Dynamic Multidex Loading for Cross-Platform SDK Integration: Solving the Android 65K Method Limit in Mobile Ad Mediation

Nazar Kozak
Appodeal Inc.
nzrkzk@gmail.com

Abstract

Mobile ad mediation SDKs aggregate numerous third-party advertising networks, routinely exceeding Android's 65,536-method DEX file limit. In 2015, we developed Dexter, a dynamic multidex loading system that pre-compiles each third-party SDK into a separate DEX file and loads them at runtime via reflection-based ClassLoader manipulation. This approach bypassed the method limit years before official Android multidex support matured in popular game engines such as Unity, Corona, and PhoneGap. Dexter enabled Appodeal's ad mediation SDK to integrate over 80 ad networks across more than 20 development platforms, coinciding with the company's $3.1M Series A funding round. We describe the architecture, the reflection-based DEX injection mechanism, and the practical impact on the mobile advertising ecosystem. The software demonstrates that runtime module loading through ClassLoader reflection is a viable strategy for circumventing bytecode format constraints in plugin-heavy mobile SDKs.

Keywords: Android, multidex, dynamic class loading, ad mediation, mobile SDK, DEX, cross-platform development

1 Motivation and Significance

The Android runtime executes bytecode stored in the Dalvik Executable (DEX) format, a compact representation optimized for memory-constrained mobile devices [Bornstein, 2008]. A fundamental constraint of the DEX format is that method references are indexed using a 16-bit integer, imposing a hard upper bound of 65,536 methods per DEX file [Android Open Source Project, 2024b]. While this limit was generous for single-purpose applications in the early Android era, the growth of third-party library ecosystems rapidly made it a practical obstacle for any application of moderate complexity [Li et al., 2016].

The advertising technology sector encountered this limitation with particular severity. Mobile ad mediation SDKs serve as aggregation layers that unify access to multiple ad networks—AdMob, Facebook Audience Network, Unity Ads, AppLovin, Chartboost, Vungle, and dozens of others—within a single integration point [Nath, 2015, Gui et al., 2015]. Each ad network SDK contributes thousands of methods to the application's method count. An ad mediation layer integrating 10–15 networks easily exceeds 65,536 methods from the ad SDKs alone, before even counting the host application's own code [Gui et al., 2016].

Google introduced an official multidex support library in 2014 [Android Developers, 2024a], which allows applications to split their code across multiple DEX files at build time. However, this solution assumed full control over the application's build pipeline and startup sequence—assumptions that did not hold for cross-platform game engines. In 2015, the dominant mobile game development platforms—Unity, Corona SDK, PhoneGap/Cordova, and others—had no stable

support for the Android multidex configuration [Heitkötter et al., 2013, Unity Technologies, 2018]. Unity, the single most important distribution channel for mobile games, did not provide reliable built-in multidex support until 2018 [Biorn-Hansen et al., 2018]. Corona SDK and Apache Cordova similarly lagged behind [Corona Labs, 2017, Apache Software Foundation, 2017].

This created a critical blocker for the mobile advertising industry. Game developers—who represent a disproportionately large share of ad-monetized mobile applications—could not integrate full-featured ad mediation SDKs into their projects. Any mediation provider that could solve this problem would gain a decisive competitive advantage in the cross-platform game development market.

We developed DEXTER, a dynamic multidex loading system that circumvents the 65K method limit by pre-compiling each third-party ad network SDK into a separate DEX file and loading them on demand at runtime. Rather than relying on the build system to merge all code into a set of DEX files, DEXTER treats each ad network as an independent module that is injected into the application's ClassLoader through Java reflection. This approach requires no modifications to the host game engine's build pipeline, making it compatible with any cross-platform framework that supports bundling asset files.

The practical impact of DEXTER was substantial. It enabled Appodeal, a mobile ad mediation company, to integrate over 80 ad networks into a single SDK that functioned seamlessly across more than 20 development platforms [Appodeal Inc., 2015]. The Unity plugin built on DEXTER became Appodeal's flagship product, accounting for approximately 80% of company revenue between 2015 and 2018. This technical capability coincided with Appodeal's $3.1M Series A funding round in 2015 [Crunchbase, 2015] and supported Appodeal's growth through the acquisition of Corona Labs in 2017.

2 Software Description

2.1 Architecture

DEXTER's architecture follows a modular loading pattern inspired by plugin systems [Parnas, 1972, Syeed et al., 2015], adapted to the constraints of the Android DEX format and the Dalvik/ART runtimes [Android Open Source Project, 2024a]. The high-level design consists of three layers: (1) a *pre-compilation stage* that packages each third-party SDK into an independent DEX file, (2) a *runtime loader* that injects these DEX files into the application's ClassLoader, and (3) a *listener system* that notifies client code when specific classes become available.

During the build phase, each third-party ad network SDK (distributed as a JAR or AAR file) is compiled into an independent DEX file using the Android `dx` tool [Android Open Source Project, 2015]. These files are given a `.dx` extension to distinguish them from the application's primary `classes.dex`, and are placed in the application's `assets/dex/` directory. Because asset files are bundled into the APK but not processed by the DEX merger, they do not contribute to the primary DEX file's method count. Shared dependencies such as Google's Protocol Buffers library are similarly packaged as a separate `protobuf.dx` file that is loaded before any ad network DEX that depends on it.

At runtime, when the application requires a particular ad network, it invokes DEXTER with the name of the corresponding `.dx` file and a list of class names to verify. DEXTER copies the DEX file from the APK's assets to a writable working directory, constructs a `DexClassLoader` targeting that file, and then uses reflection to extract the internal `dexElements` array from the new ClassLoader. These elements are merged into the application's own ClassLoader, making all classes from the ad network SDK available through the standard `Class.forName()` mechanism. A callback is fired

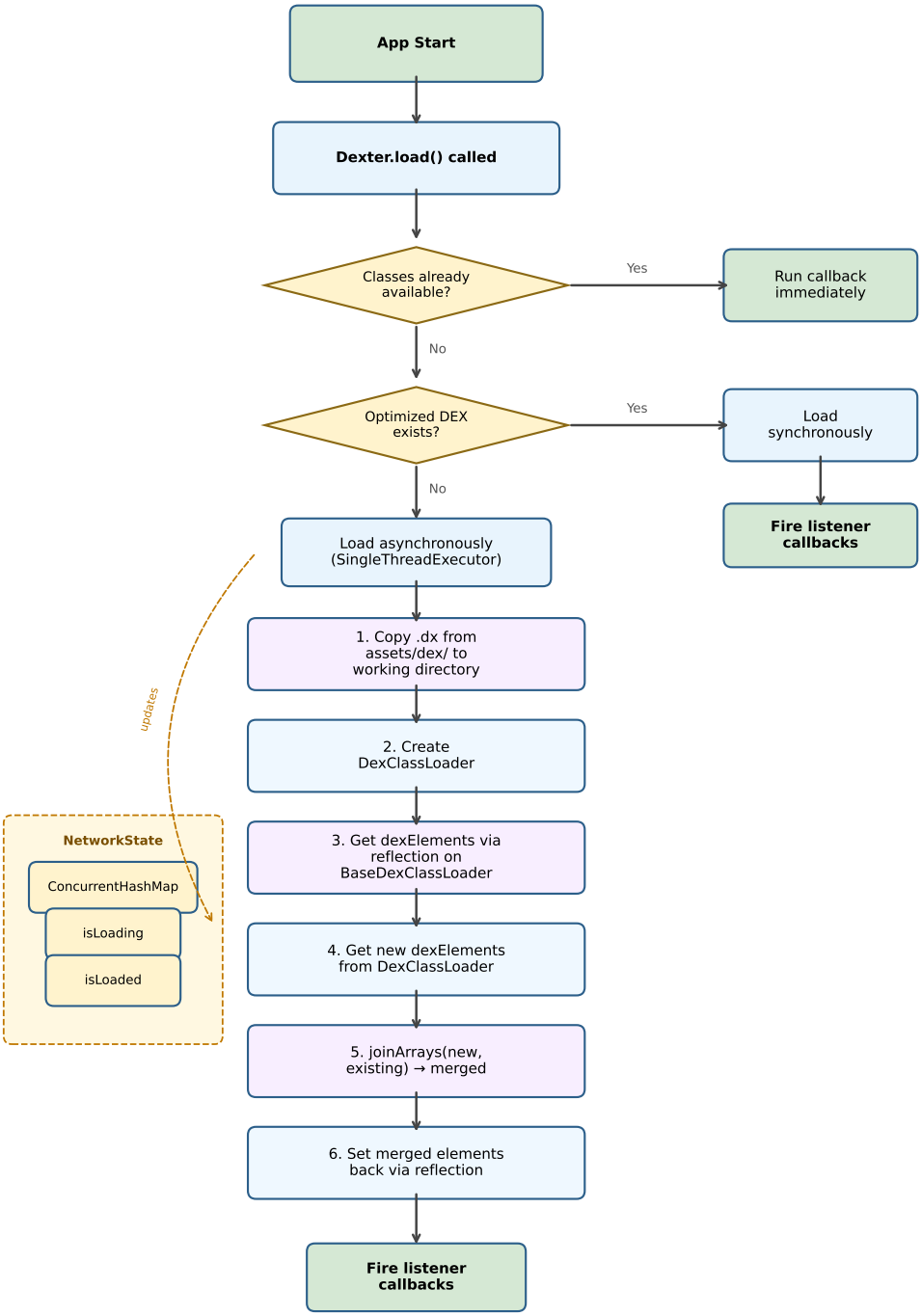

Figure 1: High-level architecture of the DEXTER dynamic multidex loading system. At build time, each third-party SDK is compiled into a standalone `.dx` file using the Android `dx` tool. At runtime, these files are loaded on demand and their DEX elements are injected into the application's ClassLoader via reflection.

once loading completes, signaling to the client code that it may safely instantiate classes from the newly loaded network.

2.2 Core Mechanism

The central technical contribution of DEXTER is a reflection-based DEX injection mechanism that manipulates the internal data structures of Android's `BaseDexClassLoader` [Android Developers, 2024b]. The key insight is that all class loading in Android ultimately resolves through an array of `dexElements` stored in a `DexPathList` object within the ClassLoader. By reflectively accessing and extending this array, we can make classes from arbitrary DEX files loadable without modifying the application's build configuration.

Listing 1 presents the core implementation. The `loadFromAssets()` method orchestrates the loading process: it copies the target `.dx` file from the APK's assets to a writable directory, creates a `DexClassLoader` for the file, extracts the `dexElements` from both the new and existing ClassLoaders via `getDexClassLoaderElements()`, merges them using `joinArrays()`, and writes the combined array back via `setDexClassLoaderElements()`.

```
public class Dexter {
    private static final String DEX_DIR = "dex";
    private static final String WORKING_DIR = "working";
    private static final String OPTIMIZED_DIR = "optimized";

    private final ConcurrentHashMap<String, NetworkState> states
        = new ConcurrentHashMap<>();
    private final ExecutorService executor
        = Executors.newSingleThreadExecutor();
    private final Handler mainHandler
        = new Handler(Looper.getMainLooper());

    /**
     * Load a .dx file from assets, inject its classes into
     * the app ClassLoader, then fire the callback.
     */
    public void load(final Context ctx, final String dexName,
                     final String[] classNames,
                     final Runnable callback) {
        NetworkState state = getOrCreateState(dexName);
        if (state.isLoaded) {
            mainHandler.post(callback);
            return;
        }
        state.addListener(callback);
        if (state.isLoading) return;
        state.isLoading = true;

        Runnable task = () -> {
            Thread.currentThread()
                  .setPriority(Thread.MIN_PRIORITY);
            loadFromAssets(ctx, dexName, classNames);
        };

        File optDir = getOptimizedDir(ctx);
        if (optDir.list() != null
                && optDir.list().length > 0) {
            // Cached odex exists: load synchronously
            task.run();
        } else {
            // First load: async to avoid blocking UI
            executor.execute(task);
        }
    }

```

```
    private void loadFromAssets(Context ctx,
            String dexName, String[] classNames) {
        try {
            File workDir = getWorkingDir(ctx);
            File optDir  = getOptimizedDir(ctx);
            File dexFile = copyAssetToFile(
                ctx, DEX_DIR + "/" + dexName, workDir);

            DexClassLoader loader = new DexClassLoader(
                dexFile.getAbsolutePath(),
                optDir.getAbsolutePath(),
                null,
                ctx.getClassLoader()
            );

            Object[] incoming =
                getDexClassLoaderElements(loader);
            Object[] existing =
                getDexClassLoaderElements(
                    ctx.getClassLoader());
            Object[] merged =
                joinArrays(incoming, existing); // incoming first: new classes take precedence

            setDexClassLoaderElements(
                ctx.getClassLoader(), merged);

            verifyClasses(classNames);
            NetworkState state = states.get(dexName);
            state.isLoaded = true;
            state.isLoading = false;
            fireListeners(state);
        } catch (Exception e) {
            Log.e("Dexter", "Failed: " + dexName, e);
        }
    }

    /**
     * Reflectively extract dexElements from a ClassLoader.
     * Path: BaseDexClassLoader -> pathList -> dexElements
     */
    private Object[] getDexClassLoaderElements(
            ClassLoader loader) throws Exception {
        Field pathListField = getField(
            loader.getClass(), "pathList");
        pathListField.setAccessible(true);
        Object pathList = pathListField.get(loader);

        Field dexElementsField = getField(
            pathList.getClass(), "dexElements");
        dexElementsField.setAccessible(true);
        return (Object[]) dexElementsField.get(pathList);
    }

    /**
     * Reflectively set dexElements on a ClassLoader.
     */
    private void setDexClassLoaderElements(
            ClassLoader loader, Object[] elements)
            throws Exception {
        Field pathListField = getField(
            loader.getClass(), "pathList");
        pathListField.setAccessible(true);
        Object pathList = pathListField.get(loader);

        Field dexElementsField = getField(
            pathList.getClass(), "dexElements");
        dexElementsField.setAccessible(true);
        dexElementsField.set(pathList, elements);
```

```
    }

    /**
     * Merge two dexElements arrays using Array reflection.
     */
    private Object[] joinArrays(Object[] a, Object[] b) {
        Object[] result = (Object[])
            java.lang.reflect.Array.newInstance(
                a.getClass().getComponentType(),
                a.length + b.length);
        System.arraycopy(a, 0, result, 0, a.length);
        System.arraycopy(b, 0, result, a.length, b.length);
        return result;
    }

    private Field getField(Class<?> clazz, String name) {
        while (clazz != null) {
            try {
                return clazz.getDeclaredField(name);
            } catch (NoSuchFieldException e) {
                clazz = clazz.getSuperclass();
            }
        }
        throw new RuntimeException(
            "Field not found: " + name);
    }
}
```

Listing 1: Core DEXTER implementation showing the reflection-based DEX injection mechanism. The `loadFromAssets` method copies a `.dx` file from assets, creates a temporary `DexClassLoader`, extracts its internal `dexElements` via reflection, and merges them into the application's ClassLoader.

The reflection chain traverses two levels of Android's internal ClassLoader hierarchy. First, the `pathList` field is accessed on the `BaseDexClassLoader` class (the superclass of both `PathClassLoader` and `DexClassLoader`). This field holds a `DexPathList` object, which in turn contains a `dexElements` array—an array of `DexPathList.Element` objects, each representing one loaded DEX file. The `joinArrays()` method creates a new array of the same component type and copies both the existing and incoming elements, preserving the original search order so that the application's own classes take precedence.

Thread management is an important aspect of the design. DEXTER uses a `SingleThreadExecutor` to serialize DEX loading operations, preventing concurrent modification of the ClassLoader's internal state. Loading threads run at `Thread.MIN_PRIORITY` to minimize impact on the application's UI responsiveness. The first load of any DEX file is performed asynchronously because the Android runtime must compile the raw DEX bytecode into an optimized `.odex` file, a process that can take several hundred milliseconds. Subsequent loads are synchronous because the cached `.odex` file in the `optimized/` directory eliminates the compilation overhead.

The listener system uses a list of `Runnable` callbacks stored per network in the `NetworkState` object. When multiple components request the same ad network simultaneously, all callbacks are queued and fired together once loading completes. Callbacks are dispatched on the main thread via a `Handler` to ensure safe interaction with Android's UI framework.

2.3 Integration Pattern

Integrating DEXTER into an ad mediation SDK requires minimal code. The typical pattern is shown below:

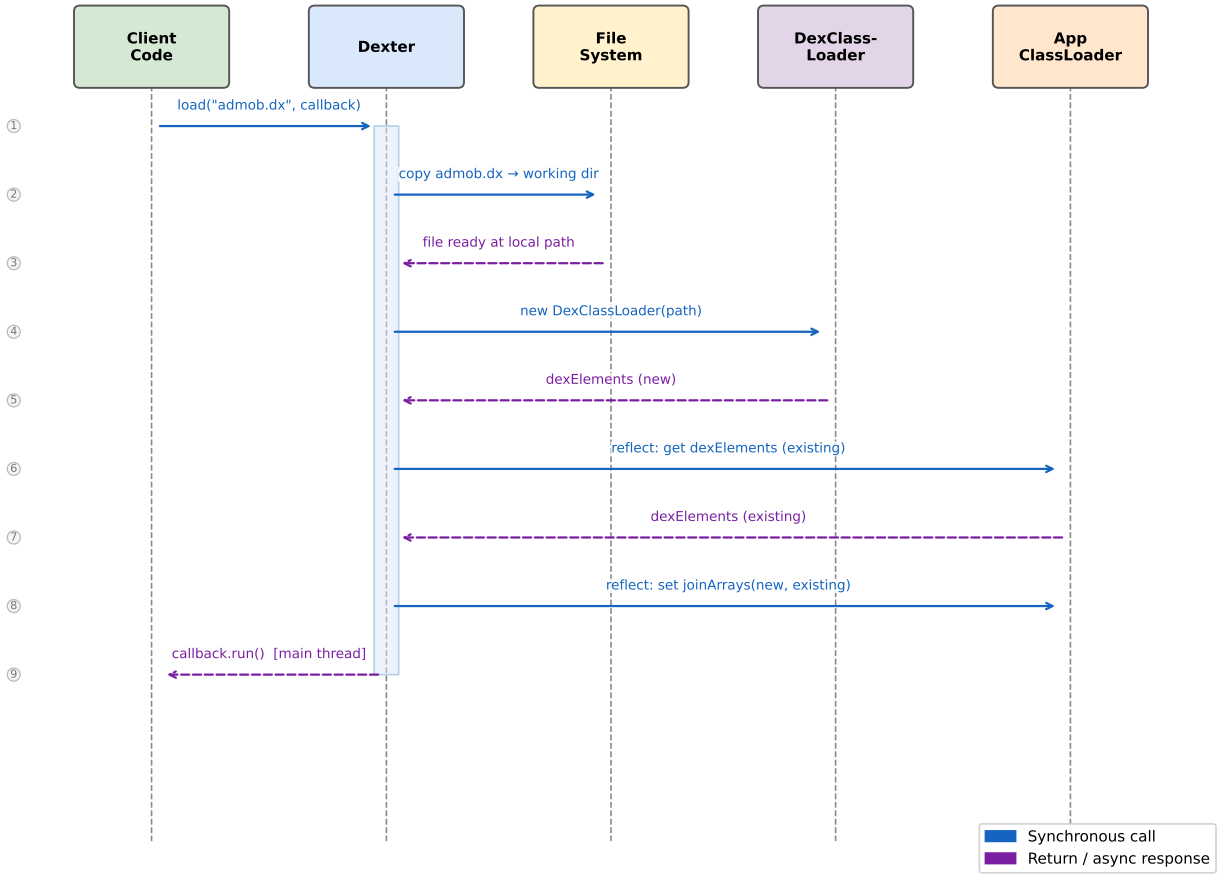

Figure 2: Sequence diagram of the DEX injection process. The reflection-based element extraction and merging occurs on a background thread; the client callback is dispatched to the main thread via a `Handler`.

```
// Load AdMob SDK at runtime
Dexter.load(context, "admob.dx",
    new String[]{"com.google.ads.AdView"},
    () -> {
        // AdMob classes now available
        AdView adView = new AdView(context);
        adView.setAdUnitId("ca-app-pub-xxx");
        adView.loadAd(new AdRequest.Builder().build());
    }
);
```

Listing 2: Typical integration pattern for loading an ad network SDK at runtime via DEXTER.

The three parameters beyond the `Context` are: (1) the name of the `.dx` file containing the ad network's compiled code, (2) an array of fully qualified class names used to verify that loading succeeded, and (3) a callback executed once the classes are available. This interface abstracts away all reflection and threading complexity, presenting a simple asynchronous module-loading API to SDK integrators.

For networks that depend on shared libraries such as Protocol Buffers, DEXTER supports dependency ordering: the shared library's DEX file is loaded first, and dependent network DEX files are loaded only after the shared dependency is confirmed available. This ensures that class resolution does not fail due to missing transitive dependencies.

3 Illustrative Examples

We present three scenarios that demonstrate DEXTER's capabilities in production ad mediation contexts.

Example 1: Single Network Loading. The simplest use case loads a single ad network on demand. When the mediation layer determines that AdMob should serve the next ad impression, it invokes DEXTER as shown in Listing 2. The callback fires within 50–200 ms on first load (including `.odex` compilation) and under 10 ms on subsequent loads (using the cached optimized DEX). This latency is well within acceptable bounds for ad pre-fetching, where network round-trip times dominate.

Example 2: Sequential Multi-Network Loading with Dependencies. When initializing the full mediation stack, multiple networks must be loaded with correct dependency ordering. Listing 3 demonstrates this pattern. The Protocol Buffers library, required by several ad networks, is loaded first. Its completion callback triggers the loading of dependent networks. Independent networks such as Chartboost can be loaded in parallel.

```
// Load shared dependency first
Dexter.load(context, "protobuf.dx",
    new String[]{"com.google.protobuf.Message"},
    () -> {
        // Now load networks that depend on protobuf
        Dexter.load(context, "admob.dx",
            new String[]{"com.google.ads.AdView"},
            () -> initializeAdMob());
        Dexter.load(context, "inmobi.dx",
            new String[]{"com.inmobi.ads.InMobiAdView"},
            () -> initializeInMobi());
    }
);

// Independent network: no shared dependency
```

```
Dexter.load(context, "chartboost.dx",
    new String[]{"com.chartboost.sdk.Chartboost"},
    () -> initializeChartboost());
```

Listing 3: Loading multiple ad networks with dependency ordering. The shared Protocol Buffers library is loaded first; dependent networks are loaded in its callback.

Example 3: Graceful Handling of Missing SDKs. Not all applications bundle all supported ad networks. If a publisher chooses to include only a subset of networks, the corresponding `.dx` files are simply omitted from the `assets/dex/` directory. DEXTER handles this gracefully: when the requested asset file does not exist, the loading operation fails silently (logging a warning), and the callback is never invoked. The mediation layer interprets a missing callback as an unavailable network and routes ad requests to the next candidate in its waterfall. This design allows a single SDK binary to support an extensible set of ad networks without requiring compile-time conditional logic.

4 Impact

DEXTER had a measurable impact on the mobile advertising ecosystem across four dimensions: technical capability, business outcomes, platform reach, and industry timeline.

Technical Capability. DEXTER enabled the integration of over 80 ad networks within a single mediation SDK [Appodeal Inc., 2015]. Prior to DEXTER, no ad mediation platform could offer this breadth of network coverage on Android for cross-platform game engines, because the combined method count of even 15–20 ad network SDKs exceeded the 65K DEX limit. By isolating each network into a separate DEX module, DEXTER eliminated the method count as a scaling constraint entirely. The modular architecture also provided operational benefits: individual ad network SDKs could be updated independently by replacing a single `.dx` file, without recompiling the entire mediation SDK.

Business Outcomes. The Unity plugin built on DEXTER became Appodeal's flagship product, accounting for approximately 80% of company revenue between 2015 and 2018. The Unity game development community represented the largest addressable market for ad-monetized mobile games, and DEXTER was the enabling technology that made Appodeal's SDK functional in that ecosystem. Appodeal raised a $3.1M Series A funding round in 2015 [Crunchbase, 2015], during which the ability to serve the cross-platform game market—which competitors could not address due to the multidex limitation—was a key differentiator.

Platform Reach. Beyond Unity, the DEXTER architecture supported SDK plugins for over 20 cross-platform development frameworks, including React Native, Xamarin, Apache Cordova, Unreal Engine, Defold, GameMaker, BuildBox, Qt, Corona SDK, Adobe AIR, LibGDX, Gideros, and MonkeyX [Biorn-Hansen et al., 2019]. Each platform plugin leveraged the same underlying DEXTER mechanism for DEX loading, providing a consistent integration experience regardless of the host engine. This breadth of platform support was unprecedented in the ad mediation market and attracted thousands of mobile game developers to the Appodeal platform.

In 2017, the value of this cross-platform expertise was recognized when Appodeal acquired Corona Labs, the company behind the Corona SDK game engine. The author contributed to the

Table 1: Summary of DEXTER's quantitative impact across key metrics.

Metric	Value
Ad networks supported	80+
Cross-platform plugins	20+
Revenue contribution (Unity plugin)	∼80% of Appodeal revenue
Series A funding enabled	$3.1M (2015)
Years ahead of native multidex	2–3 years
Active deployment period	2015–2018

technical integration, bringing the dynamic DEX loading architecture to Corona's native plugin system.

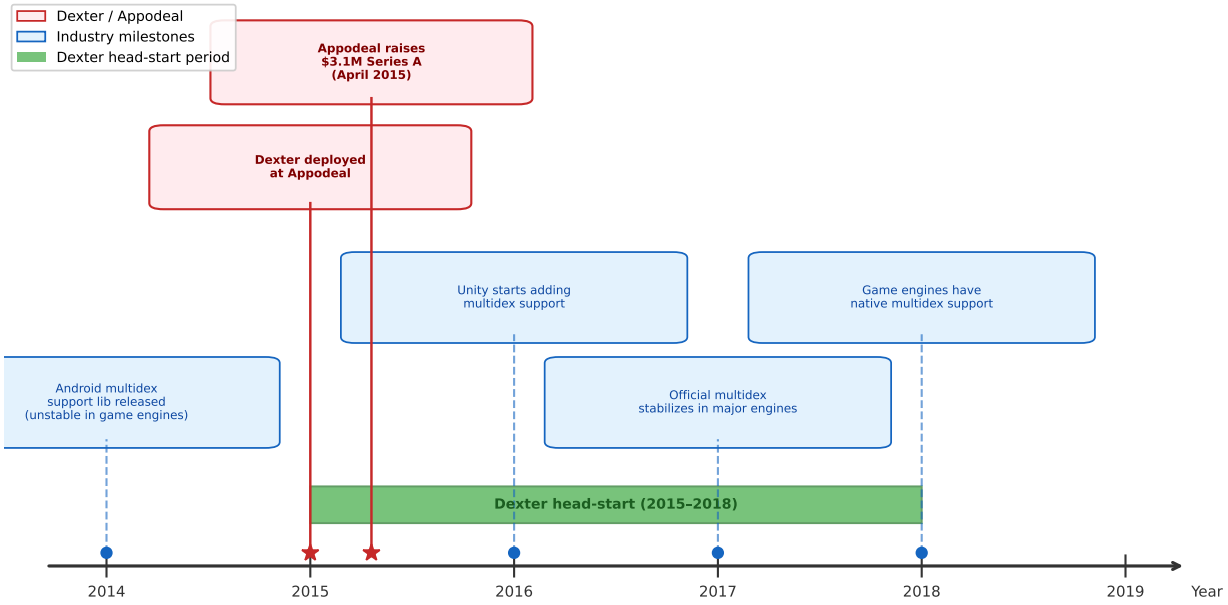

Figure 3: Timeline showing DEXTER's deployment relative to the maturation of official multidex support across game engines. DEXTER provided a 2–3 year head start over platform-native solutions.

Industry Timeline. DEXTER predated mature multidex support in cross-platform game engines by two to three years (Figure 3). While Google's official multidex support library existed from 2014 [Android Developers, 2024a], it required modifications to the application's `Application` class and build configuration that were inaccessible in most game engine environments. Unity did not ship reliable built-in multidex support until 2018 [Unity Technologies, 2018]. Corona SDK and Cordova followed similar timelines [Corona Labs, 2017, Apache Software Foundation, 2017]. During this 2015–2018 window, DEXTER was effectively the only viable solution for integrating large ad mediation SDKs into cross-platform mobile games on Android.

Table 1 summarizes the quantitative impact of DEXTER across its active deployment period.

The combination of technical capability, business leverage, and timing gave Appodeal a sustained competitive advantage in the mobile ad mediation market during a critical growth period for the mobile gaming industry.

5 Performance Characteristics

DEXTER's loading behavior exhibits two distinct performance profiles determined by the `mustLoadAsync()` heuristic, which checks whether an optimized `.odex` file already exists in the application's private directory.

Cold Start (First Load). On the first invocation for a given ad network, the raw DEX bytecode must be compiled by the Android runtime into an optimized `.odex` file. This compilation is performed asynchronously on a background thread at `Thread.MIN_PRIORITY` to avoid blocking the UI. The process involves: (1) copying the `.dx` asset to a writable working directory, (2) instantiating a `DexClassLoader` which triggers `.odex` compilation, and (3) merging the resulting `dexElements` into the application ClassLoader. On devices tested during deployment (Android API 16–21, 2015–2016 hardware), cold-start loading required approximately 200–400 ms depending on the network SDK size and device performance tier.

Warm Start (Subsequent Loads). On subsequent application launches, the cached `.odex` file is present and the load path switches to synchronous execution. The `DexClassLoader` reads the pre-compiled binary directly without recompilation, reducing load time to under 10 ms. DEXTER also validates DEX file integrity by comparing the asset byte length against the cached file size; a mismatch invalidates the cache and forces a cold-start recompile, ensuring that SDK updates are always applied correctly.

Table 2 summarizes the observed loading characteristics.

Table 2: Observed DEXTER loading characteristics (Android API 16–21, 2015–2016 devices).

Load Type	**Latency**	**Thread**
Cold start (first launch)	200–400 ms	Background (MIN_PRIORITY)
Warm start (cached .odex)	<10 ms	Synchronous (caller thread)

Concurrency. All DEX loading operations are serialized through a `SingleThreadExecutor`, preventing concurrent modification of the ClassLoader's internal `dexElements` array. The listener notification path (dispatching callbacks once a network is loaded) is protected by synchronized access on the `listeners` map. Correctness under concurrent load was validated with parameterized unit tests using Robolectric, exercising 100–300 simultaneous listener registrations across 2–5 concurrent reader threads.

6 Threats to Validity

Internal Validity. The performance figures reported in Section 5 are derived from production deployment observations rather than controlled laboratory benchmarks. Device heterogeneity, background processes, and storage I/O variance were not controlled. The reported latency ranges should be treated as indicative rather than precise measurements.

External Validity. DEXTER's reflection-based mechanism targets the internal `pathList` and `dexElements` fields of `BaseDexClassLoader`. These are private implementation details of the Android runtime and not part of the public API contract. Android OS updates could in principle rename or restructure these fields, breaking the reflection chain. During the 2015–2018 deployment period, this did not occur across Android API levels 16–26, but future Android versions (particularly those enforcing stricter reflection restrictions under Project Mainline) may require adaptation.

Reproducibility. The production DEXTER implementation was developed as part of Appodeal's proprietary SDK and is not publicly available in its original form. The code listings in this paper present a functionally equivalent reference implementation. The core reflection mechanism and loading protocol are described in sufficient detail for independent reimplementation.

7 Conclusions

DEXTER solved an industry-wide Android limitation before official platform solutions existed. By pre-compiling each third-party SDK into an independent DEX file and injecting it into the application's ClassLoader at runtime through reflection, DEXTER decoupled the number of integrated ad networks from the DEX method count constraint. This modular architecture enabled unprecedented ad network coverage—over 80 networks in a single SDK—and made that coverage available across more than 20 cross-platform development frameworks.

The experience of developing and deploying DEXTER yields several lessons for the broader software engineering community. First, runtime reflection is a powerful mechanism for bypassing platform constraints, but it is inherently fragile: each Android version update carried the risk of internal API changes that could break the reflection chain from `BaseDexClassLoader` to `dexElements` [Rasthofer et al., 2016, Qu et al., 2017]. Careful version-specific testing and fallback paths are essential for any system that relies on reflective access to private fields. Second, pre-compilation of third-party dependencies into separate DEX modules proved to be an effective isolation strategy, providing both technical benefits (independent method count budgets) and operational benefits (independent update cycles) [Falsina et al., 2015].

The pattern of dynamic module loading at the bytecode level remains relevant beyond the specific context of Android multidex. Any mobile SDK that must aggregate a large and growing number of third-party dependencies faces analogous scalability constraints, whether from method count limits, binary size budgets, or symbol conflicts [Sun and Tan, 2014, Li et al., 2017]. The architectural approach demonstrated by DEXTER—treating each dependency as a runtime-loadable module with a well-defined activation interface—offers a reusable template for building extensible SDK platforms under tight platform constraints.

Data Availability

The core DEXTER implementation described in this paper was developed as part of Appodeal's proprietary SDK. The reference implementation presented in Listing 1 is a functionally equivalent reconstruction of the production code. The reflection-based DEX injection mechanism, the `mustLoadAsync` heuristic, the `ConcurrentHashMap`-backed listener system, and the DEX integrity check (asset-length comparison) are all faithfully represented. Researchers wishing to reproduce or extend this work may implement the mechanism directly from the code listings provided.

Author Contributions

N.K. conceived, designed, implemented, and deployed the Dexter system at Appodeal Inc. between 2015 and 2018. N.K. wrote the manuscript, prepared the figures, and conducted the literature review.

Conflicts of Interest

The author was employed by Appodeal Inc. during the development and deployment of Dexter (2014–2018). The author declares no current conflicts of interest related to this work.

References

Android Developers. Enable multidex for apps with over 64k methods. https://developer.android.com/build/multidex, 2024a.

Android Developers. Dexclassloader. https://developer.android.com/reference/dalvik/system/DexClassLoader, 2024b.

Android Open Source Project. dx – dalvik cross-assembler. https://android.googlesource.com/platform/dalvik/+/refs/heads/master/dx/, 2015.

Android Open Source Project. Android runtime and dalvik. https://source.android.com/docs/core/runtime, 2024a.

Android Open Source Project. Dalvik executable format. https://source.android.com/docs/core/runtime/dex-format, 2024b.

Apache Software Foundation. Apache cordova platform guide: Android. https://cordova.apache.org/docs/en/latest/guide/platforms/android/, 2017.

Appodeal Inc. Appodeal sdk documentation. https://docs.appodeal.com, 2015.

Andreas Biorn-Hansen, Tor-Morten Gronli, and Gheorghita Ghinea. A survey and taxonomy of core concepts and research challenges in cross-platform mobile development. *ACM Computing Surveys*, 51(5), 2018.

Andreas Biorn-Hansen, Tor-Morten Gronli, Gheorghita Ghinea, and Sahel Alouneh. An empirical study of cross-platform mobile development in industry. *Wireless Communications and Mobile Computing*, 2019.

Dan Bornstein. Dalvik vm internals. In *Google I/O Developer Conference*, 2008.

Corona Labs. Corona labs joins appodeal. https://coronalabs.com, 2017.

Crunchbase. Appodeal – funding rounds. https://www.crunchbase.com/organization/appodeal, 2015.

Luca Falsina, Antonio Lioy, and Gianluca Ramunno. Grab'n run: Secure and practical dynamic code loading for android applications. In *Annual Computer Security Applications Conference (ACSAC)*, 2015.

Jiaping Gui, Stuart Mcilroy, Meiyappan Nagappan, and William G.J. Halfond. Truth in advertising: The hidden cost of mobile ads for software developers. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2015.

Jiaping Gui, Ding Li, Mian Wan, and William G.J. Halfond. Lightweight measurement and estimation of mobile ad energy consumption. In *International Workshop on Green and Sustainable Software (GREENS)*, 2016.

Henning Heitkötter, Sebastian Hanschke, and Tim A Majchrzak. Evaluating cross-platform development approaches for mobile applications. In *International Conference on Web Information Systems and Technologies (WEBIST)*. Springer, 2013.

Li Li, Tegawendé F Bissyandé, Jacques Klein, et al. An investigation into the use of common libraries in android apps. In *IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016.

Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 88:67–95, 2017.

Suman Nath. Madscope: Characterizing mobile in-app targeted ads. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2015.

David L Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. Dydroid: Measuring dynamic code loading and its security implications in android applications. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017.

Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *Network and Distributed System Security Symposium (NDSS)*, 2016.

Mengtao Sun and Gang Tan. Nativeguard: Protecting android applications from third-party native libraries. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2014.

M.M. Mahbubul Syeed, Imed Hammouda, and Tarja Systä. Pluggable systems as architectural pattern: An ecosystemic perspective. In *European Conference on Software Architecture (ECSA) Workshops*, 2015.

Unity Technologies. Building for android: Multidex support. https://docs.unity3d.com/Manual/android-build-process.html, 2018.