Aragorn : Enhancing User Privacy in Camera-based Apps

Anonymous Author(s)

ABSTRACT

Mobile app developers often rely on high-quality cameras to implement rich features. However, giving apps unfettered access to the mobile camera poses a serious threat to user privacy when camera frames capture sensitive information that is beyond the scope of the app. To mitigate this threat, we present Aragorn , a novel privacy-enhancing system for mobile cameras that provides fine grained control over what information can be present in camera frames before apps can access them. Aragorn automatically detects app-relevant regions in camera frames and blocks out all other regions to protect user privacy while retaining app utility. Aragorn can cater to a wide range of camera apps and incorporates knowledge distillation and crowdsourcing based mechanisms to extend robust support to previously unsupported apps.

In context of a credit card scanning user study, we find that Aragorn mitigates a potential camera based user identification attack by 92% without any impact on card scanning functionality. Our implementation of Aragorn within the Android camera subsystem achieves frame rates of 20 frames per second. Overall, our evaluation shows that Aragorn can accurately and efficiently sanitize camera frames.

1 INTRODUCTION

Billions of smartphone users use camera-driven apps (or camera apps) to capture photos/videos [7, 18, 21, 45], experience AR/VR capabilities [36, 47], scan documents [3, 12], or use assistive technologies [4, 11]. Mobile cameras also enable app developers to design robust security challenges [8, 10, 15, 16, 23, 50]. For example, mobile cameras can be used for scanning QR codes to implement two-factor authentication [1, 55], privately share contact details [35], or verify possession of credit cards [15, 16].

Despite their utility, the rich information captured by cameras potentially exposes users to privacy violations from third-party apps. Currently, while users can control which apps can access camera frames, they cannot control what can be accessed within these frames. There have been several instances of users being subject to humiliation owing to the camera inadvertently capturing sensitive objects from their environment [34, 41]. Even in the absence of explicitly sensitive objects, apps could use this background information to profile user interests [46], identify returning users (§ 7.1.1) or even collude with other apps to track them.

Apps can exploit the diversity in backgrounds captured by camera frames as a fingerprinting vector. They could adopt such fingerprints to track users in response to mobile operating systems limiting support for device identifiers [6, 20]. This is analogous to websites relying on fingerprinting-based stateless tracking in response to browsers limiting support for cookies [51]. While users can take efforts to remove potentially identifying or sensitive objects from their environment, it is challenging, even for careful users, to ensure that their background cannot be exploited for tracking.

Prior research to protect user privacy from the visual information contained in images can be classified into two categories: those based on blocklists that conceal parts of images that are potentially sensitive to user privacy, and those based on allowlists that reveal certain parts of images and conceal everything else [17]. Solutions based on blocklists are strongly tied to certain sensitive objects within images (such as faces [2, 24, 44].) As a result, extending their approach to other sensitive objects is difficult. In contrast, while recent solutions based on allowlists are not confined to certain objects, they pose a manual and cognitive burden on users [27, 40]. For example, WaveOff [40] requires users to provide precisely marked references of objects to be preserved in camera frames. Thus, such systems only serve careful users who can bear this burden. To the best of our knowledge, prior research lacks an automated solution that operates with minimal user effort while also remaining extensible to cater to diverse camera applications.

In this paper, we present Aragorn —an automated and extensible privacy-enhancing system for mobile cameras that protects user privacy from camera apps while preserving app utility and user experience. To this end, Aragorn uses a machine learning based *sanitizer* to automatically remove everything but objects that are relevant to an app's utility from camera frames before it can access them. We integrate the sanitizer within the operating system for improved security and compatibility with all apps.

Aragorn employs a novel training strategy to retrain the sanitizer whenever apps seek support for previously unrecognized objects. For example, there are some apps in use today that scan COVID vaccination cards – an object that did not exist prior to 2021. In such cases, Aragorn gathers training images from different sources, including the users themselves as they interact with camera apps, automatically infers their training labels without manual intervention and incorporates crowdsourced validation to ensure the robustness of the sanitizer against data poisoning.

Our evaluation with a user study to scan credit cards shows that Aragorn accurately removes sensitive objects in the background while preserving the card scanning utility. Specifically, we show that the accuracy of a user fingerprinting attack drops from 97% to near random chance without any degradation in card scanning functionality. Our implementation of Aragorn in Android's camera subsystem produces frames at a rate of 20 frames per second (FPS), which is adequate for real time operation of most camera apps [40].

Our key contributions include:

- An automated approach to sanitize camera frames for relevant objects while preserving app utility;
- An extensible approach to sanitize camera frames for previously unrecognized objects in a robust manner; and
- A backward compatible and fast implementation on Android.

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.
Proceedings on Privacy Enhancing Technologies YYYY(X), 1–17
© YYYY Copyright held by the owner/author(s).
https://doi.org/XXXXXXX.XXXXXXX

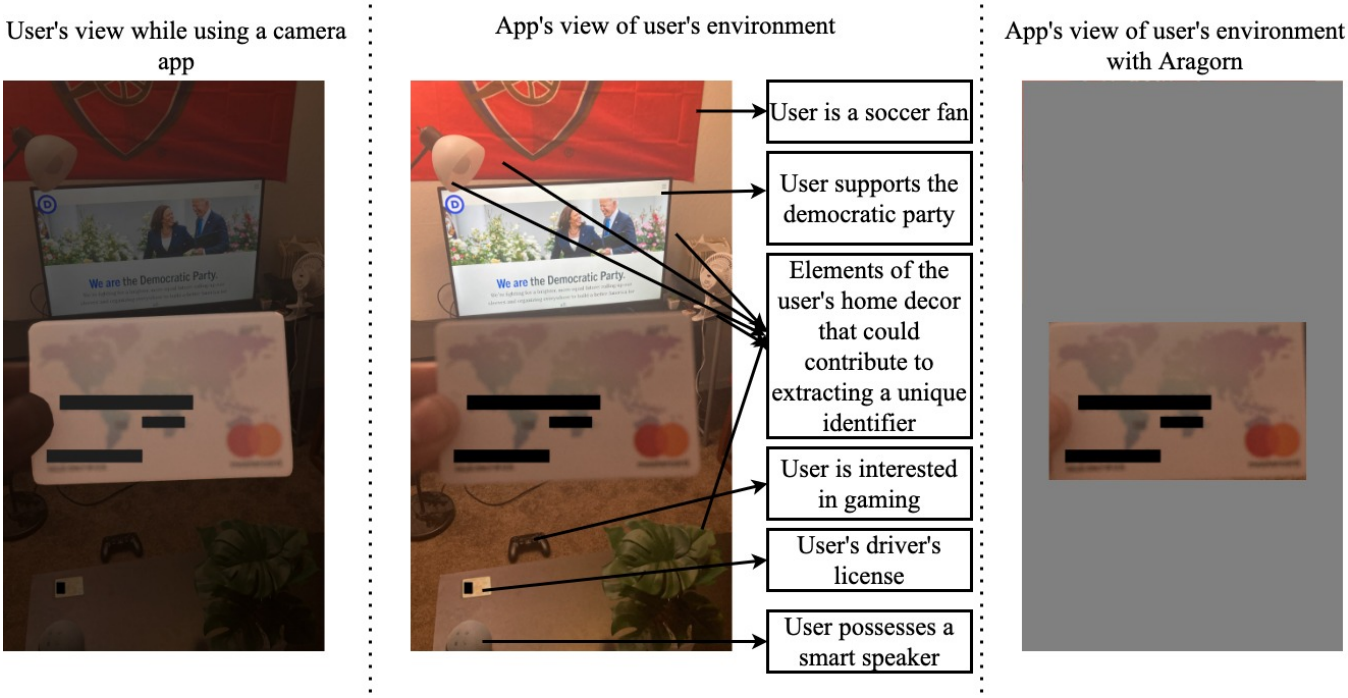

Figure 1: The figure on the left shows the user's perspective while using a camera app to scan her credit card. We have highlighted the credit card in this figure since the user expects the app to only process the card from the camera frame. The figure in the middle shows other inferences that the app can potentially make about the user's environment from regions of the camera frame that do not contain the card. The figure on the right depicts the frame shown to the app by Aragorn to ensure that the app cannot make unintended inferences about the user's environment while preserving its utility.

2 MOTIVATING EXAMPLE

In this section, we use an illustrative example of a credit card scanner to describe the interaction between the different entities involved in mobile camera apps.

Alice makes a purchase on an app, *MalApp* on her phone. MalApp asks her to scan her credit card to prove possession of her payment method [16]. Sitting in her living room, she positions her card in front of the camera as shown in Figure 1. On receiving frames from her camera feed, MalApp processes them to extract the card information and verify her payment method.

We draw attention to the fact that while Alice expects MalApp to only extract credit card information from her camera frames, MalApp can also make other inferences from them. In the example shown in Figure 1, MalApp has access to information present on her driver's license, it can infer her political preference, her interest in gaming, her interest in soccer etc. MalApp could also collude with other apps to extract an identifier for Alice from this visual information (such as the position of her lamp, the soccer poster on her wall, her decorative plants, the color of her walls and other home decor) to track her activity.

Aragorn ensures that the camera frames sent to MalApp only contain credit cards so that Alice can use the app while ensuring that it cannot make any inferences about her environment as shown in the image on the right of Figure 1. More generally, Aragorn provides fine grained control over what can be present in camera frames before allowing apps to access them.

To cater to diverse camera applications, Aragorn has to have the ability to recognize any object that a user may want to reveal to an app, including those that could emerge in the future. While Aragorn could rely on pretrained models or public datasets for some common objects (such as pets, computers, faces etc), it is difficult to even procure training images of more nuanced objects like credit cards. Aragorn adopts a novel training strategy based on knowledge distillation and crowdsourcing to support such cases.

3 BACKGROUND

We characterize the 4 different entities involved in the mobile camera app ecosystem based on the described example: a *user* (Alice), an *app* (MalApp), an *object of interest* associated with the app (credit card) and the user's *environment* (Alice's living room where she scans her card). We describe each of these entities in detail:

User. The user runs an app on their mobile device that requires access to the camera. The user trusts the app to provide some utility (verify the payment method and complete the transaction in the example) by allowing it to access camera frames but wants to ensure that the app does not capture sensitive information about their environment.

	I-Pic [2]	Cardea [44]	Bystander privacy [24]	PrivateEye [40]	WaveOff [40]	Aragorn [this paper]
Compatible with diverse camera apps	55	55	55	51	51	51
Requires minimal manual setup effort	55	55	51	55	55	51
Requires minimal effort to use the camera	51	51	51	55	51	51
Incurs minimal camera latency	55	55	?1	51	552	51
Compatible with live feed	55	55	55	51	51	51

The authors of the paper [24] did not report latency.
Potentially, we expect the latency of WaveOff [40] to increase with the number of marked samples for each object instance.

Table 1: Comparison of Aragorn with other research on protecting sensitive user information captured by camera frames. Compared to existing research, Aragorn caters to a wide range of apps without compromising user experience.

App. The app runs on the user's device and promises some utility to the user by accessing frames from the user's camera. Based on the expected utility, apps are typically associated with a particular object of interest (users may also choose to associate an object of interest with an app that is not tied to any object, like a photo-sharing app). The app may run a computer vision/machine learning model on the user's device to process these camera frames. Malicious apps may attempt to leak information about the user's environment from them.

Environment. The user's environment contains information about the physical location where the user holds the camera. In addition to the object of interest, the environment could contain sensitive information about the user and/or other people who are in the background. Defining which objects in the environment are privacy sensitive is subjective and difficult, with the entire environment itself potentially being sensitive.

Object of interest. Objects of interest are objects present in the user's environment that are tied to the expected utility of the app. These are also the only objects that the user has consented to allow the app to access.

3.1 Related Work

Hasan et al. [24] proposed a blocklist based approach to protect the privacy of bystanders in images by using visual information to automatically detect and obscure them. Their approach being strongly tied to bystanders means that they cannot detect other sensitive objects that could be present in the environment. This limits their extensibility and makes them unsuitable for protecting users from untrusted mobile camera apps in general. In addition to not being extensible, other blocklist based systems, such as Cardea [44] and I-Pic [2], also require significant user effort for operation. For example, I-Pic only protects the privacy of bystanders faces and also requires them to broadcast their presence, visual signatures privacy policy etc. In order to adopt their approach to protect users from diverse camera apps, users would have to identify sensitive objects in their environment in different contexts and provide appropriate visual signatures and privacy policies for each context which would hamper their user experience.

In contrast, allowlist based systems like PrivateEye and WaveOff [40] let users define objects of interest using visual markers and treat everything else as sensitive. Subsequently when an app accesses the camera, they run computer vision algorithms to detect the object of interest in camera frames and block out everything else. While extensible to different objects, their approach still poses significant burden on the user. PrivateEye requires users to manually draw markers around the object(s) of interest prior to using the camera and requires them to carefully position the camera to ensure that the markers are visible.

WaveOff provides users with a graphical user interface that they can use to capture and mark their objects of interest prior to using a camera app. Once marked, users are protected from camera apps without any additional effort. The accuracy of WaveOff, however, is dependent on the user's skill in precisely marking these objects. Thus, only careful users who pay close attention while marking these regions can benefit from WaveOff. Additionally, WaveOff only detects object instances and not objects in general, which increases the effort required from users to mark each object instance. For example, a user with multiple credit cards who seeks to use a card scanning app, would have to capture and mark each credit card before using the app. We also note that WaveOff struggles to reliably detect certain objects such as human faces (which are commonly associated with camera apps) even with precisely marked references (see its evaluation in §7.5).

Aragorn also follows an allowlist based approach to associate apps with different objects of interest to ensure that apps can only see relevant objects from camera frames. In contrast to prior work, Aragorn leverages deep learning to accurately localize the object of interest in camera frames while incurring minimal overhead. To extend support to objects that were previously not recognized by Aragorn , we employ a novel training strategy that uses training images from different sources and automatically infers their training labels. Aragorn only requires users to initially associate objects with camera apps and occasionally answer yes or no questions to ensure the robustness of our trained model against data poisoning. Thus, with Aragorn , users get privacy protection from a wide range of camera apps without incurring significant effort. We summarize the comparison of Aragorn with prior approaches in Table1.

3.2 Threat model

In our threat model, a user seeks to use a camera app on their phone. We assume that the user has explicitly provided permission to the app to access the camera. Although the user has granted this permission, the app is not trusted with the information it could potentially infer from the camera frames beyond the intended object of interest. For example, in the case of a credit card scanning app, we trust the app with information about the credit card, but not

with the background that captures information about the user's scanning environment.

We assume that the app could make privacy-invasive inferences about the user's environment (e.g., by running machine learning models) or transmit information about the environment to a third-party. While the app is unconstrained with what it can do with information about the user's environment, we assume that it cannot tamper with the phone's operating system.

We envision OS vendors (such as Google or Apple) implementing Aragorn within the OS as part of their efforts to protect user privacy [5, 22]. Integrating Aragorn within the OS protects it from the app. Aragorn relies on a trusted remote server (maintained by the OS vendor in our threat model[1]) referred to as Aragorn server when extending support to new objects. We assume that the majority of users are not malicious and cooperate with Aragorn to improve user privacy. However, a small fraction of users can collude with one or more malicious apps to compromise Aragorn .

Under this threat model, our goal is to protect user privacy without degrading user experience or app functionality over a wide range of camera apps.

4 OVERVIEW

4.1 Design Principles

Devise a task-agnostic architecture: Camera apps accomplish a wide range of tasks. Devising independent privacy solutions for each task would not scale, would be difficult to maintain and could potentially increase Aragorn 's attack surface. Accordingly, Aragorn introduces novel mechanisms to train, evaluate, and deploy a machine learning model (referred to as the sanitizer) that detects a given object of interest associated with an app and sanitizes the camera frames by obscuring everything else (§5.3).

Minimize load on users: Aragorn seeks to protect the privacy of all users, regardless of their intent to take up manual or cognitive load. In contrast to prior work that requires users to provide precisely marked samples of their object of interest [40], Aragorn only requires a fraction of its users to occasionally answer yes or no questions to secure the privacy of all users (§5.3.3).

Exercise the principle of least privilege: Since camera apps are untrusted in our threat model, Aragorn operates within the OS and limits the capabilities provided to them. For instance, Aragorn does not permit apps to directly access camera frames, instead, it lets apps access sanitized frames that only contain the intended object of interest (§6).

4.2 Aragorn architecture

Aragorn runs within the mobile operating system to intercept camera frames before they can be accessed by camera apps. Aragorn employs a machine learning based object detector, referred to as the sanitizer , to ensure that apps can only access a pre-established object of interest from camera frames. Aragorn also contains an auxiliary remote server referred to as the Aragorn server to help extend the sanitizer to detect previously unrecognized objects.

Within the operating system, Aragorn operates in one of two workflows based on whether the sanitizer can reliably detect a given object of interest (we refer to this as the *steady-state* workflow), or requires additional training to do so (we refer to this as the *transient* workflow).

[1]We assume that users trust the OS vendors and thus, also trust Aragorn server.

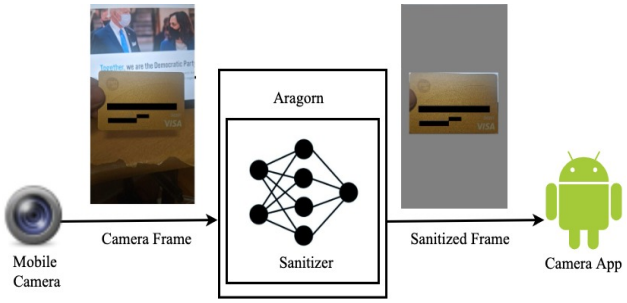

Figure 2: In the steady-state workflow, Aragorn 's sanitizer ensures that camera frames only contain the preestablished object of interest and obscures everything else before third-party camera apps can access them.

During its initial invocation in the steady-state workflow, an app specifies its object of interest from a list of supported objects (such as credit cards, driver's licenses, etc.), which can optionally be overridden by the user. Then, during the app's execution, the sanitizer blocks everything from camera frames retaining just the object of interest before sending them to the app. We summarize this workflow in Figure 2.

Aragorn predominantly operates in the steady-state workflow. However, in case apps require support for a new object (such as scanning a COVID vaccination card as discussed in §1), Aragorn switches them to the transient workflow. Prior to operating in this workflow, an app requests Aragorn to extend support to this new object. Upon approving the app's request, Aragorn allows it to process complete camera frames to ensure that users can use the app, while using these frames to train the sanitizer with assistance from Aragorn server. Eventually, once the sanitizer can recognize this object, Aragorn switches the app over to the steady-state workflow.

5 DESIGN

In this section, we first provide a detailed account of Aragorn 's steady-state and transient workflows from the perspective of both apps and users. We then describe the design of the sanitizer which drives the overall design of Aragorn .

5.1 Steady-state workflow

Aragorn invokes the steady-state workflow in cases where Aragorn 's sanitizer has already been trained to reliably detect and recognize the object of interest in context. Such an object could either correspond to a common user object that is well represented in public datasets [29, 32] (pets, computers, food items etc.), and hence supported by the sanitizer during Aragorn 's initial deployment, or a more nuanced object for which the sanitizer was previously retrained via the transient workflow.

A camera app specifies a requirements file containing its intended object of interest from a list of objects supported by Aragorn . Subsequently, when the user invokes the app for the first time,

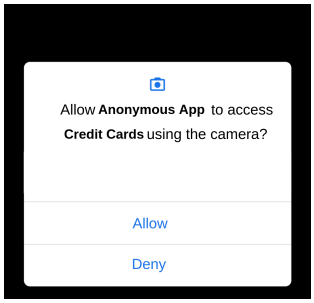

Figure 3: Dialog box presented to users during an app's initial invocation. Users can use this interface to grant access to a specific object requested by the app, via the camera.

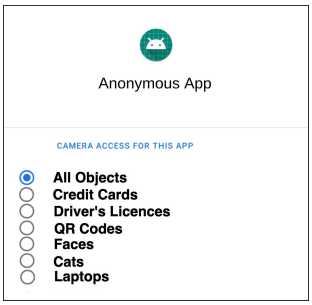

Figure 4: Users can optionally use this interface to override previously assigned objects of interest to camera apps, or assign objects of interest to those that did not have one previously.

Aragorn presents them with a permission dialog asking to grant access to the specified object of interest as shown in Figure 3. Aragorn also allows users to optionally override a previously assigned object of interest to an app using the interface shown in Figure 4. Users can also use this interface to assign an object of interest to an app whose utility is not tied to any particular object or an app that did not have one previously (e.g. assigning pet to a photo-sharing app when uploading the image of a pet on the app).

Upon establishing the object of interest, the app requests camera frames as part of its execution. Aragorn then uses its sanitizer to only pick out the object of interest from camera frames and blocks out the rest to produce *sanitized frames*, which are then sent to the app. Aragorn also performs JPEG compression on these frames to purge high-frequency camera fingerprints (i.e., Photo-Response Non-Uniformity or PRNU [33]).

While the app only gets access to the sanitized frames, Aragorn renders camera frames completely onto the display to ensure that the user experience is not hampered. Accordingly, we do not allow apps to read the display to stop them from accessing the complete camera frames. However, we do allow apps to write to the display. While granting write access helps apps implement their own interfaces to better engage with users, malicious apps could abuse this to attempt to invade user privacy by asking them to position objects other than the established object of interest. Aragorn thwarts such attempts in the steady-state workflow, since the sanitizer can accurately detect the object of interest. As a concrete example, consider a case where the established object of interest is a credit card. Even if an app attempts to trick users by asking them to position a different object, say a driver's license next to the card (or instead of the card), Aragorn would not detect this object as the object of interest, thereby protecting user privacy.

However, even in the steady-state workflow, Aragorn needs to communicate with users to get their feedback on its predictions (§5.3.3), and apps can abuse their write access to interfere with this communication. To prevent this, Aragorn only communicates with users for their feedback after the app has completed its execution.

5.2 Transient workflow

Aragorn invokes the transient workflow whenever there is a need to extend the sanitizer to detect a previously unrecognized object. When an app wishes to access such an object from camera frames, it submits a request to Aragorn server, specifying the name of the intended object.[2] Once the request is approved, Aragorn allows the app to initially operate in the transient workflow. In this workflow, Aragorn lets the app process camera frames completely (so that users can still use the app) while also using them to train the sanitizer for the new object. Since OS vendors (who are trusted in our threat model) maintain Aragorn server, one way to train the sanitizer would be to transmit these frames to Aragorn server for centralized training. However, users may consider Aragorn server to be honest-but-curious in which case they would not trust it with the confidentiality of their data. Thus, an alternate way to train the sanitizer would be to adopt federated learning. Once the sanitizer is trained to recognize the object, Aragorn adds it to its list of supported objects and moves the app into the steady-state workflow.

During the initial invocation of the app, Aragorn informs users that they are in the transient workflow and requests permission to use their frames for training. Since Aragorn does not need training images from all users (§7.3), it allows users to opt out of contributing their data to train the sanitizer .There is no feedback phase in this workflow since the sanitizer is not invoked.

5.3 Sanitizer

The sanitizer is an object detector (YOLOv3 [42]) that can classify and localize a given object from its input. Given a camera frame as input, the sanitizer predicts the classes and locations of objects contained in it. The classes predicted by the sanitizer map to one or more of the objects it has been trained to recognize and the locations are bounding boxes of the predicted classes within the frame. In the steady-state, Aragorn preserves those objects from the sanitizer 's predictions that correspond to the established object of interest and blocks everything else to produce a sanitized frame which is then made accessible to the app. In case none of the detected objects in a camera frame map to the specified object of interest, Aragorn blocks the entire frame.

[2]We discuss supporting requests from users in Appendix 10.6.

Figure 5: Examples of frames sent to a credit card scanner, a QR scanner and a face verification system. In all 3 examples, we see that the respective objects of interest (credit card, QR code or face) are also the salient objects present in the frame.

5.3.1 Sanitizer training. To train the sanitizer (a supervised object detector), we need labeled frames containing the intended object. To localize and classify this object, the labels should contain a class label as well as a localization (bounding box) label. This data needs to be gathered from a reliable source. While in some cases, we may be able to source this data from public datasets [29, 32], there could be more nuanced objects (such as credit cards) that are not well represented in any public dataset. We cannot simply source this training data from the app since we don't trust it in our threat model. Thus, we source this training data (unlabeled frames) from the users (most of whom are trusted in our threat model), while they use the app. We adopt a knowledge-distillation based strategy to automatically label this data to train the sanitizer . In this section, we describe our training strategy when employing centralized approach to gather images at Aragorn server. We discuss adopting the training strategy with a federated approach in Appendix 10.1.

We first describe the flow when the training data is provided by good users and then discuss potential attacks from malicious users and their mitigations. To gather training data from users, we allow the app to initially run in the transient workflow where it can access complete camera frames. In the intended execution of the app, we expect these frames to contain the object of interest and thus, we forward them to the Aragorn server (from the users who consented to share their frames). Note that users merely interact with the app as usual, and we use frames from their interaction as training images. Once we have gathered this data, we have training images containing the object of interest. The name of the object for which the app requested support, serves as a common class label for these images. However, we do not yet have the location (bounding box) of the object within these images.

To compute the bounding box labels for our training data, we rely on the fact that in most frames the object of interest is also expected to be the salient object. Figure 5 shows camera frames passed to a credit card scanner, a QR scanner and a face verification system, and we note that for each frame the object of interest is also the salient object in the frame. Accordingly, we use an off-the-shelf salient object detector (U^2-Net [38]) to localize the object of interest in these frames. The localization label coupled with the common object label corresponding to these frames is used to train a supervised object detector, which serves as our updated sanitizer

Figure 6: In this figure, the red box shows the prediction of our supervised YOLOv3 [42] based sanitizer , while the blue box shows the prediction of a U^2-Net [38] salient object detector.

. This entire training procedure is executed at the Aragorn server, and once trained, the new sanitizer is shipped to user devices (as an update to the OS).

Once trained to recognize the new object, Aragorn switches the app back to the steady-state workflow. We note that during the initial phase, when an app operates in Aragorn 's transient workflow, user privacy is unprotected from the app. We refer to the state when Aragorn is able to protect user privacy in context of a new object as the state of attaining "herd protection" for that object. We quantify the duration of the unprotected phase in terms of the number of images required until herd protection is attained in §7.3.

Can a salient object detector function as the sanitizer ? A salient object detector alone could potentially be used as the sanitizer since we know that the object of interest is expected to be the salient object in most cases. However, due to its inability to classify the object of interest, it can pick any salient object as the object of interest. While we can address these concerns by augmenting the salient object detector with a classifier, we instead choose to use an integrated supervised object detector for the following reasons. First, an integrated supervised object detector combines classification and localization into a single model [42]. The two tasks inform each other leading to more accurate localization and classification as opposed to training two separate models, one for each task.

Second, at run time, we get both classification and localization results from a single model inference with a supervised object detector, as compared to two separate inferences, one with a salient object detector and the other with a classifier. Thus, a supervised object detector gives Aragorn higher efficiency.

Anecdotally, we see in Figure 6 that even in the presence of multiple objects in a frame, Aragorn 's sanitizer (a supervised object detector) more accurately localizes to the correct object (credit card in the figure) as compared to U^2-Net (a salient object detector). During training, the sanitizer learns to distil common object information from multiple salient labels to only pick out the relevant object.

5.3.2 Threat of data poisoning. When gathering data from users, attaining herd protection relies on users operating as intended by only positioning the expected object of interest in front of the camera. However, since not all users are trusted in our threat model

Figure 7: This figure shows a user positioning a credit card next to a driver's license to provide poisoned data to Aragorn . If unchecked, Aragorn 's sanitizer could learn the driver's license as being part of a credit card, and detect them from users' backgrounds whenever they use a card scanning app.

(or users can be tricked into scanning incorrect objects by malicious apps), herd protection is subject to data poisoning attacks from users hired by malicious apps.

As a concrete example, consider a scenario where we seek to attain herd protection for credit cards. A malicious app, intending to evade Aragorn to invade user privacy could initially hire users who intentionally scan other objects, such as driver's licenses, in addition to credit cards. As a result, once trained, Aragorn 's sanitizer would also recognize driver's licenses as credit cards. At this stage, once other users start using the app, any licenses present in the background would also be exposed to the app. We show a poisoned image sample in Figure 7. More generally, users colluding with an app can poison the training data with any generic objects that are expected to be in user backgrounds to help the app interfere with Aragorn 's intended operation.

5.3.3 Crowdsourced validation to defend against data poisoning attacks. Aragorn relies on two different crowdsourced strategies to defend against data poisoning attacks. The first strategy, which we refer to as Aragorn 's *preemptive defense*, operates during the transient workflow and discards poisoned samples from the gathered images. The second strategy, which we refer to as Aragorn 's *reactive defense*, operates in the steady-state workflow and detects if a previously trained sanitizer has been poisoned. We rollback and retrain the sanitizer with fresh data on detecting that it had been poisoned.

Aragorn 's preemptive defense: This defense relies on our assumption that the majority of Aragorn 's users (i.e. all mobile users in general) can be trusted, even if all the users of a particular app at a given point in time are potentially malicious. We thus, take help from users to detect and discard poisoned images in the data we gather from other users. Concretely, we first detect the salient objects in the gathered images and block out the other portions. Then, we blur the regions containing the salient objects and randomly ask other users if these images only contain the expected object of interest. We only include the corresponding images in our training set for which users respond with a "yes".

Figure 8: Example of an image shown to users as part of our preemptive defense to combat data poisoning. We ask users if the blurred regions in this image only contain a credit card.

Figure 8 shows an example of an image displayed to users when we're gathering data to detect credit cards. In this case, we randomly ask other users if the blurred portion of the image only contains credit cards.

Drawing from existing research [2, 24, 44] we blur images to retain overall information about the object present in the image, while obscuring object specific information. However, despite blurring, some information can still leak about the object of interest. For example, in our evaluation in §7.4.1, we see that some users were able to guess the gender and hair color of users from their driver's licenses. We argue that targeted attempts by malicious users to gather this data are not possible since Aragorn server randomly picks users to validate user data. When OS vendors deploy Aragorn at a large scale (i.e., with billions of Android and iOS users) the probability of a particular user's images being validated by the same user would be negligible [3]. Nevertheless, Aragorn server can also choose to not employ the preemptive defense for an object that it deems to be sensitive and rely exclusively on the reactive defense to thwart data poisoning attacks against that object. As we discuss in the next section, doing so would require more frequent user involvement, which could lead to poor user experience.

This defense would be ineffective when detecting certain objects that look similar to other objects, especially since users only get to see blurred images. For example, when asking users if an image only contains a particular document, it would be difficult for users to respond appropriately since the contents of the document would not be visible to them. In such cases, users can respond with a "Not Sure" option. Aragorn 's reactive defense helps cover such cases.

Aragorn 's reactive defense: This defense relies on the fact that all apps, including malicious ones that hire users to provide poisoned data, would eventually engage with users who are not controlled by the app. Thus, once a camera app completes its execution in the steady-state workflow, we annotate the frames produced during the execution with the sanitizer 's predictions and show them to users for their feedback. Here, we do not blur or obscure the images in any way, since users only validate predictions on their own camera frames. Like the preemptive defense, users only

[3]When deployed at a large scale, the frequency at which we request a response from the same user would also be low, thereby having minimal impact on user experience.

respond with a “yes" or a “no" to indicate whether they were confident in the sanitizer 's predictions (unlike the preemptive defense, the images are unblurred, making it easier for users to respond). If the number of negative responses from users crosses a particular threshold, we infer that our trained sanitizer has been subject to poisoning and roll it back to a previous version known to be robust to poisoning. We transition apps associated with the suspected poisoned object back to the transient workflow to gather fresh training images. We trigger manual inspection in case the sanitizer needs to be rolled back repeatedly to detect and stop apps that attempt to force Aragorn back to the transient workflow. We keep running this defense periodically instead of running it for a fixed period to ensure that apps cannot evade this defense by initially hiring users to provide false feedback. The reactive defense also serves as a way to assess the generalizability of the sanitizer and detect potential changes to the design of the object of interest.

While requiring all users to run the reactive defense during all their interactions with camera apps would allow us to extensively evaluate the sanitizer , doing so would come at the cost of user experience. We thus, run this defense across an optional channel for all users, where they can choose to not provide feedback and a forced channel where random users are necessarily asked to provide feedback. The optional channel allows us to evaluate against diverse use cases and backgrounds to better identify potential corner cases where the sanitizer makes mistakes (such as those where mistakes happen in the presence of specific objects in the background). However, relying solely on the optional channel could lead to not only insufficient, but also potentially false feedback from users colluding with the app. We, thus, correlate the number of mistakes made by the sanitizer in the optional and forced channels to come to a decision on whether the sanitizer needs to be retrained. Despite both channels, there could be some cases where mistakes occur in presence of specific triggers that could go undetected. For example, the sanitizer could be poisoned with a backdoor such that it correctly detects credit cards in most cases, but detects the background too, if a specific trigger object, such as a pair of glasses is present in the background [53]. However, such cases can be mitigated using the preemptive defense. Thus, Aragorn relies on both, the preemptive and reactive defenses to cover for each other's shortcomings to defend against a wide range of data poisoning attacks.

6 IMPLEMENTATION

We implement Aragorn within Android 11 by modifying the camera subsystem of the Android Open Source Project (AOSP). Since we envision Aragorn being implemented by mobile OS vendors, we integrate it within the Android OS in our prototype instead of implementing it as a standalone library. This also provides protection from apps that attempt to tamper with or bypass Aragorn to directly access the camera.

Android's camera subsystem is broadly split into 3 distinct processes that communicate using binder interfaces: the hardware abstraction layer (HAL), the camera application framework, and the camera service. On one end, the HAL directly interacts with the camera driver and at the other, apps use camera application frameworks (such as Camera2 or CameraX) to access the camera. The camera service connects the application framework to the HAL

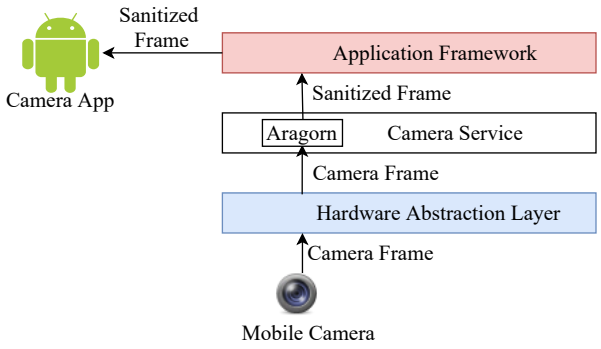

Figure 9: The camera service is ideally suited to incorporate Aragorn , since it is both, hardware independent (unlike the hardware abostraction layer) and isolated from the app in a separate process (unlike the application framework).

by exposing standard interfaces to both. Concretely, the application framework uses these interfaces to submit a request to the HAL to capture camera frames, and the HAL invokes callbacks present in the camera service to deliver frames in response.

We integrate Aragorn 's sanitizer with the camera service to reap dual benefits of preventing apps from bypassing the sanitizer as well as not having to maintain multiple hardware dependent implementations. Since the camera service runs in its own process and is the only process that can access the HAL, apps would necessarily have to compromise the OS to bypass the sanitizer . Further, since the camera service is hardware independent, the same implementation can be used on all devices. Alternatively, integrating the sanitizer within the HAL would also protect it from apps, since it is the only process that can access the camera driver, but doing so would require maintaining different implementations based on the hardware. In contrast, integrating the sanitizer with application frameworks would not require different implementations, but apps would be able to evade Aragorn by directly accessing the camera service. Figure 9 summarizes this discussion.

Aragorn 's sanitizer that supports 3 distinct objects (credit cards, QR codes and faces) occupies 5.6 MB on disk. We implement the sanitizer using TensorFlow Lite C++ API [49]. In the steady-state workflow, an app that complies with Aragorn would provide a requirements file that specifies its intended object of interest from the global list of objects recognized by Aragorn . As discussed in §5.1, users can choose to grant or deny permission to the app to access the specified object, or assign a different object of interest to the app (again from the global list of objects). However, Aragorn is also compatible with apps that do not produce such a requirements file, since users can assign objects of interest to them (from the global list of objects recognized by Aragorn). Thus, Aragorn is compatible with all camera apps regardless of their awareness of Aragorn or their intent to comply.

On a Google Pixel 2 phone, without using hardware acceleration, the sanitizer uses the equivalent of around 2 CPU cores and consumes 42 MB of memory. Performance evaluation of this implementation shows that the sanitizer takes 38ms on average for

execution, correspondingly producing frames at roughly 20 frames per second. The performance of the sanitizer is largely unaffected as it scales to support more objects. See Appendix 10.5 for details.

7 EVALUATION

7.1 Privacy

We first evaluate Aragorn 's effectiveness in sanitizing camera frames against an attack to identify users based on the background information captured in their camera frames. We then evaluate Aragorn 's ability to remove sensitive objects in the background.

7.1.1 User Identification. We first show an example of how malicious apps can identify users using background information captured in their camera frames as a fingerprint. We then evaluate Aragorn 's effectiveness in thwarting such an attack. While we present results with credit cards as the object of interest, our goal with these experiments is to explore the extent to which users can be identified from the background information in the camera frames irrespective of the presence of identifying information in the object of interest itself.

User Study. We conduct a user study where we ask users to visit a link containing 30 different credit card images. Visiting the link renders the credit card images on the web page, one at a time, allowing users to navigate between them. We ask users to scan these cards using a custom iOS app using the Daredevil card scanner [15]. The app sends their scan videos to our server for analysis.

From these videos, we obtain 56,973 frames from 19 users in our user study.[4] Since most users in our study scanned all 30 cards in one sitting, the background for each user does not significantly vary across frames. This simulates a scenario where it is easier for malicious apps to extract fingerprints, and is correspondingly more challenging for Aragorn .

Attack: Closed-world scenario We first consider a multi-class attack setting, where the malicious app has representative images from each of its users. Given a new camera frame, the app's goal is to use the background information in the camera frame to identify a returning user. Concretely, we train a 19-class classifier, with each class representing a distinct user in our study. While this closed-world scenario is not practical in the real world, it simulates a harder use case for Aragorn . (We subsequently evaluate Aragorn with a more practical attack.)

To ensure that the model only uses background information to identify users, we block out the credit cards in the frames and also purge high-frequency fingerprints tied to the device, i.e. PRNU [14, 33] via JPEG compression [52] (See Appendix 10.8).

We train the classifier using a convolutional neural network (CNN) on frames from the user study. The model attains an accuracy of 97.32% on a 10% held-out test set.

Attack: Open-world scenario We now consider a more practical scenario, where a malicious app does not need to know the total number of users to identify them. The app employs a model, which we refer to as a *background fingerprint model*, to determine if the background information contained in a pair of images map to the same user. The app needs at least one reference frame for a user, and all subsequent frames obtained are compared against the reference to identify the user.

Taking inspiration from existing research [14, 30, 43], we use the triplet loss to train the background fingerprint model. We train the model to produce embeddings such that the Euclidean distance between those extracted from frames captured by the same user are below an empirically determined threshold, while the distance between those from two different users is above the threshold. We train this model on frames from 15 users in our user study and evaluate it on the remaining 4 users. On 1,000 random pairs of frames captured by the same users, the model makes correct predictions on 804 pairs. On another set of 1,000 random pairs of frames from different users, the model makes correct predictions on 863 pairs. The overall test accuracy of the attack is 83.41%. These results demonstrate that malicious apps can use background information in credit card scans to identify users.

	Multi-Class Classifier	Background Fingerprint Model
No sanitizer	97.32%	83.41%
With sanitizer	5.25%	50.09%

Table 2: Aragorn vs user identification attacks. The first row shows the accuracy of two different models that identify users from the background information of their camera frames. The second row shows the degradation in the accuracy of the same models on Aragorn 's sanitized frames.

Object of interest	Classification False Positive Rate	Localization False Positive Rate
Credit cards	10.99%	7.96%
Faces	0.00%	8.81%

Table 3: Low false positive rates in detecting credit cards and faces indicate that Aragorn 's sanitizer captures minimal background information, leading to better user privacy.

Aragorn vs. user identification attacks. We test models trained in both scenarios on sanitized frames produced by Aragorn . We use the sanitizer to sanitize images from the same test sets described above while not changing the training images. Table 2 shows that the multi-class classifier's attack accuracy drops from 97.32% on complete camera frames to 5.25% on sanitized frames. Similarly, the background fingerprint model's attack accuracy drops from 83.41% on complete camera frames to 50.09% on sanitized frames.

For both attacks, the attack accuracy drops to near random chance after sanitization by Aragorn . Specifically, 5.25% accuracy for 19-class classification is equivalent to randomly guessing between 19 classes. 50.09% accuracy for the background fingerprint model is also equivalent to random guessing (binary decision of whether two frames were captured by the same user). We were unable to train similar fingerprint models when training on sanitized frames. See Appendix 10.3 for details.

In conclusion, these results show that Aragorn 's sanitizer removes the background information in camera frames that can be used by apps to identify users.

[4] Our university's IRB reviewed our user study and exempted it from IRB.

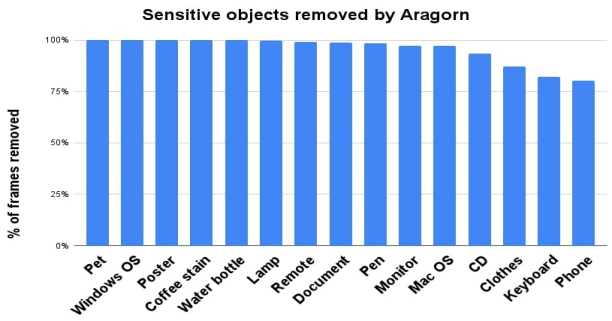

Figure 10: Chart showing the proportion of frames in our card scanning user study where Aragorn removed sensitive objects from the background.

7.1.2 Removing sensitive objects. We evaluate the effectiveness of Aragorn 's sanitizer in removing background information. This evaluation measures the sanitizer 's ability to accurately detect and localize to the object of interest. In context of user privacy, we are interested in measuring the sanitizer 's classification false positive rate, i.e., mistakenly detecting a background object as the object of interest, and the localization false positive rate, i.e., regions that were predicted to contain the object of interest but contain the background.

Table 3 reports the results. For credit cards as the object of interest, on a set of 2,206 frames obtained from 20 random scan videos from our user study (§7.1.1), the sanitizer has a classification false positive rate of 10.99% and a localization false positive rate of 7.96%.

Similarly, for faces as the object of interest, we run our quantitative evaluation on images from the public CASIA-Webface dataset [13]. We pick 200 random face images from this dataset and combine them with 200 random indoor background images (to evaluate images that do not contain a face) from the indoor scene recognition dataset [39]. On this dataset, the sanitizer has a classification false positive rate of 0% and localization false positive rate of 8.81%.

These low values for the false positive metrics indicate lesser information about user background being captured in the sanitized frames, leading to improved user privacy.

We also qualitatively evaluate Aragorn 's ability to remove potentially senitive information from camera frames. On 2,206 frames from 15 random videos in our user study dataset, we manually identified 15 potentially sensitive objects (such as a document containing the user's address, a pet, etc.). We then measure the ratio of frames where these objects were blocked by the sanitizer to the frames that originally contained them. Figure 10 shows that the sanitizer removes 11 out of the 15 identified objects from 97% of frames and removes the other 4 objects on over 80% of frames that contained them. Overall, these results show that Aragorn 's sanitizer significantly reduces the number of frames containing potentially sensitive objects in our user study.

7.2 Utility

Aragorn should not prevent apps from providing their intended utility to users. We evaluate the impact of Aragorn on the utility of a public credit card scanner, Daredevil [15] and a public face verification system, MobileFaceNet [48].

	Precision	Recall
No sanitizer	96.15%	83.33%
With sanitizer	100.00%	86.66%

Table 4: Results showing that Aragorn does not affect the utility of Daredevil card scanner [15].

	Accuracy on same identity	Accuracy on different identities
No sanitizer	91.09%	92.55%
With sanitizer	92.01%	92.02%

Table 5: Results showing that Aragorn does not affect the utiltiy of MobileFaceNet [48] face verification system.

7.2.1 Credit card scanner. We measure Daredevil credit card scanner's precision and recall on a subset of 30 scan videos from our user study §7.1.1. We then sanitize the frames from these videos and measure the card scanner's precision and recall on them for comparison. Here, precision refers to the ratio of the number of scan videos on which the card scanner is able to predict the right number to the number of scan videos on which it predicted a credit card number (including incorrect credit card numbers). Recall is the ratio of the number of scan videos on which the card scanner is able to predict the right number to the total number of scan videos.

The metrics in Table 4 show a slight increase in both precision and recall when using Aragorn . We attribute this increase to the removal of extraneous information, helping the card scanner localize the card number better. Running Daredevil card scanner on our prototype of Aragorn yields an average scanning duration of 6.72 seconds. Equivalent runs without incorporating Aragorn into the camera subsystem yield an average scanning duration of 6.52 seconds. Thus, Aragorn only increases the latency of Daredevil card scanner by 3.07%.

7.2.2 Face verification. We first measure the verification accuracy of the MobileFaceNet face verification system [48] on a subset of 2,104 images from 220 distinct identities from the standard Labeled Faces in the Wild [25] dataset. We then measure its verification accuracy on the same set of identities after sanitization.

We evaluate 10,000 random pairs of images of the same identities and 10,000 random pairs of images of different identities selected from the dataset. Before sanitization, MobileFaceNet has an accuracy of 91.82% in detecting if image pairs belong to the same or different identities. Repeating the experiment on sanitized images yields an accuracy of 92.01%. We report detailed results in Table 5.

MobileFaceNet crops to a tight bounding box around the user's face as an initial preprocessing step before extracting an embedding for verification. We suspect that this step extracts the same crop in both cases, leading to similar accuracies. We do not measure the latency introduced by Aragorn to MobileFaceNet since their system does not work with a live camera feed.

Num. Images		Card		Driver's License	
Card	DL	Accuracy	IOU	Accuracy	IOU
449	85,669	100.00%	84.85%	100.00%	93.30%
85,649	461	100.00%	89.71%	100.00%	90.40%
1697	1719	100.00%	83.52%	100.00%	92.24%
0	882	N/A	N/A	100.00%	82.43%
869	0	100.00%	80.12%	N/A	N/A

Table 6: Results showing the number of images needed to attain herd-protection with credit cards and driver's licenses. We measure the number of images needed for each object to attain acceptable localization and classification accuracies on a held-out test set. The results show that the sanitizer needs less than 500 images of the new object, when there is a large number of images of the other object. Additionally, to support both objects, the sanitizer needs close to 1700 images of each object. The sanitizer needs less than 900 images to support each object in isolation.

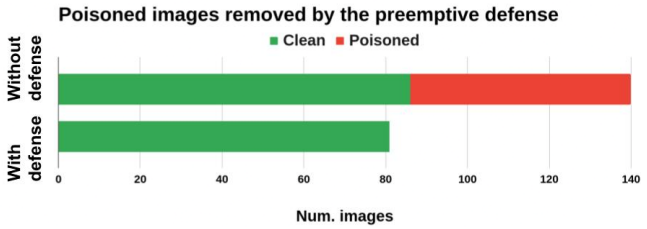

Figure 11: Chart showing the effectiveness of Aragorn 's preemptive defense in detecting poisoned images. Based on the majority of responses, we were able to remove 100% of poisoned images in our dataset while losing less than 6% of clean images.

7.3 Herd protection

In this section, we quantify the number of training images to be gathered to attain herd protection for certain objects. It is important to estimate when herd protection can be attained since user privacy is unprotected until then. Accordingly, in this section, we evaluate training the sanitizer in two potentially difficult scenarios, one where we assume that we had previously gathered driver's license images and need to gather images of credit cards and vice versa. We consider these to be difficult scenarios since these objects look similar and have the same form factor, thereby making it difficult for the sanitizer to discern differences between them.

We report the number of training images we need for each object given that we had previously gathered data for the other object. Concretely, we report the number of credit card images we need to train the sanitizer in three different use cases: (1) where we had previously gathered a large number of driver's licenses (85, 669 images), (2) previously gathered fewer driver's licenses (1, 719 images) and (3) not gathered any driver's licenses. We then report the equivalent metrics for the number of driver's license images needed given that we had previously gathered credit card images. We computed these metrics by iteratively increasing the number of images of the new object until we attain acceptable classification and localization accuracies.

Dataset. We use two datasets corresponding to scanning credit cards and scanning driver's licenses. For credit cards, we use the data from our user study §7.1.1 and for driver's licenses, we procure a similar dataset by scanning 30 different driver's licenses obtained from the authors of Daredevil [15].

Labeling. We use the U^2–Net salient object detector [38] to generate the bounding boxes and assign class labels based on the objects in the dataset (i.e., credit card or driver's license).

Results. Table 6 reports our results on held-out test sets containing 270 samples for each object. From these results, we see that in all use-cases fewer than 1, 800 images of the new object are sufficient to attain herd protection. Concretely, we see that we attain 100% classification accuracy and over 84% localization accuracy (IOU) with fewer than 500 images of the new object when we had previously trained on close to 86, 000 images of the other object. Similarly, we see that 1, 697 images of the new object are sufficient to attain 100% classification and 83.52% localization accuracy when we had previously trained the sanitizer on 1, 719 samples of the other object. Lastly, training on the new object with no previous history of the other object also requires fewer than 900 images of the object to attain 100% classification and over 80% localization accuracy.

Thus, even if we use one frame per user for training, as long as the app seeking Aragorn to recognize a new object can engage with at least 1, 719 users, we would attain herd protection relatively quickly, even in case the sanitizer needs to detect similar objects like credit cards and driver's licenses. These numbers validate the flexibility in our design where users can choose to opt out of sharing their camera frames with Aragorn server.

7.4 Crowdsourced validation

7.4.1 Preemptive defense. We simulate Aragorn 's preemptive defense with a user study, where we show blurred and sanitized images (using the salient object detector as described in §5.3.3) of clean and poisoned credit cards (we define clean and poisoned credit card images in context of our evaluation in the next paragraph) and ask participants if the images *only* contain credit cards. We also ask them to describe any other potentially sensitive information they are able to glean from these images. We then quantify the proportion of poisoned images that remain after removing images based on responses from the study.

Dataset. We capture clean and poisoned images of credit cards. The clean images simulate users scanning credit cards, where objects could be present in the background but are not intentionally placed around the card. For poisoned images, we either capture images of driver's licenses in place of cards, or position driver's licenses close to credit cards, similar to the image shown in Figure 7. Overall, our dataset contains 140 images, of which 38.75% (54 images) are poisoned.

User Study. We obscure images in our dataset to only retain blurred out salient objects as described previously (§5.3.3). We randomly show these images to participants of our user study (using Google forms) and ask them if each image only contains a credit card. Participants can respond with a "Yes", "No" or "Not

Object of interest	WaveOff		Aragorn	
	Class.	Loc.	Class.	Loc.
Credit cards	80.24%	31.50%	91.32%	79.82%
Faces	50.04%	2.78%	98.16%	75.17%

Table 7: Results showing that Aragorn more reliably detects and localizes to both credit cards and faces when compared to WaveOff.

Sure". Each image also has a text box where participants are asked to describe any potentially sensitive information they can infer from the image.

Results. We ran the user study for 2 weeks and obtained results from 35 different participants. Based on the majority of responses, we remove all 54 poisoned images from our dataset. Of the 86 clean images, participants were unsure of 4 images, and incorrectly marked one clean image as poisoned. Thus, with a conservative approach of treating unsure responses as poisoned, we are able to retain 94.16% of clean images from the dataset, while completely removing all poisoned images (see Figure 11). We manually peruse the descriptive responses from participants and observe that none of them were able to infer any sensitive or personally identifiable information from the images shown to them. At best, they were able to identify certain cards as belonging to Visa or MasterCard, and in some cases guess the hair color or gender of users based on their image in driver's licenses. We were also unable to sharpen these images to recover any sensitive information with simple traditional vision filters as well as complex deep learning algorithms. See Appendix 10.7 for details.

7.4.2 Reactive defense. With the reactive defense, we expect most users to clearly respond to whether the sanitizer had made the right prediction, since they are shown predictions from the sanitizer on unobscured frames from their own camera. To verify this expectation, we ran an experiment where 3 authors of this paper independently observed the sanitizer 's predictions on 10 random frames each from 10 different scans from our card scanning user study (§7.1.1). As expected, all three authors ended up picking the exact frames as frames on which the sanitizer had made mistakes.

7.5 Baseline comparison with WaveOff

In this section, we compare the accuracy of Aragorn and WaveOff to detect objects that are often associated with camera apps. Concretely, we measure the classification and localization accuracies of WaveOff on credit cards and faces and compare them to Aragorn . WaveOff matches BRISK features [31] extracted from a given test image against those from a marked reference to localize to the object of interest in the test image. In our evaluation, we threshold the number of matches to classify the presence of the object of interest within the test images[5] and draw the smallest bounding box that covers all matches to localize to the object of interest.

7.5.1 Comparison on credit cards. We use data from our card scanning user study described in §7.1.1. We pick 1, 021 random frames corresponding to 10 random scan videos, each containing a different card. We then take one sample image of each card in this set and mark a tight bounding box around the card to use as a reference image for each scan (to evaluate WaveOff in context of a careful user). At the empirically determined optimal threshold, WaveOff has a classification accuracy of 80% on this test set while Aragorn has a classification accuracy of 91.32%. Correspondingly, WaveOff has a localization accuracy of 31.50%, while Aragorn has a localization accuracy of 79.82%. We summarize these results in Table 7.

7.5.2 Comparison on faces. We observe that WaveOff is unable to detect human faces, despite using tightly cropped reference images, even when user backgrounds do not alter significantly. We first quantitatively evaluate WaveOff on 200 random face images from the CASIA-Webface dataset [13] and 200 random indoor background images from the indoor scene recognition dataset [39]. Across 10 different thresholds for the number of matchings, WaveOff has a rough classification accuracy of 50%, with the highest recorded accuracy being 50.04%. WaveOff correspondingly has a localization accuracy of 2.78%. On the same dataset, Aragorn reports a classification accuracy of 98.16% and a localization accuracy of 75.17%. While the CASIA-Webface dataset allows us to evaluate with a large number of faces, the images are not representative of users looking into their mobile cameras. We thus run a qualitative evaluation of WaveOff, using 10 images each of 2 authors of this paper looking into the camera. On these images too, we observe that WaveOff was unable to localize to the face.

We attribute the lower accuracy of WaveOff to the inability of BRISK features to sufficiently represent variations in human faces, with Aragorn overcoming this using learned features.

8 LIMITATIONS

Requiring high dimensional data from user devices: Aragorn gathers user images at the remote Aragorn server in the transient workflow in order to retrain the sanitizer . While we can expect users to trust Aragorn server since it is maintained by the OS vendors, a proportion of users can choose to opt out of sharing their images. As mobile hardware continues to improve [9], this limitation can be mitigated by on-device training of the sanitizer in a federated manner.

Supporting minority apps: Aragorn can support objects that are represented in public datasets as well as more nuanced objects that are requested by apps, since we gather training data from the users of the app. Aragorn , however, cannot support arbitrary objects requested by minority apps with limited user bases. We can mitigate such cases with a salient object detector that is better suited to work with images produced by mobile cameras (as opposed to a generic model like U^2-Net). We can also use such a model to protect user privacy from the app in the transient workflow.

9 CONCLUSION

While mobile cameras allow app developers to implement rich functionalities, their unfettered access also poses a serious threat to user privacy. Thus, it is important to limit camera access of mobile apps. To address this problem, we presented Aragorn , a new privacy-enhancing system that uses deep learning, knowledge distillation,

[5]We also attempted to threshold based on the area covered by the matched regions but observed higher accuracies for WaveOff when thresholding using the number of matches.

and crowdsourcing to automatically and extensibly implement fine grained object-level permissions for mobile camera apps. Aragorn protects user privacy by limiting access of camera apps to sanitized camera frames that contain only user-designated objects. Our evaluation showed that Aragorn can accurately sanitize camera frames without degrading user experience or app functionality.

REFERENCES

[1] 1Password. Use 1Password as an authenticator for sites with 2FA. https://support.1password.com/one-time-passwords/#save-your-qr-code.

[2] 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*, MobiSys '16, page 235–248, New York, NY, USA, 2016. Association for Computing Machinery.

[3] Adobe. Adobe Mobile Scanner. https://acrobat.adobe.com/us/en/acrobat/mobile/scanner-app.html.

[4] Aipoly Vision. Vision AI for the Blind and Visually Impaired. https://www.aipoly.com/.

[5] Apple. App Tracking Transparency. https://developer.apple.com/documentation/apptrackingtransparency.

[6] Apple. User Privacy and Data Use. https://developer.apple.com/app-store/user-privacy-and-data-use/.

[7] Apple Inc. iCloud photos. https://www.icloud.com/photos.

[8] Apple Inc. Use Face ID on your iPhone or iPad Pro. https://support.apple.com/en-us/HT208109.

[9] AppleInsider. In iOS 13, Apple applies Machine Learning to find cats and dogs. https://appleinsider.com/articles/19/06/26/in-ios-13-apple-applies-machine-learning-to-find-cats-and-dogs.

[10] Zhongjie Ba, Sixu Piao, Xinwen Fu, Dimitrios Koutsonikolas, Aziz Mohaisen, and Kui Ren. Abc: Enabling smartphone authentication with built-in camera. *25th Annual Network and Distributed System Security Symposium, NDSS 2018*.

[11] Be My Eyes. Bring sight to blind and low vision people. https://www.bemyeyes.com/.

[12] CamScanner. CamScanner for high work and learning efficiency. https://camscanner.com/.

[13] Center For Biometric and Security Research. CASIA-WebFace database. http://www.cbsr.ia.ac.cn/english/casia-webFace/casia-webfAce_AgreEmeNtS.pdf.

[14] Davide Cozzolino and Luisa Verdoliva. Noiseprint: a cnn-based camera model fingerprint. *CoRR*, abs/1808.08396, 2018.

[15] Z. Din, H. Venugopalan, H. Lin, A. Wushensky, S. Liu, and S. T. King. Doing good by fighting fraud: Ethical anti-fraud systems for mobile payments. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, pages 1728–1745, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[16] Zainul Abi Din, Hari Venugopalan, Jaime Park, Andy Li, Weisu Yin, HaoHui Mai, Yong Jae Lee, Steven Liu, and Samuel T. King. Boxer: Preventing fraud by scanning credit cards. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1571–1588. USENIX Association, August 2020.

[17] Sandeep D'Souza, Victor Bahl, Lixiang Ao, and Landon P. Cox. Amadeus: Scalable, privacy-preserving live video analytics. *CoRR*, abs/2011.05163, 2020.

[18] Facebook Inc. Instagram. https://www.instagram.com/.

[19] farmaker47. Portrait creator. https://github.com/farmaker47/Portrait_creator.

[20] Google. Advertising ID. https://support.google.com/googleplay/android-developer/answer/6048248.

[21] Google. The home for your memories. https://www.google.com/photos/about/.

[22] Google. The Privacy Sandbox. https://privacysandbox.com/intl/en_us/.

[23] Google. Unlock your Pixel phone with your face. https://support.google.com/pixelphone/answer/9517039.

[24] R. Hasan, D. Crandall, M. Fritz, and A. Kapadia. Automatically detecting bystanders in photos to reduce privacy risks. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 318–335, 2020.

[25] Gary B. Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, October 2007.

[26] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks. *CoRR*, abs/1611.07004, 2016.

[27] Suman Jana, Arvind Narayanan, and Vitaly Shmatikov. A scanner darkly: Protecting user privacy from perceptual appli. In *2013 IEEE Symposium on Security and Privacy*, pages 349–363, 2013.

[28] Orest Kupyn, Volodymyr Budzan, Mykola Mykhailych, Dmytro Mishkin, and Jiri Matas. Deblurgan: Blind motion deblurring using conditional adversarial networks. *CoRR*, abs/1711.07064, 2017.

[29] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper R. R. Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Tom Duerig, and Vittorio Ferrari. The open images dataset V4: unified image classification, object detection, and visual relationship detection at scale. *CoRR*, abs/1811.00982, 2018.

[30] Tomer Laor, Naif Mehanna, Antonin Durey, Vitaly Dyadyuk, Pierre Laperdrix, Clémentine Maurice, Yossi Oren, Romain Rouvoy, Walter Rudametkin, and Yuval Yarom. DRAWNAPART: A device identification technique based on remote GPU fingerprinting. *CoRR*, abs/2201.09956, 2022.

[31] Stefan Leutenegger, Margarita Chli, and Roland Y. Siegwart. Brisk: Binary robust invariant scalable keypoints. In *2011 International Conference on Computer Vision*, pages 2548–2555, 2011.

[32] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

[33] J. Lukas, J. Fridrich, and M. Goljan. Digital camera identification from sensor pattern noise. *IEEE Transactions on Information Forensics and Security*, 1(2):205–214, 2006.

[34] news.com.au. Woman goes viral after x-rated item spotted on the shelf during bbc wales report, 2021.

[35] Noopur Shreyas. Select. Scan. Share. Instant Contact Sharing App. https://ohhi.me/.

[36] Oberlo. 10 best AR apps for iOS and Android 2021. https://www.oberlo.com/blog/augmented-reality-apps.

[37] Pramuditha Perera and Vishal M. Patel. Learning deep features for one-class classification. *IEEE Transactions on Image Processing*, 28(11):5450–5463, nov 2019.

[38] Xuebin Qin, Zichen Zhang, Chenyang Huang, Masood Dehghan, Osmar Zaiane, and Martin Jagersand. U2-net: Going deeper with nested u-structure for salient object detection. volume 106, page 107404, 2020.

[39] Ariadna Quattoni and Antonio Torralba. Recognizing indoor scenes.

[40] 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*, MobiSys '16, page 249–261, New York, NY, USA, 2016. Association for Computing Machinery.

[41] reddit.com/user/birdlawatty/. When you don't check your background before going on the local news., 2022.

[42] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.

[43] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. *CoRR*, abs/1503.03832, 2015.

[44] Jiayu Shu, Rui Zheng, and Pan Hui. *Cardea: Context-Aware Visual Privacy Protection for Photo Taking and Sharing*, page 304–315. Association for Computing Machinery, New York, NY, USA, 2018.

[45] SNAP Inc. Snapchat. https://www.snapchat.com/.

[46] 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*, SenSys '17, New York, NY, USA, 2017. Association for Computing Machinery.

[47] Statista. Number of VR/AR users in United States. https://www.statista.com/statistics/1017008/united-states-vr-ar-users/.

[48] Syaringan. MobileFaceNet-Android. https://github.com/syaringan357/Android-MobileFaceNet-MTCNN-FaceAntiSpoofing.

[49] TensorFlow. Deploy machine learning models on mobile and IoT devices. https://www.tensorflow.org/lite/.

[50] Erkam Uzun, Simon Chung, Irfan Essa, and Wenke Lee. rtcaptcha: A real-time captcha based liveness detection system. 02 2018.

[51] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. Fp-Scanner: The privacy implications of browser fingerprint inconsistencies. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 135–150, Baltimore, MD, August 2018. USENIX Association.

[52] G.K. Wallace. The jpeg still picture compression standard. *IEEE Transactions on Consumer Electronics*, 38(1):xviii–xxxiv, 1992.

[53] Emily Wenger, Josephine Passananti, Yuanshun Yao, Haitao Zheng, and Ben Y. Zhao. Backdoor attacks on facial recognition in the physical world. *CoRR*, abs/2006.14580, 2020.

[54] Felix X. Yu, Ankit Singh Rawat, Aditya Krishna Menon, and Sanjiv Kumar. Federated learning with only positive labels, 2020.

[55] Zoom Inc. Setting up 2FA. https://support.zoom.us/hc/en-us/articles/360038247071-Setting-up-and-using-two-factor-authentication-2FA-.

10 APPENDIX

10.1 Aragorn with federated learning

In case of federated learning, we would execute the training procedure described in Section 5.3.1 within the user's device to compute

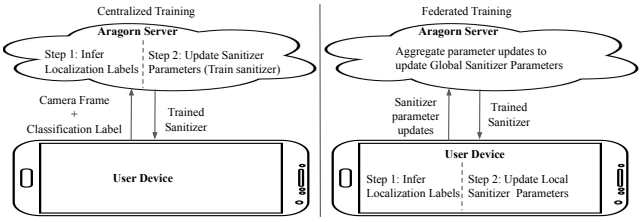

Figure 12: A centralized implementation (left) of the transient workflow to train the sanitizer would involve transmitting camera frames to Aragorn server which would then infer localization labels and train the sanitizer . In contrast, a federated implementation (right) would locally infer the localization labels and compute updates to the parameters (trains) of a local sanitizer within the user's device. Aragorn server would then aggregate updates from multiple user devices to train a global sanitizer .

parameter updates to a local sanitizer. Then, we would transmit these updates to Aragorn server which would aggregate updates from multiple devices to update the global sanitizer . We provide an overview of the variants in Figure. Upon training, user devices would be updated with the global sanitizer to operate in the steady-state workflow.

Training a multiclass sanitizer with federated learning In practice, using a multiclass object detector for the sanitizer that has been trained to recognize multiple objects would generalize better than multiple single class object detectors as independent sanitizers , each of which have been trained to recognize a different object [37], particularly when dealing with similar looking objects. For example, a sanitizer that has only been trained on credit card images may mistakenly recognize a driver's license as a credit card. With centralized learning, training a multiclass sanitizer while extending support to a new object would be trivial with access to training images (including those gathered previously for other objects) at Aragorn server. This would allow retraining the model from scratch with the appropriate number of objects. However, we cannot follow the same approach with federated learning since we would not have access to training images. Previously trained models cannot be reused as is, since their output layers would have fewer dimensions than the required number of dimensions for extension. We discuss the following methods to train the sanitizer in a federated manner. First, we could require all users, including those running apps in the steady-state workflow to contribute to training by computing parameter updates on their respective objects of interest. This way, Aragorn server will obtain model parameter updates for all objects, which it can aggregate to update the global model. This approach would be the federated equivalent of retraining the sanitizer from scratch. However, this approach would impose additional load on all user devices (in terms of running additional computations), including those devices that are not running apps that seek extension to recognize a new object. One way to reduce the load would be to fine-tune a previously trained sanitizer, where we replace the last layer with random weights of appropriate dimensions. Fine-tuning (updating the parameters of only the last layer) this model would require fewer updates for previous objects, and can be obtained from a smaller set of users.

Alternatively, we could use multiple single class sanitizers in case of federated learning. This would not increase the load on all user devices, but could lead to less accurate predictions from the sanitizer, especially when objects similar to the intended object of interest are present in the background. Despite this limitation, this approach could still protect user privacy in most cases since similar looking objects to the intended object of interest may not be present in the user's environment. In our user study in context of a credit card scanning application (7.1.1), none of the scan videos from 19 different users contained objects looking similar to credit cards in the background.

Lastly, adapting federated learning to scenarios where we only have access to data from a particular class is an active area of research. Aragorn could employ appropriate methods from that line of research such as Federated Averaging with Spreadout (FedAwS) [54].

Implementation The biggest challenge with Aragorn 's federated implementation is running ML inference on a salient object detector within the user's device. We run a public TensorFlow Lite version of U2-Net [19] as the on-device salient object detector on a Google Pixel 2 phone. The model occupies 43 MB on disk and on average takes 44s for inference on a single image on the CPU. The high latency contributed by this operation necessitates running the federated variant in the background (regardless of the latency of computing gradient updates to the local sanitizer) once the user has completed their interaction with the app to preserve user experience.

10.2 Preventing apps from interfering with Aragorn 's feedback to users

10.2.1 Steady-state workflow. In the steady-state workflow, once users have completed their interaction with a camera app, Aragorn annotates the frames produced during the interaction with the sanitizer 's predictions and renders them onto the phone's display for user feedback as part of the reactive defense 5.3.3. We do not show these annotations while the user interacts with the app to prevent the app from tampering with the annotations in an attempt to interfere with the feedback. While we could have alternatively prevented apps from writing to the display while using the camera, doing so would also restrict benign apps from implementing custom interfaces to better interact with users. In this section, we first discuss different ways in which an app could have run interference if we had provided feedback while the app was still in use. We then discuss the impact of these interferences on Aragorn 's intended execution to justify our decision to show feedback after app execution is complete.

When the sanitizer 's predictions accurately localize to the object of interest: In this case, a malicious app could tamper with Aragorn 's annotations such that the predictions look incorrect to users. The app's incentive to do so would be to force Aragorn to restart the training for that object, thereby allowing the app to once again access complete camera frames in the transient workflow. However, if the app attempts to do so repeatedly, Aragorn would begin its manual inspection and detect the app's interference leading to the app being blocklisted by Aragorn . Thus, in cases

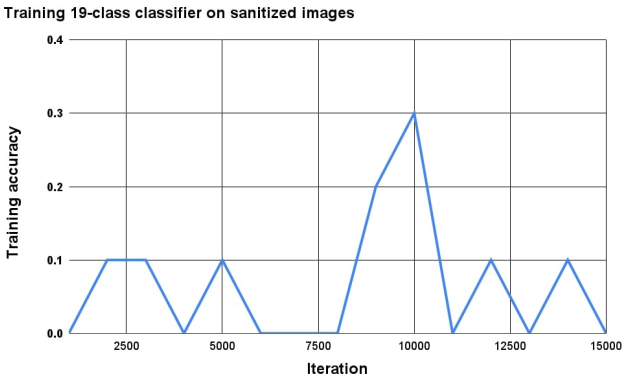

Figure 13: Training the 19-class classifier on sanitized frames from the sanitizer does not even lead to any improvement in training accuracy. This provides empirical evidence of a lack of fingerprintable information in the sanitized frames.

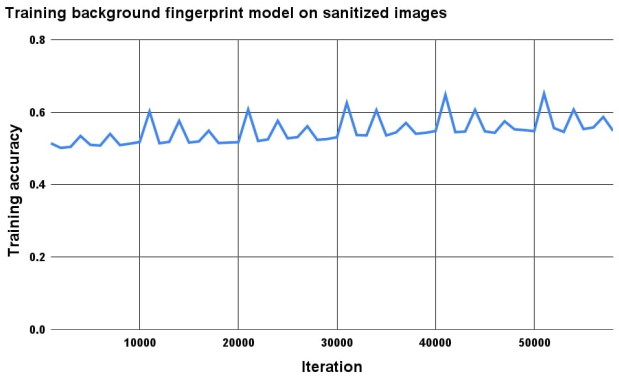

Figure 14: Training the background fingerprint model on sanitized frames from the sanitizer does not even lead to any improvement in training accuracy. This provides empirical evidence of a lack of fingerprintable information in the sanitized frames.

where the sanitizer 's prediction accurately localizes to the object of interest, providing feedback to users while the app is in use would only help the app invade user privacy initially, but not in the long run. However, our design of displaying these annotations after the app has completed its execution completely protects user privacy when the predictions accurately localize to the object of interest.

When the sanitizer 's predictions do not capture the object of interest: In this case, if a malicious app could tamper with Aragorn 's annotations such that they sufficiently localize to the object of interest, users would not be able to provide feedback on the mistake, and Aragorn would be unable to rectify these mistakes. However, we argue that it would be impossible for the app to do so, even if we were to provide feedback while the app is in execution. Since the app can only see the regions predicted by the sanitizer , which in this case, do not contain the object of interest, the app would have to accurately guess the location of the object in these frames in order to trick users to sending false feedback.

When the sanitizer 's predictions do not sufficiently localize to the object of interest: In this case, the sanitizer 's predictions capture the object of interest, but also capture other parts of the user's environment. In this case, a malicious app could alter the annotations to have them sufficiently localize to the object, thereby fooling users into providing false feedback to Aragorn . This use case cannot be handled if Aragorn were to show its annotations to the user while the app is still in execution, but is completely evaded by our design choice of providing feedback after the app has completed execution.

10.2.2 Transient workflow. There is no feedback to communicate with the user in the transient workflow since this workflow does no use the sanitizer . However, allowing apps to write to this display allows them to potentially trick users into scanning incorrect objects, leading to providing poisoned data to train the sanitizer . We still allow the app to write to the display in this workflow to help benign apps and handle such attempts by malicious apps to poison the sanitizer using our crowd-soruced defenses 5.3.3.

10.3 Are fingerprints present in the sanitized frames produced by Aragorn ?

We seek to understand if there is information contained in the sanitized images produced by Aragorn that can be used to fingerprint users.

We answer this question by training a 19-class classifier and a background fingerprint model on the sanitized frames produced by Aragorn on frames from our user study 7.1.1. However, we were neither able to train a background fingerprint model nor a classifier to fingerprint the sanitized images. Both models are unable to even reliably make predictions on their respective training sets, with the background fingerprint model attaining no more than 65% accuracy on its training set, and the classifier attaining no more than 40% on its training set. This inability of the classifier (a simpler model to train than the background fingerprint model) to even learn the training data provides strong empirical evidence of lesser information being present in the sanitized images that can be used for fingerprinting.

Figure 13 and Figure 14 show the variation in training accuracy of the 19-class classifier and the background fingerprint model respectively on the sanitized frames with training iterations.

10.4 Security Analysis

In this section, we list ways in which apps could attempt to undermine Aragorn to invade user privacy and how Aragorn is robust to such attacks.

Apps requesting support for redundant or nonsensical objects Apps could request Aragorn to extend support to an object that is already supported in an attempt to run in the unprotected transient workflow. For example, say Aragorn supports "Credit Card" as the object of interest. A malicious card scanning app that is aware of this, could purposely request support for "Payment Card". Firstly, this would lead to the app operating in the transient workflow where it can access complete camera frames from the

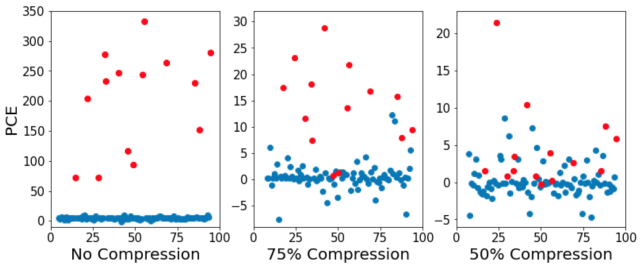

Figure 15: PCE ratios for 100 candidate images at three different compression amounts; red values are ratios from the same device, blue values are different devices. With Aragorn , we use 50% compression to obscure PRNU fingerprints.

app. Second, it would add a redundant object to the list of objects supported by Aragorn that could potentially affect its ability to accurately localize to the object. Similarly, a malicious app could also request support for some nonsensical object defined through some gibberish. Aragorn is robust against such attacks since requests for to extend the sanitizer to new objects go through manual review for approval. Requests for redundant or nonsensical objects would thus be caught and discarded in the review. The manual review also thwarts cases where the same app repeatedly requests support for different objects to continually operate in the transient workflow.

Apps forcing users to grant access to complete camera frames Aragorn also supports cases where users there is no object of interest and users have to show complete camera frames (All Objects option in Figure 4). Similar to some websites forcing users to disable ad blockers to access them, malicious camera apps could force users to grant access to complete camera frames. However, users can instead choose honest variants of such apps that provide the same utility which do not forcing users to do so.

10.5 Scaling to a large number of objects

In our experience, we observe that the classification and localization accuracies of object detectors such as YOLOv3 (which we use for the sanitizer) decrease when they support more than 15 objects. While this could raise concerns over how Aragorn can scale to support a large number of objects, such concerns can be easily addressed by employing multiple sanitizers , each of which supports a different set of objects. Then, based on the established object of interest, we can load the appropriate sanitizer to interface with the camera to protect user privacy.

We also ran a performance evaluation of a sanitizer supporting 15 objects on a Google Pixel 2 phone. This sanitizer roughly consumed the same amount of CPU and memory as the sanitizer supporting 3 objects (mentioned in Section §6), while adding less than 1ms to the execution time.

10.6 Supporting requests from users to support new objects

Before Aragorn 's initial deployment (prior to any invocations to the transient workflow), we train the sanitizer to detect common objects from public datasets to cover most objects that users could

Figure 16: Results of attempting to sharpen a blurred credit card image. The leftmost image is a blurred credit card image, followed by the result of sharpening the image using a sharpening filter, sharpening the image using DeblurGAN [28] and sharpening using Pix2pix GAN [26]. These results show the difficulty malicious users would face in attempting to recover sensitive information from blurred images shown to them as part of Aragorn 's preemptive defense.

seek to scan that are not associated with any particular app. In case multiple users request support for a previously unrecognized object, Aragorn can support them as long as the object can either be associated with at least one camera app, or is present in a public dataset. Currently, Aragorn cannot support user requests to support arbitrary objects that do not map to either of these two options.

10.7 Users attempting to sharpen blurred images in the preemptive defense

As discussed in §5.3.3, the preemptive defense validates images gathered from users by showing them to other users. In order to protect the privacy of the gathered images, we blur these images before showing them to other users. Malicious users could attempt to sharpen the images shown to them in order to recover and leak any private information contained in them. We ran experiments to sharpen the blurred images using image sharpening filters as well as two different deep learning based conditional generative adversarial networks (GANs) [26, 28], but were unable to recover any sharpen the images to recover any sensitive information from them. We show an example of the sharpened images produced by these methods in Figure 16.

10.8 How Aragorn 's quality reduction affects the performance of PRNU based camera fingerprinting attacks

Aragorn uses JPEG compression to obscure inherent high-frequency camera fingerprints also known as photo-response nonuniformity (PRNU). This ensures that apps cannot use them to fingerprint/track users. In this section, we evaluate the effectiveness of JPEG compression in obscuring PRNU. We use a public implementation of the ABC protocol[10] that calculates the Peak-to-Correlation Energy ratio (PCE) between two images to determine if the same PRNU can be detected from both images. A higher PCE implies that the two images are likely to have the same PRNU, meaning that they were captured on the same device.

Concretely, we used 6 devices (Samsung Galaxy S7, Samsung Galaxy Tab S7, Google Pixel 2, Google Pixel 3A, Google Pixel 4 and Redmi Note 7) and picked 20 reference images from them. We then pick 100 candidate images randomly chosen from these devices for comparison. A fingerprint can be successfully extracted if it is

feasible to draw a clear bound between the PCE values of the devices that are the same as the reference device, and those that are different. As seen in Figure 15, that there is a clear separation between the PCE values when both the reference images and the candidate images have not not gone through JPEG compression. However, for 75% and 50% JPEG compression, there is significantly more overlap between the PCE values for the candidate images, indicating that JPEG compression is an effective technique at obscuring PRNU.

We repeated the experiment by considering JPEG compressed reference images and observed similar results.