Erebus: Access Control for Augmented Reality systems

***Index Terms*—Access control; continuous sensing; augmented reality; mixed reality**

1. Comments

Sanket: Sanket's comments
Yoonsang: Yoonsang's comments
Amir: Amir's comments
Arie: Arie's comments

Yoonsang: Also, let's not forget to mention the provision of two-factor auth. of Erebus. We check twice every time an ARCore function is executed: (1) Runs dev's/user's logic (from the Erebus Langauge input) and (2) checks if the current user/time/location is included in the trusted entity.

Yoonsang: Abstract, Intro/background, conclusion, acknowledgements

Yoonsang: A section summarizing (with bulletpoints) the contribution of our paper

2. AR application frameworks

The functional use of adding Augmented Reality (AR) features into any application is to create an immersive experience for user interaction. These applications capture the user's surroundings using sensor input, such as video, depth sensor, location, or audio, and then overlay virtual content on live camera feed through devices like smartphones or Head-Mounted Displays (HMDs). But in order to uniquely cater the virtual content, app developers need to derive context from the real world environment. In some cases, the app needs to know exactly what the user is looking at and produce digital content based on a visual marker. Alternatively the app may simply need to display 3D augmented reality models overlaying on live camera feed and thus only requires raw visual input. In a separate instance, app may require local information like user's location, walking direction, or road signs for generating content primarily governed by where the user is in the physical world.

For the purposes of this paper, we consider the three most common ways application developers implement AR functionality in their apps [14]. Let's take an edutainment app like Quiver [40] for example which uses AR techniques to make interactive coloring pages for kids. In order to display an animation, the app needs to know what the user is pointing their camera at that particular page. Quiver uses coloring pages, that can be downloaded from their website, which contain distinctive pictures or shapes that are easily recognized by the app.

Such applications where the digital world is anchored to the real world, using markers, are termed as *marker-based* applications. On the contrary, some applications require the user to interact with the virtual object and be able to move it around without being anchored to any real world entity. One such application is the Ikea Place app [28], which lets the user place a virtual furniture and move it around their living room. These applications, termed as *markerless* apps, typically do not require any "anchor" to the real world, but may place virtual objects on a flat surface to increase realism. Some more advanced use cases of AR provide realism with context to user's location where the virtual world is a part of the physical space, also known as *location-based* AR. Google Maps AR navigation [8] is one such example which collects GPS location, accelerometer, compass, and gyroscope data to localize and tailor the content based on the user's exact surroundings.

In order to create augmented reality, our devices first need to understand objects in the real world. Today, developers have a range of AR SDKs to choose from that provide a variety of tools for environmental understanding. Google's ARCore [2] and Apple's ARKit [11], among others, provide user level APIs that application developers can leverage to build an understanding of the user's physical surroundings and create virtual content. App developers can simply abstract only the required information from raw sensor streams without having to process the data themselves. While several such AR toolkit were originally intended for specific frameworks and hardware, recent push towards open-source SDKs and cross-platform support suggest a move towards standardization of AR application frameworks. However the current monolithic architecture of AR apps, where it has to include all the user-space libraries provided by the SDK, is inadequate from a privacy standpoint. While AR development frameworks have matured, the OSes that run them have not evolved sufficiently to support them.

Sanket: Do we need more statistics/table to show this categorization?

We investigated 45 most popular AR apps, on Google Play and Apple's App Store, and found that apps use a small subset of AR functions only yet retain full permissions to phone sensors. We manually inspected app description, user comments, and video descriptions to understand how different apps utilize AR functionality — whether they use markerless, marker-based, or location based AR techniques. We found that a large majority of apps (nearly 77%) implement a markerless AR approach, while only 6% of the apps in our sample set use a marker for their functionality. A small number of apps also use a combination of markers and location based tracking for their use cases. However, across all of these apps, we found a significant level of permission over-privilege.

This is not particularly surprising given that mobile OSes still offer only a *coarse-grained* access control over the sensors, with the manifest model [1] being quite prevalent yet insufficient in enforcing *least-privilege* [18]. AR applications today necessitate a rethinking of the way access control is enforced. Instead of permissions being baked into an app as an after-thought, users should be provided with the ability to decide *where, when,* and *how* an app uses certain resources.

3. Motivation

3.1. A furniture viewing app

Let us consider a popular category of applications, the AR furniture apps like IKEA Place [28], that largely provide the same functionality of allowing users to visualize furniture or home decor elements in their surrounding space. In order to achieve this, it leverages a *markerless AR* approach as we mentioned in Section 2. The app first needs to derive the context of user's surrounding by identifying flat surfaces, dimensions of the space, and lighting present around the room. Once the app has obtained all these necessary information, it generates the virtual content (furniture) scaled to user's surrounding and overlays it on the raw camera feed.

In order to derive context from user's surrounding, a developer can use either of two approaches. One would be to implement all the logic of extracting visual information from raw camera feed and package it with the app itself. This allows the developer to optimize on the information extraction and possibly develop a feature-rich application. However, this places the onus on the developer to use powerful computer vision models that build their own understanding of the real world; in effect limiting the features that can be implemented in a single app due to complexity. The second, and largely popular, approach is to use AR SDKs that provide user level APIs for environmental understanding. ARCore [2] and ARKit [11] support SDKs for many of the most popular development environments, providing native APIs for all essential AR functions. IKEA Place is one of the most popular apps developed using ARKit for iOS, and several others that are built using ARCore for Android.

3.2. Limitations in existing permissions model

Currently, irrespective of the approach used, AR function accesses are explicitly controlled by the app and remain largely opaque from the user. Android requires apps to define their permissions in a single manifest file [1], which is then used by the OS to enforce access control over the sensors. IKEA Place app, for instance, would request access to *Camera* and *Location* sensors. This information is communicated to the user via the *Settings* interface, either as a pop-up dialog requesting access or through permissions manager window as shown in Figure 1. Users can conditionally allow or block accesses to specific sensor data for individual apps, however most of these permissions remain persistent in practice giving apps access all the time. *Tristate permissions* were introduced with Android 10 [3] to restrict access to sensors only while the app is in use, however the pace of adoption has been limited to only a handful of sensors till now. We argue that although tristate permissions is a step in the right direction for data privacy, it is still not sufficient for AR applications and even cumbersome to manage due to the manifest model.

Consider the data access paths shown in Figure 1 for an application to access sensor information. A furniture AR app requires only a small subset of sensor data, like information about the plane surfaces in user's surrounding. Furthermore, it may only need that information at specific locations like when the user is at Home. But app manifests allow granularity at the sensor level and not individual functions, forcing developers to request full access to *Camera* even though the required access is just *Plane detection*. This is not only a classic case of an over-privilege but also underlines the opacity of an app's AR functional use. Users, and in practice the system as well, remain completely oblivious to the usage of information from the sensor stream. The retrieved information is processed entirely within the app context, making it improbable to distinguish if an app is using *Plane detection* or *Face tracking* internally, or just recording raw camera streams. All of these functionalities request access to Camera but some have substantially more privacy concerns than the others.

App stores do require developers to provide sufficient semantics, through app description and privacy policies, that justify their access to resources but by virtue of current permission model they are not descriptive about AR functions. IKEA Place's description says *"Scan the floor in your place. Select a product that you want to place. Move and place the product into the space"*. This gives the impression that functional requirement is to scan the room, create a 3D model of the product, and visualize it in context with the room. However, there is no existing mechanism to enforce this or even verify these functional accesses. Since all AR functionality is packaged within the application, including the ARCore SDK libraries, there is a non-existent policy check to ensure the app is not violating input privacy by recording raw camera stream or using face recognition semantics for user surveillance.

3.3. Violations to Input Privacy

We focus on scenarios where an unsuspecting user installs a trivial AR application on their device and grants all permissions it requests. We assume the device which runs the application, the operating system, and the hardware sensors to be trusted as shown in Figure 1. The application itself however can be arbitrarily malicious but it runs with user-level privileges and can access system resources, including device sensors, through trusted APIs controlled by the underlying operating system. We assume that the app

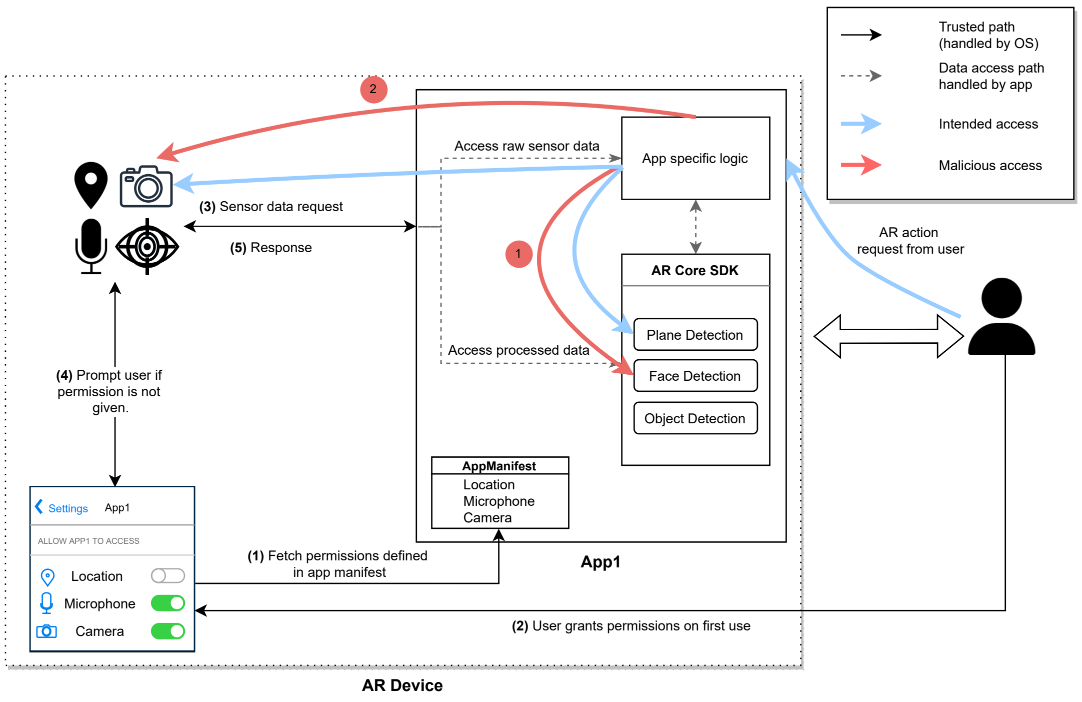


Figure 1. A reference model of application flow for an AR app Sanket: Rewrite caption to be more descriptive

is also developed using an AR application framework, as is generally the case, and thus has the ARCore libraries packaged with the app itself. As a realistic scenario we also assume the app uses third-party cloud-based services that collect user information, social media plugins, and syndicate ad services that are not directly associated with the app or its' developers. In Figure 1 we highlight two kinds of data leakage that different classes of attackers, described below, can perform in such an ecosystem.

Untrusted application exfiltrates sensor data. We assume that the installed application, either erroneously or maliciously, can overcollect user-specific semantic information and send it to remote servers. In the current ecosystem, lacking a proper vetting mechanism, AR apps retain persistent access to device sensors while masquerading with a trivial AR functionality like *Plane detection*, as we discussed in Section 3.2. This implies that apps can behave maliciously, or may be benign but buggy, and share sensitive information from raw sensors with an untrusted party. App can use data flow ① to access ARCore functionality that it is not supposed to without user's knowledge. This enables a potential attacker to derive highly sensitive visual semantics (like facial features) from the environment while masquerading as a benign application. An attacker may also perform remote surveillance by recording user's raw camera feed blatantly, as shown with data flow ②, and send it over to a remote server.

Ad brokers collecting user-targeted data. It is not uncommon for mobile applications to include syndicated advertisements in their platforms for monetization. Ad brokers notoriously tend to over-collect information on the user and use it to create targeted ads. We consider a scenario where an attacker can trick app developer into sharing continuous sensor data, using data flow ②, with the ad broker in return for better monetization guarantees. In this scenario, we assume the developer and the app itself may be honest, in the sense that they intentionally do not misuse permissions, but are tricked into sharing raw sensor information with an adversary.

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. More specifically, the protection mechanism today's app markets use is a *vetting process*, which uses static code analysis and malware signatures to detect apps that include malicious code. Even with proactive measures, millions of users are already affected by the time such apps are identified and removed [39] and yet still require users to manually uninstall them. This further highlights the urgency of a viable alternative to manage permissions for an AR application today.

4. Reimagining app permissions

We envision a new kind of permissions model and access control framework that is needed to support AR applications today. We focus on scenarios involving third-party AR apps that rely on native resources on the user's device, requesting access to onboard camera, GPS, microphone, and other sensors, to create the AR content. We are specifically interested in minimizing the sensor data that an app can access on user's device and ensuring that it follows the semantics of its functional use.

4.1. Threat Model

In line with prior work on security and privacy of augmented reality systems [23, 30–32, 36, 37], we assume the application is untrustworthy and may intentionally deviate from ideal scenarios with the goal of stealing user semantics that it should not have access to (i.e., sensor data not required for its functional use cases). Our goal is to ensure *minimal* exposure of sensor information to the app, approaching least-privilege, governed only by their functional requirement. In doing so, we focus our threat model on minimizing two distinct types of overprivilege in current AR systems.

Function-level overprivilege. 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. These unregulated functional access may lead to apps unintentionally or maliciously deriving sensitive user data from their environmental context. The different classes of attacks we mentioned in Section 3.3 leverage this form of overprivilege and gain control of privacy sensitive data that they should not have access to in the first place.

Attribute-level overprivilege. Privacy concerns over sensor data access extend much beyond just functional-level overprivilege. It is not enough to simply regulate access to AR functions as a *Allow/Deny* policy but also imperative to restrict accesses in unintended scenarios. One of the key problems with existing permission granting mechanisms is that application behaviors remain completely opaque to the system. Consequently, there is no mechanism for the user to explicitly declare their permission-granting intent, keeping them restricted only to a twofold policy. We consider the *tristate permission* model as a step in the right direction for allowing *attribute based policies* that let a user enforce conditional policies [15] restricting access to only when the app is in use. But this alone is not sufficient for AR applications.

4.2. Design Goals

Sanket: Need to rewrite this to highlight SDK in OS level
Also Goals should be more specific that we can justify in our design section

We identify the following design goals that a access control framework for AR applications should support.

G1 *Block direct access to sensors:* Apps should be restricted from accessing the sensors, especially Camera, directly. Since AR applications mostly depend on AR SDKs to provide abstract semantic information from raw camera feed, accessing raw camera should be restricted unless necessary. Furthermore, the operating system should support native APIs that apps can use directly instead of depending of user-level libraries.

G2 *Least privilege AR functional access:* We also need a policy implementation that approaches least-privilege for AR applications. Apps should not be allowed to access unnecessary APIs in the first place, so the access control framework should allow developers to specify their requirement at the granularity of functional use cases (*"Only Plane detection"*) and enforce the policy as such.

G3 *Transparency to user:* Even with a least-privilege access control framework, we cannot completely trust the developers to act in the best interests of the user. So we must allow users the ability to review developer's policies and update them as needed. In order to do so, the policy engine should be expressive enough to understand and implement user's intent of access control.

Yoonsang: Like we discussed on Zoom, don't forget to add a concise explanation of visual-protection layer (The return value of the API AFTER the permission check). Please refer to the 'Raw type' and 'Visual-privacy protecion module' subsections under the Erebus AR Manager section

5. Design of Erebus

In this work we present ***Erebus***, an access control framework complete with a policy specification language explicitly designed for AR applications that can be deployed today. We build upon the design principles first highlighted by D'Antoni et al. that operating systems themselves need to evolve to support AR applications [10]. Quite specifically, we develop a framework for access control to enforce input privacy policies for AR systems by reimagining how policy enforcement can be done in Android.

The goal of this system is to achieve a least-privilege model for AR functions and enforce policies at a system level. We achieve this by first proposing a new kind of policy specification language that lets developers express their intent at a functional granularity, thus aiding in transparency to the user. We also show a mechanism how user can review and update these policies to reflect their desired intent. Secondly, we demonstrate how the underlying operating system needs to evolve to support this access control framework. We develop our prototype on an Android system and showcase how different apps can leverage our design for their AR functionality.

Assumptions. The recent trend of developers primarily using SDKs to incorporate AR functionality into their apps, and the increasing support across popular app development environments suggest a natural progression towards making these SDKs available at an OS level. ARCore and ARKit are already quite extensively used for developing AR apps for Android and iOS respectively. We posit that these

AR libraries should become part of the base operating system, allowing apps to invoke AR functions through system-level APIs as opposed to user-level APIs. Our system is developed with this assumption, thus we develop our policy enforcement mechanism from a system-level perspective.

5.1. A policy specification language

We draw on the proposed theory of *language-based data minimization* [4] and its successful applications for enhancing data privacy in different platforms [9, 38]. A key aspect of any data minimization framework is through enforcement of data protection policies, not by hard-coding them, but by letting the system decide at run time. The system should be able to determine the minimal amount of data that is required by each individual subject at run time depending on how the data subject is making use of different functionalities. For example, a system that satisfies G1 has to ensure that apps only receive the semantic data they require from the raw sensor stream. The semantic data can be a face, a human body, flat surfaces, or any particular kind of object like QR codes if the app is accessing camera sensor for instance. This means that the underlying system should have transparency into the application's exact functional use case. Ideally we could design a system that learns the exact requirement of an AR app from their source code and enforce policies accordingly, but understanding intent requires static and dynamic analysis techniques [20, 33] which is not feasible to implement on a user device.

We take a different approach of expressing functional intent by looking at *rule-specification* in trigger-action platforms (TAPs). Popular TAPs like IFTTT [26] allow users to create simple *rules* that easily convey 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 [45] that lets even non-programmers create efficient automation policies. The key idea behind our language design is expressing functional intent of AR applications in the form of *If-This-Then-That* policies, allowing tandem policy writing through *developer* $\leftrightarrow$ *user* interaction (achieving G3) as well as enforcing a data-minimization framework (achieving G2).

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 [25]. They are essentially simple code snippets that are used to create custom application flows based on a set of attributes. We define our policy language in a similar way. Consider an application which requests access to GPS sensor on user's device for its functionality but the user only uses this app when they are at Home. The only granularity that current frameworks provide limit access while the app is in use, disregarding any other environmental factors. We present a scenario, in Listing 1, where the user wants to restrict access even further — limiting access to GPS only when they are at Home and at a certain time slot.

```
function GetGPSLocation()
{
  let curLoc = GetCurrentLocation();
  let trustedLoc = GetTrustedLocation("Home");
  let curTime = GetCurrentTime();
  if ( curLoc.within(trustedLoc) )
  {
    if ( curTime > 1800 )
    {
      Allow;
    }
  }
}
```

Listing 1. Policy to restrict access to GPS sensor only at Home and after 6:00PM

Sanket: review and rewrite — maybe We propose a similar policy specification language, like *filter codes*, that allows developers (and users) to write custom rules for sensor data access.

5.1.2. Application specific filter codes. Sanket: Most AR applications are designed for similar use cases, and can often be grouped together based on their functionality. This allows for certain groups of applications to share functionality specific access logic. With the IFTTT style filter codes, it becomes increasingly simple for similar types of applications to re-use the access control logic.

5.1.3. Mining policies from natural language. Sanket: The simplicity of having a policy definition language in the format of If-this-then-that allows us the flexibility to mine policies as natural language input from the users.
We developed a statistical named entity recognition model using spacy to demonstrate how access policies for AR applications can be easily mined from user's natural language input.

Yoonsang: Sanket, please don't forget to mention the grammar of the policy. Especially the 'keyboard' argument in GetRawCameraPixels( ): var whitelistObj = "Keyboard" CurRawDataFrame.Contains(whitelistObj). I'm using this in "Visual privacy protection module" under Erebus AR Manager

Yoonsang: A grammar specification(similar to Valve's Table3) will be good

5.2. Erebus Permission Manager

The permission manager is an interface that bridges the AR device owner and the permission control system. The role of the permission manager ranges from providing a user means to configure subject attribute, object attribute [24], and Erebus policy, to maintaining data up-to-date with a locally stored file.

5.2.1. Permission management. Permission manager requires the device owner to configure two things to safeguard the device sensor data from AR applications.

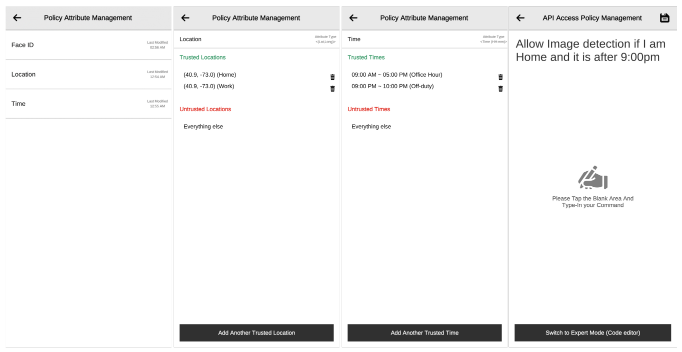
```
function    :
    ’function’ funcname ’(’ (ID (’,’ ID)*)?
    ’)’
        ’{’ bodystmt*  ’}’;

bodystmt    : ifstmt | assignstmt | action;

assignstmt  : ’let’ ID ASSIGN (value|api) ’;
    ’;

ifstmt  : ’if’ ’(’ conditionExpr ’)’
        ’{’ bodystmt ’}’;
action  : POLICY ’;’;

conditionExpr : logicalExpr;

logicalExpr
    : logicalExpr logicOp logicalExpr
    | ’!’ logicalExpr
    | comparisonExpr
    | ’(’ logicalExpr ’)’
    | logicalEntity
    ;

comparisonExpr
    : cmpOperand cmpOp cmpOperand
    | cmpOperand CONTAINS ’(’ cmpOperand ’)’
    ;

cmpOperand : opExpression;

logicalEntity : (TRUE|FALSE)
            | ID | STRING ;

opExpression
    : LPAREN opExpression RPAREN
    | numericTerm
    ;

numericTerm: ID | NUMBER ;

funcname : ID;
value    : ID | NUMBER | STRING | list |
    array ;

cmpOp: ’>’ | ’>=’ | ’<’ | ’<=’ | ’==’ | ’!=’
    ;
logicOp: ’&&’ | ’||’;

api: ID ’(’ value? ’)’;
list: ’[’ STRING (’,’ STRING)* ’]’;
array: ’[’ NUMBER (’,’ NUMBER)* ’]’;

CONTAINS: ’.includes’ | ’.matches’ | ’.
    within’;
POLICY  : ’Allow’ | ’Deny’;

fragment INT: [0-9]+;
NUMBER: INT (’.’(INT)+)?;
ID: [a-zA-Z_][a-zA-Z0-9_]*;
STRING: ’"’ (~(’\n’ | ’"’ ))* ’"’;
```

Listing 2. Language grammar Sanket: Need to reduce this to only the important parts

Attribute. The user can add or remove a set of user-defined constraints or attributes. In this paper, we concentrate on three object attributes : *Time*, *Location*, and *Face ID*. Figure 2 demonstrates the four examples of user-configured attributes. ‘Office Hour’ and ‘Off-duty’ for *Time* attribute, and ‘Home’ and ‘Work,’ for *Location* attribute.

Erebus policy. While the Attribute provides the ingredients how Erebus should distinguish the benign and malicious entities, the Policy produces the recipe to scrutinize AR API access permission. However, as discussed in Section 5.1, plain code-format may introduce an additional barrier to the lay-users. Thus, we leverage natural language to readily implement the policy for permission checking. Once the implementation is complete, the permission manager converts the given policy into C# Assembly in the runtime so that the policy is enforced on top of the logic of an AR application.

5.2.2. Data management. Permission manager uses a json file to save and load all the configurations. This system configuration file contains the Attribute data as well as the Policy data, and is synchronized upon user’s input. Note that we hypothesize that this configuration file is secure from any malicious modification attempts and is accessible only via permission manager.


Policy Attribute Management
Face ID
Last Modified 02:56 AM
Location
Last Modified 12:54 AM
Time
Last Modified 12:55 AM
Policy Attribute Management
Location
Attribute Type <(Lat,Long)>
Trusted Locations
(40.9, -73.0) (Home)
(40.9, -73.0) (Work)
Untrusted Locations
Everything else
Add Another Trusted Location
Policy Attribute Management
Time
Attribute Type <Time (HH:mm)>
Trusted Times
09:00 AM ~ 05:00 PM (Office Hour)
09:00 PM ~ 10:00 PM (Off-duty)
Untrusted Times
Everything else
Add Another Trusted Time
API Access Policy Management
Please Tap the Blank Area And Type-In your Command
Switch to Expert Mode (Code editor)

Allow Image detection if I am Home and it is after 9:00pm

Figure 2. Erebus Permission Manager (left: Attribute management, middle: x, right: Policy management) Yoonsang: Update description later

5.3. Erebus AR Manager

Erebus AR Manager is the core of Erebus framework because this component not only communicates directly with the AR applications, but also enables the Erebus policy to be enforced while the applications are running. Erebus AR Manager consists of three major internal layers: AR Function provision layer, Access control layer, and User interaction layer.

5.3.1. AR Function provision layer. AR Function Provider is a layer that enables the AR applications to access ARCore API from a single entry point. We brought the ARCore functions together and placed them under one layer within Erebus to have a full control over each function call. This gives us the ability to control the accessibility of each

ARCore function based on the policy enforced from Erebus. As shown in Figure ( ), we classify a set of ARCore functions into two categories: *Abstract* type and *Raw* type.

Abstract type. The majority of ARCore functions fall under this type. The AR functions belonging to this type only return a fraction of information of an object in the physical world. These functions internally process the raw data into a more high-level data to to provide an abstraction, thus, facilitating the AR app developers to get the access to the desired AR functionality. One example of these functions is *ARRaycast* function. 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 the place a virtual object on top of a physical flat surface such as floor or a desk. However, this function does not require any knowledge of the techniques used behind the function (Localization, feature extraction, feature matching, surface detection, and so on). It entirely abstracts the algorithms and returns the developer only limited high-level information.
Therefore, for ARCore functions that fall under Abstract type, we directly forward the result retrieved from the ARCore function once the function passes the permission checks implemented from the Erebus permission manager.

Raw type. Raw-typed functions access the device sensors such as camera sensor and depth sensor (or inferred depth data obtained from visual context), and return the acquired raw data to the caller. An example of the functions with this type is *TryAcquireLatestCpuImage* function. This function returns an instance that contains the raw camera data of the device at the point of the function call.
A complete access to user's camera sensor poses a serious threat to the user's privacy. However, even the Erebus policy cannot fully prevent this privacy leakage. The Erebus policy is configured to either '*Allow*' or '*Deny*' the access of an AR function. In other words, it will only be able to restrict the AR application from running at all if it attempts to gain access to the raw camera data, or allow entirely. Also, we cannot limit all AR applications from using raw camera data and oblige them to only use the functions provided from ARCore (*Abstract type* functions). To solve this problem, we introduce the Visual privacy protection module.

Visual privacy protection module. Visual privacy protection module is specifically designed for *Raw type* functions. The purpose of this module is to maintain the provision of *Raw type* functions to the AR applications, but obscuring any unwanted visual data from being exposed. Erebus achieves this goal using the four steps: (i) Objects detection, (ii) object instance segmentation, (iii) target label retrieval from Erebus policy, and (iv) target instance whitelisting. The steps are visually explained in Figure ( ). Yoonsang: The figure must include a more detailed visuals of the "Target label" in Erebus policy The visual privacy protection module first acquires the raw camera frame data from ARCore API. Then, the camera data is fed into the object detection/segmentation model. Once the inference from the model is complete, the module parses the Erebus policy of the associated AR application to obtain the label of the requested target object. Finally, the module filters out all detected instances except the instances with the same label as the target object.
Using our visual privacy protection module, Erebus is able to limit the degree of freedom the AR applications have with the *Raw type* function (*TryAcquireLatestCpuImage*), while providing the necessary information the applications requests.

5.3.2. Access control layer. The access control layer is the middle layer where it protects the user's invaluable information from being involuntarily leaked. To achieve this, Erebus implants two checkpoints within Erebus. The purpose of double checkpoints is to not only provide a more secure permission control by utilizing both the *attributes* and the *Erebus policy*, but also, provide an early permission rejection method for unauthorized applications. These two checkpoints are executed independently and respectively.

Shallow check. Erebus performs a shallow attribute permission check on the AR application that is executed. This checking is initiated only once at the beginning of the application. For this check, the access control layer first extracts the list of *object attributes* from the Erebus policy of the application. That is, the attributes that are expected to be used during the run of the application. Next, the access control layer checks if the current application is included in the list of trusted entities the user defined on the Erebus permission manager.

In-depth check. Unlike Shallow check, In-depth check performs a more granular permission checking. It does not perform a simple string parsing and an array matching. Rather, it makes the decision based on the logic provided from the Erebus policy. This checking function resides within the Erebus AR Function provision layer, and once an application invokes an Erebus AR API call, before passing the query result back to the application, it checks if the application has the permission to access the corresponding API. Then, based on the permission check result, the Erebus determines whether to the return the result of the API call, or an empty (C# 'default') value.

5.3.3. User interaction layer. The task of the user interaction layer is lighter than the other layers. It is responsible for interacting with the user. The interaction includes the face ID input interface which is similar to 'Face unlock' in Android, and a permission pop-up menu resembling the permission pop-up menu in Android. The purpose of this layer is to replicate the interfaces used in Android system and create a similar mock-environment of Android for our proof-of-concept of Privacy-protected OS-level AR API.

Col1	Col2	Col2	Col3
1	6	87837	787
2	7	78	5415
3	545	778	7507

TABLE 1. SUMMARY OF ATTACKS PREVENTED USING EREBUS'S POLICIES

6. Implementation

In this section, we explain the technical details of Erebus framework. The section summarizes the development environment, and the specifics within our three major components of Erebus framework : Erebus Language, Erebus Permission Manager, and Erebus AR Manager.

6.1. Erebus Language

Write any technical details of Erebus Language if necessary

6.1.1. NL DL Model explanation. Write any technical details of NL Model/Grammar, etc if necessary Yoonsang: Check the 'Erebus Permission Manager' subsection below if you are going to write sth here. I might have already covered this

6.1.2. API Grouping for Erebus NL Input Grammar. The jargon used in the domain of AR may present a barrier to the lay users. Thus, we group a list of AR APIs that are frequently used together, and categorized them with more user-friendly terms. The grouping of API is based on Unity's classification of Trackables [43] used in its cross-platform AR development framework, AR Foundation. The four types we define for Erebus policy are : Plane detection, Image detection, Object detection, and Location detection. Note that the grouping of APIs was based on our five prototype applications which represent the majority of AR application in the market with varying types : Markerless AR, Marker-based AR, and Location-based AR. The list below shows our grouping of AR APIs.

1) Plane detection
   - Raycast
   - GetPlane
   - ARPlaneTrackables
   - RegisterEventOnPlanesChange
   - UnRegisterEventOnPlanesChange
2) Image detection
   - RegisterEventOnTrackedImagesChanged
   - UnRegisterEventOnTrackedImagesChanged
   - AddImageReference
   - RemoveImageReference
3) Object detection
   - GetRawPixels
4) Location detection
   - GetCurrentGPSLocation

Sanket: Maybe this would be better as a Table instead of listing

Yoonsang: Maybe I should start referening Unity's ARFoundation instead of ARCore in all the contexts, because we are using ARCore on top of ARFoundation; Please remind me if I'm not realizing it until the deadline date

6.2. Erebus Permission Manager

We developed Erebus Permission Manager referencing the Settings app in Android because this is where the user implement Erebus permission policy and define attributes similar to the Permission settings in Android. The application is an independent application and was developed using Unity (Android, OpenGLES3, Mono, .NET 4.0).

6.2.1. 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 following : `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 similar to the concept of Geofencing in Android [13]. It is used to verify if the user is within the pre-defined trusted geographical location. We define a constant radius value (1Km) and compute the Haversine distance of the two locations. We retrieve more accurate GPS location using Android's native location function than Unity's. We store the list of *FaceId* by converting the pixel data into byte array along with a tag. We maintain the face tag to be unique.

6.2.2. Policy management. We allow the users to define their Erebus Policy in Erebus Permission Manager application. 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 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 Entity recognition model. (ii) Transpile the Erebus language to C# code using ANTLR4 [5]. Finally, (iii) Runtime compile the C# code to assembly bytes using Roslyn C# Compiler [29]. 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, and that this local server is not exposed to the public and only is utilized for the purpose of performing computationally expensive operations such as Entity recognition model inference. By utilizing a remote inference over the network, we were able to reap accurate, yet streamlined (in terms of development), conversion. Step (ii) is more simple than step (i) because the conversion grammar is statically included during the build-time of Erebus Permission Manager. Thus, we execute this module on the device. The runtime compilation in step (iii) also occurs on the device. However, there are a few configuration requirements that need to be applied. We configure the scripting backend of our permission manager application in Unity to *Mono*. This is because the *IL2CPP*

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

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

Listing 3. Yoonsang: Add Caption

```
public byte[] GetObjectRawPixels()
{
    //Perform In-depth 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 4. Yoonsang: Add Caption

scripting backend in Unity performs AOT (Ahead-Of-Time) compilation, whereas *Mono* scripting backend performs JIT (Just-In-Time) compilation. In other words, any runtime code compilation or assembly loading is prevented under the *IL2CPP* scripting backend configuration. Moreover, we utilize Roslyn C# Compiler to create Unity's Assembly reference assets [44] and compile code in the runtime. They help us add references to the built-in C# libraries such as 'System.Collections' or 'System.Reflection,' for library dependencies, and perform runtime compilation.

6.3. Erebus AR Manager

The Erebus AR Manager is a wrapper around the ARCore API while protecting the privacy of the user from potentially malicious AR applications. To showcase our proof-of-concept, we provide a Unity C# script that must be attached to a specific object instance in Unity (*'AR Session origin' gameobject*) when developing the AR application. The application was developed using Unity (Android, OpenGLES3, Mono, .NET 4.0), AR Foundation, ARCore XR (Provider) package, OpenCV [12], and OpenCVForUnity [16] for image processing operations.

6.3.1. AR Function provider. AR Function provider layer is a wrapper around AR Foundation's AR API. It returns the result of the requested AR API after performing the In-depth permission checking at the Access control layer of Erebus. The pseudocode code in Figure ( ) demonstrates how Erebus handles the API call of *Raycast* function.

Erebus simply forwards the returned result of the API to the AR application because the *Raycast* function is classified as *Abstract type*. However, as the pseudocode in Figure ( ) shows, *Raw type* function such as *TryAcquireLatestCpuImage* function which is mapped to a function named '*GetObjectRawPixels*' in Erebus, operates in a more convoluted way.

After performing the In-depth permission checking, the *Raw type* function retrieves the result of the corresponding AR API, in this case, the camera image data. Then, it applies the visual-privacy protection filter to exclude the list of unauthorized object instances from the image data, and whitelist only the permitted information.

6.3.2. Visual-privacy protection module. A secure visual-privacy protection is necessary in order to protect the users from the exploitation of *Raw type* AR API. We developed the visual-privacy protection module by applying three different methods all together. We utilize an instance segmentation/object detection model, Mask-RCNN [21], object tracking model, ByteTrack [46], and a mathematical method called Conflation [22]. We first run the detection/segmentation model on an input image. However, instead of the result being passed back to the caller instantly, we conflate the probabilities of the detection result over multiple frames to gain more confidence on the decision. Also, to achieve this, we track and associate the instances of multiple frames using the object tracker. We perform this processing on our *local processing server* same as our Entity recognition language model, to obtain an output that is both accurate and able to achieve near real-time performance.

6.3.3. Permission checking. There are two kinds of permission checking. Shallow checking and In-depth checking. As described in Section 5.3.2 the Erebus access control layer performs a check by confirming if the current AR application contains all the access permission to the attributes it will use. The In-depth check occurs every invocation of an Erebus AR API. In Figure ( ) and ( ), the function `ExecErebusPolicy("RayCast")` and function `ExecErebusPolicy("GetObjectRawPixels")` represent the In-depth permission checking. The policy is executed by loading the compiled assmebly bytes into the current scripting domain in the runtime and invoking the code in the assembly using C# Reflection.

6.3.4. Face detection & recognition. User face recognition happens at the beginning of the application from the User interaction layer. We built our user face recognizer using a face detector called YuNet [34] and a face recognizer called SFace [47]. The pre-trained models are from the official model zoo of OpenCV and the examples from OpenCVForUnity.

6.4. Prototype applications

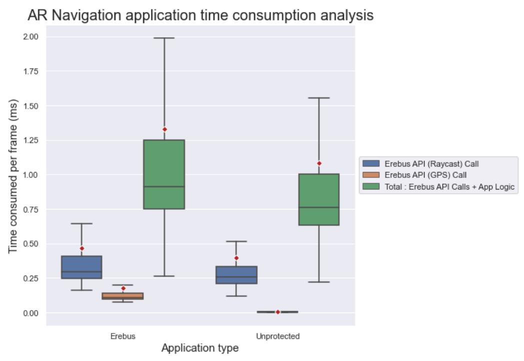
We implemented five distinct types of AR applications to demonstrate the use-case of our framework and to conduct an analysis. Our first three applications are : AR Furniture preview, AR Toy monsters, and AR Navigation. They respectively represent, Markerless AR, Marker-based AR, and Location-based AR applications. Another application we developed, is AR Face filter. Recently, there has been numerous applications that place a virtual face filter with a significant accuracy [6][41]. We anticipate that this was not possible without the advancement of deep model-based facial landmark detection. Thus, we add an application that requires a deep-learning model in the app logic as well as an access to the raw data of the front-facing camera. Finally, we consider an application with increased complexity. An AR application that involves more than one user, and be able to synchronize data across multiple users in real-time. Thus, we add our application called AR Remote maintenance. Note that all of our prototype applications were developed inspired by existing applications in the public mobile application market, and each type represents an application in the market (IKEA Place [27], Quiver App [40], Google Maps Live view [19], Snapchat Lenses, TeamViewer Assist AR [42]).

Yoonsang: Please remind me if I haven't added the acknowledgements for all the 3D models/assets/icons I used in the applications

7. Case study

In this section, we perform a case-study of Erebus framework. We quantitatively analyze the performance of our proposed methods and evaluate our approach in five different scenarios.

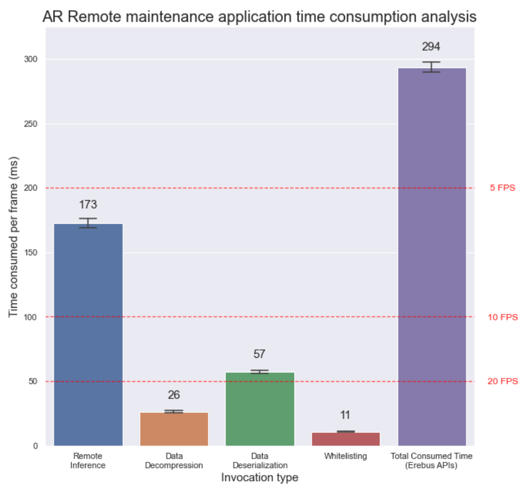
7.1. AR Navigation



Yoonsang: THIS FIGURE IS JUST A PLACEHOLDER
Mention the performance measurement of API call

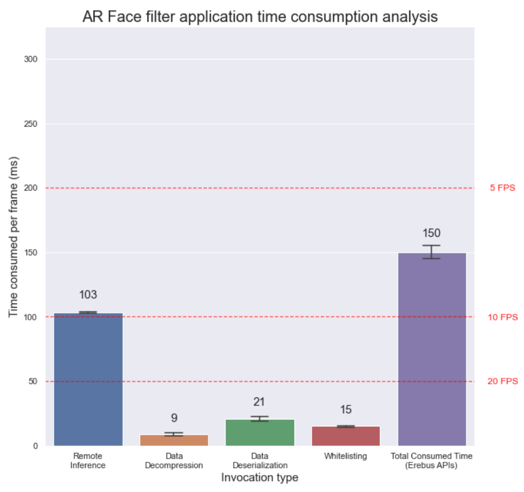
7.2. AR Remote maintenance

Latency analysis of the application
Yoonsang: THIS FIGURE IS JUST A PLACEHOLDER



7.3. AR Face filter

- Network-based : Latency analysis of the app Yoonsang:



THIS FIGURE IS JUST A PLACEHOLDER

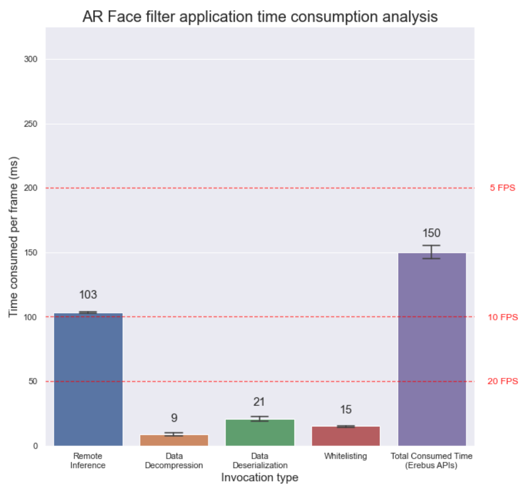
- On-device : Latency analysis of the app (Improved)

Unity Barracuda C# (Erebus framework-advanced) Yoonsang: Why ARM64 Cannot run in Unity - ARCore Barracuda compatibility issue - IL2CPP (AOT) compat. issue

Yoonsang: Also mention about the On-device ML module and its execution environment (Mono, OpenGL, Vulkan, etc)

7.4. AR Toy monsters

User defined policy



7.5. AR Furniture preview

User modifies the developer-defined policy + Adds user's own policy

8. Related Work

Sanket: Will add few more papers for comparative study, also need to rewrite relation with prior work

Prior studies have indicated that many applications are often over-privileged within the Android ecosystem [7, 17], and furthermore users typically show little comprehension of Android manifests [18]. While these works argue against the effectiveness of manifest based permissions control, it still proves to be one of the most prevalent ways for a developer to communicate a super-set of permissions an app may need. An effective access control mechanism then simply needs to address the shortcomings of manifests, enabling permissions that are in-context of the application and provide least-privilege access to sensor inputs.

Our approach is motivated by the research directions first identified by D'Antoni et al. where they posit that access control for AR applications should be implemented with Operating systems support [10]. They very accurately identify the research problem with AR systems, i.e., how OS should manage natural user input which is presented to applications ensuring that the input is in band to the requirements of the application. Building on this principle, Jana, Narayanan, and Shmatikov proposed a privacy protection layer in Android systems which restricts access to sensitive information by limiting access to the OpenCV APIs thus passing only the required objects to the untrusted app[30]. This technique, however, is very domain specific and does not provide an effective general purpose framework for extending access control to other sensory inputs. Furthermore, their proposal puts the trust in the application developer to define the objects that are needed from the raw camera feed. PrivateEye and WaveOff [35] addresses this by proposing *privacy markers* that real-world objects can use to define public and private regions to the camera app, thus allowing an untrusted app to only receive raw feed of the public region. A similar approach was also undertaken by Roesner et al. where they proposed *privacy passports* as a mechanism for real world objects to declare their privacy policies[37]. While these mechanisms relieve the user's burden for permission management, they are only effective if they get widely deployed and adapted in real-world scenarios.

Our approach differs from these prior work by providing a general framework for policy writing that can be easily adopted today. Our policy language aims at bringing transparency of permissions control to end users, while allowing application developers to still leverage the full flexibility of existing manifest based permission management.

References

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

[2] Android. *Google AR & VR — ARCore*. https://arvr.google.com/arcore/. [Online; accessed May 06, 2022]. 2022.

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

[4] Thibaud Antignac, David Sands, and Gerardo Schneider. "Data minimisation: a language-based approach". In: *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer. 2017, pp. 442–456.

[5] ANTLR. *ANTLR4*. https://www.antlr.org/download.html. [Online; accessed August 17, 2022]. 2022.

[6] Apple. *Apple Memoji*. https://support.apple.com/en-gb/HT208986. [Online; accessed Aug 17,2022]. 2022.

[7] Adam Barth et al. "Protecting browsers from extension vulnerabilities". In: (2010).

[8] Bibhash Biswas. *Navigation in Augmented Reality from Google Maps*. https://medium.com/swlh/the-new-google-maps-in-ar-587285a5d523. [Online; accessed August 03, 2022]. 2022.

[9] Yunang Chen et al. "Practical Data Access Minimization in Trigger-Action Platforms". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 2929–2945. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/chen-yunang-practical.

[10] Loris D'Antoni et al. "Operating system support for augmented reality applications". In: *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*. 2013.

[11] Apple Developer. *ARKit6 - Augmented Reality*. https://developer.apple.com/augmented-reality/arkit/. [Online; accessed August 03, 2022]. 2022.

[12] OpenCV Developer. *OpenCV*. https://opencv.org/. [Online; accessed August 17, 2022]. 2022.

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

[14] Applause Digital. *3 Different Types of AR Explained: Marker-Based, Markerless & Location*. https://applausedigital.com/3-different-types-of-ar-explained-marker-based-markerless-location/. [Online; accessed August 03, 2022]. 2019.

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

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

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

[18] Adrienne Porter Felt et al. "Android permissions: User attention, comprehension, and behavior". In: *Proceedings of the eighth symposium on usable privacy and security*. 2012, pp. 1–14.

[19] Google. *Google Maps*. https://www.google.com/maps. [Online; accessed Aug 17,2022]. 2022.

[20] Sigmund Albert Gorski et al. "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, pp. 25–36.

[21] Kaiming He et al. "Mask r-cnn". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2961–2969.

[22] Theodore P Hill and Jack Miller. "How to combine independent data sets for the same quantity". In: *Chaos: An Interdisciplinary Journal of Nonlinear Science* 21.3 (2011), p. 033102.

[23] 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, pp. 14–27.

[24] Vincent C Hu et al. "Guide to attribute based access control (abac) definition and considerations (draft)". In: *NIST special publication* 800.162 (2013), pp. 1–54.

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

[26] IFTTT. *If This Then That*. https://ifttt.com/. [Online; accessed August 14, 2022]. 2020.

[27] IKEA. *Ikea Place*. https://www.ikea.com/au/en/customer-service/mobile-apps/say-hej-to-ikea-place-pub1f8af050. [Online; accessed Aug 17,2022]. 2022.

[28] 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. [Online; accessed Mar 06,2022]. 2018.

[29] Trivial Interactive. *Roslyn C# - Runtime Compiler*. https://trivialinteractive.co.uk/products.html. [Online; accessed August 17, 2022]. 2022.

[30] Suman Jana, Arvind Narayanan, and Vitaly Shmatikov. "A scanner darkly: Protecting user privacy from perceptual applications". In: *2013 IEEE symposium on security and privacy*. IEEE. 2013, pp. 349–363.

[31] Suman Jana et al. "Enabling {Fine-Grained} Permissions for Augmented Reality Applications with Recognizers". In: *22nd USENIX Security Symposium (USENIX Security 13)*. 2013, pp. 415–430.

[32] 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, pp. 743–753.

[33] Lucky Onwuzurike et al. "Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version)". In: *ACM Transactions on Privacy and Security (TOPS)* 22.2 (2019), pp. 1–34.

[34] Hanyang Peng and Shiqi Yu. "A systematic iou-related method: Beyond simplified regression for better localization". In: *IEEE Transactions on Image Processing* 30 (2021), pp. 5032–5044.

[35] Nisarg Raval et al. "What you mark is what apps see". In: *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. 2016, pp. 249–261.

[36] Franziska Roesner et al. "User-driven access control: Rethinking permission granting in modern operating systems". In: *2012 IEEE Symposium on Security and Privacy*. IEEE. 2012, pp. 224–238.

[37] Franziska Roesner et al. "World-driven access control for continuous sensing". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 2014, pp. 1169–1181.

[38] Laurens Sion, Dimitri Van Landuyt, and Wouter Joosen. "An Overview of Runtime Data Protection Enforcement Approaches". In: *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE. 2021, pp. 351–358.

[39] 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. [Online; accessed August 14, 2022]. 2022.

[40] Google Play Store. *Quiver - 3D Coloring App*. https://play.google.com/store/apps/details?id=com.puteko.colarmix&gl=US. [Online; accessed August 03, 2022]. 2022.

[41] Snatpchat Lense Studio. *Snatpchat Lenses*. https://ar.snap.com/lens-studio. [Online; accessed Aug 17,2022]. 2022.

[42] TeamViewer. *TeamViewer Assist AR*. https://www.teamviewer.com/en-us/augmented-reality/. [Online; accessed Aug 17,2022]. 2022.

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

[44] Unity. *Script Compilation Assembly Definition Files*. https://docs.unity3d.com/Manual/ScriptCompilationAssemblyDefinitionFiles.html. [Online; accessed August 17, 2022]. 2022.

[45] David Weintrop. "Block-based programming in computer science education". In: *Communications of the ACM* 62.8 (2019), pp. 22–25.

[46] Yifu Zhang et al. "Bytetrack: Multi-object tracking by associating every detection box". In: *arXiv preprint arXiv:2110.06864* (2021).

[47] Yaoyao Zhong et al. "SFace: Sigmoid-constrained hypersphere loss for robust face recognition". In: *IEEE Transactions on Image Processing* 30 (2021), pp. 2587–2598.