P-Box: Preventing Unwanted Data Flows using Permission Sandboxes on Android

Lucas Becker*
TU Darmstadt
Darmstadt, Germany
lbecker@seemoo.tu-darmstadt.de

David Breuer*
TU Darmstadt
Darmstadt, Germany
dbreuer@seemoo.tu-darmstadt.de

Matthias Hollick
TU Darmstadt
Darmstadt, Germany
mhollick@seemoo.tu-darmstadt.de

Abstract

One of the core privacy features of smartphone operating systems is a permission framework that requires explicit user consent before granting apps access to private data. Such systems are deeply integrated into Google's Android and Apple's iOS, which together account for the majority of the smartphone operating system market. While permission systems can be seen as milestones in user empowerment and privacy protection, they offer users only a binary choice: whether an app can access a specific resource or not. As soon as an app is allowed to read a resource, the operating system loses control over its further use. Most apps have Internet access and can send permission-protected data, like a user's location, over the Internet, which can harm user privacy. To solve this problem, we present an addition to current permission systems that splits apps into multiple sandboxed processes to enforce fine-grained privacy and data-flow controls on smartphones. By default, our design forces apps to process permission-protected data locally on the device, thereby eliminating the need for apps to request runtime permissions for local-only use cases. We implement a proof-of-concept based on the Android Open Source Project code base. We showcase our framework's practicability by adapting multiple app use cases to our system, benchmarking its computational overhead, and discussing the implications for platform operators, developers, and users.

Keywords

Android, AOSP, privacy, sandbox, permission system

1 Introduction

Smartphones provide programmatic access to a multitude of private user and sensor data. To prevent third-party apps from accessing unwanted information, smartphone operating system (OS) providers have introduced and refined so-called *permission systems*.

Android's and iOS' permission systems allow users to exercise control over the private information they want to share with each installed app [5, 22]. Permissions include, for example, access to a device's location, user contacts, calendar, camera, or microphone [14]. However, users' control ends exactly there. After granting permissions to an app, it can use the permission-protected resources in arbitrary ways, e.g., send them over the Internet or share them with advertising networks, until the user revokes the permissions manually. Thus, preventing malicious apps from misusing user data with Android's and iOS' current permission systems is impossible.

One recent example of misuse is the accumulation of data through advertising software development kits (SDKs) and the sale of the gathered data to various companies and government agencies worldwide, allowing fine-grained tracking of user devices [98]. Additionally, related work has shown that apps tend to request more permissions than they need [38], and app developers use SDKs that require additional permissions without fully understanding their internals [37, 69]. Recent research shows that apps use covert channels on Android to pass data to non-privileged apps [88], further amplifying this problem.

More than 98% of Android apps request the `INTERNET` permission according to a study by Kollnig et al. [71], and all iOS apps are granted Internet access without any permission. As soon as an app has Internet access, it can send any data it accesses, including all permission-protected resources, to arbitrary web locations. As the most common permissions on Android and iOS include location access, camera or photo library access, account information, and calendar access [71], an app user's privacy is constantly at risk.

This unsafe status quo leads to our following research questions:

RQ1: Can mobile operating systems fundamentally prevent undisclosed leakage of permission-protected data?

RQ2: Can a restrictive permission system support real-world usage scenarios?

RQ3: What are the performance trade-offs for such a system?

In this paper, we present *P-Box*, a new permission paradigm for Android that addresses these questions, resolves underlying privacy challenges, and enhances user control over private data.

Our design is based on the core principle that apps are not allowed to access any permission-protected resources by default.

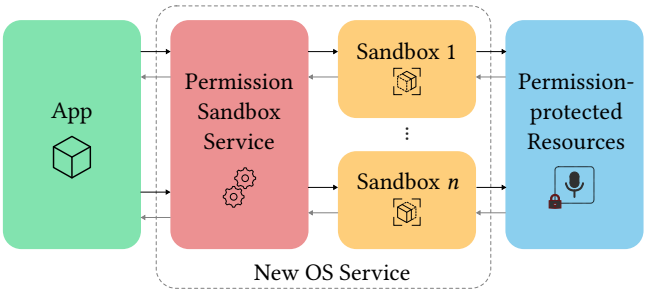


Figure 1: Architectural overview of P-Box: We introduce a new OS service, orchestrating permission sandboxes for apps that request access to permission-protected resources.

*Both authors contributed equally to this research.

Proceedings on Privacy Enhancing Technologies 2026(4), 232–247

https://doi.org/10.56553/popets-2026-0118

Instead, if an app wants to access a permission-protected resource, it must start sandboxed processes that handle said access. These *permission sandboxes* are fully managed by an intermediate OS service with restricted input and output channels. The return value of such a sandbox is not the permission-protected data itself, but a *handle* only pointing to the corresponding OS-managed resource. The app itself cannot access a handle's value (subsequently called *dereferencing* the handle). Nevertheless, it can use this handle as an input to another permission sandbox to perform arbitrary computations on it. In this case, P-Box ensures that no private data can leave a permission sandbox; therefore, there is no need for an app to request runtime permissions, leading to fewer user-facing prompts. This allows even untrusted third-party apps to operate on private data while guaranteeing its confidentiality. An architectural overview is shown in Figure 1.

Certain permissions comprise not only data sources, but also data sinks, e.g., the `INTERNET` permission. To pass data from sources to sinks, we allow apps to request *flow permissions*. These require explicit user consent via runtime prompts and allow a handle of a specific source to be dereferenced in a sandbox with access to a specific data sink. This approach allows fine-grained control over data flows, minimizes privacy leakage, empowers users, and still allows legitimate apps to process sensitive data. For example, a photo editing app can access a user's photo library without permission to share it online. Simultaneously, the same app might still gather crash information and send it to its analytics server.

Furthermore, sandboxes can render permission-protected data to the screen while the app's main process cannot access this information. This provides a transparent and convenient way to present private information to users without compromising their privacy.

We implement a proof-of-concept of P-Box in Android. In contrast to Apple's iOS, Android is the perfect candidate for prototyping new OS features due to its open-source code base, published as Android Open Source Project (AOSP) [8]. We base our implementation on AOSP 15 (API level 35). Additionally, we provide an example backward-compatibility layer to facilitate migrating standard Android apps to P-Box. We showcase our system using a real-world app, a real-world map SDK running in our sandbox, and an example app demonstrating fine-grained permission flows. To estimate the performance and memory overhead of our system, we also run benchmarks based on permission-usage patterns from five popular apps and find that the upper-bound overhead is within the capabilities of modern smartphones. We are confident that AOSP or iOS system developers can draw inspiration from our proposal to enhance their platforms' privacy and users' data autonomy.

In summary, our paper claims the following main contributions:

- *P-Box*: A new privacy-enhancing permission framework for Android.
- An AOSP 15-based open-source implementation of P-Box, available on https://github.com/seemoo-lab/p-box.
- Adaptation of a real-world open-source app and an SDK to our framework, coupled with a benchmark based on five popular apps.

2 Current Smartphone Permission Systems

Android and iOS share the majority of the mobile OS market [95]. For this reason, we compare their current permission systems, which act as the de facto standards for permission handling for almost all smartphone users worldwide. We describe the main similarities and differences of Android 16 and iOS 26.1 in the following sections and discuss their relevant shortcomings, which are later addressed by P-Box. As Android is open source and the foundation of our prototype, we describe it in more detail than iOS.

2.1 Android Permission System

All software running on Android is subject to strict *application sandboxing* [48]. This means that apps can communicate only through well-defined interfaces and cannot write to the filesystem at will. If apps want to access data outside their own sandbox, they must obtain the required permission to access the requested resource. Internally, a lot of the isolation guarantees of apps are implemented using SELinux [101]. SELinux is a mandatory access control system, originally built for the Linux Kernel and designed to enforce fine-grained policies.

Android distinguishes between *install* (or normal), *signature*, and *runtime* (or dangerous) permissions [5]. Install permissions are granted to apps during installation without user interaction. They have to be declared in an app's manifest. By doing this, they can be presented to consumers prior to installation on app stores, e.g., Google Play or the open-source platform F-Droid. Signature permissions are treated similarly but are granted only to an app that is signed with the same key as the app that declared the permission. This mechanism is used to ensure that certain permissions are granted only to system apps signed with the same key as the Android version, or to implement custom permissions that allow data sharing between multiple apps from the same vendor. Additionally, *privileged* permissions can be applied to apps that are pre-installed in a distinct directory on Android installation. Only OS distributors can do this [99].

Runtime permissions protect private resources of extraordinary value, e.g., the phone's camera, photo library, or location. These permissions must be declared in an app's manifest, too, but are granted only with explicit user consent. Apps must request them during runtime when the app first attempts to access the protected resource. In this case, a prompt is shown to the user, explaining the permission. If a user consents to the request, the permission is granted. The permission remains granted until the user manually revokes it or, if the user does not interact with the app for multiple months, the OS automatically resets it as part of the app hibernation mechanism [9]. If the user rejects the request, the app is not granted the runtime permission and can proceed to offer reduced functionalities without sensitive data access. The runtime permissions for location, camera, and microphone are subject to more fine-grained controls, offering the options "all the time" (background use — only available for the location permission), "allow only while using the app", "ask every time", and "don't allow" instead [51].

In addition to these options, Android also offers toggles for the camera and microphone [13], as well as the location subsystem [54]. When off, these toggles globally prevent access to the corresponding data, regardless of an app's permission state. They are useful for

quickly locking down sensitive data sources, but restrict all apps on the device, reducing usability.

Permission-protected resources are provided by system processes and cannot be accessed directly from non-privileged app processes. For each resource access, an app must send its request to the respective system service that provides the data. The system service that receives the request then checks if the calling app has been granted the necessary permission. If the permission check passes, the service returns the requested resource. Otherwise, a security exception is thrown [99].

Access to a phone's camera and microphone is not only runtime permission-protected, but also considered important enough to notify Android users whenever an app uses such resources. The UI elements reflecting this are called *privacy indicators* [102]. If an app accesses either of the two resources, a green icon appears in the status bar. Users can get more information on the triggering app by clicking on the indicator. The icon remains visible for 15 seconds after the last access. Privacy indicators are enforced by Android's *App-Ops*, which track all apps' accesses to runtime permission-protected application programming interfaces (APIs) [100].

2.2 Google's SDK Runtime

Typically, Android apps and their corresponding third-party SDKs are statically linked into a single Android Package (APK) and run in the same process. However, as part of Google's *privacy sandbox*, a self-proclaimed effort to strengthen user privacy across all their services, they introduced a feature called *SDK runtime* on Android [49]. Here, the operating system provides separate sandboxes for an app and its SDKs. In Google's approach, app and SDK developers must adjust their code base to be compatible with the SDK runtime. Additionally, SDKs must be installed separately from the app that requires them.

While a first version of this technology, targeting only ad-related SDKs, is already implemented in AOSP, it will be deprecated and removed in future Android releases. According to a statement by Google's privacy sandbox team, this is due to "ecosystem feedback about their expected value and in light of their low levels of adoption" [18]. It is targeted just at SDKs that must be specifically built as "runtime-enabled" SDKs, while our implementation allows developers to execute arbitrary parts of their app or any third-party SDK in sandboxes. Google only offers a small set of permissions within the SDK runtime, while our system supports any set of permissions in a sandbox and enforces fine-grained permissions flows, which must be communicated and consented to by the user.

2.3 iOS Permission System

Apple's iOS permission system works conceptually similarly to Android's scheme. The core of it is the individual sandboxing of app processes and, by this, strict confinement of access to the phone's resources [22, 75]. To grant apps access to private resources, iOS uses two mechanisms: *Entitlements* and the *Transparency, Consent, and Control (TCC)* framework. Entitlements are granted at an app's installation while TCC checks are performed during runtime and require explicit user consent, similar to Android's runtime permissions [23, 24]. iOS shows similar privacy indicators as Android [21].

We assume that our proposed framework can also be implemented on iOS, given the resemblance between Apple's and Google's permission checks. However, since iOS is only partially open-source and altering it requires jailbreaking, extensive reverse engineering efforts, and binary patching, we refrain from modifying it. Instead, by implementing our new permission system in AOSP, we fundamentally prove its feasibility independent of the platform.

2.4 Shortcomings of Current Systems

Android's and iOS' permission systems put trust in third-party developers. While apps must ask users for permissions to access specific user data, it is opaque to the users how their data is used. The majority of apps (98.7% according to a study from 2022 [71]) are granted the `INTERNET` permission, potentially allowing them to transmit all data accessed by an app to arbitrary servers.

In recent years, Apple and Google introduced privacy labels in their app stores, where developers have to declare, in addition to their privacy policies, how user data is handled [25, 47]. While privacy policies are often hard to parse by non-lawyers, the easier-to-understand privacy labels lack details, are still misunderstood, and are often wrong [67, 70]. Furthermore, the OS does not enforce privacy labels, but only permissions. The OS trusts the app to handle user data in accordance with the developer's promises.

Android and iOS permissions are granted per app and make the protected resources available to all program parts. As it is common development practice to use third-party SDKs to conveniently add features to an app without implementing everything from scratch, this practice leads to those SDKs inheriting all of the app's permissions and, therefore, access to the app's private user data. As many Android apps request more permissions than necessary [38], the included SDKs can access the same data. Journalists have exposed ad brokers selling their data to government agencies worldwide, as this data is precise enough to track individuals in their everyday lives or while traveling across borders [98]. This emphasizes the tremendous risk smartphone users are facing when using apps that include any number of SDKs. Furthermore, research has shown that developers often misconfigure SDKs, do not fully understand their internals, do not test their functionality, or that SDKs do not have privacy-compliant configuration options at all, leading to privacy risks for users [37, 69, 71, 89].

In another work, researchers discovered multiple Android apps that share permission-protected information through covert channels with apps that do not hold the required permissions [88].

User awareness and understanding of app permissions is another problematic field. Research found that most users do not correctly assess the current permissions of their apps [85], and they tend not to fully understand the effects of permissions [39].

3 Threat Model

In our threat model, we assume an app developer who ostensibly requests permission-protected data for a specific use case but misuses it, e.g., for user profiling or tracking. An equivalent situation might arise due to a benign developer including malicious third-party SDKs. These SDKs might be written by untrustworthy developers and can use all data they acquire for their own undeclared purposes. As the SDK owns the same permissions as the app, which

are required for legitimate use cases, the SDK might abuse these to access the same data for illicit uses.

To counteract such attackers, we reduce the trust placed into third-party apps and their developers. We treat apps as non-user and non-system-controlled entities that cannot be trusted. Apps should only gain access to permission-protected data under very specific circumstances, which have to be explicitly declared. All flows of permission-protected data, called permission flows, are contained and controlled by the OS. Since we build on Android's existing permission system, we have to rely on its soundness. Hence, we exclude permission bypasses within Android itself from our scope.

In the case of an adversarial developer, our system prevents privacy leakage as long as legitimate use cases can be fully executed on-device. As soon as data leaves the device, e.g., to the Internet, our system cannot prevent a malicious app developer from abusing the data. Still, due to the explicit prompt *Allow this app to send [RESOURCE_NAME] to the Internet?*, for each permission flow, we aim to inform users and clarify that user data leaves the device. The underlying concept of flow permissions is explained in Section 4.5.

In the case of a benign developer, our system can prevent privacy leakage to untrustworthy SDKs by running the untrustworthy code in a permission sandbox, decoupling it from the rest of the app.

As our threat model includes the execution of arbitrary code within permission sandboxes, measurable side effects that allow the construction of covert channels to transfer data from the sandbox to the parent app are a warranted concern [107]. Due to their enormous scope, we exclude covert channels from our threat model with regard to our prototype's implementation. However, we discuss approaches on how to deal with them in Section 7.1.

4 A New Permission Paradigm

In contrast to status quo permission systems as described in Section 2, we propose the following architecture with our enhanced threat model in mind (see Section 3). We shift as much trust as possible from third-party developers to the OS. By this, we enable users to achieve data sovereignty, allowing them to precisely decide which data they want to share with app companies and which data should remain on their devices. Additionally, our system encourages developers to avoid over-asking for permissions that might not be necessary. Our framework is based on the following three key design lemmas:

- **L1:** General opaqueness of private data to apps: Third-party apps cannot access permission-protected data by default. They can never obtain the permissions required to directly interact with the relevant system services.
- **L2:** App code within a *permission sandbox* can access permission-protected data if the app holds the correct permission.
- **L3:** Permission-protected data can only leave P-Box with explicit user consent and is restricted to precise data flows.

From these lemmas, new challenges arise, leading to the further design of P-Box. In the following sections, we introduce and discuss these challenges and explain how our system solves them.

4.1 Permission Sandboxes

For apps wanting to operate on permission-protected data, we provide a sandboxing mechanism that allows these operations to be executed without leaking information to the app itself. *Permission sandboxes* are isolated processes controlled by the OS that cannot communicate with the calling app besides well-defined input and output channels, as required by L3. The result of a sandboxed calculation is returned to the calling app as a handle, an opaque reference to the actual data. When an app starts a permission sandbox, it passes the requested permissions to the OS and must declare its sandbox usage in its manifest using `SANDBOXED` permissions, described in the next section.

4.2 Sandboxed Permissions

App developers must declare which permissions each sandbox is allowed to use. To differentiate these new permissions from standard OS permissions, we introduce a new type: `SANDBOXED` permissions. All former permissions are now only available within a permission sandbox in accordance with L1[1].

`SANDBOXED` permissions represent permissions that an app is allowed to request when spawning a permission sandbox (L2). They are labeled as install-time permissions, meaning apps do not have to explicitly request them at runtime. Each `SANDBOXED` permission corresponds to a standard Android permission, which is granted to a permission sandbox by our system service. For example, the `SANDBOXED_READ_CONTACTS` permission allows an app to spawn permission sandboxes that are granted the permission `READ_CONTACTS`, and by this, access contact data within a sandbox. This approach reduces the number of runtime prompts users receive during app usage while keeping their permission-protected data safe, as it cannot leave the sandbox.

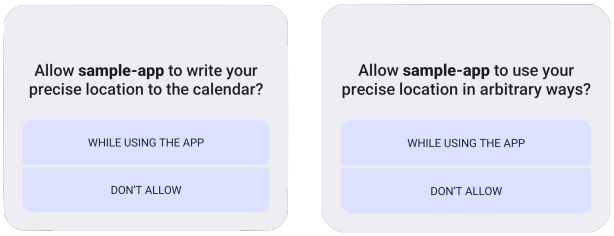
4.3 Handles

The computational result of a permission sandbox is returned as a *handle* to the calling app. This handle contains only a reference to the actual value, which is managed by the OS. The app only knows that it got a value, but cannot access the data, thus ensuring L2. If it wants to use the data behind the handle, it can pass it to a matching sandbox for further computations. Additionally, the OS could also provide APIs for commonly used functionality that directly consume a handle, given that these APIs adhere to our permission model.

4.4 Tag System

As handles can be used as inputs to other sandboxes, our system must store the permissions used to generate each handle, ensuring only permitted data flows from one sandbox to another (L3). To solve this, handles and sandboxes are labeled with the permission sets they correspond to, which we refer to as their *tags*. The handle containing the sandbox's computation result is tagged according to the sandbox's requested permissions. If another sandbox wants to use this handle, it must own all permissions contained in the handle's tags. The OS manages and enforces all tags as a trusted entity. Because developers decide which permissions to request for sandboxes they spawn, they are in full control of the sandboxes' and corresponding handles' tags, avoiding undesired overtagging.

[1]We still grant the `INTERNET` permission to apps for performance and convenience reasons. As it only represents a sink regarding permission-protected data, this does not affect our security model.



(a) Consent request to write the user's location to the calendar.

(b) Consent request to pass location data to the app process.

Figure 2: New consent prompts for permission flows. Users can now grant individual data flows to distinct data sinks.

4.5 Flow Permissions

Apps might need to send permission-protected data to external servers or write to external storage as part of valid use cases. For this, we first distinguish between permission-protected data acting as a *data source* that emits data, and permissions resembling a *data sink* that consumes data. Examples for sources are location or network state, and sinks are, e.g., Internet or storage locations shared with other apps. Then, we introduce a second new permission type: `PS_FLOW` permissions.

These permissions resemble exact data flows from permission-protected sources to sinks. They must be requested during runtime and explicitly approved by the user (L3), as shown in Figure 2(a). If a flow permission is not granted, our system prevents the respective data flow. For example, without the permission `PS_FLOW_LOCATION_FINE_TO_WRITE_CALENDAR`, attempts to spawn individual sandboxes that have both the `WRITE_CALENDAR` and the `ACCESS_FINE_LOCATION` permission, but also dereference operations of `ACCESS_FINE_LOCATION`-tagged handles within a `WRITE_CALENDAR`-tagged sandbox are blocked by the system. If multiple different permissions are requested when spawning a sink sandbox, the app has to ask for permission flows for every corresponding data flow, all of which the user has to accept.

After data is passed into a sink, it is not controlled by our system anymore, and all privacy guarantees are voided. Especially, the Internet sink can be potentially misused by passing permission-protected data to a remote server and fetching it again to access the data in plaintext. We do not consider this an additional privacy threat, as the data must have exited our system's boundaries with explicit user permission before, as data sink access is enforced by Android's permission system.

Keeping flow permissions at a minimum results in fewer permission requests to users. As apps that request many permissions are perceived as untrustworthy by users [97], our approach benefits benign app developers.

To enable legacy apps and an easier transition to our paradigm for developers, we introduce the *declassify sink*. This sink is gated by special `DECLASSIFY` flow permissions that allow an app to dereference a handle of a corresponding tag in its main process. This resembles the same functionality of a status quo Android or iOS permission, with the added benefit of control over which permission-protected data an app is allowed to declassify. The corresponding request is shown in Figure 2(b).

4.6 Private UI Drawing

Drawing permission-protected data to the screen is a common flow that our system supports in a privacy-preserving manner. Usually, an app's main thread handles all UI related functionalities and keeps track of the complete UI state, including all information drawn to the screen. In our system, Lemma 1 does not allow an app to dereference a permission handle outside of a sandbox by default. Hence, it is not possible for an app to directly draw permission-protected data to the screen. Instead, we provide capabilities for permission sandboxes to render sub-views within the app's user interface (UI). We explain two approaches for implementing such capabilities in Android in Section 5.3.

4.7 Backward Compatibility

To allow existing apps to continue working within our system, we propose the use of compatibility shims, which transparently spawn required permission sandboxes. These shims can be used together with the declassify sink introduced in Section 4.5. If, for example, an app not yet migrated to our architecture would want to call some SDK function requiring a permission, the compatibility library would then provide the same function as before, but internally spawn a sandbox that calls this function. Then, the compatibility function dereferences the result and returns it to the app. Hence, the only change required of the app is to request all needed flow permissions to the declassify sink.

Since granting these permissions to apps reduces the privacy guarantees of P-Box, they must be protected by informative permission prompts according to L3, informing users of their implications. A potential strategy for platform operators could be to allow these permissions for a transition period, and afterward prohibit all apps that declare them by the respective app store's policy.

5 Android Implementation

We implement our new permission paradigm as part of AOSP. It is open source and, therefore, easier to modify than iOS. While the concept itself is applicable to iOS too, we decided against implementing it for Apple's OS, as only parts of it are openly available. Additionally, being in control of the complete code base, we can ensure that our implementation holds all the security and privacy guarantees that we aim for. We base our implementation on AOSP 15 (API level 35) and test it on the *Cuttlefish* emulator [12] and a *Google Pixel 9* smartphone.

5.1 System Service

At the core of our architecture lies our system service, called *PsHandleManager*. It spawns and manages permission sandboxes, handles the creation and dereferencing of handles, bridges Intents originating from sandboxes, and enables private UI drawing. This system service performs all permission checks involved in these operations and is the primary new trusted computing base that we introduce. When spawning sandboxes, the service first checks that the app holds the appropriate `SANDBOXED` permissions. If that is the case, it registers a new permission sandbox with the activity and package managers and grants the corresponding permissions for each `SANDBOXED` permission. The service then prepares the execution environment and starts the sandbox process. If the sandbox exits

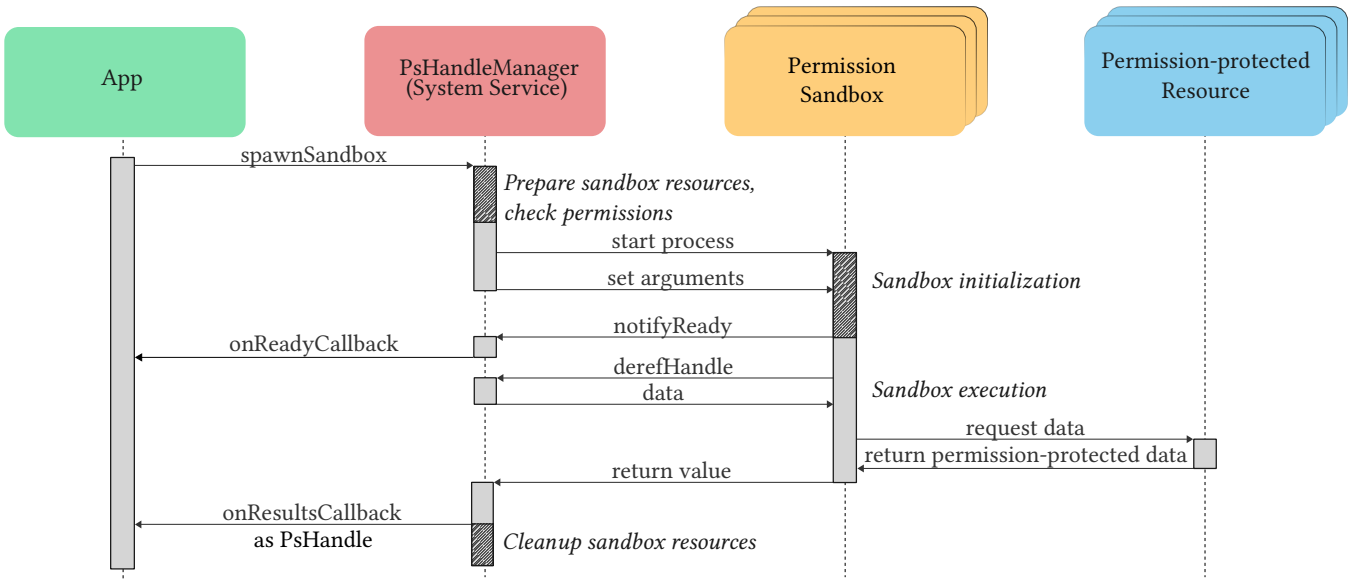


Figure 3: Sequence diagram of the complete permission sandbox life cycle.

or the app terminates, the PsHandleManager stops potentially left-over processes and unregisters the sandbox from the activity and package managers.

Second, it creates and manages handles representing the results of sandboxed computations. Internally, handles are stored within a map, linking randomly generated UUIDs to corresponding data. The system service wraps the UUID and associated tags with our *PsHandle* class and passes it to the requesting process. On handle initialization, the service stores the app's user ID it created the handle for and checks that ID when a client tries to dereference it. This check enforces that only sandboxes belonging to the same app or the app itself can dereference a handle. Additionally, this allows transferring handles between the sandboxes of an app, as they can still dereference them if they hold the appropriate permissions.

Third, the PsHandleManager bridges Android Intents originating from a permission sandbox, e.g., to set an alarm from a sandbox using the `SANDBOXED_SET_ALARM` permission. Our architecture needs to limit Intents originating from sandboxes, as they represent an otherwise potentially leaking inter-process communication (IPC) channel. For this reason, our service only permits allowlisted system components as targets for Intents originating from the sandbox. The rationale behind this design decision is that system components are assumed to be trusted, and allowlisting permits the exclusion of components that could be abused as a covert channel, e.g., if they allow setting data that other apps can then retrieve. While it is possible in Android to register third-party apps as Intent handlers, this requires explicit user interaction and therefore does not constitute privacy leakage under our threat model.

5.2 Sandbox Thread

We use Android's *Zygote* mechanism [2] to spawn new sandbox processes. We differentiate between two different sandbox types: *Lightweight* and *Full* sandboxes.

Lightweight Sandboxes. These sandboxes are intended for short-lived operations that should impose minimal runtime overhead. They do not support rendering to the UI nor loading an APK's embedded resources. The code for lightweight sandboxes is included in the app's assets directory during build and stored in the DEX format. When loading lightweight sandboxes, our system service copies the DEX file containing the sandbox class from the app's directory into a system-server-controlled directory.

Full Sandboxes. In contrast, full sandboxes can render to the UI when being passed a corresponding token (see Section 5.3) and access an APK's embedded resources. This implies that they can inflate view layouts defined by XML resource files. When spawning a full sandbox, our system service first copies the app's APK into an isolated directory and prepares a storage directory for the sandbox.

For both sandbox types, our system service forks a new process from Zygote and loads our custom *SandboxThread* class wrapping Android's *ActivityThread*. When spawning this new process, the system service also applies a custom SELinux context designed to isolate our sandboxes and sets appropriate user and group IDs. This SELinux context is based on Android's *untrusted_app* context but minimizes the number of privileges granted to our sandboxes. For lightweight sandboxes, the wrapper class copies the DEX file into memory. In contrast, for full sandboxes, it sets up a class loader to load the given APK and sets up the required infrastructure for rendering. Then it loads the requested classes and prepares arguments, such as the app context structure. Finally, it invokes the main method of the class specified in the initial spawn request using reflection. After the loaded sandbox code returns, it invokes the PsHandleManager service's callback to return the result to the calling app. Here, our service wraps the result in a PsHandle, ensuring this occurs from a privileged context and cannot be bypassed from within a permission sandbox. Figure 3 gives an overview of the complete permission sandbox spawning process.

5.3 Private UI Drawing in Android

We propose two options for enabling privacy-preserving rendering from sandboxes in Android: In our first approach, apps can use our system service to draw permission-protected data to the screen. An app that wants to show the corresponding data for a handle on-screen requests that the system service draws the data to the UI at a location chosen by the app. The data is then drawn as a *system overlay*, which is not accessible from the app's code.

The second option uses primitives already provided by Android. By creating a *SurfacePackage* and passing it back to the app, sandboxes can manipulate the associated *SurfaceControlViewHost* to render to the SurfacePackage [16]. The SurfacePackage encloses the view hierarchy as defined by the sandbox while preventing direct access to its contents. Instead, it is a reference that an app can use to render the contained UI elements, allowing the sandbox that holds the SurfaceControlViewHost to update the view hierarchy continuously. Our SandboxThread automatically initializes the SurfaceControlViewHost if the app passes an *InputTransferToken* as an argument when spawning a sandbox. The InputTransferToken is necessary so that the app can forward input events to the sandbox. Since this is an unidirectional channel into the sandbox, it does not leak private data from the sandbox. After the app obtains this package, it can then construct a *SurfaceView* to display the encapsulated view to the user. When doing so, Android's privileged UI compositor *SurfaceFlinger* [7] manages the rendering process. By design, the parent app cannot access the contents of the embedded view because the underlying window buffers are not exposed.

Both options can technically be bypassed by taking screenshots of the app's UI or by using Android's accessibility features. But, as these options rely on explicit user interaction, we do not view them as privacy leakage in the context of our threat model.

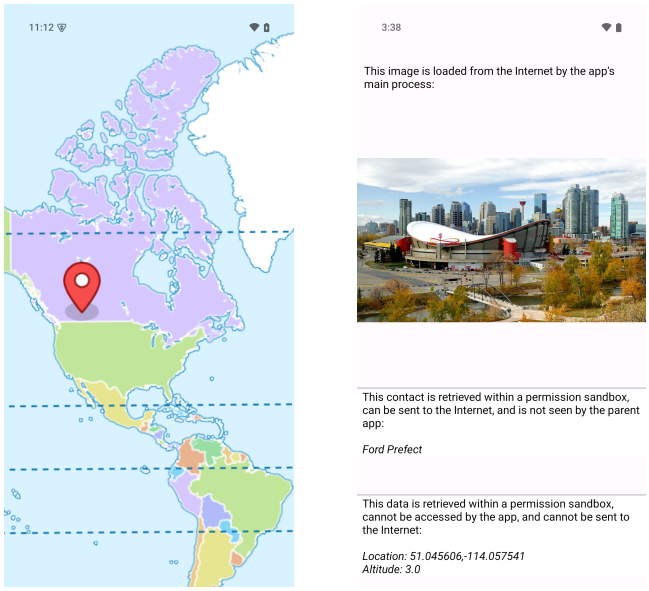
6 Evaluation

To answer **RQ2**, we evaluate the practicality of our architecture twofold: First, we build multiple example apps to highlight how typical data flows can be realized within the constraints of our permission system. Second, we migrate an existing open-source audio recorder app to our architecture. Lastly, we address **RQ3** by performing benchmarks to estimate the runtime and memory overhead of our implementation.

6.1 Example Apps

We implement two example apps to evaluate the practicality of our architecture from an app developer's perspective and showcase the capabilities of our prototype.

Map App. This app showcases private UI drawing and how former runtime permissions can be downgraded to install-time permissions without any privacy leakage. The app spawns one permission sandbox containing the *MapLibre Native* SDK [77] (see Figure 4(a)). This sandbox is granted the `ACCESS_FINE_LOCATION` permission, requests location data, and renders a map to the screen without the parent app being able to access the view's content. The parent app declares only the `SANDBOXED_ACCESS_FINE_LOCATION` permission, which is needed to spawn the respective permission sandbox.



(a) Sample app using a permission sandbox, running the *MapLibre Native* SDK.

(b) Sample app combining two permission sandboxes with private UI rendering.

Figure 4: Example apps running on a Google Pixel 9 device within our custom AOSP-based framework.

Touch input is passed through from the parent app, so users can still interact with the sandboxed map view.

This example only uses the permission-protected location data to render a map view within a permission sandbox. It does not need to request any runtime permission from the user, as the location data never leaves the permission sandbox. By this, it cannot leak a user's location to any sinks, even if the main app had the `INTERNET` permission to download additional map data.

Map apps usually download map data dynamically from the Internet based on the current map view or user location. To prevent fine-grained location leakage, this is not possible in this example. Nevertheless, multiple map apps allow users to manually download specific broader areas [26, 84]. By doing this from within the main app, privacy leakage is minimized to broader map areas, and no precise location data ever leaves the permission sandbox. We added the option to pass files to a sandbox to enable this process. The system service copies them to the sandbox's working directory but prevents the main app from accessing it.

Further, this example highlights that SDKs can be used within our sandboxes, including any native libraries they may require.

Flow Permission Demo. The second example app demonstrates how flow permissions enable fine-grained control over an app's data-sharing capabilities (see Figure 4(b)). The app uses two permission sandboxes: one accesses the device's current location, while the other reads contacts and sends them to the Internet. Both sandboxes draw privately to two distinct parts of the device's screen. The app itself also uses the Internet permission (which we allow

an app to request for convenience and performance reasons, see Section 4.2). The important principles our architecture enforces are:

- The app itself can neither access location nor contacts.
- Although the app declares the `SANDBOXED_INTERNET` permission, it cannot spawn a sandbox with both the `INTERNET` permission and the permission to access the device's location, as it lacks the corresponding flow permission.
- The app's only runtime permission that requires explicit user consent is `PS_FLOW_READ_CONTACTS_TO_INTERNET`. With this, the app can spawn a permission sandbox to both read and upload contact data to the Internet. It is important to note that it still cannot send the location data anywhere.

This approach allows the same app to include a non-trustworthy advertisement SDK that can never access private data, which is only handled within the boundaries of the two permission sandboxes.

6.2 Migrating an Existing App

We migrate an existing open-source app to our architecture to explore the challenges developers face. For that, we use the *Fossify Voice Recorder* app (50K+ downloads on Google Play) [53, 82] to highlight how apps can handle complex permission flows (audio recording, file system operations, and Internet access) in practice. This app is written in Kotlin and uses a dedicated Android service for recording. Its main UI elements, providing recording, playback, and recycle-bin functionality, are constructed by embedding view fragments into the view hierarchy. This modular design pattern is well-suited for adaptation to our architecture, as we can directly replace the initialization of view fragments with SurfaceViews backed by dedicated sandboxes, as shown in Figure 5. The sandbox wraps the existing view classes and adds 140 lines, while modifications to each view fragment and its corresponding adapter class account

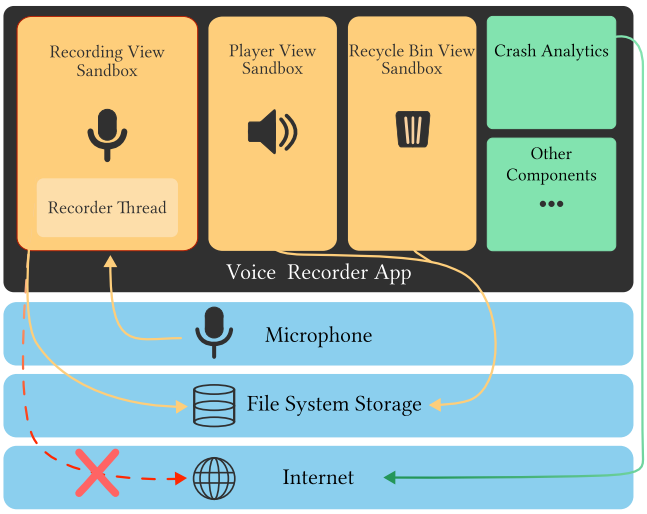


Figure 5: Permission flows in the migrated Fossify Voice Recorder app. Permission-protected data is only accessed within permission sandboxes (shown in light orange). Even though the app has Internet access for crash analytics, it cannot transmit recorded audio to the Internet.

for the majority of the changes, numbering 493 additions and 549 deletions. To adapt the recording service, we convert it to a thread-based implementation and move it into the recording fragment sandbox. This change requires 255 additional lines, but replaces the old service of roughly the same size. Using a thread eliminates the need to perform IPC within our sandbox, which the original implementation uses to control the recording state. Additionally, by moving it into the UI-controlling sandbox, we can directly request the required permission to record audio within this sandbox. Finally, we need to adapt the shared configuration object that was initially wrapped in the context and explicitly pass it to the sandboxes.

The app relies on Android's *Storage Access Framework (SAF)* [4] to obtain file access permissions. Hereby, the app can request a system-provided consent dialog for accessing a user-chosen directory. These dialogues must be initiated from an Android *Activity* and cannot be used directly within a sandbox in our current prototype implementation. Instead, our prototype supports an SAF inheritance mechanism, where a sandbox can be spawned with the `SAF_INHERITANCE` permission. This permission enables the sandbox to use its parent app's SAF grants. Because the file system is a potential sink for privacy-sensitive data, the SAF inheritance permission is marked as such. It requires the corresponding flow permissions when obtained simultaneously with other permissions. Using our SAF inheritance mechanism, migrating file accesses within the sandboxes is as simple as granting them the `SAF_INHERITANCE` permission. In total, we modified 18 files, resulting in 955 insertions and 1014 deletions, to adapt the app to P-Box.

Our version of the app provides the benefit that users can be assured that the recorded audio can only be written to the file system. Since the app does not have any sandbox with file system access and another sink, such as Internet access, it is not possible for the app to leak this recording. In this particular instance, the app is already designed to be privacy-friendly, as reflected by its lack of the `INTERNET` permission by default. However, these guarantees also hold if it were updated to add Internet access for analytics or advertising purposes.

An important observation regarding the ease of transitioning to our proposed permission system is that it highly depends on the app in question. Apps with a modular architecture will be easier to port than those that rely on tightly coupled components, especially if those components involve many different permissions. Additionally, apps that are privacy-friendly by design and require only strictly necessary permissions are easier to adapt, since they need fewer sandboxes to manage access to permission-protected data.

6.3 Backward Compatibility

We implement a minimal proof-of-concept of a backward compatibility library wrapper to test the practicality of the approach introduced in Section 4.7. We choose to use the open-source PocketMaps app [59] as an example for a real-world app. After some minor tweaks to make the build system compatible with the Android API level corresponding to our prototype, we only had to include our compatibility library, redirect corresponding calls from the original SDK class to our compatibility class, and request the required declassify permission. These changes only required trivial modifications of around a dozen lines of code, as the architecture of

the app does not have to be adapted. Our example wrapper replaces two different functions from the *LocationManager* API. It transparently calls the original methods within a sandbox and dereferences the result. To increase the convenience further, the library wrapper could wrap an app's context to return an instance of the compatibility class instead of the original SDK class, similar to how Android's *Appcompat* layer [17] facilitates backward compatibility.

6.4 Prototype Test Suite

We have written a test suite, based on Android *instrumentation tests* [3], to evaluate the privacy guarantees and functionality of our proof-of-concept reproducibly. It consists of 32 distinct test cases that validate permission access within sandboxes, flow permissions, and private UI drawing. Further, it tests for unintended communication channels by examining whether sandboxes can communicate with other processes via IPC mechanisms, such as shared memory, Binder, files, or sockets. We run all tests after each development step on our emulator and, finally, on an actual Google Pixel 9 smartphone to verify our final artifact.

6.5 Performance Evaluation

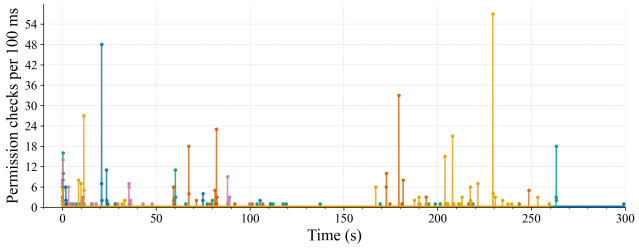
To answer **RQ3**, we first benchmark the resource consumption of individual sandboxes and, second, extract permission usage patterns from popular Android apps to estimate P-Box's real-world performance overhead. All benchmarks run on our custom AOSP build on a *Google Pixel 9* smartphone.

Benchmarks. We use Android Studio's *microbenchmarks* [1] to measure the runtime overhead of individual sandboxes. For this, we spawn a permission sandbox that returns only its input value as a handle. We measure the time it takes from calling *spawnSandbox* until the *onResultsCallback* is fired in the calling app and the memory consumption of the sandbox process. We repeat each measurement 50 times. Full sandboxes introduce an average runtime overhead of $\sim 112ms$ ($SD = 14ms$) per spawned sandbox. Using our lightweight sandbox option, this is reduced to $\sim 73ms$ ($SD = 11ms$). It is important to note that this runtime overhead originates in the sandbox initialization process and IPC transactions to and from the permission sandbox. There is no noticeable UI latency while interacting with sandboxes that use private UI drawing. In terms of memory consumption, full sandboxes allocate $\sim 120MB$ while lightweight sandboxes use $\sim 105MB$. As current smartphone manufacturers increase the built-in memory of their devices to enable on-device machine learning features [55], we do not regard memory consumption as a major issue. Permission sandboxes are mostly short-lived and do not continuously occupy a device's memory.

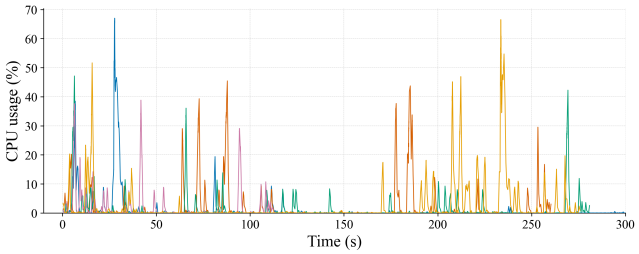
Real Apps. To estimate the performance impact of P-Box on real-world apps, we use the following methodology. We select five popular apps from Google Play [50] across different categories: CapCut (a video editor) [30], Claude (an LLM-based chat assistant) [52], Duolingo (a language learning app) [35], QR Code Reader (a utility app) [57], and Signal (an instant messenger) [93]. We run each app on an Android smartphone for up to 5 minutes, performing typical user behavior, but focusing on app interactions that trigger permission checks, such as contact, camera, or location access. We log each time the app under test causes a successful permission check. Since we grant all permissions requested by the app, our logging includes all permissions the app actively requests, but filters out unrelated background permission checks performed by the OS. Hence, the number of permission checks serves as an upper bound for the number of permission-protected API accesses performed by the app. We use these permission logs as input to a custom benchmarking app that runs on P-Box and spawns a new permission sandbox whenever the tested actual app has triggered a permission check. We record system traces during the benchmarks.

Figure 6(a) shows how often each of the five apps caused a successful permission check, divided into 100ms bins. These values reach a maximum of 56 checks in one case, but on average, only around 4.2 checks are performed for non-empty bins, with an average of 128 checks total for the whole trace duration.

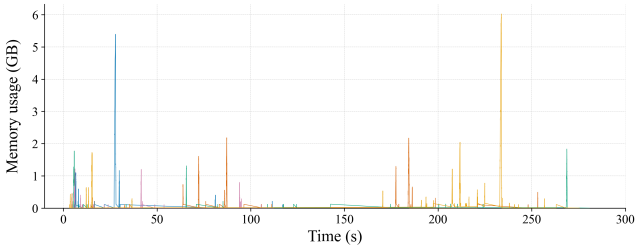
When executing a sandbox for each permission check of the corresponding real-world app, the overhead introduced by P-Box



(a) Permission checks triggered by real-world apps during the first five minutes of usage, split into 100ms bins.



(b) Combined CPU usage of the sandboxes and the system server process, imitating the permission behavior of the real-world apps on P-Box (scheduled events per 500 ms).



(c) Combined memory usage of the sandboxes, imitating the permission behavior of real-world apps (continuous data).

Figure 6: CPU and memory usage while imitating the permission-access behavior on P-Box of the following five real-world apps: ● CapCut, ● Claude, ● Duolingo, ● QR Code Reader, and ● Signal.

results in the CPU and memory usage shown in Figures 6(b) and 6(c). The majority of the time, the CPU utilization is below 20%, and the memory footprint is below 500 MB, peaking at 68% CPU usage and up to 6 GB of memory. Though these values are high for Android apps, modern smartphones have sufficient resources to handle this overhead, e.g., a Google Pixel 9 has 12GB of memory [56].

We want to emphasize that the number of permission requests derived from the real-world apps is a worst-case scenario. Our benchmarks show bursts of checks for the same permission within a short time span. These bursts probably either belong to a single API call, which often performs multiple checks from different call sites due to implementation details, or originate from the same functionality of the app. In practice, we expect that many of the corresponding permission requests can be performed in the same sandbox, hence drastically reducing the performance overhead in our system compared to this upper bound.

While performance is not the primary focus of our proof-of-concept, we see optimization potential by reducing the preloaded classes in the initial Zygote process and by tweaking the Android runtime for our purposes.

7 Discussion

To answer **RQ1**, we discuss the security and privacy implications of P-Box, its potential impact on stakeholders, and its use in light of cross-platform frameworks in the following paragraphs.

7.1 Security and Privacy Considerations

Fundamentally, the security and privacy of our architecture rely on the existing guarantees provided by Android. As we make only `SANDBOXED` permissions available to apps, Android's permission system prevents apps from obtaining any permission-protected data, assuming the absence of permission-bypassing vulnerabilities. Hence, apps are by design unable to access any permission-protected data. Instead, permissions to access private data can only be used from within a permission sandbox. This security boundary is enforced by our system service, which grants requested permissions only if the app holds the corresponding `SANDBOXED` permission. Here, our service also considers combinations of permissions: If the app tries to spawn a sandbox with multiple permissions, all applicable permission flows are checked against the app's currently granted flow permissions.

As a consequence, the only place where permission-protected data can originate within our architecture is within sandboxes. Additionally, by checking the declared `SANDBOXED` permissions, the service prevents unexpected permissions from being used within a sandbox, thus still guaranteeing inspectability by examining the app's manifest as a single source of truth.

Sandbox Communication. Without obtaining `PS_FLOW` permissions, permission sandboxes have only a single capability for returning data to the "outside world", which is through our system service. To enforce this, our SELinux policy prohibits IPC channels between a sandbox and other apps or sandboxes. This includes Android's Binder communication [11], Unix sockets, file system access, and shared memory devices. The creation of local networking sockets is restricted by the `INTERNET` permission. However, if a sandbox has the `INTERNET` permission and wants to communicate with another entity that also possesses the `INTERNET` permission, then that is trivially possible by bridging everything over the Internet, even without using local sockets. In this case, the sandbox requires a flow permission to simultaneously access the Internet and other permission-protected data, such as a device's location. Hence, allowing the `INTERNET` permission does not break our requirement that sandboxes cannot communicate with each other.

As our service is the only available back channel towards the app, it is sufficient to wrap the results of sandboxes in a handle. By this, an app can only obtain the opaque handle to the result, which the service manages internally. The only way to obtain the value associated with this handle is to query this service. Hence, it is sufficient for the service to check whether the current caller, which is either the app or one of its sandboxes, has the required permission to dereference the handle. For sandboxes, this means they must hold the corresponding permission, whereas apps require the `DECLASSIFY` version of the related permission. Because flow permissions are checked at sandbox spawn, apps can only spawn sandboxes with permissions within the bounds of their granted flow permissions. The `DECLASSIFY` permission is primarily intended for backward compatibility (see Section 4.7), since it comes with severe privacy implications. Once the app obtains the value of a handle, that data is outside our threat model, and the app might perform arbitrary operations with it, as is the current behavior of Android.

Covert Channels. Our security model prohibits any information flows from a sandbox to its parent app, besides the opaque handle a sandbox might return. However, covert channels threaten this security principle. The obvious, design-inherent covert channel our architecture exposes is a *covert termination channel* [96]. Since code running within sandboxes is controlled by the untrusted app, a sandbox might delay its termination depending on the value of permission-protected data. Geofences that stop their execution only when a user enters a specific area would be one instance of such a termination channel. Unfortunately, recent work by Wei et al. [107] has shown that techniques often used to address similar covert channels, such as predictive mitigation and fuzzy clocks, are not fail-safe. We argue, though, that this covert channel is not different from other available covert channels inherent to aspects such as CPU caches [90], system load [78], hardware sensors [29], or dynamic display refresh rate switching [34].

When assuming the presence of an arbitrary unmitigated covert channel, the security guarantees of our architecture degrade slightly. Since we convert runtime permissions to install-time permissions for use in sandboxes, runtime-permission-protected data could be exfiltrated by exploiting a covert channel. This attack would then bypass the runtime permission prompts in Android's current system. However, apps still have to declare these permissions in their manifest to access this data at all. This guarantee still holds under a threat model including covert channels. The only difference is that in this threat model, a malicious app does not need to obtain user consent explicitly. To some extent, this scenario resembles the well-known existing issue of colluding apps [88]. If two colluding apps are installed on the same device, they can exchange the sensitive data they have access to, enabling data flows that are impossible in isolation. For example, app A, with access to all phone contacts but no `INTERNET` permission, could use IPC to transfer this data to app

B, which holds the `INTERNET` permission. App *B* can then upload the received sensitive data to the Internet. Assuming that app *A* has a valid reason to request contact access, this attack effectively bypasses the contact-permission prompt for app *B*. Hence, this attack results in an outcome similar to that of our architecture under the assumption of exploitable covert channels.

7.2 Potential Impact on Stakeholders

As our proposed framework introduces deep modifications to a smartphone's OS, we discuss the resulting changes for the three main stakeholders of mobile OSs, *users*, *developers*, and *platform operators*, individually.

Users. The main focus of our framework is to enhance smartphone users' privacy. Therefore, we count user-facing changes as the most important. First of all, we enhance user privacy by preventing apps from accessing permission-protected resources in their main process by default. If an app wants to pass private data to a sink, it must disclose the precise data flow and request the OS-enforced flow permission from the user via a runtime prompt. This is designed to improve the transparency for users and increase their trust in their mobile devices.

Second, if apps use permission-protected data only within permission sandboxes without any flows, they do not have to request explicit user consent, as the app cannot retrieve a value from such a sandbox. Still, the app can render its computation result to the screen. Due to the private UI drawing mechanisms (see Section 4.6), it is still impossible for the app to retrieve this data from the screen.

P-Box only shows permission requests for occasions when apps request to pass private data to non-sandboxed locations, namely via flow permissions. Apps that process private data mostly locally have to show fewer user-facing permission requests, overall resulting in fewer permission requests from apps to users while enhancing user privacy by limiting an app's options for handling their private data. By focusing only on these potentially dangerous data flows, we assume that this highlights the importance of those user-facing permission prompts. Related work has shown that smartphone users tend to misunderstand permissions [28, 33]. With our precise descriptions of flow permissions and the general reduction of permission prompts for privacy-friendly apps, we aim to reduce user confusion about the capabilities of app permissions.

Developers. To comply with our permission system, app developers have to modify their apps. In Section 6.2, we outline possible approaches for this process and discuss how the expected effort to migrate apps to our system depends heavily on the app's architecture. We also describe beneficial architectural design decisions for an app that ease the transition to our system. Additionally, we introduce an example backward-compatibility layer to simplify the migration process for app developers (Sections 4.7 and 6.3). While this effort, at first sight, might raise the development costs for an app, it might also result in more revenue for the following reason:

For now, users have had to trust app developers' privacy promises. While they disclose their data-handling practices in privacy policies and app store privacy labels, these are not enforced by the OS. With our system, these can be further enforced and can strengthen users' trust in developers who build privacy-friendly apps. Since each requested flow permission results in a user-facing prompt, we hope to nudge developers to minimize their flow permissions in their apps. This results in fewer user-facing prompts, potentially increasing an app's usability and popularity among users.

Platform Operators. Our framework relies on OS changes. Therefore, it can only be implemented by smartphone platform operators, most prominently Google in Android's case and Apple for iOS. These companies can enforce new programming paradigms on developers because they have sole architectural control over their platforms. We envision such a drastic change with a grace period that allows developers to migrate their apps piece by piece to the new system, using our backward-compatibility APIs. There are multiple past examples of Google or Apple introducing breaking changes to their platforms, e.g., the introduction of runtime permissions [6, 20], the enforcement of transport encryption for network access [10, 22], Apple's migration of iOS to the arm64 platform [19], and Android's change in how apps can access external storage [15]. By this, these platforms have proven they have the power to introduce our proposed framework to their platforms. Especially for a company like Apple, which emphasizes its focus on user privacy, our system could reinforce user trust in the company and thus provide a market advantage over competitors. On the other hand, if one company introduces such a system, the other platform might feel the need to match the competitor's privacy guarantees.

Theoretically, it is possible for smaller Android distributions, such as *GrapheneOS* [58], to enforce our system. Unfortunately, this would break their compatibility with standard Android apps.

7.3 Cross-Platform Apps

Cross-platform apps, such as Flutter [43], React Native [81], and Cordova [103], are popular alternatives to writing native Android or iOS apps [94]. Their basic premise is that developers need to write their app logic only once, while the respective frameworks mostly abstract platform differences. Different strategies have been explored for writing such cross-platform apps. For example, Flutter comes with its own virtual machine to run app code written in the Dart language [41], while React Native uses JavaScript to communicate with their native rendering system [79].

Since all of these frameworks have their own architectures, it is infeasible for us to cover each in detail. Hence, we focus on a high-level outline of how they are compatible with P-Box. In a first step, existing apps could use a compatibility shim as described in Section 4.7. In the long term, cross-platform frameworks would need to adapt to our permission model to avoid deterring their app's users by over-asking for permissions. Frameworks wanting to implement our sandboxing paradigm face two separate challenges.

First, they require a way to render a *SurfaceView* or a derivative UI element to display sensitive data from within a sandbox. Depending on their rendering architecture, this can be trivial if they support embedding Android's native views, as is the case for Flutter [42] and React Native [80]. For Cordova, there is no officially documented way to achieve the same, as their rendering is based on *WebViews*. However, one option is to manually add an Android view and position it at the desired position on top of Cordova's WebView.

The second challenge is to support the compartmentalization of permission-related code into sandboxes. As cross-platform frameworks typically use a programming language different from the platform's native language, such as Dart or JavaScript, simply spawning our sandboxes is not directly possible. We envision two different solutions to enable such use cases: the simplest option is to bootstrap the respective framework's engine within a permission sandbox. This is supported by our prototype, as native library loading is already implemented. However, it is unclear how much overhead this process would entail, especially when aiming for short-lived, ephemeral sandboxes. A second approach would be to extend the API used for spawning sandboxes to support native processes directly, thereby removing the JVM startup overhead. We leave the exact design and evaluation of such an approach to future work.

8 Limitations and Future Work

Our paper's scope is the overall design of a privacy-enhanced permission system for mobile OSs, as well as the technical challenges of an Android implementation and the migration of existing apps. As our framework introduces user-facing changes, future work should evaluate how smartphone users perceive our design and if it reduces potential confusion around smartphone permissions.

Google Play and Apple's App Store require developers to declare privacy nutrition labels that state how they handle user data [25, 47]. With our fine-grained control over permission-protected data flows, it is possible to partially enforce these privacy labels, e.g., which data is sent to the Internet and which resides on the device.

One unsolved problem is how to best implement data-dependent resource retrieval in a privacy-conserving manner. For example, an app might want to display the opening hours of a nearby institution. To do so, it might need to perform an Internet query to retrieve suitable institutions based on the user's location. In our prototype, this would require the user to agree to at least the flow of coarse-grained location data to the Internet. A malicious app can then abuse this flow to obtain the user's location for arbitrary use. In this specific case, the relevant aspect is the information about nearby institutions rather than the upload of location data itself. The private information retrieval (PIR) research field [72] addresses this exact problem statement. However, it remains to be investigated whether PIR can be implemented as an OS API to enable data retrieval using handle-protected data as part of the index.

While our proof-of-concept is fully functional, we did not test it with all available Android permissions or all possible combinations of permission flows, as each additional permission introduces additional engineering challenges in making its respective protected resources available within a permission sandbox. The next step to bring this concept to a production-ready state in Android would include testing and implementing all remaining permissions and permission flows. Still, each resource made available in a permission sandbox needs to be carefully evaluated to determine whether it introduces new privacy leaks. For example, allowing arbitrary file access within a sandbox introduces new attack vectors for ransomware, even if the file contents themselves remain private.

As described in Section 6.5, the P-Box prototype introduces a high memory overhead. For a production-ready version, we recommend implementing a specialized version of Android's Zygote mechanism for sandboxes to reduce their memory footprint by loading only strictly necessary resources. Furthermore, CPU overhead can be reduced by preloading sandboxes in the background.

Another interesting question is to analyze the technical feasibility of our system on Apple's iOS. While Apple's iOS permission system is conceptually similar to Android (see Section 2.3), it is mostly closed-source. An implementation outside Apple's own capabilities would rely heavily on reverse-engineering, jailbreaking, and binary patching of existing iOS functionality. Therefore, we consider this out of scope for this paper.

9 Related Work

We identified multiple works aiming to improve Android's permission system. The following publications focus on solving the problem of information disclosure within apps by employing *software compartmentalization*, examining *permission flows*, introducing *library permissions*, conducting *taint analysis*, using *more fine-grained permissions*, or analyzing *cross-library data harvesting*.

Software Compartmentalization. The compartmentalization of software is a well-studied field [74]. A program is divided into distinct parts that are run with varying privilege levels in separate processes. Lefeuvre et al. have written a recent overview of different approaches, which we recommend to the interested reader [74]. In the following, we focus on Android-related publications.

The conceptually closest work to our proposal is written by Fernandes et al. [40]. They have designed a system that separates an application's execution into two parts: One that can access confidential data and compute on it, and one that contains the application's main code. Similar to our approach, data is only visible as a handle to an application and can only be processed in sandboxes. While their system is designed for the IoT domain and is implemented as a user-space app, we implement our system as part of AOSP, incorporating Android's security guarantees and permission system, and focus on arbitrary Android apps. This enforces our paradigm at the system level.

Further work is focusing on the separation of application parts operating on sensitive and non-sensitive data. Khatiwala et al. introduce the concept of data sandboxing for C programs, where applications are split into parts based on the sensitivity of the data they handle [68]. Similar works on Android focus on the division of SDK code and app code [92, 109] and program parts handling camera and Internet access [63]. Herbster et al. propose a concept where apps can request a special state that allows for access to sensitive resources but prohibits Internet access [61]. Apps can only revert to their normal state through a restart, which reduces the usability in comparison to our approach.

Qiu et al. propose the instrumentation and modification of apps to inspect and control dynamically loaded libraries by SDKs [86]. Lee et al. automatically separate an app's execution into two processes with different sets of permissions [73]. They focus on the mitigation of security vulnerabilities in native code, allow communication between the two processes, and do not primarily pay attention to user privacy in case of malicious application parts.

Huang et al. compartmentalize by exchanging calls into SDKs from apps with calls into separated processes containing the formerly called SDKs [65]. Wang et al. designed a developer tool that

facilitates the separation of app and third-party code into isolated sandboxes [104]. Nevertheless, it is not enforced by the user or operating system, but relies on the developer's indulgence. Another work proposes a user-level solution to separate data and permission accesses between apps and their advertisement SDKs [112]. This is achieved by hooking function calls to system services.

The previously discussed publications focus mainly on smartphone or app instrumentation, pre-processing, or user-level solutions, and thus only provide patches while the underlying system remains unmodified. In contrast, our framework is part of the OS, enforces privacy guarantees with a new permission paradigm, and solves the problem fundamentally.

Permission Flows. The concept of permission flows and resulting flow permissions, which allow specific data flows between components, is investigated in the following papers. Fragkaki et al. examined Android's security model in 2012 and identified missing control over data flows that handle permission-protected data [44]. They propose a system that blocks certain sets of permissions for an app if that app is forbidden from transferring data across these permission domains. Holavanalli et al. introduce the term flow permissions [62]. In their work, they provide a methodology for statically analyzing flows of permission-protected data on Android and present the resulting flow permissions to users at installation time. A similar approach, focusing on privacy leakage via IPC, including runtime analysis of apps and more fine-grained permissions, is proposed by Zhang et al. [111]. Chen and Zhu define permission flows as a "justification problem" [32]. Only data flows that trigger a UI state change are classified as justified and allowed.

Library Permissions. Zhan et al. leverage the call stack to inspect the calling library on permission checks and enforce permissions per library [91, 108]. Similar works suggest instrumenting apps at runtime and distinguishing between permission requests from SDKs and the main app [31, 45, 46, 106, 110]. While these approaches block access of non-trusted libraries to permission-protected resources, apps still run in one process, program parts can exchange data, and permission flows are neither tracked nor enforced.

Taint Analysis. Multiple works track permission-protected data flows in Android apps using taint analysis [27, 36]. While approaches based on this technique can identify privacy leakages in arbitrary Android apps, they do not fundamentally solve the problem by enforcing strict user-controlled flow permissions.

More Fine-grained Permissions. Huang et al. propose an improved version of Android's accessibility permission. They sandbox all parts of an app touching accessibility APIs to prevent the leakage of private data [64]. Rasthofer et al. designed a system to control data flows across multiple Android apps, where global policies can be applied to more fine-grained permissions. Apps are instrumented in a pre-processing step, and their system runs on an unmodified Android version [87]. Other works focus on the general improvement of Android's permission system by introducing more granular permissions [66] or by adding the ability to block an app's individual API accesses that are permission-protected [83].

Cross-library Data Harvesting. In the emerging area of *cross-library data harvesting* mitigations [105], He et al. enhance third-party libraries by modifying the SDK's bytecode [60]. Lu et al. claim to solve the problem by separating the SDK's execution from the main process, inspired by Google's SDK runtime [76]. Nevertheless, their code is based purely on modifications by app developers, runs entirely in user space, and therefore provides no additional privacy guarantees for users.

Summary. None of these ideas solves the problem of non-essential permission flows holistically. Either they are based on static or dynamic analysis and modification of apps, or they work entirely in the user space. While the former is error-prone and may break an app's functionality, the latter does not provide all the security guarantees of Android and is excluded from our trust model (see Section 3). In contrast, we provide an extension to AOSP and an associated app development paradigm. By doing this, we give developers full control over their app, as we do not instrument the app at installation time or during execution. At the same time, we empower the user through the OS to enforce limitations on the app's use of permissions.

Additionally, related work is mostly focused on the separation of apps and their SDKs or advertisement libraries. We instead propose a holistic approach limiting the abuse of arbitrary permission-protected resources within apps regardless of being accessed out of SDKs or the main app.

10 Conclusions

In our paper, we propose *P-Box*, a new permission system for Android. With our open-source implementation, we demonstrate the feasibility of our system and the practical enforcement of fine-grained app permissions and data flows. We provide multiple app examples using P-Box, including sandboxed SDK usage and the migration of a real-world app. We discuss the implications of P-Box for stakeholders in the mobile app ecosystem, namely platform providers, developers, and users. Overall, we aim to inspire Android and iOS OS developers to strengthen their platforms' privacy guarantees, increase user protection, and data usage transparency.

Ethical Considerations

Our research did not involve any human subjects or private data in any form and, therefore, did not require approval from our institutional review board. We aim to improve smartphone users' privacy while ensuring that developers can continue to efficiently implement apps and support arbitrary app use cases, as discussed in Section 7. Overall, we identify no negative ethical implications for any of the stakeholders.

Availability

Our source code and dataset are publicly available on https://github.com/seemoo-lab/p-box.

Acknowledgments

This research work was supported by the National Research Center for Applied Cybersecurity ATHENE.

References

[1] Android. 2024. Microbenchmark | App Quality | Android Developers. Retrieved 2026-02-17 from https://developer.android.com/topic/performance/benchmarking/microbenchmark-overview

[2] Android. 2025. About the Zygote Processes. Retrieved 2026-02-23 from https://source.android.com/docs/core/runtime/zygote

[3] Android. 2025. Build Instrumented Tests | Test Your App on Android. Retrieved 2026-02-24 from https://developer.android.com/training/testing/instrumented-tests

[4] Android. 2025. Open Files Using the Storage Access Framework | App Data and Files. Retrieved 2026-02-25 from https://developer.android.com/guide/topics/providers/document-provider

[5] Android. 2025. Permissions on Android | Privacy. Retrieved 2025-11-18 from https://developer.android.com/guide/topics/permissions/overview

[6] Android. 2025. Runtime Permissions. Retrieved 2026-02-24 from https://source.android.com/docs/core/permissions/runtime_perms

[7] Android. 2025. SurfaceFlinger and WindowManager | Android Open Source Project. Retrieved 2026-02-19 from https://source.android.com/docs/core/graphics/surfaceflinger-windowmanager

[8] Android. 2026. Android Open Source Project. Retrieved 2026-02-13 from https://source.android.com/

[9] Android. 2026. App Hibernation. Retrieved 2026-04-16 from https://developer.android.com/topic/performance/app-hibernation

[10] Android. 2026. Behavior Changes: Apps Targeting API Level 28+ | Android Developers. Retrieved 2026-02-24 from https://developer.android.com/about/versions/pie/android-9.0-changes-28

[11] Android. 2026. Binder | API Reference. Retrieved 2026-02-23 from https://developer.android.com/reference/android/os/Binder

[12] Android. 2026. Cuttlefish Virtual Android Devices. Retrieved 2026-05-12 from https://source.android.com/docs/devices/cuttlefish

[13] Android. 2026. Explain Access to More Sensitive Information | Privacy | Android Developers. Retrieved 2026-04-20 from https://developer.android.com/training/permissions/explaining-access

[14] Android. 2026. Manifest.Permission | API Reference | Android Developers. Retrieved 2026-02-23 from https://developer.android.com/reference/android/Manifest.permission

[15] Android. 2026. Storage Updates in Android 11. Retrieved 2026-02-24 from https://developer.android.com/about/versions/11/privacy/storage

[16] Android. 2026. SurfaceControlViewHost | API Reference. Retrieved 2026-02-19 from https://developer.android.com/reference/android/view/SurfaceControlViewHost

[17] Android Team. 2025. Appcompat | Jetpack. Retrieved 2026-02-20 from https://developer.android.com/jetpack/androidx/releases/appcompat

[18] Anthony Chavez. 2025. Update on Plans for Privacy Sandbox Technologies. Retrieved 2025-11-20 from https://privacysandbox.com/news/update-on-plans-for-privacy-sandbox-technologies/

[19] Apple. 2023. About iOS 11 Updates. Retrieved 2026-02-24 from https://support.apple.com/en-us/102991

[20] Apple. 2023. About iOS 6. Retrieved 2026-02-24 from https://support.apple.com/en-us/102995

[21] Apple. 2023. About the Orange and Green Indicators in Your iPhone Status Bar. Retrieved 2026-02-13 from https://support.apple.com/en-us/108331

[22] Apple. 2024. *Apple Platform Security*. Technical Report. Apple.

[23] Apple. 2025. Entitlements. Retrieved 2025-11-20 from https://developer.apple.com/documentation/bundleresources/entitlements

[24] Apple. 2025. Requesting Access to Protected Resources. Retrieved 2025-11-20 from https://developer.apple.com/documentation/uikit/requesting-access-to-protected-resources

[25] Apple. 2026. App Privacy Details on the App Store. Retrieved 2026-02-13 from https://developer.apple.com/app-store/app-privacy-details/

[26] Apple. 2026. Download offline maps on iPhone. Retrieved 2026-05-11 from https://support.apple.com/guide/iphone/download-offline-maps-iphcfb5f5bc6/ios

[27] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[28] Bingyu Shen and Lili Wei and Chengcheng Xiang and Yudong Wu and Mingyao Shen and Yuanyuan Zhou and Xinxin Jin. 2021. Can Systems Explain Permissions Better? Understanding Users' Misperceptions under Smartphone Runtime Permission Model. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association.

[29] Kenneth Block, Sashank Narain, and Guevara Noubir. 2017. An Autonomic and Permissionless Android Covert Channel. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '17)*. Association for Computing Machinery, New York, NY, USA, 184–194. https://doi.org/10.1145/3098243.3098250

[30] CapCut. 2026. CapCut AI-video-editor. Retrieved 2026-05-11 from https://www.capcut.com/nl-nl/

[31] Xin Chen, Heqing Huang, Sencun Zhu, Qing Li, and Quanlong Guan. 2017. SweetDroid: Toward a Context-Sensitive Privacy Policy Enforcement Framework for Android OS. In *Proceedings of the 2017 on Workshop on Privacy in the Electronic Society*. ACM, Dallas Texas USA, 75–86. https://doi.org/10.1145/3139550.3139552

[32] Xin Chen and Sencun Zhu. 2015. DroidJust: Automated Functionality-Aware Privacy Leakage Analysis for Android Applications. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. ACM, New York New York, 1–12. https://doi.org/10.1145/2766498.2766507

[33] Manila Devaraja and Sameer Patil. 2025. Understanding User Prioritization and Comprehension of Smartphone Permissions. *Proc. ACM Hum.-Comput. Interact.* 9, 5 (Sept. 2025), MHCI035:1–MHCI035:46. https://doi.org/10.1145/3743739

[34] Gaofeng Dong, Jason Wu, Julian De Gortari Briseno, Akash Deep Singh, Justin Feng, Ankur Sarker, Nader Sehatbakhsh, and Mani Srivastava. 2024. RefreshChannels: Exploiting Dynamic Refresh Rate Switching for Mobile Device Attacks. In *Proceedings of the 22nd Annual International Conference on Mobile Systems, Applications and Services (MOBISYS '24)*. Association for Computing Machinery, New York, NY, USA, 359–371. https://doi.org/10.1145/3643832.3661864

[35] Duolingo. 2026. Duolingo - Learn a Language for Free. Retrieved 2026-05-11 from https://www.duolingo.com/

[36] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association, Vancouver, BC.

[37] Feal, Álvaro, Julien Gamba, Juan Tapiador, Primal Wijesekera, Joel Reardon, Serge Egelman, and Narseo Vallina-Rodriguez. 2021. Don't Accept Candy from Strangers: An Analysis of Third-Party Mobile SDKs. *Data Protection and Privacy* 13 (2021), 1–28. https://doi.org/10.5040/9781509941780.ch-001

[38] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*. ACM, Chicago Illinois USA, 627–638. https://doi.org/10.1145/2046707.2046779

[39] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. 2012. Android Permissions: User Attention, Comprehension, and Behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security (SOUPS '12)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/2335356.2335360

[40] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. 2016. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. (2016).

[41] Flutter Team. 2025. Flutter Architectural Overview. Retrieved 2026-02-05 from https://docs.flutter.dev/resources/resources/architectural-overview.md

[42] Flutter Team. 2025. Hosting Native Android Views in Your Flutter App with Platform Views. Retrieved 2026-02-05 from https://docs.flutter.dev/platform-integration/android/platform-integration/android/platform-views.md

[43] Flutter Team. 2026. Flutter - Build Apps for Any Screen. Retrieved 2026-02-05 from https://flutter.dev/

[44] Elli Fragkaki, Lujo Bauer, Limin Jia, and David Swasey. 2012. Modeling and Enhancing Android's Permission System. In *Computer Security – ESORICS 2012*, Sara Foresti, Moti Yung, and Fabio Martinelli (Eds.). Springer, Berlin, Heidelberg, 1–18. https://doi.org/10.1007/978-3-642-33167-1_1

[45] Jiaojiao Fu, Yangfan Zhou, Huan Liu, Yu Kang, and Xin Wang. 2017. Perman: Fine-Grained Permission Management for Android Applications. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. 250–259. https://doi.org/10.1109/ISSRE.2017.38

[46] Jiaojiao Fu, Yangfan Zhou, and Xin Wang. 2019. Component-Based Permission Management of Android Applications. *Software: Practice and Experience* 49, 9 (2019), 1402–1418. https://doi.org/10.1002/spe.2734

[47] Google. 2023. Provide Information for Google Play's Data Safety Section. Retrieved 2026-02-13 from https://support.google.com/googleplay/android-developer/answer/10787469?hl=en

[48] Google. 2025. Application Sandbox. Retrieved 2025-11-19 from https://source.android.com/docs/security/app-sandbox

[49] Google. 2025. SDK Runtime Overview | Privacy Sandbox. Retrieved 2025-11-20 from https://privacysandbox.google.com/private-advertising/sdk-runtime/architecture

[50] Google. 2026. Android Apps on Google Play. Retrieved 2026-05-11 from https://play.google.com/store/games?hl=en

[51] Google. 2026. Change App Permissions on Your Android Phone - Android Help. Retrieved 2026-04-16 from https://support.google.com/android/answer/9431959?hl=en

[52] Google. 2026. Claude by Anthropic - Apps on Google Play. Retrieved 2026-05-11 from https://play.google.com/store/apps/details?id=com.anthropic.claude

[53] Google. 2026. Fossify Voice Recorder Beta - Apps on Google Play. Retrieved 2026-05-11 from https://play.google.com/store/apps/details?id=org.fossify.voicerecorder

[54] Google. 2026. Manage Your Android Device's Location Settings - Google Account Help. Retrieved 2026-04-20 from https://support.google.com/accounts/answer/3467281?hl=en#zippy=%2Cwhen-location-is-on%2Cwhen-location-is-off

[55] Google. 2026. Pixel 10a - Explore the Specs. https://store.google.com/product/pixel_10a_specs

[56] Google. 2026. Pixel 9 - Explore the specs. Retrieved 2026-05-11 from https://store.google.com/product/pixel_9?hl=de

[57] Google Play. 2026. QR Scanner - Apps on Google Play. Retrieved 2026-05-11 from https://play.google.com/store/apps/details?id=com.scannerreader.qrcode.creatorfree&hl=en

[58] GrapheneOS. 2026. GrapheneOS: The Private and Secure Mobile OS. Retrieved 2026-02-20 from https://grapheneos.org/

[59] Guo Junjun. 2026. Junjunguo/PocketMaps. Retrieved 2026-02-17 from https://github.com/junjunguo/PocketMaps

[60] Fannv He, Jice Wang, Yuhang Huang, Xiancui Peng, and Yuqing Zhang. 2024. LibGuard: Protecting Sensitive Data In Android Third-Party Libraries From XLDH Attacks. In *2024 33rd International Conference on Computer Communications and Networks (ICCCN)*. 1–6. https://doi.org/10.1109/ICCCN61486.2024.10637585

[61] Raul Herbster, Scott DellaTorre, Peter Druschel, and Bobby Bhattacharjee. 2016. Privacy Capsules: Preventing Information Leaks by Mobile Apps. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, Singapore Singapore, 399–411. https://doi.org/10.1145/2906388.2906409

[62] Shashank Holavanalli, Don Manuel, Vishwas Nanjundaswamy, Brian Rosenberg, Feng Shen, Steven Y. Ko, and Lukasz Ziarek. 2013. Flow Permissions for Android. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 652–657. https://doi.org/10.1109/ASE.2013.6693128

[63] Jinhan Hu, Andrei Iosifescu, and Robert LiKamWa. 2021. LensCap: Split-Process Framework for Fine-Grained Visual Privacy Control for Augmented Reality Apps. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, Virtual Event Wisconsin, 14–27. https://doi.org/10.1145/3458864.3467676

[64] Jie Huang, Michael Backes, and Sven Bugiel. 2021. A11y and Privacy Don't Have to Be Mutually Exclusive: Constraining Accessibility Service Misuse on Android. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 3631–3648.

[65] Jie Huang, Oliver Schranz, Sven Bugiel, and Michael Backes. 2017. The ART of App Compartmentalization: Compiler-based Library Privilege Separation on Stock Android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1037–1049. https://doi.org/10.1145/3133956.3134064

[66] Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd Millstein. 2012. Dr. Android and Mr. Hide: Fine-Grained Permissions in Android Applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '12)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/2381934.2381938

[67] Ishika Keswani, Kerick Walker, Adrian Clement, Eusila Kitur, Nannapas Wonghirundacha, Ryan Aubrey, Vivien Song, and Eleanor Birrell. 2025. User Understandings of Technical Terms in App Privacy Labels. In *Twenty-First Symposium on Usable Privacy and Security (SOUPS 2025)*.

[68] Tejas Khatiwala, Raj Swaminathan, and V.N. Venkatakrishnan. 2006. Data Sandboxing: A Technique for Enforcing Confidentiality Policies. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. 223–234. https://doi.org/10.1109/ACSAC.2006.22

[69] Simon Koch, Manuel Karl, Robin Kirchner, Malte Wessels, Anne Paschke, and Martin Johns. 2025. The Impact of Default Mobile SDK Usage on Privacy and Data Protection. *Proceedings on Privacy Enhancing Technologies* 2025, 1 (Jan. 2025), 808–823. https://doi.org/10.56553/popets-2025-0042

[70] Simon Koch, Malte Wessels, Benjamin Altpeter, Madita Olvermann, and Martin Johns. 2022. Keeping Privacy Labels Honest. *Proceedings on Privacy Enhancing Technologies* 2022, 4 (Oct. 2022), 486–506. https://doi.org/10.56553/popets-2022-0119

[71] Konrad Kollnig, Anastasia Shuba, Reuben Binns, Max Van Kleek, and Nigel Shadbolt. 2022. Are iPhones Really Better for Privacy? A Comparative Study of iOS and Android Apps. *Proceedings on Privacy Enhancing Technologies* (2022).

[72] E. Kushilevitz and R. Ostrovsky. 1997. Replication Is Not Needed: Single Database, Computationally-Private Information Retrieval. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*. 364–373. https://doi.org/10.1109/SFCS.1997.646125

[73] Jehyun Lee, Akshaya Venkateswara Raja, and Debin Gao. 2019. SplitSecond: Flexible Privilege Separation of Android Apps. In *2019 17th International Conference on Privacy, Security and Trust (PST)*. 1–10. https://doi.org/10.1109/PST47121.2019.8949067

[74] Hugo Lefeuvre, Nathan Dautenhahn, David Chisnall, and Pierre Olivier. 2025. SoK: Software Compartmentalization. In *2025 IEEE Symposium on Security and Privacy (SP)*. 3107–3126. https://doi.org/10.1109/SP61157.2025.00075

[75] Jonathan Levin. 2018. **OS Internals. Volume 3: Security & Insecurity* (2nd edition ed.). Technologeeks.com, Edison, N.J.

[76] Haoran Lu, Yichen Liu, Xiaojing Liao, and Luyi Xing. 2024. Towards Privacy-Preserving Social-Media SDKs on Android. *33rd USENIX Security Symposium* (2024), 647–664.

[77] MapLibre. 2026. MapLibre Native. Retrieved 2026-02-19 from https://maplibre.org/projects/native/

[78] Nikolay Matyunin, Nikolaos A. Anagnostopoulos, Spyros Boukoros, Markus Heinrich, André Schaller, Maksim Kolinichenko, and Stefan Katzenbeisser. 2018. Tracking Private Browsing Sessions Using CPU-based Covert Channels. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. ACM, Stockholm Sweden, 63–74. https://doi.org/10.1145/3212480.3212489

[79] Meta Platforms, Inc. 2022. Fabric · React Native. Retrieved 2026-02-05 from https://reactnative.dev/architecture/fabric-renderer

[80] Meta Platforms, Inc. 2025. Core Components and Native Components · React Native. Retrieved 2026-02-05 from https://reactnative.dev/docs/intro-react-native-components

[81] Meta Platforms, Inc. 2026. React Native · Learn Once, Write Anywhere. Retrieved 2026-02-05 from https://reactnative.dev/

[82] Naveen Singh. 2026. FossifyOrg/Voice-Recorder. Retrieved 2026-02-20 from https://github.com/FossifyOrg/Voice-Recorder

[83] Ricardo Neisse, Gary Steri, Dimitris Geneiatakis, and Igor Nai Fovino. 2016. A Privacy Enforcing Framework for Android Applications. *Computers & Security* 62 (Sept. 2016), 257–277. https://doi.org/10.1016/j.cose.2016.07.005

[84] Organic Maps OÜ. 2026. Organic Maps: Offline Hike, Bike, Trails and Navigation. Retrieved 2026-05-11 from https://organicmaps.app/

[85] Sarah Prange, Pascal Knierim, Gabriel Knoll, Felix Dietz, Alexander De Luca, and Florian Alt. 2024. "I Do (Not) Need That Feature!" – Understanding Users' Awareness and Control of Privacy Permissions on Android Smartphones. (2024).

[86] Jun Qiu, Xuewu Yang, Huamao Wu, Yajin Zhou, Jinku Li, and Jianfeng Ma. 2022. LibCapsule: Complete Confinement of Third-Party Libraries in Android Applications. *IEEE Transactions on Dependable and Secure Computing* 19, 5 (Sept. 2022), 2873–2889. https://doi.org/10.1109/TDSC.2021.3075817

[87] Siegfried Rasthofer, Steven Arzt, Enrico Lovat, and Eric Bodden. 2014. DroidForce: Enforcing Complex, Data-centric, System-wide Policies in Android. In *2014 Ninth International Conference on Availability, Reliability and Security*. 40–49. https://doi.org/10.1109/ARES.2014.13

[88] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 2019. 50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System. In *28th USENIX Security Symposium (USENIX Security 19)*. 603–620.

[89] David Rodriguez, Joseph A. Calandrino, Jose M. Del Alamo, and Norman Sadeh. 2025. Privacy Settings of Third-Party Libraries in Android Apps: A Study of Facebook SDKs. *Proceedings on Privacy Enhancing Technologies* 2025, 2 (April 2025), 173–187. https://doi.org/10.56553/popets-2025-0056

[90] Gururaj Saileshwar, Christopher W. Fletcher, and Moinuddin Qureshi. 2021. Streamline: A Fast, Flushless Cache Covert-Channel Attack by Enabling Asynchronous Collusion. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 1077–1090. https://doi.org/10.1145/3445814.3446742

[91] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Taesoo Kim, and Insik Shin. 2016. FLEXDROID: Enforcing In-App Privilege Separation in Android. In *Proceedings 2016 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2016.23485

[92] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. 2012. AdSplit: Separating Smartphone Advertising from Applications. In *21st USENIX Security Symposium (USENIX Security 12)*. USENIX Association, Bellevue, WA, 553–567.

[93] Signal Technology Foundation. 2026. Signal Messenger: Speak Freely. Retrieved 2026-05-11 from https://signal.org/

[94] Stack Overflow. 2024. Technology | 2024 Stack Overflow Developer Survey. Retrieved 2026-02-05 from https://survey.stackoverflow.co/2024/technology

[95] StatCounter. 2026. Market Share of Mobile Operating Systems Worldwide from 1st Quarter 2009 to 4th Quarter 2025, by Quarter [Graph]. Retrieved 2026-02-12 from https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

[96] Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Maziéres. 2012. Addressing Covert Termination and Timing Channels in Concurrent Information Flow Systems. *SIGPLAN Not.* 47, 9 (Sept. 2012), 201–214. https://doi.org/10.1145/2398856.2364557

[97] Mohammad Tahaei, Ruba Abu-Salma, and Awais Rashid. 2023. Stuck in the Permissions With You: Developer & End-User Perspectives on App Permissions & Their Privacy Ramifications. In *Proceedings of the 2023 CHI Conference on*

Human Factors in Computing Systems (CHI '23). Association for Computing Machinery, New York, NY, USA, 1–24. https://doi.org/10.1145/3544548.3581060

[98] Byron Tau. 2024. How the Pentagon Learned to Use Targeted Ads to Find Its Targets—and Vladimir Putin. Retrieved 2026-02-23 from https://www.wired.com/story/how-pentagon-learned-targeted-ads-to-find-targets-and-vladimir-putin/

[99] The Android Open Source Project. 2022. Android Permissions for System Developers. Retrieved 2025-11-18 from https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/permission/Permissions.md

[100] The Android Open Source Project. 2024. App-Ops. Retrieved 2025-11-19 from https://android.googlesource.com/platform/frameworks/base/+/refs/heads/main/core/java/android/app/AppOps.md

[101] The Android Open Source Project. 2024. Security-Enhanced Linux in Android. Retrieved 2026-02-11 from https://source.android.com/docs/security/features/selinux

[102] The Android Open Source Project. 2025. Privacy Indicators. Retrieved 2025-11-19 from https://source.android.com/docs/core/permissions/privacy-indicators

[103] The Apache Software Foundation. 2026. Apache Cordova. Retrieved 2026-02-05 from https://cordova.apache.org/

[104] Fabo Wang, Yuqing Zhang, Kai Wang, Peng Liu, and Wenjie Wang. 2016. Stay in Your Cage! A Sound Sandbox for Third-Party Libraries on Android. In *Computer Security – ESORICS 2016*, Ioannis Askoxylakis, Sotiris Ioannidis, Sokratis Katsikas, and Catherine Meadows (Eds.). Springer International Publishing, Cham, 458–476. https://doi.org/10.1007/978-3-319-45744-4_23

[105] Jice Wang, Yue Xiao, Xueqiang Wang, Yuhong Nan, Luyi Xing, Xiaojing Liao, JinWei Dong, Nicolas Serrano, Haoran Lu, XiaoFeng Wang, and Yuqing Zhang. 2021. Understanding Malicious Cross-Library Data Harvesting on Android. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 4133–4150.

[106] Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du. 2014. Compac: Enforce Component-Level Access Control in Android. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*. ACM, San Antonio Texas USA, 25–36. https://doi.org/10.1145/2557547.2557560

[107] Shijia Wei, Austin Harris, Yongye Zhu, Prakash Ramrakhyani, Calvin Lin, and Mohit Tiwari. 2024. Tail Victims in Termination Timing Channel Defenses Beyond Cryptographic Kernels. In *2024 International Symposium on Secure and Private Execution Environment Design (SEED)*. 11–22. https://doi.org/10.1109/SEED61283.2024.00012

[108] Jiawei Zhan, Quan Zhou, Xiaozhuo Gu, Yuewu Wang, and Yingjiao Niu. 2017. Splitting Third-Party Libraries' Privileges from Android Apps. In *Information Security and Privacy*, Josef Pieprzyk and Suriadi Suriadi (Eds.). Springer International Publishing, Cham, 80–94. https://doi.org/10.1007/978-3-319-59870-3_5

[109] Xiao Zhang, Amit Ahlawat, and Wenliang Du. 2013. AFrame: Isolating Advertisements from Mobile Applications in Android. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC '13)*. Association for Computing Machinery, New York, NY, USA, 9–18. https://doi.org/10.1145/2523649.2523652

[110] Yuan Zhang, Min Yang, Guofei Gu, and Hao Chen. 2016. Rethinking Permission Enforcement Mechanism on Mobile Systems. *IEEE Transactions on Information Forensics and Security* 11, 10 (Oct. 2016), 2227–2240. https://doi.org/10.1109/TIFS.2016.2581304

[111] Zi-Peng Zhang, Ming Fu, and Xin-Yu Feng. 2019. A Lightweight Dynamic Enforcement of Privacy Protection for Android. *Journal of Computer Science and Technology* 34, 4 (July 2019), 901–923. https://doi.org/10.1007/s11390-019-1949-1

[112] Xiaonan Zhu, Jinku Li, Yajin Zhou, and Jianfeng Ma. 2020. AdCapsule: Practical Confinement of Advertisements in Android Applications. *IEEE Transactions on Dependable and Secure Computing* 17, 3 (May 2020), 479–492. https://doi.org/10.1109/TDSC.2018.2814999