Secrets Unlocked: Evaluating LLMs for Secrets Detection in Android Apps

MARCO ALECCI, JORDAN SAMHI, TEGAWENDÉ F. BISSYANDÉ, JACQUES KLEIN

Mobile apps frequently embed sensitive secrets, such as API keys, access tokens, client secrets, and private keys that support internal functionality or enable integration with external systems and third-party services. Developers frequently embed these secrets into Android apps, which allows attackers to extract them through reverse engineering. Once exposed, attackers can exploit them to access sensitive data, manipulate resources, or abuse APIs, resulting in severe security and potential financial risks.

In this paper, we present the first large-scale empirical evidence that off-the-shelf large language models (LLMs) can automatically identify secrets in Android apps without any domain-specific prior knowledge, thereby substantially lowering the barrier for attackers. On a benchmark of 5135 Android apps from prior work, LLMs rediscovered 93% of previously known secrets and identified 4361 additional valid credentials (+195%). Extending our analysis to 50 000 Google Play apps collected between August and October 2025, we conducted the largest-scale study to date on secret detection in Android apps, identifying secrets in 17 590 apps (35%). Among the 18 908 detected secrets, 1802 remained active at discovery, including, among others, critical credentials such as Stripe payment keys, OpenAI API keys, and GitHub personal access tokens. We responsibly contacted all the affected developers, of whom 170 confirmed the issues and updated their apps accordingly.

Our findings empirically demonstrate the reality of vibe hacking: anyone can now leverage publicly accessible AI models to perform complex offensive security tasks with minimal expertise.

ACM Reference Format:
Marco Alecci, Jordan Samhi, Tegawendé F. Bissyandé, Jacques Klein. 2026. Secrets Unlocked: Evaluating LLMs for Secrets Detection in Android Apps. 1, 1 (April 2026), 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Mobile application (app) developers must implement diverse functionalities, including data storage, authentication, advertising, and payment processing, that often require integration with external systems and third-party services. To enable these integrations, developers embed secrets such as API keys, access tokens, client secrets, and private keys directly into their apps. However, a pervasive and well-documented problem is that developers frequently embed these secrets into Android apps, making them vulnerable to extraction through reverse engineering [1–3]. The consequences of exposed secrets are severe: adversaries can gain unauthorized access to sensitive data, manipulate or corrupt system resources, incur substantial financial costs through API abuse (e.g., a leaked Google Cloud API key leading to over $55 000 in unauthorized charges [4]), or trigger denial-of-service (DoS) conditions by exhausting rate limits [1]. This weakness pattern, formally classified as "CWE-798: Use of Hard-coded Credentials" [5], remains critical enough to appear in the 2024 CWE Top 25 Most Dangerous Software Weaknesses [6]. Our empirical analysis confirms that this issue has concrete, real-world consequences: we uncovered *numerous active*, high-impact secrets such as payment credentials, developer tokens, and AI API keys, embedded in publicly distributed Android apps.

Author's Contact Information: Marco Alecci, Jordan Samhi, Tegawendé F. Bissyandé, Jacques Klein.

ACM XXXX-XXXX/2026/4-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Historically, extracting such secrets at scale required attackers to master specialized tools, handcrafted rules, or domain-specific knowledge, placing these activities largely in the hands of experienced attackers or security analysts. Recent advances in large language models (LLMs), however, challenge this assumption: unlike traditional analysis techniques, LLMs may reason about code and embedded values in a flexible, context-aware manner, without relying on handcrafted rules or task-specific training for secret detection in Android apps. This raises an important security question: *could general-purpose LLMs, without any task-specific tuning, be capable of identifying secrets in real-world Android apps?*

In this paper, we present the first large-scale empirical evaluation of this phenomenon. Rather than proposing a new detection technique, we aim to measure how easily existing, publicly accessible LLMs can be misused for large-scale secret discovery, even by non-experts. This attacker-centric perspective reveals a critical emerging risk: modern LLMs may dramatically lower the technical barrier to performing large-scale secret harvesting. Anyone with a public LLM and basic scripting skills can automate such operations.

We first evaluated LLMs on the benchmark dataset of 5135 Android apps from Li *et al.* [1], which represents the current state-of-the-art in secret detection research for Android apps. Our analysis showed that LLMs successfully rediscovered **93%** of all previously reported secrets and uncovered an additional **4361** valid credentials, achieving an improvement of over **195%** compared to prior existing tools based on handcrafted rules and/or static analysis. Indeed, in addition to lowering the technical barrier to secret harvesting, LLMs identified secrets missed by existing tools, including several OpenAI keys (many of which remain active) and Base64-encoded credentials. These findings demonstrate the ability of LLMs to detect high-impact secrets in diverse formats without relying on predefined rules or signatures. Subsequently, we extended our analysis to a new dataset comprising 50 000 apps collected from Google Play between August and October 2025, representing, to the best of our knowledge, the largest study to date on secret exposure in Android apps. LLMs identified secrets in **17 590 apps (35%)** of this corpus. Among the 18 908 we detected, **1802** of credentials were verified as still active, including keys granting access to payment services (e.g., Stripe), developer platforms (e.g., Github personal access tokens), and AI APIs (e.g., OpenAI keys). We implemented a responsible disclosure process and notified all affected developers, of whom **170** confirmed the issues and subsequently acted accordingly.

Our findings underscore the **dual-use nature** of LLMs in software security. On the one hand, these models can serve as powerful tools for defensive auditing, automating large-scale vulnerability assessments with minimal effort [7–11]. On the other hand, this same capability could be exploited by malicious actors who could automate large-scale secret extraction with minimal technical effort, relying on general off-the-shelf LLMs. This highlights the urgent need for secure-by-design development practices, responsible AI usage, and improved secret management mechanisms.

Contributions. We make the following key contributions:

- We present the first large-scale evaluation of off-the-shelf LLMs for detecting secrets in Android apps.
- On the benchmark dataset of 5135 apps from prior work, LLMs rediscovered **93%** of previously known secrets and uncovered **4361** additional valid credentials, achieving a **195%** improvement over state-of-the-art tools.
- We further evaluated LLMs on a new dataset of **50 000 apps** collected from Google Play, constituting, to the best of our knowledge, the largest study to date on secret exposure in Android apps. LLMs identified secrets in **17 590 apps (35%)**, including **1802 active credentials** from major third-party services.

- We conducted a responsible disclosure process, notifying affected developers and tracking remediation outcomes to ensure the ethical handling of all findings. **170** of them already acknowledged the findings.

2 Background and Existing Work on Secret Detection in Android Apps

In this section, we outline the key concepts underlying secret exposure in mobile apps and Android app analysis, and summarize prior work on secret detection in Android Apps.

2.1 Background

Secrets in Mobile Apps. *Secrets* are sensitive tokens or credentials, such as API keys, client IDs, access tokens, or private keys, that enable authentication, authorize access to resources, and support integration with third-party services in mobile apps. Apps rely on these secrets to identify themselves to backend systems, regulate access permissions, and enforce billing or usage policies. If exposed, adversaries can misuse them to: ① gain unauthorized access to sensitive data, ② corrupt or manipulate resources, ③ generate unexpected financial costs by invoking paid APIs, or ④ exhaust invocation limits, leading to denial-of-service (DoS) [1, 5]. In this paper, we focus specifically on *app secrets*, which authenticate the app itself or its associated backend services. These are different from *user-sensitive data*, such as contacts or photos, which are usually targeted through runtime data exfiltration [3] and are outside the scope of this paper.

Android App Analysis. Android apps are distributed as compiled APK files. An APK is a zip-compressed binary archive containing compiled bytecode, resources, and metadata. Although the original source code is not directly available, reverse-engineering tools enable recovery of human-readable representations of code and assets [12–16]. Because these tools are widely accessible, any secret hardcoded in an app can be extracted once the APK is public. This dual-use reality motivates our investigation into how general-purpose LLMs can accelerate or automate the harvesting of secrets at scale.

2.2 Existing Work on Secret Detection

While most existing tools/studies for secret detection focus broadly on code and code-sharing platforms, Li *et al.* [1] recently conducted a systematic categorization of secret-detection approaches applicable to Android apps, covering more than ten existing tools [2, 17–28]. The approaches were classified into three main families of techniques: ① *intrinsic-value-based analysis*, which relies on properties such as entropy and regular expression (regex) matching; ② *static analysis*, which traces data flows through APIs; and ③ *machine-learning-based approaches*, which classify candidate strings based on contextual features.

The authors then tested one approach from each category (specifically, Meli *et al.* [17] for intrinsic-value analysis, `LeakScope` [2] for static analysis, and `PassFinder` [24] for machine-learning-based analysis) on a dataset of 5135 Android apps, discovering 2142 secrets affecting 2115 different apps [1]. While each family of detection techniques has its own advantages, they all share a critical limitation: they require knowing exactly what to look for in advance. Regex-based detectors [17] rely on carefully crafted patterns (e.g., Google API keys with fixed structure "`AIza[0-9A-Za-z_]35`"), static analysis tools [2] depend on predefined API signatures and encoded rules, and machine-learning methods [24] require labeled training data that may not capture novel or obfuscated secrets. As a result, these tools are fundamentally constrained to detecting known or previously characterized credential types. As briefly discussed in Section 1, LLMs can also help overcome some of these limitations, potentially identifying secrets even without predefined patterns or API signatures.

3 Experimental Setup

In this section, we present our experimental setup outlining our methodology, implementation details, and the datasets involved in the experiments.

3.1 Methodology

Our evaluation aims to measure the extent to which off-the-shelf LLMs can automatically identify and classify embedded secrets in Android apps from an attacker's perspective. Prior work has shown that naively submitting the entire contents of an APK to LLMs does not scale due to token limits, latency, and the monetary cost of large-scale API queries [29–31]. Consequently, we developed a two-module LLM-based pipeline focusing on two main string-bearing sources, thereby offering a cost-effective yet high-yield strategy for large-scale secret harvesting. Module A exposes the app's resource strings to the model by parsing `strings.xml` (i.e., the centralized Android resource file that stores UI, configuration, and other strings referenced by the app) and preserving the XML structure and tag semantics to provide contextual cues. Module B extracts string literals from the bytecode and presents them to the model together with the full bytecode of the class where the string literal is found. We emphasize that our focus on these two elements is methodological and attacker-centric: we simulate large-scale, automated secret harvesting that a realistic adversary could perform at low cost, without an exhaustive inspection of every file. A comprehensive, file-by-file analysis of all APK contents may reveal additional secrets and is left for future work, as further discussed in Section 6. Moreover, to cope with scalability, each module operates through two sequential phases: *identification* and *labeling*. These allow the model to first localize candidate secrets and then assign appropriate specific labels (e.g., "OPEN API KEY"). This two-step decomposition strikes a balance between precision and scalability: within each module, Phase 1 rapidly surfaces candidates at low cost, while Phase 2 refines these results using richer contextual information to reduce errors and hallucinations.

In total, our pipeline has four phases:A_1, A_2, B_1, and B_2, described in detail below. An overview of the experimental pipeline is shown in Figure 1.

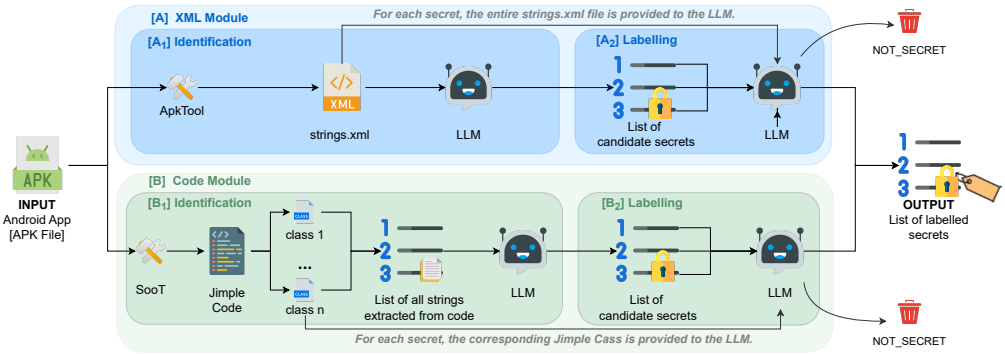


Fig. 1. Overview of the LLM-based experimental pipeline for detecting embedded secrets in Android apps.

[A_1] XML Strings Identification. In the first phase, we extract the app's `strings.xml` file using *ApkTool* [16]. To test whether the LLM can identify sensitive information from such context, the content of the entire XML file is provided as input, and the model is asked to return a list of candidate secrets[1]. The candidate list produced by this phase is passed to the subsequent labeling phase for per-candidate verification and labeling.

[1]All prompts used in this work are available in our released artifacts.

[A_2] XML Strings Labeling. Each candidate string identified in Phase A_1 is reanalyzed individually by the LLM, using the full XML file as supporting context. The LLM is instructed to focus exclusively on the current candidate string and to freely assign a descriptive label indicating the presumed type or provider of the secret (for example, *Google API key*, *OpenAI API key*, or *RSA private key*). This reflects the attacker's lack of prior knowledge about secret formats. Providing the entire XML file again enables the LLM to focus on the string in its original structural and semantic context, thereby mitigating erroneous attributions from the first pass. Indeed, the LLM may also return the special label `NOT_SECRET` when, based on the contextual evidence, it determines that the candidate is not a secret. The output of A_2 is a list of labeled secrets extracted from XML resources.
[B_1] Code String Identification. Using *Soot*, we systematically extract all string literals from the code along with their corresponding class identifiers (as this context will later be used in Phase B_2). We selected the *Soot* framework since it is a robust and widely adopted framework for analyzing Android app bytecode, which relies on the Jimple intermediate representation [15]. The full set of extracted strings is then provided to the LLM to identify potential secrets. Importantly, in B_1 we intentionally supply no additional contextual information about a string's origin or usage, and we analyze all the strings collectively rather than individually. This design simulates an attacker aiming to rapidly harvest secrets across many apps with minimal per-app inspection; the rationale is further discussed hereafter.
[B_2] Code String Labeling. Each candidate secret identified in Phase B_1 is then analyzed individually by the LLM for labeling. Along with the candidate string, the entire Jimple class from which it was extracted is provided as context, allowing the model to analyze the textual and semantic characteristics of each candidate and assign an appropriate label. The LLM may also return the special label `NOT_SECRET` when, based on the surrounding code context, it determines that the string is not a secret. This re-evaluation phase mitigates potential errors and hallucinations from Phase B_1, yielding a refined list of labeled secrets extracted from the app's codebase.
Considerations about Phase [B_1]. Phase B_1 processes all extracted strings collectively and without contextual information, while Phase B_2 examines a smaller subset with full class-level context. This design reflects an attacker-centric perspective, prioritizing scalability and efficiency over completeness. From an adversarial point of view, conducting large-scale secret scanning across thousands of apps requires rapid screening rather than deep semantic analysis. Consequently, B_1 serves as a coarse-grained filtering step capable of highlighting high-risk candidates with minimal computational overhead, while B_2 refines these results through context-aware inspection. To evaluate the impact of this trade-off, we also tested a variant in which each string was analyzed individually, together with its corresponding Jimple class, already in Phase B_1. While this approach slightly improved detection in rare cases, it drastically increased runtime and cost. Therefore, we adopt the baseline configuration for the remainder of the experiments, reflecting the attacker's perspective.
LLM Choice. For the LLM components of our pipeline, we primarily employ `gpt-4o-mini` [32] by OpenAI, chosen for its favorable balance of accuracy, inference speed, and cost efficiency. The impact of model selection is analyzed in RQ3 (Section 4.3), where we compare the proprietary `gpt-4o-mini` with six different open-source LLMs: `gpt-oss-20b` [33], `deepseek-r1-32b` [34], `qwen3-30b` [35], `gemma3-27b` [36], `phi3-14b` [37], and `llama3.1-8b` [38]. All open-source models were executed locally on an NVIDIA RTX5000 ADA GPU via the `ollama` [39] deployment framework. For each model family, we selected the largest variant (in terms of parameter count) that could be run within our available hardware resources.

3.2 Datasets

Our evaluation is based on two complementary datasets: ❶ a benchmark dataset from prior literature [1], and ❷ a newly collected dataset of 50 000 Android apps from Google Play. To avoid

repeatedly referring to datasets with long descriptive names, we assign symbolic notations to each dataset used in our evaluation.

❶ **Benchmark Dataset (D_b).** We denote by D_b the benchmark dataset introduced by Li et al. [1], which includes 5135 Android apps collected from Google Play (covering the top 200 apps in each category). According to Li et al., the benchmark contains 2142 valid checked-in secrets distributed across 2115 apps (approximately 41% of the total). These results were obtained by comparing three representative detection families: regex-based analysis (Meli et al. [17]), static analysis (`LeakScope` [2]), and machine-learning-based analysis (`PassFinder` [24]). Notably, the machine-learning approach failed to detect any secrets; therefore, only the first two techniques contributed to the reported results [1]. Table 1 summarizes the distribution of confirmed secret types in the benchmark.

Method	Google	Twitter	Facebook	Twilio	Total	Union
Regex-based	664	0	0	2	666	2142
LeakScope	1816	10	10	0	1836	

Table 1. Distribution of confirmed checked-in secrets in the benchmark dataset [1].

Within this benchmark, we distinguish two complementary subsets: D_{bs}, the subset of 2115 apps containing at least one confirmed checked-in secret; and D_{bns}, the complementary subset of 3020 apps with no confirmed secrets, such that $D_{bs} \cup D_{bns} = D_b$.

❷ **New Dataset (D_{new}).** The second dataset, denoted as D_{new}, consists of 50 000 Android apps newly collected from Google Play between August and October 2025, in collaboration with the maintainers of AndroZoo [40, 41], who allowed us to use their crawlers to retrieve apps from Google Play during this period. This dataset provides a realistic and up-to-date view of current development practices, as it includes production apps that were indeed present on Google Play during this period. As an additional note, because the collection occurred well after the training cutoff of `gpt-4o-mini` (October 1, 2023 [32]), it is likely that none of these apps' versions would have been accessible to the model during pretraining. To the best of our knowledge, this represents the largest study to date on hardcoded secret detection in Android apps, in terms of the number of analyzed apps. For ethical reasons, the complete list of analyzed apps is not publicly released; however, it can be made available upon request to authenticated researchers to support reproducibility and responsible research use.

4 Experimental Results

In this section, we present the results of our empirical evaluation of LLMs for secret detection in Android apps. We organize our analysis around the following research questions (RQs). The first three RQs are evaluated using the benchmark dataset D_b:

- **RQ1:** How effectively can LLMs detect secrets in Android apps compared to existing approaches?
- **RQ2:** How accurately can LLMs categorize or label the identified secrets?
- **RQ3:** How does the choice of the underlying LLM affect performance on secret detection?

The remaining two RQs focus on a new set of real-world apps from Google Play (D_{new}):

- **RQ4:** What types of secrets can LLMs identify in a new large dataset of real-world Android apps from Google Play, and what does this reveal about the current state of secret exposure in the wild?
- **RQ5:** How many of the identified secrets are still active (i.e., still exploitable by attackers) at the moment of discovery?

4.1 RQ1: Comparison with Existing Approaches.

To address the first research question, we evaluated the effectiveness of LLMs in identifying secrets compared to existing automated approaches. We ran our LLM-based detection pipeline on the

dataset D_{bs}, where secrets previously detected by regex-based tools or by LeakScope were manually confirmed by the authors. A total of 56 apps could not be analyzed due to issues with Soot or ApkTool and were therefore excluded from the analysis. Figure 2 summarizes the overlap between secrets detected by the LLM and those identified by existing tools.

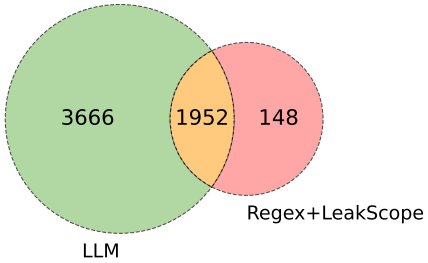


Fig. 2. Comparison between secrets detected by existing tools and by the LLM-based pipeline.

Overall, the LLM rediscovered the vast majority of secrets previously identified by existing tools (1952), covering approximately 93% of all confirmed secrets in the benchmark. Notably, the LLM also uncovered a substantial number of additional secrets (3666) that were missed by prior tools.

Given the large number of additional secrets detected by the LLM, making manual validation impractical, we employed two complementary verification strategies to estimate the reliability of these additional results: ❶ manual inspection of a statistically significant random subset, and ❷ validation using regex patterns as confirmation tools (i.e., regex were not used to rediscover secrets but to check whether the new results produced by the LLM conformed to known credential formats).

❶ Manual Inspection. We performed a manual inspection by decompiling and analyzing the code of the app, a commonly adopted technique in related literature [1, 42, 43]. We used a statistically significant random sample (using a 95% confidence level and a 10% margin of error) of 94 secrets drawn from the 3666 "extra" secrets exclusively detected by the LLM (i.e., the green portion of Figure 2).

Our findings can be summarized as follows:

(1) Out of the 94 inspected samples, 76 (81%) were confirmed as valid secrets.
(2) The majority of valid secrets corresponded to API keys and tokens for third-party services. Several new categories emerged that were not covered by the original benchmark, including OpenAI, Razorpay, LeanPlum, Kakao, and Mapbox API keys. Some of these were particularly sensitive, such as a GitHub Personal Access Token found in one app.
(3) Other secret types not tied to a specific service were identified, such as RSA private keys, JWT tokens, database credentials, and even an Android keystore password used for signing APKs.
(4) One noteworthy case involved a Google API key that was Base64-encoded in the app as "`QUl6YV******...`", which decodes to "`AIza**...`". This simple encoding evaded traditional regex-based detectors; yet, the LLM correctly inferred its sensitive nature from contextual cues (i.e., the LLM automatically recognized that the string was base64 encoded and decoded it without being explicitly instructed to do so).
(5) Among the false positives, 12 (13%) corresponded to benign values that resembled secrets but were public, such as Razorpay test keys, Mapbox public tokens, RSA public keys, or public identifiers.
(6) The remaining 6 (6%) false positives were cases where the LLM misclassified variable names (e.g., "`password`" or "`access_token`") as actual credentials. Interestingly, while these were not real secrets, their surrounding code often revealed additional security weaknesses. In one case, examining the context of a misclassified "`password`" variable led to the discovery of multiple

vulnerabilities, including SQL injection, plaintext password storage, and hardcoded credentials in a `resetDatabase()` method. These observations on false positives and their implications are discussed further in Section 5.

In summary, manual inspection confirmed that the majority of the additional secrets detected by the LLM were genuine, validating the model's ability to generalize beyond predefined regex patterns and to identify previously unseen or obfuscated credentials. From an attacker's perspective, this demonstrates that LLMs can autonomously uncover new categories of secrets without prior knowledge of their formats, thereby substantially lowering the technical barrier to large-scale secret harvesting.

❷ **Validation Using Regular Expressions.** After manually validating a statistically significant random sample of the LLM-discovered secrets, we also decided to use an automated method to confirm their authenticity. To this end, we decided to use regex over the 3666 additional secrets identified by the LLM (the green area in Figure 2). The idea is to use regex purely as a confirmation tool over the LLM results (i.e., to verify whether the LLM-discovered secrets are genuine), not as a method for discovering new secrets. We first used the regex patterns defined by Li et al. [1] to determine how many of the LLM's findings matched known credential formats (e.g., Google or AWS keys). Regex with ambiguous or poorly defined patterns (e.g., Facebook or Twitter tokens) [1] were excluded as suggested by the study of Li et al. [1]. Additionally, since several secrets that we confirmed during manual validation exhibited well-structured formats (e.g., OpenAI, Razorpay), we defined new regex patterns for these services and applied them as a further confirmation step. Once again, this process was not intended to rediscover secrets but to verify that the newly identified cases (missed by existing tools) conformed to known credential formats. For instance, we found one OpenAI secret during our manual analysis and wanted to check whether others had also been identified by the LLM. After decoding Base64-encoded strings identified by the LLM, we confirmed a few additional keys hidden using this trivial obfuscation method. The results are summarized in Table 2.

Existing Regex (from Li et al.)		**New Regex (from Manual Validation)**	
Service	**Confirmed Secrets**	**Service**	**Confirmed Secrets**
Google API Key	500 (+5)*	OpenAI API Key	3
Google OAuth ID	1072	Razorpay (Live) API Key	6
Stripe Standard API Key	2	RSA Private Key	2
Amazon AWS Access Key ID	2	JWT Token	20
		LeanPlum API Key	4
		Kakao API Key	6
		Mapbox API Key	4
Total [Existing + New]			**1626**

*Indicates keys decoded from Base64.

Table 2. Secrets confirmed via regex verification of LLM-identified results over D_{bs}.

As shown in Table 2, many additional findings were validated through existing regex rules, particularly those related to Google services. This pattern aligns with Li et al.'s observations, where most detected credentials belonged to Google ecosystems [1] (in their study 99% of the confirmed secrets were Google-related). In total, we validated 1621 (44%) secrets through regex-based confirmation.

These results (both from manual validation and regex-confirmation) indicate that a significant portion of the additional LLM-detected secrets correspond to valid credentials that existing automated tools failed to capture. Regex-based validation, however, only applies to secrets following known structural patterns, reinforcing the complementary role of LLMs in this context.

Considerations about Google-secrets missed by existing tools. While the existing tools tested by Li et al. have the technical capability to detect Google credentials, they still missed a significant number of them. Citing their work: "LeakScope is capable of extracting checked-in secrets even if they are stored as environment variables in files such as XML or JSON" [1]. This means that,

technically, LeakScope should have been able to detect them. However, regarding the approach by Meli et al. [17] (the regex-based one), citing Li et al.: "For the string extraction, the source code is not directly accessible in APK files, we used Soot, which is capable of identifying all the variables of type java.lang.String to extract all the strings within the APK files [1]. Although the authors explicitly mention "all the strings within the APK files", their Soot-based extraction pipeline is not described in sufficient detail to determine exactly how resource-defined strings were processed. Small differences in how such strings are extracted or represented could help explain why we were able to detect additional Google-related secrets.

LLM Detection in Previously Unflagged Apps. Li et al. [1] reported valid secrets only in 2115 (D_{bs}) out of 5135 apps. To verify whether LLMs could detect overlooked credentials, we ran our LLM-based pipeline on the remaining 3020 apps previously classified as "clean" (D_{bns}). The LLM-based pipeline detected 3416 secrets in 2170 of these apps. Regex confirmation (using both the original and new validation patterns derived from manual inspection) validated a large number of these findings, 2735 (80%), summarized in Table 3.

Existing Regex (from Li et al.)		New Regex (from Manual Validation)	
Service	**Confirmed Secrets**	**Service**	**Confirmed Secrets**
Google API Key	2050 (+3)*	OpenAI API Key	2
Google OAuth ID	646	Razorpay (Live) API Key	7
PayPal Braintree	1	JWT Token	17
AWS Access Key ID	1	LeanPlum API Key	2
		Kakao API Key	2
		Mapbox API Key	4
Total [Existing + New]			**2735**

*Indicates keys decoded from Base64.

Table 3. Secrets confirmed via regex verification of LLM-identified results over D_{bns}.

Final Considerations. In total, the LLM identified at least 4361 additional valid secrets across D_b ($D_{bs} \cup D_{bns}$), representing a 195% improvement over state-of-the-art tools reported by Li et al. [1]. It is important to emphasize that this figure should be interpreted as a conservative lower bound. Indeed, due to the high volume of predictions and the wide variety of secret types, it was infeasible to manually verify all newly detected secrets. Instead, we validated a statistically significant subset, providing strong evidence of accuracy, although complete coverage cannot be guaranteed. Consequently, the true number of valid secrets discovered by the LLM is likely even higher. Our analysis also shows that a non-negligible fraction of predictions are false positives, including benign values resembling secrets or public identifiers. From an attacker's perspective, however, a modest false-positive rate is likely tolerable, as large-scale automated exploitation remains feasible. The broader implications and strategies for handling such false positives are discussed in Section 5.

These findings underscore a concerning trend: the large-scale discovery of valid secrets can now be performed with minimal effort and no specialized security knowledge, simply by leveraging a publicly available LLM. In practical terms, the technical barrier for effective secret mining has dropped substantially.

Answer to RQ1: The LLM rediscovered 93% of previously known secrets and uncovered at least 4361 new valid ones (195% more than prior tools). It also detected new secret categories and mildly obfuscated keys, highlighting both the strengths of LLMs in generalizing beyond fixed patterns and their potential misuse for large-scale secret harvesting.

4.2 RQ2: Accuracy of LLM Assigned Labels.

Beyond identifying potential secrets, our LLM-based pipeline also assigns a label to each detected secret, indicating its type or the service to which it most likely belongs. To evaluate the accuracy

of these labels, we analyzed the labels generated by the LLM-based pipeline for the 2115 apps with confirmed secrets (D_{bs}). To do so, we adopted two complementary validation strategies, similar to those used in RQ1: ❶ comparing the LLM-generated labels against known categories in D_{bs}, and ❷ manually inspecting a statistically significant random sample of results.

❶ **Validation over known secrets.** In D_{bs}, each secret is associated with a category derived from the matching rule that identified it (e.g., a regex or API signature). These categories do not represent semantic annotations per se, but rather the source of the detection pattern (e.g., "Google API Key" indicates that the secret was identified using the regex for "Google API Keys"). For this analysis, we compared those pattern-derived categories with the semantic labels produced by the LLM. Specifically, we examined whether the model's labels were consistent with the reference categories. The comparison results are summarized in Figure 3.

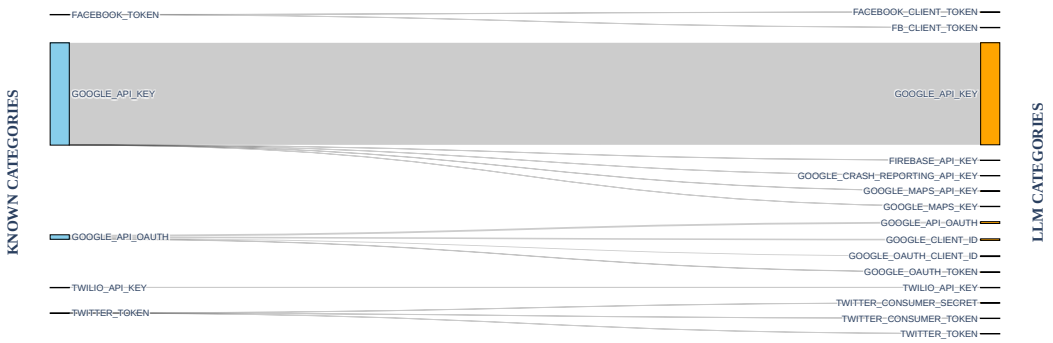
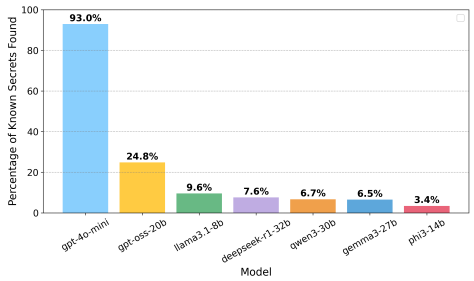


Fig. 3. Mapping between D_{bs} known categories and the semantic labels generated by the LLM.

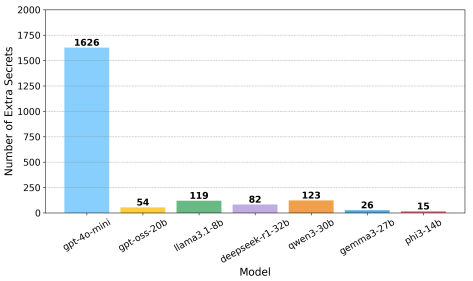
As observed, the labels assigned by the LLM were largely consistent with the benchmark categories. In several cases, the model generated more fine-grained or context-aware labels than the original pattern-based annotations. For example, while regex-based methods only identified that a credential belonged to the broader "Google Services" group, the LLM could often specify the precise service (e.g, Google Maps). From an attacker's standpoint, such granularity may facilitate targeted misuse, as it enables the identification of the specific platform or service associated with each credential. In a few instances, the model produced semantically equivalent label variants, i.e., slightly different formulations for similar tokens (e.g., Facebook/Twitter). However, these do not significantly impact the attacker's ability to identify the underlying service or secret type.

❷ **Manual Inspection.** To complement the benchmark-based validation, we manually inspected a random sample of 94 detected items (95% confidence level, 10% margin of error) drawn once again from the 3666 "extra" secrets exclusively detected by the LLM (i.e., the green portion of Figure 2). Note that this sample of secrets is different from the one used in RQ1. Out of the 94 samples, we observed the following outcomes:

- **15 cases (16%)** were false positives, i.e., non-secret strings incorrectly labeled as secrets (e.g., a Razorpay test key labeled as `RAZORPAY_API_KEY`).
- **3 cases (3%)** were confirmed secrets but misclassified (e.g., a Google OAuth ID labeled as a Google API key).
- **2 cases (2%)** were valid secrets but assigned overly generic labels. For instance, a StableDiffusion API key was labeled simply as `API_KEY`, while a Baron Weather API credential was labeled as `PRIVATE_KEY`.



(a) **Detection Recall.** Percentage of known secrets correctly identified in D_{bs}.



(b) **Extra Secrets.** Number of additional valid secrets detected over D_{bs} and verified by regex.

Fig. 4. Comparison of the performance of different LLMs across D_{bs}.

- The remaining **74 cases (79%)** were correctly identified and accurately labeled. In these cases, the LLM's outputs often captured the specific service or credential type, providing information that could directly aid an attacker in understanding the nature and potential use of the exposed secret.

Overall, the vast majority of labels were accurate, with most errors attributable to false positives (where a non-secret string was labeled as a secret) and a smaller fraction to misclassifications or overly generic categories. From an attacker's perspective, the ability of an LLM to accurately infer the type or service associated with a credential substantially increases the value of detected secrets. Even when the model's predictions are not perfectly normalized, the semantic cues embedded in the generated labels (e.g., explicit service names or token types) could help an adversary prioritize which secrets are most exploitable.

Answer to RQ2: LLMs assigned labels that were largely accurate and often more specific than pattern-based categories. This semantic precision helps not only in identifying secrets but also in understanding their context, enabling more targeted exploitation for an attacker.

4.3 RQ3: Impact of the Underlying LLM Choice.

So far, to address the first two RQs, we have relied on `gpt-4o-mini` by OpenAI. In this RQ, we compare the performance of `gpt-4o-mini` against six open-source LLMs (see Section 3.1) by running our LLM-based pipeline on the same benchmark dataset D_{bs} used previously.

We then evaluated each model using two complementary metrics: ❶ the percentage of known secrets in D_{bs} successfully identified, and ❷ the number of additional valid secrets detected in D_{bs}, as verified through regex-based validation (using the same set of regex from RQ1). The results are summarized in Figure 4, which presents both the detection recall for known secrets (Figure 4a) and the number of additional secrets confirmed through regex validation (Figure 4b).

As shown in Figure 4, the performance gap between proprietary and open-source models is significant. The proprietary `gpt-4o-mini` identified 93% of the known secrets in D_{bs}, whereas the best-performing open-source alternative, `gpt-oss-20b`, achieved only 25%. Among the open-source models, `gpt-oss-20b` consistently outperformed the others, which exhibited similarly low recall. A comparable trend is observed when considering the number of additional secrets confirmed through regex validation: `gpt-4o-mini` detected 1626 additional valid secrets, compared to only 54 for `gpt-oss-20b`.

Based on these results, we decided to rely on `gpt-4o-mini`. While it is true that other proprietary models, and even more powerful ones from OpenAI itself, could be evaluated, we believe that `gpt-4o-mini` offers the best trade-off between performance and cost, with the analysis of a single app costing less than $0.01. From an attacker's perspective, such affordability and public availability make `gpt-4o-mini` a practical choice for large-scale secret harvesting, as it combines strong detection capabilities with minimal infrastructure requirements and no need for fine-tuning. Since our primary goal is to assess the capabilities of LLMs for this specific task, rather than to benchmark every available model, we consider the performance of `gpt-4o-mini` to be sufficiently strong to support our study and provide meaningful insights into both the potential and misuse implications of LLM-based secret detection. At the same time, an adversary could still rely on off-the-shelf open-source models like `gpt-oss-20b` to perform large-scale scanning with moderate success (and could potentially achieve better results with more powerful hardware), demonstrating that secret detection does not require any specialized or custom-trained models.

Answer to RQ3: The choice of LLM has a major impact on detection performance. `gpt-4o-mini` far outperforms current open-source alternatives, which recover only a small fraction of known secrets and detect far fewer new ones. This makes `gpt-4o-mini` a practical option for large-scale secret harvesting.

4.4 RQ4: LLM Over a New Set of Android Apps.

So far, we have evaluated the capabilities of LLMs using the benchmark dataset D_b, which contains apps with previously confirmed secrets. In this research question, we extend the analysis by using our LLM-based pipeline over a new dataset D_{new} composed of 50 000 "fresh" Android apps recently crawled from Google Play.

A total of 857 apps could not have been analyzed due to Soot or ApkTool errors and were excluded from the study. In the remaining apps, the LLM detected 27 561 secrets in 17 590 of the 49 143 analyzed apps (corresponding to 35% of the dataset). The LLM assigned a total of 1486 distinct labels, even if, as discussed in RQ2, many likely refer to semantically similar or overlapping secret types. Given the extensive number of detected secrets and the large diversity of assigned labels, performing an exhaustive manual validation was infeasible. Therefore, as in RQ1, we again apply a two-step validation process to estimate the reliability of the detections: ❶ manual inspection of a statistically significant sample, and ❷ regex-based confirmation of the overall corpus.

❶ **Manual Inspection.** We manually inspected a statistically significant random sample of 96 secrets (95% confidence level, 10% margin of error) drawn from the 27 561 secrets returned by the LLM. Our findings can be summarized as follows:

(1) Out of the 96 inspected samples, 74 (77%) were confirmed as valid secrets.
(2) The nature of the false positives was the same as observed in RQ1, i.e., they corresponded either to benign values (e.g., public tokens or keys) or to identifiers and variable names (e.g., "token") mislabeled as real secrets.
(3) As in RQ1, the majority of valid secrets corresponded to API keys and tokens for third-party services. During this analysis, additional categories and providers emerged, including Alibaba, Sentry Auth Tokens, Mixpanel, and Bugsnag. We also found an OpenAI key with a different format from the one observed in RQ1 ("sk-proj-**…"). This further highlights how regex-based systems cannot keep pace with the evolving nature of credential formats.
(4) Other secret types not tied to a specific service were also identified, such as JWT tokens or passwords for different services. We report two particularly interesting examples. First, we found a hardcoded string which was used to initialize an AES `SecretKeySpec`. In practice, this

means that the app employed this fixed, hardcoded value as its encryption key, causing all encryption and decryption operations to rely on the same static key embedded in the code. As a result, anyone who extracts the app can easily recover the key and decrypt all associated data. Second, we found a plaintext password (the corresponding username was also hardcoded in the code) used in the app's POST body to a custom API endpoint *https://api.****.com/token*. Anyone who decompiles the APK could use these credentials to obtain access tokens and impersonate the legitimate app or account. Notably, none of these examples follows any specific or recognizable pattern, underscoring the limitations of existing detectors.

❷ **Validation Using Regular Expressions.** Following the same procedure as in RQ1, we used regex only as a confirmation tool, applying known and newly defined patterns to the 27 561 LLM-detected secrets to estimate the validity of detections. The results are summarized in Table 4.

Existing Regex (from Li et al.)		New Regex (from Manual Validation)	
Service	**Confirmed Secrets**	**Service**	**Confirmed Secrets**
Google API Key	15044 (+28)*	OpenAI API Key	17
Google OAuth ID	3602 (+2)*	Razorpay (Live) API Key	37
Stripe Standard API Key	5	RSA Private Key	5
Amazon AWS Access Key ID	23	JWT Token	99
MailChimp	1 (+1)*	LeanPlum API Key	3
		Kakao API Key	15
		Mapbox API Key	26
Total [Existing + New]			**18908**

*Indicates keys decoded from Base64.

Table 4. Secrets confirmed via regex verification of LLM-identified results over D_{new}.

As shown in Table 4, a substantial portion of the secrets identified by the LLM correspond to real, structurally valid credentials. The distribution closely mirrors that observed in RQ1, with Google-related keys dominating the confirmed cases, along with other major services. From an attacker's standpoint, however, these same findings demonstrate that mass secret harvesting remains both feasible and profitable: a simple pipeline, powered by an off-the-shelf LLM, can automatically extract thousands of valid credentials with minimal effort.

As was the case for RQ1 (see Section 4.1), it is important to note that the number of confirmed secrets is likely a conservative lower bound on the true number of embedded secrets present in the analyzed apps (D_{new}). Indeed, due to the high volume of predictions and the wide variety of secret types, it was infeasible to manually verify all newly detected secrets, leading us to analyze a statistically significant random sample. Moreover, confirming what was observed in RQ1, some false positives may still exist. We discuss these limitations and their implications in Section 5. Finally, given the potential security impact of exposed credentials, we nevertheless proceed with responsible disclosure to developers.

Disclosure to Developers Given the possible implications of our findings, we decided to contact the app developers to report the identified vulnerabilities. For safety and ethical reasons, the exact credential strings were not included in our communications; instead, developers were informed of the presence of a potentially embedded secret in their application. We also provided information on how an attacker could exploit these secrets and recommended actions to prevent such issues. Additionally, we asked the developers to kindly confirm whether what we had detected was indeed an embedded secret. We considered all 17 590 apps in which we found at least one secret. Even though we could not validate all of these secrets, we believed it was better to act cautiously and still advise developers to review their apps. From these apps, we successfully retrieved the email addresses from Google Play for 15 587 of them, to which we sent automated emails in October 2025. A total of 3387 emails were undeliverable. Two weeks after sending the emails, we received 1704 automated replies with messages such as "opening a ticket", "we have taken your request into

account", or similar AI-generated responses. We then received a total of 441 non-automated replies, whose details can be found in Table 5. 170 developers confirmed the presence of the LLM-identified secrets. 71 either tightened their configurations or removed and revoked the secrets.

Overall, the responses illustrate that developers generally appreciated the notification and, in many cases, promptly remediated the issues.

Category	Count [%]
Forwarded to the dev team / said they would check (no further reply)	*248 [56.2%]*
Developers stated that the finding was not a secret or not sensitive	*23 [5.2%]*
Total who confirmed the presence of the identified secret	*170 [38.5%]*
Already properly secured after inspecting the rules or configurations	*70 [15.9%]*
Fixed thanks to our disclosure (keys revoked and removed)	*46 [10.4%]*
Tightened rules or configurations (e.g., Firebase rules)	*25 [5.7%]*
Did not intend to fix (e.g., app to be decommissioned or lack of priority)	*17 [3.9%]*
Asked us to fix it on their behalf / proposed collaboration	*7 [1.6%]*
The secret was present in a previous version and has now been fixed	*5 [1.1%]*
Total Non-Automated Replies	**441 [100%]**

Table 5. Summary of developer responses.

Answer to RQ4: Across the 50 000 apps in D_{new}, the LLM found hardcoded secrets in a substantial fraction of cases, with many confirmed via regex. This shows that numerous production apps still embed exploitable secrets, making large-scale harvesting feasible. All affected developers were notified through responsible disclosure.

4.5 RQ5: Identified Secrets Still Active.

In RQ4, we confirmed the presence of secrets through both manual and regex-based validation across the 50 000 apps comprising D_{new}. Since these apps were available on Google Play between August and October 2025, we aim to investigate whether any of the detected credentials remain *active* in production systems. This verification was conducted in October 2025, prior to the disclosure to developers process previously described in RQ4.

To determine whether some of the LLM-identified secrets were still active, we employed two complementary approaches: ❶ using TruffleHog [20], a widely used open-source framework that verifies the validity of exposed credentials, and ❷ performing an empirical, request-based check on Google Maps for Google API keys, which will be described in detail hereafter.

4.5.1 TruffleHog validation. `TruffleHog` [20] is a widely used open-source tool that can identify exposed secrets and verify whether they are still active across multiple platforms, including version control systems (e.g., Git). For secret detection, it relies on regex and entropy-based heuristics, which limit its coverage to a subset of popular services, similar to other existing approaches.

For the verification phase, `TruffleHog` performs a lightweight validation by issuing a single, non-intrusive API request per detected credential, targeting a static identity or token-introspection endpoint (e.g., `GET /user` for GitHub). This process does not alter any external state or perform privileged operations; it only checks whether the secret is still accepted by the corresponding service. From an ethical standpoint, this approach minimizes interaction with third-party infrastructures and ensures that only passive, read-only checks are performed, aligning with responsible disclosure and research practices, as seen in prior work [1, 44]. It is important to emphasize that, in our experiment, `TruffleHog` was not used to *discover* new secrets. Instead, it was used exclusively to *verify* whether the credentials previously identified by our LLM-based pipeline (similarly to the regex-based validation) in D_{new} were still active, i.e., to emphasize the extent of the problem and show that attackers can still exploit them.

Table 6 summarizes the validation results. A secret is labeled as *active* when `TruffleHog` successfully confirms that it remains functional. Naturally, `TruffleHog` cannot validate all possible services detected by the LLM, as its verification modules cover only a subset of popular platforms. Therefore, the results presented here should be interpreted as a practical but partial view of the current status of the LLM-detected secrets.

Table 6. Number of LLM-detected secrets in D_{new} confirmed as active by TruffleHog.

Service	#Active	Service	#Active	Service	#Active	Service	#Active
Alibaba	82	GitHub	12	HubSpot API Key	3	LocationIQ	1
PubNub Publish Key	81	Twilio	8	Mailchimp	2	OpenWeather	1
RazorPay	28	Stripe	5	Slack	2	SendGrid	1
OpenAI	16	HuggingFace	4	Klaviyo	2		
PubNub Subscription Key	16	Flutterwave	4	Anthropic	2		
AWS	16	Azure Storage	3	Replicate	2		
						Total	**293**

Overall, 293 credentials were confirmed as still active, indicating that a substantial fraction of exposed secrets discovered by the LLM remained immediately exploitable at the time of analysis. Several of these correspond to access keys for cloud or AI platforms (e.g., AWS, OpenAI, ...), implying possible unauthorized access to data, compute resources, or communication channels.

After Vulnerability Disclosure. Following our vulnerability disclosure process we reran TruffleHog two weeks after sending the notifications (as aligned with RQ4), and observed that the number of active secrets decreased to 241. This suggests that many of the developers we contacted have taken action in response to our notification, even if they did not explicitly confirm their remediation by replying to our email.

4.5.2 Google API keys validation. Since TruffleHog is not capable of validating Google API keys, i.e., the most frequent type of secret found in our experiments, we followed the same strategy adopted by Li *et al.* [1] to assess their validity. In particular, for each secret matching the pattern of a Google API key, we sent a request to the Google Maps service, as this test does not involve any sensitive information. Note that not all the keys we tested are necessarily able to access the Google Maps API; this experiment merely serves as an example to estimate how many unrestricted keys could be detected. To ensure that our tests did not cause any significant financial impact, each checked-in key was used only once to send a request to https://maps.googleapis.com/maps/api/geocode/json, which returns a JSON object representing a specific city. The results are summarized in Table 7.

Message	Count [%]
Success	1509 [10.0%]
This API project is not authorized to use this API.	11754 [78.0%]
This API key is not authorized to use this service or API.	660 [4.4%]
You must enable Billing on the Google Cloud Project at ...	470 [3.1%]
This IP, site, or mobile app is not authorized to use this API key.	465 [3.1%]
The provided API key is invalid.	169 [1.1%]
The provided API key is expired.	45 [0.3%]
Total	**15072 [100%]**

Table 7. Responses from Google Maps Geocoding API when testing Google API keys.

As shown, 1509 API keys (10%) were unrestricted and allowed anyone in possession of the key to access the Google Maps API, potentially causing the legitimate owner to incur financial costs. This does not imply that the remaining keys are invalid or non-functional; rather, they may simply be restricted to other Google services or APIs that were not tested in this context. We intentionally limited our validation to the Google Maps API because its use does not expose or interact with any

sensitive data, ensuring an ethical and non-intrusive verification process. Consequently, additional unrestricted keys may exist across other Google-related services beyond those evaluated in this experiment.

> **Answer to RQ5:** Across both TruffleHog and Google API key checks, we identified 1802 active secrets before disclosure. Although this is a lower bound (many additional active secrets were likely present but could not be verified), these residual secrets nevertheless represent valuable entry points for potential exploitation.

5 Discussion

Several important considerations arise from our results.

Large-scale secrets harvesting. The LLM-driven approach we evaluated is inherently scalable and well-suited for adversarial, large-scale secrets harvesting in Android apps. It is straightforward to deploy and automate: there is no need to manually craft (and constantly update) extensive lists of regex or API signatures, and no requirement for specific fine-tuning. An attacker would need only basic scripting skills to pipe decompiled strings into an off-the-shelf generic LLM and retrieve secrets that can be exploited. Moreover, LLMs can detect interesting credential types, including slightly obfuscated ones (for example, base64-encoded), that existing rule-based scanners often miss. This capability was reflected during our vulnerability disclosure interactions: a developer confirmed a database password, a custom API key (common to all installations) used to authenticate requests to our backend, and SendGrid credentials. These would likely have been missed by rule-based scanners because they don't follow any specific pattern. The simplicity, generality, and automation-friendly nature of the approach, therefore, lowers the bar for mass secret harvesting and heightens the urgency of secure secret storage.

Exploitability of secrets and attacker objectives. Some secrets are highly sensitive and easily exploitable, such as GitHub personal access tokens or OpenAI API keys. For instance, an attacker could use a compromised OpenAI key (or any other LLM-related key) to perform further LLM-based secret harvesting at zero cost to themselves while causing the key owner to incur financial charges. Conversely, certain secrets, notably Google API keys, are often treated as "semi-public" by developers and documentation, but their real-world safety depends entirely on backend configuration and access controls. As also shown by our analysis of Google API keys, these keys are not always properly restricted, and misconfigured projects can turn a seemingly low-sensitivity key into a severe vulnerability. During our disclosure process, developers reported that they reviewed their Firebase configurations/rules after being notified; this highlights two points: ① developers (especially less experienced ones) may assume a key is harmless until notified, and ② an attacker could still exploit these secrets if the project/key is not properly configured. Similar observations were reported by Li et al. [1].

More broadly, what constitutes an exploitable secret, or a "false positive", depends mostly on the attacker's objectives. An adversary can easily adapt the LLM prompt to focus only on high-impact and easily exploitable credentials (e.g., OpenAI or cloud service keys), effectively trading recall for precision. At the same time, strings that may appear non-sensitive, such as advertising or analytics identifiers (especially if not properly configured), can still enable indirect abuse, such as generating fake traffic or triggering fraudulent events, potentially causing reputational harm to the app owner.

False positives can still be useful for attackers. Even when an extracted string cannot be directly exploited, false positives are far from useless in an adversarial setting. A string flagged as a "secret" can guide an attacker toward a narrow region of code worthy of deeper manual reverse engineering. In our manual evaluation for RQ1, we observed a case where a misclassified string (e.g., a variable named `password`) led us to a Jimple class containing SQL injection primitives and plaintext password handling. For automated attacker pipelines, a modest false-positive rate

is also acceptable: attempts to validate or abuse candidate secrets are typically cheap and highly parallelizable, making large-scale exploitation feasible even in the presence of noise.

6 Limitations and Threats to Validity

In this section, we discuss the main limitations of our approach, focusing on both technical constraints and methodological challenges.

String Extraction Methodology. Our LLM-based pipeline relies on two primary sources of string data: values defined in `strings.xml` and hardcoded string constants extracted from the app's code. This design captures the vast majority of embedded credentials, but some limitations remain.

- ① *Dynamic string construction.* Strings composed at runtime (e.g., via concatenation or `StringBuilder`) are not retrieved, since we focus on statically defined constants. This is a common limitation of static analysis.
- ② *String obfuscation.* Our study does not handle secrets hidden via complex encryption or custom encodings. While identifier renaming is fairly common, prior work (e.g., Dong et al. [45]) indicates explicit string encryption remains rare in real-world Android apps (0% of apps they analyzed from Google Play and 0.01% of apps from third-party markets). Our approach did handle simple obfuscation attempts, such as Base64.
- ③ *Partial file coverage.* Secrets can also appear in other app resources: configuration files, the manifest, or embedded JSON objects, that were excluded from our current setup. We deliberately limited our scope to the most common and structurally consistent sources to enable large-scale, uniform evaluation. Prior work shows that APKs contain far more than code and XML (e.g., more than 1,000 distinct file types) [44, 46]. Consequently, secrets may reside outside the string-based sources we analyze.

From an attacker's perspective, these restrictions do not substantially reduce feasibility. Our study aims to assess large-scale, automated secret harvesting using minimal effort and off-the-shelf LLMs. A determined adversary could always increase extraction depth (for example, by parsing additional files). The current setup (including the prompts and LLMs used), therefore, represents a lower-bound estimate of what a motivated attacker could achieve with comparable effort. Systematically addressing the above gaps is left for future work.

LLM-Related Challenges. LLMs can occasionally misclassify or hallucinate, resulting in both false positives and false negatives. These imperfections stem from the inherent limitations of current models rather than our experimental design. From an attacker's standpoint, however, such noise is rarely prohibitive; as discussed in Section 5, false positives can still provide useful leads. Moreover, attackers could readily refine prompts or employ more powerful LLMs to potentially enhance performance. In contrast, our primary objective was to evaluate the baseline capabilities of LLMs for secret detection, rather than to optimize performance through prompt or model engineering. A potential threat to our study's validity is training-data leakage: some of the 5135 apps in D_b may appear in the corpora used to train commercial LLMs. Although we cannot audit those datasets, we partially mitigated this risk by analyzing a separate set of 50k "fresh" apps that are less likely to be included in any training data. Some leakage may still exist; however, from an attacker's perspective, this does not reduce the practical effectiveness of our approach.

Recall Estimation. Measuring absolute recall remains challenging due to the absence of a definitive ground-truth dataset. The benchmark dataset B_s by Li et al. [1] offers the closest reference, yet our

analysis uncovered numerous additional secrets not present in that dataset. To mitigate this, we employed a two-step validation process, combining regex-based verification and manual review, to estimate detection quality, though some uncertainty inevitably remains. While this does not yield a perfect recall measure, it suffices for our purpose: to empirically characterize the extent to which LLMs, even without domain-specific tuning, can autonomously detect secrets in Android apps at scale.

7 Related Work

In this section, we review prior studies on secret detection in Android and generic software repositories, as well as related work on Android security analysis.

Secrets Detection in Mobile Apps. Beyond the work of Li et al. [1], discussed extensively in Section 2, Schmidt et al. [44] more recently conducted a large-scale, cross-platform study of secret exposure in mobile ecosystems, analyzing over 10 000 Android and iOS apps. Their analysis relied on an existing tool (`TruffleHog`) with minor extensions which, while effective, is inherently limited to a fixed set of predefined service patterns due to its regex-based design. In contrast, we directly evaluate an off-the-shelf, general-purpose LLM for secret detection, without fine-tuning or predefined rules, and conduct the largest study to date by analyzing 50 000 Android apps.

Secrets Detection in Code. As noted by Li et al. [1], most secret-detection tools were originally designed for generic source code rather than mobile apps. Prior work has largely focused on open-source repositories, particularly GitHub. Tools such as *Gitleaks* [18] and *TruffleHog* [20] scan Git histories using regular expressions and entropy heuristics to detect credentials like AWS and Google API keys. Commercial tools, including *GitGuardian's ggshield* [22] and *SpectralOps* [47], extend these approaches with large pattern libraries and ML-assisted detection. Basak et al. [42] showed that such regex-based methods generally suffer from low precision and recall. Academic work complements these efforts: Meli et al. [17] proposed a regex-based framework for leaked keys, Feng et al.'s *PassFinder* [24] leveraged contextual embeddings for password detection, Wen et al. [25] used reinforcement learning, and El Yadmani et al. [48] studied secret leakage in cloud APIs. Despite these advances, existing approaches rely on prior knowledge—such as predefined patterns, known API formats, or labeled data. In contrast, we evaluate whether a general-purpose LLM can infer likely secrets without any prior knowledge, from an attacker's perspective.

8 Conclusion

This work presented an empirical evaluation of LLMs for detecting hardcoded secrets in Android apps. We find that even straightforward, off-the-shelf LLMs, without any task-specific tuning, are capable of accurately identifying exposed credentials. These models not only match but often surpass traditional pattern- or rule-based tools, successfully uncovering both known and previously unseen secret types hidden in real-world apps. In our large-scale analysis, we validated thousands of exposed secrets, a significant portion of which remained active and potentially exploitable at the time of discovery.

From an attacker's perspective, this capability drastically lowers the barrier for large-scale, automated secret harvesting across Android apps. Anyone with minimal scripting skills can leverage a publicly available LLM to perform such extraction at scale. These findings underscore the need for enhanced secret management practices, stricter build-time safeguards, and secure-by-design development pipelines to prevent the exposure and subsequent exploitation of secrets.

Data Availability

Due to the sensitive nature of the experiments (and the presence of exploitable secrets), access to all relevant artifacts is available to *authenticated researchers* upon request with appropriate justification to support further research and replication. [2].

[2]For the purpose of peer review, we have provided reviewers with full access to all experimental artifacts at: https://anonymous.4open.science/r/LLMforAndroidSecrets-777

References

[1] K. Li, L. Ling, J. Yang, and L. Wei, "Automatically detecting checked-in secrets in android apps: How far are we?" 2024. [Online]. Available: https://arxiv.org/abs/2412.10922

[2] 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.

[3] L. Wei, H. Huang, S.-C. Cheung, and K. Li, "How far are app secrets from being stolen? a case study on android," *Empirical Software Engineering*, vol. 30, no. 3, p. 90, 2025.

[4] R. Singh, "Cs student receives $55,444 google cloud bill after api key exposed on github," *Lapaas Voice*, Dec. 2025. [Online]. Available: https://voice.lapaas.com/cs-student-google-cloud-bill-55444-api-key-leak-2025/

[5] MITRE Corporation / CWE, "Cwe-798: Use of hard-coded credentials," https://cwe.mitre.org/data/definitions/798.html, Sep. 2025, accessed: 29 September 2025.

[6] The MITRE Corporation, "2024 CWE top 25 most dangerous software weaknesses," https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html, MITRE, 2024, accessed: 2025-10-24.

[7] Z. Sheng, Z. Chen, S. Gu, H. Huang, G. Gu, and J. Huang, "Llms in software security: a survey of vulnerability detection techniques and insights," *ACM Computing Surveys*, 2025.

[8] S. M. Taghavi Far and F. Feyzi, "Large language models for software vulnerability detection: a guide for researchers on models, methods, techniques, datasets, and metrics," *International Journal of Information Security*, vol. 24, no. 2, p. 78, 2025.

[9] Y. Li, X. Li, H. Wu, M. Xu, Y. Zhang, X. Cheng, F. Xu, and S. Zhong, "Everything you wanted to know about llm-based vulnerability detection but were afraid to ask," *arXiv preprint arXiv:2504.13474*, 2025.

[10] J. Lin and D. Mohaisen, "From large to mammoth: A comparative evaluation of large language models in vulnerability detection," in *Proceedings of the 2025 Network and Distributed System Security Symposium (NDSS)*, 2025.

[11] N. S. Mathews, Y. Brus, Y. Aafer, M. Nagappan, and S. McIntosh, "Llbezpeky: Leveraging large language models for vulnerability detection," *arXiv preprint arXiv:2401.01269*, 2024.

[12] pxb1988, "dex2jar: Tools to work with android `.dex` and java `.class` files," 2025, accessed: 2025-10-01. [Online]. Available: https://github.com/pxb1988/dex2jar

[13] desnos et al., "Androguard: A full python tool to play with android files," 2025, accessed: 2025-10-01. [Online]. Available: https://github.com/androguard/androguard

[14] M. Project, "Mobile security framework (mobsf)," 2025, accessed: 2025-10-01. [Online]. Available: https://github.com/MobSF/Mobile-Security-Framework-MobSF

[15] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[16] "Apktool," https://apktool.org/, accessed: 29 September 2025.

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

[18] "gitleaks: Find secrets with gitleaks," https://github.com/gitleaks/gitleaks, Sep. 2025, accessed: 29 September 2025.

[19] okta-graveyard / repo-supervisor, "repo-supervisor: Scan your code for security misconfiguration, search for passwords and secrets," https://github.com/okta-graveyard/repo-supervisor, Sep. 2025, accessed: 29 September 2025.

[20] trufflesecurity / trufflehog, "trufflehog: Find, verify, and analyze leaked credentials," https://github.com/trufflesecurity/trufflehog, Sep. 2025, accessed: 29 September 2025.

[21] Skyscanner / whispers, "whispers: Identify hardcoded secrets in static structured text," https://github.com/Skyscanner/whispers, Sep. 2025, accessed: 29 September 2025.

[22] GitGuardian / ggshield, "ggshield: Detect and validate 500+ types of hardcoded secrets with advanced checks," https://github.com/GitGuardian/ggshield, Sep. 2025, accessed: 29 September 2025.

[23] I. Sasovets, "git-secrets-hunter: Tool that helps to detect sensitive information in public github repositories," Sep. 2025, accessed: 29 September 2025. [Online]. Available: https://github.com/IgorSasovets/git-secrets-hunter

[24] 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.

[25] 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.

[26] R. Han, H. Gong, S. Ma, J. Li, C. Xu, E. Bertino, S. Nepal, Z. Ma, and J. Ma, "A credential usage study: flow-aware leakage detection in open-source projects," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 722–734, 2023.

[27] A. Saha, T. Denning, V. Srikumar, and S. K. Kasera, "Secrets in source code: Reducing false positives using machine learning," in *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*. IEEE, 2020, pp. 168–175.

[28] 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.

[29] X. Qian, X. Zheng, Y. He, S. Yang, and L. Cavallaro, "Lamd: Context-driven android malware detection and classification with llms," *arXiv preprint arXiv:2502.13055*, 2025.

[30] M. Alecci, N. Sannier, M. Ceci, S. Abualhaija, J. Samhi, D. Bianculli, T. Bissyandé, and J. Klein, "Toward llm-driven gdpr compliance checking for android apps," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, 2025, pp. 606–610.
[31] T. Sun, M. Alecci, Y. Song, X. Tang, K. Kim, J. Samhi, T. F. d. A. BISSYANDE, and J. Klein, "Raml: Toward retrieval-augmented localization of malicious payloads in android apps," in *The 40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025*. IEEE/ACM, 2025.
[32] OpenAI, "GPT-4o mini model documentation," https://platform.openai.com/docs/models/gpt-4o-mini, 2024, accessed: 2025-10-24.
[33] OpenAI, "Introducing gpt-oss: Open-weight reasoning models," https://openai.com/index/introducing-gpt-oss/, 2025, accessed: 2025-11-13.
[34] DeepSeek-AI et al., "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," 2025. [Online]. Available: https://arxiv.org/abs/2501.12948
[35] A. Yang *et al.*, "Qwen3 technical report," 2025. [Online]. Available: https://arxiv.org/abs/2505.09388
[36] Gemma Team, "Gemma 3 technical report," 2025. [Online]. Available: https://arxiv.org/abs/2503.19786
[37] M. Abdin *et al.*, "Phi-3 technical report: A highly capable language model locally on your phone," 2024. [Online]. Available: https://arxiv.org/abs/2404.14219
[38] Meta AI, "Introducing llama 3.1," https://ai.meta.com/blog/meta-llama-3-1/, 2024, accessed: 2025-11-13.
[39] Ollama, "Ollama: Local llms made easy," https://ollama.com, 2025, accessed: 2025-10-30.
[40] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 468–471. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903508
[41] M. Alecci, P. J. R. Jiménez, K. Allix, T. F. Bissyandé, and J. Klein, "Androzoo: A retrospective with a glimpse into the future," in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 389–393.
[42] 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.
[43] 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.
[44] D. Schmidt, S. Schrittwieser, and E. Weippl, "Leaky apps: Large-scale analysis of secrets distributed in android and ios apps," 2025.
[45] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding android obfuscation techniques: A large-scale investigation in the wild," in *International conference on security and privacy in communication systems*. Springer, 2018, pp. 172–192.
[46] P. J. R. Jiménez, J. Samhi, T. F. Bissyandé, and J. Klein, "Dissecting apks from google play: Trends, insights and security implications," in *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2025, pp. 728–739.
[47] "Spectral: Software composition analysis with automated codebase security," https://spectralops.io/, n.d., accessed: 2025-10-10.
[48] S. El Yadmani, O. Gadyatskaya, and Y. Zhauniarovich, "The file that contained the keys has been removed: An empirical analysis of secret leaks in cloud buckets and responsible disclosure outcomes," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 3180–3198.