JETTYPED: A Study of Cross-Language Type Bugs in Android's JavaScriptEngine

Abhishek Tiwari, Jyoti Prakash, Dimitrios Dafnis, Mikkel Baun Kjærgaard
SDU Software Engineering, The Mærsk McKinney Møller Institute
University of Southern Denmark
Odense, Denmark
{jyp, abti, dafn, mbkj}@mmmi.sdu.dk

***Abstract*—Android applications leverage Jetpack `JavaScriptEngine` to execute JavaScript from Java or Kotlin. This API is convenient, but it also introduces a subtle boundary contract: `evaluateJavaScriptAsync` always returns a `String`, and silently returns `""` when the final JavaScript value is not a string. This behavior is documented but not reflected in the Java type signature, making related failures hard to diagnose. We present the first empirical study of Jetpack `JavaScriptEngine` usage in the wild. We collected 46 open-source consumers of the API, together with 867 forks, and analyzed them using JETTYPED, a hybrid analysis framework that works on both source code and APKs. JETTYPED found 27 bugs across 18 repositories, affecting 105 program locations. Most affected locations arise from type-return mismatches or null/undefined propagation through unchecked wrappers. We reported a high-confidence subset to developers; seven were confirmed or acted on, and four already led to code changes.**

***Index Terms*—Cross-language analysis, Android Hybrid apps, static analysis, dynamic analysis, type bugs**

I. INTRODUCTION

Android's Jetpack `JavaScriptEngine` [1] lets apps execute JavaScript from Java or Kotlin for tasks such as scriptable business logic, data conversion, and wallet or media processing. This creates a non-trivial cross-language boundary: JavaScript script construction, engine execution, and result consumption often occur in different parts of the application. The key difficulty is a subtle contract in `evaluateJavaScriptAsync`: it returns a `Future<String>`, but silently resolves to `""` when the final JavaScript value is not a string. Although Android documents this behaviour[1], the Java type signature does not expose this behavior. Consequently, failures may surface only later when the returned value is parsed, deserialized, or otherwise processed by downstream application logic.

This behaviour gives rise to a family of *cross-language boundary bugs* (Table I). In this paper, we focus on four recurring categories: (1) *type-return mismatches* (B1), where the final JavaScript expression has the wrong type; (2) *JSON-structure mismatches* (B2), where the produced JSON shape does not match the Java-side access pattern; (3) *null/undefined propagation* (B3), where the script evaluates to `null` or `undefined` and the wrapper does not guard against the resulting empty string; and (4) *arity mismatches* (B4), where a function inside the submitted script is called with the wrong number of arguments. These failures evade single-language analyses and often surface only at runtime, where the bug is observed at a different location from the communication site.

To understand how often this problem occurs in practice, we conducted the first empirical study of Jetpack `JavaScriptEngine` adoption. We systematically searched GitHub together with six complementary code-search platforms (grep.app, GitLab, Codeberg, Sourcegraph, SearchCode, and Bitbucket), and validated the resulting GitHub corpus against the F-Droid and AndroZoo corpora. This process identified 46 open-source repositories together with 867 forks (Section IV-A). Note that the adoption is still early but growing. Among these repositories, 32 exhibit strict API usage in project code. As of March 2026, these repositories constitute the discoverable user base that we could identify for this API.

Existing tools do not model this boundary precisely. General-purpose analysers such as SpotBugs and SonarQube do not encode the `evaluateJavaScriptAsync` contract. Various cross-language analyzers for WebView (e.g. *HybridDroid* [2], *JNSAF* [3], *ωTest* [4], *IWanDroid* [5], *WebViewJSdetect* [6]) focus on a different bridge API. The bugs we study arise from a documented but under-enforced coercion at the language boundary, and this coercion is invisible at the Java type-signature level.

To study this problem, we built JETTYPED, a hybrid program analysis framework for Jetpack `JavaScriptEngine`. It supports both source repositories and compiled APKs, enabling repository-based studies as well as analysis of released applications through two complementary pipelines (Fig. 3). First, the source-code pipeline performs demand-driven static analysis: it discovers boundary sites, computes interprocedural boundary chops around boundary functions, wrappers, and consumers, infers cross-language contracts, and classifies violations (B1–B4). Boundaries whose contracts cannot be statically resolved are emitted as a refinement plan for downstream dynamic validation. Second, the APK pipeline performs hybrid binary analysis by combining APK-level bridge summarization, Smali-level chopping, and selective dynamic analysis restricted to boundaries that remain unresolved after static analysis. Finally, JETTYPED reasons about an under-enforced semantic contract induced by type coercion and produces a report of affected program locations.

We use the source-code pipeline as the primary basis for the

[1]https://developer.android.com/develop/ui/views/layout/webapps/jsengine

ecosystem study, and the APK pipeline as a complementary binary-side analysis. Across the source-code corpus, the source pipeline found 27 bugs in 18 repositories, manifested at 105 affected program locations. The bugs originate at unchecked `JavaScriptEngine` boundaries or wrappers; the affected locations are the later parser, UI, RPC, or transaction uses reached by those results. Type-return mismatches (B1) and null propagation through unguarded wrappers (B3) account for most affected locations. To complement this source-level result, the APK pipeline detected 89 affected program locations across 15 compiled applications. Selective dynamic analysis yielded app-specific dynamic success in eight APKs. We reported 18 high-confidence cases to developers; seven were confirmed or acted on, and four already led to code changes.

In summary, this paper makes the following contributions:

1) The first empirical study of Jetpack `JavaScriptEngine` adoption, covering all 46 discoverable repositories and characterizing cross-language violations in the source-code corpus (Sections IV-B and IV-C).
2) A semantic bug taxonomy (B1–B4) for under-enforced coercion at the `JavaScriptEngine` boundary, which defines the bug classes studied in this work (Section II).
3) JETTYPED[2], a hybrid analysis framework based on *boundary-contract inference*, which identifies meaningful engine boundaries, infers the expected cross-language contract, and applies selective dynamic analysis only when static analysis is incomplete (Section III).
4) A repository-based empirical study that identifies 27 bugs in 18 repositories, manifested at 105 affected program locations, complemented by APK-side analysis that detects 89 affected program locations in compiled apps. We reported a high-confidence subset, and developers acknowledged or acted on seven reports (Sections IV-C and IV-D).

II. MOTIVATION AND BACKGROUND

A. Jetpack JavaScriptEngine

The Jetpack `JavaScriptEngine` library [1] enables Android applications to evaluate JavaScript inside a sandboxed process without instantiating a `WebView`. Its public API has three principal components.

1) `JavaScriptSandbox`: a long-lived connection to the sandboxed V8 engine process, obtained via `createConnectedInstanceAsync(...)` and generally reused across the application.
2) `JavaScriptIsolate`: an isolated V8 execution context created from a sandbox connection via `sandbox.createIsolate()`, within which subsequent evaluations share state.
3) `evaluateJavaScriptAsync(String code)` [7]: the main string-based entry point for script evaluation, which submits JavaScript to the isolate's V8 engine and returns a `ListenableFuture<String>`.

[2]https://github.com/mig40000/JetTyped

The library also provides overloads that accept JavaScript through `AssetFileDescriptor` and `ParcelFileDescriptor`. Although the input form differs, the returned result follows the same contract as `evaluateJavaScriptAsync(String code)`.

The contract is encoded in the return type, i.e., the result is always a `String`. If the final expression of the evaluated script yields a JavaScript `String`, the future resolves to that string. If the sandbox supports promise return values, a JavaScript `Promise` may also resolve to a `String`. For other JavaScript result types (`number`, `boolean`, `object`, `null`, or `undefined`), the engine silently returns the empty string `""`. For this specific non-string-result case, no exception is thrown, and no warning is logged. This silent coercion is the core mechanism behind B1 and B3, and it also amplifies B4. More broadly, it illustrates the under-enforced boundary contract that motivates all four bug categories.

Unlike the WebView `addJavascriptInterface` bridge, `JavaScriptEngine` provides *no mechanism for JavaScript to invoke Java callbacks*. There are no bridge variables, no proxy objects, and no `@JavascriptInterface` annotations in this API. Binary data can be passed to a script via `provideNamedData(name, bytes)` (primarily for WebAssembly modules), but control flow always originates in Java and terminates with a `String` result. JETTYPED is scoped entirely to this API; WebView-based bridges are out of scope.

B. Motivating Example and Bug Taxonomy

Listing 1 illustrates how this contract can fail in practice. `StockHelper.java` is a hypothetical class from an Android investment app that delegates financial computations to Jetpack `JavaScriptEngine`, a common pattern when script logic needs to be updated without recompiling the app. The class exposes four methods, each submitting a different script via `evaluateJavaScriptAsync` and parsing its `String` result back to a Java type. Table I summarizes the four bug categories illustrated by this example. All four methods contain a distinct cross-language type bug (B1–B4); none of them produces a compile-time error or a warning from the Java compiler, SpotBugs, or any JavaScript type checker.

The taxonomy follows from the documented `evaluateJavaScriptAsync` return contract [7] and from the host-side patterns that consume the returned `String`. B1 and B3 capture value-level failures caused by the engine returning `""` for non-string, `null`, or `undefined` JavaScript results. B2 captures structured-data mismatches after a string has crossed the boundary. B4 captures script-side argument mismatches that can propagate `undefined` or `NaN` to the final result. These categories define the scope of JETTYPED in this paper; they are not intended to cover all possible `JavaScriptEngine` bugs.

B1 – Type-return mismatch (`profitLoss`). `computePnL` computes profit-and-loss as a JavaScript arithmetic expression and returns the raw `number` result without converting it to a string. The Jetpack contract dictates that any non-`string` final value produces `""`; `Double.parseDouble("")` then throws a `NumberFormatException`. A JavaScript type checker sees a

```java
public class StockHelper {
  private final JavaScriptIsolate iso;

  // B1: computePnL returns a JS *number*, not a String
  public double profitLoss(double buy, double sell, int qty)
      throws Exception {
    String script =
      "function computePnL(b,s,q){ return (s-b)*q; }" + // number!
      "computePnL(" + buy + "," + sell + "," + qty + ")";
    String r = iso.evaluateJavaScriptAsync(script).get();
    // r == "" because number silently coerced by engine
    return Double.parseDouble(r); // throws NumberFormatException
  }

  // B2: JSON key "ticker" produced, but "symbol" consumed
  public String tickerSymbol(String code) throws Exception {
    String script =
      "JSON.stringify({ticker:'" + code + "',price:182.5})";
    String r = iso.evaluateJavaScriptAsync(script).get();
    // r == {"ticker":"AAPL","price":182.5} (key is "ticker")
    return new JSONObject(r).getString("symbol"); // JSONException
  }

  // B3: absent map key returns undefined; engine yields ""
  public double exchangeRate(String ccy) throws Exception {
    String script =
      "var fx={USD:1.0,EUR:0.91,GBP:0.79}; fx['" + ccy + "']";
    String r = iso.evaluateJavaScriptAsync(script).get();
    // r == "" when ccy == "JPY" (undefined -> "")
    return Double.parseDouble(r); // throws NumberFormatException
  }

  // B4: weightedAvg(px,wt,scale) called with only 2 args
  public double weightedAvg(double[] px, double[] wt)
      throws Exception {
    String script =
      "function weightedAvg(px,wt,scale){" +
      "  return px.reduce((a,p,i)=>a+p*wt[i],0)*scale;}" +
      "weightedAvg([182.5,141.8,93.2],[0.5,0.3,0.2])"; // scale
          omitted
    String r = iso.evaluateJavaScriptAsync(script).get();
    // scale==undefined => NaN (a JS number) => engine returns ""
    return Double.parseDouble(r); // throws NumberFormatException
  }
}
```

Listing 1: `StockHelper.java`: four methods, each exhibiting one of the four bug categories (B1–B4) at the `evaluateJavaScriptAsync` bridge.

numeric return (correct for pure JS) and raises no warning. A Java static analyser sees a well-typed `String` variable and also raises no warning. Only cross-language reasoning can flag this.

B2 – JSON-schema mismatch (`tickerSymbol`). The script produces a JSON object with keys `"ticker"` and `"price"`, but the Java caller accesses key `"symbol"`, which does not exist in the schema. The JSON string is well-formed (the return type is `string`, satisfying B1 and B3), so the engine does not return `""`. The fault is a *schema-level* mismatch: the inferred JavaScript schema $\{$`"ticker"`: `string`, `"price"`: `number`$\}$ is not a subtype of the Java access schema $\{$`"symbol"`: `string`$\}$.

B3 – Null/undefined propagation (`exchangeRate`). The script performs a map lookup; for an absent key (e.g., `"JPY"`), JavaScript returns `undefined`. The engine coerces `undefined` to `""`, which again triggers a `NumberFormatException` at `Double.parseDouble`. Unlike B1, the return type is *conditional*: the script is well-typed for known currencies but propagates `undefined` for unknown ones. Detecting B3 requires tracking data-dependent `null`/`undefined` flows across the bridge.

TABLE I: Bug Taxonomy

Bug	Name	Description
B1	Type-return mismatch	Final JavaScript value is non-string, so the engine returns `""`.
B2	JSON-schema mismatch	Returned JSON string is well-formed, but its schema does not match the Java-side access pattern.
B3	Null/undefined propagation	The script evaluates to `null` or `undefined`, which collapses to `""`.
B4	Arity mismatch	A JavaScript call supplies the wrong number of arguments, often cascading into B1 or B3.

B4 – Arity mismatch (`weightedAvg`). `weightedAvg` is declared with three formal parameters (`px`, `wt`, `scale`), but the call site supplies only two arguments. The third parameter `scale` binds to `undefined`; `undefined` in arithmetic produces `NaN`, a JavaScript `number`, which cascades to B1: the engine returns `""` and `parseDouble` throws. The arity mismatch is the root cause; the runtime symptom is identical to a plain B1 bug.

All four failures surface at the same Java statement (`parseDouble` or `getString`), an empty or malformed string, making them impossible to distinguish, or even attribute to the JavaScript side, without cross-language analysis.

III. Methodology

A. Abstract Language Model

To facilitate the discussion of JetTyped, we introduce the abstract language model for hybrid Android apps that use the *Jetpack JavaScript Engine*. Fig. 1a and Fig. 1b define the abstract syntax of the Java host and JavaScript guest languages, respectively, with a focus on the constructs relevant to cross-language type bugs. The model abstracts away from language features that are irrelevant to our analysis, such as Java's class hierarchy and JavaScript's prototype chain. The two languages are connected by two bridge forms: $\mathbf{evaluateAsync}(e_s)$ for submitting a script expression e_s to the engine, and $\mathbf{parseAs}(v_r, T)$ for parsing the returned string v_r into a concrete Java type T. In the concrete Android API, $\mathbf{evaluateAsync}(e_s)$ corresponds to calls to `evaluateJavaScriptAsync` and its file-descriptor variants.

In the JavaScript model, n, s, b denote numeric, string, and boolean literals; `null` is a distinguished value; $\mathbf{func}(\overline{x})$ introduces anonymous functions; and object/array literals use standard notation. The program form $prog ::= e_{\text{top}}$ makes explicit that the entire script reduces to a single value whose type is checked against the bridge contract.

Figure 2 maps the four categories to the abstract language model. The checks happen at two points. At $\mathbf{evaluateAsync}(e_s)$, B4 arises when a JavaScript call in the submitted script does not match the callee arity. Missing arguments become `undefined`; if this value reaches the final expression, the failure can later appear as B1/B3. At $\mathbf{parseAs}(v_r, T)$, the Java/Kotlin side consumes the returned string. B1 and B3 capture value-level boundary failures: B1

occurs when the inferred JavaScript type $\tau(e_{\text{top}})$ is not `string`, and B3 when it is `null` or `undefined`. B2 is a schema-level mismatch: the boundary returns a string, but JavaScript's produced JSON does not match the Java-side access pattern.

B. Approach Overview

JETTYPED analyzes violations at the `JavaScriptEngine` boundary, where Java/Kotlin code constructs a script, invokes the engine, and consumes the returned value. The analysis has to address four practical challenges. First, the boundary is not always a direct call to `evaluateJavaScriptAsync`; it may be hidden behind helper methods or wrapper classes. Second, the value returned by the engine may pass through `Future.get`, wrapper methods, fields, assets, named data, or runtime values. Finally, unresolved cases should not all be executed dynamically; otherwise the runtime phase becomes too broad.

JETTYPED provides two complementary modes: a source-code mode for static analysis over repositories, and an APK mode for hybrid analysis over binaries. In source code, JETTYPED builds a chop around each boundary to connect script construction, engine invocation, and result consumption. In APKs, it applies the same idea to DEX and Smali because the packaged app can expose value flows that differ from the source view. Dynamic execution is used only when static analysis cannot recover the script or value flow. Both modes then use the same contract checks and reporting stage.

Figure 3 shows the overall workflow. The source-code static-analysis track (top) performs boundary discovery, boundary chopping, contract inference, and refinement plan generation over repositories. The APK hybrid-analysis track (bottom) performs boundary discovery, boundary chopping, contract inference, and selective dynamic analysis over binaries before entering the shared bug-reporting stage. When source analysis fully resolves a boundary, it proceeds directly to bug reporting. Refinement plans are generated only for unresolved boundaries and may later guide selective dynamic analysis. The optional dashed edge in the figure transfers these unresolved obligations from the source track to the APK track to focus runtime effort.

Algorithm 1 summarizes the full workflow of JETTYPED. At line 3, SOURCESTATICANALYSIS produces source-side contracts $\mathcal{C}_S$ and refinement plans $\mathcal{P}$. At line 9, APKSTATICANALYSIS produces APK-side contracts $\mathcal{C}_A$ and unresolved crossings $\mathcal{U}_A$. At line 13, SELECTIVEDYNAMICANALYSIS refines unresolved APK contracts with runtime evidence. Finally, at line 16, the shared bug-reporting stage checks $\mathcal{C}_S \cup \mathcal{C}_A$ and produces the bug set $\mathcal{B}$ and guarded crossings $\mathcal{G}$.

C. Source-Code Static Analysis

Algorithm 2 expands the source-analysis phase at line 3 (Algorithm 1). It performs a static pass over a source repository r. For each discovered boundary site, it computes a local boundary chop and infers contract features (script origin, consumption pattern, and guards). If the script origin is statically recoverable, the analysis infers JavaScript type and schema and records a concrete source-side contract in $\mathcal{C}_r$. Otherwise, it records the site in the unresolved set $\mathcal{U}_r$. After all boundary sites in r are processed, unresolved sites are converted into a refinement plan $\mathcal{P}_r$ for later use in selective dynamic analysis. In this way, the source pipeline reports precise contracts when possible and defers ambiguous cases when necessary.

a) Source-code boundary discovery: The boundary-discovery stage searches source code for three markers:

Algorithm 1 STAGEDBOUNDARYANALYSIS

Require: Source repos $\mathcal{R}$, APK set $\mathcal{A}$, test drivers $\mathcal{T}$
Ensure: Bug set $\mathcal{B}$, guarded crossings $\mathcal{G}$

Source-code static analysis
1: $\mathcal{C}_S \leftarrow \emptyset$; $\mathcal{P} \leftarrow \emptyset$
2: **for all** $r \in \mathcal{R}$ **do**
3: $\quad \mathcal{C}_r, \mathcal{P}_r \leftarrow$ SOURCESTATICANALYSIS(r)
4: $\quad \mathcal{C}_S \leftarrow \mathcal{C}_S \cup \mathcal{C}_r$
5: $\quad \mathcal{P} \leftarrow \mathcal{P} \cup \mathcal{P}_r$
6: **end for**
APK Hybrid Analysis
7: $\mathcal{C}_A \leftarrow \emptyset$; $\mathcal{U}_A \leftarrow \emptyset$
8: **for all** $a \in \mathcal{A}$ **do**
9: $\quad \mathcal{C}_a, \mathcal{U}_a \leftarrow$ APKSTATICANALYSIS(a)
10: $\quad \mathcal{C}_A \leftarrow \mathcal{C}_A \cup \mathcal{C}_a$
11: $\quad \mathcal{U}_A \leftarrow \mathcal{U}_A \cup \mathcal{U}_a$
12: **end for**
Selective dynamic analysis
13: $\mathcal{C}_A \leftarrow$ SELECTIVEDYNAMICANALYSIS$(\mathcal{C}_A, \mathcal{P}, \mathcal{U}_A, \mathcal{T})$
Bug reporting
14: $\mathcal{B}, \mathcal{G} \leftarrow \emptyset, \emptyset$
15: **for all** $c \in \mathcal{C}_S \cup \mathcal{C}_A$ **do**
16: $\quad \mathcal{B}, \mathcal{G} \leftarrow$ checkContract$(c, \mathcal{B}, \mathcal{G})$ ▷ B1–B4 rules
17: **end for**
18: **return** $\mathcal{B}, \mathcal{G}$

Algorithm 2 SOURCESTATICANALYSIS

Require: Source repository r
Ensure: Source contracts $\mathcal{C}_r$, refinement plan $\mathcal{P}_r$

1: $\mathcal{C}_r \leftarrow \emptyset$; $\mathcal{U}_r \leftarrow \emptyset$ ▷ contracts; unresolved
2: $E_r \leftarrow$ discoverBoundaries(r) ▷ boundary sites
3: **for all** $e \in E_r$ **do**
4: $\quad chop_e \leftarrow$ computeBoundaryChop(e)
5: $\quad \langle origin_e, pattern_e, guard_e \rangle \leftarrow$ inferContract$(chop_e)$
6: $\quad$ **if** $origin_e$ is statically determined **then**
7: $\quad\quad \tau, S \leftarrow$ inferJSTypeAndSchema$(origin_e)$
8: $\quad\quad \mathcal{C}_r \leftarrow \mathcal{C}_r \cup \{\langle e, \tau, S, pattern_e, guard_e \rangle\}$
9: $\quad$ **else**
10: $\quad\quad \mathcal{U}_r \leftarrow \mathcal{U}_r \cup \{e\}$ ▷ unresolved
11: $\quad$ **end if**
12: **end for**
13: $\mathcal{P}_r \leftarrow$ buildRefinementPlan$(\mathcal{U}_r)$
14: **return** $\mathcal{C}_r, \mathcal{P}_r$

$$
\begin{array}{lll}
T & := prim \mid JSONObject \mid \cdots & \textit{Java types} \\
v & := variables & \\
e & := v \mid v.f \mid \textbf{new}\ T(\overline{e}) \mid v.m(\overline{e}) \mid T.m(\overline{e}) \mid (T)\,e & \\
s & := T\ x = e \mid v.f = e \mid v.m(\overline{e}) & \\
 & \mid\ \textbf{evaluateAsync}(e_s) & \textbf{[B4 site]} \\
 & \mid\ \textbf{parseAs}(v_r,\ T) & \textbf{[B1 / B2 / B3 site]}
\end{array}
$$

(a) Java Language Model

$$
\begin{array}{lll}
v & := n \mid s \mid b \mid \textbf{null} \mid \textbf{func}(\overline{x})\ \{\ \textbf{return}\ e\ \} \mid \{\ \overline{str{:}v}\ \} \mid [\,\overline{v}\,] & \\
e & := v \mid x \mid e(\overline{e}) \mid e[e] \mid e[e]{=}e \mid \textbf{delete}\ e[e] & \\
 & \mid\ \textbf{var}\ x{=}e \mid e\ \textbf{op}\ e \mid \textbf{typeof}\ e \mid \textbf{JSON.stringify}(e) & \\
prog & := e_{\text{top}} \qquad \tau(e_{\text{top}}) \xrightarrow{\text{coerce}} \texttt{string} & \\
\multicolumn{3}{l}{\text{If } \tau(e_{\text{top}}) \neq \texttt{string} \Rightarrow \textbf{B1} \text{ or } \textbf{B3} \text{ (if null/undef)}}
\end{array}
$$

(b) JavaScript Language Model

Fig. 1: Abstract language model for the Jetpack `JavaScriptEngine` bridge. **evaluateAsync**/**parseAs** are the two bridge forms (Fig. 1a); e_{top} is the final expression of the JS program whose type is coerced to `String` (Fig. 1b).

Bug	Name	JavaScript condition	Java grammar site	Kind
B1	type-return mismatch	$\tau(e_{\text{top}}) \neq \texttt{string}$ and $\tau \neq \texttt{?}$	$\textbf{parseAs}(v_r, T)$ for any T	Definite
B2	JSON-schema mismatch	$\tau(e_{\text{top}}) = \texttt{string},\ S_{JS} \not\sqsubseteq_{lin} S_{Java}$	$\textbf{parseAs}(v_r, \textit{JSONObject})$	Def. / Guarded
B3	null/undefined propagation	$\tau(e_{\text{top}}) \in \{\texttt{null}, \texttt{undefined}\}$	$\textbf{parseAs}(v_r, T)$ for any T	Definite
B4	arity mismatch	$\lvert\overline{a}\rvert \neq \lvert\overline{x}\rvert$ in call $f(\overline{a})$ inside e_s	$\textbf{evaluateAsync}(e_s)$	Definite

Fig. 2: Bug Taxonomy

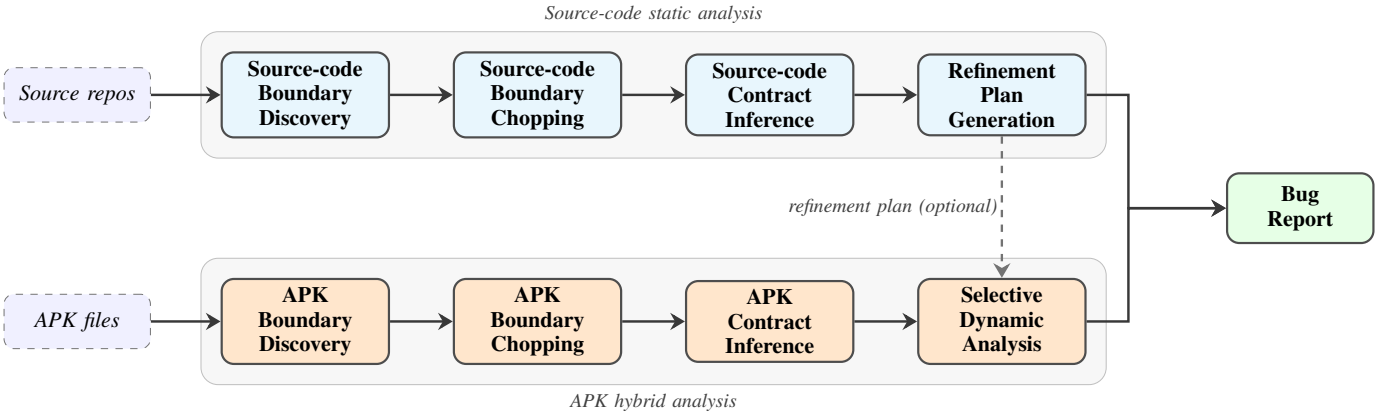

Fig. 3: JETTYPED architecture with source-code and APK analysis tracks.

(1) direct invocations of `evaluateJavaScriptAsync`; (2) Jetpack boundary-management calls such as `JavaScriptSandbox.createConnectedInstanceAsync` and `JavaScriptIsolate.close`; and (3) uses of engine-returned values, such as `Future.get` on a `ListenableFuture<String>` that flows from the engine. The discovered boundary sites are collected into the set E_r for repository r, which is then passed to boundary chopping.

b) Source-code boundary chopping: For each discovered boundary site $e \in E_r$, Algorithm 2 computes a boundary chop $chop_e$ as an interprocedural chop [8] that combines backward and forward slicing over the discovered boundary function. The backward slice follows definitions that contribute to the script argument, wrapper state, and receiver object that reach the boundary. The forward slice follows uses of the returned value through accessor methods (`Future.get()` calls), wrapper-return paths, and downstream consumers. This connects the code that constructs the JavaScript input with the code that eventually interprets the engine result, even when these steps are split across helper methods. The chop is intentionally centered on this boundary, covering script producers and result consumers while excluding unrelated application code. We also consider an object-sensitive chop so that flows through distinct receiver objects remain separated when receiver identity is informative.

We represent the resulting chop as $chop_e = \langle loc_e, prod_e, cons_e, refs_e \rangle$, where loc_e is the boundary source location, $prod_e$ and $cons_e$ are the interprocedural producers and consumers connected by the chop, and $refs_e$ records code references where unsafe argument or result handling occurs. These fields feed directly into contract inference.

c) Source-code contract inference: Over each boundary chop $chop_e = \langle loc_e, prod_e, cons_e, refs_e \rangle$, JETTYPED performs lightweight contract inference to derive $\langle origin_e, pattern_e, guard_e \rangle$. Here, $origin_e$ denotes the recovered script origin, $pattern_e$ summarizes stringification behavior, expected Java/Kotlin result kind, and any arity or schema assumptions, and $guard_e$ records checks against empty or null-like results. These contracts are then mapped to the bug taxonomy: B1 captures type-return mismatches, B2 JSON-schema mismatches, B3 null or undefined propagation, and B4 arity-mismatch risk. Each boundary site is classified as *statically resolved and safe* if the contract is fully determined and no violation is present, *concretely violated* if a B1–B3 violation is statically established, or *unresolved* if static evidence does not suffice to determine the boundary contract.

Algorithm 3 APKSTATICANALYSIS

Require: APK a
Ensure: APK contracts $\mathcal{C}_a$, unresolved crossings $\mathcal{U}_a$
1: $E_a \leftarrow \mathsf{boundaryAnalysisDex}(a)$
2: $\mathcal{C}_a \leftarrow \emptyset$; $\mathcal{U}_a \leftarrow \emptyset$
3: **for all** $e \in E_a$ **do**
4: $\quad chop_e \leftarrow \mathsf{computeBoundaryChopSmali}(e)$
5: $\quad \mathcal{C}_a \leftarrow \mathcal{C}_a \cup \mathsf{extractContracts}(chop_e)$
6: $\quad$ **if** $\mathsf{isUnresolved}(chop_e)$ **then**
7: $\quad\quad \mathcal{U}_a \leftarrow \mathcal{U}_a \cup \{e\}$
8: $\quad$ **end if**
9: **end for**
10: **return** $\mathcal{C}_a, \mathcal{U}_a$

Unresolved sites become candidates for selective dynamic analysis.

d) Refinement plan generation: The source pipeline extracts each unresolved boundary site $e \in \mathcal{U}_r$ into a refinement-plan entry $plan_e = \langle id_e, kind_e, bugs_e, pkg_e, mode_e \rangle$. Here, id_e records the semantic identity of the boundary (repository, fully qualified method, and line number), $kind_e$ is the inferred boundary kind, $bugs_e$ is the hypothesized bug set, pkg_e is the expected package identity, and $mode_e \in \{\texttt{generic}, \texttt{source\_guided}\}$ is the recommended runtime-driving mode. The resulting refinement plan serves as an optional demand oracle for selective dynamic analysis. When both source and APK are available for the same application, the APK pipeline refers to these entries to prioritise boundaries for which static analysis remains incomplete. When the APK mode operates without source, it selects dynamic targets from the underlying static analysis phase.

D. APK Hybrid Analysis

Algorithm 3 expands the APK analysis phase at line 9 of Algorithm 1. The APK mode accepts a compiled Android app and produces contracts and bug reports without requiring source code. It is intended for released binaries and source-incomplete settings, where repository structure may be unavailable or may differ from the shipped artifact. The source pipeline sees repository code as authored, whereas the APK pipeline sees only the packaged artifact after build-time pruning, dependency resolution, shrinking, and bytecode lowering. The APK view is therefore complementary rather than redundant. It can recover shipped boundary crossings and consumer flows even when the corresponding source repository is unavailable, incomplete, or structurally different from the deployed app.

a) APK boundary discovery: Algorithm 3 begins with $\mathsf{boundaryAnalysisDex}(a)$, which scans DEX bytecode for `JavaScriptEngine` boundary sites and builds the initial crossing set E_a. This stage identifies `evaluateJavaScriptAsync` call sites, their associated `Future.get()` consumptions, and downstream parsers such as `Double.parseDouble`, `new JSONObject`, and `Gson.fromJson`. Each recovered crossing is summarized as a bridge candidate that relates the script-side behavior to the Java-side expectation.

Algorithm 4 SELECTIVEDYNAMICANALYSIS

Require: APK contracts $\mathcal{C}_A$, APK unresolved crossings $\mathcal{U}_A$, test drivers $\mathcal{T}$, (optional) source refinement plan $\mathcal{P}$
Ensure: Refined APK contracts $\mathcal{C}_A$
1: $\mathcal{Q} \leftarrow \mathsf{selectTargets}(\mathcal{U}_A, \mathcal{P})$
2: **for all** $q \in \mathcal{Q}$ **do**
3: $\quad e \leftarrow \mathsf{boundaryOf}(q)$
4: $\quad a \leftarrow \mathsf{owningAPK}(e)$
5: $\quad a' \leftarrow \mathsf{instrument}(a, e)$
6: $\quad trace_e \leftarrow \mathsf{execute}(a', \mathcal{T})$
7: $\quad \mathcal{C}_A \leftarrow \mathsf{mergeEvidence}(\mathcal{C}_A, e, trace_e)$
8: **end for**
9: **return** $\mathcal{C}_A$

b) APK boundary chopping: For each $e \in E_a$, the second step computes $chop_e = \mathsf{computeBoundaryChopSmali}(e)$, the APK-side analogue of the boundary chop, by following the low-level def-use structure around engine return values in Smali bytecode. This view is necessary because many misuses are not visible as direct API misuse. They emerge only after the result value traverses several bytecode-level hops (`move-result-object`, field stores and loads, casts, and inter-method returns) before reaching a parser, a callback, or a UI sink. JETTYPED therefore traces each `evaluateJavaScriptAsync` return register through `move`, `check-cast`, and `iput`/`iget` instructions until it reaches the use, producing a per-boundary Smali chop $chop_e$.

c) APK contract inference: The third step applies $\mathsf{extractContracts}(chop_e)$ to each Smali chop and accumulates the resulting contracts in $\mathcal{C}_a$. For each boundary site $e \in E_a$, the extracted contract summarizes wrapper-return structure, consumer-site patterns, and whether the crossing is *Statically Resolved (SR)* or *Dynamically Resolved (DR)*, depending on whether the script argument can be fully determined by interprocedural constant propagation and string folding. The unresolved set is then defined as $\mathcal{U}_a = \{e \in E_a \mid \mathsf{isUnresolved}(chop_e)\}$. The static APK stage therefore produces two aligned outputs: contracts $\mathcal{C}_a$ and unresolved crossings $\mathcal{U}_a$.

d) Selective dynamic analysis: Algorithm 4 expands the selective dynamic-analysis phase at line 13 of Algorithm 1. It targets only unresolved sites after static analysis. Thus, dynamic execution is used to resolve static uncertainty, not to re-run the whole analysis. The algorithm first forms the target set $\mathcal{Q} = \mathsf{selectTargets}(\mathcal{U}_A, \mathcal{P})$ by combining source-guided refinement-plan entries (if any) with APK-side unresolved crossings. For each target $q \in \mathcal{Q}$, it resolves the underlying boundary $e = \mathsf{boundaryOf}(q)$, instruments that boundary, executes the instrumented app under the test drivers $\mathcal{T}$, and merges the resulting runtime trace $trace_e$ back into $\mathcal{C}_A$.

Instrumentation is restricted to `JavaScriptEngine` boundaries, wrapper-return paths, and immediate consumer sites. The recorded runtime evidence is used only to refine contracts that static analysis could not resolve. When source-level refinement plans are available, $\mathcal{P}$ narrows $\mathcal{Q}$ toward semantically prioritised boundaries; otherwise, $\mathcal{Q}$ is derived entirely from $\mathcal{U}_A$. A DR site not exercised during dynamic analysis retains

$$n \vdash \texttt{number} \qquad s \vdash \texttt{string} \qquad \mathbf{null} \vdash \texttt{null}$$

$$\frac{e \vdash \tau}{\mathbf{JSON.stringify}(e) \vdash \texttt{string}}\ (\textsc{Stringify})$$

$$\frac{e \vdash \tau}{\mathbf{typeof}\ e \vdash \texttt{string}}\ (\textsc{Typeof})$$

$$\frac{\forall i.\ e_i \vdash \tau_i}{\{\overline{k_i{:}e_i}\} \vdash \texttt{object}(\{\overline{k_i{:}\tau_i}\})}\ (\textsc{Obj})$$

$$\frac{e \vdash \texttt{object}(S) \qquad (k : \tau') \in S}{e[k] \vdash \tau' \sqcup \texttt{undefined}}\ (\textsc{Acc})$$

$$\frac{e_1 \vdash \tau_1 \qquad e_2 \vdash \tau_2}{e_1\ \mathsf{op}\ e_2 \vdash \tau_1 \sqcup \tau_2}\ (\textsc{Op}) \qquad \frac{\text{computed access or } \texttt{eval}}{e \vdash \top}\ (\textsc{Dyn})$$

Fig. 4: Representative type inference rules for SR-site scripts.

$$S ::= \epsilon \mid (k : \tau) \cdot S \qquad S \sqsubseteq_{lin} \epsilon\ (\text{L-}\textsc{Empty})$$

$$\frac{(k : \tau') \in S_1 \qquad \tau' \sim \tau \qquad S_1 \setminus k \sqsubseteq_{lin} S_2}{S_1 \sqsubseteq_{lin} (k : \tau) \cdot S_2}\ (\text{L-}\textsc{Field})$$

Fig. 5: Linear JSON structural subtyping rules for B2 checking.

type ? and remains unresolved rather than being flagged as a concrete violation.

E. Bug reporting

At line 16, Algorithm 1 enters bug reporting. For each SR site in either pipeline, we extract the embedded script and apply flow-sensitive type inference over the lattice $\bot \sqsubset \texttt{null} \sqsubset \texttt{undefined} \sqsubset \texttt{number} \sqsubset \texttt{string} \sqsubset \texttt{object}(S) \sqsubset \top$, where $\texttt{object}(S)$ is parameterised by the schema S. Dynamic constructs (e.g., computed property access, `eval`) are conservatively approximated to $\top$.

Figure 4 gives representative rules for the cases most relevant to bug reporting: (ACC) exposes B3 through missing-key access, while (STRINGIFY) and (TYPEOF) produce `string` and therefore do not trigger B1. (OBJ) constructs object schemas fieldwise, (OP) propagates uncertainty through joins of operand types, (DYN) captures imprecise dynamic features by mapping them to $\top$, and the literal judgments initialize the base cases for numbers, strings, and `null`.

Once the boundary contracts are populated by either pipeline, JETTYPED applies type-consistency checking [9] to convert inferred contracts into bug reports. Two types are *consistent* ($\tau_1 \sim \tau_2$) if $\tau_1 = \tau_2$ or either is ?. For JSON-returning crossings, JETTYPED checks $S_{JS} \sqsubseteq_{lin} S_{Java}$ using the inference rules in Fig. 5, where each Java-required field must match exactly one JSON field.

Given a bridge contract (τ_{JS}, T_{Java}), JETTYPED flags **B1** when $\tau_{JS} \notin \{\texttt{string}, ?\}$, **B2** when $\tau_{JS} = \texttt{string}$ but $S_{JS} \not\sqsubseteq_{lin} S_{Java}$, **B3** when $\tau_{JS} \in \{\texttt{null}, \texttt{undefined}\}$ and no empty-string guard is present, and **B4** when a call $f(a_1, \ldots, a_n)$ has $n \neq$ the declared arity. When $\tau_{JS} = ?$, JETTYPED records the site as a guarded crossing rather than a concrete bug, deferring final type resolution to runtime evidence. These rules are semantically shared across both pipelines; the source pipeline applies them over inferred source-level contracts, while the APK pipeline applies them over DEX-level bridge summaries merged with dynamic evidence.

IV. EVALUATION

We evaluate the prevalence and severity of cross-language type misuse in Android's Jetpack `JavaScriptEngine` through four research questions.

RQ1 (Adoption). How widely and in which domains is the Jetpack `JavaScriptEngine` API adopted across open-source Android apps?

RQ2 (Violation Landscape). What cross-language type violations exist, and how are they distributed across bug categories and application domains?

RQ3 (Binary and Runtime Evidence). How does static analysis of APKs compare with source code analysis? Can dynamic execution confirm boundary reachability in statically unresolved cases, and do the reported violations reflect real bugs in practice?

RQ4 (Comparison with Existing Tools). Do existing analyses detect the `JavaScriptEngine` bugs found by JETTYPED?

A. Experimental Setup

a) Subject selection: Our aim was to curate a corpus of Android apps and projects that leverage Jetpack `JavaScriptEngine` API. To this end, we conducted a multi-source search spanning approximately seven days, using 8 custom scripts and 211 distinct search queries across ten platforms. Figure 6 summarises our collection pipeline. The search strategy had two goals: (1) to enumerate open-source apps on GitHub, and (2) to explore widely-used Android app hosting platforms.

To search GitHub, we used three classes of queries against the GitHub Search API. First, code queries targeting method names (`evaluateJavaScriptAsync`, `createConnectedInstanceAsync`), fully qualified class references (`JavaScriptSandbox`, `JavaScriptIsolate`, `androidx.javascriptengine`), and build-file identifiers such as `androidx.javascriptengine:javascriptengine` in `build.gradle`, `.kts`, and `.toml` files. Second, repository keyword queries combining natural-language and identifiers. Finally, topic queries focusing on curated repository tags. Since the GitHub search API caps code search results at 1,000 per query, we partitioned each core query by repository creation date into 12 disjoint windows spanning 2020 to 2026, ensuring full temporal coverage. The initial search phase (22 code and build-file queries) yielded 199 candidate repositories; the subsequent exhaustive phase (42 repository-keyword, 23 code, and 6 topic queries) yielded 157 further candidates. After merging and deduplicating these two GitHub phases, the GitHub enumeration produced 297 candidate repositories.

We then applied a *true-consumer* criterion: a repository qualifies if it contains at least one executable `evaluateJavaScriptAsync` call site together with at least one

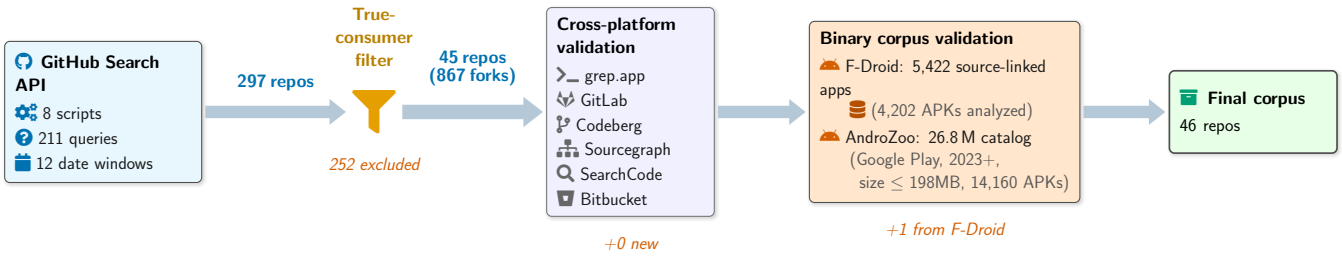

Fig. 6: Data collection pipeline.

`JavaScriptSandbox` or `JavaScriptIsolate` class reference, or a direct `import androidx.javascriptengine` statement. This filter excludes repositories that declare the library as a transitive dependency without invoking it directly, and repositories that reference the string `evaluateJavaScript` only through the unrelated `WebView` API. Applying the criterion and deduplicating across GitHub results yielded 45 true consumers. We also enumerated 867 forks of these repositories via the GitHub fork API; none contained unique consumer code.

Next, we repeated the core queries on six additional platforms: grep.app, GitLab, Codeberg, Sourcegraph, SearchCode, and Bitbucket, together with a Maven Central dependency-graph lookup. None of these sources revealed a new repository. We further explored two binary corpora. We first scanned the F-Droid open-source catalog [10] in two passes. First, we retrieved all 5,749 metadata .yml files, of which 5,422 linked to GitHub repositories. Grepping these source trees for the DEX bytecode signatures (`Landroidx/javascriptengine/` and `evaluateJavaScriptAsync`) yielded no new apps. In the second pass, we downloaded 4,202 F-Droid APKs (4,194 downloaded successfully) and inspected them for the same signatures. This scan identified one additional app not present in the GitHub enumeration, bringing the total to 46.

Finally, we queried the full 26.8 million-entry AndroZoo dataset [11], but restricted the scan to Google Play APKs from 2023 onward, since Jetpack `JavaScriptEngine` was introduced in late 2022. We then applied the same DEX-signature checks. The main binary sweep analyzed 14,160 APKs over 25.8 hours and yielded no new apps beyond the 46 already identified. Taken together, the GitHub crawl, cross-platform checks, and binary-corpus validation yielded a final corpus of 46 repositories and 867 forks.

b) Dataset: For the 46 final repositories, we classify each boundary target into four code-provenance categories: *project code* (application logic written by the project authors), *project-internal helper/library code* (helper or wrapper modules bundled with the project), *third-party library*, and *test/framework infrastructure*. For the main violation counts in RQ2, we restrict to the first two categories and exclude third-party libraries and test/framework code. The 46 repositories span several primary languages, including Kotlin, Java, C++, TypeScript, Dart, and C#. At the time of collection, GitHub star counts range from 0 to 23,318 (Chromium); among the remaining repositories, ten exceed 100 stars, including `androidx/androidx` (5,944), `anddea/revanced-patches` (1,587), and `cwuom/NeriPlayer` (1,023). Of the 44 repositories with available creation metadata, 25 (57%) were created in 2024 or later, consistent with the API's introduction in late 2022. We executed JETTYPED's source-code analysis pipeline on all 46 repositories for RQ1 and RQ2, with an average static-analysis time of 1.58 s per repository.

TABLE II: Distribution of `JavaScriptEngine` consumers by domain (excluding one decompiled-source repository). *Repos* = repositories with confirmed API usage; *Aff. repos* = repositories with ≥1 bug in RQ2; *Bugs* = unchecked boundaries or wrappers; *Loc.* = affected program locations.

Domain	Repos	Aff. repos	Bugs	Loc.
Browser / platform	14	0	0	0
Crypto / finance	4	4	9	42
Media	4	4	6	16
Productivity / education	5	5	5	32
Utility / developer tools	5	5	7	15
Framework / test infra	9	0	0	0
Total	**41**	**18**	**27**	**105**

For RQ3, we analyzed APKs corresponding to repos in the survey. Not all 46 repositories could contribute here, because many do not directly produce a standard APK and instead require non-standard build or packaging workflows. To keep the source and APK-level views comparable, we used APKs built from the same sources as the source-code analysis whenever possible. As a result, the repository corpus subsumes the APK corpus in terms of available code, but the two views still need not match one-to-one: the source pipeline sees the full repository, whereas the APK pipeline sees only the packaged artifact, which may exclude test code, benchmark modules, or other non-packaged paths.

The evaluation was executed on a commodity laptop running macOS 26.4, with an Apple M3 Max CPU and 36 GB of RAM.

B. RQ1: Adoption and Usage Patterns

JETTYPED confirmed active `JavaScriptEngine` API calls in **42 of the 46 analyzed repositories** (91.3%). The remaining four repositories referenced JSEngine-related identifiers only in build manifests or documentation strings, with no executable `evaluateJavaScriptAsync` call sites. One of the 42 confirmed consumers (Mohsenabn78/macro-reverse-engineering) consists of source code produced by decompiling a production APK; we retain it in the prevalence count but exclude it from the source-analysis results in RQ2.

TABLE III: Recurring `JavaScriptEngine` usage patterns in the 32 consumer repositories of the study corpus.

Pattern	Repos	Share
Wrapper-based engine access	22	69%
Capability checks via `isFeatureSupported`	21	66%
Result stringification via `JSON.stringify`	18	56%
Named data via `provideNamedData`	17	53%
Structured JSON-based exchange	15	47%
Console monitoring via `setConsoleCallback`	15	47%

Table II shows the distribution of the 41 remaining repositories across application domains. Browser and platform repositories account for the largest group (14 repositories), mostly Chromium/Cobalt-derived projects that embed the `android_webview` integration-test suite. Crypto-finance applications account for four repositories. Cosmostation and a derived fork use `JavaScriptIsolate` to execute transaction-signing scripts, while AirGap Vault and AirGap Wallet employ a sandboxed JS environment to protect key-derivation logic. The remaining repositories span media players and patch tools (4), productivity and education applications (5), utility or developer-facing projects (5), and framework or test infrastructure (9). The latter use JSEngine exclusively within test harnesses or framework adapters; their call sites reside in `benchmark_or_test` or `framework_or_sdk_source` categories and fall outside the main violation scope examined in RQ2.

a) Usage patterns: Beyond raw adoption, we characterized how the 32 repos employ the API by classifying each of 831 boundary targets in project/internal code into usage archetypes. Table III summarises the dominant patterns, where three observations stand out. First, repositories rarely invoke the engine as an isolated one-off API call: 22 of 32 place engine access behind wrappers, and 21 of 32 guard usage with `isFeatureSupported`. This suggests that developers already perceive the API as feature-sensitive infrastructure that must be carefully embedded into application logic.

Second, result handling is often made explicit. More than half of the repositories stringify results before returning them across the boundary, and roughly half rely on named data or structured JSON exchange. This suggests that developers are actively compensating for the API's string-only return channel by building their own conventions for data transport. Third, monitoring and debugging are not edge cases: 15 repositories wire in `setConsoleCallback`, showing that script execution is often treated as a maintained runtime component rather than as a hidden implementation detail.

b) Script origins and consumer patterns: The origin of the evaluated JavaScript script determines the effectiveness of static analysis. Across the 831 boundary targets in project/internal code, 662 (80%) involve dynamically constructed or runtime-supplied scripts. Named-data provisioning accounts for 123 targets (15%), inline literals for 41 (5%), and asset/file scripts for 5 targets (<1%). On the consumer side, JSON parsing and RPC construction are the two most common consumption patterns (39 targets each across 15 repositories), followed by UI rendering (15) and string coercion (7). This distribution has two consequences. First, the high proportion of dynamic origins directly motivates the selective dynamic refinement stage described in Section III-D. Second, dynamic script construction does not always preclude static reasoning: many such scripts remain recoverable through constant propagation and string folding, which is why static analysis can still cover a substantial portion of the boundary surface. Section IV-C separates originating bugs from the affected locations where they later appear. In total, 773 of these 831 targets are statically resolved and amenable to contract inference; the remaining 58 are included in the 210-target dynamic refinement plan used in RQ3.

Answer to RQ1. The `JavaScriptEngine` API is confirmed in 91.3% of surveyed repositories (42/46) and spans six repository domains. Developers use it as a computation bridge, commonly placing engine access behind wrappers, guarding invocation with capability checks, and combining it with explicit result stringification and named data exchange. Although 662 of the 831 project/internal targets involve dynamically constructed or runtime-supplied scripts, 773 targets are still statically resolved. The unresolved targets motivate selective dynamic refinement.

C. RQ2: Violation Landscape

a) Detection results: The reported bugs arise when an unchecked `evaluateJavaScriptAsync` boundary or wrapper returns a result that can violate the expected Java/Kotlin-side contract. The same result can then be used at several other locations, such as parsing, UI rendering, RPC construction, or transaction handling. Thus, we report the boundary or wrapper as the bug, and the propagations as affected program locations.

Across the source-code corpus, JETTYPED found 27 bugs in 18 repositories, affecting 105 program locations. These bugs cover B1, B2, and B3. B4 requires the recovered script to expose both a local callee definition and a matching call site; we observed no such cases in the current corpus. Targets whose script body cannot be recovered statically are emitted to the dynamic refinement plan. Sites with an explicit empty-string check are treated as guarded, while null-only checks remain bugs because the API returns `""` for non-string JavaScript values. Table IV reports the bugs by application, together with their affected locations.

b) Bug categories: Across the 105 affected program locations, B3 is the largest category (65 locations, 61.9%). These cases arise when wrappers return engine results without guarding against empty-string outcomes. B1 accounts for 34 locations (32.4%) and appears when JavaScript expressions can produce non-string results at a boundary that the Java/Kotlin side still treats as a valid `String`. B2 accounts for six locations (5.7%), where structured engine output reaches JSON parsing or RPC construction with an incompatible expected shape. We observed no B4 case in the current corpus.

c) Dynamic refinement plan: Static analysis leaves 210 boundary targets unresolved across 23 repositories because the script body is available at runtime. JETTYPED records

TABLE IV: Source-code bugs and their affected locations (application/internal code only). *Bugs* = unchecked `JavaScriptEngine` boundaries or wrappers; *Loc.* = affected program locations reached by those bugs.

Application	Bugs	Loc.
Crypto / finance		
Cosmostation	4	20
Cosmostation (fork)	3	14
AirGap Vault	1	4
AirGap Wallet	1	4
Media / patch tools		
NeriPlayer	1	5
YTStream	1	1
ReVanced patches	2	5
Morphe patches	2	5
Productivity / education		
NeutriNote	1	9
NeutriNote fork	1	9
EduCapture	1	8
Wulkanowy	1	3
DanXi	1	3
Utility / developer tools		
HID Barcode Scanner	1	3
AndroidxTypescriptRunner	2	3
jsengine-bug-demo	2	4
nowavewater/JavaScriptEngine	1	3
Android JSEngine Benchmark	1	2
Total	27	105

these sites in a **dynamic refinement plan** for the APK phase (Section IV-D). Only 58 targets belong to project/internal code; the rest are in test, framework, and third-party boundaries. Three subject groups with no violations in the project code account for 66 such targets: `keiji/jp2k-decoder-android`, AndroidX, and the two AdServices forks combined.

Answer to RQ2. JETTYPED found 27 bugs in 18 repositories, affecting 105 program locations. B3 is the largest category with 65 locations, followed by B1 with 34 and B2 with six. The source pipeline also emitted a 210 target dynamic refinement plan.

D. RQ3: Binary and Runtime Evidence

We use the APK pipeline as a complementary binary-side analysis rather than as a second ecosystem. It analyzed 25 compiled Android apps and identified Jetpack `JavaScriptEngine` usage in 16 of them (12 unique packages). Fifteen contain at least one statically detectable affected program location, resulting in 89 affected program locations and 187 bridge contracts. The static phase averaged 159.4 s per APK. The dominant issue categories are wrapper-mediated result passthrough and type-return mismatch, together accounting for over 80% of detected sites. Cosmostation yielded 31 APK-level affected program locations, compared to 20 in the source analysis. This is due to bytecode-level tracking, which exposes more downstream uses of raw engine results through wrapper chains.

a) Alignment with source analysis: Across the overlapping source and APK artifacts, the APK analysis is largely consistent with the source analysis. The counts are close for Android-Javascript-Engine-Benchmark (2 vs. 2), AndroidxTypescriptRunner (3 vs. 4), and NeriPlayer (5 vs. 7). The APK analysis exposes more consumer-side locations in Cosmostation (20 vs. 31) and the two AirGap applications (4 vs. 8 in both cases). At the same time, it under-approximates source-visible repository paths in NeutriNote (9 vs. 2) and EduCapture (8 vs. 2). These differences mainly reflect a representation gap rather than a contradiction. The source pipeline accesses the full repository, including test, benchmark, bundled, and other non-packaged code, whereas the APK pipeline sees only what is shipped in the compiled app.

TABLE V: Observed consequences in developer-reported cases.

Impact	Effect	#
Transaction/RPC failure	crash, malformed RPC/transaction payload, or invalid signature	3
Incorrect app output	empty UI value, empty scanner output, or broken stream result	3
Parsing failure	parse exception or malformed structured result	2
Lost failure signal	ambiguous fallback, swallowed error, or missing failure signal	3
Silent propagation	empty or invalid result returned to caller without validation	7

b) Selective dynamic analysis: Selective dynamic analysis is a follow-up stage for unresolved APK-side targets. In the dynamic run, we obtained app-specific dynamic success for 8 apps. These runs provide concrete runtime confirmation of the statically predicted boundary behavior. In Cosmostation, the instrumentation recorded five `evaluateJavaScriptAsync` invocations, six executions of affected locations along wrapper-return paths, and one bridge contract that could not be resolved statically. In NeriPlayer, the run reached the `YouTubeEjsChallengeSolver` boundary and confirmed the predicted affected program location. Both NeutriNote variants reached the notes-editor JavaScript path and confirmed one B1-affected program location each. EduCapture reached JavaScriptEngine invocations but did not confirm an affected program location, and therefore remains an open dynamic target.

c) Manual validation and impact: In addition to the dynamic analysis, three authors manually inspected all 27 source-level bugs. For each bug, each author independently checked the `JavaScriptEngine` boundary or wrapper, the expected return contract, the downstream consumer, the guard condition, and the affected locations reached by the unchecked result. A bug was kept in the final source-level count only when all the authors agreed on this evidence. We then applied a stricter criterion for reporting it to the developer: we reported cases we could reproduce or for which we had concrete evidence that made the bug actionable for developers. This resulted in 18 bug reports. For the remaining nine bugs, we observed the boundary-contract problem in the code, but did not report them because we could not produce a sufficiently concrete reproduction by the time of submission. Developers acknowledged or acted on seven of them: four resulted in code changes, one was planned for a later release, and two were closed as completed. One report was closed as not planned because the developer considered the return value intentionally unused. The remaining reports were not rejected at the time of submission. We also classified the reported cases by their observable consequence. Table V shows the breakdown.

d) Case study: Cosmostation: Cosmostation is a production cryptocurrency wallet (100K+ downloads) that manages Solana, Bitcoin, and Aptos transactions through three

dedicated JavaScript wrapper classes (`SolanaJs`, `BitcoinJs`, `AptosJs`). These wrappers use `evaluateJavaScriptAsync` for cryptographic operations such as transaction signing, address derivation, and transfer-payload serialization. They then return the raw result to the Kotlin consumer code. The source pipeline found four bugs in these wrappers, affecting 20 program locations across B1 and B3. At the source level, the common pattern is that the wrappers guard against a `null` isolate reference, but not against the empty string that JSEngine returns for non-string JavaScript values. Manual inspection and the APK pipeline show that downstream code parses some results with `Gson().fromJson` or passes serialized transaction strings into RPC request parameters. Consequently, a JavaScript evaluation failure in a signing path does not raise an exception; instead, an empty string reaches `fromJson`. This can produce a malformed transaction payload or an invalid signature submitted to the blockchain RPC endpoint.

The APK pipeline found 31 affected locations, 11 more than the source analysis. The additional sites arise from bytecode-level back tracking. The APK analysis reveals additional downstream uses of raw engine results via wrapper chains. Dynamic instrumentation recorded five engine invocations, six executions of affected locations along wrapper-return paths, and one statically unresolved bridge contract. We reported this case, and the developer accepted the bug and fixed the wrapper layer by checking null and empty `JavaScriptEngine` results.

e) Case study: NeutriNote: NeutriNote is a note-taking app that evaluates user-supplied JavaScript through `Utils.cliEvalJS`. In interactive mode, the app wraps the script in a string-conversion template. In snippet, inline, and file modes, it sends the JavaScript expression directly to `evaluateJavaScriptAsync`. If the expression returns a number or an object, JSEngine returns the empty string. The source pipeline found one bug in `cliEvalJS`, affecting nine program locations. One is the direct boundary misuse in `cliEvalJS`. The others arise in `DisplayDBEntry.expandShortcut`, where the same result flows to UI, string conversion, numeric parsing, JSON parsing, and RPC construction without an empty-string guard. This produces a visible user-facing failure, e.g., a command such as `5+5` yields an empty result instead of `10`.

The APK pipeline found two affected program locations. This under-approximates the source view because only the packaged notes-editor path remains visible in the compiled artifact. The dynamic run reached that path and confirmed one B1-affected program location. We reported this case, and the developer acknowledged it and planned a fix for a later release.

Answer to RQ3. The APK pipeline found 89 affected locations. Dynamic refinement yielded app-specific success for eight apps, including value-flow confirmation in Cosmostation. We reported 18 high-confidence cases; seven were confirmed or acted on, and four already led to code changes.

E. RQ4: Existing Analysis Tools

We ran PMD [19], SpotBugs [17] and SonarQube [18] on the source-code corpus. We also considered state-of-the-art Android cross-language analyzers published after the introduction of `JavaScriptEngine`: IWanDroid [5], ωTest [4], and WebViewJSdetect [6]. At the time of writing, ωTest was not publicly available. We used each tool with its expected input: IWanDroid analyzes APKs, whereas WebViewJSdetect analyzes JavaScript, so we ran IWanDroid on the APK subjects and WebViewJSdetect on the snippets passed to `evaluateJavaScriptAsync`. None of these tools detected the bugs found by JETTYPED. This result is expected, as these bugs are not visible as Java type errors, JavaScript syntax errors, or Android API misuses. From the Java/Kotlin side, the result still appears to be a valid `String`, even when the engine has silently converted a non-string JavaScript value to `""`.

Answer to RQ4. Existing analysis tools did not detect the bugs found by JETTYPED. General-purpose tools do not model the hidden `JavaScriptEngine` return contract, while Android cross-language tools target WebView-style bridges.

F. Threats to Validity

Internal validity: The source code analysis is susceptible to both false positives and false negatives. For example, when runtime guards are present but not captured by the model, or when the JavaScript code cannot be fully recovered. The dynamic analysis phase partly mitigates this by re-examining unresolved sites under concrete execution, but it covers only the APK subjects we validated dynamically. Finally, B4 is reported only when the recovered script exposes both the JavaScript callee definition and the corresponding call site. The current corpus, therefore, may under-approximate arity mismatches.

External validity: The dataset covers all 46 publicly identifiable open-source consumers of the Jetpack `JavaScriptEngine` API, plus 867 forks, confirmed through GitHub enumeration, complementary code-search platforms, and two binary corpora. Because the API was introduced in late 2022 and is still in an early adoption phase, the results may not generalize to future consumers or to other Java/JavaScript bridge APIs such as WebView's `addJavascriptInterface`. They also may not generalize to closed-source applications that are not publicly recoverable through our collection pipeline. The underlying analysis ideas, i.e., boundary type-coercion checking, null-propagation detection, and schema mismatch identification, are broader than this API.

V. RELATED WORK

To the best of our knowledge, no prior work studies the specific cross-language type bugs induced by Jetpack `JavaScriptEngine`'s string-coercing boundary contract. Our closest connections are to hybrid-app analysis, host-language bug detection, and JavaScript static and dynamic analysis. HybriDroid [2] builds a cross-language call graph for `WebView.addJavascriptInterface`; JN-SAF [3] extends taint analysis across the same bridge; and Dual-Force [12] applies cross-language forced execution to WebView malware. Tiwari et al. [13] study hybrid-app adoption and later propose demand-driven information flow analysis for WebView [5]. Hu et al. [4] present ωTest for WebView-oriented testing, and Yuan et al. [6] detect JavaScript vulnerabilities in WebView via concurrent

abstract interpretation. These systems target bidirectional callback bridges with explicit interface objects and cross-language call edges. By contrast, Jetpack `JavaScriptEngine` exposes a one-directional, string-typed boundary in which Java submits a script and failures surface later in host-language consumers. JETTYPED therefore targets a different cross-language analysis problem. Outside Android, HarmoBridge [14] studies cross-language static analysis for ArkTS/C++, PolyCruise [15] performs dynamic cross-language information-flow analysis, and ReUnify [16] analyzes JavaScript/Java interactions in React Native apps; none targets this string-coercing API boundary.

SpotBugs [17] analyzes Java bytecode, while SonarQube [18] and PMD [19] provide general-purpose source-code analysis across multiple languages. They do not model the cross-language contract of `evaluateJavaScriptAsync`, so they miss the bridge bugs studied here. On the research frontier, TAJS [20] performs whole-program static analysis for JavaScript, while Jalangi2 [21] provide dynamic instrumentation frameworks for JavaScript execution. These systems reason about JavaScript itself, but do not model how Android host code constructs scripts, invokes the engine, and then interprets the returned string. JETTYPED instead combines lightweight JavaScript-side inference over recovered script snippets with Java-side contract extraction over source code and bytecode, so that the reported bugs are tied to the cross-language boundary rather than to JavaScript behavior in isolation.

VI. CONCLUSION

In this work, we presented JETTYPED, a hybrid program analysis framework for detecting cross-language type bugs at the Android Jetpack `JavaScriptEngine` boundary. The API's silent coercion of non-string JavaScript values to `""` underlies the bug categories studied in this paper, which are invisible to both the Java compiler and the JavaScript interpreter. Our study shows that these bugs already appear in this still-young ecosystem: the source pipeline found 27 bugs in 18 repositories, affecting 105 program locations. The complementary APK analysis and selective dynamic refinement confirmed that these bugs are observable in compiled apps.

ACKNOWLEDGEMENTS

This work is funded by Digital Research Centre Denmark (DIREC), National Defence Technology Centre (NFC) and Danish Industry Foundation (project 2025-0766).

REFERENCES

[1] Google LLC, "Run javascript and WebAssembly — Jetpack JavaScriptEngine," https://developer.android.com/develop/ui/views/layout/webapps/jsengine, 2024, accessed 2025.

[2] S. Lee, J. Dolby, and S. Ryu, "Hybridroid: static analysis framework for android hybrid applications," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 250–261. [Online]. Available: https://doi.org/10.1145/2970276.2970368

[3] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang, "Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1137–1150. [Online]. Available: https://doi.org/10.1145/3243734.3243835

[4] J. Hu, L. Wei, Y. Liu, and S.-C. Cheung, "ωtest: Webview-oriented testing for android applications," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 992–1004. [Online]. Available: https://doi.org/10.1145/3597926.3598112

[5] A. Tiwari, J. Prakash, and C. Hammer, "Demand-driven information flow analysis of WebView in Android hybrid apps," in *Proc. 34th IEEE ISSRE*. IEEE, 2023, pp. 415–426.

[6] Z. Yuan, Z. Yang, J. Tan, and H. Zhang, "Webviewjsdetect: Javascript vulnerability detection in android webview via coverage-guided thread-adaptive concurrent abstract interpretation," *Computer Networks*, vol. 275, p. 111908, 2026. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128625008746

[7] Google LLC, "JavaScriptIsolate.evaluateJavaScriptAsync(String) api reference," https://developer.android.com/reference/kotlin/androidx/javascriptengine/JavaScriptIsolate, 2024, accessed 2026.

[8] T. Reps and G. Rosay, "Precise interprocedural chopping," in *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. SIGSOFT '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 41–52. [Online]. Available: https://doi.org/10.1145/222124.222138

[9] J. G. Siek and W. Taha, "Gradual typing for functional languages," in *Scheme and Functional Programming Workshop*, 2006, pp. 81–92.

[10] F-Droid Contributors, "F-Droid: Free and open source Android app repository," https://f-droid.org, 2026.

[11] 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: Association for Computing Machinery, 2016, p. 468–471. [Online]. Available: https://doi.org/10.1145/2901739.2903508

[12] Z. Tang, J. Zhai, M. Pan, Y. Aafer, S. Ma, X. Zhang, and J. Zhao, "Dual-force: understanding webview malware via cross-language forced execution," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 714–725. [Online]. Available: https://doi.org/10.1145/3238147.3238221

[13] A. Tiwari, J. Prakash, S. Groß, and C. Hammer, "A large scale analysis of android — web hybridization," *Journal of Systems and Software*, vol. 170, p. 110775, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220301898

[14] J. Wu, J. Deng, Y. Zhao, L. Li, and H. Wang, "Harmobridge: Bridging arkts and c/c++ for cross-language static analysis on harmonyos," in *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2025, pp. 3168–3179.

[15] W. Li, J. Ming, X. Luo, and H. Cai, "PolyCruise: A Cross-Language dynamic information flow analysis," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 2513–2530. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/li-wen

[16] Y. Liu, X. Chen, P. Liu, J. Grundy, C. Chen, and L. Li, "Reunify: A step towards whole program analysis for react native android apps," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 1390–1402.

[17] SpotBugs Contributors, "SpotBugs: Find bugs in Java programs," https://spotbugs.github.io, 2016.

[18] SonarSource SA, "SonarQube: Continuous inspection of code quality," https://www.sonarsource.com/products/sonarqube, 2007.

[19] PMD Contributors, "PMD: An extensible cross-language static code analyser," https://pmd.github.io, 2002.

[20] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for javascript," in *Static Analysis*, J. Palsberg and Z. Su, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 238–255.

[21] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: a selective record-replay and dynamic analysis framework for javascript," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 488–498. [Online]. Available: https://doi.org/10.1145/2491411.2491447