2019 IEEE European Symposium on Security and Privacy (EuroS&P)

Up-To-Crash: Evaluating Third-Party Library Updatability on Android

Jie Huang, Nataniel Borges, Sven Bugiel, Michael Backes
CISPA Helmholtz Center for Information Security
{jie.huang, nataniel.borges, bugiel, backes}@cispa.saarland

Abstract—**Buggy and flawed third-party libraries increase their host app's attack surface and put the users' privacy at risk. To avert this risk, libraries have to be kept updated to their newest versions by the app developers that integrate them into their projects. Recent researches revealed that the prevalence of outdated third-party libraries in Android apps is indeed a rampant problem, but also suggested that there is a great opportunity for drop-in replacements of outdated libraries, which would not even require cooperation by the app developers to update the libraries. However, all those conclusions are based on static app analysis, which can only provide an abstract view.**

In this work, we extend the updatability analysis to the runtime of apps. We implement a solution to update third-party libraries with drop-in replacements by their newer versions. To verify the feasibility of this developer-independent update mechanism, we dynamically test 3,000 real world apps for 3 popular libraries (78 library versions) for runtime failures stemming from incompatible library updates. To investigate the updatability of libraries in-depth, exploration enhanced dynamic testing is adopted to monitor the runtime behaviors of 15 apps before and after library updating. From our test, we find that the prior reported updatability rate is under real conditions overestimated by a factor of 1.57–2.06. Through root cause analysis, we find that the underlying problems prohibiting easy updates are intricate, such as deprecated functions, changed data structures, or entangled dependencies between different libraries and even the host app. We think our results not only put a more realistic light on the library updatability problem in Android, but also provide valuable insights for future solutions that provide automatic library updates or that try to support the app developers in better maintaining their external dependencies.

I. Introduction

Third-party dependencies are frequently imported into applications to quicken the app development process. Compared with writing functional code from scratch, such as HTTP communication, image loading, or advertising, an existing well-encapsulated third-party package is a preferable choice for app developers. However, such external dependencies are a double-edged sword. Since third-party libraries are developed by other organizations and the app developers know little to nothing about the libraries' internals, the attack surface of the host app is unavoidably increased if an included library contains vulnerabilities. Previous studies have highlighted this problem for Android [4], [8], [14], [34] and have shown that vulnerable third-party dependencies are actively used in apps, e.g., a surprising $\approx$70% [33] of vulnerable free apps owe their vulnerabilities to integrated libraries.

The most straightforward countermeasure against such vulnerabilities are updates: a third-party library vendor would release a fixed version and then the applications that include the vulnerable library version can be fixed by updating the library as soon as possible. Unfortunately, most of the library updates cannot be delivered to applications in such a smooth way. Recent studies of third-party library updates [22], [30] show that most of the developers do not deem library updates as a reason for app version increment. Developers tend to preserve the outdated library versions to avoid additional efforts for resolving incompatibility with the newer library versions. Investigation of vulnerable apps' lifetimes [16], [31] also reveals the lack of incentives for non-functional updates. Considering this situation, an automated updating mechanism could be a way out of this dilemma [22]. Purely based on the API compatibility between versions of the same library, it was estimated that with such automated library updates 85.6% of libraries have at least one higher version available for update and 48.2% could even be updated to their latest version without any additional host code adoption. The problem is that the updatability rate is derived from static app analysis results, which can only provide a glimpse into automated library updates from a theoretical and syntactic perspective. It ignored potential factors for version incompatibility that can come into one's mind immediately, such as obsolete APIs, intra-function changes, or secondary dependencies. So far, no ground truth exists about the existence and severity of those additional factors. To bridge this gap, we try to answer in this paper the open questions *"What is the actual library updatability?"*, *"Do the updated libraries exhibit incompatibilities that prevent an easy drop-in replacement of library versions?"* and *"What are the primary causes for those incompatibilities?"*.

To answer those questions, we opt in this paper for studying apps' runtime behavior before and after applying drop-in replacements of API-compatible library updates. The best approach to do so could be 1) an implementation of an automatic library updating solution and 2) behavioral profiling of apps' runtime for both the original app and the one with library updates deployed. Several existing works have dug into the problem of *patching* vulnerabilities in existing applications, such as Appsealer [37], PatchDroid [28], or Instaguard [19]. Unfortunately, none of them specifically focuses on library code. PatchMan [36] considered libraries, but only takes system libraries into account. Most importantly, however, the setting for a library updatability solution, which has to con-

DOI 10.1109/EuroSP.2019.00012

sider multiple update candidate versions, code changes beyond "simple" function-level changes, and potentially entangled dependencies (see Section VI-B), differs a lot from vulnerability patching solutions (e.g., a static rewriting solution cannot deal with entangled dependencies, or in-memory patching is limited to very local, small changes). Thus, none of the existing solutions is applicable as a suitable solution to the automated library updatability problem.

To extend the status quo and investigate in-depth the proposed drop-in replacement of API-compatible library versions, this paper presents a two-stage experiment. In this experiment, an automatic drop-in replacement library update framework based on classloader customization is put forward in the first stage, and then, in the second stage, dedicated, dynamic tests are carried out to evaluate the runtime behavioral differences between the original app and the one with an updated library.

To the best of our knowledge, this work is the first to investigate the semantic problems and consequences for Android library updatability in a real-world setting in contrast to the previously estimated numbers purely on syntactic updatability. Our study focuses on three popular, previously studied libraries (*OkHttp*, *Facebook SDK*, *Facebook Audience*). Our dynamic analysis results revealed that at runtime 4.08% (success rate 95.92%) of the tested updates experienced crashes after the drop-in library update. We discovered that multiple factors impede the automatic integration of a compatible library version. Through a source code study of crashed library versions, we discover incompatibilities beyond the public API, including deprecated public methods, changed data structures and library initializations that are only documented in the library changelogs, or entangled dependencies between the updated library and other libraries or the host app. Further analyzing the source code of 1,430 versions of 44 libraries showed that those discovered impeditive factors are prevalent in all kinds of other libraries and the claimed library updatability rate by prior works [22] should be adjusted. To provide a clear understanding of the library updatability, we re-calculate the updatability rate on a set of 332,432 apps after considering all those discovered factors. The comparison result shows that for *OkHttp* and *Facebook SDK* the picture is rather bleak, and their updatability rates sink 93.40%↘45.45% and 94.06%↘53.69% in the worst case, respectively, in comparison to previous estimates. Thus, our work confirms the *technical* feasibility of an automatic drop-in replacement for library updates, but our test results also clearly show the existence of impeditive factors that prevent a drop-in library update from working correctly *in practice*. We think that our results provide valuable insights for the design of projected library update solutions that are independent of the app developer (e.g., drop-in replacements at the market or on-device) as well as for solutions that want to support app developers in maintaining up-to-date dependencies (e.g., through an IDE extension).
To summarize our contributions:

1) API-compatibility based library update framework: To measure the realistic gap for drop-in library updates on Android, we first need a library update framework that follows the state-of-the-art proposal in prior work [22]. This work is first to present the design and implementation of a drop-in based Android library update framework. With this framework, a new library version can be opted into the original app at app launching time and be used as a replacement for the previous library version, which enables us to hunt library update-related runtime mal-functions further.

2) App runtime behavior profiling: Using our library updating approach, two kinds of dynamic tests are carried out on real-world apps to not only validate the feasibility of our updating solution but also study the actual feasibility of drop-in library updates and re-evaluate the results of static app analysis in existing work [22]. By profiling the runtime behaviors of apps before and after library updates, we detect the occurrence of malfunctions introduced by the library update despite the library versions being API-compatible.

3) In-depth study of the obstacles for functional drop-in replacements: By analyzing the malfunctioning cases, we discovered several factors brought by library evolution that prohibit the drop-in replacement of a target library to be functional. Based on those discovered factors, a follow-up study is conducted to evaluate the prevalence of those impeditive cases in other libraries. Our results show that those impeditive factors are important considerations for future solutions that target automatic library updates or that support app developers in their task of updating libraries.

Outline: In Section II, we give a brief introduction to Android's software update ecosystem and background on library updating. We motivate our work more explicitly in Section III. Section IV describes our two-stage experiment to practice and evaluate API-compatibility based drop-in library updates. The experiment findings together with a follow-up study of library source code are presented in Section V. In Section VI, we discuss our work and future prospects. We conclude our paper in Section VII.

II. Android Software Updating and Testing

A. Android Software Update Ecosystem

The official sources for updates for Android software can be differentiated into four classes: App developers, Android Open Source Project (AOSP) by Google, upstream Linux kernel, and system-on-chip (SoC) manufacturers. The updates from those sources can be delivered to end users and take effect in their corresponding software stack layer through different update routines as shown in Figure 1.

The Android platform is highly diversified and fragmented. The updates from the lower layers are distributed in an arduous and time-consuming way. All the updates from AOSP, Linux kernel, and SoC manufacturers should be delivered to device manufacturers first. After being integrated into the manufacturers' specific systems, some of those updates can be pushed to the end users by device manufacturers, and some of them should also go through a carrier technical acceptance test at the network operator side before being delivered to users. Existing work [32] has pointed out that device manufacturers

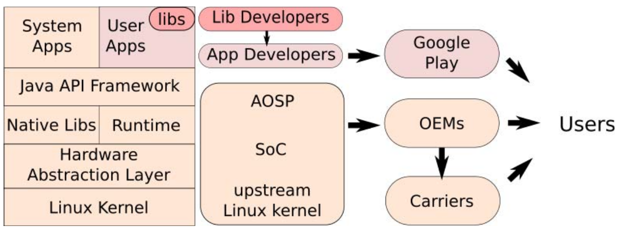

Fig. 1. Android Software Update Ecosystem

are the bottleneck in this update chain. Despite the founding of the Android Update Alliance [12], phone vendors generally lack the incentives to provide updates frequently, resulting in a long update latency or, even worse, no updates at all.

The update routine for user applications is, in contrast, pretty straightforward. The app developers submit new app versions to Google Play, then the target app on end devices will be reinstalled and replaced with the updated version [11]. The app updates can be delivered to user devices efficiently without any intermediate bottleneck. It is noteworthy that the updating of third-party libraries, which we are concerned with in this paper, follows a similar mode to the low layer components. New versions of third-party libraries are released by the library developers first, and after integrating the new versions into the application code by app developers, those libraries can be sent to end users together with upgraded apps through Google Play. Despite the ”new library versions available” warning provided by the built-in Lint plugin [6] of Android Studio—the most commonly used IDE for Android app development—the integration of the updated library is highly dependent on the app developer incentives, and currently, there is no official automatic mechanism to ease this process. Google Play rejects apps (updates) that include libraries with known security vulnerabilities to force the developers to update those libraries through their app security improvement program[1], but this mechanism only works for a small, limited set of libraries, e.g., Apache Cordova. A lot more vulnerable libraries are still exempted from this vetting process.

B. Software Patching Techniques

Apart from going through the standard update chain described earlier, an Android software can also be fixed by third-party patches and application autonomous hotfixes.

Third-Party Patching reduces the vulnerability window of software as much as possible. Since the patches or patching framework are released by neither software developers nor official sources, they are not bound to the standard release procedure and can be deployed to fix software more efficiently. *Patchdroid* [28] applies in-memory patching techniques to update both userspace native code and Dalvik bytecode at runtime. *Embroidery* [38] uses both static and dynamic rewriting techniques to patch vulnerabilities in the Android framework and kernel. To be resilient against Android fragmentation and ensure system functionality across devices, Embroidery rewrites binaries at code-line granularity. With *reference hijacking* [36] the underlying system libraries are patched by redirecting library references to security-enhanced alternatives. *InstaGuard* [19] takes advantage of debugging features to enforce rules that block the vulnerability exploitation and avoid injecting new code while patching. *KARMA* [20] establishes a multi-level adaptive patching model to filter malicious input to the kernel. *Appsealer* [37] alters an app's intermediate representation to mitigate component hijacking attacks through a patched app version. None of the above solutions focuses on patching third-party libraries inside user apps. *OSSPATCHER* [23] targets at third-party libraries, but only open-sourced C/C++ libraries are concerned. There are also more works [24], [26] that automatically generate patches from source code. However, they do not apply to libraries included in applications that are usually not open sourced. Most recent work [25] rewrites app code to provide a library updating and sharing solution which is distinguished from our incompatibility root cause investigation purpose.

Application Autonomous Hotfix is a technique for self-healing apps where fixes to the app code are applied at runtime by the application itself. Some hotfix frameworks [1], [2], [9] have been put forward to ease the distribution of *minor* patches. In those solutions, an official patch is first delivered to the app, and then the patch code is dynamically loaded into memory instead of outdated code. There is no need to reinstall the target app. With autonomous hotfixes, small fixes can be distributed to users swiftly without any user disturbance or central distribution point (e.g., Play). Unfortunately, those hotfix plugins are required to be integrated by app developers and the patches should be released by them as well, which highly depends on developer incentives and is not applicable to efficiently update libraries within already existing apps. However, the flexibility of those dynamic code integration techniques and plugin techniques [5] is quite inspiring and our third-party library updating solution is established based on them.

Patching vs. updating: Most of the patching solutions use techniques, such as static rewriting, in-memory function patching, or vulnerable path blocking, to mitigate vulnerabilities. However, the scenario for library updatability includes but is not limited to rolling out those pinpointed code fixes that are prevalent in patching scenarios. Library updates usually concern not only intra-function changes, but also inter-function changes, secondary dependency updates, and resource file changes, especially when upgrading across multiple versions. For this reason, a full library drop-in replacement update exceeds highly localized patching as described in the existing works. Prior work [36] also applies full library replacement for system libraries. However, the statically integrated third-party libraries in apps, in contrast, vary from app to app and in their versions, which prohibits a central, system-wide replacement of a third-party library. Furthermore, our paper studies the problem of library updatability and not specifically of ”patching security vulnerabilities” since for the mobile

[1]https://developer.android.com/google/play/asi

library ecosystem, prior work [22] reports that security and privacy patches are unfortunately commonly mingled with minor/major releases, and unfortunately very few library developers report security and privacy relevant changes in their logs. There is an expected high dark figure of ”silent patches.” Thus, patching security and privacy issues of libraries currently boils down to keeping library dependencies up-to-date. Our work tries to investigate the root causes of incompatibilities in this process for auto-updates.

C. Android Test Input Generation

To evaluate the app behavior and identify differences caused by a library update we rely on Android test input generation techniques, which can be broadly classified according to their underlying exploration engine into *random*, *model-based*, and *systematic*. Testing tools with *random* exploration engines create semi-random chains of events to explore the app’s behavior. This type of strategy is employed by Monkey [10], Android’s default test generator, which we used in our large-scale experiment, as well as DroidMate [18], the open source test generator we used in our runtime behavior profiling. While these approaches are unlikely to perform complex tasks, such as adequately logging in to an account, they have been shown to perform effective explorations [21]. To overcome the limitations of random testing in our analysis, we extend DroidMate with a plug-in that contains specific, non-random actions for relevant screens (e.g., login, registration) and allows us to reach more functionality.

Model based tools infer models from apps using static and/or dynamic analysis and use them to generate test cases. Tools in this category include *GUIRipper* [13], which dynamically traverses an app’s GUI and creates a state machine model, and *SmartDroid* [39], which uses static analysis to identify paths that should be dynamically explored. DroidMate [18] relies mainly on dynamic analysis. It extracts an app model during analysis and uses it for re-identification of UI elements, reducing re-exploration of known UI elements and guiding the test towards new ones.

Systematic testing tools employ different algorithms to exhaustively test apps or to generate tests which trigger specific behaviors. *Sapienz* [27], for example, combines search-based algorithms with random fuzzing to improve test coverage, while *IntelliDroid* [35] uses symbolic execution to create sequences of events to trigger specific behaviors. While these approaches may lead to more accurate and useful explorations in specific scenarios, their reliance on static information mitigates their applicability in scenarios where the app under test heavily relies upon external sources (e.g., web content), native code, or obfuscation (like reflection or encryption). In this category, and closely related to this work, is *Brahmastra* [17], which rewrites the app binary to jump-start specific third-party code. While this approach significantly increases the probability of reaching third-party code that is accessed deep within the application, it modifies the app behavior, which would affect the accuracy of our results.

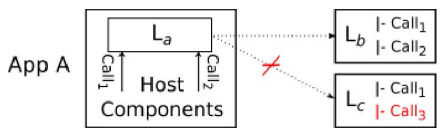

Fig. 2. A typical scenario for API-compatibility based updatability.

III. Motivation

Considering the alarming rate of outdated libraries and the inefficient third-party library updating chain explained in Section II-A, we put our focus in this paper on evaluating the runtime library updatability situation under an automatic third-party library updating framework, as well as tracing the root causes for potential side-effects brought by updating.

Typical scenario for API-compatibility based updatability: Existing studies have highlighted thrilling API compatibility across different library versions. Here, we describe a typical scenario based on Derr’s et al. work [22] and their *LibScout* tool (see Figure 2). App *A* contains library *L* in version *a* with invocations $Call_1$ and $Call_2$. If interfaces $Call_1$ and $Call_2$ still exist in the successor version L_b, but only partially exist in version L_c (e.g., parameters or types of a method have changed or a method was removed), *LibScout* reports library L_b as compatible with library L_a inside App *A* but not L_c.

Implementation of an automated library update framework: To investigate the updatability and catch potential incompatibilities beyond the theoretical results of prior works, we need a library update framework that follows the methodology proposed in the existing studies. In our paper, we follow the proposal of Derr’s et al. study [22]. Given the scenario above, an implementation of an automated library update framework should try to update library *L* from version *a* to version *b*. Another precondition of this API-compatibility based library updating solution is that the update should be a drop-in replacement and no host code adoption performed. The library upgrade could be done before or after app build without new host code adoption. In our work, we focus on a post-build upgrade, because compared with a pre-build upgrade, which is done through IDE plugins by app developers, a post-build upgrade is more flexible and can deliver the updated library version promptly, circumventing the upgrading bottleneck brought by the developer-dependence. Considering the complexity of the library updating scenario, which can include changes, such as inter-function code changes, secondary dependencies, or resource files, a naive static rewriting solution would cause an immediate crash/misbehavior (e.g., app failed to log into Facebook when the app’s signature was changed by static rewriting), which is then detrimental to exploring update incompatibilities. To try our best to eliminate unnecessary interferences and explore incompatibilities as reliably as we can while upholding conditions proposed in prior work [22], here we borrow the idea of opting in codes by classloader customization from existing frameworks [1], [5] and carefully design a dynamic library drop-in replacement framework (with

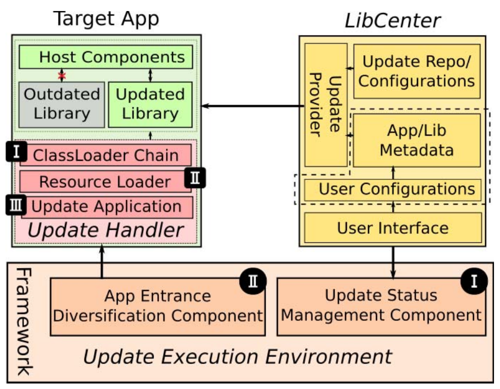

Fig. 3. Overview of Library Update Framework with three modules *Update Execution Environment*, *Update Handler*, and *LibCenter*.

secondary dependencies included) to support automatic library upgrading across both minor and major versions.

Automated library update testing: The API-compatibility based library updatability results presented in previous work [22] are based on static app analysis, which can only reflect a theoretical and syntactic situation. To understand the actual feasibility of an API-compatibility based library update, further runtime testing is necessary. The most established dynamic app testing is automated user interface (UI) testing, which performs a series of UI operations on the target app. By doing so, the app behavior can be profiled, and potential failures and dysfunctional behavior after library updating be discovered by comparing the runtime profiles of the original and the updated app. It is noteworthy that the feasibility of our library update framework can be confirmed in this context since behavioral correctness is a strict baseline for our testing.

IV. Two-stage Updating Experiment

The goal of our study is to evaluate if a simple drop-in replacement update is a viable option to solve the problem of outdated libraries on Android. In this section, we describe a two-stage experiment to test apps' runtime behaviors before and after a library update. In the first stage, we apply an automated library updating framework that we developed according to the proposal of prior work [22]. This framework allows replacing an outdated library inside an app with a newer version without additional host code adoption. During the second stage, two automated user interface (UI) tests are performed to evaluate the behavioral correctness of target apps after drop-in replacements of library updates. This approach allows us to report on the gap between the theoretical updatability rate in the literature and the actual runtime rate and its impeditive factors.

A. Stage-1: Automated Library Update Framework

To support automated library updating without host code adoption, this work implements a dynamic updating framework that takes advantage of the class domain isolation and dynamic code loading features of Android's classloader hierarchy. The outdated libraries are automatically updated at app load-time by loading the new library from a well-defined place by a customized classloader, and in this process no additional code adoption is required for the host app's code.

The framework is composed of three modules as shown in Figure 3: *Update Execution Environment*, *Update Handler*, and *LibCenter*. *Update Execution Environment* is established on a customized build of Android, which is extended with components to support library updates. *Update Handler* is a customized classloader chain together with auxiliary components for applying library updates at app load-time. This customized classloader chain isolates the loading of library code and host components at runtime. As a result, the library update can be opted-in as a replacement of the original library by solely altering the library class loading path. *LibCenter* is the centralized library management module. All the library updates and included library information for installed apps are maintained by it. It is also the user interface for update configuration. Through this app, the library update for a target app can be configured and delivered to the target app. Together those three modules enable automatic distribution and application of library updates without developer support and ease our testing by allowing us to flexibly roll-out library updates to the installed apps-under-test.

1) Update Execution Environment: To update a library of an app, the updated version should be available to the app. However, we have to abstain from modifying the app to avoid malfunctions due to induced bugs and also to adhere to the proposed methodology we are testing. Thus, the system should opt in the updated version before app initialization, which we accomplish through an update execution environment as an extension to vanilla Android. This environment consists of two key components (see also Figure 3): ❶ an update status manager to maintain a global update status of apps; and ❷ an app entrance diversification component to enable library updating for an app. The internals of update execution environment are illustrated in Figure 4.

❶ *Update Status Management Component* manages the update status for each app and allows us to control if an app runs with its original or an updated library version. Its *UpdateStatusService* is a dedicated system service that records each app's update status according to update events sent by *LibCenter* and unifies the update operations from system-side in the ❷ *App Entrance Diversification Component* and the update configuration from user-side in *LibCenter*. Client processes can reach the service over Binder IPC via a custom manager, *UpdateStatusManager* (***O3***), to set and get the update status for each app. *LibCenter* sets the status of target apps (***O1***) and ❷ *App Entrance Diversification Component* retrieves (***O2***) at app load-time the status for the loading app to determine which library update actions should be taken.

❷ *App Entrance Diversification Component* is the actual update deployment site and takes care of loading the updated library version into the application process. As can be seen in Figure 4, *App Entrance Diversification Component*

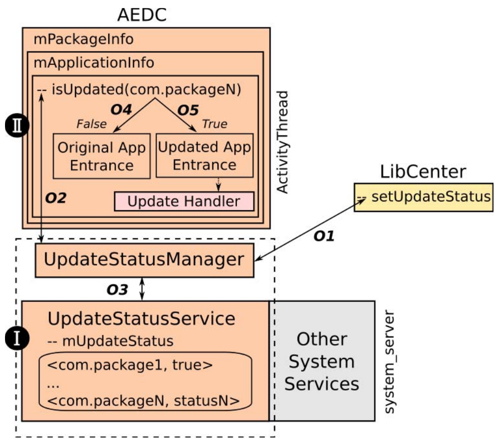

Fig. 4. Update Execution Environment: App Entrance Diversification Component is a customized *ActivityThread* to run the target app.

is implemented as a customized app launching process with an additional *Application* class interface. The *Application* class is the first class loaded in each app's process lifecycle. Initialization of the app and of the included library is usually executed inside the *Application* class to ensure they take effect at an early stage in the process' lifetime. Through this *Application* class, we added a new app entrance to an *Update Application* (Section IV-A2) to the original app launching process so as to control which library version should be loaded during app launching based on the updating status gained from the ❶ *UpdateStatusService*, which was set via *LibCenter*. With this modified launch process, the target app can switch between the original library version (***O4***) or the updated version (***O5***). In case of a library update, *Update Handler* continues the app launch process.

2) Update Handler: As shown in Figure 5, *Update Handler* is a bridge connecting host app and library update and is responsible for activating a library update for the app. To ensure any app on our modified Android can activate library updates, we integrated *Update Handler* into the Android framework as a static library that is automatically loaded into all app processes. *Update Handler* is composed of an I *Update ClassLoader Chain* for separating the target library code loading from the rest of the app code loading, an II *Update Resources Loader* to attach resources of the updated library to the original app, and an III *Update Application* to activate *Update ClassLoader Chain* and *Update Resources Loader* before the initialization of the app-under-test.

I *Update ClassLoader Chain* is a customized classloader chain specifically for dynamic library updating. Android inherited Java's parent-delegation mode in which a series of classloaders are chained together and each non-root classloader will delegate a class loading request to its parent classloader first before loading the requested class by itself. Only the root classloader will try to load the target class by itself directly. This parent-delegation mode separates the

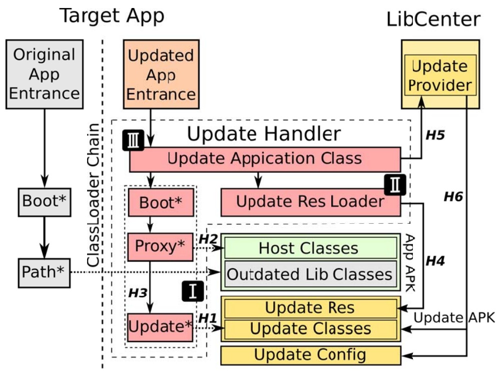

Fig. 5. Update Handler: Left part is the *ClassLoader Chain* for original Android while right part shows *Update ClassLoader Chain* and the involved components for library updating. (Suffix * indicates ClassLoader)

loaded code into different security domains according to their path, which prohibits a low priority classloader from exposing high priority code. For instance, *PathClassLoader* is in charge of loading installed application classes (class path in `/data/app/package.name`) and cannot load non-installed packages (e.g., class path in `/sdcard/`). Same class loaded by different classloaders is treated as different classes and cannot be cast to each other. In our design, classes from the updated library version should be loaded instead of the original outdated ones. However, the app package, including both the app code and libraries code, is a fixed bundle and the classes inside a user application are in general loaded by Android's default *PathClassLoader*. To suppress the loading of the originally contained library and opt-in the classes of the updated library, a new classloader chain is introduced in our design to isolate the loading of the updated library from the host application. Different with existing classloader customization based patching solutions [1], [7] which replace all outdated classes to updated ones directly, our solution constructs an isolated container for the interaction between updated library and its updated dependencies. Thus, both of the original and updated secondary dependencies are preserved in this design while updating the target library (first dependency) so as to provide better updating compatibility for cases where host codes involve invocations to secondary dependencies. Figure 5 shows how the two classloaders are customized for this new classloader chain.

UpdateClassLoader is an extension of *BaseDexClassLoader*, which is capable of loading dex files from a designated path. It is responsible for loading updated libraries without additional app code merging (***H1*** in Figure 5). This update-specific classloader is independent of the update, i.e., as soon as a newer library version becomes available, that version can be integrated with the host app by simply replacing the library file for updates and without touching the app package itself. However, objects created by different classloaders are not available to each other, which could complicate the interaction between the library and host application. To alleviate this problem, *UpdateClassLoader* has to be a node in the system classloader chain. It is linked as a child to *ProxyClassLoader*, the newly created classloader for application code.

ProxyClassLoader is an extension of *PathClassLoader*, which can only load installed applications files. Apart from loading the updated library's classes, the original host application should also be loaded. The app code is simply loaded by the default *PathClassLoader*. However, the original library code is intertwined with the host components inside the original app package (i.e., dex file). The original library code will also be loaded automatically by *PathClassLoader* when being invoked by the host components. To create a clear boundary between the host components and the library code that should be replaced with the updated version, *ProxyClassLoader* is constructed to delegate the loading of all updated library classes to *UpdateClassLoader*. To minimize the impact of this modification, *ProxyClassLoader* is initialized on the basis of the original *PathClassLoader*. Everything of *PathClassLoader* is preserved ***(H2)*** except for an additional class name filter when loading classes. When the class to load is from the target library, the name filter in *ProxyClassLoader* will distinguish the library package prefix in the class name and the loading request for this class is delegated to *UpdateClassLoader (**H3**)*, which will finish the class loading ***(H1)***. This way, the original, tightly integrated library will be replaced at app loading time with the newer library version.

II *Update Resources Loader* integrates the resources of the updated library ***(H4)*** into the app. Though not all libraries require additional resources, still a large fraction does in order to enhance their functionality, e.g., *Facebook SDK* requires resources to customize the login button. Since Android resources are labeled with a 32-bit ID, there could be ID conflicts between the original app resources and resources of the drop-in library. Our solution is to compile the library update within a wrapper application (*com.wrapper*) as a shared library, so the generated resource IDs will not be constants and can be reassigned to a separate range at runtime. To enable the usage of resources inside the added library update, its resources should be attached to the app space through *addAssetPathAsSharedLibrary* interface. Since the assigned IDs for the new resources might differ from the resource IDs used with the library code, we rewrite all of the individual library *R* classes with values in the merged resource file from wrapper package. After that, the new resources are available to both the library code and host application and no ID collision can happen between the original and the new library resources.

III *Update Application* is a customized *Application* class. The main idea is to ensure the updated library is activated before any host application code takes control. Considering that some library initialization is by default done in app's *Application* class, the activation of the new library should be handled before that. The most convenient and least intrusive solution is to hook the application initialization process by replacing the original app *Application* class with *Update Application* class that is described in Section IV-A1. After the replacement, the system will treat it just as the original *Application* class and finish the application initialization process. In this initialization process, a request for library updates will be sent to *LibCenter* from the target app's process space ***(H5)***. *LibCenter* will return an authorized URI that can be used to copy the library package and configuration files to the target app's storage ***(H6)***. Furthermore, the creation and initialization of both I *Update ClassLoader Chain* and II *Update Resources Loader* are also accomplished here based on the files retrieved from *LibCenter*. Last but not least, the newly generated classloader chain is enabled and the application can be launched as usual. To minimize system modifications in integrating the new classloader chain, we follow the classloader hooking approach used in former works [1], [5].

3) LibCenter: LibCenter acts as a centralized library repository from which library updates are retrieved. All precompiled library packages and metadata are stored here. Using *LibScout*[2] as part of *LibCenter*, we collect all installed apps' metadata including information about used libraries and library compatibility information based on the library API calls from the host apps. *LibCenter* uses that information together with user preferences set via *LibCenter*'s UI to create linking information about which API-compatible library update can be exposed to the *Update Handler* in target apps through an *Update Provider* (see Figure 3).

Precompiled Updates are a set of wrapper applications containing different versions of different libraries. Once an update is activated, its corresponding wrapper application will be copied to the app's process space so the updated library version inside the wrapper is available to the target app (see description earlier). The generation of wrappers for each library version is automated using *Gradle* with a template app. By altering the dependency library information in the *build.gradle* file of the template app, *Gradle* can synchronize the specific library version from its central repository and build the final wrapper application for this library version. There are two advantages in wrapping the updated library in an application with *Gradle*. First, considering that those target libraries also need their own dependencies, e.g., *OkHttp* depends on *okio*, we automatically bundle the target library together with its dependencies to avoid conflicts between the newly added library and the original library dependencies. Second, by wrapping the library bundle into an apk file, the resource file *R.java* can be generated and automatically arranged with *aapt*[3], which is necessary when invoking library calls that need resources. Here we compile libraries as shared libraries to avoid resource ID conflicts as described earlier.

Update Configurations are a set of files that describe the generated wrapper packages. As mentioned in Section IV-A2, information such as library class prefix and resource classes are necessary for correctly loading library code and resources. *Update Configurations* carry all the requested information of

[2]https://github.com/reddr/LibScout

[3]https://android.googlesource.com/platform/frameworks/base/+/master/tools/aapt

a library update and are sent to the target app together with the library package.

User Interface allows personalized settings for library updates, e.g., the target app, target library, and update version and is used by us to set up our test scenarios.

Update Linking Rules are created to dispatch a proper library update to a target app. They depend on both the *LibScout* generated library API compatibility metadata and user preferences (e.g., targeted library version). Library API compatibility metadata records the relationship between the host application and target libraries gained from offline library detection. For example, in the scenario described in Figure 2, a profile for the relation between app A and library L version a will be created in a form of quintuple *[A, A's version, L, a, [b]]*, where *[b]* is the list of API-compatible library versions. User preference designates the target app and library as well as the target library version. Combining the quintuple and user preference for an app, *LibCenter* can link a specific library update to the target app. This linking information will be used by *Update Provider* for exposing the correct wrapper application to the target app.

Update Provider is simply a *FileProvider* to share files, here wrapper applications, between target apps and *LibCenter*. It uses *Intent*s containing the URI for the corresponding wrapper application in response to requests by *Update Handler* to allow *Update Handler* to retrieve the library update from *LibCenter*.

B. Stage-2: Automated User Interface Tests

In the second stage of this experiment, we choose top ranking libraries as our case studies for library updates and run multiple dynamic tests on real-world apps from Google Play that include those libraries in order to have a close look at apps' runtime behaviors after API-compatible library updates. To ensure the comprehensiveness of this experiment, firstly, we carried out a large-scale dynamic test to provide a macro-view of not only the feasibility of our update framework but also of immediate malfunctions, like crashes, in target apps brought by those drop-in library replacements. Second, we execute a more intensive test to explore more app functionality so as to trigger more hidden malfunctions introduced by the updates, e.g., changed side-effects of library methods, although a full anomaly detection is beyond the scope of this paper.

1) Target Libraries & Apps: Different libraries have different integration approaches with their host apps. We carefully select three libraries, with 78 library versions in total, from different library categories [3] as target libraries: *OkHttp* from *Development Tools*, *Facebook SDK* from *Social SDKs* and *Facebook Audience* from *Ad Networks*. Those three libraries are the most popular libraries from reputable companies which are well-maintained and include secondary dependencies. Instead of targeting more libraries, the experiment setting here is more to utilize limited dynamic testing in highlighting a lower bound on the existence of incompatibilities when considering various library versions. To compile a list of apps that contain those libraries, we run *LibScout* on an app repository containing 332,432 free apps crawled from Google Play with 128

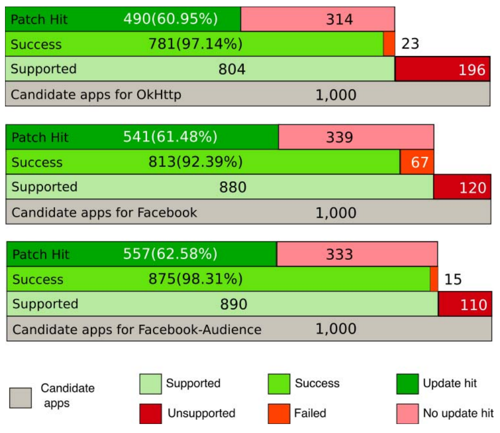

Fig. 6. Monkey Evaluation Results for Apps including *OkHttp*, *Facebook SDK*, *Facebook Audience*

library profiles for three libraries from the *LibScout* project. We found 379,429 library-app pairs. *LibScout* can not only provide a list of apps that contain a target library but also the detailed API usage of the library. To make the evaluation more comprehensive, we extend *LibScout* with a ranking module to cluster apps into different sets based on the library APIs invoked in the host app components. For dynamic testing, we select 3,000 apps, 10 apps from each of the top 100 frequently used API sets for each target library. There are 78 library versions (25 from *OkHttp*, 33 from *Facebook SDK*, 20 from *Facebook Audience*) in our final data set.

2) Monkey Test: Our update execution environment is deployed on Android v7.0. We test our framework on two Pixel C devices that are flashed with our customized system. In this large-scale evaluation, we try to update each library to the latest, API-compatible version. To measure the hit rate of (updated) libraries during testing, we used soot[4] to inject log statements into the frequently invoked interfaces of each library. To better scale the dynamic testing, we chose the open source monkey-troop tool[5] to install test apps and execute them using Google's official application exerciser, *Monkey* [10], on both devices in parallel and automatically. In each monkey run, 500 random events will be explored. During this process, the execution log is recorded for measuring the library hit rate.

Ground truth: Considering that some apps could be malfunctioning for reasons like failed download, obsolete APIs for our test platform, buggy design, etc., we first run monkey-troop on the original apps. Only if the first run executes successfully without errors, those apps remain in the test app set and are considered for library updates.

Test results: UI exploring results for the 3 libraries are shown in Figure 6. The uppermost graph describes the update result for *OkHttp* library, where 781 of 804 supported tested apps passed *Monkey* without a crash, giving a success rate

[4]https://github.com/Sable/soot/wiki/Tutorials
[5]https://github.com/Project-ARTist/monkey-troop

97.14%. Among all those apps supported, 60.95% of them hit the updated library. For *Facebook SDK* library, 813 of 880 supported target apps did not crash during monkey exploration, resulting in a 92.39% success rate, and 61.48% of those supported cases hit the library. The result for *Facebook Audience* is quite similar with a 98.31% success rate for the 890 supported apps and a 62.58% hit rate. From all those results, we can see a success rate of 95.92% in total, which is close to ideal. However, the overall hit rate of 61.69% is not very inspiring. More than 38 percent of apps passed *Monkey* without hitting the library. The reason could be, for instance, that the library is integrated at a hidden position, which cannot be hit easily (e.g., the target library is only invoked after purchasing a product successfully), or the *Monkey* failed to explore the specific path (e.g., clicking at a specific position on screen to jump to another page). Those are the common limitations of large-scale dynamic testing with random exploration by *Monkey* and have already been noticed in various existing works. To complement our results, a more in-depth but also time-intensive testing based on DroidMate is conducted to evaluate the internal misbehaviors of the host app after library updating.

3) DroidMate Test: In our second test, we focus on measuring the impact of updating a library onto the apps' behavior. Changes in behavior between the original app and the one with library update might indicate more intricate errors than crashes (e.g., changes in side-effects of library methods). For this experiment, we selected a random subset of 15 apps—5 for each library—from the monkey test and performed a comparative analysis of its behavior before and after updating. We consider as app functionality its source code blocks—including libraries—executed at runtime and as behavioral change a variation of the blocks reached before and after updating (for instance, exception handling routines would cause a deviation of the behavior).

To reach deeper functionality and obtain a more accurate impact measurement, we developed a plug-in for the open source DroidMate [18] platform[6] to bypass restrictions of the monkey analysis, such as lack of login/registration information. We performed this second test using four Pixel C devices and four emulators, also configured as Pixel C. Both sets of devices used the same customized version of Android v7.0 from the monkey-based testing. To ensure the accuracy of the results, each app was entirely tested either on a Pixel C device or on an emulator. We instrumented each test app, including libraries, with ARTist[7] [15] and obtained a list of all possible blocks, which we use as ground truth. This list may be an over-approximation of the actual possible behavior, as some blocks may be unreachable due to app's usage or to our test configuration. During the test execution, we logged all reached blocks—except for those belonging to the targeted library which by design would differ between tests—and monitored the log for a *library reached* tag to ensure that the target library was hit. We discarded runs that did not reach the library.

Test design: For each app, we executed the original app and the updated one until we obtained 10 executions of each that have reached the library. We opted for 10 runs to mitigate the variability caused by the random exploration as well as by non-deterministic content such as advertisements, while also achieving a reasonable time trade-off for testing duration. Each execution consists of 500 events on valid UI elements, such as clicks, long clicks, and swipes, where valid UI elements are those that are visible, enabled, and can be clicked on the screen. Our DroidMate plug-in executed predefined actions on login and registration screens—entering user name, entering password, and clicking the login button—and explored the remaining screens with DroidMate standard biased-random approach, which prioritizes UI elements that have not been previously interacted with in order to increase chances for discovering new path and code coverage. For evaluation, we grouped the executions output in two clusters: original (O) and updated (U). For each cluster we computed the set of reached blocks (B_O and B_U), that is, the set of blocks that were reached by at least one run. To measure the behavior change between the original and updated app, we compared the intersection between these clusters ($B_I = B_O \cap B_U$) against the set of blocks reached in the original runs ($|B_O - B_I|$). If this difference is greater than $3\times$*standard deviation* of the average coverage among the elements in B_O ($|B_O - B_I| > 3\times\sigma(O)$)—which covers 99.7% of the values assuming the coverage variation follows a normal distribution—we considered that there was a behavioral change.

Test results: The test results are shown in Table I. For each app we evaluated the overall block coverage, i.e., the percentage of blocks from both the app and its libraries, which could be reached during testing, as well app code coverage, i.e., considering only blocks belonging to the same package as the app. We use this coverage as an indication of the test depth and relevance.

Our DroidMate plug-in achieved, on average, 37% overall block coverage on the original apps and 36% on the updated ones, with a minimum of 12.66% and a maximum of 60.86%. Considering only app block coverage, it achieved 35% coverage on both app versions, with a minimum of 11.76% and a maximum of 60.50%, which falls within the range of expected coverage for state-of-the-art test generators [21].

Using the *3 standard deviation* tolerance as a metric, only the *SnapOdo* app, which uses the *Facebook SDK*, displayed a behavioral change. A manual inspection of the exploration results of the updated version showed that this app was no longer able to log in to Facebook, which in turn restricted the number of reachable blocks for exploration. *LOOM CLUB* also showed a significant coverage difference (5%), however, due to the highly non-deterministic nature of the app, this same difference was observed within the runs of its original version and thus was characterized as exploration noise.

[6]https://github.com/uds-se/droidmate
[7]https://github.com/Project-ARTist

TABLE I

RESULTS OF IN-DEPTH ANALYSIS WITH DROIDMATE PLUG-IN. COMPARISON BETWEEN CODE COVERED FOR ORIGINAL AND UPDATED APP.

App			**Original**			**Updated**		
Name	**Version**	**Library**	**App**	**Overall**	**St.Dev.**	**App**	**Overall**	**Change**
Shalom Shalom Radio	2.0	OkHttp	18.59%	17.81%	3.17%	19.36%	16.76%	No
Maurin Hyundai	3.0.4	OkHttp	13.33%	48.93%	1.06%	13.33%	48.93%	No
Blur Effect Keyboard	1.185.1.102	OkHttp	31.36%	37.36%	2.87%	33.58%	35.85%	No
UK Online FM	1.0	OkHttp	48.35%	60.86%	0.71%	48.35%	60.46%	No
Sanimedius Apotheke	2.1.10	OkHttp	56.58%	37.94%	3.07%	57.89%	37.90%	No
LOOM CLUB	4.785	Facebook	23.62%	29.03%	2.26%	20.25%	23.98%	No
Farmacia Charo Ferrá	0.01	Facebook	11.76%	36.80%	4.16%	11.76%	34.62%	No
SnapOdo	**0.1.0**	**Facebook**	**11.76%**	**51.61%**	**1.24%**	**11.76%**	**46.53%**	**Yes**
Close Up	2.2 Forest	Facebook	58.60%	54.38%	0.17%	57.67%	54.26%	No
Stevenson Student Activities	5.63	Facebook	42.72%	48.29%	0.46%	42.55%	46.91%	No
Metal Tombstone	4.1 Pea Green	FacebookAudience	60.50%	19.88%	0.12%	61.00%	19.65%	No
Personal Tracker	1.5	FacebookAudience	54.72%	30.54%	1.14%	64.15%	30.78%	No
Paris Metro Map	1.1	FacebookAudience	23.08%	25.72%	2.81%	23.08%	25.83%	No
Burak Yeter Songs	1.4	FacebookAudience	50.00%	50.00%	0.00%	50.00%	50.00%	No
Maquillaje Halloween 2017	13.0.0	FacebookAudience	24.49%	12.66%	1.55%	24.23%	12.56%	No

TABLE II

CATEGORIZED EXCEPTIONS REPORTED BY MONKEY TEST.

Exception Category	**#App**	**%Failure**	**Library**	**Version Updated (#)**	**Error Message Example**
AbstractMethodError	17	73.91%	OkHttp	3.0.0-RC1 – 3.9.0 (17)	java.lang.AbstractMethodError: abstract method ”void okhttp3.Callback.onResponse(okhttp3.Call, okhttp3.Response)”
ClassNotFoundException	4	17.39%	OkHttp	3.2.0 – 3.9.0 (2) 3.3.0 – 3.9.0 (1) 3.3.1 – 3.9.0 (1)	java.lang.NoClassDefFoundError: Failed resolution of: Lokhttp3/internal/Platform
FacebookException	52	77.61%	Facebook	4.0.1 – 4.26.0 (1) 4.1.0 – 4.26.0 (2) 4.2.0 – 4.26.0 (1) 4.3.0 – 4.26.0 (1) 4.5.0 – 4.26.0 (1) 4.6.0 – 4.26.0 (8) 4.7.0 – 4.26.0 (1) 4.8.0 – 4.26.0 (1) 4.8.2 – 4.26.0 (8) 4.9.0 – 4.26.0 (15) 4.16.0 – 4.26.0 (1) 4.17.0 – 4.26.0 (12)	java.lang.RuntimeException: A valid Facebook app id must be set in the AndroidManifest.xml or set by calling FacebookSdk.setApplicationId before initializing the sdk

This table only lists library related exception cases.
#App: the number of failed apps that reported this exception.
%Failure: the percentage of apps that failed for this exception among all the monkey failures of this library.

V. ROOT CAUSE ANALYSIS

Our two-stage experiment in Section IV demonstrates the occurrence of app runtime behavioral deviations after API-compatible library updates and shows that library updating is not as straightforward as the existing work [22] claimed it to be. In this section, we deep-dive into the failure cases in our tests to study the factors that impede library updating.

A. Findings from Monkey Testing

We analyze the *Monkey* logs of the failed apps and categorize all failures according to the reported exception messages. Though we did a pre-run on *Monkey* for each app to filter out those apps with innate faults, a flawed app can still survive the first run and crash in the second run because of the random behavior triggered by *Monkey*. Since we are not working on app debugging, investigating the failure reasons for all failure cases would be a wild-goose chase. Here, we concentrate only on the failures that have an obvious relationship with updating libraries. We consider all the failures that contain library specific keywords in their exception messages. Table II provides an overview of those failure instances.

It can be observed that both *OkHttp* and *Facebook SDK* have interesting exceptions at runtime after updating them, while we discovered nothing of interest for *Facebook Audience*. For *OkHttp*, 17 apps failed because of *AbstractMethodError* and 4 apps failed because of *ClassNotFoundException*, which together make 91.30% of all failure cases for *OkHttp*. Library *Facebook SDK* has 52 apps throwing *FacebookException*, which equals 77.61% of all failures for that library.

1) AbstractMethodError: This exception is thrown when an abstract method is called but the definition of a target class, here class *okhttp3.Callback*, is incompatible with the currently executing method. All of the 17 crashes happened when updating *OkHttp* from version 3.0.0-RC1 to 3.9.0. Version 3.0.0-RC1 is the first version with the 3.x API. This is a breaking upgrade that even changed their package name from *com.squareup.okhttp* to *okhttp3*. Version 3.9.0 is the latest *OkHttp* version in our library repository. With all those background information and our test setting that libraries are always updated to their newest version among all the compatible versions, this is a strong indication for incompatible changes between those two library versions.

```java
// version 3.0.0-RC1  release date: 2016-01-02
public interface Callback {
void onFailure(Request request, IOException e);
void onResponse(Response response) throws IOException;
}

// version 3.0.0  release date: 2016-01-13
public interface Callback {
void onFailure(Call call, IOException e);
void onResponse(Call call, Response response) throws
  IOException;
}

// version 3.9.0  release date: 2017-09-03
// the same as 3.0.0
```

Listing 1. Evolution trace of *okhttp3.Callback* class.

Source code analysis: Library *OkHttp* is open source, and we investigate the source of *okhttp3.Callback* and find its evolution trace, which is shown in Listing 1. We find that in version 3.0.0, *OkHttp* modified the interfaces defined in *Callback* by taking an additional *Call* object as a parameter for both *onFailure* and *onResponse* interfaces to facilitate invocations to the *Call* object inside the *Callback* as described in its changelog. This change remained up to the newest version. This kind of mismatch should be detected as an incompatibility between versions, and its update should be disallowed in our test settings. However, *LibScout* detects library invocations via root package matching. Since interface implementations are usually named under a host package prefix (e.g., `com.host.package.Callback`), they are attributed as a host call by *LibScout* when invoking *onResponse* interfaces of a *Callback* host implementation and escape from the library compatibility check. To eliminate this kind of false positive cases, *LibScout* should also take the library's public interfaces into consideration. In this case, all of the 17 apps will be non-updatable under these new constraints. Also, the claimed update rate by earlier work should be updated.

2) *ClassNotFoundException:* This exception is thrown when a classloader failed to load the target class by name in the classloader chain. While updating *OkHttp* from various versions to the newest one, four apps were reported as a crash because of a failure in finding class *okhttp3.internal.Platform* in the path of the library update.

Source code analysis: We discovered that *Platform* class in versions before 3.4.0-RC1 of *OkHttp* is named as *okhttp3.internal.Platform*, which conflicts with the one named *okhttp3.internal.platform.Platform* in version 3.9.0. From the exception stack, we know that those failed apps all include *OkHttp Logging Interceptor* (*okhttp3.logging.HttpLoggingInterceptor*) library, which is a sibling library of *OkHttp* and uses it as a dependency. As mentioned before, *LibScout* uses a root package matching to detect library invocations. That way, invocations between sibling libraries like *OkHttp Logging Interceptor* and *OkHttp*, whose method signatures start with the same root package, will be misreported as a library internal call. Thus, changes in interfaces exposed to sibling libraries will be missed by *LibScout*. Even finding such cases with auxiliary information besides the library API is hard, for instance, the *OkHttp*

```java
// version 4.18.0 November 30, 2016
public static synchronized void sdkInitialize(...){}

// version 4.19.0 January 25, 2017
@Deprecated
public static synchronized void sdkInitialize(...) {
...
// We should have an application id by now if not throw
if (Utility.isNullOrEmpty(applicationId)) {
throw new FacebookException("A valid Facebook app id
  must be set in the AndroidManifest.xml or set by
  calling FacebookSdk.setApplicationId before
  initializing the sdk.");
}
...
}

// version 4.26.0 August 24, 2017
// the same as 4.18.0
```

Listing 2. Evolution trace of *sdkInitialize* method.

changelog for the whole okhttp family does not mention an internal *Platform* class renaming, since this class is not supposed to be invoked from outside this library family. To update the library based on API compatibility more effectively, a more fine-grained matching filter for sibling libraries and internal public interfaces should be applied to the lib usage detection logic of *LibScout*, which would very likely decrease the reported rate for updates to the max version. For example, in this case, 3 out of 4 apps could still be updated to the intermediate library version 3.3.1, the last version before *Platform* renaming.

3) *FacebookException:* This exception is a Facebook custom exception that is thrown when an internal error happened in *Facebook SDK*. In our test set, 52 *Facebook SDK* failure apps reported an application ID missing error during SDK initialization after update to version 4.26.0 (the newest *Facebook SDK* version in our repository) and the original library versions vary from 4.0.1 to 4.17.0. Thus, the *Facebook SDK* initialization must have changed with some version after 4.17.0. We look into the *Facebook SDK* upgrade guide and find a description about upgrading 4.18.0 to 4.19.0:

"The Facebook SDK is now auto-initialized on Application start. If you are using the Facebook SDK in the main process and don't need a callback on SDK initialization completion, you can now remove calls to FacebookSDK.sdkInitialize."

Source code analysis: To verify if this modification is the main reason of failures, we check the source code of *Facebook SDK* and discover that before version 4.19.0, the *Facebook SDK* is usually initialized manually via interface *FacebookSdk.sdkInitialize* (see Listing 2). The application ID could be set either in *AndroidManifest.xml* file or *setApplicationId* method. The ID could be set either before or after *sdkInitialize*. However, starting from version 4.19.0, interface *sdkInitialize* is labeled as deprecated, and now it is called by *Facebook SDK* automatically without explicit code invocation in host components. Deep within the initialization code, we find that the application ID must be set before invoking *sdkInitialize* as shown in Listing 2 or otherwise an exception is thrown. Thus, the application ID should be set as early as possible to avoid any failure. In

```
// file assets/www/js/services.js
facebookConnectPlugin.api('/me?fields=about,bio,
email,name,first_name,last_name&access_token=' +
   authResponse.accessToken, null, ...);
```

Listing 3. Graph Request in *SnapOdo*.

fact, to support automatic initialization, Facebook imported a new ContentProvider component *FacebookInitProvider* in 4.19.0. ContentProvider components can be initialized at the beginning of app launching ahead of any other components. By invoking *FacebookSdk.sdkInitialize* in *FacebookInitProvider*, the *Facebook SDK* can be initialized at a very early stage. In a standard *Facebook SDK* integration, *FacebookInitProvider* in *Facebook SDK*'s custom library *AndroidManifest.xml* file will be merged with the app's *AndroidManifest.xml* file during app building, and the application ID should be configured in *AndroidManifest.xml* file to ensure the application ID is available during *FacebookInitProvider* initialization at app launching time. Changes to the *AndroidManifest.xml* are excluded from our test settings, and all the original library SDK configuration is kept as in the original app. Thus, some apps with lower library versions that set the application ID after invoking *sdkInitialize* will fail with the newer library versions.

B. DroidMate Finding

To explore the incompatibility of libraries beyond crashes, we investigate the case for which we found a deviation in the runtime behavior in the DroidMate test after updating the *Facebook SDK* library. The *Facebook SDK* of app *SnapOdo* is updated from version 4.15.0 to the latest version 4.26.0 and after that failed to login to the facebook account. From the official changelog, we know that a Graph API upgrade occurred in version 4.16.1. According to the changelog of Graph API version 2.8, some deprecations happened, including the removal of a *"bio"* field on the *User* node. In Android apps, *GraphRequest* is usually created by either JavaScript or Java code integration with some fields defined in the graph path string. We decompile the *SnapOdo* package and find the *GraphRequest* creation in a JavaScript file as shown in Listing 3. The usage of the *"bio"* field is incompatible with the new Graph API used in newer *Facebook SDK* versions and leads to the login failure in this app. This case reflects potential updating obstacles beyond API-compatibility. For both integration options, field *"bio"* works just as a part of a string parameter that is definitely out of the range of *LibScout* detection.

C. Case Study

From those failure cases, we noticed that even though the APIs of different library versions are compatible, some internal execution logic changes could prohibit a simple drop-in update. We use the factors discovered in case of the Facebook exception as a case study and perform a large-scale analysis to evaluate the prevalence of such impeding factors for drop-in replacements in other libraries. It is worth noting that 1) *Facebook SDK* labeled interfaces which are not recommended to use after some updates with a *"deprecated"*

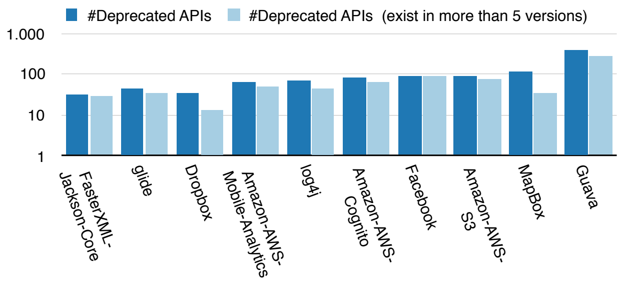

Fig. 7. Number of public deprecated APIs in libraries source code and the number of them that exist in more than 5 versions.

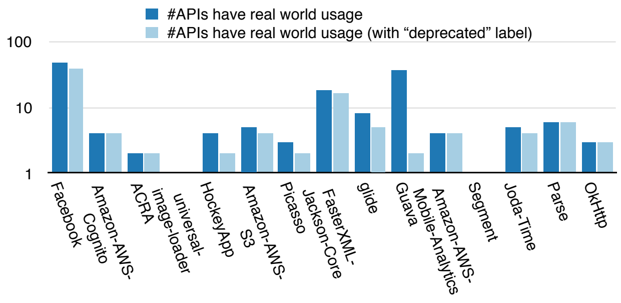

Fig. 8. Number of public deprecated APIs that exist in more than 5 versions and that are used in apps (total vs. with *"deprecated"* label).

annotation instead of removing them directly, which puts them outside of *LibScout*'s API compatibility analysis; 2) a drop-in update cannot change the configurations defined in *AndroidManifest.xml* file, which could be different between different versions.

Deprecated methods: We carried out a statistical analysis of the source code of 1430 different versions of 44 open source libraries that we gathered from maven repository. We extend *javadocextractor*[8], which is a wrapper of *javaparser*[9], to check the occurrence of deprecated interfaces in libraries. We find that 32 of 44 (72.73%) libraries have deprecated methods. Among all those libraries with deprecated interfaces, 24 of them have deprecated interfaces present in more than 5 versions, which indicates the prevalence and permanence of deprecated methods. Figure 7 lists the deprecated API details for 10 libraries. To quantify the impact of those deprecated methods in real-world apps, we compared those deprecated interfaces that exist in more than 5 library versions with library invocation calls detected by *LibScout* from a more extensive app repository which contains 9,902,533 profiles for 2,041,017 apps. Since an interface is usually used before being deprecated, we also distinguished the usage situation for both non-deprecated versions and deprecated versions. Our results show that 20 of 24 libraries, 158 APIs in total, are detected as used in real-world apps. In those 20 libraries, 15 of them with

[8]https://github.com/ftomassetti/javadoc-extractor
[9]https://github.com/javaparser/javaparser

TABLE III
LIBRARY MANIFEST CHANGES ACROSS DIFFERENT VERSIONS.

Manifest Entries	#Changed Cases	#Library Concerned
Activities	16	ACRA, CleverTap, Facebook-Audience, Facebook, HockeyApp, Paypal, braintree-payments, leakcanary, vkontakte
Services	7	ACRA, MapBox, Parse, braintree-payments
Content Providers	2	ACRA, Facebook
Broadcast Receivers	3	CleverTap, vkontakte
Permissions	10	ACRA, CleverTap, Facebook-Audience, HockeyApp, Parse, Paypal, braintree-payments, leakcanary

TABLE IV
RULES TO IDENTIFY INCOMPATIBLE UPDATES WHEN CONSIDERING OUR DISCOVERED FACTORS.

Library	Side Effect	Original Version	Update Version	Features
OkHttp	AbstractMethodError	= 3.0.0-RC1	> 3.0.0-RC1	Existing host implementation of *okhttp3.Callback*
OkHttp	ClassNotFoundException	< 3.4.0-RC1	>= 3.4.0-RC1	Using library LoggingInterceptor together with OkHttp.
Facebook Sdk	FacebookException	< 4.19.0	>= 4.19.0	Invoking *sdkInitialize* without either invoking *setApplicationId* or defining applicationId in AndroidManifest.xml.
Facebook Sdk	Login Failed	< 4.16.1	>= 4.16.1	Using field ”bio” in graph requests.

TABLE V
RESULTS OF LIBRARY UPDATABILITY RE-ESTIMATION(* INDICATES RE-ESTIMATION RESULTS).

Library	#Apps	#Updatable	#Latest Updatable	*#Updatable	*#Latest Updatable
OkHttp	104,046	97,176 (93.40%)	45,962 (44.17%)	94,550 (90.87%)	37,934 (36.46%)
Facebook Sdk	199,007	187,191 (94.06%)	145,817 (73.27%)	187,189 (94.06%)	134,035 (67.35%)

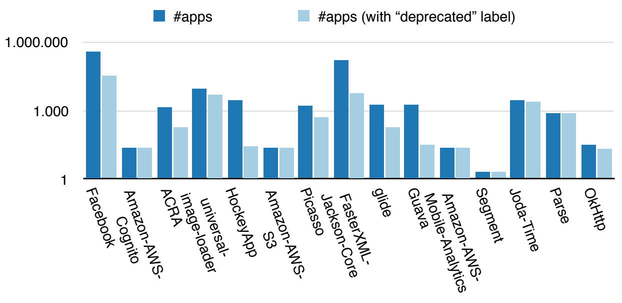

Fig. 9. Number of apps that use public deprecated APIs (exist in more than 5 versions) and their usage (total vs. with *”deprecated”* label).

94 (59.49%) APIs in total, are used under deprecated status. Figure 8 shows the target API usage details and highlights the deprecated usage for 15 libraries. The amount of apps affected by deprecated APIs is also remarkable. In our results, 561,671 app profiles are reported containing target API calls, while 47,966 of them include those calls under deprecated status. Figure 9 lists the number of apps that include target APIs under deprecated status for 15 libraries. From the results above, we can see that most of the libraries have deprecated methods. A deprecated method is supposed to be removed in the near future, but based on our results, those methods usually remain for an extended period, which gives developers the chance to keep using outdated code and also brings false positives to API-compatible library updating. The prevalence of deprecated cases further shows that a plain drop-in replacement cannot work as good as expected.

Manifest changes: Usually, library developers define necessary components and permissions in library manifest files which will be automatically merged with the app's manifest file when building the app with *Gradle*. This process could be opaque to app developers. In a drop-in replacement library updating, those manifest modifications, e.g., *FacebookInitProvider* registration in our test, will be ignored since no app rebuilding is performed. This can impede the library updating as we have discovered for the *Facebook SDK*. To gain insights on the extent of this problem, we gathered 362 Android Archive packages (i.e., manifest plus code) for 15 libraries and analyze the component and permission changes in manifest files across different versions. The result is shown in Table III. Among all 15 libraries, 16 Activity changes happened in 9 libraries, 7 Service changes happened in 4 libraries, 2 ContentProvider changes in 2 libraries, 3 BroadcastReceiver changes in 2 libraries, and 10 permission changes in 8 libraries. In other words, 11 out of 15 libraries have at least one entry modified between versions. These frequent changes indicate a high potential for incompatible drop-in replacements despite API compatibility.

D. Library Updatability Re-Estimation

Our dynamic testing results reveal that failed library updates come from both flaws in the *LibScout* tool and library internal changes. Our case study confirms the prevalence of those factors across different libraries. The API-compatibility based updatability rate reported by *LibScout* should be adjusted. Here, we set *OkHttp* and *Facebook SDK* libraries as two typical examples and re-estimate the API-compatible based updatability rate after considering the discovered factors. We use the same app set as in our automated UI tests (332,432 apps in total). First, we gathered the theoretical API-compatible based updatability rate according to the compatibility definition of *LibScout* [14]. Then, we create rules to identify apps with incompatible library updates when considering our findings, as shown in Table IV. Lastly, we scan app profiles and filter out all the apps that match one of the rules. Method call information like *sdkInitialize* and *setApplicationId* is gathered by *LibScout* already, we only need to extend it with host interface implementation checking, manifest metadata (applicationId), and JavaScript analysis results (field ”bio”). Considering field ”bio” can be added to graph requests through not only JavaScript but also Java code, we take advantage of the Artist [15] tool to filter any field ”bio” usage in graph request construction relevant string flows. The final re-estimation results are shown in Table V. We find that the updatability rate varies between 93.40% to 90.87% for *OkHttp* and stays (94.06%) for *Facebook SDK*. However, the updatability rate to the latest version varies more significantly between 44.17% and 36.46% for *OkHttp* and between 73.27% and 67.35% for *Facebook SDK*. The re-estimation result exhibits a decrease of the updatability rate compared to plain *LibScout*, in particular, the latest version updatability rate, when taking our discovered impeding factors into consideration. With runtime app behavior profiling, we find that a drop-in replacement for library updates is *technically* possible, but if a functioning continuous updating model is expected, the joint efforts from library developers, app developers, and *LibScout* tool developers are necessary to address those factors.

VI. Discussion

We discuss the limitations and prospects of our study.

A. Research Sample

We used three libraries from different categories for our study. Although those are popular libraries, their results might not generalize and cover all kind of potential problems. However, our work still revealed important issues of library updates and shows that API-compatibility alone is not a good indicator for library updates. Further, we investigated 1.4k other library versions and 2M real word apps for identical problems and could confirm the prevalence of those problems, which we think makes them representative. Moreover, scaling the analysis to larger-scale and more intricate problems is naturally limited by the small-scaling of dynamic testing. Future work could investigate certain problem classes in a focused way.

B. Entangled Dependencies

A crucial observation of our tests is entangled dependencies between different libraries and even the host app. For instance Figure 10: both the app and the library L_a depend on library L_b. When updating L_a, not only $Call_{ha}$ but also $Call_{ab}$ should be taken into consideration. A more complicated case is that

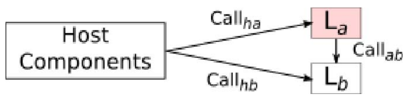

Fig. 10. An example of entangled dependencies inside an app.

the host application creates an object from secondary dependency L_b and passes it to L_a as a parameter. It is not supported by our classloader customization based test framework. API static analysis result from our test samples shows that 1.7% of library APIs could be affected by this problem and also two failure cases in the unknown crashes of the dynamic test are confirmed to be related to this problem. This exceptional case is the limitation of our framework setting, but we here only focus on incompatibility cases brought by library updating. The crash cases reported in Section V are not affected by this exceptional case. Apart from that, our framework ensures that all dependencies are correct for host app and updated library respectively since the original library and its dependencies are still in the app. Obviously, the numbers reported in previous Section V-D are an *optimistic* estimate when no direct dependency conflicts occur. The situation for entangled dependencies in real-world might be far from desirable. We looked into the impact of dependencies on library updatability. We crawled library dependency information from Maven Repository[10] and limit a library's possible update to only versions that share the same dependency set with the original one. Compared to a purely API-compatibility based update, this API/dependency-based update shrinks the updatability rate significantly. The rate for the same 332,432 apps reduces by 47.95% (93.40%-45.45%) for *OkHttp*, 40.37% (94.06%-53.69%) for *Facebook SDK*, and 36.38% (99.94%-63.56%) for *Facebook Audience*. This multiple dependency situation is not a corner case. Static analysis of those apps shows that 57.50% of the apps that integrate *OkHttp* have invocations to *OkHttp*'s dependencies in either their host code or *other* libraries and even 96.03% for *Facebook SDK* and 97.76% for *Facebook Audience*. Hence, whether a lib can be updated in reality might also be constrained by further dependencies by other libs or app code to its own dependencies and, hence, in case of a conflict, prevent an update of the target lib without doing extensive updates of other libraries (potentially creating a ”dependency hell”).

C. Framework and UI-based Testing Limitations

Although very carefully designed to avoid errors/crashes of the apps and libraries due to erroneous drop-in replacements, we cannot entirely exclude that some of the crashes of apps come from our framework, since it is unrealistic to debug all the failed apps from our testing. However, our investigation focused on those crashes with clear problems stemming from the library integration and internal changes. Further, we only test control flows starting at Activities and achieve with this

[10]https://mvnrepository.com/

on average 35% app block coverage. Thus, our results form a lower bound on the potential problems of the tested libraries. The emphasis of our work is on confirming the existence of API-compatibility based update problems and identifying advice for future library update tool developers/researchers about what impedes library updatability.

D. Efforts from Multiple Parties

The main idea of this work is evaluating ways of (supporting developers in) maintaining dependencies, starting with evaluating the feasibility of drop-in updates and discussing the relevance of our results for library updatability. Our discovered problems are intricate, and hence any support for automatic lib updates or even tools that help developers in making a judgment of the library updatability have to consider those non-trivial problems, e.g., clear connections to changelogs, changed data structures, or code annotations. Multiple parties are involved in the library update chain and there is a call for action to better support lib updates in the mobile ecosystem, including better tools for app developers to judge and realize library updates or a call to system vendors to rethink the static linking of libraries in favor of more dynamic approaches (e.g., on Linux) that not only can profit compartmentalization of third-party code [29] but also its updatability.

E. Updating in Automated App Testing

In our DroidMate test, we observed a case of a highly non-deterministic app that resulted in exploration noise. The reason for the non-determinism is that the app has a lot of random actions, for example, loading different advertisements in different runs. Considering that our update framework opts in library updates as a replacement of the original library without any actual app code modification, we plan to investigate the possibility of migrating our lightweight framework to blacklisting unwanted libraries in automatic app testing.

VII. Conclusion

Outdated third-party libraries are prevalent in apps. To alleviate the unpleasant situation, prior work suggested an API-compatibility based library update solution using drop-in replacements of outdated libraries. In this paper, we study the library updatability using such drop-in updates. We implemented a library update framework for Android and used it on 3,000 real-world apps for 3 popular libraries. Using dynamic testing of those apps, this gave us insights into the runtime behavior of an API-compatibility based updating solution. To discover more intricate incompatibility cases, an automated user interface testing was carried out on 15 apps both before and after library updates. Our tests revealed intricate factors that prevent a drop-in replacement of libraries. Studying the source code of libraries that failed to update and using static app analysis, we confirm the prevalence of those problems in other libraries. Our re-estimation of prior estimates of the library updatability rate under consideration of the discovered impeding factors shows a decrease in the rate by more than half due to entangled library dependencies. This work is the first to confirm the existence of API-compatibility based update problems and can provide valuable insights for future library update tool developers/researchers on what should be taken into account when updating libraries.

References

[1] "Amigo," https://github.com/eleme/Amigo, April 2019, accessed: 2019-04-12.

[2] "Andfix," https://github.com/alibaba/AndFix, April 2019, accessed: 2019-04-12.

[3] "Appbrain: Android library statistics," https://www.appbrain.com/stats/libraries, April 2019, accessed: 2019-04-12.

[4] "Applovin security notice," https://blog.applovin.com/applovin-security-notice/, April 2019, accessed: 2019-04-12.

[5] "Droidplugin," https://github.com/DroidPluginTeam/DroidPlugin, April 2019, accessed: 2019-04-12.

[6] "Improve your code with lint checsdkssfks," https://developer.android.com/studio/write/lint, April 2019, accessed: 2019-04-12.

[7] "Instant run," https://developer.android.com/studio/run#instant-run, April 2019, accessed: 2019-04-12.

[8] "Js-binding-over-http vulnerability and javascript sidedoor: Security risks affecting billions of android app downloads," https://www.fireeye.com/blog/threat-research/2014/01/js-binding-over-http-vulnerability-and-javascript-sidedoor.html, April 2019, accessed: 2019-04-12.

[9] "Tinker," https://github.com/Tencent/tinker, April 2019, accessed: 2019-04-12.

[10] "Ui/application exerciser monkey," https://developer.android.com/studio/test/monkey, April 2019, accessed: 2019-04-12.

[11] "Update your android apps," https://support.google.com/googleplay/answer/113412?hl=en, April 2019, accessed: 2019-04-12.

[12] "What happened to the android update alliance?" https://arstechnica.com/gadgets/2012/06/what-happened-to-the-android-update-alliance/, April 2019, accessed: 2019-04-12.

[13] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI Ripping for Automated Testing of Android Applications," in *Proc. 27th IEEE/ACM International Conference on Automated Software Engineering (ASE '12)*, 2012.

[14] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proc. 23rd ACM Conference on Computer and Communication Security (CCS'16)*. ACM, 2016.

[15] M. Backes, S. Bugiel, O. Schranz, P. von Styp-Rekowsky, and S. Weisgerber, "Artist: The android runtime instrumentation and security toolkit," in *Proc. 2nd European Symposium on Security and Privacy (EuroS&P '17)*. IEEE, 2017.

[16] A. R. Beresford, "Whack-a-mole security: Incentivising the production, delivery and installation of security updates," in *IMPS@ ESSoS*, 2016.

[17] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall, "Brahmastra: Driving apps to test the security of third-party components." in *Proc. 23rd USENIX Security Symposium (SEC '14)*. USENIX Association, 2014.

[18] N. P. Borges Jr., J. Hotzkow, and A. Zeller, "Droidmate-2: A platform for android test generation," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 916–919. [Online]. Available: http://doi.acm.org/10.1145/3238147.3240479

[19] Y. Chen, Y. Li, L. Lu, Y.-H. Lin, H. Vijayakumar, Z. Wang, and X. Ou, "Instaguard: Instantly deployable hot-patches for vulnerable system programs on android," in *Proc. 25th Annual Network and Distributed System Security Symposium (NDSS '18)*. ISOC, 2018.

[20] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei, "Adaptive android kernel live patching," in *Proc. 26th USENIX Security Symposium (SEC' 17)*. USENIX Association, 2017.

[21] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?(e)," in *Proc. 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*, 2015.

[22] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep me updated: An empirical study of third-party library updatability on android," in *Proc. 24th ACM Conference on Computer and Communication Security (CCS'17)*. ACM, 2017.

[23] R. Duan, A. Bijlani, Y. Ji, O. Alrawi, Y. Xiong, M. Ike, B. Saltaformaggio, and W. Lee, "Automating patching of vulnerable open-source software versions in application binaries," in *Proc. 26th Annual Network and Distributed System Security Symposium (NDSS '19)*. ISOC, 2019.

[24] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proc. 35th International Conference on Software Engineering (ICSE '13)*. IEEE, 2013.

[25] B. Li, Y. Zhang, J. Li, R. Feng, and D. Gu, "Appcommune: Automated third-party libraries de-duplicating and updating for android apps," in *Proc. 26th International Conference on Software Analysis, Evolution and Reengineering (SANER'19)*. IEEE, 2019.

[26] F. Long and M. Rinard, "Automatic patch generation by learning correct code," *ACM SIGPLAN Notices*, vol. 51, no. 1, pp. 298–312, 2016.

[27] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proc. 25th International Symposium on Software Testing and Analysis (ISSTA '16)*. ACM, 2016.

[28] C. Mulliner, J. Oberheide, W. Robertson, and E. Kirda, "Patchdroid: Scalable third-party security patches for android devices," in *Proc. 29th Annual Computer Security Applications Conference (ACSAC '13)*. ACM, 2013.

[29] P. Pearce, A. Porter Felt, G. Nunez, and D. Wagner, "AdDroid: Privilege separation for applications and advertisers in Android," in *ASIACCS'12*. ACM, 2012.

[30] P. Salza, F. Palomba, D. Di Nucci, C. D'Uva, A. De Lucia, and F. Ferrucci, "Do developers update third-party libraries in mobile apps?" in *Proc. 26th Conference on Program Comprehension (ICPC '18)*. ACM, 2018.

[31] D. R. Thomas, A. R. Beresford, T. Coudray, T. Sutcliffe, and A. Taylor, "The lifetime of android api vulnerabilities: case study on the javascript-to-java interface," in *Cambridge International Workshop on Security Protocols*. Springer, 2015.

[32] D. R. Thomas, A. R. Beresford, and A. Rice, "Security metrics for the android ecosystem," in *Proc. 5th ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM '15)*. ACM, 2015.

[33] T. Watanabe, M. Akiyama, F. Kanei, E. Shioji, Y. Takata, B. Sun, Y. Ishi, T. Shibahara, T. Yagi, and T. Mori, "Understanding the origins of mobile app vulnerabilities: A large-scale measurement study of free and paid apps," in *Proc. 14th International Conference on Mining Software Repositories*. IEEE, 2017.

[34] T. Wei, Y. Zhang, H. Xue, M. Zheng, C. Ren, and D. Song, "Sidewinder: Targeted attack against android in the golden age of ad libraries," *Black Hat*, 2014.

[35] M. Y. Wong and D. Lie, "Intellidroid: A targeted input generator for the dynamic analysis of android malware," in *Proc. 23rd Annual Network and Distributed System Security Symposium (NDSS '16)*. ISOC, 2016.

[36] W. You, B. Liang, W. Shi, S. Zhu, P. Wang, S. Xie, and X. Zhang, "Reference hijacking: Patching, protecting and analyzing on unmodified and non-rooted android devices," in *Proc. 38th International Conference on Software Engineering (ICSE '16)*. ACM, 2016.

[37] M. Zhang and H. Yin, "Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications." in *Proc. 21th Annual Network and Distributed System Security Symposium (NDSS '14)*. ISOC, 2014.

[38] X. Zhang, Y. Zhang, J. Li, Y. Hu, H. Li, and D. Gu, "Embroidery: Patching vulnerable binary code of fragmentized android devices," in *Proc. 33rd Conference on Software Maintenance and Evolution (ICSME '17')*. IEEE, 2017.

[39] C. Zheng, S. Zhu, S. Dai, G. Gu, and X. Gong, "SmartDroid: an Automatic System for Revealing UI-based Trigger Conditions in Android Applications," in *Proc. 2nd ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM '12)*. ACM, 2012.