LensCap: Split-Process Framework for Fine-Grained Visual Privacy Control for Augmented Reality Apps

Jinhan Hu, Andrei Iosifescu, Robert LiKamWa
Meteor Studio, Arizona State University
Tempe, Arizona, USA
jinhanhu,aiosifes,likamwa@asu.edu

ABSTRACT

Augmented Reality (AR) enables smartphone users to interact with virtual content spatially overlaid on a continuously captured physical world. Under the current permission enforcement model in popular operating systems, AR apps are given Internet permission at installation time, and request camera permission and external storage write permission at runtime through a user's approval. With these permissions granted, any Internet-enabled AR app could silently collect camera frames and derived visual information for malicious intent without a user's awareness. This raises serious concerns about the disclosure of private user data in their living environments.

To give users more control over application usage of their camera frames and the information derived from them, we introduce LensCap, a split-process app design framework, in which the app is split into a camera-handling visual process and a connectivity-handling network process. At runtime, LensCap manages secured communications between split processes, enacting fine-grained data usage monitoring. LensCap also allows both processes to present interactive user interfaces. With LensCap, users can decide what forms of visual data can be transmitted to the network, while still allowing visual data to be used for AR purposes on device. We prototype LensCap as an Android library and demonstrate its usability as a plugin in Unreal Engine. Performance evaluation results on five AR apps confirm that visual privacy can be preserved with an insignificant latency penalty (< 1.3 ms) at 60 FPS.

CCS CONCEPTS

• **Security and privacy** → *Software security engineering*.

KEYWORDS

Augmented Reality Security; Visual Privacy; AR Application Development; Split-Process Control; Unreal Engine

1 INTRODUCTION

Augmented Reality (AR) provides a unique interactive experience of virtual objects overlaying on top of the real-world environment enhanced by continuous capturing, processing, and rendering of visual data through a mobile device. The development of mobile devices and AR frameworks have enabled applications of AR in many fields, including education, entertainment, medicine, navigation, shopping, etc., and the future of AR market is expected to continue its rapid growth [8, 18, 24, 25, 42, 54].

Unfortunately, running AR apps on today's mobile devices poses serious privacy concerns, potentially revealing private user information in a user's visual environment to third party entities without the user's knowledge. Under the current permission enforcement model, an AR app is given Internet permission at installation time and granted camera permission and external storage write permission at runtime by users. Developers are required to prompt users with contextual information about why certain permissions are required, but such permissions are seemingly justified for proper AR operation; camera frames are necessary to visually integrate virtual content with a user's physical environment and Internet connectivity is needed for cloud-powered services or multiplayer networking. But once enabled, malicious developers of AR apps could silently collect camera frames and the information derived from them for malicious intent, including sending visual data to a private server, unbeknownst to the user. Without granular control over what kind of visual data is accessible for local storage or cloud storage, those collected camera frames could contain very private data at any given time, ranging from credit cards left on the table, text recognized from business documents on laptop monitors, to critical facial identities. **How do we protect users from surreptitious collection of visual data while maintaining usable visual computing for AR applications?**

Related solutions attempt to protect visual privacy by processing camera frames into privacy-preserving visual features and only give apps access to those features, or a region of the camera frame defined by the users [2, 28, 41, 46, 47, 50], as shown in Figure 1. However, this is too restrictive for AR apps, which need the ability to visually render the entire frame to provide the camera view as a backdrop for virtual AR overlays. Information flow control protects sensitive data through dataflow analysis and taint tracking [7, 16, 19, 62]. However, most information flow control works are only effective on data types involving low throughput. TaintDroid introduces 500 ms latency capturing still pictures. FlowFence consumes 100 ms processing 612x816 camera frames for face recognition in security camera apps. This latency overhead is untenable for AR apps, which must process at 16.6 ms per frame to maintain 60 FPS. Compartmentalization attempts to partition an app to confine private data in a secured hardware or software environment [14, 15, 23, 26, 45, 61], usually against threats from external third-party plugins or advertisement libraries.

MobiSys '21, June 24-July 2, 2021, Virtual, WI, USA

ACM ISBN 978-1-4503-8443-8/21/07…$15.00
https://doi.org/10.1145/3458864.3467676

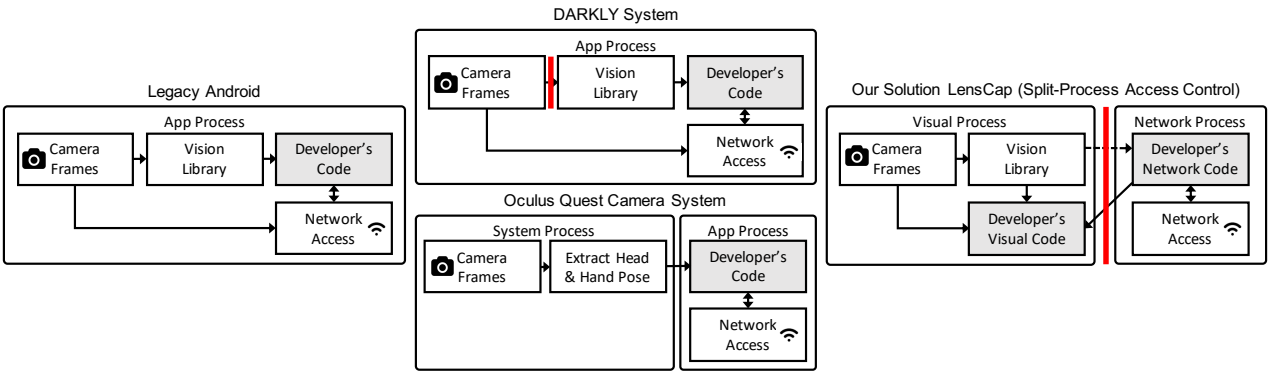


Figure 1: In legacy Android, AR developers could collect any camera frames and information derived from them without user awareness. Related solutions in DARKLY [28] and Oculus Quest system [1] only allow developer access to pre-defined processed visual data, and do not allow rendering of the camera frame. LensCap adopts split-process access control to allow developer's code to freely access frames for AR rendering while managing what visual data is accessible to the network.

To address the privacy disclosure of continuous camera usage, we introduce LensCap, an application development framework built on top of split-process access control [29], which allows users with fine-grained and proactive control over the app's potential transmission of camera frames and the information derived from them. The idea of split-process access control is not new; Android OS splits the `mediaserver` process into multiple processes to restrict their usage (after v7.0 [3]). In LensCap, the split-process paradigm is adopted in the application layer, which is integrated into the app development flow. An AR application is split into a *visual process* with full access to operate on camera frames (but with network permission revoked) and a *network process* to maintain Internet communications (but with camera permission revoked), enforced by extending the legacy Android permission enforcement. We enable both processes to present user interfaces through screen-based overlay composition. Then, data related to camera frames that need to be used in the network process can only be transmitted out of the visual process boundary through our trusted LensCap communication services, wrapped around trusted AR frameworks, and subject to the user's monitoring and approval through LensCap data usage notifications at a fine granularity. If users wish to allow network access to entire camera frames, e.g., for social media sharing or cloud-powered vision [22, 57], they can enable such permission. On the other hand, if a user wants to limit network access to only the camera pose, e.g., for multiplayer purposes, the user will be able to do so while still enjoying a full AR overlay on the device.

Thus, LensCap split-process app development framework enables: *(i)* AR apps and vision libraries to have expressive access of camera frames, their processing, and their rendering; *(ii)* fine-grained user control of the potential transmission of visual data; *(iii)* detailed context provided to users, regarding what data is sent to the cloud and at what times. We acknowledge that split-process frameworks may still be vulnerable to security threats through covert channel and side channel attacks [35, 48], which are beyond the scope of this paper. However, process-based partitioning of resources narrows the attack surface and could enable protection measures, such as permission plugins [45]. Ultimately, LensCap relies on the operating system to protect the communication channels and the app's internal storage to secure visual privacy across the split-process boundary [6, 10].

We prototype LensCap as an Android library that can work with standalone Android projects, as well as with Unreal Engine (UE) projects. In UE, LensCap serves as a plugin, through which a UE game compilation process will automatically generate and compile the split-process Android project structure. The communication channels between split processes in UE are provided to AR developers as Blueprint-callable and leverage the Android environment through the Java Native Interface (JNI). The data usage monitor is implemented as Android notifications, which provides the user with a rendered status of potential visual data collection.

We evaluate LensCap in five cloud-based AR applications that require the sharing of different types of image features, including camera pose, light estimation, point cloud, face region, and the camera frame. We find that the interactive latency between split processes and the overhead in app performance is negligible, even at 60 frames per second. Our user study further validates the performance similarity from the user's perspective and the improvement in user confidence while using untrusted AR apps.

We make the following contributions:

- We introduce an app development framework that is built on top of split-process access control, which allows users to proactively control their visual privacy in multi-user and cloud-based AR apps at an unprecedented fine granularity.
- We prototype a split-process compilation tool for AR developers of Android, as well as for Unreal Engine to develop visual privacy-preserving AR apps.
- We evaluate our system in five cloud-based AR apps that require the upload of different types of data in the UE-Android environment. We demonstrate that visual privacy in AR apps can be controlled by users without sacrificing the performance of their AR experiences.

The rest of this paper is organized as follows: §2 background, §3 threat and trust model, §4 design, §5 programming model, §6 implementation, §7 evaluation, §8 related works, and §9 discussion.

2 BACKGROUND

In this paper, we study the AR development flow in UE, as well as the permission control model and security enforcement in Android OS. They are similar across other game development platforms and operating systems such as Unity for iOS.

2.1 Mobile AR Development

Powerful real-time 3D creation tools such as Unreal Engine [17] have gained popularity in creating cutting-edge content in immersive interactive experiences. The basic building block in UE is the module. Each module exposes itself to other modules through a public interface. Developers can include a set of modules to realize desired app functionalities. For example, to develop AR apps relying on the ARCore library, app developers need to declare the `AugmentedReality` module and the `GoogleARCoreBase` module in dependencies. Apart from those standard modules, developers can create plugins to add per-project code and data to extend runtime gameplay functionality of the app.

UE supports all popular mobile platforms including Android and iOS. To build and run UE apps in the Android environment, UE creates an intermediate Android project based on two workflows. First, UE provides all source files that are necessary for the intermediate Android project to be compiled and run, such as a `GameActivity.java` template, on top of which developers can customize logic and functionalities. Second, all necessary assets for developing the UE project such as the level Blueprint are compiled into a `.so` library as the native code. The compiled Android application interacts with UE features through JNI.

2.2 Permission Control

Android requires apps to define permissions in a signed manifest file to manage the security of various operations [5]. Permissions are categorized into different protection levels as normal, signature, and dangerous, in which dangerous permissions must be prompt to and further granted by users. Permissions for Internet (`permission.INTERNET`), camera (`permission.CAMERA`), and external storage write (`permission.WRITE_EXTERNAL_STORAGE`) are essential to an AR app.

Screen buffer capture (screen reading/recording) is also governed by Android signature permissions. Android apps can specifically prompt the user when screen buffer capture is required. Users can enable this at their own discretion, understanding that everything on the screen will be accessible by the application, including any visible camera feeds.

Internet permission is categorized under the normal protection level. It is automatically granted at installation time by the system and users will not be notified that the permission is granted. Camera permission is essential for protecting user's visual privacy, and is therefore categorized under the dangerous protection level. Users will be notified to grant the camera permission in a prompt dialog. In Android 11, users are able to grant one-time permission to application's camera usage called "Only this time". However, the app will still have continuous camera access during that one-time. Write external storage permission is also categorized under the dangerous protection level which requires the user approval at runtime.

2.3 Security Enforcement

In Android, security is enforced through app sandboxing. Each app runs in a separate sandbox with a unique application identity (UID) given during installation. Sandboxing ensures each app has its own process and data storage associated with the UID. App sandboxes cannot interact with each other and only have limited access to system services by default. If a certain permission is granted, it will be reflected in the context of the application package and UID. Android conducts a permission check if the app sandbox requests access to specific system services (e.g. camera service) or resources belonging to other apps.

Sandboxing introduces the need of inter-process communication (IPC). Android implements Binder IPC as an essential mechanism to perform operations between processes, such as passing messages and requesting system services. Binder IPC provides functionalities to bind to functions and data between different execution environments. The Binder IPC driver is implemented in the kernel. It exposes basic kernel-understood functions to the application developers through the IBinder interface at the framework layer defined using the Android Interface Definition Language (AIDL) [4].

3 THREAT AND TRUST MODEL

Threat model We focus on scenarios involving third-party AR apps that require Internet connectivity. Relying on cloud or edge-based computing platforms empowers mobile AR apps with additional computing power and dynamic access to networked resources, e.g., game state, object models/textures, content updates, etc. This model also includes collaborative multi-user AR games in which different AR users could share information such as camera states, point clouds, and lighting estimations for more accurate tracking and rendering. However, while they provide useful and engaging functionality, **we assume that all such third-party AR apps cannot be fully trusted.** Privacy leakage through camera frames could happen at any time during the AR experience, with apps surreptitiously collecting visual data without user awareness. The visual data may be full camera frames themselves, derived semantic information (e.g., text or face identities), or compressed representations. The data could be immediately transmitted upon capture or stored locally before sending the data over the network, e.g., bundled with other data upload.

Trust model *(i)* We assume that operating systems such as Android and iOS are trusted to perform runtime permission check, hardware platforms are secured against attacks, and official AR frameworks such as Google ARCore and Apple ARKit are trusted to operate on camera frames for AR functionalities. These components are usually secured through a set of operating systems and hardware security measures [6, 10]. *(ii)* We assume that visual data sharing is valid as long as users are aware of it and specifically grant it. That is, AR applications could be allowed to share camera frames or information derived from them with the user's approval. *(iii)* We assume that data downloaded from the Internet or read from the memory is not tampered with. The protection of AR visual output is actively discussed in other research works [36, 37, 49].

Challenges *(i)* The AR experience must be quick to respond to user movement and interaction; the system solution should not contribute any visible performance overhead. *(ii)* Visual details need

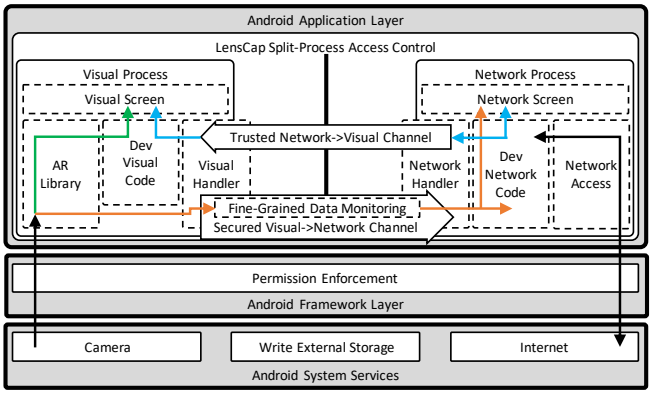


Figure 2: LensCap in Android. AR library output can go to developer's visual code to render on visual screen (green), or to network code through secured LensCap Visual→Network channel (orange). The blue arrow shows Network→Visual dataflow, defined in §4.2.

to be protected without reducing the amount of information that an AR application requires. *(iii)* Apps may require Internet connection to utilize more powerful cloud- and edge-based computing and/or maintain state on networked resources. *(iv)* There is a trust gap to be mitigated between users and AR apps in terms of the data claimed to be collected and the data actually collected.

4 DESIGN

We propose LensCap, a framework that secures visual privacy in AR apps through *(i)* enforced split-process access control, *(ii)* secured communication channels for processes to securely exchange data, *(iii)* screen-based overlay composition, and *(iv)* fine-grained data monitoring. Figure 2 shows LensCap in the Android system.

4.1 Enforced Split-Process Access Control

LensCap requires applications to be split into a visual process and a network process. Through this process-based partitioning, the permission enforcement in the operating system can assure that AR apps isolate visual processing from network access except via explicit user approval.

The visual process is responsible for processing and rendering camera frames, and thus is allowed to operate on them directly. Expressive functional use of the frames and visual data allows developers to program computer vision and image processing operations freely, as long as the features derived from those camera frames do not leave the visual process boundary, except with explicit user approval.

The network process is not allowed to operate on camera frames or other visual data, except with explicit user approval. However, the network process is critical to edge- or cloud-based AR apps for its capability of providing Internet communications, as well as writing to the external storage. LensCap sequesters the network process from direct access to the camera frames by revoking camera permissions, but offers it upload and download access to the network and write access to external storage.

The permission enforcement, residing inside the Android runtime framework, verifies that when a process attempts to open the camera, it does not have either the network permission or write external storage permission granted. LensCap also verifies that when a process attempts to write to the external storage, it does not have the camera permission granted. Violations will throw exceptions to notify users about potentially malicious behavior.

4.2 Secured Communication Channels between Split Processes

To securely support a range of operations between the visual process and the network process, LensCap governs communication between the two processes through two data *Handlers*, one for each process, which enact two channels: *Network→Visual* and *Visual→Network*.

Network→Visual is implicitly trusted. Thus, the network process could send data to the visual process freely, as visual privacy would still be confined inside the visual process. The protection of visual rendering is beyond the scope of this work, but studied in related works [36, 37].

Visual→Network needs to be explicitly secured. LensCap scrutinizes the data sent from the visual process to the network process and presents access control and access logs to users. Users should assume that any data that travels across the Visual→Network channel are visible to the network. This involves any data that could possibly be used to invade user's privacy, ranging from visual details as small as the camera pose to data as large as the entire camera frame.

To prevent developers from hiding visual information in computed data, LensCap's Visual→Network channel can only transmit specific untainted visual data, including camera frames or direct outputs from the AR library, e.g., camera pose and face tracking features. LensCap restricts all other forms of transmission on the Visual→Network channel. In §4.4, we go into further detail into how such visual data access is monitored by the system, presented to the user, and selectively permitted by the user.

4.3 Screen-Based Overlay Composition

The screen-based overlay composition allows developers to expressively create screen-based interactions through the visual process screen and the network process screen. Both screens include support for the full array of touch-based user interfaces: buttons, sliders, swipes, or custom-designed screen-based interactions. LensCap composes the visual output by overlaying the network process screen surface over the visual process screen surface.

Figure 3 shows an example in which the tap-hold-rotate behavior to interact with the virtual object is implemented in the visual process and rendered in the visual process screen, as it would be in a legacy app. Meanwhile, an interactive search bar is hosted in the network process screen, through which the user can search for different 3D models and download them for rendering.

4.4 Fine-Grained Data Monitoring

LensCap monitors the usage of visual data with a *data usage monitor* and a *data usage switch*, as shown in Figure 3.

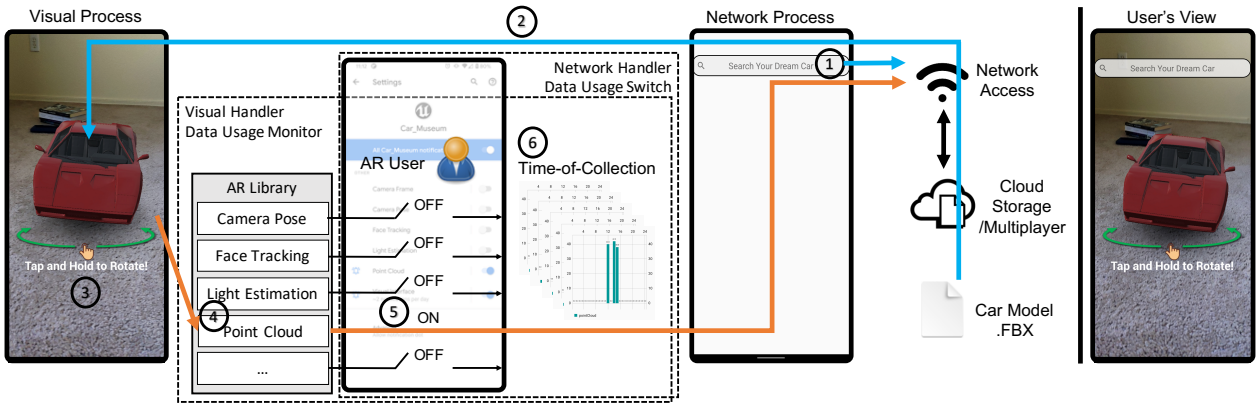


Figure 3: ① A user searches for a car in a search bar presented through the network process screen. ② The corresponding car model is downloaded from the cloud, transmitted to the visual process through LensCap Network→Visual channel, and rendered on the detected AR tracking plane in the visual process. ③ The user can spatially interact with the virtual 3D car model. Meanwhile, ④ point cloud positioning data is shared over the network for shared tracking for a multiplayer AR experience, subject to ⑤ user's approval secured by LensCap data usage monitor and switch in Visual→Network channel. ⑥ Visual data usage can be reviewed by the user.

The data usage monitor The data usage monitor wraps around the vision library API for three purposes: *(i)* it lets app developers utilize the AR library and the Visual→Network communication channel; *(ii)* it checks the user permission and then documents when and how often each monitored function is called; *(iii)* it ensures data computed from the trusted vision library and the data sent through the visual data handler are identical, i.e., untainted.

For *(i)*, LensCap wraps around the AR library, preserving original usability. The developer's app invokes the LensCap AR functions to obtain original AR library function output. The developer's app invokes LensCap transmission functions to request the sending of AR library output or camera frames to the network process. For *(ii)*, LensCap monitors each wrapper function with a separate permission label, counter, and timer. LensCap updates the permission label according to the user's choice in the data usage switch. Upon visual data transmission across the Visual→Network channel, LensCap increments the counter and timer, documenting the time of visual data access from the network process. For *(iii)*, LensCap acquires a copy of the data when each monitored wrapper function is invoked by the app. The visual data is checked for identical comparison with the copy before transmitted through the Visual→Network channel. We have found that memory comparison (e.g., `memcmp`) is sufficiently performant and hash comparisons are not needed.

The data usage switch We design a data usage switch to present data usage notifications and logs to users with two considerations. First, the data usage switch allows users to customize the visual data they allow to be shared at a fine granularity, i.e., users are able to specifically disable the Visual→Network communication for each monitored function to prohibit its output from being passed out of the visual process boundary. Second, the data usage switch collects access timing information from the data usage monitor and presents access charts to users visually. Through this, users are able to transparently see what visual features are potentially shared over the network and at what times.

5 PROGRAMMING MODEL

As with other operating system privacy changes, LensCap requires developers to alter their app development patterns. In LensCap, developers must partition their applications into the visual and network processes and manage communications between the processes. Here, we describe four template scenarios to illustrate the programming model with LensCap split-process access control integrated into an AR development flow, as shown in Figure 4.

LensCap allows developers to consider the "principle of least privilege" and specify only the necessary level of access control for each process. In this light, Scenarios 1 and 2 are supported without any special Visual→Network privilege while still enabling immersive interactive AR functionality. Meanwhile, Scenario 3 only requires the user to allow specific visual data, e.g., camera pose, to be shared across the boundary. Finally, Scenario 4 allows developers to share camera frames and visual data over the network with the explicit permission of the user. These scenarios serve as examples of how applications can be developed in the LensCap environment. Complex applications can be thought of as combinations of these scenarios; Figure 3 shows an app that has elements of Scenarios 1 (green), 2 (blue), and 3 (orange). Note that it is developer's responsibility to take care of the application behavior if the visual data requested are not permitted by users in scenario 3 and 4.

Scenario 1: AR camera overlays and spatial interactions. Developers can program spatial interactions with the AR scene in the visual process, e.g., allowing users to place, spin, scale, and otherwise interact with virtual 3D object content. In this scenario, the application does not need to request any LensCap permissions, as all visual data can be contained in the visual process.

Scenario 2: Network requests without visual data needs. Developers can program network requests in the network process. These requests can be triggered by non-AR on-screen canvas user interface (UI) elements (buttons, sliders, etc.), by time-based events, or by other activities that don't require visual data. Notably, in this

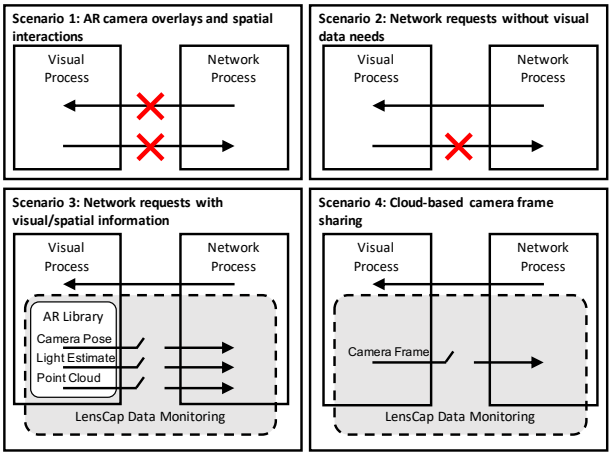


Figure 4: LensCap protects AR users in apps where outputs from AR libraries or the entire camera frames are to be sent to the network.

scenario, LensCap allows network process to influence activities in the visual process through the unidirectional Network→Visual channel, e.g., specifying what to render based on button UIs that are tapped in the network process. Downloaded data can also influence the spatial rendering, providing object models, data to visualize, image textures, synchronized game state, etc. As in Scenario 1, the application does not need to request any LensCap permissions, as nothing from the visual process needs to interact with the network process.

Scenario 3: Network requests with visual/spatial information. Developers may need to share visual or spatial information among multiple devices for multiplayer positioning, joint illumination estimation, or other cloud-based activities. In this case, LensCap allows developers to request user permission to expose specific AR library outputs over the Visual→Network channel. For example, developers can explicitly request the camera pose permission in order to use AR positioning data inferred in the visual process for network-based rendering. Similarly, the developer can explicitly request point cloud permission to enable point cloud sharing for a shared multiplayer AR rendering experience. This scenario also facilitates AR-based spatial interactions that trigger network downloads, as spatial interactions may be inextricable from both visual data and network data. Altogether, in this scenario, LensCap only requires users to enable specific visual data to be shared, without exposing their entire camera frame and the potential secrets or embarrassments therein.

Scenario 4: Cloud-based camera frame sharing. In some apps, users may want to send their entire camera frame to the cloud to enable livestreaming and social media sharing. Other apps may require cloud-based processing of camera frames for resource intensive and/or collective vision operation. In these apps, developers can request users to allow camera frames to pass through in the LensCap Visual→Network channel. For this scenario, users can be made aware that their visual privacy might be leaked and selectively disable camera frame access at their discretion. Furthermore, users can review the LensCap data usage monitor to observe when camera frames are collected to infer malicious intent.

6 IMPLEMENTATION

In this section, we describe a prototype of LensCap in the Android ecosystem. We implement developer support in the form of libraries and automated compilation tools to support both standalone Android development and/or UE development revolving around the ARCore framework. The implementation is generic to other game design platforms, e.g., Unity, and other AR frameworks, e.g., Apple ARKit.

6.1 LensCap Permission Enforcement

We implement the split-process permission enforcement in the Android framework layer (AOSP v9.0.0_r46). The implementation involves `CameraManager` in the `Camera2` API, `ContextWrapper` in the `Content` API, and the Android namespace defined in the `xmlns:android`.

LensCap defines the `permission.LENSCAP` manifest permission attribute. Once the app is started, LensCap prompts users to ask if they "allow the app to use LensCap to monitor and validate visual information uploaded to the Internet". If users choose "ALLOW", LensCap permission system will be enforced. Inside the `CameraManager`, LensCap verifies if either Internet or write external storage permission is granted when camera permission is to be granted, relying on the `PackageManager` and `AppGlobals` for retrieving the permission of each app based on its UID. Violations will throw security exceptions to prevent the app from accessing the camera service. The implementation in `ContextWrapper` for protecting the visual privacy from being written to an external storage is similar. The app calls `getExternalFilesDir()` to inquire the absolute path of the directory on the primary shared/external storage device, prompting users to grant write external storage permission. Here, LensCap verifies if the app also has camera permission granted. Violations will lead to `null` returned as the path to invalidate the writing.

6.2 Split-Process for UE Development

Developers can use LensCap to split their app in the standard Android environment. However, to facilitate game engine-based development, we implement an automated tool to generate LensCap-provisioned Android projects from UE projects as part of the compilation process.

We keep the structure of the UE-generated intermediate Android project and reuse its main application module as either the visual process or the network process. Then *(i)*, for the UE process, we automate the insertion of our LensCap APIs into the main `GameActivity.java` file by adding function definitions into the `<gameActivityClassAdditions>` inside the plugin's XML file. These APIs are described in §6.3 for realizing LensCap-verified interactions between the network process and the visual process. In addition, startup permissions related to write external storage and Internet communication given to the visual process are automatically removed by modifying the source `AndroidManifest.xml` file. *(ii)*, for the non-UE process, we implement it as an individual app package which has its own workspace, from source code to build

configurations. To automate the generation of non-UE processes, we provide the source code of the LensCap app package as a third-party library for UE, together with the two data handlers. During compilation, the source code is copied to the build directory of the project and built with the visual process app together. Developers can implement processing logic inside UE as in the legacy development flow and/or utilize the Android process scenario templates LensCap provided to create complex network-visual interactions.

6.3 Secured Communication Channels

The visual/network data handlers are implemented in both the Android and the UE environment.

In Android The two data handlers are implemented as two libraries written in Java and Kotlin. The visual data handler is compiled with the visual process, whereas the network data handler is compiled with the network process. Both data handlers have a transmitter service and a receiver service, for which the transmitter service of the network data handler binds to the receiver service of the visual data handler and vice versa. Data transmitted between the two handlers are shared through Android shared memory (`ashmem`). At app level, developers only need to initialize the two handlers in each process accordingly. Then, they can use the following APIs to send and receive data securely between split processes.

To receive data, we use Android's AIDL feature to create an `onData()` listener for monitoring and receiving the incoming data. Then, we expose a `Receiver<>` interface in the data handler to be registered with the desired string identifier and the data to be received as `ByteArray`.

```
fun onReceived(id:String, data:ByteArray) {}
```

Similarly, we expose a `Send()` interface to transmit the content through the data handler service as `ByteArray` associated with the desired identifier as string.

```
fun send(id:String, data:ByteArray) {}
```

In UE Data handler implementation involves: *(i)* exposing UE Blueprint callables, *(ii)* transferring data between the UE-Android boundary and exposing Android Java APIs.

First, the two data handler plugins are implemented in C++ to expose UE Blueprint callable functions for transmitting validated AR library outputs. For example, the following LensCap function can be called in UE Blueprint to collect the camera pose acquired from the AR library.

```
UFUNCTION(BlueprintCallable, meta = (DisplayName =
    "LensCap_GoogleARCore_Collect CameraPose",
    HidePin = "LastPose"))
static void VDH_Send_Camera_Pose(FTransform&
    LastPose);
```

Then, to pass data between the UE-Android boundary, LensCap plugins implement send and receive APIs through Android JNI. Currently, we provide support for UE-Android compatible data types, such as `int`, `float`, and `bool`, which are sent and received in the form of arrays. The send JNI function checks if the caller has a valid tag (to differentiate LensCap ARCore wrappers), converts UE data to JNI types, and is exposed to Android to connect data to or from UE.

6.4 Screen-Based Overlay Composition

We utilize the Android `WindowManager` to overlay the network process screen on the visual process screen. Although `WindowManager` allows the network process to draw its overlay over the visual process screen, the network process cannot observe the pixels of the visual process screen, preserving visual privacy. Adding the screen overlay requires `permission.SYSTEM_ALERT_WINDOW`, subject to user's approval at runtime. For correct overlay, the layout of the network process screen matches the visual process screen, such as width and height.

Inside the network screen overlay, we are able to implement on-screen touch interfaces to initiate user interactions with the visual screen. To do so, we override the `onTouch(view: View, motionEvent: MotionEvent)` function in the network process to send touch coordinates to the visual process. If developed in UE, the visual process translates the received touch coordinates into the UE coordinate system, which will finally be stacked and processed in UE's Android input interface. From the developer's perspective (and user's perspective), the app with the screen-based overlay operates exactly the same as the legacy application.

6.5 Fine-Grained Data Monitoring

The data usage monitor The monitor is linked to the Android visual data handler and the UE plugin with the following steps. First, we manually inspect the vision library API to determine the functions to be monitored. In general, the goal of an AR framework is to provide functionalities that operate on camera frames to determine trackables and estimate surroundings. In UE, the `GoogleARCoreFunctionLibrary` only contains hundreds lines of code and 50 functions written in C++, among which we focus on monitoring functions that *(i)* operate on camera frames, *(ii)* have a return value, *(iii)* work on a block of memory. After narrowing it down, only 25 functions need to be wrapped. Overall, the workload for inspecting the vision library is trivial, even when the vision library needs to be updated for a newer version iteratively. Note that not all of these functions may expose user privacy at a serious level. This presents a future research opportunity to investigate better UI for data usage monitoring that eases the user's burden to grant permissions at runtime, e.g., grouping functions exposing similar types of data and assign each group a risk level [5].

Second, we write a wrapper around each function being monitored. The wrapper function is statically tagged. Upon invocation, the LensCap user permission is checked. If the user allows specific visual feature collection, the monitored function is then called. Next, this wrapper gets each value from the output, concatenates them, stores them for a data integrity check, and sends them in the format of data array, together with the function tag, through the Visual→Network channel provided by the visual data handler.

Third, based on the tag of the function, we add a property into a `LensCapCounterStruct` and a `LensCapDateStruct` in the visual data handler accordingly to record how often and when the monitored function is called.

Finally, to check the data integrity before `send()` in the visual data handler is called, we directly expose a native function from the UE binary library to our visual data handler, which sends the data to be transmitted back to UE and compares with the original

data copy stored. Transmission to the network data handler occurs only if the data match.

The data usage switch The switch notifies users about what visual data is used in the network process and further presents interfaces for users to allow or disallow the transaction of each visual feature. The implementation involves a user permission inquiry and verification mechanism, as well as a notification interface. Figure 3 shows an AR app [20] with the data usage switch notifications shown in settings (in the middle). In this example, the data usage switch allows users to disable the Visual→Network transmission of camera pose, lighting estimation, face tracking, point cloud, and camera frame, which are the five use cases evaluated in §7.

For permission inquiry and verification, we implement a class `LensCapPermissionStruct` in the visual data handler to store `boolean` values for each type of output from AR library, as well as a function `boolean getPermTag(String tag)` to respond permission inquires. Then, we expose a permission inquiry function to UE through JNI again to help validate the transmission of a specific visual feature.

LensCap visual data handler service leverages the Android notification channels to present ON/OFF switches for users to selectively allow/disallow the transmission of a monitored visual feature. An ON indicates that the user approves the corresponding visual feature to be transmitted to the network process and the transmission will further be stored in the timer allowing users to analyze in detail about Time-of-Collection in the form of a bar chart. On the contrary, when a notification channel is disabled, values in `LensCapPermissionStruct` will be updated accordingly to prevent visual data from being transmitted.

7 EVALUATION

In this section, we demonstrate the performance of our system in various cloud-based AR apps, comparing legacy single-process app behavior with split-process LensCap app behavior, along with an interview-based user study, to ensure that visual privacy can be preserved without sacrificing app performance or user experience at runtime.

7.1 Benchmark Applications

To cover popular AR use cases, we build and evaluate five cloud-based AR apps [20] that share different types of visual AR data across multiple devices. These apps are developed in UE (v4.24) and then deployed to Android devices (Google Pixel 4 XL). A local desktop server serves as a cloud server, storing and passing data among mobile devices. Uploading and downloading data uses an OkHttp client implementation [59] with a WiFi connection.

The first app shares the *camera pose*, containing the estimated camera location and rotation, which is critical to tracking and rendering. Collaborative AR apps could share camera poses to achieve shared user viewport geometry, improve camera calibration, or provide runtime user/object tracking [53]. The second app shares the *lighting estimation*, encoding environmental illumination towards rendering virtual objects realistically. Multiple AR users could share radiance samples from multiple perspectives to achieve more accurate lighting estimation for more immersive rendering [44]. The third app shares the AR *point cloud*, which contains the 3D visual

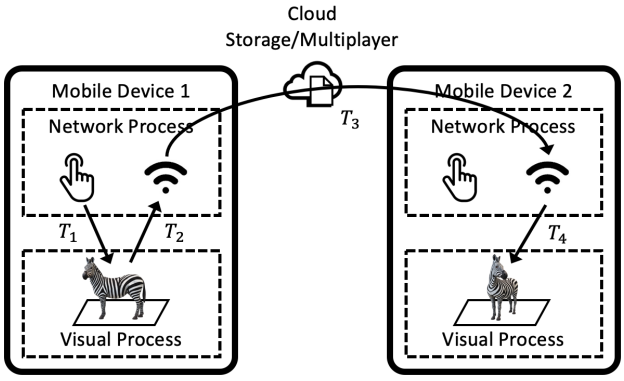


Figure 5: We evaluate the interactive latency with a combination of T_1 (Touch→Visual), T_2 (Visual→Network), T_3 (Upload&Download), and T_4 (Network→Visual).

corner points that are used to track the space. AR apps can share point clouds among users for shared positioning and/or send it to the cloud for object detection [13] and/or image-based localization [58]. The fourth app shares the *face tracking* result. Face tracking detects and tracks the image regions of faces between camera frames, sharing these over the network, e.g., for identity verification. The last app relies on sharing the *full frame*. In this model, AR apps offload camera frames to the cloud, e.g., for livestreaming, video chat, social media, or cloud-/cloudlet-based vision processing.

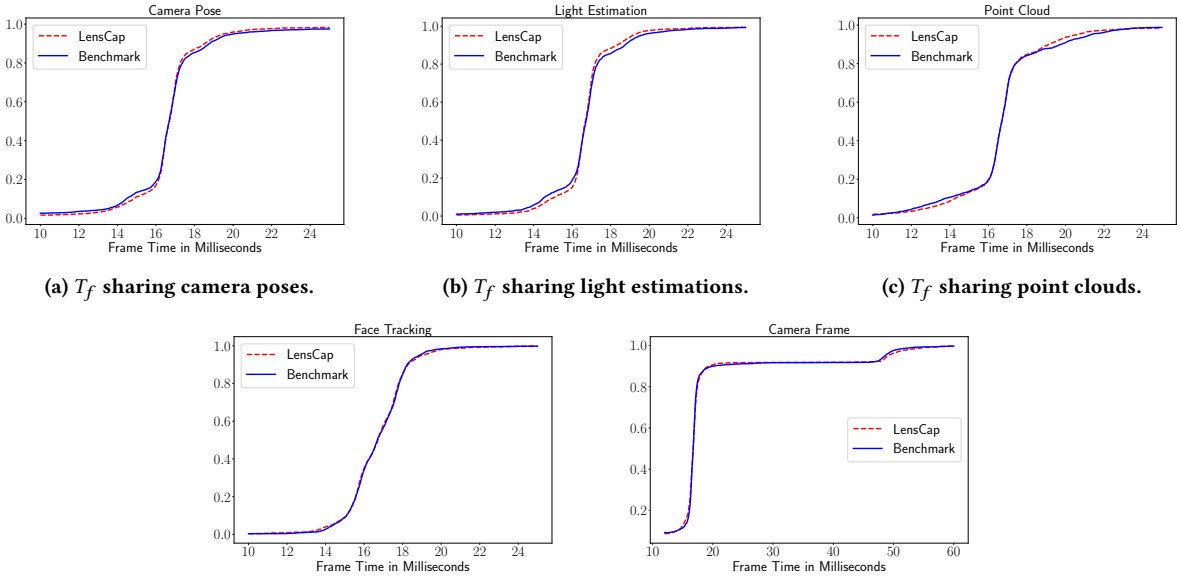
7.2 Evaluation Metrics

To explore LensCap's influence on application performance, we monitor the time interval between consecutive frames T_f (inverse to the frame rate). T_f is measured by the *deltaTime* (app time elapsed between frames) acquired from the `Tick()` UE Blueprint function, which is called every frame.

In addition, we measure and compare the interactive latency of four time intervals, T_1 to T_4 when running as the legacy single-process benchmark apps and as LensCap-enabled split-process benchmark apps. Depicted in Figure 5, the time intervals are defined as follows:

(1) T_1 represents the time elapsed between a user behavior and a visual rendering event.
(2) T_2 represents the time elapsed to transfer data from the visual process to the network process, which includes several actions, i.e., visual process to visual data handler, visual data handler to network data handler, and network data handler to network process.
(3) T_3 represents the roundtrip time elapsed to upload and download data between the network process and the cloud component, e.g., sending face detection results to the cloud and receiving a response.
(4) T_4 represents the time elapsed between when the network process acquires data from the cloud and when it is applied to the visual process, e.g., utilizing light estimation to improve rendering. (T_4 and T_2 are similar but reversed.)

All time intervals are measured by calculating the difference between system timestamps. We synchronize the system clock



(a) T_f sharing camera poses. **(b) T_f sharing light estimations.** **(c) T_f sharing point clouds.**

(d) T_f sharing face regions. **(e) T_f sharing camera frames.**

Figure 6: CDFs of T_f for collecting fives different types of visual data in every 10 frames demonstrate no performance overhead comparing between the LensCap-integrated apps and the legacy apps.

between the cloud server, the Android device, and the UE environment. For each app, we run the experiment for 2 minutes to acquire thousands of data samples, during which the data collection is performed roughly every 10 frames, an interval very commonly used in many keyframe-based continuous mobile vision applications [39].

7.3 Application Performance

We use T_f to compare and analyze the application performance in each example use case. A comparison of T_f between the benchmark and the LensCap-integrated app is shown in Figure 6 and its averaged value can be found in Table 1.

We first evaluate the app performance in the context of programming scenario 3 and 4, in which the network process requests visual data to be uploaded and downloaded from the cloud. In Figure 6a, 6b, 6c, and 6d, results show that most T_f is within 16 ms to 17 ms in both LensCap-integrated and benchmark apps, which indicates a very comparable AR performance that could be maintained at as high as 60 FPS, no matter for collecting camera poses, light estimations, point clouds, or face tracking results. In Figure 6e, the result shows that collecting the entire camera frames incurs latency overheads in both benchmark and LensCap-integrated apps. However, the additional latency comes from processing image planes of camera frames in UE in the visual process. Thus, the overall app performance is the same comparing between the benchmark and the LensCap-integrated apps, i.e., data communication between split processes does not incur noticeable latency overhead. In particular, in this worst case, the app performance could be maintained at around 55 FPS, if camera frames are collected every 10 frames.

In addition, we use app 4 to evaluate the app performance for programming scenario 1, in which the app just detects faces and draws overlays locally in the visual process, without network interactions. Results show that the average of T_f is 16.8 ms in the legacy app and 16.7 ms in the LensCap-integrated app. Furthermore, we combine app 1, 2, and 3 together into one app to evaluate programming scenario 2, in which the data downloaded from cloud is sent to the visual process repeatedly in a sequence after clicking an on-network-screen button. Results also show similar app performance compared between the legacy app and the LensCap-integrated app (both have an average of 16.7 ms T_f).

Summary The result demonstrates that the adoption of split-process access control does not appear to influence app performance, likely due to: (i) the computational ability of mobile devices to handle the operation of an additional process, and (ii) the non-blocking data sharing between split processes. Visual privacy can thus be monitored at the process boundary and preserved on the device subject to user's decision, without penalizing the app's performance. From our experiences in the evaluation, the AR experience is robust, smooth, and comparable (without noticing any differences) between LensCap-integrated and benchmark apps.

7.4 Interactive Latency

We use T_1, T_2, T_3, and T_4 to compare and analyze the interactive latency in each example use case. A comparison of their averaged values across all data samples between the benchmark and the LensCap-integrated app is also shown in Table 1.

Camera pose Results show that our system introduces an average of 0.2 ms T_1 Touch→Visual latency for initiating user interactions, as well as 0.3 ms T_2 Visual→Network and 0.3 ms T_4 Network→Visual latency for transmitting tens of bytes of camera pose data between processes.

	Camera Pose		Lighting Estimation		Point Cloud		Face Tracking		Camera Frame	
	Benchmark	LensCap	Benchmark	LensCap	Benchmark	LensCap	Benchmark	LensCap	Benchmark	LensCap
T_1	8.9	9.1	8.4	8.7	8.7	8.9	8.7	9.0	8.7	9.0
T_2	N/A	0.3	N/A	0.3	N/A	0.3	N/A	0.4	N/A	1.2
T_3	99	108	117	121	128	111	124	120	1253	1176
T_4	N/A	0.3	N/A	0.4	N/A	0.4	N/A	0.4	N/A	1.3
T_f	16.8	16.8	16.7	16.7	16.8	16.8	16.7	16.7	18.4	18.4

Table 1: Averaged evaluation results for all time intervals in milliseconds (ms). Note that the interactive latency between processes (T_2 and T_4) does not apply to benchmark applications.

Lighting estimation Results show that our system introduces an average of 0.3 ms T_1 Touch→Visual latency for initiating user interactions, as well as 0.3 ms T_2 Visual→Network and 0.4 ms T_4 Network→Visual latency for transmitting tens of bytes of lighting estimation data between processes.

Point cloud Results show that our system introduces an average of 0.2 ms T_1 Touch→Visual latency for initiating user interactions, as well as 0.3 ms T_2 Visual→Network and 0.4 ms T_4 Network→Visual latency for transmitting hundreds to thousands of bytes of point cloud data between processes.

Face tracking Results show that our system introduces an average of 0.3 ms T_1 Touch→Visual latency for initiating user interactions, as well as 0.4 ms T_2 Visual→Network and 0.4 ms T_4 Network→Visual latency for transmitting hundreds to thousands bytes of face tracking data between processes.

Full frame Results show that LensCap introduces an average of 0.3 ms T_1 Touch→Visual latency for initiating user interactions, and 1.2 ms T_2 Visual→Network and 1.3 ms T_4 Network→Visual latency for transmitting megabytes of camera data between processes.

Summary Split-process access control introduces negligible latency in its inter-process communications, as *(i)* the 0.2 ms to 0.3 ms overhead in T_1 is all but invisible, compared with the latency needs for gaming and other interactive touch-based applications; according to Jota et al [32], humans cannot differentiate touch latencies between 1 and 40 ms; and *(ii)* the 0.3 ms to 1.3 ms latency of T_2 and T_4 caused by the inter-process communication is negligible (without impairing the app performance), even for transmitting the entire camera frame, and even for round-trip operations and interactions across the two processes. On the other hand, cloud communication latency T_3 consumes hundreds of milliseconds (varying based on the network conditions).

7.5 User Study

Apart from the previous quantitative evaluation, we perform a user study to observe users' hands-on experience of LensCap-integrated AR apps. The user study is approved by our institution's IRB. We recruit a total number of 8 undergraduate and graduate students majoring in engineering to participate in this user study. The user study serves three purposes:

- We would like to find out whether the legacy apps and the LensCap-integrated apps have similar performance, from the user's perspective.
- We want users to freely express privacy concerns while using AR apps and evaluate whether their concerns are mitigated by LensCap.
- We invite users to explore and evaluate the LensCap data usage monitoring UI and brainstorm together with us for a better UI design.

User study procedure The user study is interview-based, which contains five activity-interview phases, described as below:

(1) *Preparation.* In this phase, besides reading the consent form, we asked the participants several questions to get their background in AR. For example, we asked them "What AR applications have you used before?", "Where do you usually use them?", and "How often do you use them?".
(2) *Application performance study.* In this phase, we conducted a blind user study, in which the legacy app and the LensCap-integrated app were presented to users in a random order. To make the two versions of apps identical from appearance, we temporarily disabled the LensCap data monitoring module. Participants were given enough time to explore both apps freely, e.g., putting a virtual car model and interact with it. Then, we asked them about their overall experience, including "Do you think both apps perform smoothly? If not, which app do you prefer?" and "What differences can you identify between these two apps?".
(3) *Privacy exposure awareness study.* In this phase, we let participants explore the legacy app again. Then, we asked them several questions to understand the baseline of user's trust in AR apps. These questions included "Imagine that some malicious app developers want to steal your identity, what kind of information in your AR experience do you think they can exploit?", "What makes you trust or not trust an AR app?", and "Do you think the current permission model can protect your visual privacy?".
(4) *LensCap introduction.* In this phase, we educated participants to be familiar with the LensCap app development framework. We explained to participants how LensCap splits the app into two process, how least-privileged split-process access control is managed, how the network screen is overlaid on top of the visual screen, and how users are able to control and visualize the data transactions between process boundaries at a fine granularity. Then, we answered any questions they had.
(5) *LensCap data usage monitoring UI study.* In this phase, we let participants explore the LensCap-integrated app again.

We evaluated the current LensCap data monitoring UI from both the usage aspect and the trust improvement aspect. We also invited participants to help us envision a better UI to be design in future that balances the usage and the trust. In particular, we asked participants several questions, including "Do you think LensCap can help you trust untrusted AR apps by allowing you to control what can be collected by the network?", "With the LensCap functioning logo rendering on the top left, do you feel confident (protected) while using untrusted AR apps even with no data usage notifications prompted?", and "What other types of notification or permission models would you like to be deployed that can further improve your trust in AR apps?".

User study observations From the interview responses, we garnered the following observations, validating the ability of LensCap from the user's perspective for maintaining app performance while endowing users with trust in random AR apps:

- *All participants are already familiar with AR technology.* They have more or less used AR apps before, in which Pokemon Go and social media apps such as TikTok and Snapchat are the most popular ones.
- *Smartphone is the main AR portal.* Other types of AR devices, such as wearable glasses had been barely used. Users noted that a killer app on those devices would be needed to boost their usage.
- *AR would be used everywhere at anytime.* Though currently AR usage is limited by apps, all participants anticipate to use AR at various places (from home to outside) and some of them even want to use AR all the time, including walking in the street.
- *Participants cannot differentiate between the legacy app and the LensCap-integrated app performance-wise.* All of them agree that the two versions of apps run equally smooth.
- *Participants want AR to be deployed in a wide range of fields.* Apart from simply putting virtual 3D content on top of the real world, participants want to see AR in "medical settings, to revolutionize surgery", "education, explaining complicated chemistry concepts by visualizing reactions", "designing and decorating", and "gaming and social interactions".
- *Participants are aware that visual privacy can be exposed in AR apps.* In particular, they are aware that sensitive information such as credit cards, faces, photos left around, and location captured by the camera could be stolen and utilized by malicious AR apps. One participant pointed out that "anything my camera is looking at, can be used for targeted advertisements".
- *Participants can trust an AR app only if they are given control of the visual stream,* e.g., "telling the app not to take my data", "as long as the visual data stays within the phone", and "clarification on what gets sent to the network and when". All participants think the legacy Android permission model is far from satisfying this condition.
- *All participants felt that LensCap would improve their confidence while using untrusted AR apps.* They already feel safe when LensCap functioning logo is displayed and even safer when notification banners prompted for asking their permissions, with one participant "extremely" liking this setting. Furthermore, all participants think the Time-of-Collection info provided by LensCap is useful.

8 RELATED WORK

Previous works tried to protect user's privacy from various aspects, including protecting visual data, information flow control, as well as isolation and compartmentalization.

Protecting visual data Many previous works attempt to deprive untrusted vision applications from accessing the whole camera frame [2, 41, 50]. Jana *et al.* introduced the Darkly system [28] to address the threat of data over-collection and aggregation in untrusted third-party vision applications. In Darkly, camera frames are turned into opaque references and untrusted vision applications can only dereference them through trusted library APIs. Similarly, the Oculus Quest camera system [1] directly prohibits apps from accessing the passthrough camera. Instead, developers are only able to utilize camera poses, controller poses, and hand poses. These types of works might limit some vision apps that do require to work on objects and features of a camera frame directly and render the results. In [46] and [47], Raval *et al.* provided tools to give AR users finer-granularity control over their camera frames. AR users are able to define the part of camera frames that can be seen by untrusted vision applications. However, these tools might restrict AR experiences because AR applications need to provide the whole camera frame for virtual object overlay. Lebeck *et al.* introduced the Arya platform [37] to address the privacy and security risks in visual output caused by malicious or buggy untrusted third-party AR apps. Arya equips a trusted output module together with a set of output policies monitoring and filtering AR output in between the output device and the untrusted application. Unlike our threat model, some works focus on securing AR output [36, 49]; they could be integrated with LensCap to enable further secured AR applications, e.g., by implementing security policies in the network process to protect the screen overlay.

Information flow control Information flow control provides users with more control and visibility over how their private data is used in third-party apps [12, 21, 30, 31, 33, 34, 38, 40, 64, 65]. Enck *et al.* introduced TaintDroid [16] to taint, analyze, and track user's sensitive information at the granularity of variables, messages between applications, native methods, and files. TaintDroid is able to dynamically track those tainted data and identify how they are impacting other data that might cause data leakage. Arzt *et al.* presented a static taint-analysis system FlowDroid [7] to address data leakage in malicious applications. FlowDroid can model the complete lifecycle of an Android application and precisely monitor contexts, flows, fields, and objects with affordable performance overhead through on-demand alias analysis. Fernandes *et al.* presented FlowFence [19] to protect user data in IoT application frameworks. FlowFence first separates the operations on sensitive data into quarantined module. Then, it requires app developers to declare intentions for the dataflow. Otherwise, undeclared dataflows are prohibited from getting out of the quarantined modules. Wang *et al.* proposed LeakDoctor [62], a system that automatically detects privacy disclosure and determines if those privacy disclosures are

necessary for functionalities of the app. These types of works cannot be directly applied to protect visual data not only because of the cost for operating on large amounts of sensory data, but also due to a lack of tools and methods to tag camera frames based on intention. However, we anticipate that integrating static analysis into LensCap would enhance its ability by further pre-screening any privacy threats possibly injected by malicious app developers at the split-process boundary.

Isolation and compartmentalization Isolation and compartmentalization are adopted in both hardware and software domains to separate the execution of untrusted code from others [9, 11, 27, 29, 43, 55, 66]. Herbster *et al.* proposed Privacy Capsules [23] which also targets on protecting the leakage of user information through untrusted third-party applications. Privacy Capsules enforces applications to first execute in the unsealed phase in which the application has no access to the sensitive input but full access to the network resource, and then in the sealed phase in which the application gains access to the sensitive input but losing the capability of network communications. Raval *et al.* in [45] proposed to isolate plugins from the application as an individual app. It allows users to mediate resource requests made by apps which further enables more flexible authorizations to them. Dawound *et al.* in [15] also pointed out the necessity of application compartmentalization to protect privacy. DroidCap [15], a system that associates each IPC object with permissions for capability-based access control, could be integrated with our work for capability-based access control between split processes. Kilpatrick introduced Privman [61] as a C library for partitioning applications in UNIX environment to ease developer's burden when developing partitioned applications. In Privman, developers need to separate their applications into a privilege server process and a main application process, in which the main application process only has limited privileges. Apart from software solutions, Intel introduced Software Guard Extensions (SGX) [14] to allow users to define private regions of memory for secured execution, which further inspired tons of security and privacy works [52, 56]. However, none of these work explored integrating isolation and compartmentalization into the development flow of AR applications. Our system is largely inspired by the idea of app compartmentalization to isolate potentially malicious behavior from accessing visual data streams.

9 LIMITATIONS AND FUTURE WORK

AR library certification A significant limitation is that LensCap only provides services for verified third-party library APIs. In this paper, we demonstrate our system by implementing it around the Google ARCore library, though the implementation is generic to other libraries such as Apple's ARKit. For untrusted vision libraries, security experts could certify their operations through a scrutiny of source code. Then, a validated vision library could be signed to grant flexible access to the data handlers for network communications. We will explore this aspect in our future work towards a more comprehensive security solution.

Automatic application partitioning The split-process paradigm currently requires developers to rethink app logic. However, a deeper integration of LensCap into AR development frameworks could enable the compilation process to make decisions of where functionality moves across the boundary for performance and efficiency, while protecting visual data according to a user's wishes. An intelligent solution may even be able to dynamically migrate tasks between the visual and network processes subject to the contextual situation to optimize efficient operation. In future works, we could enable game engine compilers to bring automatic split-process partitioning to AR app development practices.

Scalability for verifying cloud services The LensCap system could also be extended to assist in securing visual offloading to cloud services. For example, we could verify that user data is sent to a cloud service associated with a verified URL address. Further verification would require certification that the cloud service only computes/stores the expected user data but does not send information to other untrusted third-party entities. To do so, LensCap could integrate with other works that protect user data against malicious cloud services and secure content sharing in multi-user collaborative AR apps [51, 63]. SAFE [60], as an example, equips a set of modules including an OS, a runtime, and proxy to enforce user policies in cloud services such that user data can only be released to another SAFE system or a system allowed by SAFE policies. Our system could run underneath this type of work as a trusted OS and app framework to provide strict control over user data that goes to unverified cloud services, while providing more latitude to trusted cloud services.

The adoption of LensCap development framework LensCap is currently implemented in Android and Unreal Engine environment. We are actively working on extending LensCap to other platforms such as iOS and game engines like Unity. To further ease AR developer's burden, besides automatic application partitioning, we would like to build a community of support such that developers can find help and get trained to develop LensCap-adopted, privacy-protected, and user-trusted AR apps. In addition, we would like to invite operating system vendors to participate in the development of LensCap such that verification can be imposed before AR apps are uploaded to app stores.

10 CONCLUSION

In this paper, we introduce LensCap, a split-process app development framework to protect user's visual privacy in cloud-based AR apps. LensCap isolates the processing of camera frames into a distinct visual process, meanwhile maintaining the cloud communication through another network process, with the data transactions between split processes monitored and shown to users for approval at a fine granularity. We prototype LensCap as an Android library that could be integrated into the AR development flow of Unreal Engine as a plugin. We evaluate LensCap in five UE projects developed for Android platforms. Results collected from the performance evaluation together with an interview-based user study demonstrate that visual privacy could be preserved and user confidence could be improved with LensCap split-process access control implemented in untrusted AR apps, without any noticeable performance penalty.

Acknowledgment We sincerely thank the anonymous shepherd for shepherding the final version of this paper and all of the valuable comments given by the anonymous reviewers.

REFERENCES

[1] Facebook Technologies, LLC. 2021. Mixed Reality Capture. https://developer.oculus.com/documentation/native/pc/dg-mrc/?locale=en_US. (2021).

[2] Paarijaat Aditya, Rijurekha Sen, Peter Druschel, Seong Joon Oh, Rodrigo Benenson, Mario Fritz, Bernt Schiele, Bobby Bhattacharjee, and Tong Tong Wu. 2016. I-Pic: A Platform for Privacy-Compliant Image Capture. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '16)*. ACM, New York, NY, USA, 235–248. https://doi.org/10.1145/2906388.2906412

[3] Android Developer. 2021. Media Framework Hardening. https://source.android.com/devices/media/framework-hardening. (2021).

[4] Android Developers. 2021. Android Interface Definition Language (AIDL). https://developer.android.com/guide/components/aidl. (2021).

[5] Android Developers. 2021. Permissions overview. https://developer.android.com/guide/topics/permissions/overview. (2021).

[6] Android Developers. 2021. Secure an Android Device. https://source.android.com/security. (2021).

[7] 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, 259âĂŞ269. https://doi.org/10.1145/2594291.2594299

[8] Gina Ashe. 2017. What Mary MeekerâĂŹs Internet Trends Report Means for the State of In-Store. https://blog.thirdchannel.com/mind-the-store. (2017).

[9] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp Von Styp-Rekowsky. 2015. Boxify: Full-Fledged App Sandboxing for Stock Android. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*. USENIX Association, USA, 691âĂŞ706.

[10] Bingkun Guo. 2014. iOS Security. https://www.cse.wustl.edu/~jain/cse571-14/ftp/ios_security/index.html. (2014).

[11] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry. 2011. Practical and lightweight domain isolation on Android. *Proceedings of the ACM Conference on Computer and Communications Security* (10 2011). https://doi.org/10.1145/2046614.2046624

[12] Supriyo Chakraborty, Chenguang Shen, Kasturi Rangan Raghavan, Yasser Shoukry, Matt Millar, and Mani Srivastava. 2014. ipShield: A Framework For Enforcing Context-Aware Privacy. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 143–156. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/chakraborty

[13] Jintai Chen, Biwen Lei, Qingyu Song, Haochao Ying, Danny Z. Chen, and Jian Wu. 2020. A Hierarchical Graph Network for 3D Object Detection on Point Clouds. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[14] Victor Costan and Srinivas Devadas. Intel SGX Explained. (????).

[15] Abdallah Dawoud and Sven Bugiel. 2019. DroidCap: OS Support for Capability-based Permissions in Android. In *NDSS Symposium 2019*. https://publications.cispa.saarland/2818/

[16] 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 *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 393–407. http://dl.acm.org/citation.cfm?id=1924943.1924971

[17] Epic Games, Inc. 2021. Unreal Engine. https://www.unrealengine.com/en-US/. (2021).

[18] FACEBOOK. 2021. Introducing Project Aria. https://about.fb.com/realitylabs/projectaria/. (2021).

[19] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. 2016. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 531–548. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/fernandes

[20] Google AR. 2021. Google ARCore SDK for Unreal. https://github.com/google-ar/arcore-unreal-sdk. (2021).

[21] J. Grubert, T. Langlotz, S. Zollmann, and H. Regenbrecht. 2017. Towards Pervasive Augmented Reality: Context-Awareness in Augmented Reality. *IEEE Transactions on Visualization and Computer Graphics* 23, 6 (2017), 1706–1724.

[22] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2014. Towards Wearable Cognitive Assistance. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '14)*. Association for Computing Machinery, New York, NY, USA, 68âĂŞ81. https://doi.org/10.1145/2594368.2594383

[23] 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 (MobiSys '16)*. ACM, New York, NY, USA, 399–411. https://doi.org/10.1145/2906388.2906409

[24] Jinhan Hu, Alexander Shearer, Saranya Rajagopalan, and Robert LiKamWa. 2019. Banner: An Image Sensor Reconfiguration Framework for Seamless Resolution-Based Tradeoffs. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '19)*. Association for Computing Machinery, New York, NY, USA, 236âĂŞ248. https://doi.org/10.1145/3307334.3326092

[25] Jinhan Hu, Jianan Yang, Vraj Delhivala, and Robert LiKamWa. 2018. Characterizing the Reconfiguration Latency of Image Sensor Resolution on Android Devices. In *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications (HotMobile '18)*. Association for Computing Machinery, New York, NY, USA, 81âĂŞ86. https://doi.org/10.1145/3177102.3177109

[26] A. âĂIJ. Huang. 2020. Betrusted: Improving Security Through Physical Partitioning. *IEEE Pervasive Computing* 19, 2 (2020), 13–20. https://doi.org/10.1109/MPRV.2020.2966190

[27] 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, 1037âĂŞ1049. https://doi.org/10.1145/3133956.3134064

[28] Suman Jana, Arvind Narayanan, and Vitaly Shmatikov. 2013. A Scanner Darkly: Protecting User Privacy from Perceptual Applications. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, Washington, DC, USA, 349–363. https://doi.org/10.1109/SP.2013.31

[29] Jk Jensen, Jinhan Hu, Amir Rahmati, and Robert LiKamWa. 2019. Protecting Visual Information in Augmented Reality from Malicious Application Developers *(WearSys '19)*. Association for Computing Machinery, New York, NY, USA, 23âĂŞ28. https://doi.org/10.1145/3325424.3329659

[30] Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. 2013. Run-Time Enforcement of Information-Flow Properties on Android. In *Computer Security – ESORICS 2013*, Jason Crampton, Sushil Jajodia, and Keith Mayes (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 775–792.

[31] Haojian Jin, Minyi Liu, Kevan Dodhia, Yuanchun Li, Gaurav Srivastava, Matthew Fredrikson, Yuvraj Agarwal, and Jason I. Hong. 2018. Why Are They Collecting My Data? Inferring the Purposes of Network Traffic in Mobile Apps. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 2, 4, Article 173 (Dec. 2018), 27 pages. https://doi.org/10.1145/3287051

[32] Ricardo Jota, Albert Ng, Paul Dietz, and Daniel Wigdor. 2013. How Fast is Fast Enough?: A Study of the Effects of Latency in Direct-touch Pointing Tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 2291–2300. https://doi.org/10.1145/2470654.2481317

[33] Thivya Kandappu, Archan Misra, Shih-Fen Cheng, Randy Tandriansyah, and Hoong Chuin Lau. 2018. Obfuscation At-Source: Privacy in Context-Aware Mobile Crowd-Sourcing. 2, 1, Article 16 (March 2018), 24 pages. https://doi.org/10.1145/3191748

[34] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information Flow Control for Standard OS Abstractions. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. ACM, New York, NY, USA, 321–334. https://doi.org/10.1145/1294261.1294293

[35] Butler W. Lampson. 1973. A Note on the Confinement Problem. *Commun. ACM* 16, 10 (Oct. 1973), 613âĂŞ615. https://doi.org/10.1145/362375.362389

[36] K. Lebeck, K. Ruth, T. Kohno, and F. Roesner. 2017. Securing Augmented Reality Output. In *2017 IEEE Symposium on Security and Privacy (SP)*. 320–337.

[37] K. Lebeck, K. Ruth, T. Kohno, and F. Roesner. 2018. Arya: Operating System Support for Securely Augmenting Reality. *IEEE Security Privacy* 16, 1 (January 2018), 44–53. https://doi.org/10.1109/MSP.2018.1331020

[38] S. M. Lehman and C. C. Tan. 2017. PrivacyManager: An access control framework for mobile augmented reality applications. In *2017 IEEE Conference on Communications and Network Security (CNS)*. 1–9.

[39] Haomin Liu, Chen Li, Guojun Chen, Guofeng Zhang, Michael Kaess, and Hujun Bao. 2017. Robust Keyframe-based Dense SLAM with an RGB-D Camera. *CoRR* abs/1711.05166 (2017). arXiv:1711.05166 http://arxiv.org/abs/1711.05166

[40] Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. 2016. Practical DIFC Enforcement on Android. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 1119–1136. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/nadkarni

[41] K. Olejnik, I. Dacosta, J. S. Machado, K. Huguenin, M. E. Khan, and J. Hubaux. 2017. SmarPer: Context-Aware and Automatic Runtime-Permissions for Mobile Devices. In *2017 IEEE Symposium on Security and Privacy (SP)*. 1058–1076.

[42] James Paine. 2020. 10 Real Use Cases for Augmented Reality: AR is set to have a big impact on major industries. https://www.inc.com/james-paine/10-real-use-cases-for-augmented-reality.html. (2020).

[43] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. 2012. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS '12)*. Association for Computing Machinery, New York, NY, USA, 71âĂŞ72. https://doi.org/10.1145/2414456.2414498

[44] Siddhant Prakash, Alireza Bahremand, Linda D. Nguyen, and Robert LiKamWa. 2019. GLEAM: An Illumination Estimation Framework for Real-time Photorealistic Augmented Reality on Mobile Devices. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '19)*. ACM, New York, NY, USA, 142–154. https://doi.org/10.1145/3307334.3326098

[45] Nisarg Raval, Ali Razeen, Ashwin Machanavajjhala, Landon P. Cox, and Andrew Warfield. 2019. Permissions Plugins as Android Apps *(MobiSys '19)*. Association for Computing Machinery, New York, NY, USA, 180âĂŞ192. https://doi.org/10.1145/3307334.3326095

[46] Nisarg Raval, Animesh Srivastava, Kiron Lebeck, Landon Cox, and Ashwin Machanavajjhala. 2014. MarkIt: Privacy Markers for Protecting Visual Secrets. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication (UbiComp '14 Adjunct)*. ACM, New York, NY, USA, 1289–1295. https://doi.org/10.1145/2638728.2641707

[47] Nisarg Raval, Animesh Srivastava, Ali Razeen, Kiron Lebeck, Ashwin Machanavajjhala, and Lanodn P. Cox. 2016. What You Mark is What Apps See. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '16)*. ACM, New York, NY, USA, 249–261. https://doi.org/10.1145/2906388.2906405

[48] 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)*. USENIX Association, Santa Clara, CA, 603–620. https://www.usenix.org/conference/usenixsecurity19/presentation/reardon

[49] Talia Ringer, Dan Grossman, and Franziska Roesner. 2016. AUDACIOUS: User-Driven Access Control with Unmodified Operating Systems. 204–216. https://doi.org/10.1145/2976749.2978344

[50] Franziska Roesner, David Molnar, Alexander Moshchuk, Tadayoshi Kohno, and Helen J. Wang. 2014. World-Driven Access Control for Continuous Sensing. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 1169–1181. https://doi.org/10.1145/2660267.2660319

[51] Kimberly Ruth, Tadayoshi Kohno, and Franziska Roesner. 2019. Secure Multi-User Content Sharing for Augmented Reality Applications. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 141–158. https://www.usenix.org/conference/usenixsecurity19/presentation/ruth

[52] Jose Rodrigo Sanchez Vicarte, Benjamin Schreiber, Riccardo Paccagnella, and Christopher W. Fletcher. 2020. Game of Threads: Enabling Asynchronous Poisoning Attacks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 35âĂŞ52. https://doi.org/10.1145/3373376.3378462

[53] D. Schmalstieg and G. Hesina. 2002. Distributed applications for collaborative augmented reality. In *Proceedings IEEE Virtual Reality 2002*. 59–66.

[54] A. Shaikh, L. Nguyen, A. Bahremand, H. Bartolomea, F. Liu, V. Nguyen, D. Anderson, and R. LiKamWa. 2019. Coordinate: A Spreadsheet-Programmable Augmented Reality Framework for Immersive Map-Based Visualizations. In *2019 IEEE International Conference on Artificial Intelligence and Virtual Reality (AIVR)*. 134–1343. https://doi.org/10.1109/AIVR46125.2019.00028

[55] 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. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/shekhar

[56] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 955âĂŞ970. https://doi.org/10.1145/3373376.3378469

[57] Pieter Simoens, Yu Xiao, Padmanabhan Pillai, Zhuo Chen, Kiryong Ha, and Mahadev Satyanarayanan. 2013. Scalable Crowd-Sourcing of Video from Mobile Devices. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '13)*. Association for Computing Machinery, New York, NY, USA, 139âĂŞ152. https://doi.org/10.1145/2462456.2464440

[58] P. Speciale, J. L. SchÃűnberger, S. B. Kang, S. N. Sinha, and M. Pollefeys. 2019. Privacy Preserving Image-Based Localization. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 5488–5498.

[59] Square, Inc. 2021. OkHttp. https://square.github.io/okhttp/. (2021).

[60] Adriana Szekeres, Irene Zhang, Katelin Bailey, Isaac Ackerman, Haichen Shen, Franziska Roesner, Dan R. K. Ports, Arvind Krishnamurthy, and Henry M. Levy. 2020. Making Distributed Mobile Applications SAFE: Enforcing User Privacy Policies on Untrusted Applications with Secure Application Flow Enforcement. (2020). arXiv:cs.CR/2008.06536

[61] Freenix Track and Douglas Kilpatrick. 2003. Privman: A Library for Partitioning Applications. (2003).

[62] Xiaolei Wang, Andrea Continella, Yuexiang Yang, Yongzhong He, and Sencun Zhu. 2019. LeakDoctor: Toward Automatically Diagnosing Privacy Leaks in Mobile Applications. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 3, 1, Article 28 (March 2019), 25 pages. https://doi.org/10.1145/3314415

[63] Guowen Xu, Hongwei Li, Shengmin Xu, Hao Ren, Yinghui Zhang, Jianfei Sun, and Robert H. Deng. 2020. Catch You If You Deceive Me: Verifiable and Privacy-Aware Truth Discovery in Crowdsensing Systems. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS '20)*. Association for Computing Machinery, New York, NY, USA, 178âĂŞ192. https://doi.org/10.1145/3320269.3384720

[64] Yuanzhong Xu and Emmett Witchel. 2015. Maxoid: Transparently Confining Mobile Applications with Custom Views of State. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 26, 16 pages. https://doi.org/10.1145/2741948.2741966

[65] Zhi Xu and Sencun Zhu. 2015. SemaDroid: A Privacy-Aware Sensor Management Framework for Smartphones. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY '15)*. Association for Computing Machinery, New York, NY, USA, 61âĂŞ72. https://doi.org/10.1145/2699026.2699114

[66] 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, 9âĂŞ18. https://doi.org/10.1145/2523649.2523652