15] "Abbreviations in code," https://github.com/abbrcode/abbreviations-in-code, Accessed: 2024-10-14.

[16] "David47k: top english wordlists," https://github.com/david47k/top-english-wordlists, Accessed: 2024-10-14.

[17] "Dwyl: english words," https://github.com/dwyl/english-words/tree/master, Accessed: 2024-10-14.

[18] "GitGuardian: state-of-secrets-sprawl-report-2024," https://www.gitguardian.com/state-of-secrets-sprawl-report-2024, Accessed: 2024-10-14.

[19] "GitHub: PassFinder," https://github.com/Aoa0/PassFinder, Accessed: 2024-10-14.

[20] "Machine learning acronyms and abbreviations," https://github.com/AgaMiko/machine-learning-acronyms, Accessed: 2024-10-14.

[21] "NLTK," https://www.nltk.org/, Accessed: 2024-10-14.

[22] "PyPI: textblob," https://pypi.org/project/textblob, Accessed: 2024-10-14.

[23] "PyPI: wordninja," https://pypi.org/project/wordninja, Accessed: 2024-10-14.

[24] "Gitleaks rule:generic-api-key," https://github.com/gitleaks/gitleaks/blob/master/config/gitleaks.toml, Accessed: 2024-10-15.

[25] "Trufflhog rule:generic-api-key," https://github.com/trufflesecurity/trufflehog/blob/34e443adcf1548e573a9d8f8496efb49b5d8a9c1/examples/generic.yml#L2, Accessed: 2024-10-15.

[26] "Ian Ahl: When AI Gets Hijacked," https://permiso.io/blog/exploiting-hosted-models, Accessed: 2024-10-21.

[27] "Internet Archive breached again through stolen access tokens," https://www.bleepingcomputer.com/news/security/internet-archive-breached-again-through-stolen-access-tokens/, Accessed: 2024-10-21.

[28] "Mobile App Supply Chain Vulnerabilities Could Endanger Sensitive Business Information," https://www.security.com/threat-intelligence/mobile-supply-chain-aws, Accessed: 2024-10-21.

[29] "PyPI: pwnedpasswords," https://pypi.org/project/pwnedpasswords/, Accessed: 2024-10-21.

[30] "Amazon AWS Secrets Manager," https://aws.amazon.com/secrets-manager/, Accessed: 2024-10-29.

[31] "Opencagedata guides: create a new api key," https://opencagedata.com/guides/how-to-create-a-new-api-key, Accessed: 2024-10-29.

[32] "Vault," https://www.vaultproject.io, Accessed: 2024-10-29.

[33] "2023 Active Adversary Report," https://news.sophos.com/en-us/2023/08/23/active-adversary-for-tech-leaders, Accessed: 2024-10-4.

[34] "Removing sensitive data from a repository," https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/removing-sensitive-data-from-a-repository, Accessed: 2024-10-4.

[35] "GitHub: auth0/repo-supervisor," https://github.com/auth0/Repo-supervisor, Accessed: 2024-10-5.

[36] "GitHub: awslabs/git-secrets," https://github.com/awslabs/git-secrets, Accessed: 2024-10-5.

[37] "GitHub: GitGuardian/ggshield," https://github.com/GitGuardian/Ggshield, Accessed: 2024-10-5.

[38] "GitHub: gitleaks/gitleaks," https://github.com/gitleaks/gitleaks, Accessed: 2024-10-5.

[39] "GitHub: Skyscanner/whispers," https://github.com/Skyscanner/whispers, Accessed: 2024-10-5.

[40] "GitHub: trufflesecurity/trufflehog," https://github.com/trufflesecurity/trufflehog, Accessed: 2024-10-5.

[41] "Glib: g_bytes_ref," https://docs.gtk.org/glib/method.Bytes.ref.html, Accessed: 2024-11-11.

[42] "IPTV: Internet Protocol Television," https://en.wikipedia.org/wiki/Internet_Protocol_television, Accessed: 2024-11-11.

[43] "OpenAI Help Center: Security and API Key Safety," https://help.openai.com/en/collections/3675944-security-and-api-key-safety, Accessed: 2024-11-11.

[44] "Youzan," https://ir.youzan.com/en, Accessed: 2024-11-11.

[45] "WeChat Mini-programs: View Layer," https://developers.weixin.qq.com/miniprogram/en/dev/framework/view/, Accessed: 2024-11-4.

[46] "Wikipedia: Entropy," https://en.wikipedia.org/wiki/Entropy_(information_theory), Accessed: 2024-11-4.

[47] "Youzan: Get the access_token," https://doc.youzanyun.com/resource/develop-guide/41355/49258, Accessed: 2024-11-4.

[48] "Youzan: the API document," https://doc.youzanyun.com/list/API/1286, Accessed: 2024-11-4.

[49] "PyPI: Pandas," https://pypi.org/project/pandas/, Accessed: 2024-11-6.

[50] "PyPI: PyYAML," https://pypi.org/project/PyYAML/, Accessed: 2024-11-6.

[51] "PyPI: xmltodict," https://pypi.org/project/xmltodict/, Accessed: 2024-11-6.

[52] "Python Docs: plistlib," https://docs.python.org/ja/3/library/plistlib.html, Accessed: 2024-11-6.

[53] "KellanZ/KEYSENTINEL: a secret detection tool." https://github.com/KellanZ/KEYSENTINEL/tree/main, Accessed: 2025-04-08.

[54] A. Alali, H. Kagdi, and J. I. Maletic, "What's a typical commit? a characterization of open source software repositories," in *2008 16th IEEE international conference on program comprehension*. IEEE, 2008, pp. 182–191.

[55] 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)*. IEEE, 2023, pp. 1–12.

[56] 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)*. IEEE, 2023, pp. 347–351.

[57] P. E. Black, "Ratcliff/obershelp pattern recognition," *Dictionary of algorithms and data structures*, vol. 17, 2004.

[58] H. Chai, L. Ying, H. Duan, and D. Zha, "Invoke-deobfuscation: Ast-based and semantics-preserving deobfuscation for powershell scripts," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2022, pp. 295–306.

[59] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[60] M. Dahlmanns, C. Sander, R. Decker, and K. Wehrle, "Secrets revealed in container images: an internet-wide study on occurrence and impact," in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, 2023, pp. 797–811.

[61] C. Farinella, A. Ahmed, and C. Watterson, "Git leaks: Boosting detection effectiveness through endpoint visibility," in *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2021, pp. 701–709.

[62] R. Feng, Z. Yan, S. Peng, and Y. Zhang, "Automated detection of password leakage from public github repositories," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 175–186.

[63] Y. Gu, L. Ying, Y. Pu, X. Hu, H. Chai, R. Wang, X. Gao, and H. Duan, "Investigating package related security threats in software registries," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1578–1595.

[64] S. Josefsson and S. Leonard, "Textual encodings of pkix, pkcs, and cms structures," Tech. Rep., 2015.

[65] E. Kenneally and D. Dittrich, "The menlo report: Ethical principles guiding information and communication technology research," *Available at SSRN 2445102*, 2012.

[66] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *biometrics*, pp. 159–074, 1977.

[67] Y. Li, H. Wang, and K. Sun, "A study of personal information in human-chosen passwords and its security implications," in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 2016, pp. 1–9.

[68] S. Lounici, M. Rosa, C. M. Negri, S. Trabelsi, and M. Önen, "Optimizing leak detection in open-source platforms with machine learning techniques." in *ICISSP*, 2021, pp. 145–159.

[69] E. J. Mascha and T. R. Vetter, "Significance, errors, power, and sample size: the blocking and tackling of statistics," *Anesthesia & Analgesia*, vol. 126, no. 2, pp. 691–698, 2018.

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

[71] 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, p. 59, 2022.

[72] 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. 168–075.

[73] C. Shen, T. Yu, H. Xu, G. Yang, and X. Guan, "User practice in password security: An empirical study of real-life passwords in the wild," *Computers & Security*, vol. 61, pp. 130–141, 2016.

[74] V. S. Sinha, D. Saha, P. Dhoolia, R. Padhye, and S. Mani, "Detecting and mitigating secret-key leaks in source code repositories," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 396–400.

[75] D. Wang, Q. Gu, X. Huang, and P. Wang, "Understanding human-chosen pins: characteristics, distribution and security," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 372–385.

[76] 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)*. IEEE, 2022, pp. 123–130.

[77] D. L. Wheeler, "zxcvbn:{Low-Budget} password strength estimation," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 157–073.

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

[79] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," *Advances in neural information processing systems*, vol. 28, 2015.

[80] Y. Zhang, Y. Yang, and Z. Lin, "Don't leak your keys: Understanding, measuring, and exploiting the appsecret leaks in mini-programs," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2411–2425.

[81] Z. Zhang, Q. Hou, L. Ying, W. Diao, Y. Gu, R. Li, S. Guo, and H. Duan, "Minicat: Understanding and detecting cross-page request forgery vulnerabilities in mini-programs," in *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[82] C. Zuo, Z. Lin, and Y. Zhang, "Why does your data leak? uncovering the data leakage in cloud from mobile apps," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1296–1310.

Appendix A.
Algorithm Descriptions for the TBS Filter (T4)

This section provides a detailed description of the T4, designed to analyze and assess the semantic meaning of secrets using tokenized word lists.

To analyze the semantics of secrets at a fine-grained level, we split them into tokens. Traditional tools like NLTK [21] and TextBlob [22] struggle with concatenated strings, while WordNinja [23] can effectively address this issue. However, WordNinja may incorrectly split abbreviations (e.g., splitting `auth` into `au` and `th`). We observed that abbreviations often appear connected by non-alphabetic symbols (e.g., `a_b_c`) and typically do not exceed five characters in length [20], [15]. Therefore, we first use regular expressions to split strings at non-alphabetic symbols, which helps preserve abbreviations. We then examine the length of the resulting tokens: if a token exceeds five characters, it may still be a concatenation of words, so we apply WordNinja to split it further. To reduce noise, we remove tokens shorter than three characters, as they are less likely to be meaningful words or abbreviations. The detailed algorithm is presented in Algorithm 2.

To perform semantic analysis, we tokenize detected secrets using Algorithm 2 and compare them against a pre-collected word list. Experiments show that a word-to-length ratio of over 35% generally signals meaningful content. Additionally, words longer than four characters are uncommon in highly random secrets, so we apply extra checks based on word and secret length. Further details are outlined in Algorithm 3.

Algorithm 2 Word Splitter

```
function SPLIT_WORD(secret_match, words_list)
    ▷ Split by punctuation
    tokens ← split(secret_match, '[._{}#$%&]')
    tokens ← filter(tokens, length ≥ 1)        ▷ Remove empty strings
    new_tokens ← ∅
    for all token in tokens do
        if token ∉ words_list and length(token) > 5 then
            ▷ Split long tokens
            split_tokens ← wordninja.split(token)
            ▷ Filter short split tokens
            split_tokens ← filter(split_tokens, length > 2)
            extend(new_tokens, split_tokens)
        else
            append(new_tokens, token)   ▷ Keep token if already valid
        end if
    end for
    return new_tokens
end function
```

Appendix B.
Secret Relationship

We assume two matched strings, α and β, with the following possible scenarios:

- **Case 1**: *α is a substring of β:* For each file, we traverse and analyze the relationships among the secrets, documenting those that qualify as substrings. Subsequently, we filter out these substrings to derive the final candidate secrets (❹ in Figure 1).
- **Case 2**: *α equals β:* In such a situation, we need to determine a unique secret at the same position. Since different types of secrets have different filtering properties, we cannot immediately decide which secret should be retained. The final decision will be determined during the process of merging similar secrets (❽ in Figure 1).
- **Case 3**: *α and β intersect:* In this situation, similarity algorithms can assess whether two secrets are similar, thereby determining if they represent the same secret. If they are indeed the same, this situation aligns with Case 2.
- **Case 4**: *α and β do not overlap*: In such a situation, all secrets are directly retained.

Algorithm 3 Tokenization-based Semantic Filter

```
function TBS_FILTER(secret, words_list, secret_fixed_prefixes)
    secret.word_weight ← 0
    have_meaning ← False
    ▷ Split the candidate secret to words list.
    words_list ← split_word(secret.value, words_list)
    more4 ← 0                    ▷ Counter of words with length > 4
    for all k in words_list do
        if k in secret_fixed_prefixes then
            continue                    ▷ Skip words within the secret
        end if
        secret.word_weight ← secret.word_weight + length(k)
        if length(k) > 4 then
            more4 ← more4 + 1
        end if
        if length(secret.value) ≤ 30 and more4 > 0 then
            have_meaning ← True
            return have_meaning
        else if 30 < length(secret.value) ≤ 60 and more4 > 1 then
            have_meaning ← True
            return have_meaning
        end if
        if secret.word_weight / length(secret.value) > 0.35 then
            have_meaning ← True
            return have_meaning
        end if
    end for
    return have_meaning
end function
```

Appendix C.
Detailed Results of Secret Detection on Different Platforms

To assess filter performance across platforms, we conducted ablation studies on GitHub, PyPI, and WeChat datasets by removing each filter and observing changes in precision, recall, and F1 scores. Table 13 shows the results and the F1 score changes (δF1) relative to the full system, highlighting each filter's contribution. Table 14 compares the precision, recall, and F1 scores of various secret detection tools on GitHub, PyPI, and WeChat, offering a concise cross-platform performance overview.

TABLE 13: Results of filter ablation experiments across different platforms.

Platform	Method	Precision	Recall	F1	δF1 ↓	Filter Objects
GitHub	**(w/o) T6: Prefix**	**50.64%**	**84.97%**	**0.6346**	**-0.2234**	**Unstructured secret**
	(w/o) T3: Seg-Entropy	66.58%	83.55%	0.7410	-0.1170	Machine-generated secret
	(w/o) T4: TBS	70.89%	81.60%	0.7587	-0.0993	Machine-generated secret
	(w/o) T5: PWD	80.97%	81.93%	0.8145	-0.0435	Human-chosen secret
	(w/o) T1: Str-assisted	83.97%	82.36%	0.8316	-0.0264	Structured file
	(w/o) T2: Pattern	87.47%	81.63%	0.8445	-0.0135	Machine-generated secret
	KEYSENTINEL	90.85%	81.29%	0.8580	0.0000	All
PyPI	**(w/o) T6: Prefix**	**28.87%**	**85.79%**	**0.4320**	**-0.4390**	**Unstructured secret**
	(w/o) T4: TBS	43.76%	82.92%	0.5729	-0.2981	Machine-generated secret
	(w/o) T3: Seg-Entropy	46.31%	82.47%	0.5931	-0.2779	Machine-generated secret
	(w/o) T5: PWD	82.13%	83.08%	0.8260	-0.0450	Human-chosen secret
	(w/o) T2: Pattern	84.76%	83.11%	0.8392	-0.0318	Machine-generated secret
	(w/o) T1: Str-assisted	87.68%	85.09%	0.8636	-0.0074	Structured file
	KEYSENTINEL	92.43%	82.35%	0.8710	0.0000	All
WeChat	**(w/o) T4: TBS**	**10.30%**	**79.69%**	**0.1824**	**-0.6828**	**Machine-generated secret**
	(w/o) T3: Seg-Entropy	49.51%	79.20%	0.6093	-0.2648	Machine-generated secret
	(w/o) T5: PWD	78.52%	78.91%	0.7871	-0.0781	Human-chosen secret
	(w/o) T6: Prefix	74.50%	94.56%	0.8334	-0.0318	Unstructured secret
	(w/o) T1: Str-assisted	90.64%	81.92%	0.8606	-0.0035	Structured file
	(w/o) T2: Pattern	95.03%	79.01%	0.8628	-0.0024	Machine-generated secret
	KEYSENTINEL	89.91%	83.38%	0.8652	0.0000	All

δF1 ↓: Difference in F1 score compared to KEYSENTINEL.

TABLE 14: Comparison of secret detection tools across different platforms.

Tool	Platform	Raw Output				Unique Secret Normalization			
		# Reported	Precision	Recall	F1	# Reported	Precision	Recall	F1
Ggshield	GitHub	2,376	74.07%	20.90%	0.3260	1,471	77.43%	26.91%	0.3994
	PyPI	133	2.56%	2.85%	0.0270↓	133	42.11%	3.55%	0.0655↓
	WeChat	17	76.47%	1.37%	0.0269↓	14	78.57%	1.64%	0.0322↓
Gitleaks	GitHub	7,618	61.29%	55.95%	0.5850	4,948	58.95%	63.93%	0.6134
	PyPI	10,143	16.69%	55.76%	0.2569↓	10,143	9.26%	59.79%	0.1603↓
	WeChat	9,331	9.06%	87.56%	0.1641↓	8,002	7.16%	85.52%	0.1321↓
TruffleHog	GitHub	48,994	1.68%	12.27%	0.0296	25,695	2.66%	14.40%	0.0449
	PyPI	3,940	6.17%	11.73%	0.0809↑	3,940	4.44%	13.95%	0.0674↑
	WeChat	62	20.97%	1.47%	0.0275↓	37	32.43%	1.94%	0.0366↓
Repo-supervisor	GitHub	54,451	0.35%	3.08%	0.0064	36,608	0.18%	1.41%	0.0032
	PyPI	95,083	0.05%	2.60%	0.0010↓	95,083	0.05%	2.88%	0.0010↓
	WeChat	488,852	0.07%	42.51%	0.0014↓	372,789	0.07%	38.96%	0.0014↓
Git-secret	GitHub	181	2.76%	0.06%	0.0012	98	23.47%	0.53%	0.0103
	PyPI	240	0.00%	0.00%	0.0000↓	240	3.75%	0.55%	0.0096↓
	WeChat	2	0.00%	0.00%	0.0000↓	1	0.00%	0.00%	0.0000↓
Whispers	GitHub	21,652	5.05%	22.55%	0.0826	5,066	16.52%	18.32%	0.1737
	PyPI	10,286	1.83%	6.25%	0.0283↓	10,286	1.23%	7.83%	0.0213↓
	WeChat	71	35.21%	2.64%	0.0492↓	63	34.92%	3.28%	0.0600↓
KEYSENTINEL	GitHub	6,784	**90.85%**	**81.29%**	**0.8580**	3,886	**89.37%**	**82.70%**	**0.8591**
	PyPI	2,749	**92.43%**	**82.35%**	**0.8710**↑	1,480	**90.74%**	**84.65%**	**0.8759**↑
	WeChat	902	**89.91%**	**83.38%**	**0.8652**↑	587	**92.72%**	**80.18%**	**0.8600**↑

Note: Arrows show performance changes vs. GitHub: ↑ improvement, ↓ decline.

Appendix D. Meta-Review

D.1. Summary

This paper proposes a new framework for leaked secret detection, which leverages a set of 910 carefully manually crafted regular expressions. A benchmark, consisting of 11K labeled secrets across 1.8M files from GitHub, PyPI, and WeChat, is developed and used to evaluate the framework, achieved an F1 score of 0.86, notably outperforming existing tools and LLMs like GPT-4 in precision and recall. Additionally, the authors conducted an ablation study and processed 80M files from public repositories, finding that 30% of repositories contain secrets.

D.2. Scientific Contributions

- Independent Confirmation of Important Results with Limited Prior Research
- Provides a New Data Set For Public Use
- Creates a New Tool to Enable Future Science
- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field

D.3. Reasons for Acceptance

1) A key-consideration in the acceptance of this work was the section comparing the paper's approach to common LLMs in terms of efficiency and detection performance. Several reviewers were convinced that this aspect, even though tangential to the paper itself, is the actual key contribution providing notable value to the community.
2) Beyond the comparison to LLMs, the paper presents a diligent with which it successfully raises the bar in terms of leaked secret detection capabilities. With that, it does not only provide a valuable step forward in an established field, but also contributes a new base-line against which future non deterministic secret detection methods (LLM, ML) can be benchmarked.
3) Beyond providing a base-line, the paper documents a benchmarking methodology which will be valuable in future work.

D.4. Noteworthy Concerns

1) Despite a generally positive perspective on the method, some reviewers raised concerns about the manual labeling in terms of reliability and scalability.
2) While the improvements on secret detection are noticeable, the general approach is the direct, yet work-intensive, application of established principles, i.e., the approach itself is considered not sufficiently novel by some reviewers.

Appendix E. Response to the Meta-Review

We sincerely thank the reviewers for their detailed and constructive comments. Below, we address their major concerns point by point.

Regarding Concern 1: We acknowledge that the dataset labeling process cannot be entirely free from manual involvement. However, we did not label secrets directly from the raw files by hand. Instead, we employed existing secret detection tools to label the extracted candidates, thereby reducing manual workload. To ensure a fair and comprehensive comparison in our experiments, we used the largest number of extraction methods (seven tools in total) to maximize the coverage of potential secrets. This approach also enables future researchers to further expand the dataset using newer tools.

Moreover, during the labeling process, we observed many recurring secret patterns (see Section 3.2). These can be leveraged to design automated filtering strategies based on pattern matching, further reducing the need for manual effort. Consistency checks between different labellers can also be used to assess the quality and reliability of the labeling results.

Regarding Concern 2: Our filtering method is not a simple aggregation of existing techniques; rather, it introduces novel strategies that have not been proposed or explored in prior work. As detailed in Section 4.4, we enhanced several existing techniques (T2, T3, T4, and T5) and proposed two new strategies (T1 and T6) to address critical challenges left unresolved by previous studies. Specifically, T1 is our original design to mitigate false positives caused by incomplete or truncated secret content, and T6 is another newly introduced strategy to reduce errors arising from prefix mismatches—both of which are first proposed in this work. As demonstrated by the ablation experiments in Section 6.1, these new strategies significantly improve filtering accuracy and represent key contributions beyond existing methods.