Can Exposed API Credentials Be Abused? Evaluating API Credential Protection Mechanisms in Android SDKs

Ziyi Zhou, Yikun Hu*, Dawu Gu*
Shanghai Jiao Tong University
Shanghai, China
{jou.dzyi, yikunh, dwgu}@sjtu.edu.cn
*Corresponding authors: Yikun Hu (yikunh@sjtu.edu.cn) and Dawu Gu (dwgu@sjtu.edu.cn)

***Abstract*—Android applications commonly integrate third-party SDKs to access cloud-backed services such as maps, advertising, authentication, and social sharing. These SDKs often require developer-provisioned API credentials, which may be exposed in app clients. Prior work mainly focuses on discovering exposed credentials, but credential exposure alone does not determine whether an attacker can abuse the protected API, because modern SDK backends may deploy additional protection mechanisms to authenticate the calling client, such as checking the package name, app signing certificate fingerprint, caller IP, or relying on remote attestation to verify the client identity.**

This paper presents APICredCheck, an automated framework for evaluating API credential protection mechanisms in Android SDKs. APICredCheck uses LLM-assisted screening to identify credential-dependent SDKs, statically analyzes legitimate SDK apps to extract credentials and client-identity evidence, and dynamically tests whether replacing these values in an attacker app enables unauthorized API invocation. In our study of the top 1,000 mobile SDKs, APICredCheck classifies 388 SDKs as involving Android API credentials. A manual audit of the top 50 SDKs in the same ranking yields a screening precision of 66.7% and a recall of 88.9%. Among the 18 SDKs manually confirmed to use API credentials, we obtain valid credentials for 15 and conduct dynamic evaluation using APICredCheck. APICredCheck successfully abuses 13 of these 15 SDKs by reusing extracted credentials and client-identity information. These results show that exposed API credentials remain practically abusable in many SDKs, even when additional protection mechanisms are deployed.

***Index Terms*—Android SDKs, API credentials, Client authentication, Authorization, Credential protection**

I. Introduction

Android applications (apps) increasingly rely on third-party SDKs to access cloud-backed services, such as maps, advertising, authentication, and social sharing. To use these services, developers often provision API credentials through SDK provider consoles and embed them into the app source code or configuration files such as the `AndroidManifest.xml` or `local.properties`. These credentials may take different forms, such as API keys [1], app secrets [2], client tokens [3], app identifiers [4] or other service-specific values. Because such credentials are needed by client-side SDKs during request construction, they are often observable in APK files or at runtime.

The exposure of such credentials has been widely recognized as a security concern in mobile apps [5]–[10] and other client-side software [11], [12]. Prior studies have also shown that secrets and API credentials can be recovered from source-code or compiled code, and have proposed various techniques for detecting such leakage [9], [13]–[17]. However, most existing work focuses on discovering exposed credentials or measuring the scale of leakage. This leaves an important question insufficiently answered: once an API credential is exposed, can an attacker actually abuse it from an unauthorized Android application?

Answering this question is nontrivial because modern SDK backends often do not rely on the credential alone. In addition to checking credential validity, SDK providers may authenticate the calling client using app-specific identity evidence, such as package names, signing certificate fingerprints, caller IP ranges, or remote-attestation-protected client identity information from mechanisms such as Play Integrity [18], DeviceCheck [19], or App Attest [20]. Under these protections, extracting an API credential from a legitimate application does not necessarily imply that an unauthorized Android app can successfully invoke the protected API.

At the same time, many checked identity values originate from the client side and may be observed or replaced at runtime. Therefore, the practical risk of exposed API credentials depends on whether the surrounding protections can resist such replacement and, if not, how much effort attackers need to bypass them.

In this paper, we study the abuse feasibility of exposed API credentials under a controlled attacker model. The attacker can build an Android app, referred to as the attacker app, that integrates the same SDK as a legitimate app but lacks the corresponding privileges. The attacker can analyze the legitimate app to recover API credentials and client-identity information, and can dynamically instrument the attacker app during execution. The goal is to determine whether the SDK backend can be tricked into accepting API calls issued from the attacker app.

To support this study, we present APICredCheck, an automated framework for evaluating API credential protection

mechanisms in Android SDKs. APICredCheck first uses a large language model (LLM) to screen top-ranked Android SDKs and identify those that use API credentials. For each selected SDK, we prepare our own legitimate app instead of using real-world apps for ethical reasons. APICredCheck then extracts credentials, client-identity data, and authentication-related evidence from the legitimate app, uses an LLM to infer methods that may handle such values, and finally validates whether a pre-built attacker app can invoke protected SDK APIs after runtime replacement.

Our evaluation shows that exposed API credentials remain practically abusable in many Android SDKs even when additional protection mechanisms are deployed. In our study of the top 1,000 mobile SDKs, APICredCheck classifies 691 as supporting Android and 388 as involving API credentials. A manual audit of the top 50 SDKs in the same ranking yields a screening precision of 66.7% and a recall of 88.9%. Among the top 50 SDKs, 18 were manually confirmed to use API credentials. Three of these 18 SDKs could not be further evaluated because the SDK providers required an app-market listing URL or detailed real-world company information before issuing API credentials; for ethical reasons, we did not perform these steps. For the remaining 15 SDKs, APICredCheck successfully abuses 13 by reusing derived credentials and client-identity information, allowing the attacker app to bypass SDK server-side authentication and invoke protected APIs. The dataset, source code, and detailed experimental results are publicly available at https://github.com/ZhouZiyi1/APICredCheck.

Contributions. This paper makes the following contributions:

- We highlight API credential abuse feasibility as a post-detection security question, distinguishing credential exposure from actual unauthorized API invocation.
- We present APICredCheck, an automated framework that combines static analysis, dynamic analysis, and LLM-assisted reasoning to evaluate API credential protection mechanisms in Android SDKs.
- We conduct an empirical study on popular Android SDKs and confirm that 13 SDKs can be abused through runtime replacement of credentials and client-identity information, revealing the practical security risk of API credential protection mechanisms.

Ethical Considerations. All devices, SDK developer accounts, and API credentials used in our experiments were created by the authors themselves. The authors did not use any API credentials extracted from real-world apps, and the experiments did not pose security risks to real-world apps or their users. None of the legitimate or attacker apps were published to app markets or distributed outside the research artifact. In the public artifact, all valid author-owned API credentials have been removed from the released source code, thereby avoiding potential risks to the authors themselves.

II. API Credentials and Protection

A. API Credentials in SDKs

In this paper, we use API credentials as a broad term covering API keys [1], app secrets [2], client tokens [3], app identifiers [4], and other service-specific values. Exposure of these credentials may allow adversaries to gain unauthorized access to sensitive data [9], manipulate service resources, incur substantial financial costs [21], or trigger denial-of-service conditions.

B. API Credential Protection Mechanisms

SDK providers often protect API credentials by binding them to client identity attributes. We manually examined the credential registration and configuration workflows of the 50 most popular Android third-party SDKs, and categorize the observed attributes as follows.

Package Name. The Android package name, or application ID [22], uniquely identifies an app on Android devices and in Google Play.

Signing Certificate Fingerprint. Android requires each APK to be signed before installation [23]. The signing certificate is associated with the developer's signing key, and the hash digest of this certificate is commonly referred to as the certificate fingerprint [23], such as SHA-1 or SHA-256 digests.

Caller IP. Some SDK providers support IP-based restrictions, under which credentials can be used only by requests from specified IP addresses or IP ranges.

Attestation-protected Client Identity. Remote attestation [24], [25] allows a server to verify client properties using evidence backed by a trusted execution environment or hardware root of trust. SDK backends can use such evidence to authenticate the calling app instead of trusting replayable client-side values directly.

III. Threat Model

We consider an attacker who can obtain the target SDK and build an attacker app that integrates the same SDK as a legitimate app. The attacker app uses invalid or unauthorized API credentials and differs from the legitimate app in package name and signing certificate digest. The attacker's goal is to make the SDK server accept API requests issued from the attacker app.

We do not consider provider backend compromise and developer account compromise. Meanwhile, we do not assume that the attacker can perform authorized-IP spoofing or hardware-backed attestation forgery. Here, authorized-IP spoofing refers to sending requests from an IP address within an SDK provider's authorized IP range, for example through a network proxy. Hardware-backed attestation forgery refers to bypassing hardware-backed remote attestation mechanisms. Therefore, for SDKs protected by IP-based restrictions or hardware-backed remote attestation, we assume that the attacker cannot bypass these protections to invoke the protected APIs.

IV. APICredCheck

A. Overview

To detect whether exposed API credentials can be abused, we design and implement an end-to-end detection framework named APICredCheck. As shown in Fig. 1, its pipeline consists of three main stages: SDK Screening, Legitimate App Analysis, and Runtime Validation. Before running APICredCheck, we prepare a list of target SDKs and a paired APK setup for each SDK, including a legitimate APK and an attacker APK. The legitimate APK integrates the target SDK with valid API credentials. The attacker APK implements similar SDK functionality, but does not contain valid authorized credentials and uses a different package name and signing certificate from the legitimate APK. Starting from the SDK list, APICredCheck identifies SDKs that may require developer-provisioned API credentials (Section IV-B), analyzes the legitimate APK prepared for each selected SDK (Section IV-C), and then dynamically validates whether the attacker APK can invoke protected SDK APIs by reusing legitimate-APK-derived credentials and client-identity data (Section IV-D).

B. SDK Screening

In this step, APICredCheck screens the target SDK list and identifies Android SDKs that may involve developer-provisioned API credentials.

APICredCheck first performs LLM-assisted screening over a list of top-ranked Android SDKs. Given an SDK name and vendor, APICredCheck sends the information to a remote online LLM and asks it to retrieve and analyze related documentation. Based on the retrieved documentation, the LLM outputs whether the SDK supports Android, whether it requires API credentials, and what those credentials are called.

The screening stage produces structured fields including Android support, API credential requirement, and credential name.

C. Legitimate App Analysis

In this step, APICredCheck analyzes the legitimate APK to identify authentication-related methods and collect the legitimate-app identifiers for later runtime replacement. It first extracts relevant evidence from the APK files, and then uses an LLM to identify authentication-related methods from the extracted evidence. APICredCheck further filters out unsafe hook targets that may crash the app. Finally, it constructs a legitimate app profile that records the values to be replayed during runtime validation.

1) Evidence Extraction: The goal of Evidence Extraction is to collect authentication-related evidence from the legitimate APK for later auth point identification. Authentication-related evidence refers to code snippets and metadata entries that indicate how the SDK obtains, processes, or transmits API credentials and client-identity data.

Specifically, APICredCheck first unpacks and decompiles the APK, and then analyzes both the decompiled code (i.e., smali code) and app configuration files (i.e., XML, JSON, and properties files).

(1) Extracting Evidence from Decompiled Code. For decompiled code, APICredCheck searches for line-level authentication-related evidence involving package-name access, signing-certificate access, certificate digest computation, network request construction, and platform service interaction. The regular patterns used in this process are summarized in Table I.

This analysis produces line-level code evidence, where each record contains the evidence type, the source location (file path and line number), and a matched code line. These records indicate concrete code locations where credential or client-identity material may be read, transformed, or attached to requests.

(2) Extracting Evidence from Configuration Files. APICredCheck also scans configuration files because API credentials and client-identity values may be declared outside SDK code, such as in manifest metadata, resource strings, JSON fields, or properties entries, before being loaded by the SDK code. In such cases, scanning only decompiled code may leave the LLM with incomplete evidence and weaken its ability to identify authentication-relevant code locations.

For configuration files, APICredCheck scans `AndroidManifest.xml`, `local.properties`, SDK-specific JSON/properties files, and resource XML files such as `strings.xml`, `values.xml`, and other SDK-specific XML files. It extracts printable strings and matches them against credential-related patterns, including API-credential-like formats, key-assignment patterns near API credential, and manifest metadata entries related to API credentials. The regular expressions used in this process are summarized in Table II. This scan produces configuration evidence records, where each record contains the evidence type, the source location, and the matched value or key.

(3) Selecting Evidence-related Context Files. After extracting evidence from decompiled code and configuration files, APICredCheck selects files to construct the LLM context for auth point identification. Rather than sending all files with matched evidence, APICredCheck first includes `AndroidManifest.xml` if it exists, and then scans decompiled `.smali`, `.xml`, `.json`, and `.properties` files. It filters out low-value paths, such as common framework or library directories, using the keywords listed in Table III.

Each remaining file is then scored using a simple rule. A file receives 5 points if it contains configuration-file evidence (corresponding to Table II), 2 points if it contains decompiled-code evidence (corresponding to Table I), and 4 points if its path contains authentication-related keywords listed in Table IV. APICredCheck ranks files with positive scores and selects up to the top 60 files with positive scores. For each selected file, APICredCheck sends the relative file path and the full file content to the LLM, together with the structured evidence. Besides, the total context is capped at 180,000 characters, and APICredCheck stops adding files once this limit would be exceeded.

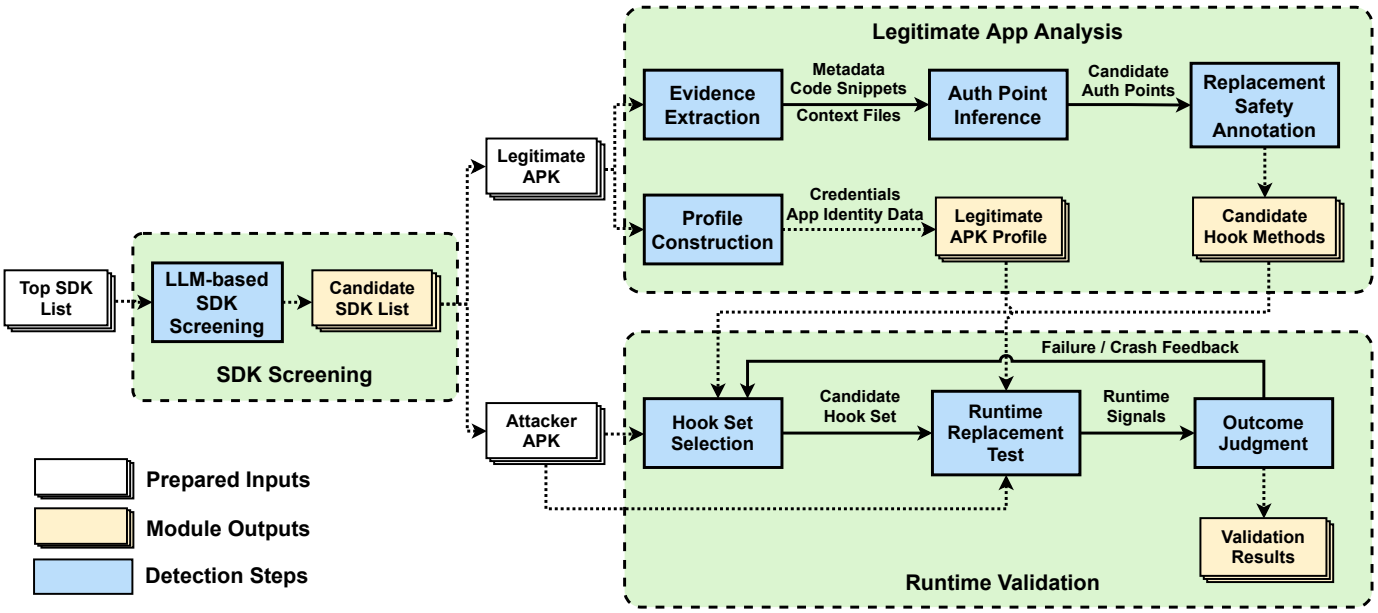


Fig. 1. System architecture of APICredCheck.

TABLE I
REGULAR EXPRESSIONS FOR LOCATING AUTHENTICATION-RELATED FEATURES IN DECOMPILED CODE.

Evidence type	Regular expressions

Package Name Access	`\b(getPackageName|getPackageInfo|packageName)\b`
	`\b(getApplicationInfo|metaData|ApplicationInfo)\b`
	`(?i)x-android-package`

Signing Certificate Access	`\b(GET_SIGNATURES|GET_SIGNING_CERTIFICATES)\b`
	`\b(SigningInfo|signatures|getApkContentsSigners)\b`
	`\b(getSigningCertificateHistory)\b`

Certificate Digest Computation	`\b(SHA-1|SHA1|SHA-256|SHA256|MessageDigest)\b` `(?i)x-android-cert`

Network Request Construction	`\b(OkHttpClient|Request\$Builder|`
	`HttpURLConnection|Retrofit|Volley|URLConnection|`
	`Socket|writeTo|enqueue|execute)\b`

Platform Service Interaction	`\b(IBinder|Parcel|transact|DynamiteModule|`
	`GoogleApiAvailability|GoogleSignatureVerifier|`
	`com/google/android/gms)\b`

TABLE II
REGULAR EXPRESSIONS FOR LOCATING CREDENTIAL-RELATED FEATURES IN CONFIGURATION FILES.

Evidence type	Regular expressions
Credential Value	`AIza[0-9A-Za-z_\-]{20,}`

Credential Assignment	`(?i)(api[_\- ]?key|apikey|client[_\- ]?key|`
	`access[_\- ]?token)`
	`[^A-Za-z0-9_\-]{0,20}`
	`([A-Za-z0-9_\-:.]{16,})`

Credential Metadata	`(?i)android:name="[^"]*`
	`(api[_\- ]?key|apikey|maps|places)[^"]*"`
	`\s+android:value="([^"]{6,})"`

2) Auth Point Inference: An auth point refers to a method that may read, process, or transmit authentication-related values. The goal of Auth Point Inference is to identify candidate auth point methods based on the extracted evidence and

TABLE III
KEYWORDS FOR IDENTIFYING LOW-VALUE PATHS WHEN SELECTING CONTEXT FILES.

Category	Excluded path fragments
Android Framework Libraries	`/android/support/`, `/androidx/`
Kotlin Runtime And Tooling	`/kotlin/`, `/kotlinx/`, `/org/intellij/`, `/org/jetbrains/`
Common Third-party Libraries	`/com/android/volley/`, `/com/bumptech/`, `/com/google/android/material/`, `/com/google/common/`, `/com/google/gson/`, `/com/squareup/`, `/okhttp3/`, `/okio/`
Large Resource Directories	`/res/drawable`, `/res/mipmap`

TABLE IV
AUTHENTICATION-RELATED PATH KEYWORDS USED FOR FILE SCORING.

Path keywords					
api	key	cert	sign	digest	sha
package	metadata	auth	request	http	url
network					

context files.

Specifically, APICredCheck sends these evidence records and context files to a remote LLM, enabling the LLM to analyze both structured evidence records and surrounding file context. The LLM is then asked to infer methods that may serve as candidate auth points.

3) Replacement Safety Annotation: In the attacker app, the arguments or return values of the candidate auth point methods will be replaced in the later dynamic validation step. However, replacing values at some candidate methods is dangerous and may lead to app crashes, because these values may also be used by normal app execution paths. For example, replacing `Context.getPackageName` can affect Android component resolution; in some cases, it may cause the app to construct an invalid activity target that combined an attacker-app activity class with the legitimate package name, leading to an `ActivityNotFoundException`. Similarly, replacing broad infrastructure methods such as `PackageManager.getPackageInfo` or `MessageDigest.digest` may pollute package lookup or local signature verification.

Therefore, the LLM is further required to mark whether each candidate is `replace_safe`. Specifically, for each candidate auth point, the LLM reasons about a source-transform-sink chain: the source indicates where a credential or client-identity value is obtained, the transform indicates how the value is processed, and the sink indicates where the value is attached to an outbound authentication request. A method is marked as `replace_safe` only when the evidence shows a clear connection from an authentication-related source to an outbound method.

Finally, the LLM is required to report candidate methods in a structured format, including `SmaliMethodName`, `EvidenceType`, `SourceLocation`, `Confidence`, `ReplaceSafety`, and `AuthChain`. Here, `Confidence` describes the LLM's confidence in the candidate, and `ReplaceSafety` indicates whether the method can be safely replaced. The prompt used to guide the LLM in this step is summarized in Table V. We explicitly describe the task as defensive security testing to remain within the safety boundaries of online LLM services [26], [27].

Input: base hook list $\mathcal{B} = [b_1, \ldots, b_m]$,
LLM candidates $\mathcal{C} = [h_1, \ldots, h_n]$
Output: selected hook set $\mathcal{S}$
$\mathcal{S} \leftarrow \emptyset$
$\mathcal{U} \leftarrow \mathcal{B} \| \mathit{SortByConfidence}(\mathcal{C})$
for each $u_i \in \mathcal{U}$ **do**
  $T \leftarrow \mathcal{S} \cup \{u_i\}$
  $r \leftarrow \mathit{RuntimeReplacementTest}(T)$
  $\mathit{outcome} \leftarrow \mathit{OutcomeJudge}(r)$
  if $\mathit{outcome} = \mathit{crash}$ **then**
    continue
  $\mathcal{S} \leftarrow T$
  if $\mathit{outcome} = \mathit{success}$ **then**
    return $\mathcal{S}$
return $\mathcal{S}$

Fig. 2. Hook set selection procedure.

4) Profile Construction: Besides identifying auth point methods, in the Legitimate App Analysis stage, APICredCheck constructs a replacement profile from the legitimate APK, including the legitimate package name, signing certificate digests, and API credentials. This profile serves as the source of API credentials and client-identity data to be replayed in the attacker app during dynamic validation.

D. Runtime Validation

Runtime Validation evaluates whether the attacker app can invoke protected SDK APIs by reusing legitimate-app-derived credentials and client-identity data. For each legitimate app, we prepare a paired attacker app that integrates the same SDK and implements the same functionality, but uses an invalid credential, a different package name, and a different signing certificate. The attacker app is adapted for automated validation: after launch, it triggers the target SDK API without manual interaction and emits machine-readable signals indicating success, failure, or crash.

APICredCheck takes the auth point method list and replacement profile as inputs, instruments the attacker app at runtime, and replaces credential or identity values with legitimate-app-derived values. By observing the emitted signals, APICredCheck determines whether the SDK backend accepts the API call from the unauthorized app.

TABLE V
SIMPLIFIED PROMPT FOR LLM-BASED AUTH POINT IDENTIFICATION.

System message: You are a careful Android SDK security auditor. You help analyze Android SDK authorization behavior for defensive security testing. You should focus on evidence-based auditing and refuse to provide bypass guidance, spoofing steps, or value-replacement instructions.

...

User message: You are helping with defensive security testing of an Android APK owned by the tester. Identify SDK authorization signals that are observable in the app process. Focus on evidence that the SDK reads or sends API keys, package names, Android certificate digests, manifest meta-data, Binder/Dynamite payloads, or HTTP headers/query/body fields used by SDK servers.

...

Prioritize authorization-chain relevance over broad identity reads. A method is a good replacement candidate only when it directly builds, writes, or returns values used in a remote SDK authentication/authorization request. For each candidate, identify the source → transform → sink chain that makes it relevant to remote authentication.

...

Do not provide bypass, spoofing, value replacement, or evasion instructions. Return an audit report with likely exposed credential locations, likely protection signals, candidate auth point methods, confidence and evidence file names, and whether each candidate is `replace_safe`.

...

At the end, add a section named `FRIDA_OBSERVATION_POINTS`. In that section, list one candidate method per line in this format: `SmaliMethodName | EvidenceType | SourceLocation | Confidence | ReplaceSafety | AuthChain`. Use `replace_safe` only when the evidence shows a clear connection from an authentication-related source to an outbound SDK authentication sink; otherwise use `observe_only`. `AuthChain` must be a short source → transform → sink explanation.

...

Static metadata: ⟨serialized APK static report⟩

...

Code context follows: ⟨selected file contents⟩

1) Hook Set Selection: The goal of Hook Set Selection is to select the methods to be instrumented during runtime validation. The pseudocode of this procedure is shown in Fig. 2.

APICredCheck considers two candidate sources: a predefined base hook list $\mathcal{B} = [b_1, \ldots, b_m]$ and an LLM-inferred target-specific list $\mathcal{C} = [h_1, \ldots, h_n]$. $\mathcal{B}$ contains generic hooks for common credential and identity carriers, such as package-name APIs, package-information APIs, and certificate digest computation methods; Table VI lists these hooks. APICredCheck sorts $\mathcal{C}$ by the LLM-reported `Confidence` field and forms a unified list $\mathcal{U} = \mathcal{B} \| sort(\mathcal{C})$, where $\|$ denotes list concatenation. It then greedily builds the selected hook set $\mathcal{S}$ from $\mathcal{U}$. For each candidate $u_i \in \mathcal{U}$, APICredCheck tests $T = \mathcal{S} \cup \{u_i\}$. Each test consists of Runtime Replacement and Outcome Judgment, which are detailed in the following two subsections. If the trial crashes, u_i is discarded; otherwise, u_i is retained in $\mathcal{S}$, even if the current trial still fails, because it may contribute to a later successful attempt. The process stops once the outcome judge reports `success` or all candidates have been tested. The output is the final selected hook set $\mathcal{S}$.

2) Runtime Replacement Test: In this step, APICredCheck automatically installs and launches the pre-built attacker app, instruments the methods selected by the Hook Set Selection step, and replaces runtime arguments or return values according to the legitimate APK profile. After the attacker app triggers the SDK API call, it emits machine-readable runtime signals indicating the invocation result. APICredCheck uses these signals to determine whether the protected API can be successfully invoked from the attacker app.

3) Outcome Judgment: Under our threat model, the attacker can customize the attacker app. To support fully automated testing, each attacker app is adapted to automatically trigger the target SDK API after launch and emit machine-readable runtime signals, including `success`, `failure`, `crash`, and `timeout`. In our implementation, these signals are emitted as predefined Android Log messages, which APICredCheck captures through instrumentation of Android logging APIs and a logcat monitor.

If the outcome is `success`, APICredCheck stops the selection process and outputs the current target-specific hook set. Otherwise, it stops the attacker app, returns the outcome to Hook Set Selection, and repeats the runtime replacement test with an updated hook set until a successful hook set is found or no remaining candidate can be added.

E. Implementation

APICredCheck is implemented as a Python-based detection framework. The source code, dataset, and detailed experimental results are publicly available at https://github.com/ZhouZiyi1/APICredCheck.

TABLE VI
BASE HOOK SET USED BY APICREDCHECK.

Category	Instrumented Methods	Replaced Content

Package information acquisition	`ContextWrapper.getPackageName`
	`ContextImpl.getPackageName`
	`ApplicationPackageManager.getApplicationInfo`
	`ApplicationPackageManager.getPackageInfo`
	`PackageManager.getPackagesForUid`
	`PackageManager.getNameForUid`
	`AppOpsManager.checkPackage`
	`UidVerifier.uidHasPackageName`
		Package name and package metadata fields.

Certificate digest computation	`MessageDigest.digest`	Signing certificate SHA-1 or SHA-256 digest.

Resource and configuration access	`Bundle.getString`
	`Bundle.putString`
	`Resources.getString`
	`Resources.getText`
	`SharedPreferences.getString`
	`SharedPreferences.Editor.putString`
		Only when the argument value matches a package name, certificate digest, or API credential.

Request header construction	`HttpURLConnection.setRequestProperty`
	`URLConnection.setRequestProperty`
	`URLConnection.addRequestProperty`
	`okhttp3.Request.Builder.header`
	`okhttp3.Request.Builder.addHeader`
		Android package header, Android certificate header, or API-key-related header.

Request parameter and payload construction	`Uri.Builder.appendQueryParameter` `JSONObject.put`	Credential or identity fields in query parameters and JSON payloads.

Connection lifecycle handling	`URL.openConnection`
	`URLConnection.connect`
	`URLConnection.getInputStream`
	`URLConnection.getOutputStream`
	`HttpURLConnection.connect`
		Authentication headers such as `X-Android-Package` and `X-Android-Cert`.

APICredCheck takes two inputs: `SDKList.csv`, which records the rank, name, and vendor of each SDK, and a pair of APKs for each SDK requiring further analysis, consisting of a legitimate APK and a corresponding attacker APK. For APK preparation, we prioritize official sample projects provided by SDK providers. If no official sample project is available, we integrate the SDK and implement the target API invocation according to the developer documentation. The final output of APICredCheck is a structured validation result for each evaluated SDK, including the final outcome and the final hook set.

Internally, APICredCheck is organized as a set of Python modules corresponding to the main stages of the pipeline.

- **The SDK Screening module** reads `SDKList.csv`, queries the remote LLM with the SDK name and vendor, and normalizes the response.
- **The Legitimate App Analysis module** unpacks and decompiles each legitimate APK with Apktool [28], extracts evidence, queries the LLM to identify auth points and assess their replacement safety, and constructs the legitimate APK profile.
- **The Runtime Validation module** installs and launches the attacker APK with Android Debug Bridge (ADB) [29], generates a Frida [30] instrumentation script based on the selected hook set and the legitimate APK profile, attaches the script to the attacker app process to replace matched arguments or return values at runtime. The attacker app reports API invocation states by emitting `APICREDTEST_*` signals through Android's `android.util.Log` API [31]. APICredCheck captures these signals by hooking `android.util.Log.i/d/w/e` with Frida, and additionally uses `adblogcat` [32] as a fallback monitor. Based on the observed signals and execution status, APICredCheck classifies each attempt as success, failure, crash, or timeout. It further outputs a standalone Frida instrumentation script for successful cases.

V. Evaluation

Our evaluation is organized around three Research Questions (RQs):

- RQ1 (API Credential Prevalence): How prevalent are API credentials among popular Android SDKs?
- RQ2 (Client-Identity Protection Mechanisms): What client-identity validation mechanisms are adopted by SDKs that rely on API credentials?
- RQ3 (Automated Abuse Feasibility): Without prior knowledge of the credential values or protection details, to what extent can an attacker automatically invoke protected SDK APIs from unauthorized Android apps?

A. Dataset and Experimental Setup

We use the all-category Top SDKs ranking [33] provided by data.ai [34] on April 1, 2026. From this ranking, we select the top 1,000 SDKs as the subjects of our study. We develop the legitimate and attacker apps with Android Studio [35]. The core detection pipeline of APICredCheck is implemented in Python 3.9.0 [36].

For LLM-assisted analysis, APICredCheck uses a remote OpenAI `gpt-5.4` model [37]. The same model configuration is used for LLM-based SDK Screening, Auth Point Inference, and Replacement Safety Annotation.

In static analysis, APICredCheck uses Apktool 2.6.0 [28] to unpack and decompile APK files. APICredCheck also invokes Android signing utilities, including `apksigner` 0.9 [38] and `keytool` [39] inside JDK 13, to extract signing certificate digests used in the legitimate APK.

In dynamic analysis, experiments are conducted on a rooted OnePlus 9R [40] smartphone running Android 11 with Oxygen OS `11.2.4.4.LE28D`. The host machine is an ASUS ROG Zephyrus M15 [41] laptop running Microsoft Windows 10, equipped with an Intel Core i7-10875H CPU and 32 GB of RAM. Runtime instrumentation is performed with Frida 16.4.10 [30].

B. Results on API Credential Prevalence

To answer RQ1, we apply LLM-based SDK Screening to the top 1,000 SDKs in our dataset. For each SDK, APICredCheck determines two properties: whether the SDK supports Android, and whether it involves developer-provisioned API credentials.

Figure 3 shows the SDK screening results on the top 1,000 SDKs. Of these SDKs, 309 are classified as non-Android SDKs, i.e., SDKs only applicable to platforms such as iOS. For the remaining 691 Android SDKs, APICredCheck identifies 388 SDKs as involving API credentials, while 303 SDKs are classified as not involving API credentials. This result suggests that API credentials are common in mainstream Android SDKs.

To evaluate the reliability of the LLM-based screening results, we manually audit the top 50 SDKs in the ranking. Table VII reports the results of this manual audit. In this table, the *Non-Credential* class includes both SDKs that do not support Android and SDKs that support Android but do not involve API credentials.

TABLE VII
Accuracy of LLM-based SDK Screening for the top 50 SDKs.

LLM Prediction Results	Manual Verification Results		
	Credential	**Non-Credential**	**Total**
Credential	16	8	24
Non-Credential	2	24	26
Total	18	32	50

According to our manual validation, LLM-based SDK Screening achieves 66.7% precision (16/24) and 88.9% recall (16/18), with a false positive rate of 25.0% (8/32) and a false negative rate of 11.1% (2/18). The relatively high recall indicates that the screening stage captures most SDKs that truly involve Android API credentials.

Figure 4 further summarizes the manually audited top 50 SDKs. Among these 50 SDKs, 11 do not support the Android platform, while 39 support Android. Of the 39 Android SDKs, 18 are manually confirmed to involve API credentials and 21 are not. The subsequent evaluation focuses on these 18 credential-dependent Android SDKs.

C. Results on Client-Identity Protection Mechanisms

We next answer RQ2 by examining what client-identity protection mechanisms are supported by SDK providers. Specifically, for the SDKs confirmed to be credential-dependent, we manually registered developer accounts and attempted to apply for API credentials. During API credential application, SDK providers often require or allow developers to configure restrictions on how the API can be invoked. These restrictions reveal the client-identity protection mechanisms adopted by SDK providers, and we can identify the protection mechanisms supported by inspecting these configuration options.

Among the 18 credential-dependent SDKs, credential application failed for three SDKs. The failures were caused by provider-side eligibility checks during credential application. For example, Facebook Login Kit SDK [42] required a real app listing URL from a mainstream app store, such as Google Play [43] or APKPure [44], before activating the assigned API credential. AppLovin [45] required a company name, a company website, and an app listing URL before allowing developer account registration. For ethical reasons, we did not submit fabricated company information or publish apps to public app stores. As a result, we could not obtain valid API credentials for these SDKs, so we excluded them from further testing.

From the remaining 15 SDKs, we observe four major types of client-identity protection mechanisms, consistent with the mechanisms discussed in Section II-B: package-name checks, signing-certificate fingerprint checks, caller-IP restrictions, and attestation-based protection. Figure 5 summarizes how frequently different protection mechanisms appear across the evaluated SDKs.

Note that these mechanisms are not mutually exclusive: an SDK may adopt multiple protection mechanisms simultaneously. For example, 11 SDKs combine package-name



Fig. 3. LLM-based SDK Screening results for the top 1,000 SDKs.

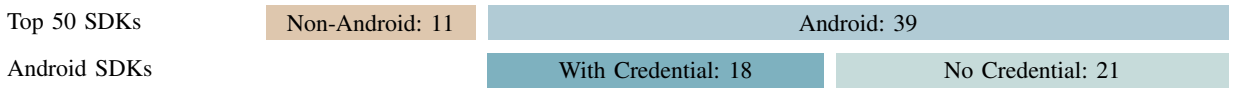


Fig. 4. Manual audit results for the top 50 SDKs.

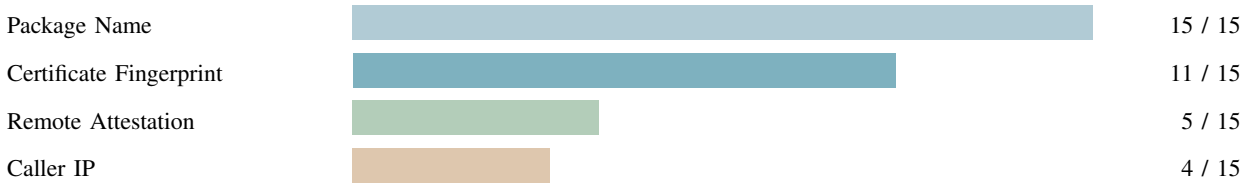


Fig. 5. Frequency of client-identity protection mechanisms among the 15 evaluated SDKs.

restrictions with signing-certificate fingerprint restrictions, and three SDKs adopt all four protection mechanisms.

Overall, these results show that modern SDK providers commonly deploy additional client-identity checks around API credentials. However, most of the observed mechanisms still rely on client-side identity values, such as package names or certificate-derived digests, which may be observable or replaceable at runtime. This motivates our subsequent abuse validation, which evaluates whether these protection mechanisms can be automatically bypassed.

D. Results on Automated Abuse Feasibility

We answer RQ3 by evaluating whether APICredCheck can automatically abuse API credentials from an unauthorized Android app without prior knowledge of credential values or protection details. This experiment focuses on the 15 SDKs for which valid API credentials could be obtained. The evaluation follows the workflow described in Sections IV-C and IV-D. Table VIII presents the evaluation results.

Overall, APICredCheck successfully invokes protected SDK APIs for 13 of the 15 SDKs, achieving an automated attack success rate of 86.7%. For the 13 successfully abused SDKs, the average number of final hooked methods is 18.15. The two failed cases are Google Firebase SDK [46] and Firebase Authentication SDK [47]. In both cases, APICredCheck does not identify any replace-safe target-specific hook method, and the final hook set contains only the base hooks. These failures suggest that the SDK server-side decision may depend on service-specific authentication logic that is not covered by the current hook set.

We also measured the running cost of APICredCheck, with detailed results reported in Table IX. For the 15 evaluated SDKs, the legitimate APK analysis stage takes 1,706 seconds in total, with an average of 113.7 seconds per SDK. The dynamic validation stage takes 387 seconds in total, with an average of 25.8 seconds per SDK. The LLM context used for auth point identification contains 129,455 characters per SDK on average.

E. Case Studies

We present two representative cases to illustrate how APICredCheck completes end-to-end abuse validation and demonstrate the concrete security threats.

1) Google Maps SDK: For Google Maps SDK [48], the LLM-based SDK screening step judges that the SDK supports Android and requires a developer-provisioned API key. We prepared a legitimate APK with a valid API key and an attacker APK with a different package name, a different signing certificate, and an invalid credential.

APICredCheck then analyzes the legitimate APK. It extracts the package name, the signing certificate SHA-1/SHA-256 digests, and a candidate API key from the manifest metadata entry `google_api_key`. From the decompiled code, APICredCheck finds multiple authentication-related evidence records involving package-name access, application metadata access, signing certificate access, certificate digest computation, and network request construction. During runtime validation, APICredCheck replaces the attacker-side package name, certificate digests, and API key with legitimate-APK-derived values. The attempt succeeds: the attacker app loads the map and emits the success signal. Such abuse may lead to unauthorized consumption of paid map services, quota exhaustion, and financial denial-of-service against the legitimate developer.

2) AdMob SDK: AdMob SDK [4] is an advertising SDK that allows app developers to monetize their apps by displaying ads. After integrating the SDK, developers can receive advertising revenue when users view or interact with ads shown in their apps. In AdMob SDK, the `AdMob app ID` serves as the API credential.

APICredCheck successfully enables the attacker app to invoke AdMob SDK APIs under the legitimate app's `AdMob app ID`. This can generate invalid traffic attributed to the legitimate app, potentially triggering ad-serving restrictions against the legitimate publisher according to AdMob's invalid-traffic policies [49], [50]. As a result, an attacker may cause a denial-of-service effect against the legitimate developer's

TABLE VIII
AUTOMATED ABUSE FEASIBILITY RESULTS FOR THE 15 EVALUATED SDKS.

SDK Rank	SDK Name	Final Hooked Methods	Final Status
1	Google Firebase SDK	26	Failure
5	AdMob SDK	5	Success
12	Google Location SDK	31	Success
17	Facebook Core SDK	7	Success
18	Google Places SDK	26	Success
19	Firebase Authentication SDK	26	Failure
22	Facebook Share Kit SDK	11	Success
23	Google Maps SDK	31	Success
28	Firebase Performance Monitoring SDK	26	Success
32	Google User Messaging Platform SDK	5	Success
36	Google Tag Manager SDK	26	Success
37	Firebase Crash Reporting SDK	26	Success
47	Google Mobile Ads SDK	5	Success
49	OneSignal SDK	26	Success
50	Facebook Messenger Share Kit SDK	11	Success

TABLE IX
RUNTIME COST AND LLM INPUT USAGE.

SDK Rank	SDK Name	Legitimate App Analysis Time (s)	Runtime Validation Time (s)	LLM Context Size (Chars)
1	Google Firebase SDK	107	39	112450
5	AdMob SDK	130	20	165575
12	Google Location SDK	104	34	137650
17	Facebook Core SDK	162	13	102513
18	Google Places SDK	113	28	105307
19	Firebase Authentication SDK	151	35	111057
22	Facebook Share Kit SDK	110	18	102519
23	Google Maps SDK	80	36	132535
28	Firebase Performance Monitoring SDK	115	23	126647
32	Google User Messaging Platform SDK	115	18	165620
36	Google Tag Manager SDK	100	23	130261
37	Firebase Crash Reporting SDK	86	27	177540
47	Google Mobile Ads SDK	157	31	141075
49	OneSignal SDK	89	22	125426
50	Facebook Messenger Share Kit SDK	87	20	105645

monetization channel.

VI. MITIGATION

Rethinking API Credentials. SDK providers and app developers should treat app-side API credentials as public identifiers rather than strong authentication secrets. In particular, SDK backends should avoid allowing a client-side credential alone to retrieve sensitive user data or access high-value paid resources.

Strengthening Client Identity Verification. SDK providers should not rely solely on replayable client identifiers such as package names or certificate digests. For high-impact APIs, SDK providers should adopt stronger hardware-backed mechanisms such as Play Integrity or App Attest.

Usage Monitoring and Alerting. SDK providers should support fine-grained monitoring, alerting, and rapid response for credential abuse. Developer consoles should expose per-app usage dashboards and warn developers about abnormal traffic patterns, such as unusual geographic regions, sudden

request spikes, or rapid quota consumption.

VII. Limitations

LLM Reliability. APICredCheck relies on a remote LLM for SDK Screening and Auth Point Inference. Although the prompts are structured and the outputs are further processed by deterministic filtering, the LLM may still produce false positives and false negatives. For example, it may miss SDK-specific authorization paths, misclassify a method as replace-safe, or fail to identify a method that is necessary for a successful runtime replacement. Such incomplete LLM output may cause APICredCheck to underestimate the abuse feasibility of some SDKs.

Manual APK Preparation. APICredCheck currently requires manual preparation of paired legitimate and attacker apps. This includes registering developer accounts, requesting API credentials, repairing build issues, and compiling APKs with different package names and signing certificates. In addition, for the attacker app, developers need to insert logic that emits machine-readable signals for API invocation results. Future work could reduce this manual burden by introducing agentic APK preparation workflows. Such workflows could use AI coding agents to inspect SDK documentation, configure sample projects, repair build errors, adapt package names and signing settings, and insert standardized runtime-result reporting logic. Meanwhile, such automation should be constrained to operate within SDK providers' terms of service when registering accounts, requesting credentials, and invoking APIs.

VIII. Related Work

Secret and Credential Leakage Detection. Prior work has extensively studied secret leakage in source-code repositories, generated code, mobile apps, and other client-side software. Meli et al. [13] characterized secret leakage in public GitHub repositories, while Feng et al. [14] proposed automated techniques for detecting password leakage from public repositories. Recent work [15] further shows that neural code completion tools may memorize and reproduce hard-coded credentials, creating another source of secret exposure. In the Android ecosystem, Zhou et al. [5] studied developer credential exposure in Android apps, and later studies [8], [16], [17] proposed tools and measurements for detecting checked-in secrets and API keys in Android apps. Alecci et al. [9] further evaluated LLMs for detecting secrets in Android apps at scale, showing that LLMs can substantially expand the coverage of credential discovery. These studies mainly focus on finding, classifying, or validating exposed credentials. In contrast, APICredCheck addresses a complementary post-detection question: whether an exposed API credential can actually be abused from an unauthorized Android client under the protection mechanisms deployed by SDK providers.

Credential Misuse in Client-side Ecosystems. Several studies have examined the security impact of leaked credentials in mobile or client-side ecosystems. Wen et al. [6] studied SDK credential misuse in iOS apps, showing that mobile SDK credentials can be exposed and misused when embedded in client applications. Zuo et al. [7] investigated cloud data leakage from mobile apps and demonstrated that mobile-side credentials may expose cloud-backed data and resources. Zhang et al. [11] studied AppSecret leaks in mini-programs and showed that exposed client-side secrets can lead to practical attacks such as service theft, resource abuse, and account-level impact. Recent work [10] has also examined API-related misuse in emerging LLM service integrations in Android apps. Compared with these studies, APICredCheck focuses on whether attackers can bypass the protection mechanisms deployed around API credentials.

IX. Conclusion

In this paper, we conducted a systematic study of whether exposed API credentials in Android SDKs can be abused from unauthorized Android clients. We presented APICredCheck, a framework that combines static analysis and runtime replacement to evaluate credential protection mechanisms. Our results show that API credentials are common in mainstream Android SDKs, and that APICredCheck can successfully invoke protected APIs for 13 of the 15 evaluated credential-dependent SDKs. These findings suggest that protections based on replayable client-side identity values remain fragile in practice.

Acknowledgments

We thank all the anonymous reviewers for their valuable comments and suggestions. This work was supported by the Shanghai Natural Science Foundation (25ZR1402214) and the National Natural Science Foundation of China (62472286 and U2336210).

References

[1] Google, "Set up an Android Studio project," https://developers.google.com/maps/documentation/android-sdk/config#step_3_add_your_api_key_to_the_project, 2026, accessed: 19 June 2026.

[2] Google, "Set up an android studio project," https://developers.google.com/maps/documentation/places/android-sdk/config#get-an-api-key, 2026, accessed: 19 June 2026.

[3] Meta Platforms, Inc., "Facebook login for android: Quickstart," https://developers.facebook.com/documentation/facebook-login/android#4-edit-your-resources-and-manifest, 2026, accessed: 19 June 2026.

[4] Google, "Set up Google Mobile Ads SDK: Android," https://developers.google.com/admob/android/quick-start, 2026, accessed: 19 June 2026.

[5] Y. Zhou, L. Wu, Z. Wang, and X. Jiang, "Harvesting developer credentials in android apps," in *Proceedings of the 8th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 2015, pp. 1–12.

[6] H. Wen, J. Li, Y. Zhang, and D. Gu, "An empirical study of sdk credential misuse in ios apps," in *Proceedings of the 25th Asia-Pacific Software Engineering Conference*, IEEE. IEEE, 2018, pp. 258–267.

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

[8] K. Piyumantha, J. Senanayake, and K. Wijayasiriwardhne, "Droidkey: A practical framework and analysis tool for api key security in android applications," in *Proceedings of the 8th International Research Conference on Smart Computing and Systems Engineering*. IEEE, 2025, pp. 1–6.

[9] M. Alecci, J. Samhi, T. F. Bissyandé, and J. Klein, "Secrets unlocked: Evaluating llms for secrets detection in android apps," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. Oakland, CA, USA: ACM, Oct. 2026, to appear.

[10] J. Wen and Q. Zhao, "Demystifying LLM API misuses: A lifecycle-based empirical study on real-world android apps," in *Proceedings of the Sixteenth ACM Conference on Data and Application Security and Privacy*, 2026, pp. 113–125.

[11] 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*. ACM, 2023, pp. 2411–2425.

[12] MITRE Corporation, "CWE-798: Use of hard-coded credentials," https://cwe.mitre.org/data/definitions/798.html, 2025, accessed: 17 June 2026.

[13] M. Meli, M. R. McNiece, and B. Reaves, "How bad can it git? characterizing secret leakage in public github repositories," in *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[14] 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*. ACM, 2022, pp. 175–186.

[15] Y. Huang, Y. Li, W. Wu, J. Zhang, and M. R. Lyu, "Your code secret belongs to me: Neural code completion tools can memorize hard-coded credentials," *Proceedings of the ACM on Software Engineering*, 2024.

[16] K. Li, L. Ling, J. Yang, and L. Wei, "Automatically detecting checked-in secrets in android apps: How far are we?" arXiv preprint arXiv:2412.10922, 2024.

[17] 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.

[18] Google, "Play Integrity API," https://developer.android.com/google/play/integrity, 2026, accessed: 17 June 2026.

[19] Apple Inc., "Create a DeviceCheck private key," https://developer.apple.com/help/account/capabilities/create-a-devicecheck-private-key/, 2026, accessed: 17 June 2026.

[20] Apple Inc., "Establishing your app's integrity," https://developer.apple.com/documentation/devicecheck/establishing-your-app-s-integrity, 2026, accessed: 17 June 2026.

[21] R. Singh, "CS student receives $55,444 Google Cloud bill after API key exposed on GitHub," https://voice.lapaas.com/cs-student-google-cloud-bill-55444-api-key-leak-2025/, 2025, accessed: 19 June 2026.

[22] Google, "Configure the app module," https://developer.android.com/build/configure-app-module#set-application-id, 2026, accessed: 23 June 2026.

[23] Google, "Sign your app," https://developer.android.com/studio/publish/app-signing, 2026, accessed: 23 June 2026.

[24] IETF, "RFC 9334: Remote ATtestation procedureS (RATS) architecture," https://datatracker.ietf.org/doc/rfc9334/, 2023, accessed: 14 May 2023.

[25] G. Coker, J. D. Guttman, P. A. Loscocco, A. L. Herzog, J. K. Millen, B. O'Hanlon, J. D. Ramsdell, A. Segall, J. Sheehy, and B. T. Sniffen, "Principles of remote attestation," *International Journal of Information Security*, vol. 10, pp. 63–81, 2011.

[26] OpenAI, "Trusted access for cyber," https://chatgpt.com/cyber, 2026, accessed: 28 June 2026.

[27] OpenAI, "Usage policies," https://openai.com/policies/usage-policies/, 2025, accessed: 28 June 2026.

[28] iBotPeaches, "Apktool," https://apktool.org/, 2026, accessed: 1 July 2026.

[29] Google, "Android Debug Bridge (adb)," https://developer.android.com/tools/adb, 2026, accessed: 1 July 2026.

[30] Frida, "Frida: A dynamic instrumentation toolkit," https://frida.re/, 2026, accessed: 1 July 2026.

[31] Google, "Log api reference," https://developer.android.com/reference/android/util/Log, 2026, accessed: 1 July 2026.

[32] Google, "Logcat command-line tool," https://developer.android.com/tools/logcat, 2026, accessed: 1 July 2026.

[33] data.ai, "Top SDKs report," https://www.data.ai/intelligence/sdk/top, 2026, accessed: 2 July 2026.

[34] data.ai, "data.ai," https://www.data.ai/, 2026, accessed: 2 July 2026.

[35] Google, "Android Studio," https://developer.android.com/studio, 2026, accessed: 2 July 2026.

[36] Python Software Foundation, "Python 3.9.0," https://www.python.org/downloads/release/python-390/, 2020, accessed: 2 July 2026.

[37] OpenAI, "Models," https://developers.openai.com/api/docs/models, 2026, accessed: 2 July 2026.

[38] Google, "apksigner," https://developer.android.com/tools/apksigner, 2026, accessed: 2 July 2026.

[39] Oracle, "keytool," https://docs.oracle.com/en/java/javase/11/tools/keytool.html, 2026, accessed: 2 July 2026.

[40] OnePlus, "OnePlus 9R specs," https://www.oneplus.in/9r/specs, 2026, accessed: 2 July 2026.

[41] ASUS, "ROG Zephyrus M15 specs," https://rog.asus.com/bt/laptops/rog-zephyrus/rog-zephyrus-m15-series/spec/, 2026, accessed: 2 July 2026.

[42] Meta Platforms, Inc., "Facebook login for android," https://developers.facebook.com/documentation/facebook-login/android, 2026, accessed: 4 July 2026.

[43] Google, "Google Play," https://play.google.com/store/apps, 2026, accessed: 4 July 2026.

[44] APKPure, "APKPure," https://apkpure.com/, 2026, accessed: 4 July 2026.

[45] AppLovin, "MAX android integration," https://support.applovin.com/en/max/android/overview/integration, 2026, accessed: 4 July 2026.

[46] Google, "Add Firebase to your Android project," https://firebase.google.com/docs/android/setup, 2026, accessed: 5 July 2026.

[47] Google, "Firebase Authentication," https://firebase.google.com/docs/auth, 2026, accessed: 5 July 2026.

[48] Google, "Maps SDK for Android," https://developers.google.com/maps/documentation/android-sdk, 2026, accessed: 6 July 2026.

[49] Google, "Invalid traffic," https://support.google.com/admob/answer/3342054, 2026, accessed: 7 July 2026.

[50] Google, "How you can prevent invalid activity," https://support.google.com/admob/answer/3342099, 2026, accessed: 7 July 2026.