Erebus: Access Control for Augmented Reality Systems

Yoonsang Kim* and Sanket Goutam*, Amir Rahmati, Arie Kaufman
Stony Brook University
{yoonsakim, sgoutam, amir, ari}@cs.stonybrook.edu

Abstract

Augmented Reality (AR) is widely considered the next evolution in personal devices, enabling seamless integration of the digital world into our reality. Such integration, however, often requires unfettered access to sensor data, causing significant overprivilege for applications that run on these platforms. Through analysis of 17 AR systems and 45 popular AR applications, we explore existing mechanisms for access control in AR platforms, identify key trends in how AR applications use sensor data, and pinpoint unique threats users face in AR environments. Using these findings, we design and implement Erebus, an access control framework for AR platforms that enables fine-grained control over data used by AR applications. Erebus achieves the principle of least privileged through the creation of a domain-specific language (DSL) for permission control in AR platforms, allowing applications to specify data needed for their functionality. Using this DSL, Erebus further enables users to customize app permissions to apply under specific user conditions. We implement Erebus on Google's ARCore SDK and port five existing AR applications to demonstrate the capability of Erebus to secure various classes of apps. Performance results using these applications and various microbenchmarks show that Erebus achieves its security goals while being practical, introducing negligible performance overhead to the AR system.

1 Introduction

The last decade has seen a growing trend in the application of Augmented Reality (AR) and Mixed Reality (MR) systems. These systems allow applications to create immersive experiences for user interaction, enriching and streamlining tasks for their users. To achieve this, AR systems capture the user's surroundings using sensor input, such as video, depth, location, and audio, and overlay virtual content on live camera feed through devices such as smartphones or Head-Mounted Displays (HMDs). Such HMDs in the form of "glasses" are often seen as the next logical evolution of personal devices, and major companies including Google, Apple, Meta, and Amazon are known to be actively developing them for mainstream use [25].

To achieve their functionality, AR devices rely on *Perceptual sensing*. Perceptual sensing is the ability of a hardware device or a software application to leverage cameras and other sensors to continuously observe their physical environment. Mobile phones, IoT devices, and gaming devices (e.g., Microsoft Kinect, Nintendo Wii) are some of the hardware devices that use perceptual sensing to support user inputs in the form of gestures or voice commands. Lately, with the growing popularity of augmented reality, many software applications, such as Pokemon Go, Ikea Place app [32] have also become prevalent. These applications fundamentally leverage continuous video input to recognize the physical objects in the user's environment and overlay virtual information on top of it to enhance their experience.

Consider the Ikea Place app, which requires the user to scan their living room, choose an item of furniture, and place it virtually anywhere using their smartphone — allowing users to experiment with furniture even before buying them. In order to achieve this functionality, the Ikea app requires continuous access to the user's video feed, effectively recording everything in the user's environment. The user intends to use the app only to scope out the living room area, however the app may inadvertently record sensitive information, such as credit card numbers, contents on computer displays, etc., present in its environment. Another case could be that the user may unknowingly use the application in a sensitive location, such as at work or in a locker room, thereby compromising the privacy of their coworkers or bystanders. In existing AR frameworks, neither the frameworks themselves nor the application provides any control to the user to restrict such over-collection of sensory information.

This form of over-privilege is further exacerbated by the current design of the manifest permissions model in the underlying operating systems [2]. For example, every Android app uses a `AndroidManifest.xml` file, which declares the permissions and the hardware or software features required by the app to the Android OS. Android does allow users to selectively grant permissions to the app, however, their choices remain

*These authors contributed equally to this work.

persistent, regardless of whether apps are in the foreground or background. Android 10 attempted to provide more granular permissions control by introducing Tristate location permissions [3], wherein users are provided with the additional choice to restrict device location access only while the app is in use. Unfortunately, a similar access control granularity is not yet available for other sensory inputs. Additionally, this manifest model is not a viable option for extending granular access control to the complex use cases posed by AR applications.

To provide a holistic solution to these challenges, we present Erebus, a language-based access control framework for Augmented Reality systems that can coexist with the existing manifest model and extends granular permissions control to the user in the form of a novel *policy specification language*. We extend the primitives of user-driven access control [10, 50] whereby permission granting is built into decisions made by the user within the application context, rather than permissions being set as an afterthought by the developer in the manifest. Our goal is to allow the users to define the *What, When,* and *Where* permissions of the sensory input for an untrusted app. For example, in the case of the Ikea app, Erebus will allow users to restrict the application to only provide plane detection in the home environment at certain times.

We implement Erebus on Google's ARCore SDK, a popular software development kit that allows for augmented reality applications, and evaluate it across five representative AR applications: (1) an AR navigation app, (2) an AR remote maintenance application, (3) a Face filter app, (4) an AR game, and (5) an AR shopping app. We demonstrate that Erebus can protect users against over-privileged applications while providing them with tools to seamlessly modify the information shared with these applications. Our experiments show that Erebus can provide fine-grained access control across AR platform while incurring modest overhead.

In summary, this paper makes the following contributions:

- We perform a study of 17 AR devices Access Control systems (Section 2) and examined how 45 popular AR applications use sensor data to achieve their functionality (Section 3). We release our app and device surveys to motivate future research.[1]
- Based on our findings, we design and implement Erebus, an access control framework for AR systems that provides users with fine-grained control over how AR applications access sensor data (Section 5).
- We implement Erebus on Google ARCore SDK and provide a comprehensive case study of five AR applications, representing the diversity of existing applications in the AR space (Section 6). Our results show that Erebus is able to enforce fine-grained access control across the five applications, while incurring minimum overhead. We open-source Erebus's implementation to motivate future research and adoption.[1]

[1] https://github.com/Ethos-lab/erebus-AR_access_control

Figure 1: AR systems today are being developed either as a standalone unit with all sensors and computation present onboard, or with a companion device such as a smartphone to offload computational tasks.

2 Background

2.1 Augmented Reality Systems

AR devices and applications are increasingly becoming a part of our digital lives. While mobile devices (through apps such as Pokemon Go, Google Maps AR Live View, and Snapchat) currently cover the most significant market segment of AR applications, they only offer rudimentary forms of AR that act as an entry point for many users. More advanced consumer AR systems involve head-mounted displays and AR smart glasses that provide a hands-free immersive experience. These devices are being marketed as the next major evolution in personal technology that will augment, if not completely replace, smartphones as users' preferred personal devices. A recent flurry of activity in the form of patent applications and prototypes, from companies such as Apple [4, 7], Google [6], Meta [43], and others [37, 60, 62] have showcased some of the emerging designs in this space. Based on these trends, we identify two key design philosophies that govern the emerging AR systems (Figure 1):

Standalone Unit. The first design form designates the AR system as a self-contained unit, incorporating different sensors, processing capability, and a virtual display on a single device. Microsoft HoloLens [44] is an example of such a system. While there are definite advantages in such all-in-one solutions, design considerations such as weight and battery capacity limit the processing capability of these devices.

Companion Device. The second design form envisions AR systems as wearable companion devices connected to existing smartphones or PCs. Such design enables AR devices to leverage the extensive computational power of modern smartphones and computers, allowing wearable glasses to focus on providing the display and additional sensors. Lenovo's ThinkReality smart glasses [40] is an example of this type which tethers to the user's laptop or smartphone to create a customized, expanded personal workspace. Extending on this companion device ideology, the heavily anticipated Apple Glasses [7] is rumored to augment the iPhone and the Apple ecosystem to create immersive user experiences [4, 5]. These early patents suggest a move toward building an inter-connected ecosystem

of wearables, where the AR functionality is informed by multiple devices, the user's smartphone acts as the central hub for processing information, and the AR glasses as the display unit.

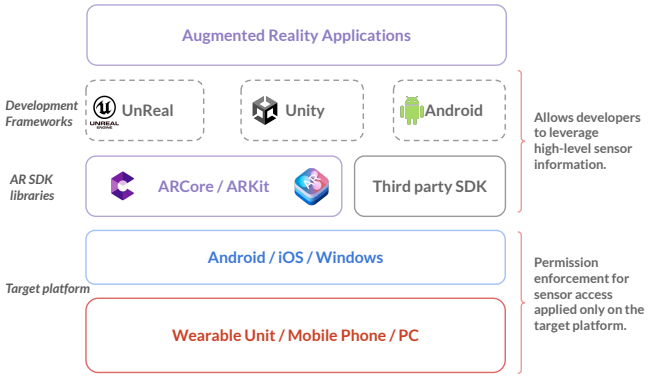

Figure 2: AR application developers use different frameworks and libraries that provide high-level functional APIs to access sensor information. These frameworks, however, do not provide any permission enforcement and default to the target platform access control.

2.2 Access Control in AR systems

To build immersive applications, developers use different frameworks and software libraries that allow seamless integration with the device. Figure 2 presents a high-level view of the computing stack used to develop AR applications. This computing stack applies to all platforms, including Smart Glasses, Smartphones, and Desktops. This cross-platform support is aided by the availability of AR SDK libraries (such as ARCore and ARKit), which provide high-level native APIs that interface with device sensors. These APIs allow application developers to derive meaningful content without worrying about interfacing with the sensor. However, considering these AR systems mostly include audio-visual and gesture-monitoring sensors, a lack of proper permission management allows the applications to abuse sensor data usage policies. We surveyed how the popular device manufacturers approach access control in their design; our results, presented in Table 1, highlights access control methods observed in existing AR platforms:

- **Developer Specified App Manifest:** Most manufacturers of AR headsets use an Android-based OS. Similar to other Android apps, these apps declare the policy specification using the app manifest file [2], which requires an explicit declaration of all the sensors to be accessed by each app. Listing 1 provides an example of such a manifest file. This form of policy specification only supports a coarse-grained gate-keeping mechanism, allowing either full access to the sensor or none at all.
- **Device Admin Policy Specification:** Microsoft HoloLens, which runs on Windows 10, provides developers and device administrators with hardware restrictions to control the availability of a Camera, Microphone, and Bluetooth interfaces, among others. Developers use an app manifest similar to Android to declare the app's capabilities. However, the policy enforcement is set by the device administrator using the Configuration Service Provider (CSP) module [14]. Administrators can update app access to a sensor by modifying the associated CSP tag. For example, in order to allow Windows apps to access the Camera interface, the device admin first needs to set the *"LetAppsAccessCamera"* CSP tag which supports a threefold setting — (1) User in control, (2) Force Allow, or (3) Force Deny. The admin can further specify which specific apps should be denied access using the tag *"LetAppsAccessCamera_ForceDenyTheseApps"*.

```
<uses-feature android:name="android.hardware.camera"
  android:required="true" />
<uses-permission android:name="android.permission.record_audio"
  android:required="true" />
<uses-feature android:name="android.hardware.location.GPS"
  android:required="true" />
<uses-feature android:name="android.hardware.sensor.heartrate"
  android:required="true" />
```

Listing 1: Developers declare sensor access permissions in AndroidManifest.xml file which only allows coarse-grained permission control. Users can only choose between "Allow access", "Allow access while the app is in use", and "Deny access".

```
let allTypes = Set([HKObjectType.workoutType(),
        HKObjectType.quantityType(forIdentifier: .restingHeartRate)!,
        HKObjectType.quantityType(forIdentifier: .bodyTemperature)!,
        HKObjectType.quantityType(forIdentifier: .bloodPressure)!,
        HKObjectType.quantityType(forIdentifier: .heartRate)!])
```

Listing 2: A fitness app using Apple Healthkit framework that needs access to the heart rate sensor data needs to declare explicitly all data types it needs. Users are prompted to Allow / Disallow individual data type permissions through the Settings interface.

Beyond these two methods, there is a third access control method that while not used by current AR systems, is widely used in other wearable devices:

- **Authorization per Data-type Access:** Listing 2 presents the access control method currently being used in smartwatches, using Apple Healthkit framework [13] as the example. These frameworks are designed for wearable devices with persistent access to users' health data. The access control is thus applied to each data type that an application can request, ranging from heart-rate samples to the user's exact location. Developers must explicitly declare the data requirements in the application code, and users are prompted each time an application requests new permissions. While more fine-grained than previous methods when dealing with processed medical data generated by the wearable platform, this approach fails to provide protection beyond binary access control when dealing with direct sensor data such as exact location.

A singular drawback across all three access control methods discussed so far makes them unfit for AR devices: All existing mechanisms assume a gate-keeping form of enforcement where an app can retain unfettered permissions once the user has granted them. This creates a particularly challenging design problem where the concern over granting perpetual access to audio-visual sensors and users' location in an application may inhibit many genuine use cases of smart glasses.

Table 1: Survey of AR smart glasses from 17 different manufacturers showcase that these devices are designed to either default to the target platforms access control method or provide none of their own.

AR Device Type	Device Name	Platform	Access Control Mechanism
Standalone Wearable	Meta Quest 2 [43]	Android	App Manifest
	Microsoft HoloLens 2 [44]	Windows	App Manifest, Policy CSP
	Magic Leap 2 [16]	Android	App Manifest
	Google Glass Enterprise [23]	Android	App Manifest
	ThirdEye X2 MR Smart Glasses [22]	Android	App Manifest
	Vuzix Blade AR [70]	Android	App Manifest
	Snap Spectacles [67]	Android	App Manifest
	Raptor AR Headset [19]	Android, iOS	No AC mechanism
	Kopin Solos [36]	Android, iOS	No AC mechanism
	Xiaomi Smart Glasses [68]	Android	No information available
With a Companion Device	Lenovo ThinkReality A3 [40]	Android, Windows	Depends on target platform
	Epson Moverio [18]	Android, Windows	Depends on target platform
	Toshiba dynaEdge [63]	Windows	No AC mechanism
	Rokid Air Pro [52]	Android, iOS	App Manifest
	NReal Light [46]	Android	No AC mechanism
	Viture One [69]	Android	No information available
	Dream Glass Flow [66]	Android, iOS	No information available

Table 2: Camera is the primary sensor required for all AR functionality. Some apps also request access to Location and Microphone sensors, but their usage is mostly supplemental to core AR functions.

Sensors Access Required	# of apps (out of 42 [2])
Camera	14
Camera, Location	7
Camera, Microphone	8
Camera, Location, Microphone	10

3 How Do AR Applications Use Sensor Data?

3.1 Survey of the Use of Sensor Data

To better comprehend the risks AR users face, we first need to understand how current AR applications access and use sensor data. To do this, we examined the top 45 applications across Android and iOS app stores. Table 2 presents the result of our survey: all AR applications primarily focus on extracting and using visual semantics from the real-world, requesting Camera permissions by default. We noticed some applications also request access to Location and Microphone sensors, but their usage is mostly supplemental to the app's core functionalities. Additionally, access to all sensor data except the Camera is regulated via the OS, while the visual semantics is provided by the AR libraries (ARCore or ARKit) alone.

To understand how these applications use sensor data, we extracted the AR functionalities used by each of these applications. Prior work on the privacy of augmented reality systems [29, 33, 34, 38, 41, 50, 51] has largely demonstrated how apps' access to the Camera sensor can result in over-privilege. We confirm this with our findings in Table 3 that the core functional requirement of AR applications can be satisfied with only a handful of semantic information obtained from the AR libraries. However, existing permission-granting mechanisms require developers to request full access to the Camera sensor irrespective of their requirements. This is not particularly surprising given that enforcement mechanisms, as discussed in Section 2.2, only offer a coarse-grained access control.

3.2 App Case Study

To demonstrate the level of over-privilege and associated risks AR apps pose with existing policy frameworks, we dive deeper with a case study of a popular application — IKEA Place.

An AR app such as IKEA Place [32], primarily allows users to visualize furniture or home decor elements in their surrounding space. In order to achieve this, the app first derives certain context from the user's surroundings by identifying flat surfaces, dimensions of the space, and lighting present around the room. Once obtained, it generates the virtual content (furniture) scaled to the user's surroundings and overlays it on the raw camera feed. IKEA Place was developed using ARKit, so all AR functions exist as user-level APIs that are packaged within the app.

3.2.1 Limitations in Existing Permissions Model

The IKEA Place app requests access to Camera and Location sensors. This information is communicated using user prompts by the permissions manager as shown in Figure 3.

[2] We excluded 3 iOS paid apps from this survey (out of 45) as Apple App Store doesn't disclose the app permissions without installation.

[3] AR libraries provide quite specific API classes that developers can leverage for building immersive applications. All of these functions leverage motion and visual sensors only. https://developers.google.com/ar/develop/fundamentals

Table 3: Survey of 45 AR apps show the mismatch in developer's requirement of sensor data and requested sensor access on the device.

AR functionality	# of apps (out of 45)	Sensor access requested	Functional description [3]
Raycasts	30	Camera	Perform raycasting to determine the correct placement of a 3D object in the scene.
Anchors	38	Camera	Use anchors to make virtual objects appear to stay in place in an AR scene.
Plane detection	25	Camera	Detects horizontal or vertical flat surfaces.
Face tracking	3	Camera	Allows detection of feature points that enable the app to automatically identify different regions of the detected face.
Object detection	8	Camera	Lets you build AR apps that can detect and augment 2D images in the user's environment, such as posters or product packaging.
Motion tracking	8	Camera, Location	Visual information is combined with inertial measurements from the device IMU to estimate the pose (position and orientation) of the camera relative to the world over time.

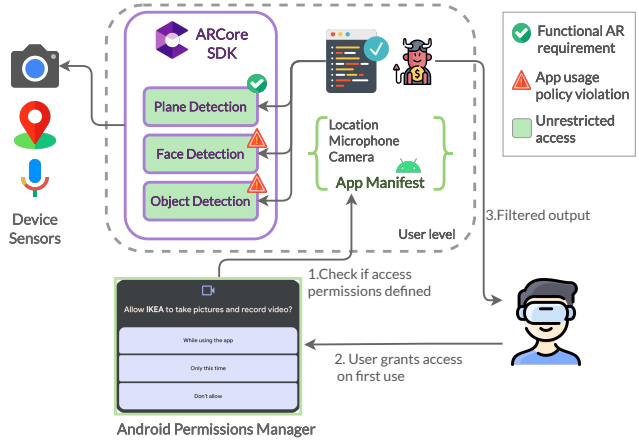

Figure 3: A reference model of application flow for an AR application.

Users can conditionally allow or block access to each sensor for individual apps; in practice, users typically grant all permissions giving continued access to apps.

Consider the data access paths shown in Figure 3 for an application to access sensor information. The IKEA Place app requires only a small subset of visual data, such as information about the plane surfaces in the user's surroundings. Furthermore, the app may have valid usage at specific locations only (e.g., at the user's Home and not at the Gym). However, current standards force developers to request full access to Camera, even though the only requirement is just Plane Detection. Users (and, in practice, the system) remain completely oblivious to the usage of the Camera sensor otherwise. The app may state only "Plane Detection" in its privacy policy but could use Face Detection or record the raw camera stream internally. All AR functionalities request access to the Camera, but some pose far greater privacy concerns than others.

4 Reimagining App Permissions

In order to support the development of next-generation AR systems, we need to envision a more appropriate access control model that regulates the risk from AR applications. We focus on third-party AR apps developed using various cross-platform frameworks that largely depend on AR libraries to create immersive visual content. We are specifically interested in limiting untrusted apps' access to visual semantics using the Camera sensor and introducing a least-privilege access control model for developing AR applications.

4.1 Threat Model

We assume that the AR application, either erroneously or maliciously, over-collects user-specific semantic information and send it to remote servers. Our case study in Section 3.2 and the data flow ①, shown in Figure 3, shows how an app can access AR functions that violate its functional requirement, allowing access to semantic information that developers do not disclose in the privacy policies. This implies that a malicious developer could masquerade as a benign app, such as a furniture-viewing app, request access to the user's camera, and surveil users' surrounding for sensitive information such as their credit card instead. Ad brokers may also leverage this to collect user-targeted data. It is not uncommon for mobile applications to include syndicated advertisements in their platforms for monetization. A malicious ad broker may trick an app developer into sharing users' raw camera feed for targeted advertising.

Although app markets often try to detect and remove such blatantly malicious apps, they are not particularly effective against such covert ways of collecting privacy-sensitive data. Even with proactive vetting mechanisms, millions of users will be affected by the time that such apps are identified and removed [58], and users are still required to manually uninstall them in many cases.

4.2 Least Privilege Sensor Access

In line with prior work on the security and privacy of AR systems [29, 33, 34, 41, 50, 51, 59], we assume an application is untrustworthy and may intentionally deviate from ideal scenarios to steal user semantics that it should not have access to (*i.e.,* visual semantics not required for its functional use cases). Our goal is to ensure the least-privilege sensor information exposure to an app governed solely by its functional requirement. Informed by our study of existing AR platforms and applications, we focus our threat model on minimizing two distinct types of over-privilege present in AR systems today:

Function-level over-privilege. Typically most applications use a small subset of functions from the AR library. Unfortunately,

under current frameworks, they retain unfettered access to all APIs at the user level regardless of whether they need that functionality in their application. This unregulated access can lead to unintentional or malicious sensitive user data leakage. The different scenarios we covered in our threat model in Section 4.1 leverage this form of over-privilege and gain control of privacy-sensitive data that apps should not have access to.

Attribute-level over-privilege. Even if applications are limited to the functions they specifically require, the context in which this access is provided may lead to over-privileged access to user information. An app that is specifically purposed for the work environment may be inappropriate for the home and vice versa. Similarly, most applications will be inappropriate to run in private spaces such as restrooms or changing rooms.

4.3 Design Goals

We posit that AR systems need to be developed keeping privacy risks from untrusted applications in mind. We identify the following design goals that any platform running these applications should satisfy.

G1 **Regulating direct access to sensors:** Apps should be restricted from accessing sensitive sensors, especially the Camera, directly. Since AR applications mostly depend on AR libraries to provide abstract semantic information from visual data, unfettered access to raw camera feed should be restricted.

G2 **Minimizing function-level over privilege:** Emerging systems should provide a policy framework that supports writing least privilege policies for AR apps. Apps should not be allowed to access APIs that violate their functional requirement. Developers should have the ability to specify an app requirement at the granularity of semantic usage (*e.g.,* "Only detects faces") and the system should enforce it as such.

G3 **Minimizing attribute-level over privilege:** Users of an AR platform should be able to review the policies and adjust the accesses based on their use cases and context. Therefore, the policy engine should be expressive and understandable to the end-user.

5 Erebus: Access Control Framework for AR

To address these design goals, we present **Erebus** — an access control framework designed for the emerging AR systems. The goal of Erebus is to provide a least-privilege access control model that developers can use to declare effective and usable policies for AR applications running on consumer devices. In doing so, we reimagine how software libraries could be developed to co-exist with the app manifest and enforce granular policy enforcement at the OS level instead of the user-space level.

We achieve this by first proposing a novel policy specification language for AR systems that allows developers to express the functional intent of an application in a more expressive form in comparison to existing methods. Based on this policy language, we implement Erebus, using the ARCore SDK on the Android platform to showcase how mobile OSs can implement this policy framework with existing AR libraries and allow transparency of an app AR function usage with the system.

5.1 Domain-Specific Language for AR Systems

A least-privilege access control model for AR systems should provide a way to achieve data-minimization over the sensor data — not only regulating full access to the sensors but also restricting the types of rich semantic information apps can derive. For example, a system that satisfies **G1** has to ensure that apps only receive the semantic data they require from the raw sensor stream. This means that the underlying system should have transparency into the the application's exact functional use case and enforce policies as such. Ideally, we could design a system that learns the exact requirement of an AR app from their source code and enforces policies accordingly, but understanding intent requires static and dynamic analysis techniques [24, 47] which is not feasible to implement on a user device.

We take inspiration on how to express functional intent by looking at rule specification in trigger-action platforms (TAPs). Popular TAPs, such as IFTTT [31], allows users to create simple rules that convey the intent of trigger-action functionality and have been widely adopted by users and developers alike. At its core, TAPs provide a very simplistic model of block-based programming [71] that lets even non-programmers create efficient automation policies. We leverage this principle in our language design to express functional intent of AR applications in the form of *If-This-Then-That* blocks, making policy definition more comprehensible (achieving **G3**), while also enforcing a data-minimization framework (achieving **G1** and **G2**) over sensor information.

5.1.1 Programming Model – Filter Codes for AR

IFTTT rules contain user-created code snippets, known as filter codes, where the set of required data is determined based on code behavior [31]. We define our policy language in a similar way. Consider an application that requests access to the Camera only to detect a QR code and perform some basic functionality using that, such as displaying a restaurant menu or launching an animation within an AR game. Functionally the only necessity for this application is a QR code image; thus, providing it with additional visual semantics makes it over-privileged. Furthermore, certain categories of AR applications may have valid use cases only in specific locations or certain time slots. The Ikea AR app, for instance, while appropriate for use at home, may raise privacy concerns if used in a Gym locker room. Existing access control frameworks, discussed in Section 2.2, do not support this level of granularity.

At the core of **Erebus** lies this expressive policy engine that defines such granular sensor data access rules. Table 4 provides a simplistic view of our policy grammar. We define user policy as a set of functions that access the Camera sensor data in some way. These functions can be any ARCore APIs or raw system calls that operate on any visual semantic information.

Access to the data returned by each function is governed by a set of rules or a singular action. Each rule is further defined as a conditional logic over a set of attributes satisfying which follows the action (Allow / Deny) that dictates whether the app can access the functionality. In our implementation, we restricted the set of attributes to {*FaceID, Location,* and *Time*} to show examples of the expressiveness of user policies based on these environmental attributes.

The core part of any rule is defined as a relation between an *attribute* and its corresponding *trusted_attribute*. Our design follows the existing models of access control in software systems that define conditional logic over attributes [15, 30]. We assume that the user has already defined a trusted value corresponding to environmental variable in the system. This basically implies that the system is aware of semantics such as "Home", "Evening", "Office hours", etc, and has identities for all users of the system with named tags (e.g., Device Owner, John Smith, etc). We consider the user (as recognized with their FaceID) as an environmental variable for policy specification because Android device policy controller (DPC) allows multiple people to share a single dedicated device [12], so it is only natural for the policy framework to allow defining policies that are specific to the user. Erebus compares the current user of the device with known users predefined in the system allowing access only if explicitly specified in the user policy — "Allow access to only John Smith". Furthermore, Erebus also acts as a data-minimizer by filtering out the specific semantic information requested by the app, as shown in Listing 3. Most existing AR applications are built with highly specific visual semantic dependency. Edutainment apps use Cameras to scan for QR codes, AR games derive environmental context by detecting flat surfaces, apps such as Snapchat only need semantic information about the user's facial features, and so on. We consider that an expressive policy framework should allow for such granular specification of rules so that the app receives only the necessary visual information.

5.1.2 Policies Based on Functional Use

We defined policy as a set of rules applied to a group of functions. These functions are essentially sets of APIs (either in ARCore or the system) that access the Camera sensor. We focus primarily on the Camera sensor since, as discussed in Section 3, AR applications unequivocally depend on visual semantics for most use cases. The challenge is determining what rules should be applied to which functions. The technical definition of system APIs and ARCore APIs may significantly differ between various platforms; however, their functionality remains more or less the same. AR Foundation, for example, is a development framework on Unity that uses the ARCore SDK. All the AR functions provided by AR Foundation are grouped based on their functionality and classified into distinct categories called Trackables [64]. In Erebus, we leverage this predefined classification of APIs to build our policies.

The core functional groups of APIs available for AR development include, but are not limited to, Plane Detection, Image Detection, Object Detection, and Location Tracking. Our survey of 45 AR apps, as discussed in Table 3, also suggests that apps mostly use these basic groups of functions together to create an AR experience for the user. The functional description of each app can be used to convey what kind of AR functions it uses "Pokemon Go detects flat surfaces and places a virtual Pokemon on top of it" suggests Plane detection, "Google Lens identifies text, images, objects, and landmarks in your photos" uses Image detection and Object detection. We leverage this understanding of ARCore functions and extend the definition of Trackables, as shown in Table 9, to group together sets of APIs that can extract similar semantic information.

Table 4: Policy specification grammar in Erebus. This allows apps to express granular policies in the form of filter codes that restrict app access to ARCore APIs based on environmental conditions and also allow writing rules for visual object extraction from raw camera feed.

⟨policy⟩	⊨	⟨function⟩∗ | ε
⟨function⟩	⊨	⟨func_name⟩{ ⟨rules⟩+}
		| ⟨action⟩
⟨rules⟩	⊨	⟨attributes⟩+;
		⟨trusted_attributes⟩+;
		⟨condition⟩+{⟨action⟩}
⟨condition⟩	⊨	if ⟨attributes⟩⟨relation⟩
		⟨trusted_attributes⟩
⟨attributes⟩	⊨	CameraFrame | Location | Time
		| User
⟨trusted_attributes⟩	⊨	⟨object⟩ | saved_location
		| allowed_time_slots
		| usernames
⟨object⟩	⊨	*Semantic object extracted*
		from visual data, like QR code,
		Faces, Animal, etc.
⟨func_name⟩	⊨	*ARCore or raw sensor APIs*
		exposed by Erebus
⟨action⟩	⊨	Allow | Deny
⟨relation⟩	⊨	⟨cmpOp⟩ | .Contains()
⟨cmpOp⟩	⊨	> | < | >= | == | !=

5.2 OS-level Enforcement

With an expressive policy specification language at hand, we now focus on the system-level enforcement of access control policies. Erebus consists of two main modules: **Permission Manager** and **AR Manager**. Erebus AR Manager directly interfaces with AR applications and acts as the gatekeeper for AR functionalities, while the Permission Manager interfaces the apps and the policy design. Erebus Permissions Manager module is based on the existing `PermissionManagerService` in Android which regulates the app accesses using the Android manifest file [2]. We add additional capabilities to this

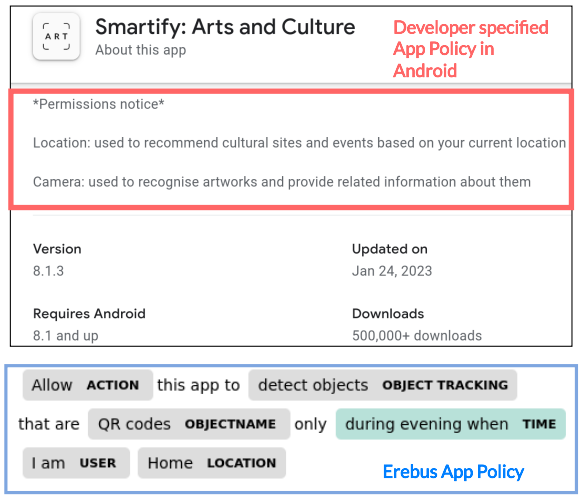

Figure 4: Erebus structured grammar allows developers/users to explicitly state the policy in a close to natural language form and the policy engine translates that into an access control rule.

```
function GetObjectRawPixels()
{
    let curLoc = GetCurrentLocation();
    let trustedLoc = GetTrustedLocation("Home");
    let curTime = GetCurrentTime();
    let validHour = GetValidHour("Evening");
    let curFace = GetCurrentFaceId();
    let trustedFaces = GetTrustedFaceId("Owner");
    let curCameraFrame = GetCurrentCameraFrame();
    let objName = "QR codes";
    if ( curLoc.within(trustedLoc) &&
        curTime.within(validHour) &&
         curFace.matches(trustedFaces) &&
          currentCameraFrame.includes(objName))
      {
              Allow;
    }
}
```

Listing 3: Enforcement rule generated from the policy input in Figure 4 restricts visual input to only QR codes from raw camera and only if certain environmental conditions are satisfied.

existing service to integrate our policy framework which is summarized below. To setup permissions for each app, we envision that developers will specify a default app usage policy that is enforced by Erebus at the OS level. Upon first-install, users would be prompted to review and finetune the developer-specified policy to fit their requirements. Figure 5 presents an overview of Erebus and how they interface with existing Android modules. In the following subsections, we discuss how we developed the different components of Erebus to enable this form of policy enforcement.

5.2.1 Erebus Permission Manager

The decision-making part of any policy within Erebus framework is governed by the data usage policy specified by the developers. In Table 4 we defined rules as a set of attributes, trusted attributes, and a relation defined over them whose satisfiability governs the access decision. Permission Manager in Erebus manages these attributes and their relations.

Similar to the Settings interface in Android, Erebus proposes an interface for the user to specify these 'trusted_attribute' values. Users can assign values for semantic tags such as "Home" and "Work" locations, "Office Hours" time slot, or associate different users of the system using their FaceIDs such as "John Smith", "Son", "Daughter", "Dad", and "Mom" for creating user-specific policies. We note that these sets of 'trusted_attributes' are variables that need only be set once and update rarely, as opposed to the set of 'attributes' whose values are fetched at run time for verification. Once the system has the information for all these tags, it can enforce the access control using the relation defined over them in the app policy. The easy interpretability of using an IFTTT model and a well-formed grammar for the policy definition language would also allow users to review the app policies and update them as necessary. Listing 3 and Figure 4 demonstrate how an effective policy of sensor access can be derived from a close-to-natural language policy text.

Different applications will require different sets of policies which would largely depend on their functional use cases. We enhanced the default PermissionsManager model of Android to allow persistent storage of Erebus policies. Developers may specify a certain policy for their app and users may review it and even update the IFTTT rule for their use cases. Erebus retains these policies on a per-app basis. We achieve this by using a JSON-formatted data store that stores the user-specified trusted_attributes making it accessible system-wide for all apps. This data store also stores the policies defined per app, similar to how Android manages app permissions. We assume the same system-level security to be applied to this data store as any other component of a trusted base OS.

5.2.2 Erebus AR Manager

Once the policies have been defined for each application, Erebus AR Manager is responsible for enforcing them and regulating access to sensor APIs. This module interfaces directly with AR applications and acts as the single vantage point for apps to use any AR functionality. It consists of two internal layers: an AR Function Provisioning layer, and an Access Control layer.

AR Function Provisioning. With Erebus, we bundle up all ARCore APIs behind an AR Function Provider to redirect all AR functionality through a single vantage point. We posit that AR SDK libraries, once integrated within the OS, should regulate all API accesses through a single entry point either as a module or as a class object. This allows for effective access monitoring and also prevents apps from using reflection to bypass any runtime checks. Erebus enforces access rules on all API calls made through this module, making this the primary gatekeeper for every application AR use case. We classify all ARCore functions into two categories: *Abstract* type and *Raw* type, based on what level of sensor information they return to the application. Erebus needs to apply the access control rules on

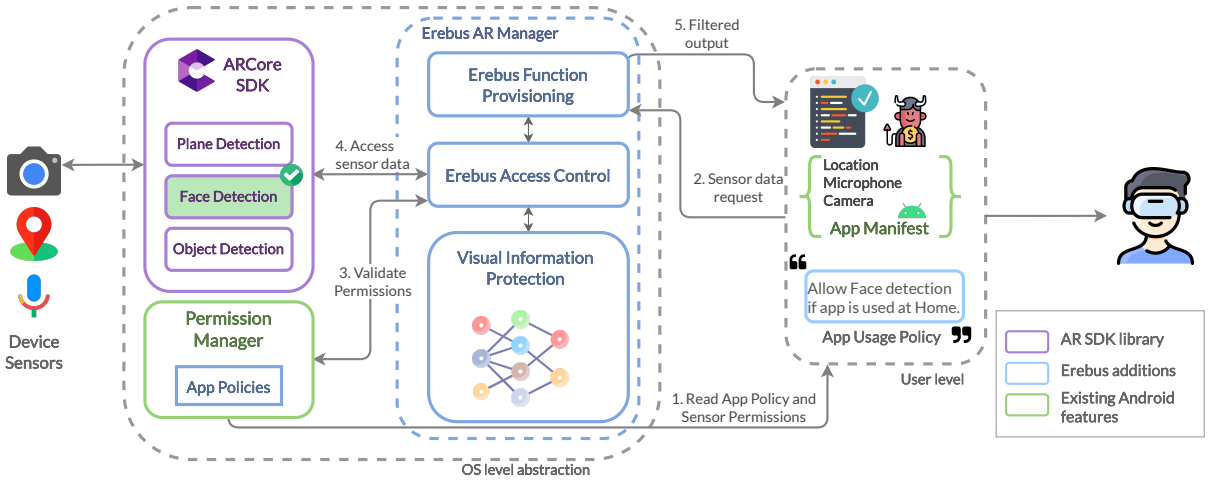

Figure 5: An overview of Erebus system design that shows all the components introduced alongside existing Android policy framework. Developers only need to specify an additional *App Usage Policy* alongside the App Manifest to adopt Erebus framework.

these functions to minimize information leakage to the caller.

Abstract Type: The majority of ARCore functions fall under this type — Functions that only return a fraction of information about an object in the physical world. These APIs internally process the raw information into rich semantics through abstraction, thus, facilitating developers to access only the desired AR functionality. One example of these functions is `ARRaycast`. It can return an estimated position of a flat horizontal surface in the physical world. This function comes in handy when a developer wishes to place a virtual object on top of a physical flat surface such as a floor or a desk. The caller doesn't require any knowledge of the techniques used behind the function (Localization, feature extraction, feature matching, surface detection, etc), so this API entirely abstracts the algorithmic computations and returns only limited high-level information requested. Therefore, for these kinds of ARCore functions, we directly forward the result retrieved once it passes the permission checks implemented by the Erebus Permission Manager.

Raw Type: Another group of functions accesses device sensors directly and exposes raw sensor data to the caller. An example of this type is `TryAcquireLatestCpuImage`, which returns an instance that contains the raw camera sensor data from the device.

Letting an app gain full access to the user's camera feed, unless required, poses serious privacy concerns. Erebus policies alone cannot prevent this privacy leakage with a simple '*Allow*' or '*Deny*' gatekeeping. We also cannot restrict all AR apps from using raw camera data and limit them to only ARCore APIs (*Abstract type* functions). Certain applications will request access to the raw camera feed if they require an object detection mechanism that is not provided by the ARCore library.

To resolve this challenge, we propose additional protection means via the Visual Information Protection module. Its use is dedicated to *Raw type* functions, and it helps Erebus limit the degree of freedom the AR applications have with the *Raw type* function (`TryAcquireLatestCpuImage`) while providing the necessary information the applications requests.

Visual Information Protection. The Visual Information Protection module is built on top of our design goal (**G2**) in Section 4.3. It provides Erebus the ability to eliminate the exposure of user's visual information by obscuring any unauthorized data at the system level while maintaining the core functionalities of *Raw type* functions.

The module achieves this goal in five steps: (i) Acquire raw camera frame from the hardware sensor, (ii) Detect potential security/privacy-sensitive objects, (iii) Perform instance-level segmentation of objects, (iv) Object class association with the counterparts in Erebus policy attributes, and finally (v) Instance-level visual information obfuscation. In Step (i), the module acquires the raw camera data via ARCore Camera API, following which, in Steps (ii) and (iii), the camera feed is passed onto our internal object detection model and the result of the detection is segmented. In Step (iv), once the inference from the model is complete, the Visual Information Protection module parses the Erebus policy of the associated AR application to obtain the list of user-authorized object classes and couple them with the classes of the detected objects. Finally, in Step (v), the module eliminates all the pixels of the camera feed except the instances with the same label as the target objects. For example, if the user has granted an AR application access to "QR Codes", similar to Listing 3, the module internally seeks to detect a QR code from the raw camera feed. Once the detection model identifies a QR code, it whitelists the segment of the detected QR code and passes only the relevant portion of the raw camera data to the app.

$$f(x) = \frac{f_1(x) f_2(x) \cdots f_n(x)}{\int_{-\infty}^{\infty} f_1(y) f_2(y) \cdots f_n(y) dy} \quad (1)$$

The Visual Information Protection module is the core layer that determines whether a piece of visual information must be protected or be exposed, to the application. An invalid exposure of visual information may lead to serious privacy leakage.

Table 5: Average latency comparison of camera sensor-based basic API and location sensor-based API in Unprotected AR application and Erebus-applied AR application. The numbers are described in milliseconds.

API Type	Erebus (ms)	Unprotected (ms)
Camera sensor-based API	0.35 ± 0.12	0.18 ± 0.04
Location sensor-based API	0.22 ± 0.04	<0.01

For increased confidence in the decision-making process, we apply a concept called Conflation [27]. The described mathematical equation behind the algorithm is shown in Equation 1. The method is used to consolidate two continuous probability distributions. For discrete input distributions like in our case, the definition of conflation is the normalized product of the probability mass functions. For instance, if the two input probabilities are 0.7 and 0.6, respectively, the conflation becomes 0.78. We heighten the confidence of the module's decision-making by "conflating" the confidence score of an object's each class across multiple frames. This is performed internally within Step (iv) above. It must be noted that, in our scenario, the risk of false positives is more critical to the framework than marginally reduced true positives.

6 Evaluation

We implement Erebus on Google's ARCore SDK and evaluate it using a series of micro and macro benchmarks. Details on our implementation can be found in Appendix A. We first evaluate how adapting Erebus affects the performance of an application in the form of additional incurred latency, along with the effectiveness of applying conflation for object segmentation. We then present our evaluation on 5 prototype applications, representing the broad categories of the AR app market today, developed using Erebus framework. These applications represent a wide breadth of existing AR applications that we analyzed in our survey. Appendix B provides performance analysis on our prototype applications and their correlation with existing applications in the market.

6.1 Microbenchmarks

We categorize our benchmarks depending on the type of visual data access each app uses. AR apps that access only high-level visual semantics, directly available through ARCore APIs, are evaluated first to demonstrate latency incurred with Erebus. We then evaluate AR apps that require access to raw Camera APIs to detect specific classes of objects. Since this group of apps utilizes the Visual Information Protection module, we evaluate each aspect of our design exhaustively to demonstrate its effectiveness.

6.1.1 Latency Analysis on Basic AR Functions

The basic AR functions represent *Abstract type* functions which require access to higher-level AR functions. Example of these functions includes AR Raycast, AR GetPlane, and AR ImageTrack (Markers) functions which fall under the categories of Raycasts, Plane detection, and Image detection AR functionalities in Table 3. We analyze the latency each AR function induces within our prototype application under Erebus, and compare it with the result of the environment without Erebus representing existing systems. We denote the latter environment as "Unprotected." We present our latency measurements in Table 5 which shows that Erebus incurs negligible overhead on the performance of the AR application for basic AR functions.

Table 6: Latency breakdown of the component in a single frame and the contribution to the latency. The top items represent the components of Erebus and the bottom items represent the components of the logic of the AR application.

Component Type	Component	Latency (ms)
Erebus	Object Detection	28.91
	Non-Max Suppression	0.02
	Object Tracking	0.25
	Conflation	0.01
	Whitelisting	0.08
Application	Async GPU Readback (Constant)	181.72
	Application Logic	33.47
Overall Latency		**244.46**

6.1.2 Latency Analysis on AR Functions with Raw Camera Access

Raw type functions intend to gain direct access to the raw camera sensor data. An example of this function is AR GetRawPixels (or `TryAcquireLatestCpuImage`) that falls under the category of Object detection AR functionality in Table 3. This function involves several computationally expensive operations. We break down the function into granular-level and analyze the latency of the components that constitute the function and investigate the performance of both the components related to Erebus and the components for the application functionalities. The latter, we denote them as "Application logic."

Our result in Table 6 indicates that Erebus can process the function call of a *Raw type* API in **29.27 ms (∼34.16 FPS)**. Also, we observed that our prototype application, while maintaining a constant delay time (181.72 ± 6.39 ms) introduced from the Async GPU Readback operation, is able to run with an acceptable overhead of **62.74 ms** (**∼15.94 FPS**). It must be noted that the total latency of the application includes the overhead of 'both' the application logic and the Erebus component. Moreover, the Async GPU Readback is a non-blocking operation that is executed in parallel in the background, allowing the application to maintain the latency it produces, constantly. Refer to Appendix B.2 to view our further analysis on improving the performance of the Unprotected components in the application.

6.1.3 Analysis of False Positive Rate with Conflation

A critical requirement for any access control system is to ensure the correct enforcement of its rules. However, object detection and classification are imperfect processes that may cause unwanted false positives in our system. The use of con-

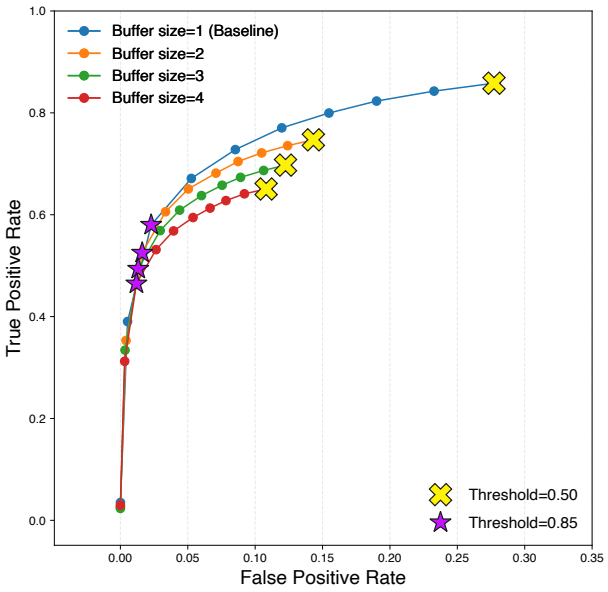

Figure 6: ROC curve obtained by varying confidence threshold of Erebus. The baseline curve refers to the FPR and TPR of Erebus when a single frame is used to determine the confidence score that informs the information obfuscation. As the number of frames used to calculate the confidence score increase, the false positive rate significantly decreases for lower confidence thresholds. This process allows Erebus to maintain a low FPR with minimum impact on its TPR.

flation (introduced in Section 5.2.2) enables us to make such decisions based on the output of multiple frames instead of one, limiting the false positive rate with minimum effect on the true positive rate. In this section, we evaluate the effectiveness of false positive removal of conflation. We denote the scenario where conflation was not applied, as "Baseline". The baseline simply uses the confidence score of an object in a single frame, to determine the rejection or the acceptance of information. We vary the number of frames used for conflation and compare the false positives with the Baseline on a public video object detection dataset [74]. As shown in Figure 6, the difference in false positive between the Baseline and the conflation applied by Erebus, is at least **14%**, at threshold 0.5 (yellow markers). Moreover, we observed that the rate of false positives converges as the threshold increases. The difference in false positive rate between the Baseline and Erebus with conflation, at threshold 0.85, is only **1%** (purple markers). This indicates that Erebus can effectively reduce the number of false positives for lower confidence thresholds using our conflation algorithm.

6.1.4 Analysis on Optimal Parameters for Conflation

We evaluated the benefits the conflation can provide to reduce the false positive rate in multi-frame object detection. Now, we present our findings on the effort to find a set of conflation parameters that are optimal to keep the false positives low, while maintaining the true positive rate to a reasonable level.

We investigate the 'elbow' of the ROC curve to pinpoint the exact threshold value where the difference in a false positive rate of the Baseline and its counterparts, starts to drastically close down. The reduced difference means that the false positive rates are retained almost persistently to a level while the true positive rate is dropping. We use the slope of a line to derive this. We connect any two points within the same threshold in the ROC Curve of Figure 6 and calculate the slope of the line. A line that has a closer angle to a vertical line (90 degrees) means that the two points have little difference in the false positive rate. Whereas, if an angle of a line is closer to a horizontal line (0 degrees), it means that the difference is large. Table 7 shows the slope of the lines which are comprised of the data points from Figure 6.

Table 7: The rate of change in TPR and FPR at each threshold using the slope. The data is equivalent to Figure 6. The difference between the baseline and Erebus is maximized at threshold=0.50, and it is minimized as it increases toward threshold=0.95. Smaller values represent bigger differences. B1,B2,B3,and B4, respectively represent, baseline, conflation buffer size=2, 3, and 4.

Confidence Threshold	B1, B2	B1, B3	B1, B4	B2, B4
0.50	0.83	1.04	1.23	2.75
0.60	1.19	1.48	1.75	3.54
0.70	1.82	2.23	2.67	5.13
0.80	3.42	4.43	5.34	10.53
0.85	**8.24**	**9.00**	**10.53**	**14.08**
0.90	32.39	30.26	36.66	41.68

We observe that the gap of false positive rate between the Baseline and the varying buffer size of conflation reduces drastically at threshold **0.85**. This tendency varied per class category, but the general observed tendency remained consistent. In other words, setting the threshold parameter to, no more than 0.85 can provide the most reasonable protection (High protection with reasonable true positive rates). For more explanation of the concept of conflation or to view our further analysis on conflation, refer to Appendix B.3.

6.2 Porting AR Apps to use Erebus

We now demonstrate how developers can leverage Erebus to build privacy-aware AR applications for consumers. In Figure 7 we apply Erebus's policy to a furniture viewing app, discussed in Section 3.2. The application uses AR Core libraries to detect flat surfaces, create virtual content and overlay it on the camera's live view. Erebus enforces restrictions on what high-level information an app can access, with no impact on the projection or the user experience. Furthermore, the expressiveness of Erebus's policy grammar allows adaptation to a wide variety of AR use cases. We ported 5 existing types of AR applications to our framework, summarized in Table 8, to demonstrate how Erebus enhances user privacy with simple app usage policy descriptions.

Once a policy is defined for the application, Erebus enforces access control on resources in two steps. First, it performs a rule validation to ensure all the parameters declared in the policy are defined. This implies that for every attribute used in the rule, the system should be aware of the *trusted_attribute* values. Erebus performs this validation to prevent malicious developers from bypassing access control by defining malformed or invalid rules; for example, "Allow Face detection if True" becomes a malformed rule because it disobeys defining

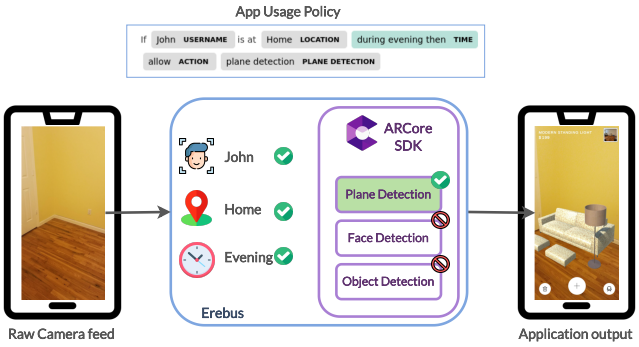

Figure 7: Erebus restricts the visual semantics that an app can access from live view based on the specified app policy. The virtual object created by the app is simply overlaid on the system's camera live view; with no impact on user's experience compared to existing implementations.

a strict relationship on user-controlled attributes. Once Erebus ensures that a valid rule has been provided, it checks for the satisfiability of the rule. This step is similar to any IFTTT programming model, where the action to be performed is determined solely by operating on the conditions.

Remote Maintenance AR with Multi-Object Segmentation. We highlight the usage of Erebus on more advanced usage of AR systems – *Remote Maintenance* type applications. This category of apps provides remote user assistance by sharing the user's live view with a remote technician allowing them to place virtual markers to help with troubleshooting. Table 8 shows the privacy risks that such an application can pose for a consumer, considering that functional use cases should be limited to sharing only the object of interest with a remote party.

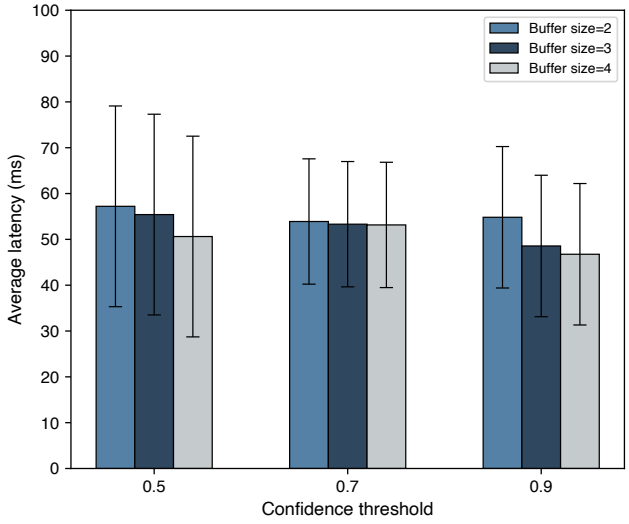

Figure 8: For multi-object segmentation use cases of Remote Maintenance app, Erebus provides a near real-time performance even with the added computational load from networking operations. The performance of the app ranged from **17.48 FPS** to **21.39 FPS**, as we varied the conflation parameters.

Erebus addresses this privacy concern with the Visual Information Protection module that allows for object segmentation and obfuscation to restrict visual data access by the app, as discussed in 5.2.2. Such an application can have requirements to be able to detect several target objects from a single raw camera frame, implying that the computational load and the complexity of policy enforcement increases with the number of objects to segment.

We developed an end-to-end prototype of a Remote Maintenance application that detects 8 distinct classes of objects using on-device object segmentation, and shares the visual artifacts with a remote entity over the network. Figure 8 shows the latency observed for accessing Camera APIs through Erebus, by varying the conflation parameters for higher segmentation accuracy. The latency of raw camera access by the application largely improves (by almost 22%) as we increase the conflation buffer size and confidence threshold for segmentation. This observation is due to the fact that the conflation algorithm is directly influenced by the amount of input data. As the threshold accuracy for object detection increases less number of segmented objects are passed into the module for post-processing with conflation, improving application performance while maintaining high detection accuracy. To view our further analysis on the advanced usage of Erebus, refer to Appendix B.4.

6.3 Discussion

False Positives. Restricting an app's access to visual input through object segmentation and obfuscation raises concerns about information leakage through False Positives. While Erebus significantly reduces the information leakage caused by AR applications and improves on False Positive rate using conflation, its performance is highly dependent on the accuracy of the underlying object detector and classifier. As the object detection and classification models improve, the performance of Erebus will similarly improve over time.

User Interaction. In our design of the user interface, we chose an IFTTT-style policy specification language because of its ease of use and its adoption across domains such as web-service automation and smart homes. However, similar to any AC system, it is inherently challenging to verify whether user-defined policies match their goals and expectations [20, 56]. In this paper, we focus on the system-design aspects of building such an enforcement mechanism and leave the usability challenges to future work.

Adapting Erebus to Other Sensors. Erebus's policy language design is motivated by how mobile and wearable platforms (iOS, Android, Apple HealthKit, Google WearOS) let apps access sensor data. These platforms provide SDKs with high-level APIs (Figure 2) that developers can leverage to fetch semantic information from sensors. Erebus can be similarly applied to any such underlying SDK with minimum modifications to support the changes. In this paper, we primarily focus on the Camera sensor because AR apps (as shown in Table 2) primarily rely on visual semantics, through ARCore SDK, and use other sensors to build environmental context. However, Erebus's language can similarly provide fine-grained access control for other sensors (*e.g.,* location accuracy in GPS, keyword detection in Microphone).

Table 8: Summary of ported AR applications to Erebus framework. Developers only need to provide the Erebus policy for their app that explicitly declares the usage criteria for their app, which can be easily reviewed and updated by the user.

Application Name	Description	Privacy Risk	Erebus Policy
Navigation AR	Immersive navigation app similar to Google Maps AR, that places virtual direction markers on a user's screen.	Perform surveillance using face detection or object detection APIs.	"Allow app to detect planes and access location."
Monsters AR	Edutainment app that detects a QR code image and places virtual objects on the screen.	These apps have highly specific use cases and do not need sensor access at all times or for all types of object detection.	"Allow this app to detect objects only for QR codes and only during the Evening."
Furniture AR	A mock-up of the IKEA AR app uses Plane detection functions to place virtual objects in the user's room.	App can scan for sensitive objects in the user's surroundings, like scanning for credit cards, face IDs of family members, etc.	"Allow Plane detection only if the user is at Home."
Face Filter	A mock-up of the Snapchat AR app that detects users' facial features and applies virtual masks or filters on their faces.	Unrestricted access to this highly sensitive data raises serious privacy concerns in sensitive locations, like the Gym or locker room.	"Allow this app to detect faces only when at Home and during permitted hours on weekends."
Remote Maintenance	Mock-up of TeamViewer Assist AR that allows technicians to assist with troubleshooting by sharing a live view of the user's surrounding over the network.	App can record sensitive objects from the live view, like Sticky Notes or personal documents, in addition to the required object needed for troubleshooting.	"Only allow Cellphones to be detected by the Camera."

7 Related Work

Our work touches on two veins of prior work: achieving a least privilege access control model for AR applications and effectively communicating sensor usage by AR applications (and their associated risks) to users.

Several works have touched on least-privilege access control, mostly from a visual-semantic obfuscation viewpoint. One of the early works in this area proposed a Recognizer abstraction [33], which limits app access to raw camera stream by filtering only the visual object that they are authorized to access. Similar approaches [1, 35, 53, 57] also focus on least privilege access to visual data by identifying and removing objects of interest from the app purview of the real world. Recognizer, specifically, proposes an OS abstraction that processes the sensor stream to "recognize" events and passes only the high-level semantic information to the apps instead of the raw stream. Darkly [34] takes a different approach and allows apps to manipulate transformed images instead of raw data. Other works have proposed a policy-driven approach to limiting app access. PrivateEye and WaveOff [49] use privacy markers to distinguish between public and private regions that the camera can recognize to prohibit applications from accessing raw camera feed in sensitive environments. Privacy passports [51] take this approach even further and allow real-world objects to declare their privacy policies using certain markers. All these works effectively filter out objects using image recognition techniques on raw camera feed; however, recent SDKs have already adopted image extraction capabilities and provide only high-level information to apps. Erebus is designed as a general-purpose framework that adapts to current implementations. PrivacyManager [39] proposes similar design goals as Erebus, but relies on domain administrators to provide app policies for the user base. We consider this in our threat model (Section 4.1) where proactive vetting mechanisms can often be ineffective against malicious developers.

Another line of work has explored the detection of resource accesses made by applications and notifying users of the risks it may pose to their privacy. Sensor access gadgets [28] proposed the idea of providing an explicit indication to users when an application is accessing sensor data. ipShield [9] provides a comprehensive list of inferences that can be drawn from a sensor data access. Erebus uses similar primitives by leveraging an understanding of AR functions and enforcing functional policies based on what inferences they make out of sensor data. LensCap [29] is a more recent work that uses a split-process development framework to achieve visual privacy specifically with cloud-based apps that share information with remote servers. Erebus is also centered on similar goals but focuses on ensuring privacy protection natively on the device itself.

8 Conclusion

The all-or-nothing access-control model in existing AR systems is insufficient to address the rich interactions of AR applications with sensory data. In this work, we presented Erebus, an access control framework designed for AR platforms that allow precise control over the sensor data shared with applications. To achieve this, Erebus uses a novel domain-specific language that allows intuitive fine-grained rules to be set on sensor data through developer-specified app usage policies, further allowing users the flexibility to customize permissions. We implement Erebus on Google's ARCore SDK and port five existing AR apps, representing the diversity of AR applications, to represent Erebus capability to secure various classes of apps. Performance results on these ported applications show that Erebus allows the AR applications to adhere to the principle of least privilege while incurring minimum overhead.

Acknowledgement

We thank Ping Hu, Saeed Boor Boor, anonymous reviewers, and our shepherd for their valuable feedback. This work was supported in part by NSF under grants IIS2107224, OAC1919752, and ICER1940302, and by IBM-SUNY and Meta awards. This work relates to the Department of Navy award N00014-20-1-2858 issued by the Office of Naval Research. The United States Government has a royalty-free license throughout the world in all copyrightable material contained herein. Any opinions, findings, or conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] Paarijaat Aditya, Rijurekha Sen, Peter Druschel, Seong Joon Oh, Rodrigo Benenson, Mario Fritz, Bernt Schiele, Bobby Bhattacharjee, and Tong Tong Wu. I-pic: A platform for privacy-compliant image capture. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications and Services*, 2016.

[2] Android. App Manifest Overview. https://developer.android.com/guide/topics/manifest/manifest-intro#groovy, 2022. [Online; accessed Mar 06,2022].

[3] Android Developers. Tristate Location Permissions. https://source.android.com/devices/tech/config/tristate-perms#tristate-screen, 2022. [Online; accessed Mar 06,2022].

[4] Patently Apple. Apple invents an hmd to work with a camera accessory that will make the public aware that the user is recording video. https://www.patentlyapple.com/patently-apple/2021/06/apple-invents-an-hmd-to-work-with-a-camera-accessory-that-will-make-the-public-aware-that-the-user-is-recording-video-more.html, 2022. [Online; accessed September 26, 2022].

[5] Patently Apple. Apple patents a flexible light guide system that could be integrated into flexible materials like a fabric apple watch band or vr glove. https://www.patentlyapple.com/2022/09/apple-patents-a-flexible-light-guide-system-that-could-be-integrated-into-flexible-materials-like-a-fabric-apple-watch-band-o.html, 2022. [Online; accessed September 26, 2022].

[6] Patently Apple. A new google patent reveals future ar glasses will work in sync with accessory devices to capture in-air gestures to control uis+. https://www.patentlyapple.com/2022/05/a-new-google-patent-reveals-future-ar-glasses-will-work-in-sync-with-accessory-devices-to-capture-in.html, 2022. [Online; accessed September 26, 2022].

[7] AppleInsider. Apple glass. https://appleinsider.com/inside/Apple-glass, 2022. [Online; accessed September 26, 2022].

[8] Alex Bewley, Zongyuan Ge, Lionel Ott, Fabio Ramos, and Ben Upcroft. Simple online and realtime tracking. In *IEEE International Conference on Image Processing (ICIP)*, 2016.

[9] Supriyo Chakraborty, Chenguang Shen, Kasturi Rangan Raghavan, Yasser Shoukry, Matt Millar, and Mani Srivastava. ipshield: A framework for enforcing context-aware privacy. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[10] Loris D'Antoni, Alan Dunn, Suman Jana, Tadayoshi Kohno, Benjamin Livshits, David Molnar, Alexander Moshchuk, Eyal Ofek, Franziska Roesner, Scott Saponas, et al. Operating system support for augmented reality applications. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*, 2013.

[11] Android developers. Create and monitor geofences. https://developer.android.com/training/location/geofencing, 2022. [Online; accessed August 17, 2022].

[12] Android Developers. Manage multiple users. https://developer.android.com/work/dpc/dedicated-devices/multiple-users, 2022. [Online; accessed August 14, 2022].

[13] Apple Developers. Healthkit. https://developer.apple.com/documentation/healthkit, 2022. [Online; accessed October 3, 2022].

[14] Microsoft Developers. Camera policy csp. https://learn.microsoft.com/en-us/windows/client-management/mdm/policy-csp-camera, 2023. [Online; accessed January 30, 2023].

[15] Microsoft Azure Docs. What is conditional access? https://docs.microsoft.com/en-us/azure/active-directory/conditional-access/overview, 2022. [Online; accessed August 14, 2022].

[16] Engadget. Magic leap 2 is the best ar headset yet, but will an enterprise focus save the company? https://www.engadget.com/magic-leap-2-ar-headset-tech-dive-143046676.html, 2022. [Online; accessed November 29, 2022].

[17] Enoxsoftware. opencvforunity. https://enoxsoftware.com/opencvforunity/, 2022. [Online; accessed August 17, 2022].

[18] Epson. Moverio bt-40 smart glasses with usb type-c connectivity. https://epson.com/For-Work/Wearables/Smart-Glasses/Moverio-BT-40-Smart-Glasses-with-USB-Type-C-Connectivity-/p/V11H969020, 2023. [Online; accessed January 30, 2023].

[19] Everysight. Raptor ar headset. https://support.everysight.com/hc/en-us, 2023. [Online; accessed January 30, 2023].

[20] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The effectiveness of application permissions. In *2nd USENIX Conference on Web Application Development (WebApps 11)*, 2011.

[21] Zheng Ge, Songtao Liu, Feng Wang, Zeming Li, and Jian Sun. Yolox: Exceeding yolo series in 2021. *arXiv preprint arXiv:2107.08430*, 2021.

[22] ThirdEye Gen. Thirdeye gen developer portal. https://thirdeyegen.gitbook.io/developer-portal/, 2022. [Online; accessed January 30, 2023].

[23] Google. Glass enterprise edition 2. https://developers.google.com/glass-enterprise/guides/get-started, 2023. [Online; accessed January 30, 2023].

[24] Sigmund Albert Gorski, Benjamin Andow, Adwait Nadkarni, Sunil Manandhar, William Enck, Eric Bodden, and Alexandre Bartel. Acminer: Extraction and analysis of authorization checks in android's middleware. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, 2019.

[25] Tom's Guide. Apple glasses: Everything we've heard so far. https://www.tomsguide.com/news/apple-glasses. (Accessed on 08/20/2022).

[26] Yuda Dian Harja and Riyanarto Sarno. Determine the best option for nearest medical services using google maps api, haversine and topsis algorithm. In *International Conference on Information and Communications Technology (ICOIACT)*. IEEE, 2018.

[27] Theodore P Hill and Jack Miller. How to combine independent data sets for the same quantity. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 2011.

[28] Jon Howell and Stuart Schechter. What you see is what they get: Protecting users from unwanted use of microphones, camera, and other sensors. In *Proceedings of Web 2.0 Security and Privacy Workshop*, 2010.

[29] Jinhan Hu, Andrei Iosifescu, and Robert LiKamWa. 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*, 2021.

[30] Vincent C Hu, David Ferraiolo, Rick Kuhn, Arthur R Friedman, Alan J Lang, Margaret M Cogdell, Adam Schnitzer, Kenneth Sandlin, Robert Miller, Karen Scarfone, et al. Guide to attribute based access control (ABAC) definition and considerations (draft). *NIST Special Publication*, 800(162), 2013.

[31] IFTTT. Building with filter code. https://help.ifttt.com/hc/en-us/articles/360052451954-Building-with-filter-code, 2022. [Online; accessed August 14, 2022].

[32] Ikea. Ikea Place app launches on Android. https://about.ikea.com/en/newsroom/2018/03/19/ikea-place-app-launches-on-android-allowing-millions-of-people-to-reimagine-home-furnishings-using-ar, 2018. [Online; accessed Mar 06,2022].

[33] Suman Jana, David Molnar, Alexander Moshchuk, Alan Dunn, Benjamin Livshits, Helen J Wang, and Eyal Ofek. Enabling {Fine-Grained} permissions for augmented reality applications with recognizers. In *22nd USENIX Security Symposium (USENIX Security)*, 2013.

[34] Suman Jana, Arvind Narayanan, and Vitaly Shmatikov. A scanner darkly: Protecting user privacy from perceptual applications. In *IEEE Symposium on Security and Privacy*, 2013.

[35] Marion Koelle, Swamy Ananthanarayan, Simon Czupalla, Wilko Heuten, and Susanne Boll. Your smart glasses' camera bothers me! exploring opt-in and opt-out gestures for privacy mediation. In *Proceedings of the 10th Nordic Conference on Human-Computer Interaction*, 2018.

[36] Kopin. Introducing solos, high performance eyewear for cyclists. https://www.kopin.com/introducing-solos-high-performance-eyewear-for-cyclists/, 2015. [Online; accessed January 30, 2023].

[37] Mira Labs. Mira prism pro. https://www.mirareality.com/, 2022. [Online; accessed October 3, 2022].

[38] Sarah M Lehman, Abrar S Alrumayh, Kunal Kolhe, Haibin Ling, and Chiu C Tan. Hidden in plain sight: Exploring privacy risks of mobile augmented reality applications. *ACM Transactions on Privacy and Security*, 2022.

[39] Sarah M Lehman and Chiu C Tan. Privacymanager: An access control framework for mobile augmented reality applications. In *IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2017.

[40] Lenovo. Thinkreality a3 smart glasses. https://www.lenovo.com/us/en/thinkrealitya3/, 2021. [Online; accessed January 30, 2023].

[41] Richard McPherson, Suman Jana, and Vitaly Shmatikov. No escape from reality: Security and privacy of augmented reality browsers. In *Proceedings of the 24th International Conference on World Wide Web*, 2015.

[42] Tim Meinhardt, Alexander Kirillov, Laura Leal-Taixe, and Christoph Feichtenhofer. Trackformer: Multi-object tracking with transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022.

[43] Meta. Meta quest 2. https://www.meta.com/quest/products/quest-2/, 2022. [Online; accessed October 3, 2022].

[44] Microsoft. Hololens 2. https://www.microsoft.com/en-IN/hololens/hardware, 2023. [Online; accessed January 30, 2023].

[45] NatML. Natml for unity. https://github.com/natmlx/NatML. (Accessed on 02/02/2023).

[46] NReal. Nreal air ar glasses. https://www.nreal.ai/light/, 2022. [Online; accessed January 30, 2023].

[47] Lucky Onwuzurike, Enrico Mariconti, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version). *ACM Transactions on Privacy and Security (TOPS)*, 2019.

[48] Hanyang Peng and Shiqi Yu. A systematic iou-related method: Beyond simplified regression for better localization. *IEEE Transactions on Image Processing*, 2021.

[49] Nisarg Raval, Animesh Srivastava, Ali Razeen, Kiron Lebeck, Ashwin Machanavajjhala, and Lanodn P Cox. What you mark is what apps see. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications and Services*, 2016.

[50] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J Wang, and Crispin Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *IEEE Symposium on Security and Privacy*, 2012.

[51] Franziska Roesner, David Molnar, Alexander Moshchuk, Tadayoshi Kohno, and Helen J Wang. World-driven access control for continuous sensing. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[52] Rokid. Rokid air pro. https://rokid.ai/rokid-air-pro/, 2023. [Online; accessed January 30, 2023].

[53] Jeremy Schiff, Marci Meingast, Deirdre K Mulligan, Shankar Sastry, and Ken Goldberg. Respectful cameras: Detecting visual markers in real-time to address privacy concerns. In *Protecting Privacy in Video Surveillance*. Springer, 2009.

[54] Kees Schollaart. Sortcs - a multiple object tracker. https://github.com/keesschollaart81/SortCS. (Accessed on 02/02/2023).

[55] scottyboy805. dotnow interpreter. https://github.com/scottyboy805/dotnow-interpreter. (Accessed on 05/19/2023).

[56] Bingyu Shen, Tianyi Shan, and Yuanyuan Zhou. Multiview: Finding blind spots in access-deny issues diagnosis. In *USENIX Security Symposium*, 2023.

[57] Jiayu Shu, Rui Zheng, and Pan Hui. Cardea: Context-aware visual privacy protection from pervasive cameras. *arXiv preprint arXiv:1610.00889*, 2016.

[58] Anthony Spadafora. Malware hits millions of Android users — delete these apps right now. https://www.tomsguide.com/news/malware-hits-10-million-android-users-delete-these-apps-right-now, 2022. [Online; accessed August 14, 2022].

[59] Animesh Srivastava, Puneet Jain, Soteris Demetriou, Landon P Cox, and Kyu-Han Kim. Camforensics: Understanding visual privacy leaks in the wild. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, 2017.

[60] Snapchat Lense Studio. Snapchat lenses. https://ar.snap.com/lens-studio, 2022. [Online; accessed Aug 17,2022].

[61] Peize Sun, Jinkun Cao, Yi Jiang, Rufeng Zhang, Enze Xie, Zehuan Yuan, Changhu Wang, and Ping Luo. Transtrack: Multiple object tracking with transformer. *arXiv preprint arXiv:2012.15460*, 2020.

[62] SparkAmpLab Editorial Team. Qualcomm reveals new ar smart glasses blueprint. https://www.sparkamplab.com/post/ar-vr-topic-analysis-qualcomm-reveals-new-ar-smart-glasses-blueprint, 2022. [Online; accessed October 3, 2022].

[63] Toshiba. dynaedge ar smart glasses. https://us.dynabook.com/smartglasses/products/index.html, 2022. [Online; accessed January 30, 2023].

[64] Unity. AR Foundation trackable managers. https://docs.unity3d.com/Packages/com.unity.xr.arfoundation@4.0/manual/trackable-managers.html, 2022. [Online; accessed August 17, 2022].

[65] Unity. Script compilation assembly definition files. https://docs.unity3d.com/Manual/ScriptCompilationAssemblyDefinitionFiles.html, 2022. [Online; accessed August 17, 2022].

[66] Dream World Vision. Dream glass flow. https://www.dreamworldvision.com/pages/dream-glass-flow-specs, 2022. [Online; accessed January 30, 2023].

[67] VR-Compare. Snap spectacles. https://vr-compare.com/headset/snapspectacles(2021), 2021. [Online; accessed January 30, 2023].

[68] VR-Compare. Xiaomi smart glasses. https://vr-compare.com/headset/xiaomismartglasses, 2021. [Online; accessed January 30, 2023].

[69] VR-Compare. Viture one. https://vr-compare.com/headset/vitureone, 2022. [Online; accessed January 30, 2023].

[70] Vuzix. Vuzix blade upgraded smart glasses. https://www.vuzix.com/products/vuzix-blade-smart-glasses-upgraded, 2023. [Online; accessed January 30, 2023].

[71] David Weintrop. Block-based programming in computer science education. *Communications of the ACM*, 62(8), 2019.

[72] Edy Winarno, Wiwien Hadikurniawati, and Rendy Nusa Rosso. Location based service for presence system using haversine method. In *ICITech*, 2017.

[73] Jialian Wu, Jiale Cao, Liangchen Song, Yu Wang, Ming Yang, and Junsong Yuan. Track to detect and segment: An online multi-object tracker. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021.

[74] Linjie Yang, Yuchen Fan, and Ning Xu. Video instance segmentation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019.

[75] Yaoyao Zhong, Weihong Deng, Jiani Hu, Dongyue Zhao, Xian Li, and Dongchao Wen. Sface: Sigmoid-constrained hypersphere loss for robust face recognition. *IEEE Transactions on Image Processing*, 30, 2021.

[76] Xizhou Zhu, Jifeng Dai, Lu Yuan, and Yichen Wei. Towards high performance video object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

A Implementation Details

A.1 Erebus Language

Our language framework constitutes two distinct components that work in sequence: Policy Language module and the Policy Transpiler. The app policy provided by the developers is passed on to the Policy Language module, which is implemented in Python that uses spaCy's entity recognition to detect the domain-specific entities of our policy framework. We used an existing open-source language model from spaCy and trained it on a self-annotated dataset of sample policies within Erebus's context. This Python program then uses the tags to generate a policy code using the grammar specified in Table 4. The generated policy language is passed on to a C# transpiler we built that converts it into the target C# code to be run on the device. We use ANTLR4 as the parser generator for our policy language and generate the target code.

A.2 Erebus Permission Manager

We referenced the Android Settings app to develop this module, to act as the primary interface for a user to review and edit app policies. This app remains independent of the AR app and is developed using Unity (Unity 2020.3.25f1, Android, OpenGLES3, Mono, .NET 4.0).

Attribute Management. We define three attributes to grant the user the ability to manage the permission of Erebus: *Time*, *Location*, and *Face ID*. The data type we define for each attribute is as follows: `TimeSlot (DateTime startTime, DateTime endTime)`, `GeoLocation (double latitude, double longitude)`, and `FaceId (byte[] id, string userFaceTag)`.

The *TimeSlot* defines the permitted/prohibited hours (`DateTime(HH:mm:ss)`) and is used along with our custom-defined C# Extension methods and Operator methods to implement the Erebus policy. Our *GeoLocation* works similarly to the concept of Geofencing in Android [11]. It is used to verify if the user is within the pre-defined trusted geographical location. We define a constant radius value (1 km) and compute the Haversine distance [26, 72] of the two locations. We store the list of *FaceId* by converting the pixel data into a byte array along with a tag. We maintain the face tag to be unique.

Policy Management. We allow the users to define their Erebus Policy in the Erebus Permission Manager app. Users implement their policy in natural language as described in Section 5.1. However, this policy must be translated into a format that can be recognized by the Erebus Access control layer. We translate the policy to a C# assembly bytes by sequentially going through the three steps : (i) Convert the natural language policy to Erebus language using the Entity recognition model. (ii) Transpile the Erebus language to C# code using ANTLR4. Finally, (iii) Runtime compiles the C# code to assembly bytes using Roslyn C# Compiler.

We built the conversion module in step (i) under a separate environment what we call *Local processing server*. We hypothesize that Erebus maintains a secure network communication channel with a trusted local server that is not exposed to the public and only is utilized for the purpose of performing computationally expensive operations such as Entity recognition model inference.

In Step (ii), the app executes the module that includes the conversion grammar, defined during the build-time of Erebus Permission Manager, on the device. In Step (iii), our permission manager runtime-compiles on the device under Unity *Mono* scripting backend. This is because the *IL2CPP* scripting backend in Unity performs AOT (Ahead-Of-Time) compilation, whereas *Mono* performs JIT (Just-In-Time) compilation. In other words, any runtime code compilation or assembly loading is prevented under the *IL2CPP* scripting backend configuration. Furthermore, we use Roslyn C# Compiler to create Unity Assembly reference assets [65] and compile code in the runtime. It provides the ease of adding references to the built-in C# libraries such as 'System.Collections' or 'System.Reflection,' for library dependencies and runtime compilation.

A.3 Erebus AR Manager

The Erebus AR Manager is a wrapper around the ARCore libraries that protects the user against potentially malicious AR apps. The AR Manager is essentially a C# application that interfaces with all the ARCore APIs but performs rule validation on top of them. We used Unity (Unity 2020.3.25f1, Android, OpenGLES3, Mono, .NET 4.0), AR Foundation, ARCore XR (Provider) package, OpenCV, OpenCVForUnity [17], NatML [45], and SortCS [54] for deep learning and image processing operations. The components included in the app are described below.

AR Function Provider. AR Function provider layer is a wrapper around AR Foundation API. It returns the result of the requested AR API after performing the necessary permission control, as discussed in Listing 4. Erebus only forwards the final result from the API to the caller after all the permission checks. So, when an app calls the *Raycast* functionality it is redirected through Erebus's function provider which internally

```
func Raycast(Vector2 screenTabPos, List hitRes, TrackableType trackableType)
{
  //Perform permission check
  var passedTest = accessController.ExecErebusPolicy("RayCast");
  if (!passedTest)
    return false;

  //Invoke API only when permission granted
  return erebus.Raycast(screenTabPos, hitRes, trackableType);
}
```

Listing 4: Pseudocode of Abstract type API, Raycast, of Erebus.

```
func GetObjectRawPixels()
{
  //Perform permission check
  var passedTest = accessController.ExecErebusPolicy("GetObjectRawPixels");
  if (!passedTest)
    return null;

  //Invoke API & Apply visual-privacy protection
  var rawCamPixels = erebus.GetRawCameraData();
  var allowedRawCamPixels = ProtectVisualPrivacy(rawCamPixels);
  return allowedRawCamPixels;
}
```

Listing 5: Pseudocode of Raw type API, GetObjectRawPixels, of Erebus

Table 9: Trackables are defined as the group of APIs that can extract specific semantic data from sensors.

Trackables	Sensor access APIs	Description
Plane detection	Raycast, GetPlane, ARPlaneTrackables, RegisterEventOnPlanesChange, UnRegisterEventOnPlanesChange	Identifies plane surfaces from the environment only
Image detection	AddImageReference, RemoveImageReference, RegisterEventOnTrackedImagesChanged, UnRegisterEventOnTrackedImagesChanged	Scans environment for specific images, such as QR Codes, Photographs (Image markers)
Object detection	GetRawPixels	Access raw camera stream and runs object detection to identify any specific object
Location detection	GetCurrentGPSLocation	Access GPS sensor information

calls the ARCore APIs and returns the final result to the caller.

Visual Information Protection Module. It is crucial for the VIP module to accurately identify objects in camera frames in order to prevent any information from being accidentally exposed while providing functionality with minimal latency. Inspired by widely-adopted concept in video object detection research [42, 61, 73, 76], our system aggregates information from each frame to increase confidence in detection. We leverage a light-weight object detector [21], a lightweight object tracker [8], and apply conflation [27] to the gain the extra confidence on detection. Furthermore, we utilize Neural Networks API (NNAPI) in Android via NatML in Unity to maximize the performance of object detection and Compute Shader to accelerate the whitelisting.

Aggregating Multiple Frames: Existing deep-learning-based video object recognition models share feature vectors across multiple frames within their models, leading to more accurate output. However, they are computationally expensive for a mobile device. We use 'Conflation' to increase detection confidence using multiple frames. For every frame, we run a tracking algorithm on the bounding boxes of the output to associate the object instances over multiple frames. Then, we conflate the confidence scores of the boxes to obtain an enhanced confidence score for each box. In our prototype, each box contains 80 classes (MSCOCO).

Whitelisting: We use a whitelisting approach to protect users' visual information. As opposed to the Blacklisting approach, which conceals only the designated targets, Whitelisting does not suffer from unexpected information exposure due to undetected targets.

Permission Checking. We perform rule validation and rule satisfiability checks for permission granting in Erebus. The access control layer of Erebus performs a validation check by confirming if the current AR app contains all the access permission to the attributes it will use. The Satisfiability of the rule checking occurs with every invocation of an Erebus AR API. The policy is executed by loading the compiled assembly bytes into the current scripting domain in the runtime and invoking the code in the assembly using the C# Reflection.

Face Detection and Recognition. User face recognition happens at the beginning of the AR app to identify the user within the Erebus AR Manager. Our user face recognizer uses the YuNet face detector [48] and SFace face recognizer [75], and uses pre-trained models from OpenCV model zoo and OpenCVForUnity examples.

B Additional Evaluations

Our five prototype AR apps evaluate Erebus in an environment that is similar to the real-world scenario. The prototype apps reflect the general category of existing apps in the market described in Table 8, categorized into apps that use *Abstract type* and *Raw type* APIs. We provide analysis on the total computation load of our prototype apps and compare the functional resemblance of the existing AR apps in Table 10. We only sample the apps that are present in Android's Google Play Store. The app latency data were collected for over 2,000 frames on an Android 13 Samsung Galaxy S22 phone by inserting checkpoints at run-time and processed after runtime. The apps were developed using Unity's Mono Scripting backend.

B.1 Basic AR Functions

To analyze the functional latency of the prototype apps that use Abstract-type API, we reference the functionalities of Google Maps AR Navigation. For a balanced comparison, we design two different apps with equivalent Unity environments, app configuration, logic, and the final output. Except that one app directly accesses the AR API via AR Foundation (ARCore), while the other via Erebus. We discovered that the majority of the latency of the Location API derives from the initialization phase of the GPS sensor.

B.2 AR Functions with Raw Camera Access

To analyze the functional latency of the prototype apps that use Raw-type API, we reference the functionalities of Snapchat Lenses. Our face filter app is configured to detect a single class object ('person') and utilizes the user-facing camera.

[4]The app total latency of Raw Type APIs increases due to the overhead of the Async GPUReadback. Note that its overhead is discrete from the performance of Erebus. Refer to Table 6 for additional analysis.

Table 10: Analysis of the computation load of the five prototype applications and the functional resemblance to the applications in the market. Prototype applications well-represent existing AR applications and provide analogous functionalities with minimal overhead discrepancy. Due to the non-blocking nature of Asynchronous GPU Readback operation, the application maintains a stable frame rate with added background computation task overhead.

Functional resemblance	Prototype App Name	Market App Name	App Total Latency (ms)	
			Erebus	Unprotected
AR Plane detection, AR Raycast, GPS sensor	Navigation AR	Google Maps Live AR view, Pokemon Go, Thyng, Horizon Explorer AR, Spyglass	8.51	8.38
AR Plane detection, AR Raycast	Furniture AR	Ikea Place, MeasureKit, Houzz, Aryzon AR, AR Sports Basketball	8.81	7.92
AR Plane detection, Image detection	Monsters AR	Quiver, Google AR Translate, Smartify, Kings of Pool, Roar	16.77	16.67
Face tracking, Object detection	Face Filter	Snapchat Lenses, Youcam Makeup, ModiFace, Leo AR, GIPHY	62.74 (244.46[4])	38.46
AR Raycast, Object Detection	Remote Maintenance	Google Lens, Vuforia Chalk, Blippar, Microsoft Dynamic Remote Assist 365, TeamViewer Assist AR	71.24 (240.31[4])	28.06

Table 11: Component-level latency breakdown of the AR Face Filter app using IL2CPP Scripting backend in Unity. The items are grouped by the component type (Erebus, App logic).

Component type	Component	Latency (ms)
Erebus	Object Detection	23.51
	Non-Max Suppression	0.01
	Object Tracking	0.15
	Conflation	0.01
	Whitelisting	0.08
Application	Application Logic	15.18
	Async GPU Readback (Constant)	104.03
Total		**142.97**

Improving the Performance of Non-Erebus Components: Erebus provides an entry point to the privacy-protected camera pixels through a Unity Texture that is fetched back to the CPU, for the developer's convenience (A CPU-fetched texture provides more degree of freedom to the developer). This is done by invoking an Asynchronous GPU Readback call once the Visual information protection module completed its task. However, we observed that the GPU Readback operation contributes 74.34% to the total latency of the application. Thus, we eliminate this overhead by providing the app developer the option to access the camera data texture directly from the GPU. Direct access to a pre-upload GPU texture allows the users to bypass the computationally-expensive Readback operation.

For our experiments, we also evaluated the performance of Erebus under ARM64 build with Unity's IL2CPP Scripting backend. We observed **41.52%** latency reduction (Table 11), leading the total latency (excluding Readback time) of the AR Face Filter app to **38.94 ms (25.68 FPS)**. The performance of the app improved in both Erebus and the app logic. However, unlike the Mono scripting backend, the app does not natively support Runtime (JIT) compilation. We implement a pre-defined class at the app implementation time in Unity. This class acts as a mock-up access controller of Erebus Permission Manager. Additional improvements to Erebus could be made by referencing Dotnow-interpreter [55] to support IL2CPP scripting backend.

B.3 Minimizing FP through Conflation

We analysed the effectiveness of conflation on a desktop environment (Intel® Xeon® Gold 6242 CPU@2.80GHz, 128GB RAM, Nvidia Quadro GV100 GPU, Ubuntu 20.04.3 LTS 64 bit). Conflation does not induce extra computational overhead as much as any deep-learning-based model. Due to the nature of its simplicity, it is not able to drastically improve the detection result. However, it is an effective way to associate multiple frames of confidence scores, thus, gaining higher confidence over the detection result, with almost no overhead.

We use a public video instance detection dataset, YouTube-VIS [74], and extract 12 different classes that intersect with the class of our pre-trained model. We follow the conventional standard, PASCAL VOC 2007 (IoU threshold=0.5) for the conflation parameter analysis. The parameters are two independent variables that are essential for the conflation algorithm: confidence threshold and buffer size.

The conflation buffer size indicates the number of frames the conflation algorithm aggregates to derive the final confidence value. *e.g.,* if an identical object with a confidence score of 0.60 is identified in two consecutive frames, the conflation algorithm merges the two confidence scores (N=2) and gives the output of 0.6923. The same confidence score in three frames(N=3), 0.7714. If the buffer size is smaller than the appeared consecutive frames, it overwrites the oldest previous confidence score and recalculates the conflated confidence value.

B.4 Analysis on Advanced Raw Camera Usage

We developed a prototype of the TeamViewer Assist AR to resemble the genre of collaborative AR apps used for telemaintenance. In our prototype, the live camera feed of the user is streamed to a remote user using TCP in NETMQ with an average latency of 34ms (round trip) for multiple 32-byte packets. We configure the VIP module to whitelist 8 distinct categories of objects: 'Clock', 'Cup', 'Bottle', 'Remote', 'Cellphone', 'Fork', 'Knife', and 'Spoon'.

According to our experiment, the receiving user was able to receive an average of **21**, **19**, and **19** camera images per second in Erebus environment with a conflation buffer size of 2, 3, and 4, respectively. While an average of 29 images were received in a second in an Unprotected (Non-Erebus) environment. Note that the result represents the number of images that are ready to be 'viewed' by the receiving user, in a second. It includes the latency introduced from the app logic: Networking (∼17 ms; Single trip), Packet encoding (∼4.6 ms), and Decoding time (∼2.3 ms), that is not part of Erebus. This indicates that Erebus is able to achieve near real-time performance even with the computationally intensive app logic.