Union under Duress: Understanding Hazards of Duplicate Resource Mismediation in Android Software Supply Chain

Xueqiang Wang, University of Central Florida; Yifan Zhang and XiaoFeng Wang, Indiana University Bloomington; Yan Jia, Nankai University; Luyi Xing, Indiana University Bloomington

https://www.usenix.org/conference/usenixsecurity23/presentation/wang-xueqiang-duress

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.
August 9–11, 2023 • Anaheim, CA, USA
978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.
Union under Duress: Understanding Hazards of Duplicate Resource Mismediation in Android Software Supply Chain

Xueqiang Wang\textsuperscript{1}\textsuperscript{*}, Yifan Zhang\textsuperscript{2}\textsuperscript{*}, XiaoFeng Wang\textsuperscript{2}, Yan Jia\textsuperscript{3}, and Luyi Xing\textsuperscript{2}

\textsuperscript{1}University of Central Florida, xueqiang.wang@ucf.edu
\textsuperscript{2}Indiana University Bloomington, \{yz113, xw7, luyixing\}@indiana.edu
\textsuperscript{3}DISSec, College of Cyber Science, Nankai University, jiay@nankai.edu.cn

1 Introduction

Today’s mobile apps increasingly rely on the software supply chain, third-party libraries (a.k.a., software development kit or SDK) in particular, to provide essential functionalities, like advertisement, analytics, and in-app payment, etc. Integration of the libraries from less reliable sources, however, could bring in security and privacy risks. Prior research shows that the libraries provided by a contaminated supply chain, once integrated into a mobile app, can harvest private data from the app \cite{37, 43, 70, 83, 85, 109} and abuse its privileges and resources \cite{42, 61, 87}. A more recent study further shows that a malicious library could attack other libraries hosted in the same app, e.g., through unauthorized access to the APIs of these libraries to exfiltrate their sensitive information \cite{96}. Natural solutions to mitigate such security risks introduced by the software supply chain include static vetting of library code and runtime inspection of its behavior, to ensure that all known malicious activities (e.g., unauthorized access to another library’s API) will not occur. Surprisingly, we found that even under such protection, a stealthy attack is still feasible across third-party libraries, thanks to a new attack surface on the Android software supply chain.

**Duplicate resource mismediation.** More specifically, a software supply chain includes the components, libraries and tools, and the processes for developing, building and publishing software \cite{45}. For the Android app, a key step in its build process is to package all its resources, including those belonging to the third-party libraries it integrates. Android libraries (Android Archive, AAR) \cite{6} typically contain diverse resources to support their operations, such as a manifest file \cite{3, 6}, asset files that often include JavaScript, and XML resource files that define the libraries’ configurations for network security, backup, etc. (Section 4.1). These resources, together with the resources from the app, are managed by the Android resource compiler (ARC, as defined by Gradle tasks in the Android Gradle plugin \cite{2}, see Section 3) during the build process. For example, each library and the app can come with a manifest file that specifies its components (e.g., activities and services \cite{5}) implemented in the code and security-sensitive attributes of the components.

\textsuperscript{*}The two lead authors contributed equally to this work and are ordered alphabetically.
(e.g., which app/component can access them); these manifests are then merged by the ARC into a single file, allowing one to easily audit all activities and services offered by the app.

A complication during this process is that multiple libraries may have duplicate resources or incompatible attributes for some resources. For example, the libraries mclib and sendtopush both have a server_url parameter in their values.xml files. When this happens, the ARC has to mediate the conflict, for example, keeping one server_url setting in the merged values.xml file. If the conflict is between a third-party library and the app, the ARC can easily resolve it by sticking to the app’s configuration (assuming the app’s server is not malicious). However, a resolution becomes harder to get when the contention happens between different third-party libraries, since a wrong decision could have a serious consequence, leading to a security breach: for example, if ARC keeps the server_url of a malicious library, all messages issued by the legitimate library will automatically go to the server under the adversary’s control. This security risk, which we call duplicate resource mismediation (Duress), is fundamental to the app build process, due to the ARC’s lack of information about individual libraries’ trustworthiness and the absence of isolation between library resources at build time. Given the diversity in library resources and the important roles they play in the host app’s configuration, Duress can have serious security and privacy implications. Despite the importance of the problem, little has been done so far to understand Duress, not to mention any attempt to control this new attack surface.

**Duress attacks.** In our research, for the first time we systematically investigated the Duress risks. We explored how ARC mediates duplicate resources from third-party libraries, and introduced a new attack surface in ARC that allows attackers to compromise resources of a victim library compiled in a malicious library compiled in the same app. The Duress risks are problems in ARC – part of the app building systems such as Android Studio, rather than the Android operating system. They are inherent to the Android app supply chain, specifically due to untrusted, malicious/invasive third-party libraries. These risks may lead to different categories of security attacks that can either lower down the protection of other libraries, or mislead users or the libraries to perform insecure operations.

The first category is insecure resource deduplication (Section 3.2). ARC allows a high-priority library to directly override the resources (e.g., assets, manifest attributes) in a low-priority library. This policy becomes a security weakness if the adversary is capable of raising a malicious library’s priority over that of a legitimate library being targeted, which was found to be feasible (Section 3.1). When this happens, the resource crafted by the adversary will replace the sensitive resource of the target library. For example, we found that the SDK of Razorpay, a popular Indian payment platform, has a CDN URL that can be overwritten by the link provided by a malicious library; as a result, sensitive user data, including banking OTP, and user credentials of Amazon Pay, Paypal and others, will be exposed to the adversary (Section 3.2). As another example, we found that a malicious library could replace the AWS credentials included in the SDK of MistPlay – a leading play-game-to-earn platform, so as to mislead the SDK to upload gamers’ streaming data to an attacker-specified AWS account (Section 5.4). Notably, unlike the prior attacks in which unauthorized operations are performed by malicious code, the Duress attack does not involve any illicit action, as all required changes to an app’s resources are done by ARC.

In the second category is insecure resource merge (Second 3.3). ARC is designed to merge duplicate manifest elements (e.g., declarations of the same activity), instead of overriding one with another, whenever possible (in the absence of conflicting attributes). Most importantly, ARC allows a library to declare a component even though the component’s code is defined in another library. As a result, the adversary can strategically select the attributes missing in the target library’s manifest and configure them in the manifest of the malicious library, in an attempt to lower down the target library’s protection. An example is the internal WebView in the applovin library, whose manifest can be contaminated by the malicious library during the merge with android:exported and deeplink-related intent filters, rendering the WebView remotely accessible to the adversary (via a malicious website or deeplinks from a third-party app).

**Finding Duress risks in the wild.** To understand the scope and impacts of the Duress risk on the Android software supply chain, particularly the pervasiveness of sensitive resources in libraries, duplicated resources across libraries, and apps affected by duplicate resources, we developed a methodology to automatically detect the attack opportunities, integration risks and affected apps. A key challenge is how to identify sensitive resources, not only from highly-structured library resources, such as manifest attributes predefined by Android, but also from ad-hoc resources specified by library developers. To this end, we utilize a semantic-driven approach to cluster library resources, label the clusters with sensitive resource types, and then classify other resources to these clusters based upon their semantic closeness to the clusters. In this way, our approach identifies 10 security- or privacy-sensitive resource types (clusters), such as cloud backend URLs and credentials that can be used to compromise the backend servers and cause data exposure, privacy disclosure related to privacy compliance, technical support messages that can be exploited for technical support scams, etc. Once the sensitive resources are detected in libraries, we further perform cross-library and library-app analysis to find the integration risks and their impacts on real-world apps.

We ran our methodology on two datasets with over 23K libraries and 150K apps and, to our surprise, discovered that 18.4% libraries could be exposed to the Duress threat, given their involvement of sensitive resources. Exploiting these opportunities, the adversary could mislead users and libraries or lower down their protection, e.g., cheating the users of
App developers often integrate different libraries that share highly sensitive resources, leading to unwanted app behaviors, e.g., introducing insecure attributes into manifest files, exposing user files with file providers, and polluting network security configurations, etc. In particular, we found that even popular libraries, such as Appsflyer and Vungle, are competing with each other when specifying backup files in their manifests. Their recommendations for resolving such conflicts are interesting: rather than generating a consolidated version of the backup rules, they both recommend developers to override the backup rules of the other library, causing the backup policies of the other library to be disabled and library data to be unexpectedly exposed to the Google Drive or adh accesses. We reported our findings to all affected parties (i.e., Google, and app developers) and also discuss the potential mitigation of the Duress risks. We released our datasets, attack demos, and responses from affected parties online [24].

Contributions. We outline the paper’s contributions below:

- Discovery of the new attack surface. We discovered a new attack surface on the Android software supply chain, whose build process cannot effectively mediate duplicated resources from different libraries. This opens an opportunity for the adversary to contaminate the supply chain with a carefully crafted library to perform a highly stealthy, cross-library attack, which lowers down a target library’s protection or misleads the target or its users to expose sensitive information.

- Understanding of the new threat. We performed the first systematic study on the impacts of the new threat. Our analysis has brought to light the pervasiveness of the risks on the supply chain and the presence of potentially affected real-world apps, which highlight the importance of elevating security protection of today’s supply chain to address the threat.

2 Background

Using third-party libraries in Android app supply chains.

App developers often integrate third-party libraries into the app supply chains in order to ease app development process with function reuse. These libraries, although facilitating faster development, have become the major cause of app supply chain attacks: the untrusted libraries are compiled in the app package and will run in the same space as app’s first-party code and other libraries. Therefore, they can lead to various attacks targeting the app [37, 42, 43, 61, 70, 83, 85, 87, 109], and even the other libraries hosted in the same app [96].

In addition to the code, also introduced in the supply chains by the third-party libraries are the library resources. Specifically, Android libraries (Android Archive, AAR) may contain all types of resources that an app can have [1, 4], e.g., a manifest file that defines the properties of library components and required hardware/software features, several XML files that store library strings and layouts, and even security-related configuration files. Because many library resources are highly sensitive, any unexpected modifications to them may lead to serious security and privacy concerns.

Building third-party libraries into Android apps. Android has developed a fully-automated build process to manage third-party libraries, and compile them into an app package. Figure 1 shows an overview of this process. In the first step, an app developer declares the libraries as app dependencies in a configuration file such as build.gradle. Then, the configuration file is passed to Gradle for automatically resolving the dependencies. Essentially, Gradle runs a few tasks [57] to check transitive dependencies (i.e., dependencies of a dependency), resolve dependency version conflicts, and retrieve dependencies from remote repositories (if necessary), etc. The output of dependency resolution is a set of libraries (including code and resources) cached in local directories, and a dependency graph that represents the relations between different libraries.

In the next step, the build tools compile the code/resources of all the app and library modules before packing them into an APK. Specifically, Android uses several code compilers (such as dx) to compile and then assemble the code into a single DEX file. Further, Android leverages an Android resource compiler (ARC) in the form of Gradle plugins to compile and combine library resources.

Processing library resources with ARC. Ideally, ARC should put the resources of all third-party libraries into one APK in their original format. However, duplicate resources are common across libraries for a number of reasons, such as common resource names used by library developers (which we discuss in Section 5.2). Hence, ARC needs to mediate duplicate resources to avoid future app errors. The current implementation of ARC supports two methods for resource mediation, i.e., resource deduplication and resource merge. The key idea of these methods is to sort third-party libraries from high priority to low priority according to the order in which they appear in the dependency graph. Then, ARC scans each library for resources, and prioritizes the resources from high-priority libraries during resource mediation.

Threat model. In this work, we study the app supply chain security risks posed by malicious third-party libraries. More specifically, we focus on the faulty logic in the app build process of Android resource compiler (ARC), and reveal a new threat that is not currently defended against by Android build system—a malicious library can negatively impact the sensitive resources of a victim library within the same app, using duplicate resources during the app build process. We do not consider malicious app/library code or vulnerabilities in the operation systems, which may or may not lead to similar
security consequences.

We assume that the adversaries of our concern, i.e., developers of malicious third-party libraries, can gather resource information of victim libraries from public sources on the Internet or by inspecting them from public apps that use the victim libraries. The adversaries can practically promote and get their malicious libraries integrated by many apps, such as by offering monetary incentives or useful functionalities (e.g., advertising, analytics), as reported in prior work [96], as well as by exploiting other attack vectors in the app supply chain, e.g., dependency confusion attacks [34]. The malicious libraries can appear as direct dependencies, or transitive dependencies that are auto-resolved and retrieved by Gradle, etc.

In the meantime, developers of victim libraries may not be fully aware of the threats and implications when the resources of their libraries may be conflicting with those of other libraries in the same app, considering the diverse resources (Table 1) and the less known logic of how ARC may mediate the conflicts.

3 Duress Attacks on Android

As we introduced earlier, resources in high-priority libraries are prioritized by ARC during the app build process. Therefore, for Duress attack to succeed, attackers first need to increase the priority of their malicious libraries to ensure that the resources in these libraries are picked up by ARC. In this Section, we start by describing how ARC determines the priority of libraries, and then propose several strategies that attackers can take to raise the priority of malicious libraries. Based upon the strategies, we illustrate the design of ARC in mediating duplicate resources and discuss how such design can be exploited in practical Duress attacks.

3.1 Raising Priority of Malicious Libraries

Determination of library priorities. At a high level, the resources of an app project are from three conceptual sources: app modules, local libraries, and remote libraries. The app modules refer to the first-party modules that are defined by app developers, such as debug/release build variants. To use a library, the app needs to declare it as a dependency in the build.gradle file. The library could be a library available in a local directory (local library) or on remote repository hosting services (remote library). Our investigation shows that, in the app build process, the app modules have the highest priority, followed by local libraries and remote libraries (Figure 2 (a)).

The priorities of the libraries within the local or remote categories are decided by the dependency graph and the built-in traversal algorithms of Gradle. In Figure 2 (b), we present a most simplified dependency graph: Libraries $L_0$, $L_1$, and $L_2$ are declared in order in the build.gradle file, and $L_1$ is a transitive dependency of $L_2$. Gradle adopts two rules to determine the priorities of these three libraries. First, it uses a “consumer-first” order which prioritzes a library over its dependencies. Hence, $L_2$ has a higher priority than its dependency $L_1$. Second, Gradle honors the ordering of the dependency declarations in build.gradle. As a consequence, $L_0$ has a higher priority than $L_2$ since $L_0$ is declared earlier. Combining the two rules, we have the priorities of the three libraries: $L_0 \rightarrow L_2 \rightarrow L_1$.

Strategy-1: Riding on victim libraries. Given the “consumer-first” rule, a straightforward approach to raising the priority of a malicious library is to turn it into a consumer, i.e., adding victim libraries as dependencies to the malicious library.

This approach is super helpful when targeting a few high-profile victim libraries. However, there are potential challenges to target a large number of victim libraries, which is often the case for attackers. First, in this case, the malicious library has to add all of the victim libraries as dependencies, which will introduce significant overhead to the apps since the dependencies have to be compiled in the apps, even for the apps that don’t use the dependent libraries. Also, a library that depends on an unreasonable number of libraries (that provide irrelevant functionalities) can quickly draw the attention of app developers and even library scanning tools. Therefore, a more advanced strategy is necessary to scale up the attack.

Strategy-2: Riding on Android platform libraries. We noticed that apps often rely on a few Android platform libraries (such as androidx.appcompat). These libraries are preset by Android Studio, and are usually located in the very beginning of the build.gradle file. Thus, in practice, the platform libraries may have higher priorities than third-party libraries. An attacker can design his strategy by combining this observation with the “consumer-first” rule: instead of depending on victim libraries, the malicious library can elevate its priority by depending on a crafted list of platform libraries.

To confirm the effectiveness of this strategy, we created a demo library that depends on three most frequently used platform libraries, i.e., androidx.appcompat, com.google.android.material and androidx.constraintlayout, and published the library on Maven Central [23]. Then we col-
lected 100 build.gradle files from 97 open-source projects on GitHub, and added the demo library as a dependency to the very end of the files. We checked whether the demo library can achieve a higher priority than the other third-party libraries when ARC compiles the 100 build.gradle files, by inspecting the library priorities in ARC. The result proved our hypothetical strategy: the demo library has a higher priority than 96.7% (1,717 out of 1,775) third-party libraries that are used by the open-source projects. Most of the remaining 3.3% libraries (such as com.tencent.edu) rely on platform libraries as well, but we believe attackers can handle them by ensembling multiple strategies.

Discussion. Notably, the demo library we temporarily posted on Maven Central is for an end-to-end, proof-of-concept implementation and confirmation of Strategy-2, which demonstrates the practicality of the strategy in the wild. It does not carry any functional code or sensitive resources.

In the library’s description, we highlighted that the library was for testing purposes and should not be used without contacting us. We also removed the library immediately after our experiment (after two days). If, by any chance, a developer used the library, the only change to the developer’s project is the introduction of the three platform libraries developed by Google (e.g., androidx.appcompat), which are commonly used by apps and are benign.

Strategy-3: Distributing malicious libraries as “local” libraries. It is infeasible for an attacker to affect the order of libraries declared in the build.gradle file of an app, since the order is mostly specified by app developers. However, the attacker can actually affect the way his malicious library can be integrated into an app, i.e., in the form of a local or remote library, which leads to another potential attack strategy.

Recall that local libraries have higher priorities than remote libraries. Therefore, to compromise libraries that are published on remote repositories, an attacker can distribute and promote his malicious library on websites, technical forums, or even code repositories like GitHub, and then instruct the app developers to download and import it as a local library. With this strategy, the malicious library may gain a higher priority than any remote third-party libraries.

3.2 Insecure Resource Deduplication

Resource deduplication mechanism. The most commonly used resource mediation method is deduplication, i.e., overriding one resource with another. To understand how it works, we systematically investigated library resources located in 19 different directories, as specified in Android official documentations [1, 4] (second column of Table 1). Then, we analyzed the ARC source code, and found that these resources are handled by multiple Gradle tasks in ARC, e.g., MergeResources task covers duplicate res/* files and MergeAssets task covers assets/* files. In Table 1, we present the library resources and associated ARC tasks. Our study shows that although the resources differ, their corresponding tasks actually share a base implementation called DataMerger. Hence, below we focus on the illustration of DataMerger.

![Figure 3: Resource deduplication mechanism (the gray boxes represent the libraries whose resources are selected)](image)

As a first step, the DataMerger assigns a priority value to each library based on the dependency graph generated by Gradle (Section 3.1), and sorts the libraries into a list according to their priorities. Then, it scans the library resources in order from the high-priority to low-priority libraries. The outcome of this step is a resource map, where the key of the map is the resource name and the value is the list of libraries (which is also ordered by priorities) that contain such resources. In the last step, based on the resource map, the DataMerger uses the resources from the highest-priority library to override the duplicate resources of other libraries (see Figure 3). This process ensures that all library resources that get compiled into the apps are from the highest-priority libraries. Note that the resources from other libraries are overridden by ARC in the background, without letting the app developers know.

Duress Risk-1: Resource-Overriding. In light of the deduplication mechanism, if a high-priority malicious library creates a crafted resource that duplicates with a sensitive resource in a victim library, the ARC would replace the resource in the victim library with the crafted resource. Then at runtime, the victim library will automatically pick up the crafted resource. Although this attack looks technically simple, it can lead to serious security issues as the crafted resource can effectively mislead the victim library or its users into performing insecure operations.

Razorpay SDK is a potential target for such an attack. Razorpay [78] is a leading online payment solution in India, which has helped over 5M businesses (including Facebook, Disney and PizzaHut) to process customer payments. We found that the Razorpay SDK uses a resource file (res/raw/rzp_config.json) to store SDK configurations, in particular its CDN URL (https://cdn.razorpay.com). The SDK uses the hard-coded CDN URL to download a
JavaScript (JS) file otpelf.js, and then loads the JS code into a WebView for processing the online banking one-time password (OTP). Our investigation shows that an attacker can conduct the Duress attack by providing a duplicate rzp_config.json file with a fake CDN URL within his malicious library. At compile time, if the malicious library and the Razorpay SDK are both added as app dependencies, and the malicious library owns a higher priority, ARC will use the duplicate file containing the fake URL to replace the file in the Razorpay SDK. At runtime, the fake URL will be picked up by the victim library automatically. This allows the attacker to distribute and load arbitrary JS code into the Razorpay’s WebView, leading to attacks such as gathering OTP from end users. Furthermore, Razorpay SDK uses the same WebView for federated authentication to multiple payment platforms, e.g., Amazon Pay and Paypal, etc. Therefore, the malicious code is also capable of collecting credentials of these payment platforms, by continuously monitoring the DOM interfaces. Also interestingly, we suspect that Razorpay is aware of the risk since the JS code distributed by the CDN is encrypted. However, the encryption won’t be able to eliminate the attack from malicious libraries: an attacker can easily reverse engineer the SDK to recover the encryption procedure and the symmetric key, and then encrypt the malicious code before distributing it. Please refer to our website [24] for more attack details.

**Duress Risk-2: Manifest-Overriding.** Another risk that is worthy of a separate discussion is the manifest deduplication risk. Library developers can place security protections on the library components using certain manifest attributes, e.g., adding android:permission to a content provider can limit its access to only clients (or apps) that have the permission. However, we noticed that Android provided a set of node markers (e.g., tools:replace, tools:remove) to high-priority libraries for defining custom duplicate mediation rules. For example, by placing the tools:replace marker to an element in the manifest file, a developer can tell the ARC to replace certain attributes in the lower-priority manifest files using the attributes in the current manifest file.

These node markers contribute to the resolution of conflicts between an app/library and its dependencies. However, they could be abused by malicious libraries as well. Specifically, in cases that a malicious library obtains a higher priority in the app build process, it is capable of manipulating any manifest attributes (including security-sensitive ones) within a lower-priority library, potentially lowering down the protections that were already in place in the victim library. An example is the HUAWEI Mobile Services SDK. By default, its content provider that stores push notifications (PushProvider) is only accessible to apps that have permission.PUSH_PROVIDER permission. However, with this Duress attack, a malicious library can remove this protection by overriding the android:permission attribute, and open the content provider’s access to any other untrusted apps.

**Discussion.** We observe that many third-party libraries contain resources that are intended to be exclusively used by the libraries’ code. Library developers may assume that these resources are authentic even after the libraries were integrated into apps. The presence of resource deduplication in ARC effectively breaks this assumption in cases where the resources from different libraries share the same name. An example is the privacy_url used by both Maxpay and Genome. In order for the libraries to display their own privacy policies within the same app, a reasonable solution for ARC might be providing several copies of privacy_url for the two libraries, rather than allowing one to override another. This leads to our mitigation proposal that we should provide isolation for highly-sensitive resources (see discussions in Section 5.5).

We are not trying to diminish the value of ARC’s work in deduplicating resources, since it not only helps to shrink the sizes of apps but most importantly reduces the developer’s effort in handling a large number of duplicates (e.g., our evaluation of the datasets in Section 5 shows that a library has an average of 2.7 duplicate resources with any other library). However, we believe that there are potential improvements that ARC can make to reduce the Duress risks. First, given that highly-sensitive resources are also deduplicated, a responsible action for ARC could be alert or provide more control to app developers when such resources are to be deduplicated (which is also confirmed by Google’s response, see Section 5.3). Second, since resource deduplication is done following the libraries’ priorities. It might be helpful for ARC to allow app developers to define the priorities in this process, to ensure that the resources in the most trustworthy libraries are chosen.

### 3.3 Insecure Resource Merge

In addition to resource deduplication, another natural choice for processing duplicate resources is to merge them into a single target. This choice is necessary for manifest. Manifest files are commonly used by apps and libraries as a global registry to register app/library components, permissions, and required software and hardware features to the Android system. Since only one manifest file is allowed in the compiled app, ARC needs to merge all the manifest files into one using manifest-related tasks, e.g., ProcessMainManifest or ProcessManifestForPackage.

**Manifest merge mechanism.** The manifest merge process follows the same library priorities as resource deduplication. Not surprisingly, the tasks first scan the libraries according to their priorities, and insert a library into a manifest provider list if the library has a manifest file. Then, the tasks take the manifest file of the main app (i.e., app developer’s manifest) as the initial merging result, and process the libraries in the manifest provider list from the high-to low-priority libraries. In this process, the tasks merge the manifest files sequentially into the merging result, therefore resulting in the merge order of main app manifest → high-priority library manifest → low-priority library manifest.

**Duress Risk-3: Manifest-Merge.** The above order de-
We believe that the addition of the extra attributes may WebView activity with the original declaration in the high-priority library, Therefore, in cases where the low-priority library has a component and set up security-sensitive attributes. It is reasonable for ARC other libraries and even the main app. Therefore, it could be allowing the library to merge sensitive manifest attributes into even though it is not implemented in the library code, a malicious library can freely declare a manifest component similar WebView activities that can potentially be affected by breaking the security promise of the high-priority library by reporting a conflicting manifest attribute with the high-priority library, Android would raise an alarm to app developers noting that the manifest files are not mergeable. But a question that we need to ask is: for the mergeable attributes in the low-priority manifest, is it always secure to merge all of them directly into the high-priority manifest?

To answer this question, we look into an example in Figure 4. As can be seen, the high-priority library uses a WebView activity com.high_priority.WebViewActivity to load web content, e.g., from websites or web ads. This activity is designed to be app/library internal (e.g., for preventing potential WebView-based attacks), and therefore is not exported externally. Unfortunately, our study reveals that a low-priority library, although it does not own the code of WebViewActivity, can declare the same activity and add extra attributes to it. Since the extra attributes don’t conflict with the original declaration in the high-priority library, they are merged into the final manifest file automatically. We believe that the addition of the extra attributes may break the security promise of the high-priority library by lowering down the security protections. In Figure 4, by adding the android:exported and deeplink-related intent filters, the activity is not internal to the app anymore. Instead, it turns into a “backdoor” that can be invoked remotely via malicious website or untrusted apps by issuing deeplink URLs. According to our measurements, 255 libraries, including popular ones such as applovin and stripe, contain similar WebView activities that can potentially be affected by this attack. It is important to note that, in addition to lowering down security protections in place, merging extra manifest attributes can lead to other security consequences, such as denial of service by making exported components internal.

**Discussion.** The major cause of the above risk is that a malicious library can freely declare a manifest component even though it is not implemented in the library code, allowing the library to merge sensitive manifest attributes into other libraries and even the main app. Therefore, it could be reasonable for ARC to track the ownership of the component, and only authorize the component owners to declare the component and set up security-sensitive attributes.

![Image](Image 66x613 to 282x720)

**Figure 4:** Merge manifest files

It appears that Android has already taken a step towards limiting the use of security-sensitive attributes: for apps that target Android 12 and higher, an android:exported attribute needs to be explicitly specified for all components that define an intent filter [54]. This requirement seems to reduce the opportunity for attacks that exploit this specific attribute, since an alarm will be raised to app developers if the mandatory attribute is missing. However, our analysis proved that it is not the case because the requirement is checked right after all manifest files from third-party libraries are merged by ARC. In other words, attackers can still exploit the above risk with a malicious library so as to change the android:exported setting of a victim library, without violating the requirement.

Further, prior research [26] showed that the Android OS (such as PackageManagerService) can not handle the registration of duplicate manifest components when installing an app, i.e., a component can override another component with the same name. Compared to this research, Risk-3 focuses specifically on the ARC issue of merging manifest attributes from multiple libraries at app build time. Risk-3 complements prior understanding of the security of duplicate manifest components by presenting a new attack vector in the design of ARC. In particular, Risk-3 is independent of prior Android OS issues, and thus the attacks can still occur on the latest Android 13 release even after the prior OS issues were fixed.

## 4 Methodology

In Section 3, we made the first step by unveiling the Duress attacks. A comprehensive study, however, has not been conducted to understand Duress in the wild. In this section, we introduce our methodology for carrying out such a study, through which we answer three research questions.

- **Q1:** Library attack opportunities. How many libraries contain sensitive resources that Duress attacks can target?
- **Q2:** Library integration risks. How likely can a library’s sensitive resources be contaminated by another library, if they are integrated into the same app?
- **Q3:** Apps affected by Duress risks. Are there any real-world apps that may have been affected by Duress risks?

**Overview.** Figure 5 presents the overview of our methodology. In the first step, we build a library dataset and an app dataset by crawling Maven Central [23] and Google Play [50], respectively. After that, we identify the sensitive library resource types, since the actual security consequences of Duress attacks vary depending on the resource types. In the next step, we run three tasks to answer the above research questions. The first task is attack opportunity analysis (Q1), in which we...
scan the libraries in our dataset to detect sensitive resources and the associated Duress risks. The second task is integration risk analysis (Q2), in which we conduct cross-library analysis to identify if two libraries share duplicate sensitive resources. The third task is to detect apps that are potentially affected by Duress risks (Q3). For this purpose, we identify all libraries that are integrated into the apps, and check if the aforementioned integration risks are present in the apps.

### 4.1 Identifying Sensitive Resource Types

While library resources are from a variety of sources and are of different forms (Table 1), we found that they fall into two main categories: manifest resources whose resource keys are predefined by Android, and other resources specified by library developers, such as constant values (e.g., strings) and raw resource files (e.g., assets). The distinction of the categories requires us to treat them differently.

#### Manifest resources.

Our approach to identifying sensitive manifest resources (or attributes) is straightforward. We first crawl Android documents [47] to gather the complete list of manifest attributes. We then search from public sources, i.e., CVE [41] and Google Scholar [32], to check whether these attributes have been mentioned in known vulnerabilities or reported by previous security and privacy research. Specifically, we use keyword matching to find the manifest attributes in CVE descriptions. For Google Scholar, we search those attributes together with a “security” or “privacy” keyword. We review the top 10 search results (sorted by relevance) for each attribute to find its security implications. In total, we searched 122 unique manifest attributes that cover 30 different manifest elements (e.g., application, activity), and were able to identify 14 sensitive ones reported by 16 previous studies and 6 CVEs.

Examples of the sensitive manifest attributes are the android:exported attribute that leads to the exposure of library components, the android:taskAffinity attribute that can result in task hijacking, the android:allowBackup and android:usesCleartextTraffic attributes that affect backup/network behaviors of an app, etc. In Table 2, we show the full list of the attributes, along with the security and privacy implications of unauthorized modification.

#### Other developer-specified resources.

Identifying the other types of sensitive resources is challenging, given the large number of library resources that are specified by library developers in an ad-hoc manner. To address the challenge, our observation is that although the exact resources of different developers don’t match, they are often semantically connected. An example is the technical support messages, e.g., com.dji uses a message “if the problem persists, contact dji support”, and app.moneytree.link shares a similar one “an error occurred, please contact moneytree support”. Therefore, we can leverage this observation to reduce the number of library resources by clustering them using semantic similarity, and then request security experts to review the clusters to determine whether the resources are security- or privacy-sensitive and what the resource types are.

Specifically, we first randomly sampled 1,000 libraries (from our library dataset in Section 5), and collected the resource names and values in the form of text data. Then we vectorized the text data using a pre-trained embedding model all-mpnet-base-v2 [89]. After that, we applied a hierarchical clustering algorithm on the distance matrix of the embeddings to find the clusters of similar resources. Particularly, we adopted fastcluster [69] with average linkage on the cosine distance. We also used a relatively small distance threshold (i.e., 0.4), for the purpose of creating closely-connected clusters. The output of this step is 1,016 resource clusters (from a total of 12,432 resources) that contain at least three similar resources.

We asked two security researchers to independently review the 1,016 clusters and determine whether they represent sensitive resources. The inter-rater reliability measured by Cohen’s kappa for this process is 0.84 (nearly perfect agreement). Then, the two researchers resolved disagreement with on-site discussion, and categorized the clusters into different resource types according to the security and privacy implications. Through this process, we were able to identify 74 sensitive resource clusters, corresponding to 10 unique resource types (we released the resource clusters online [24]).

As shown in Table 2, the resource types include the cloud backend URLs and credentials that may lead to compromise of the backend servers and user data leakage, the privacy disclosure that concerns the privacy compliance of library developers, the technical support messages that are subject to technical support scam and potential user data leakage, and various types of configuration files that affect network security, backup rules, and file providers, etc. Note that although these resource types are far from complete, they allow us to perform an under-estimation of Duress risks in the wild (see Section 5).

<table>
<thead>
<tr>
<th>Manifest resources (attributes)</th>
<th>Security/Privacy Implications</th>
</tr>
</thead>
<tbody>
<tr>
<td>android:allowTaskReparenting</td>
<td>Hijack tasks [12, 73, 79, 99]</td>
</tr>
<tr>
<td>android:taskAffinity</td>
<td></td>
</tr>
<tr>
<td>android:allowBackup</td>
<td>Data leakage [9, 10, 65, 103]</td>
</tr>
<tr>
<td>android:fullBackupContent</td>
<td></td>
</tr>
<tr>
<td>android:debuggable</td>
<td>Attach untrusted debuggers [60]</td>
</tr>
<tr>
<td>android:priority</td>
<td>Hijack broadcast [8, 88]</td>
</tr>
<tr>
<td>android:exported</td>
<td>Export internal components [7, 11, 27, 36, 58, 102]</td>
</tr>
<tr>
<td>android:isolatedProcess</td>
<td>Disable isolation [58]</td>
</tr>
<tr>
<td>android:launchMode</td>
<td>Hijack tasks [79, 99]</td>
</tr>
<tr>
<td>android:usesCleartextTraffic</td>
<td>Permit cleartext traffic [62, 71]</td>
</tr>
<tr>
<td>android:readPermission</td>
<td>Unprotected content providers [26, 58]</td>
</tr>
<tr>
<td>android:writePermission</td>
<td>Unprotected components [26, 58]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Developer-specified resources</th>
<th>Security/Privacy Implications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backend URL</td>
<td>Data leakage [93, 98, 110, 111], Insecure URL [109]</td>
</tr>
<tr>
<td>Credential</td>
<td>Inject malicious code/content [98]</td>
</tr>
<tr>
<td>Script code</td>
<td>Inject malicious code [22, 103]: Data leakage [33]</td>
</tr>
<tr>
<td>Privacy disclosure</td>
<td>Privacy non-compliance [29, 32, 97, 109]</td>
</tr>
<tr>
<td>Technical support</td>
<td>Technical support (scams) [67, 84]</td>
</tr>
<tr>
<td>Referral message/link</td>
<td>Redirect users to phishing/malware links [16]</td>
</tr>
<tr>
<td>ML model</td>
<td>Plant ML, backdoors [16, 80]</td>
</tr>
<tr>
<td>Network security config</td>
<td>MITM [11, 75]: Permit cleartext traffic [62, 71, 75]</td>
</tr>
<tr>
<td>Auto backup rule</td>
<td>Data leakage [17], DoS [68]</td>
</tr>
<tr>
<td>File provider path</td>
<td>Data leakage and overriding [15], DoS [72]</td>
</tr>
</tbody>
</table>

Table 2: Security/Privacy sensitive resources and their implications
4.2 Library and App Analysis

**Attack opportunity analysis.** We identify attack opportunities by classifying library resources into the aforementioned sensitive resource types and Duress risks.

The way we classify manifest attributes is trivial: we parse the libraries’ manifest files with minidom, and check the presence of the 14 sensitive attributes from the files. If a sensitive attribute is in the `secured` state (e.g., `android:exported="false"`), we report an attack opportunity for Risk-2 (Manifest-Overriding) as the secure protection can be lowered down by resource deduplication (Section 3.2). Similarly, if a sensitive attribute is applicable to an element but it is absent from the manifest files, we report an opportunity for Risk-3 (Manifest-Merge): the attribute can be merged by malicious libraries (Section 3.3).

For the other app-developer-specified resources, we classify them based on their semantic closeness to the 74 labeled resource clusters (Section 4.1). Specifically, we calculate the average cosine similarity between the embeddings of a resource and that of the resources in a cluster. To decide whether the resource belongs to the sensitive resource type represented by the cluster, we need a similarity threshold. In this study, we choose the threshold based on the average pairwise similarity between all resources in the cluster. We report the resource as an attack opportunity for Risk-1 (Resource-Overriding) as long as its similarity to the cluster is higher than the threshold.

Our measurement shows that sensitive resources are present in libraries, about 18.4% libraries contain sensitive resources, which represent a large scale of opportunities that Duress attacks can target (Section 5.1).

**Integration risk analysis.** An integration risk refers to a pair of libraries that share duplicate sensitive resources, which, if integrated into the same app, can result in Duress attacks. To evaluate the integration risks, we take the input of the maps between libraries and their sensitive resources, and perform cross-library comparisons to find the duplicate sensitive resources that have the same name but different values. Using this method, we were able to report that, among all libraries that have sensitive resources, 25.7% libraries are threatened by other libraries in the wild (Section 5.1).

**Analysis of potential Duress occurrences in apps.** In this step, we identify real-world apps that are affected by the above integration risks, i.e., apps integrating any pair of libraries that have duplicate sensitive resources. For this purpose, we first need to accurately identify which libraries are used by an app. This task, i.e., third-party library detection, has been well discussed in recent years [31, 66, 106, 108]. Therefore, instead of creating our own tool, we leverage an open-source and state-of-the-art library detector – LibPecker [108], which not only achieves a good precision, but is also resistant to code obfuscations, elimination, etc. Specifically, for each app, we run LibPecker on all libraries to determine the similarities between the app and the libraries. We use a similarity threshold (0.6 as used by LibPecker) to tell which libraries are integrated by the app. Then, if any pair of libraries that have duplicate sensitive resources is detected in the app, we flag the app as potentially affected by Duress risks.

5 Findings and Analysis

**Datasets.** We collected two datasets in July 2022. The first dataset is a library dataset ($D_l$), which contains 23.691 most recent versions of AAR libraries that were crawled from Maven Central [23] – the most popular public repository hosting third-party libraries. The other dataset is an app dataset ($D_a$) that includes 156,266 apps from Google Play. To obtain this dataset, we first gathered a list of package names for 7.2M Google Play apps (from AndroidZoo [28]). Then, we randomly shuffled the list to generate a subset of apps that approximates the Google Play distribution, and used a crawler [25] to download the apps from Google Play. Until this work was done, we were able to download and analyze 156,266 apps, with 74.4% apps updated after January 1, 2021 (51.6% in 2022 and 22.8% in 2021).

5.1 Landscape

To answer the aforementioned research questions, we run our methodology on the library dataset ($D_l$) and app dataset ($D_a$). Table 3 shows the overall data for the library attack opportunities, integration risks, and potential Duress occurrences in apps, categorized by sensitive resource types.

**Q1: Attack opportunities.** In total, we detected that 4,349 (18.4%) libraries contain sensitive resources, which represents a large scale of opportunities that attackers may target with Duress attacks. Specifically, 2,063 (8.7%) libraries contain sensitive resources that can be overridden by malicious libraries with misleading content (Risk-1 Resource Overriding); 2,281 (9.6%) libraries use security-sensitive manifest attributes that can potentially be disabled by high-priority malicious libraries using node markers (Risk-2 Manifest-Overriding). In addition, 2,561 (10.8%) libraries fail to explicitly specify security-related attributes (e.g., `android:exported`, `android:priority`) in their manifest files, which creates an opportunity for attackers to lower down the security protection of manifest by merging extra and insecure manifest attributes (Risk-3 Manifest-Merge).

We found the presence of all 10 types of sensitive resources as listed in Table 2, from the 2,063 libraries affected by Risk-1 Resource-Overriding. The most popular resource is library technical support, which appears in 1,359 (5.7%) libraries. As we will elaborate in Section 5.4, exploiting such technical support, an attacker (or scammer) can claim to offer legitimate support services via fake contacts so as to launch practical attacks such as targeted phishing. Another popular resource that appears in 584 libraries is the privacy disclosures. Libraries adopt such disclosures, e.g., privacy policies or in-app disclosures [51], in an effort to become privacy compliant. Therefore, an attacker that
contaminates the disclosures can easily destroy the privacy compliance of the libraries, causing serious privacy and even legal risks [35, 39, 40, 91, 94]. Following these two types of popular resources is the file provider path. We found that 460 libraries use the file provider path (an XML configuration) to define which files can be shared with other apps via a FileProvider [14]. As such, an attacker can potentially expose the files that are not meant to be shared by contaminating the file provider paths. The other types of resources are used by fewer libraries, but they may also lead to serious security risks, which we will demonstrate in case studies in Section 5.4.

Q2: Integration risks. Our methodology takes as input the 4,349 libraries that contain sensitive resources, and determines if the resources lead to integration risks by conflicting with any other libraries in $D_1$. As shown in Table 3 (6th column), over 25.7% libraries cause risks when integrated into the same app as other libraries. Specifically, libraries using network security config are most likely (80.6%) to conflict with other libraries, and can lead to the overriding and disablement of another library’s network security policies. Similarly, 61.5% libraries that use file provider paths conflict with some other libraries in $D_1$. Another resource that can lead to high risks is the credentials: about 37.3% of libraries that hard-code credentials in their resources have duplicate credentials with other libraries. Later, we will conduct a causal analysis to show why duplicate resources are common across libraries.

Q3: Potential Duress occurrences in apps. We assess potential Duress issues that may have occurred in real apps based on our datasets, i.e., $D_1$ with 156,266 apps and $D_2$ with 23,691 unique libraries on their latest versions (both collected in July 2022). Specifically, by running LibPecker (a state-of-the-art tool [108] to detect libraries in apps), we identified that 61,452 apps use third-party libraries listed in $D_1$, with each app having an average of 5.9 libraries. We found potential Duress occurrences in at least 428 apps (denoted as a set $D_{duress\_potential}$): any app $app_{dp}$ in $D_{duress\_potential}$ includes at least two libraries denoted as $lib_1$ and $lib_2$ ($lib_1 \in D_1$ and $lib_2 \in D_2$), where $lib_1$ and $lib_2$ have resource conflicts enabling Duress. We call $(lib_1, lib_2)$ as a Duress-library pair. In $D_{duress\_potential}$ we report 123 unique Duress-library pairs including 71 unique libraries denoted as DuressLibs. We note that these apps $D_{duress\_potential}$ likely have witnessed Duress since the latest versions of the libraries have the resource conflicts, while a complete, precise forensics analysis is hard due to the challenges to accurately identify library versions in the apps [105] and recover resource conflicts that may have happened.

Specifically, the apps might not use the latest versions of the libraries (DuressLibs), but rather use the earlier versions that may or may not have Duress conflicts (the underlying tool LibPecker generally does not accurately differentiate library versions). To more precisely evaluate the results, we collect all historical versions of DuressLibs that are publicly available on Maven Central [23]. To better understand likelihood of Duress occurrences, for each app $app_{dp}$ in $D_{duress\_potential}$, we do the following: for each Duress-library pair $(lib_1, lib_2)$ of the app, we take those historical versions of $lib_1$ and $lib_2$ (denoted as two sets $V_{S_{lib1}}$ and $V_{S_{lib2}}$, respectively) published before the release date of the app (app release dates are available on Google Play), and check Duress for every historical version pair, i.e., $\{ (v_{lib1}, v_{lib2}) | \forall v_{lib1} \in V_{S_{lib1}}, v_{lib2} \in V_{S_{lib2}} \}$. Hence, if all historical version pairs of $lib_1$ and $lib_2$ feature Duress conflicts, the app $app_{dp}$ highly likely has witnessed Duress; if a certain historical version pair $(v_{lib1}, v_{lib2})$ does not enable Duress — called safe pair, the particular app still may have witnessed Duress (if the app does not use the specific library version), but with lower likelihood. To better characterize the likelihood of an app $app_{dp}$ to have witnessed Duress, denoted as $P(app_{dp})$, we consider $P(app_{dp})$ to be the ratio of non-safe pairs among all historical version pairs of the app. Based on this model and our dataset, we use a violinplot-based approach to present the result: 76.5% apps have a $P(app_{dp})$ value of at least 85.0% (Figure 6), and the average $P(app_{dp})$ of all apps in $D_{duress\_potential}$ is 80.8%. Further, we report that, from a total of 1,337 historical library versions we collected, there are 18,143 unique historical version pairs for all apps in $D_{duress\_potential}$, and 76.9% of the historical version pairs feature Duress conflicts.

We sampled ten apps with the smallest $P(app_{dp})$, and found that they use some libraries, e.g., io.hippochat:hippo and com.hipay:fullservice:hipayfullservice, of which only few recent library versions contain sensitive resources because of new feature introduction. For example, io.hippochat:hippo,

![Figure 6: Potential Duress occurrences in apps](image-url)
a marketing automation platform, introduced a feature in version 3.0.5 that allows its users to checkout via com.razorpay:checkout library. Thus, only version 3.0.5 and after contain sensitive manifest resources that modify com.razorpay:checkout-related components, i.e., changing android:exported of RzpTokenReceiver and CheckoutActivity. We want to note that the likelihood of an app being affected can be further increased by adopting new library detection methods that can distinguish between library versions (which we leave as a future implementation).

The majority of the apps are reported since the manifest files of their libraries can modify each other, either via merging into insecure attributes (168 apps), or overriding existing attributes (137 apps). In particular, we found that libraries frequently compete with other libraries in the backup-related attributes, such as android:allowBackup and android:fullBackupContent. As we will discuss in Section 5.4, even popular libraries like Vungle and Appsflyer are competing with each other when specifying the attributes, resulting in weakened backup rules that unexpectedly expose library data. In addition, aligned with our observation of integration risks, the duplicate file provider paths files and network security config are also found to affect 76 and 45 apps, respectively. We find fewer or no apps affected by other types of duplicate resources.

5.2 Causal Analysis

We are curious why duplicate sensitive resources are so common across third-party libraries. Therefore, for each resource type, we sampled 20 pairs of libraries (we collect all library pairs if there are fewer than 20), and analyzed the causes of duplicate resources. In total, we analyzed 229 library pairs. Reliance on a common library. We noticed that a large portion (34.5%) of the library pairs depend on a common library. However, the common library is often not self-contained, and it requires its consumer (i.e., app or library that depends on it) to pre-configure some settings. An example is the Google Service library [22]. A consumer that uses this library has to specify a few keys, including firebase_database_url and google_api_key in its values.xml file. There won’t be an issue if the consumer is an app, since the app’s values.xml file is integral because app modules have the highest priority in ARC. However, an issue may arise when the consumer is a library: if two libraries that consume the Google Service library are integrated into the same app, the keys in the libraries’ values.xml file will become duplicate resources, which would trigger deduplication and lead to potential attacks to one library, e.g., quota theft or authorized access to Google services. Our analysis indicates that the issue is not a corner case since third-party libraries frequently rely on common libraries to provide richer functions.

Generic resource names that are prone to name collisions. The developers of 27.1% library pairs choose to use simple and generic names (e.g., password and server_url) for sensitive resources. These names, when used by multiple libraries, can introduce Duress risks. An example is the hippoagentsdk library which leverages a confirmation string to ask for explicit consent from users, i.e., “I agree to the terms of use and privacy policies on [website]”. Our study shows that at least 188 other libraries have a confirmation string that is irrelevant to user consent. Therefore, integrating these libraries together with hippoagentsdk library will potentially override the string that asks for user consent, and lead to unattended compliance risks to library developers.

Resource names from the sample code of official documents. 25.8% library pairs have duplicate resources since they all follow the resource names in the sample code of official documents. For a long time, developers have built the notion that sample code from official documents is safe to use. Unfortunately, this practice may unexpectedly cause Duress risks to third-party libraries. Take the network security config [53] feature as an example. With this feature, developers can specify network policies, such as whether HTTP is allowed, the set of trusted CAs, and certificate pinning, in a declarative XML resource file. The sample code in the official documents [49] names this file /res/xml/network_security_config.xml. Our study shows that almost all of the libraries that use this feature (90.3%, 168/186) adopt the same file name. Thus, if two of the libraries are integrated in the same app, it’s almost certain that their network security policy would override each other. Library templates and library outsourcing. The remaining duplicate resources are caused by library templates, e.g., open-source projects, or public app/library builder services. Examples are com.jbangit.app:unimini and com.rsdx.tojoy.shop:/sdk libraries. Our analysis shows that they are both created by a builder service dcloud.io [21]. Not surprisingly, the two libraries contain a configuration file with the same name assets/data/dcloud_control.xml, which stores the unique api key (appid) provided by dcloud.io. As noted earlier, in cases that the two libraries are integrated in the same app, there will be a Duress risk that causes the deduplication and misuse of the api key.

A special case for library templates is associated with library outsourcing. Using eu.genome.android:dk-payment and com.maxpay.android:dk-payment as examples. The two libraries are provided as mobile SDKs for online payment platforms – Genome and Maxpay – that serve Lithuania and Malta customers, respectively. The platforms don’t seem to be connected to each other according to online information (e.g., official websites). However, our analysis indicates that their mobile SDKs are actually built by the same party since they share almost identical code bases, except for the replacement of several resource values, e.g., Genome specifies its @string/prod_url as https://gateway.genome.eu/api/ while Maxpay uses https://gateway.maxpay.com/api/. We suspect that the reason is that the two platforms outsourced the development
to the same third party, who reused the same template during development. Similar to using open-source and app builder templates, library outsourcing is dangerous as well since the library owners may not be aware of the presence of the template, not to mention the integration risks.

5.3 Responsible Disclosure

As we discussed in Section 3, the Duress risks are rooted in the design of ARC’s mechanism of duplicate resource mediation. Therefore, we reported all the discovered risks, including a detailed description of attack cases (with anonymized library names) to Google – the owner of ARC. Google responded to us quickly. On the one hand, they emphasized that it is the developers’ responsibility to ensure the libraries they use are secure, i.e., libraries would not introduce malicious resources to exploit ARC. On the other hand, they raised a feature request in Android Studio to allow app developers to detect and resolve conflicts. At the time of writing, we did not get further updates about this feature. We are also in the process of reporting our findings to the developers of the affected apps. For more updated information about the responses, please refer to our website [24].

5.4 Case Studies

Taking over cloud backends with duplicate credentials. Our study shows that at least 217 libraries hard-code their credentials in the resources. Exposing such credentials could lead to severe consequences, e.g., cloud backend takeover.

MistPlay [19] is a leading play-game-to-earn platform (ranked #3 in the AppsFlyer growth index for Japan and Korea) that serves 200+ mobile games and over 1M users. Gaming apps may use its mobile library, LoyaltyPlay, to integrate the play-game-to-earn feature into the app so as to increase user retention. We found that LoyaltyPlay leverages Amazon Kinesis Data Firehose and S3 to collect and process the gamers’ streaming data, such as user id, in-game search query, and click events, etc. For this purpose, the LoyaltyPlay library hard-coded the AWS credentials, including the Cognito Identity Pool ID, in a resource file/res/raw/loyaltyplay_awsconfiguration.json. An attacker may use a malicious library to plug in his own credentials by overriding the resource file (Risk-1 Resource-Overriding). This would automatically “redirect” the library to an AWS account owned by the attacker, leading to the complete takeover of the cloud backend. Interestingly, we found that this attack is technically feasible because it requires no code modifications in the LoyaltyPlay library, nor additional configurations in the cloud backends (except for activating the Kinesis service in the attacker’s account). Furthermore, the attack is also stealthy and transparent to app developers and app users as it happens entirely in the background, e.g., credentials are overridden automatically in app build process, and the backend takeover doesn’t introduce extra behaviors at runtime. More details about the attack are available at [24].

Technical support scams. As noted in Section 5.1, over a thousand libraries store technical support contacts in the resources. These contacts, if contaminated in Duress attacks by fake contacts, would enable scammers to launch practical attacks such as targeted phishing attacks.

Dolyame.ru is the first digital buy-now-pay-later service available in Russia. This service provided a mobile library called ru.tinkoff.dolyame:sdk for the sellers to integrate it into their apps. We noticed that the library embedded a supportChat string (that points to WhatsApp “https://wa.me/74997000600”) in its resource file/res/raw/config.json. When there is a payment issue, the app users can follow the supportChat to connect with technical support via WhatsApp. In case that an attacker overrides the supportContact with a fake contact, he can pretend to be the legitimate support, and perform phishing attacks to exfiltrate customer sensitive information such as customer name, address, purchase history, etc. Even worse, the attacker may encourage the customers to visit malicious websites and download an unwanted app to their devices. We want to highlight that, compared to traditional smishing or sms spams, such an attack might be more convincing as it originates from a specific seller’s app and targets the users of the app.

Opening doors for man-in-the-middle (MITM) attack. To secure network connections of apps, Google introduced the network security config feature [53] in Android 7.0. With this feature, developers can specify network security policies in a declarative XML configuration file, e.g., /res/xml/network_security_config.xml, without actual coding. A malicious library can compromise the other libraries’ policies by overriding the configuration file using Duress attacks.

HitPay is an online payment gateway for small and medium-sized businesses in Singapore. The mobile library of HitPay, i.e., com.hit-pay.terminalsdk, relies on a network security configuration for protecting the network accesses from the library. As shown in Figure 7, an important measure is to enable certificate pinning for its backend server api.hit-pay.com for the purpose of thwarting man-in-the-middle (MITM) attacks. Our study shows that a malicious library is capable of arbitrarily modifying the configurations so as to disable the certificate pinning and open a door for MITM attacks, e.g., by removing the pin set, adding a pin that contains the digest of an attacker’s public key, or marking the pin set as expired, etc.

Contaminating Android backup rules. Since Android 6.0, app users can automatically backup their apps’ data to Google Drive using the Auto Backup feature [48]. By default, this feature would upload virtually all app data, including shared preference, database, and other files in internal and external storage. At the same time, Android allows developers to customize the backup rules in an XML file and specify which files can (and cannot) be uploaded to Google Drive.

As a common practice, library developers often choose not to upload highly-confidential files so as to reduce data exposure risks. An example is Appsflyer - a popular mobile
marketing analytics platform [20]. In its backup rule file (i.e., /res/xml/appsflyer_backup_rules.xml), Appsflyer explicitly asked Android to exclude the backup of the shared preference appsflyer-data.xml, which stores the app installation data, cached device ID and appsflyer ID, in-app event statistics, etc. As we demonstrated in Section 3.2, an attacker may disable the custom rule by replacing the entire rule file using a malicious library (Risk-1: Resource-Overriding), and thus cause unexpected exposure to Appsflyer internal data.

Besides the above hypothetical attack, we found that contention for backup rules has happened and been discussed in the wild. Specifically, in order to use custom backup rules, an android:fullBackupContent attribute, together with a rule file path, needs to be added to the manifest file. In cases where multiple libraries use different backup rule files, the app developer has to specify which rule file to choose using manifest node markers (see Risk-2 Resource-Overriding in Section 3.2). In our study, we noticed that Vungle library – a mobile advertising and monetization platform – has a competition with Appsflyer. Interestingly, instead of telling developers to merge the two rule files, both Vungle and Appsflyer are instructing app developers to use their own backup rules, while ignoring the rules of the other platform [30, 95]. We observed that such a contention has happened to at least 68 apps in $D_u$.

### 5.5 Potential Mitigation

**Duress** risks are caused by the design of ARC. Therefore, a long-term mitigation may require significant effort from Google, e.g., to tune the duplicate resource mediation mechanisms, or provide app developers with capabilities to resolve duplicates by themselves. Before a long-term solution is found, it is important to have an effective interim mitigation in place.

**Design choices.** An intuitive solution for resolving duplicates is to confine the libraries in isolated domains. Such library isolation approach has been extensively discussed in prior studies [55, 59, 64, 74, 76, 81, 86], which, however, only focused on isolating untrusted library code rather than resources or are too heavy-weight to apply to all libraries. In our mitigation, we adopt a similar isolation approach but applies it to library resources at compile time. Most importantly, we make design decisions to reduce the overhead and integration frictions of the mitigation.

- **In-app resource isolation.** Prior studies, such as CompaRTist [59] and SDK Runtime [55], have explored how to separate third-party libraries into multiple processes at runtime. Unfortunately, these approaches are difficult to apply to all libraries due to the high performance overhead and the complexity of handling strongly-coupled libraries [55]. Therefore, instead of completely separating the libraries with multiple processes, we provide lightweight isolation for duplicate resources by placing them into different namespaces while keeping libraries running in the same process as the app and other libraries.

  - **Compile-time resource rewriting.** To support resource namespaces, we need to rewrite the libraries before compiling them into an app. Our choice is to implement this procedure at app compile time, right ahead of ARC. This choice results in very low integration frictions compared to prior studies that require OS modifications [74, 81]: app developers can freely enable and disable the mitigation since they are in charge of app compile process.

**Implementations.** Recall that ARC relies on several Gradle tasks (e.g., MergeResources) to mediate duplicate resources. The input to the tasks is the resources from third-party libraries, while the output is the compiled resources (i.e., deduplicated and merged resources). We implemented a Gradle plugin using Groovy to preprocess the library resources that are fed into the ARC tasks.

Take the MergeResources task for example. We add our plugin to the doFirst method of this task to ensure it runs right ahead of the task. In the plugin, we leverage the MergeResources.getInputs API to extract the input to the task, i.e., the list of libraries and their resource directories (.gradle/caches/.../[library]/res/). We then scan these resource directories to identify resource duplicates. After collecting the duplicate resources, we run our semantic-based method (Section 4.2) to determine whether these duplicate resources are sensitive or not. We add a unique prefix, i.e., the libraries’ [group id]_[artifact id], to the names of sensitive duplicate resources, so that the resources are virtually placed in isolated namespaces. At the same time, we scan the code of the library and its dependencies (.gradle/caches/.../[library]/jars/classes.jar) and change the resource references to the modified resource names. Specifically, we use dex2jar [13] and baksmali [18] to decompile and decompile the bytecode into smali code. Then, we search through the smali code to find the references to the duplicate resources (e.g., in the form of R.id.[resource_name]), and add the same prefix to the resource names.

The way we isolate manifest components is slightly different. We scan all the library manifest files (.gradle/caches/.../[library]/AndroidManifest.xml) and build an ownership map between the manifest components and their hosting libraries, i.e., libraries that implement the components in the code. Then, we remove the security-sensitive manifest attributes when a library’s manifest is found to modify the components that the library does not own. With this method, we can effectively prevent an unauthorized library from mod-
ifying the other libraries’ manifest. Optionally, we also allow app developers to define an allow-list in the build script to enable cross-library modifications from trustworthy libraries.

Note that we released our plugin to the Gradle Plugin Portal, and app developers can freely integrate it into the app build process by adding the plugin’s id in the build.gradle [56]. Further, since the plugin would not change the way library resources are accessed, we don’t expect performance penalties at runtime. However, it will inevitably cause delays to the app build process, e.g., for parsing resources, and reversing the library code (when duplicate resources exist). Our evaluation shows that this overhead is acceptable, with an average delay of ~1.46 seconds for compiling the 100 open-source projects (used in Section 3.1) on a laptop running Ubuntu 20.04 with Intel Core i7-9750H CPU@2.60GHz.

6 Discussion

Our methodology relies on resource clustering and expert knowledge to determine the types of sensitive resources, which works well for common library resources (e.g., resources in Table 2), but can not scale to highly customized library resources. We are exploring approaches that can automatically tell sensitivity of library resources without expert input. Also, we identified integration risks by analyzing the most recent version of the libraries, and detected occurrences of these libraries in real apps. We were not able to conduct a full forensics analysis of the apps, and thus reported them as potential occurrences of Duress. While the apps may use historical library versions that do not cause problems, we want to highlight that an update of the apps’ libraries to the most recent version still has a high chance of enabling Duress. The mitigation we built on top of compile-time resource namespace (Section 5.5) is limited in many cases, e.g., resource names are encrypted by library code, resource references are made from dynamic code or code of other libraries, etc. We believe a long-term mitigation may require collaborative efforts from different stakeholders, e.g., native support for library resource isolation from Android, and tighter security review of sensitive resources from the repository hosting services. In addition, although we focused on Android and its ARC, evidences show that Duress can happen to other ecosystems and build tools (e.g., Maven), which we leave for future investigation.

Recently, Android proposed a new feature – SDK Runtime [55] – for the purpose of reducing data exposure risks posed by third-party SDKs. Essentially, third-party developers can voluntarily publish their SDKs to the Android app store, and then the SDK can run in a modified and isolated environment on user devices. This feature could theoretically reduce Duress risks. However, as noted by Google [55], the feature only supports advertising SDKs because of its high overhead in isolating all SDKs, and the complexity in handling SDKs that are strongly coupled with the apps. Therefore, we believe the introduction of SDK Runtime would not significantly change the profile of Duress risks.

7 Related Work

Malicious activities of third-party libraries. Previous studies have investigated different types of malicious activities in third-party libraries, such as harvesting sensitive user data [37, 43, 70, 83, 85, 96, 109], ad fraud [42, 61, 87], and tracking users without consent [77, 92], etc. Among these studies, most close to our work is XFinder [96], which investigated how malicious libraries can harvest private data of other libraries using malicious library resources. Unlike XFinder, our work sheds light on a different and new attack opportunity that is provided by the design of the Android resource compiler, which allows malicious libraries to launch attacks only using malicious library resources. Further, besides private data leakage, our study leads to the discovery of a richer set of attacks that can mislead victim libraries and end users (e.g., backend server takeover, technical support scams), and lowering down the security protection of victim libraries (e.g., by compromising network security policies and manifest attributes).

Mitigating malicious third-party libraries. A natural mitigation to malicious third-party libraries is to confine them in isolated and unprivileged domains, which has been well discussed in the past few years on Android [55, 59, 64, 74, 81, 86, 90]. Unfortunately, most previous studies [64, 76, 81, 86, 90] are not applicable to Duress risks because they apply isolation to released apps whose resources have already been processed by ARC. Some studies propose to isolate ads library in isolated processes [55, 59] or integrate them into the Android platform [74]. These studies are potentially useful to mitigate Duress attacks, but are practically limited to apply to all third-party libraries (as we discussed in Section 6). Therefore, we propose a lightweight and compile-time mitigation to reduce Duress risks (Section 5.5).

Resource squatting attacks. Essentially, Duress attack is an instance of resource squatting attack, in which a malicious party creates the same or similar resource before a victim party does so as to achieve malicious purposes. Examples of such attacks include a privilege escalation attack that fools the Android OS update process with pre-registered privileges [101], an attack that steals other iOS app’s private data by creating duplicate keychain attributes and URL schemes, etc. [100], and an attack that routes end users to a malicious skill by creating confusing skills in the Alexa skills store [63], etc. Compared to these attacks, our study reveals a new type of vulnerability that is rooted in the design of Android’s duplicate resource mediation process, which enables malicious libraries to stealthily and effectively compromise the other libraries in the app supply chain.

8 Conclusion

In this study, we discover a new attack surface that is caused by the design of the Android duplicate resource mediation process. With this attack surface, an attacker can contaminate the app supply chain with crafted malicious libraries in order
to perform highly stealthy, and cross-library attacks, such as misleading the victim library and its users to expose sensitive data, lowering down the victim library’s security protection, etc. To systematically evaluate the impact of the new threat, we design an automatic methodology to identify the attack opportunities, integration risks, and apps affected by the risks. Our measurement results brought to light the pervasiveness of the risks in the app’s supply chain. In addition, we discuss a lightweight and compile-time mitigation to help reduce the security risks by isolating library resources.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their valuable comments and suggestions. Luyi Xing was supported in part by Indiana University’s IAS Collaborative Research Award. Yan Jia is supported in part by National Natural Science Foundation of China (No. 62102198) and China Postdoctoral Science Foundation (No. 2021M691673). This research was supported in part by Lilly Endowment, Inc., through its support for the Indiana University Pervasive Technology Institute. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of our sponsors.

References


[87] Suibin Sun, Le Yu, Xiaokuan Zhang, Minhui Xue, Ren Zhou, Haojin Zhu, Shuang Hao, and Xiaodong Lin. Understanding and detecting mobile ad fraud through the


