EnTrust: Regulating Sensor Access by Cooperating Programs via Delegation Graphs

Giuseppe Petracca, *Pennsylvania State University, US;* Yuqiong Sun, *Symantec Research Labs, US;* Ahmad-Atamli Reineh, *Alan Turing Institute, UK;* Patrick McDaniel, *Pennsylvania State University, US;* Jens Grossklags, *Technical University of Munich, DE;* Trent Jaeger, *Pennsylvania State University, US*

https://www.usenix.org/conference/usenixsecurity19/presentation/petracca

This paper is included in the Proceedings of the 28th USENIX Security Symposium.

August 14–16, 2019 • Santa Clara, CA, USA

978-1-939133-06-9

Open access to the Proceedings of the 28th USENIX Security Symposium is sponsored by USENIX.

EnTrust: Regulating Sensor Access by Cooperating Programs via Delegation Graphs

Giuseppe Petracca
Penn State University, US
gxp18@cse.psu.edu

Yuqiong Sun
Symantec Research Labs, US
yuqiong_sun@symantec.com

Ahmad-Atamli Reineh
Alan Turing Institute, UK
atamli@turing.ac.uk

Patrick McDaniel
Penn State University, US
mcdaniel@cse.psu.edu

Jens Grossklags
Technical University of Munich, DE
jens.grossklags@in.tum.de

Trent Jaeger
Penn State University, US
tjaeger@cse.psu.edu

Abstract

Modern operating systems support a cooperating program abstraction that, instead of placing all functionality into a single program, allows diverse programs to cooperate to complete tasks requested by users. However, untrusted programs may exploit such interactions to spy on users through device sensors by causing privileged system services to misuse their permissions, or to forward user requests to malicious programs inadvertently. Researchers have previously explored methods to restrict access to device sensors based on the state of the user interface that elicited the user input or based on the set of cooperating programs, but the former approach does not consider cooperating programs and the latter approach has been found to be too restrictive for many cases. In this paper, we propose EnTrust, an authorization system that tracks the processing of input events across programs for eliciting approvals from users for sensor operations. EnTrust constructs *delegation graphs* by linking input events to cooperation events among programs that lead to sensor operation requests, then uses such *delegation graphs* for eliciting authorization decisions from users. To demonstrate this approach, we implement the EnTrust authorization system for Android OS. In a laboratory study, we show that attacks can be prevented at a much higher rate (47-67% improvement) compared to the first-use approach. Our field study reveals that EnTrust only requires a user effort comparable to the first-use approach while incurring negligible performance (<1% slowdown) and memory overheads (5.5 KB per program).

1 Introduction

Modern operating systems, such as Android OS, Apple iOS, Windows Phone OS, and Chrome OS, support a programming abstraction that enables programs to cooperate to perform user commands via input event delegations. Indeed, an emergent property of modern operating systems is that system services are relatively simple, provide a specific functionality, and often rely on the cooperation with other programs to perform tasks. For instance, modern operating systems now ship with voice-controlled personal assistants that may enlist apps and other system services to fulfill user requests, reaching for a new horizon in human-computer interaction.

Unfortunately, system services are valuable targets for adversaries because they often have more permissions than normal apps. In particular, system services are automatically granted access to device sensors, such as the camera, microphone, and GPS. In one recent case reported by Gizmodo [1], a ride-sharing app took advantage of Apple iOS system services to track riders. In this incident, whenever users asked their voice assistant "Siri, I need a ride", the assistant enlisted the ride-sharing app to process the request, which then leveraged other system services to record the users' device screens, even while running in the background. Other online magazines have reported cases of real-world evidence that apps are maliciously colluding with one another to collect and share users' personal data [2, 3, 4].

Such attacks are caused by system services being tricked into using their permissions on behalf of malicious apps (confused deputy attacks [5, 6]), or malicious apps exploiting their own privileges to steal data, and a combination of the two. Researchers have previously shown that such system services are prone to exploits that leverage permissions only available to system services [7]. Likewise, prior work has demonstrated that system services inadvertently or purposely (for functionality reasons) depend on untrusted and possibly malicious apps to help them complete tasks [8].

Such attacks are especially hard to prevent due to two information asymmetries. System services are being exploited when performing tasks on behalf of users, where: (1) users do not know what processing will result from their requests and (2) services do not know what processing users intended when making the request. Current systems employ methods to ask users to authorize program access to sensors, but to reduce users' authorization effort they only ask on a program's first use of that permission. However, once authorized, a program can utilize that permission at will, enabling programs

to spy on users as described above. To prevent such attacks, researchers have explored methods that bind input events, including facets of the user interface used to elicit those inputs, to permissions to perform sensor operations [9, 10, 12]. Such methods ask users to authorize permissions for those events and reuse those permissions when the same event is performed to reduce the user burden. Recent research extends the collection of program execution context (e.g., data flows and/or GUI flows between windows) more comprehensively to elicit user authorizations for sensitive operations [16, 11]. However, none of these methods addresses the challenge where an input event is delivered to one program and then a sensor operation, in response to that event, is requested by another program in a series of inter-process communications, a common occurrence in modern operating systems supporting the cooperating program abstraction.

Researchers have also explored methods to prevent unauthorized access by regulating inter-process communications (IPCs) and by reducing the permissions of programs that perform operations on behalf of other programs. First, prior work developed methods for blocking IPC communications that violate policies specified by app developers [8, 18, 19, 21, 22]. However, such methods may prevent programs from cooperating as expected. Decentralized information flow control [23, 24] methods overcome this problem by allowing programs with the authority to make security decisions and make IPCs that may otherwise be blocked. Second, DIFC methods, like capability-based systems in general [34], enable reduction of a program's permissions (i.e., callee) when performing operations on behalf of other programs (i.e., callers). Initial proposals for reducing permissions simply intersected the parties' permissions [7], which however was too restrictive because parties would have their permissions pruned after the interaction with less privileged parties. DIFC methods, instead, provide more flexibility [20], albeit with the added complexity of requiring programs to make non-trivial security decisions. Our insight to simplify the problem is that while DIFC methods govern information flows comprehensively to prevent the leakage of sensitive data available to programs, users instead want to prevent programs from abusing sensor access to obtain sensitive data in the first place.

In addition, prior work has also investigated the use of machine learning classifiers to analyze the contextuality behind user decisions to grant access to sensors automatically [14, 15]. Unfortunately, the effectiveness of the learning depends on the accuracy of the user decisions while training the learner. Therefore, we firmly believe that additional effort is necessary in improving user decision making before the user decisions can be used to train a classifier.

In this work, we propose the EnTrust authorization system to prevent malicious programs from exploiting cooperating system services to obtain unauthorized access to device sensors. At a high-level, our insight is to combine techniques that regulate IPC communications of programs of different privilege levels with techniques that enable users to be aware of the permissions associated with an input event and decide whether to grant such permissions for the identified flow context. The former techniques identify how a task is "delegated" among cooperating programs to restrict the permissions of the delegatee.[1] The latter techniques expose more contextual information to a user, which may be useful to make effective authorization decisions.

However, combining these two research threads results in several challenges. First, we must be able to associate input events with their resulting sensor operations in other programs to authorize such operations relative to the input events and sequence of cooperating programs. Prior work does not track how processing resulting from input events is delegated across programs [9, 10, 11, 12], but failing to do so results in attack vectors exploitable by an adversary. In EnTrust, we construct *delegation graphs* that associate input events with their resulting sensor operations across IPCs to authorize operations in other programs.

Second, multiple, concurrent input events and IPCs may create ambiguity in tracking delegations across processes that must be resolved to ensure correct enforcement. Prior work either makes assumptions that are often too restrictive or require manual program annotations to express such security decisions. EnTrust leverages the insights that input events are relatively infrequent, processed much more quickly than users can generate distinct events, and are higher priority than other processing. It uses these insights to ensure that an unambiguous *delegation path* can be found connecting each input event and sensor operation, if one exists, with little impact on processing overhead.

Third, we must develop a method to determine the permissions to be associated with an input event for other programs that may perform sensor operations. Past methods, including machine learning techniques [14, 15], depend on user decision making to select the permissions associated with input events, but we wonder whether the information asymmetries arising from delegation of requests across programs impair user decision making. In EnTrust, we elicit authorization decisions from users by using delegation paths. We study the impact of using delegation paths on users' decision making for both primed and unprimed user groups. Historically, there has been a debate on whether users should be considered a weak link in security [56, 57]. We examine this argument in a specific context by investigating if users can make informed security decisions given informative, yet precise, contextual information.

We implement and evaluate a prototype of the EnTrust authorization system for Android OS. We find that EnTrust significantly reduces exploits from three

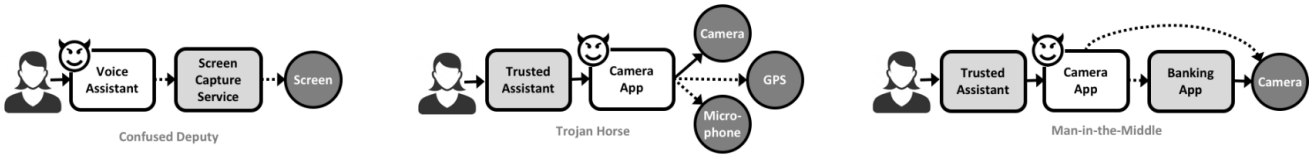

Figure 1: Possible attack vectors when diverse programs interact via input event delegations in a cooperating model. For consistency, we present the attack scenarios in terms of voice assistants receiving input events via voice commands; however, similar attack scenarios are possible for input events received by programs via Graphical User Interface (GUI) widgets rendered on the users' device screen.

canonical types of attack vectors possible in systems supporting cooperating programs, requires little additional user effort, and has low overhead in app performance and memory consumption. In a laboratory study involving 60 human subjects, EnTrust improves attack detection by 47-67% when compared to the first-use authorization approach. In a field study involving 9 human subjects, we found that - in the worst scenarios seen - programs required no more than four additional manual authorizations from users, compared to the less secure first-use authorization approach; which is far below the threshold that is considered at risk for user annoyance and habituation [33]. Lastly, we measured the overhead imposed by EnTrust via benchmarks and found that programs operate effectively under EnTrust, while incurring a negligible performance overhead (<1% slowdown) and a memory footprint of only 5.5 kilobytes, on average, per program.

In summary, we make the following contributions:

- We propose a method for authorizing sensor operations in response to input events performed by cooperating programs by building unambiguous *delegation graphs*. We track IPCs that delegate task processing to other programs without requiring system service or app code modifications.
- We propose EnTrust, an authorization system that generates delegation paths to enable users to authorize sensor operations, resulting from input events, and reuse such authorizations for repeated requests.
- We implement the EnTrust prototype and test its effectiveness with a laboratory study, the users' authorization effort with a field study, and performance and memory overhead via benchmarks.

2 Problem Statement

In current operating systems, users interact with programs that initiate actions targeting sensors, but users do not have control over *which programs* are going to service their requests, or *how such programs access sensors* while servicing such requests. Unfortunately, three well-studied attack vectors become critical in operating systems supporting a cooperating program abstraction.

Confused Deputy — First, a malicious program may leverage an input event as an opportunity to confuse a more privileged program into performing a sensitive operation. For example, a malicious voice assistant may invoke the screen capture service at each voice command (left side of Figure 1). The malicious voice assistant may therefore succeed in tricking the screen capture service into capturing and inadvertently leaking sensitive information (e.g., a credit card number written down in a note). In this scenario, the user only sees the new note created by the notes app, whereas the screen capture goes unnoticed. Currently, there are over 250 voice assistants available to the public on Google Play with over 1 million installs, *many by little known or unknown developers*.

Trojan Horse — Second, a program trusted by the user may delegate the processing of an input event to an untrusted program able to perform the requested task. For example, a trusted voice assistant may activate a camera app to serve the user request to take a selfie (middle of Figure 1). However, the camera app may be a Trojan horse app that takes a picture, but also records a short audio via the microphone, and the user location via GPS (e.g., a spy app[2] installed by a jealous boyfriend stalking on his girlfriend). Researchers reported over 3,500 apps available on Google Play Store that may be used as spyware apps for Intimate Partner Violence (IPV) [25]. In this scenario, the user only sees the picture being taken by the camera app, whereas the voice and location recordings go unnoticed, since a camera app is likely to be granted such permission. Also, the ride-sharing attack in the introduction is another example of this attack. Such attacks are possible because even trusted system services may inadvertently leverage malicious apps and/or rely on unknown apps by using implicit intents. An implicit intent enables any program registered to receive such intents to respond to IPCs when such intents are invoked. Researchers have reported several ways how programs can steal or spoof intents intended for other programs [26, 27, 28]. We performed an analysis of system services and applications distributed via the Android Open Source Project (AOSP), and found that 10 system programs out of a total of 69 (14%) use implicit intents.

Man-In-The-Middle — Third, a request generated by a program trusted by the user may be intercepted by a malicious program, which can behave as a man-in-the-middle in serving the input event in the attempt to obtain access to unauthorized data (right side of Figure 1). For example, a legitimate banking app may adopt the voice interaction intent mechanism to allow

customers to direct deposit a check via voice assistant with a simple voice command (e.g., "deposit check").[3] A malicious program may exploit such a service by registering itself with a voice assistant as able to service a similar voice interaction, such as "deposit *bank* check." Therefore, whenever the user instantiates the "deposit *bank* check" voice command, although the user expects the legitimate banking app to be activated, the malicious app is activated instead. The malicious app opens the camera, captures a frame with the check, and sends a spoofed intent to launch the legitimate banking app, all while running in the background. In this scenario, the user only sees the trusted banking app opening a camera preview to take a picture of the check. This is a realistic threat. We performed an analysis of 1,000 apps (among the top 2,000 most downloaded apps on Google Play Store) and found that 227 apps (23%) export at least a public service or a voice interaction intent. Apps were selected from the Google Play Store among those apps declaring at least one permission to access a sensitive sensor (e.g., camera, microphone, or GPS).

Security Guarantee. To mitigate such attack vectors, an authorization mechanism must provide the following guarantee, *for any sensor operation to be authorized, that operation must be: (1) initiated by an input event; (2) authorized for the input event to trigger the sensor operation; and (3) authorized for the sequence of programs receiving the input event directly or indirectly through IPCs leading to the program performing the sensor operation.* Such a guarantee ensures that any sensor operation must be initiated by an input event, the input event must imply authorization of the resultant sensor operation by the requesting program, and all programs associated with communicating the request for the sensor operation must be authorized to enable the sensitive data to be collected by the requesting program. To achieve the security guarantee above, we require a mechanism that accurately tracks the delegations leading from input events to resulting sensor operations, as well as a mechanism to authorize sensor operations to collect sensitive data given input events and delegations.

Regarding tracking delegations, a problem is that determining whether an IPC derives from an input event or receipt of a prior IPC depends on the data flows produced by the program implementations in general. Solving this problem requires data flow tracking, such as performed by taint tracking. However, taint tracking has downsides that we aim to avoid. Static taint tracking can be hard to use and be imprecise [30] and dynamic taint tracking has non-trivial overhead [29]. Instead, we aim to explore solutions that ensure all sensor operations resulting from an input event are detected (i.e., we overapproximate flows) without heavyweight analysis or program modifications.

Authorizing sensor operations to collect sensitive data, given an input event and one or more delegations, depends on determining the parties involved in the delegation as well as the user's intent when generating the event. Methods that restrict the permissions of an operation to the intersection of permissions granted to the parties involved [7], have been found to be too restrictive in practice. Decentralized information flow control [23, 24] (DIFC) prevents information leakage while allowing some privileged programs to make flexible security decisions to determine when to permit communications that are normally unauthorized, which has been applied to mobile systems [20, 13]. However, these information flow control techniques focus on preventing the leakage of sensitive information available to programs, whereas the main goal here is to prevent programs from obtaining access to sensitive information in the first place by abusing sensor access. To address this problem more directly, researchers have explored techniques that enable users to express the intent of their input events to authorize sensor operations, binding this intent to the context in which the input event was elicited, such as the graphical user interface (GUI) context [9, 10, 11]. In IoT environments, researchers have similarly explored gathering program execution context (e.g., data flows) to enable users to authorize IoT operations more accurately [16]. However, none of these techniques account for delegations of tasks to other processes. We aim to explore methods for eliciting user authorizations for sensor operations using contextual information related to the tracking of input events and subsequent delegations.

Further, researchers have explored learning methods to predict permissions for sensor operation based on prior user decisions [14, 15]. However, accurate user decision making is vital for improving the accuracy of these learning techniques.

3 Security Model

Trust Model – We assume that the system (e.g., Linux kernel, operating system, system services, and device drivers) is booted securely, runs approved code from device vendors, and is free of malice; user-level programs (e.g., applications) are isolated from each other via the sandboxing mechanism using separated processes [35, 36]; and, by default, user-level programs have no direct access to sensors due to the use of a Mandatory Access Control (MAC) policy [37, 38] enforced from boot time. We assume the use of *trusted paths*, protected by MAC, allowing users to receive unforgeable communications from the system, and providing unforgeable input events to the system. Our assumptions are in line with existing research on trusted paths and trusted user interfaces for browsers [39], X window systems [40, 41], and mobile operating systems [42].

Threat Model – We assume that users may install programs from unknown sources that may be malicious, then grant such programs access to sensors at first use. Despite the default isolation via sandboxing, programs may communicate via IPC mechanisms (i.e., intents or

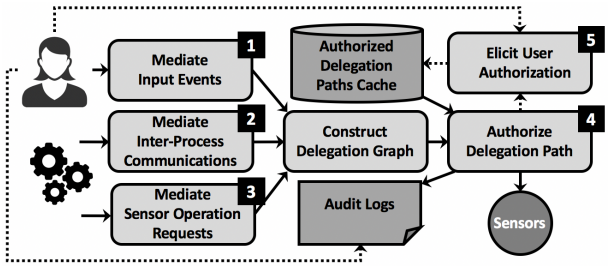

Figure 2: EnTrust Authorization Method – Input events, handoff events, and sensor operations are linked via delegation graphs to compute unambiguous delegation paths for user authorization of sensor operations.

broadcast messages). Thus, user-level programs (e.g., apps) may leverage such communication to exploit the attack vectors described in Section 2. Our objective is to provide a mechanism that helps users control how cooperating programs access sensors. *How programs manage and share the data collected from sensors is outside the scope of our research.* Researchers have already examined solutions to prevent data leakage based on taint analysis [29, 30, 31, 18] and Decentralized Information Flow Control (DIFC) [20, 23, 24, 32].

4 EnTrust Authorization Design

In this section, we describe our proposed framework, EnTrust, designed to restrict when programs may perform sensor operations by requiring each sensor operation to be unambiguously associated with an input event, even if the sensor operation is performed by a program different from the one receiving the input event. Figure 2 provides an overview of the EnTrust authorization system, which consists of five steps. In the first three steps, EnTrust mediates and records input events, inter-process communication events (handoff events), and sensor operation requests, respectively, to construct a *delegation graph* connecting input events to their handoff events and sensor operation requests. In the fourth step, EnTrust uses the constructed delegation graph to compute an unambiguous *delegation path* to a sensor operation request from its originating input event. Unless the authorization cache contains a user authorization for the constructed delegation path already, the fifth step elicits an authorization from the user for the delegation path, and caches the authorization for later use for the same delegation path. Optionally, users can review their prior decisions and correct them via an audit mechanism that logs past authorized and denied delegation graphs.

4.1 Building Delegation Graphs

The first challenge is to link input events to all the sensor operations that result from cooperating programs processing those events and then construct *delegation graphs* rooted at such input events.

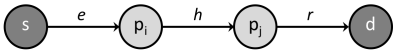

Figure 3: Delegation graphs connect input events with operation requests for sensors via handoff events.

First, for each input event received via a sensor s for a program p_i, EnTrust creates an *input event* tuple $e = (c, s, p_i, t_0)$, where c is the user interface context captured at the moment the input event occurred; s is the sensor through which the event was generated; p_i is the program displaying its graphical user interface on the screen and receiving the input event e; and t_0 is the time of the input event (step 1 in Figure 2). Note: EnTrust is designed to mediate both input events coming from input sensors (e.g., touch events on widgets rendered on the screen) as well as voice commands captured via the microphone. Voice commands are translated into text by the Google Cloud Speech-to-Text service.

Second, after receiving the input event, program p_i may hand off the event to another program p_j. EnTrust mediates handoff events by intercepting spawned intents and messages exchanged between programs [43] and models them as tuples $h = (p_i, p_j, t_i)$, where p_i is the program delegating the input event, p_j is the program receiving the event, and t_i is the time the event delegation occurred (step 2 in Figure 2).

Third, when the program p_j generates a request r for an operation o targeting a sensor d, EnTrust models the request as a tuple $r = (p_j, o, d, t_j)$, where p_j is the program requesting the sensor operation, o is the type of sensor operation requested, d is the destination sensor, and t_j is the time the sensor operation request occurred (step 3 in Figure 2).

Lastly, EnTrust connects sensor operation requests to input events via handoff events by constructing a delegation graph to regulate such operations, as shown in Figure 3. A *delegation graph* is a graph, $G = (V, E)$, where the edges $(u, v) \in E$ represent the flow of input events to programs and sensors, and the vertices, $v \in V$, represent the affected programs and sensors. Figure 3 shows a simple flow, whereby a source sensor s receives an input event e that is delivered to a program p_i, which performs a handoff event h to a program p_j that performs an operation request r for a destination sensor d. Thus, there are three types of edges: input event to program (user input delivery), program to program (handoff), and program to sensor operation request (request delivery).

Upon mediation of a sensor request r, EnTrust computes the associated *delegation path* by tracing backwards from the sensor request r to the original input event e. Hence, the operation request $r = (p_j, o, d, t_j)$ above causes a delegation path: $(c, s, p_i, t_0) \rightarrow (p_i, p_j, t_i) \rightarrow (p_j, o, d, t_j)$ to be reported in step 4 in Figure 2. Delegation paths are then presented to the user for authorization (see Section 4.3). The identified delegation path is shown to the user using natural language, in a

Figure 4: Two scenarios that create ambiguity. Multiple input events or handoff events delivered to the same program.

manner similar to first-use authorizations. We assess how effectively users utilize delegation paths to produce authorizations in a laboratory study in Section 6.2.

4.2 Computing Delegation Paths

Critical to computing delegation paths is the ability for EnTrust to find an unambiguous reverse path from the sensor operation request r back to an input event e. In particular, a delegation path is said to be *unambiguous* if and only if, given an operation request r by a program p_j for a sensor d, either there was a single input event e for program p_j that preceded the request r, or there was a single path $p_i \rightarrow p_j$ in the delegation graph, where program p_i received a single input event e.

To ensure unambiguous delegation paths without program modification, we need to define the conditions under which operations that create ambiguities cannot occur. First, ambiguity occurs if the same program p_i receives multiple input events and then performs a handoff, as depicted by the left side of Figure 4. In this case, it is unclear which one of the input events resulted in the handoff. To prevent this ambiguous case, we leverage the insight that input events are relatively infrequent, processed much more quickly than users can generate them, and have a higher priority than other processing. We observe that the time between distinct input events is much larger than the time needed to produce the operation request corresponding to the first input event. If every input event results in an operation request before the user can even produce another distinct input event, then there will be only one input event (edge) e from a source sensor (node) s to program (node) p_i, which received such input event. Therefore, there will be no ambiguous input event for program p_i. Thus, we propose to set a time limit for each input event, such that the difference between the time t_0 at which an input event e is generated and the time t_j for any sensor operation request r – based on that input event – must be below that limit for the event to be processed. Note that, once an input event is authorized (Section 4.3), repeated input events (e.g., pressing down a button multiple times) are not delayed. Indeed, repeated input events are expected to generate the same delegation path. Should the programs produce a different delegation path - in the middle of a sequence of operations spawned in this manner - then EnTrust would require a new authorization for the new delegation path, as described in Section 4.3.

Second, ambiguity is also possible if the same program p_j receives multiple handoff events before performing a sensor operation request, as depicted by the right side of Figure 4. Note that, handoff events may not be related to input events (e.g., intents not derived from input events). In this case, it is unclear which handoff is associated with a subsequent sensor operation request. Ambiguity prevention for handoff events is more subtle, but builds on the approach used to prevent ambiguity for input events. Figure 5 shows the key challenge. Suppose a malicious program p_k tries to "steal" a user authorization for a program p_j to perform a sensor operation by submitting a handoff event that will be processed concurrently to the handoff event from another program p_i, which received an input event. Should a sensor operation request occur, EnTrust cannot determine whether the sensor operation request from p_j was generated in response to the event handoff h_1 or to the event handoff h_2. So EnTrust cannot determine the delegation path unambiguously to authorize the operation request. If EnTrust knows the mapping between actions associated to handoff events and whether they are linked to sensor operations, EnTrust can block a handoff from p_k that states an action that requires an input event. EnTrust knows this mapping for system services, by having visibility of all inter-procedural calls for programs part of the operating system; however, EnTrust may not know such mapping for third-party apps whose inter-procedural control flow is not mediated to favor backward compatibility with existing apps.

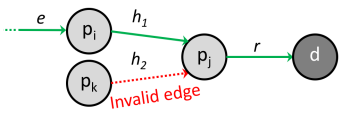

Figure 5: A program p_k attempts leveraging the input event received from program p_i to get program p_j to generate an operation request.

Thus, we extend the defense for input events to prevent ambiguity as follows: once the target program has begun processing a handoff associated with an input event, EnTrust delays the delivery of subsequent handoff events until this processing completes or until the assigned time limit ends. Conceptually, this approach is analogous to placing a readers-writers lock [44] over programs that may receive handoffs that result from input events. Note that, the use of time limits ensures no deadlock since it ensures preemption. To avoid starving input events (e.g., delaying them until the time limit), we prioritize delivery of handoffs that derive from input events ahead of other handoffs using a simple, two-level scheduling approach. We assess the impact of the proposed ambiguity prevention mechanisms on existing programs' functionality and performance in Section 7.

4.3 Authorizing Delegation Paths

For controlling when a sensor operation may be performed as the result of an input event, users are in the best position to judge the intent of their actions. This is

inline with prior work advocating that it is highly desirable to put the user in context when making permission granting decisions at runtime [16, 33, 17]. Therefore, users must be the parties to make the final authorization decisions. To achieve this objective, EnTrust elicits an explicit user authorization every time that a new delegation path is constructed (step 5 in Figure 2). Hence, to express a delegation path comprehensively, EnTrust builds an authorization request that specifies that delegation path to the user. Prior work presented users with information about the Graphical User Interface (GUI) used to elicit the input event, including GUI components [9, 10, 11], user interface workflows [13], and Application Programming Interface (API) calls made by applications [11, 16]. EnTrust, instead, presents the delegation path that led to the sensor operation, which includes the GUI context (c in the input event) and the handoffs and sensor operations. As a result, EnTrust ensures that all the programs receiving sensor data are clearly identified and reported in the authorization request presented to the user, along with the input event, handoff events, and the resulting sensor operation.

To reduce users' authorization effort, EnTrust caches authorized delegation paths for reuse. After storing an authorized delegation path, EnTrust proceeds in allowing the authorized sensor operation. For subsequent instances of the same input event that results in exactly the same delegation path, EnTrust omits step 5 and automatically authorizes the sensor operation by leveraging the cached authorization. Note that, EnTrust requires an explicit user's authorization *only* the first time a delegation path is constructed for a specific input event, similarly to the first-use permission approach. As long as the program receiving an input event does not change the way it processes that event (i.e., same handoffs and operation request), no further user authorization will be necessary. In Section 6.2, we show that such an approach does not prohibitively increase the number of access control decisions that users have to make, thus avoiding decision fatigue [45].

Further, EnTrust evicts cached authorizations in two scenarios. First, if a new delegation path is identified for an input event that already has a cached delegation path, then EnTrust evicts the cached authorization and requires a new user authorization for the newly constructed delegation path, before associating it to such an input event and caching it. Second, users can leverage an audit mechanism, similar to proposals in related work [11, 15], to review previous authorizations and correct them if needed. Denied authorizations are also logged for two reasons. First, they allow users to have a complete view of their past decisions; but more importantly, they allow EnTrust to prevent malicious programs from annoying users by generating authorization requests over a give threshold, for operations already denied by the user in the past. Also, users may set the lifetime of cached authorizations, after which they are evicted. We discuss utilizing such logs for revoking mistaken authorizations and denials in Section 8.

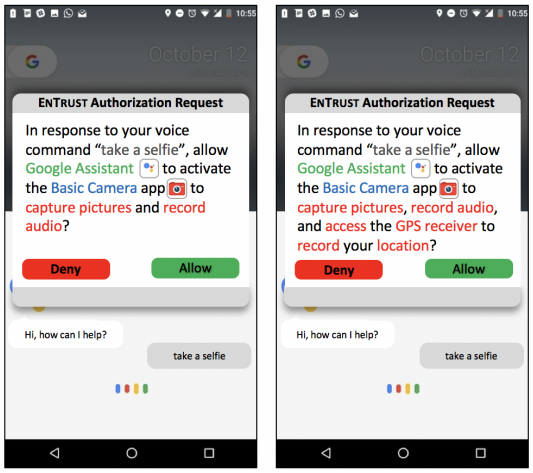

Figure 6: Authorization requests prompted to users by EnTrust upon *delegation paths* creation. Screenshots showing benign (left) and attack (right) scenarios by the Basic Camera app.

5 Implementation

We implemented a prototype of the EnTrust authorization system by modifying a recent release of the Android OS (Android-7.1.1_r3) available via the Android Open Source Project (AOSP).[4] The choice of implementing the EnTrust prototype for the Android OS was guided by its open-source nature and its wide adoption. EnTrust's footprint is 170 SLOC in C for the Linux kernel (bullhead 3.10), plus 380 SLOC in C, 830 SLOC in C++, and 770 SLOC in Java for components in the Android OS.

In this section, we provide the implementation details for event scheduling and authorization management. In Appendices C-F, we provide further implementation details regarding event authentication and mediation.

In Android, the Event Hub (part of the Input Manager server) reads raw input events from the input device driver files (`/dev/input/*`) and delivers them to the Input Reader. The Input Reader then formats the raw data and creates input event data that is delivered to the Input Dispatcher. The Input Dispatcher then consults the Window Manager to identify the target program based on the activity window currently displayed on the screen. Hence, we enhanced the Input Dispatcher to hold - for the duration of a time window - incoming input events for a target program should there be already a delivered input event for such a program that has not been processed, yet. For handoff events, instead, the Binder is the single point of mediation for inter-process communication (IPC) between isolated programs. It has the knowledge of all the pending messages exchanged as well as knowledge of the identity

of the two communicating parties. Hence, we also enhanced the Binder to hold - for the duration of a time window - incoming handoff events for a target program should the program be already involved in another communication with a third program.

EnTrust prompts users with *authorization messages* for explicit authorizations of delegation paths, as shown in Figure 6. Users are made *aware of all the programs cooperating in serving their requests* as well as of the *entire delegation path*. Also, users are prompted with programs' names and identity marks to ease their identification. EnTrust crosschecks developers' signatures and apps' identity (i.e., names and logos) by pulling information from the official Google Play Store to prevent identity spoofing. Also, EnTrust prevents programs from creating windows that overlap the *authorization messages* by leveraging the Android screen overlay protection mechanism. Finally, EnTrust prevents unauthorized modification of *authorization messages* by other programs by using isolated per-window processes forked from the Window Manager to implement a *Compartmented Mode Workstation* model [48].

6 EnTrust Evaluation

We investigated the following research questions:

▶ *To what degree is the* EnTrust *authorization assisting users in avoiding confused deputy, Trojan horse, and man-in-the-middle attacks?* We performed a *laboratory study* and found that EnTrust significantly increased (from 47-67% improvement) the ability of participants in avoiding attacks.

▶ *What is the decision overhead imposed by* EnTrust *on users due to explicit authorization of constructed delegation graphs?* We performed a *field study* and found that the number of decisions imposed on users by EnTrust remained confined - in worst case scenarios - to no more than 4 explicit authorizations per program.

▶ *Is* EnTrust *backward compatible with existing programs? How many operations from legitimate programs are incorrectly blocked by* EnTrust*?* We used a well-known compatibility test suite to evaluate the compatibility of EnTrust with 1,000 apps (selected among the most popular apps on Google Play Store) and found that EnTrust does not cause the failure of any program.

▶ *What is the performance overhead imposed by* EnTrust *for delegation graph construction and enforcement?* We used a well-known software exerciser to measure the performance overhead imposed by EnTrust. We found that EnTrust introduced a negligible overhead (order of milliseconds) unlikely noticeable to users.

6.1 Study Preliminaries

We designed our user studies following suggested practices for human subject studies in security to avoid common pitfalls in conducting and writing about security and privacy human subject research [49]. An Institutional Review Board (IRB) approval was obtained from our institution. The data collected did not contain Personally Identifiable Information (PII) and was securely stored and accessible only to authorized researchers. We recruited study participants via local mailing lists, Craigslist, Twitter, and local groups on Facebook. We compensated them with a $5 gift card. We excluded acquaintances from participating in the studies to avoid acquiescence bias. Before starting the study, participants had to sign our consent form and complete an entry survey containing demographic questions. We made sure to get a wide diversity of subjects, both in terms of age and experience with technology (details available in Appendix A). For all the experiments, we configured the test environment on LG Google Nexus 5X phones running the Android 7.1 Nougat OS. We used a background service, automatically relaunched at boot time, to log participants' responses to system messages and alerts, input events generated by participants while interacting with the testing programs, as well as system events and inter-process communications between programs. Furthermore, during the experiments, the researchers took note of comments made by participants to ease the analysis of user decision making.

6.2 Laboratory Study

We performed a *laboratory study* to evaluate the effectiveness of EnTrust in supporting users in avoiding all the three attack vectors previously identified in Section 2. We compared EnTrust with the *first-use* authorization used in commercial systems. We could not compare mechanisms proposed in related work [9, 10], because they are unable to handle handoff events. We divided participants into four groups, participants in `Group-FR-U` and `Group-FR-P` interacted with a *stock* Android OS implementing the *first-use* authorization mechanism. Participants in `Group-EN-U` and `Group-EN-P` interacted with a *modified* version of the Android OS integrating the EnTrust authorization system. To account for the priming effect, we avoided influencing subjects in `Group-FR-U` and `Group-EN-U` and advertised the test as a generic "voice assistants testing" study without mentioning security implications. On the other hand, to assess the impact of priming, subjects in `Group-FR-P` and `Group-EN-P` were informed that attacks targeting sensors (e.g., camera, microphone, and GPS receiver) were possible during the interaction with programs involved in the experimental tasks, but without specifying what program performed the attacks or what attacks were performed.

Experimental Procedures: For our experiment, we used a test assistant developed in our research lab called Smart Assistant, which provides basic virtual assistant functionality, such as voice search, message composition, and note keeping. However, Smart Assistant is also designed to perform confused deputy attacks on

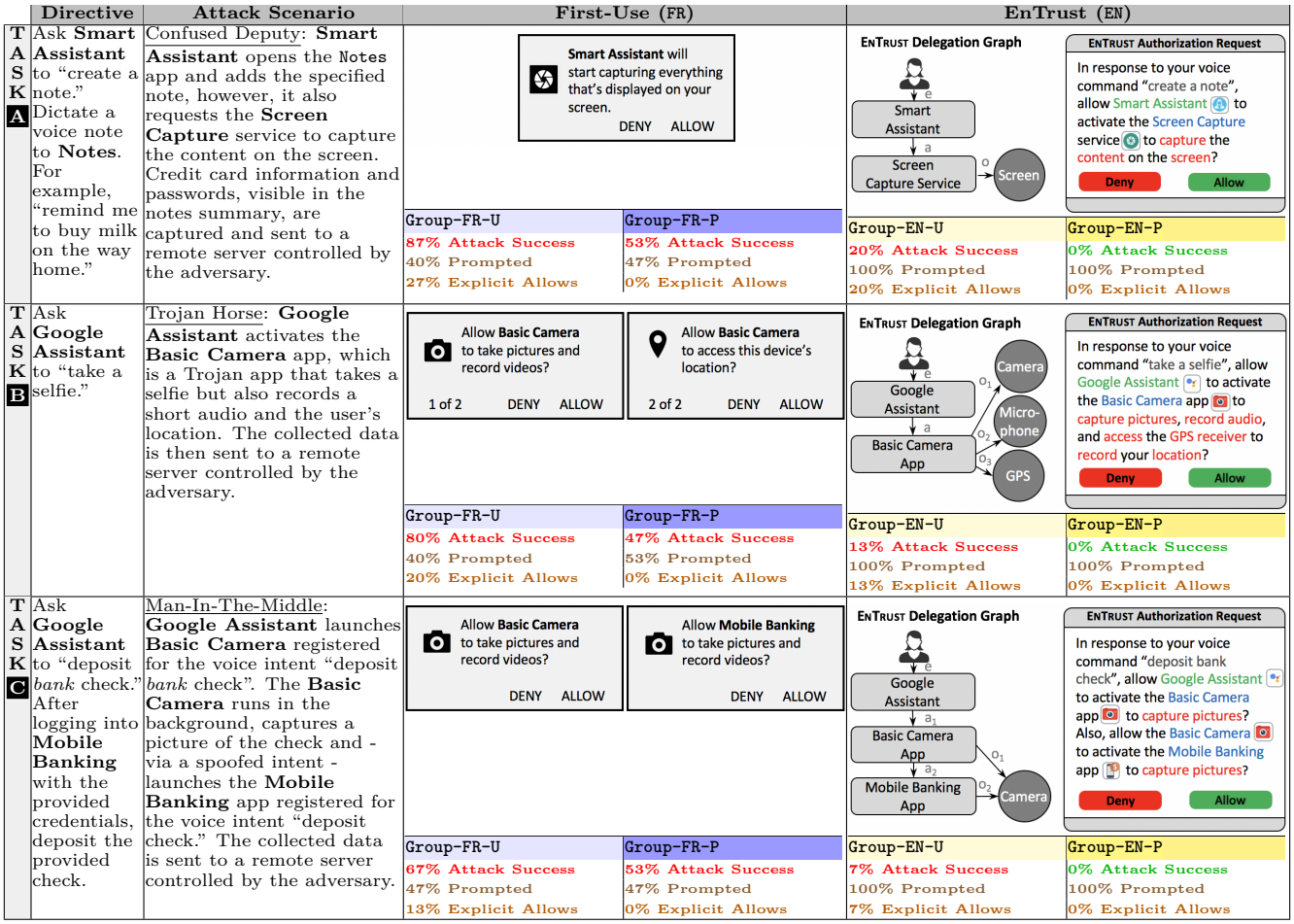

Table 1: Experimental tasks for the laboratory study, derived from the attack vectors described in Section 2. We report the authorization messages shown to subjects in the four groups as well as the delegation graphs used by EnTrust to construct such authorization messages. In the group names, the suffix U indicates *unprimed* subjects, whereas P indicates *primed* subjects. Notice that, authorization requests prompted by EnTrust include programs' identity marks (i.e., apps' icon and unique id).

system services, such as the Screen Capture service. We also used a test app, Basic Camera, developed in our research lab. It provides basic camera functionality, such as capturing pictures or videos and applying photographic filters. However, Basic Camera is also designed to perform man-in-the-middle and Trojan horse attacks for requests to capture photographic frames. Lastly, we used a legitimate Mobile Banking app, from a major international bank, available on Google Play Store. Apart from the testing apps and voice assistant, the smartphone provided to participants had pre-installed both the Google Assistant and the Android Camera app.

Our laboratory study was divided into two phases. A *preliminary phase* during which no attacks were performed. This phase enabled participants to familiarize themselves with the provided smartphone, the pre-installed apps and the voice assistants. This phase avoided a "cold start" and approximated a more realistic scenario in which users have some experience using relevant apps and voice assistants. Furthermore, this preliminary phase enabled capturing how malicious programs may leverage pre-authorized operations in the first-use approach to then perform operations not expected by the users; a malicious behavior that is instead prevented by EnTrust via the construction of per-delegation authorizations. The preliminary phase was then followed by an *attack phase*, during which participants interacted with programs performing attacks. Participants were not made aware of the existence of the two experimental phases nor of the difference between the two phases.

All instructions regarding experimental tasks to be performed were provided to participants in writing via a handout at the beginning of each experimental task. During the *preliminary phase* the participants performed the following three tasks: (1) asked a voice assistant to "take a screenshot;" (2) asked a voice assistant to "record a memo;" and (3) used a camera app to "record a video." During the *attack phase*, instead, the participants performed the three tasks described in Table 1. In each phase, each participant was presented with a different randomized order of the above tasks.

Experimental Results: In total, 60 subjects participated in and completed our laboratory study. We *randomly* assigned 15 participants to each group. In this study, *we did not observe denials of legitimate operations* for sensitive sensors for non-attack tasks performed during the preliminary phase, but we discuss the need for more study on preventing and resolving mistaken denials in Section 8. Table 1 summarizes the results of the three experimental tasks for the *attack phase*. Our focus was to study the effectiveness of EnTrust in reducing the success rate of attacks when compared to the first-use approach. During the *preliminary phase* and the *experimental tasks*, *all the participants were prompted* with the corresponding authorization messages depending on the group to which they were assigned,[5] as reported in Table 1. Prompted authorizations included legitimate operations, see left side of Figure 6 for an example of what a prompt for a legitimate operation looked like. Our analysis reports that each participant of each group was prompted at least 4 times for non-attack operations. Note that, as per definition of first-use authorization, participants in Group-FR-U and Group-FR-P were not prompted with authorization messages once again should they have already authorized the program in a previous task or during the preliminary phase. Instead, participants in Group-EN-U and participants in Group-EN-P were presented with a new authorization message any time a new delegation path was identified by EnTrust. This explains the lower percentage of subjects prompted, with an authorization request, in the first-use groups.

TASK A: The analysis of subjects' responses revealed that 9 subjects from Group-FR-U and 8 subjects from Group-FR-P interacted with Smart Assistant during the preliminary phase, or during another task, to "take a screenshot" and granted the app permission to capture their screen; thus, they were not prompted once again with an authorization message during this task, as per default in first-use permissions. In addition, 4 subjects from Group-FR-U explicitly allowed Smart Assistant to capture their screen, therefore, resulting in a 87% and 53% attack success, respectively, as reported in Table 1. On the contrary, only 3 subjects from Group-EN-U and no subject from Group-EN-P allowed the attack (20% and 0% attack success, respectively). Also, similarly to what happened in Group-FR-U and Group-FR-P, 8 subjects from Group-EN-U and 8 subjects from Group-EN-P interacted with Smart Assistant during the preliminary phase and asked to "take a screenshot." However, since the voice command "create a note" was a different command, EnTrust prompted all subjects with a new authorization message, as shown in Table 1.

TASK B: The analysis of subjects' responses revealed that 9 subjects from Group-FR-U and 7 subjects from Group-FR-P interacted with Basic Camera to take a picture or record a video, either during the preliminary phase or during another task, and authorized it to capture pictures, audio, and access the device's location. Thus, they were not prompted once again during this task as per default in first-use permissions. Also, we found that 3 subjects from Group-FR-U explicitly authorized Basic Camera to access the camera, as well as the microphone, and the GPS receiver; therefore, resulting in 80% and 47% attack success, respectively. In contrast, 2 subjects from Group-EN-U and no subject from Group-EN-P authorized access to the camera, microphone, and GPS receiver (13% and 0% attack success, respectively). Also, we found that 8 subjects from Group-EN-U and 6 subjects from Group-EN-P interacted with Basic Camera during the preliminary phase or during another task. However, none of them asked to "take a selfie" before, so all subjects were prompted by EnTrust with a new authorization message. At the end of the experiment, among all the subjects, when asked why they authorized access to the GPS receiver, the majority said that they expected a camera app to access location to create geo-tag metadata when taking a picture. In contrast, the subjects who denied access stated not feeling comfortable sharing their location *when taking a selfie*.

TASK C: The analysis of subjects' responses revealed that 8 subjects from Group-FR-U and 8 subjects from Group-FR-P interacted with Basic Camera, either during the preliminary phase or during another task, and authorized the app to capture pictures. Thus, during this task, they were not prompted with an authorization message once again as per default in first-use permissions. They were only prompted to grant permission to Mobile Banking, explaining why even the primed subjects were not able to detect the attack. In addition, 2 subjects from Group-FR-U explicitly authorized Basic Camera to capture a frame with the bank check; therefore, resulting in 67% and 53% attack success, respectively. On the other hand, only 1 subject from Group-EN-U and no subject from Group-EN-P authorized Basic Camera to capture a frame with the bank check, resulting in a 7% and 0% attack success, respectively. Notice that all subjects from Group-EN-U and Group-EN-P were prompted with a new authorization message by EnTrust for the new command "deposit *bank* check." Interestingly, the one subject from Group-EN-U, who allowed Basic Camera to capture a frame with the bank check, verbally expressed his concern about the permission notification presented on the screen. The subject stated observing that two apps asked permission to access the camera to take pictures. This is reasonable for an unprimed subject not expecting a malicious behavior.

Discussion: Comparing the results from Group-FR-U versus those from Group-FR-P, and those from Group-EN-U versus those from Group-EN-P, we observe - as expected - that primed subjects allowed fewer attacks. We find that users primed for security problems still fall victim to attacks due to first-use authorization,

even when rejecting all the malicious operations they see. On the other hand, unprimed users fail to detect attacks between 7-20% with EnTrust. So while this is a marked improvement, over the 67-87% failure for users with first-use authorization, there is room for further improvement. However, it is apparent that the delegation graphs constructed by EnTrust aided the subjects in avoiding attacks even when unprimed. EnTrust performed slightly better than first-use authorization in terms of explicit authorizations (explicit allows in Table 1); which suggests that the additional information provided by EnTrust in authorization messages (i.e., programs' name and identity mark as well as delegation information, as shown in Figure 6) may be helpful to users in avoiding unexpected program behaviors. We verified the hypothesis that the information in EnTrust authorizations helps *unprimed* users identify attacks by calculating the difference in *explicit allows*, across the three experimental tasks, for subjects in Group-FR-U versus subjects in Group-EN-U. Our analysis indeed revealed a statistically significant difference ($\chi^2 = 19.3966$; p = 0.000011).

Also, EnTrust was significantly more effective than first-use in keeping users "on guard" independently of whether subjects were primed (47-67% lower attack success with EnTrust). Indeed, different from the first-use approach, EnTrust was able to highlight whether pre-authorized programs attempted accessing sensors via unauthorized delegation paths. If so, EnTrust prompted users for an explicit authorization for the newly identified delegation path. We verified the hypothesis that EnTrust better helps *primed* and *unprimed* users in preventing attacks than first-use, by calculating the difference in *successful attacks*, across the three experimental tasks, for subjects in Group-FR-U and Group-FR-P, versus subjects in Group-EN-U and Group-EN-P. Our analysis indeed revealed a statistically significant difference ($\chi^2 = 65.5603$; p = 0.00001). Normally, the standard Bonferroni correction would be applied for multiple testing, but due to the small p-values such a correction was not necessary.

6.3 Field Study

We performed a *field study* to evaluate whether EnTrust increases the decision-overhead imposed on users. We measured the number of explicit authorizations users had to make when interacting with EnTrust under realistic and practical conditions, and compared it with the first-use approach adopted in commercial systems (i.e., Android OS and Apple iOS). We also measured the number of authorizations handled by EnTrust via the cache mechanism that, transparently to users, granted authorized operations.

Experimental Procedures: Participants met with one of our researchers to set up the loaner device, an LG Nexus 5X smartphone running a *modified* version of the Android OS integrating the EnTrust authorization framework. The loaner device had pre-installed 5 voice assistants and 10 apps selected among the most popular[6] with up to millions of downloads from the official Google Play store. For such programs, to ensure the confidentiality of participants' personal information, mock accounts were set up instead of real accounts for all apps requiring a log-in. To facilitate daily use of the loaner device, the researcher transferred participants' SIM cards and data, as well as participants' apps in the loaner device, however no data was collected from such apps. The above protocol was a requirement for the IRB approval by our Institution and it is compliant with the protocol followed in related work [33, 15, 11]. Before loaning the device, the researcher asked each participant to use the loaner device for their everyday tasks for a period of 7 days. In addition to their everyday tasks, participants were asked to explore each of the pre-installed voice assistants and apps, at least once a day, by interacting as they would normally do. Particularly, we asked the participants to interact with each voice assistant by asking the following three questions: (1) "capture a screenshot," (2) "record a voice note," (3) "how long does it take to drive back home." Additionally, we asked participants to be creative and ask three additional questions of their choice. Table 2 summarizes all the assistants and apps pre-installed on the smartphones for the field study. Because the mere purpose of our field study was to measure the decision-overhead imposed to users by EnTrust and to avoid participants' bias, the researcher advertised the study as a generic "voice assistants and apps testing" study without mentioning security implications or training the users about the features provided by EnTrust. The smartphones provided to participants were running a background service with runtime logging enabled, automatically restarted at boot time, to monitor the number of times each program was launched, the users' input events, the constructed delegation graphs, the authorization decisions made by the participants, and the number of authorizations automatically granted by EnTrust. The background service also measured the gaps between consecutive input events and handoff events, as well as the time required by each program to service each event. This data was used to perform the time constraints analysis reported in Appendix B.

Experimental Results: Nine subjects participated and completed the field study. The data collected during our experiment indicates that all user authorizations were obtained within the first 72 hours of interaction with the experimental device, after which we observed only operations automatically granted by EnTrust via the caching mechanism.

The first subject allowed us to discover two implementation issues that affected the number of explicit authorizations required by EnTrust. First, changing the orientation of the screen (portrait versus landscape) was causing EnTrust to request a new explicit user autho-

	Expl. Authorizations		Impl. Authorizations
	First-Use	EnTrust	in s 7 Days Period
Snapchat	3	3	276
YouTube	3	3	84
Facebook Messenger	2	2	93
Instagram	3	3	393
Facebook	3	3	117
Whatsapp	2	2	76
Skype	3	3	100
WeChat	2	2	101
Reddit	1	1	18
Bitmoji	3	3	127
Google Assistant	1	**4**	72
Microsoft Cortana	1	**3**	49
Amazon Alexa	1	**4**	84
Samsung Bixby	1	**4**	63
Lyra Virtual Assistant	1	**3**	56

Table 2: Apps and voice assistants tested in the field study. The last column shows the number of operations automatically authorized by EnTrust after user's authorization.

rization for an already authorized widget whenever the screen orientation changed. This inconvenience was due to the change in some of the features used to model the context within which the widget was presented. To address this shortcoming, we modified our original prototype to force the Window Manager to generate in memory two graphical user interfaces for both screen orientations to allow EnTrust to bind them with a specific widget presented on the screen. Second, for the voice commands, we noticed that differently phrased voice commands with the same meaning would be identified as different input events. For instance, "take a selfie" and "take a picture of me". This shortcoming was causing EnTrust to generate a new delegation graph for each differently phrased voice command. To address this issue, we leveraged the *Dialogflow* engine by Google, part of the AI.API.[7] *Dialogflow* is a development suite for building conversational interfaces and provides a database of synonyms to group together voice commands with the same meaning. We fixed the issues and continued our experiment with other subjects.

Table 2 reports the average number of explicit authorizations performed by the subjects. We compared them with the number of explicit authorizations that would be necessary if the *first-use* permission mechanism was used instead. The results show that EnTrust required the same number of explicit authorizations by users for all the tested apps. For all voice assistants, instead, EnTrust may require up to 3 additional explicit authorizations, when compared with the *first-use* approach; which is far below the 8 additional explicit authorizations used in prior work, which are considered likely not to introduce significant risk of habituation or annoyance [33]. These additional authorizations are due to the fact that with the *first-use* approach the programs (activated by the voice assistant to serve the user request) may have already received the required permissions to access the sensitive sensors. EnTrust instead captures the entire sequence of events, from the input event to any subsequent action or operation request, and then ties them together. Therefore, EnTrust constructs a new graph for each novel interaction. Nonetheless, the number of decisions imposed on the users remains very modest. Indeed, on average, three additional explicit user authorizations are required per voice assistant. Also, the number of explicit authorizations made by the users remained a constant factor compared to the number of operations implicitly authorized by EnTrust, which instead grew linearly over time. We measured an average of 16 operations implicitly authorized by EnTrust during a 24-hour period (last column of Table 2). Therefore, if we consider such a daily average number of implicitly authorized operations for a period of one year, we will have on the order of thousands of operations automatically authorized by EnTrust, which would not require additional explicit effort for the users.

6.4 Backward Compatibility Analysis

To verify that EnTrust is backward compatible with existing programs, we used the Compatibility Test Suite (CTS),[8] an automated testing tool released by Google via the AOSP.[9] In particular, this analysis verified that possible delays in the delivery of events introduced by EnTrust or the change in scheduling of events did not impact applications' functionality. We tested the compatibility of EnTrust with 1,000 existing apps, among the top 2,000 most downloaded apps on Google Play Store, selected based on those declaring permissions to access sensitive sensors in their manifest. The experiment took 19 hours and 45 minutes to complete, and EnTrust passed 132,681 tests without crashing the operating system and without incorrectly blocking any legitimate operation. Among the 1,000 tested apps, we also included 5 popular augmented reality multiplayer gaming app (`InGress`, `Pokémon Go`, `Parallel Kingdom`, `Run An Empire`, and `Father.io`), which typically have a high rate of input events and are very sensitive to delays. The set of tests targeting these 5 gaming apps ran for 16 minutes, during which we continuously observed the device screen to identify possible issues in terms of responsiveness to input events or glitches in the rendering of virtual objects on the screen. However, we did not identify any discernible slowdown, glitch, or responsiveness issue.

7 Performance Measurements

We performed four micro-benchmarks on a standard Android developer smartphone, the LG Nexus 5X, powered by 1.8GHz hexa-core 64-bit Qualcomm Snapdragon 808 Processor and Adreno 418 GPU, 2GB of RAM, and 16GB of internal storage. All of our benchmarks are measured using Android 7.1 Nougat pulled from the Android Open Source Project (AOSP) repository.

Delegation Graph Construction – Our first micro-benchmark of EnTrust measured the overhead incurred for constructing delegation graphs of varying sizes. To do this, we had several programs interacting

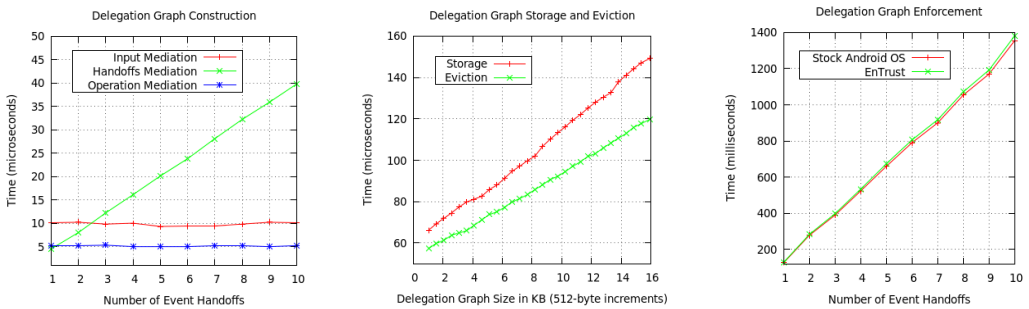

Figure 7: Overheads for Delegation Graphs Construction, Storage, Eviction, and Enforcement.

and generating a handoff-events chain varying from 1 to 10 handoffs in length and measured the time to mediate the input event, the handoff event, and the operation request. We repeated the measurements 100 times. Each set of measurements was preceded by a priming run to remove any first-run effects. We then took an average of the middle 8 out of 10 such runs for each number of handoff events. The results in Figure 7 show that the input mediation requires an overhead of 10 μs, the handoff event mediation requires an additional overhead of 4 μs per event handoff, whereas the operation mediation requires a fixed overhead of 5 μs. The overheads are within our expectations and do not cause noticeable performance degradation.

Delegation Graph Caching – Our second micro-benchmark of EnTrust measures the overhead incurred for caching delegation graphs constructed at runtime. We measured the overhead introduced by EnTrust in the authorization process for both storing a new delegation graph, as well as evicting from cache a stale one. To do this, we simulated the creation and eviction of delegation graphs of different sizes varying from 1 to 16 Kilobytes in 512-byte increments.[10] We repeated the measurement 5 times for each random size and took an average of the middle 3 out of 5 such runs. The results in Figure 7 show that the storing of delegation graphs in the cache required a base overhead of 66 μs with an additional 3 μs per 512-byte increment. The eviction instead required a base overhead of 57 μs with an additional 2.5 μs for each 512-byte increment.

Delegation Graph Enforcement – Our third micro-benchmark was designed to compare the unmodified version of the Android Nougat build for control measurement with a modified build integrating our EnTrust features for the delegation graph enforcement during authorization. To guarantee fairness in the comparison between the two systems, we used the Android UI/Application Exerciser Monkey[11] to generate the same sequence of events for the same set of programs. For both systems, we measured the total time needed to authorize a sensor operation as the time from the input event to the authorization of the resulting operation request, corresponding to the last node of the delegation graph for EnTrust. We repeated the measurement 100 times for each system by varying the number of handoff events from 1 to 10. Each set of measurements was preceded by a priming run to remove any first-run effects. We then took an average of the middle 8 out of 10 such runs for each number of handoff events. Figure 7 shows that the overhead introduced by EnTrust for the delegation graph enforcement is negligible, with the highest overhead observed being below 0.02%. Thus, the slowdown is likely not to be noticeable by users. Indeed, none of our study participants raised any concerns about discernible performance degradation or system slowdown.

Ambiguity Prevention – Our fourth micro-benchmark was designed to measure the performance implications, in terms of delayed events, due to the ambiguity prevention mechanism. For this micro-benchmark, we selected the system UI (User Interface) process, which is one of the processes receiving the highest number of input events, and the media server process that receives the highest number of handoff events and performs sensor operations with higher frequency than any other process. The time window for the construction of each delegation path was set to 150 ms. We generated 15,000 input events with gaps randomly selected in the range [140-1,500][12] ms. The time window and the gaps were selected based on data reported in Appendix B. The generated input events caused 2,037 handoff events and 5,252 operation requests targeting sensors (22,289 total scheduled events). The results indicated a total of 256 delayed events (1.15% of the total events), with a maximum recorded delay of 9 ms. Thus, the performance overhead introduced is negligible.

Memory Requirement – We also recorded the average cache size required by EnTrust to store both event mappings and authorized delegation graphs to be about 5.5 megabytes, for up to 1,000 programs.[13] Therefore, EnTrust required about 5.5 kilobytes of memory per program, which is a small amount of memory when compared to several gigabytes of storage available in modern systems. We ran the measurement 10 times and then took an average of the middle 8 out of 10 of such runs.

8 Discussion of Limitations

Evaluating mechanisms that prevent abuse of sensitive sensors while trading off privacy and usability is chal-

lenging. In this section, we discuss the limitations of our study and provide guidance on future work.

Authorization Comprehension – In designing our authorization messages, we have used the language adopted in current permission systems (e.g., Android OS and Apple iOS) and prior research work [47, 11, 15, 16] as references. However, such language may not be as effective in eliciting access control decisions from users as desired. Further improvements may be possible by studying NLP techniques and how access control questions may be phrased using such techniques. Also, a combination of text, sound, and visuals may be useful in conveying access questions to users. EnTrust is largely orthogonal to any specific way how access control questions are presented, enabling it to be used as a platform for further study.

Decision Revocation – Users may make mistakes when allowing or denying authorizations. EnTrust caches user decisions to reduce users' authorization effort, allowing such mistakes to persist. Mistakes in authorizing access to sensor operations may permit malicious applications to abuse access, albeit limited to that delegation path only. Mistakes in denying access to sensor operations prevents legitimate use of sensor operations silently as a result of caching. One possible solution to these problems is to invalidate the cache periodically to prevent stale authorization decisions. However, frequent authorization prompts negatively affect user experience. Currently, EnTrust enables users to review authorization decisions via an audit mechanism, as suggested elsewhere [53, 15]. However, to improve the effectiveness of such mechanisms, further laboratory studies will be necessary to examine how to present audit results (or other new approaches) to help users to investigate and resolve mistaken authorizations.

Study Scenarios – In this project, we focused on whether users would be able to deny attack scenarios effectively. Another problem is that users may not evaluate non-attack scenarios correctly once they become aware of possible attacks. In our study, we did not observe that users denied any legitimate sensor operations during the lab study, but it would be beneficial to extend the laboratory study to include more subtle non-attack scenarios, where we push the boundaries of what is perceived as benign, to evaluate whether these scenarios may cause false denials due to users being unable to identify that the request was indeed benign. Also, we recognize that all attacks were generated by programs unfamiliar to participants, even though they were given the opportunity to familiarize themselves with such programs during the preliminary phase of our lab study.

Study Size – The number of subjects recruited for this project, 60 for the laboratory study and 9 for the field study, is comparable with the number of subjects in similar studies [33, 14, 15, 11]. Other related work [47] had a higher number of subjects, but subjects were not required to be physically present in the laboratory during the experimental tasks having been recruited via online tools (e.g., Mechanical Turk). However, research has shown, in the context of password study, that a laboratory study may produce more realistic results than an online study [58].

Study Comprehensiveness – Our study does not focus explicitly on long-term habituation, user annoyance, and users' attitudes toward privacy. Researchers have already extensively studied users' general level of privacy concerns [51, 52, 53, 15]. Other researchers have studied users' habituation for first-use authorization systems extensively [33, 45, 50]. Our field study (Section 6.3) shows that our approach is comparable to first-use in terms of the number of times users are prompted, and the number of explicit authorizations from users is far below the 8 additional explicit authorizations used in prior work, which are considered likely not to introduce significant risk of habituation or annoyance [33].

9 Related Work

Researchers have extensively demonstrated that IPC mechanisms allow dangerous interactions between programs, such as unauthorized use of intents, where adversaries can hijack activities and services by stealing intents [18, 21, 22, 26]. Prior work has also shown that such interactions can be exploited by adversaries to cause permission re-delegations [7] in the attempt to leverage capabilities available to trusted programs (e.g., system services). Also, related work has demonstrated how trusted programs inadvertently or purposely (for functionality reasons) expose their interfaces to other programs [8], thus exposing attack vectors to adversaries. In this paper, we have demonstrated that dangerous interactions among programs can lead to critical attack vectors related to input event delegations.

Researchers have tried to regulate such interactions with automated tools for IPC-related vulnerability analysis. For instance, *ComDroid* is a tool that parses the disassembled applications' code to analyze intent creation and transition for the identification of unauthorized intent reception and intent spoofing [26]. *Efficient and Precise ICC discovery* (EPICC) is a more comprehensive static analysis technique for Inter-Component Communication (ICC)[14] calls [19]. It can identify ICC vulnerabilities due to intents that may be intercepted by malicious programs, or scenarios where programs expose components that can be launched via malicious intents. *Secure Application INTeraction* (*Saint*) [54] extends the existing Android security architecture with policies that would allow programs to have more control to whom permissions for accessing their interfaces are granted and used at runtime. *Quire* provides context in the form of provenance to programs communicating via Inter-Procedure Calls (IPC) [55]. It annotates IPCs occurring within a system, so that the recipient of an

IPC request can observe the full call sequence associated with it, before committing to any security-relevant decision. Although effort has been made to analyze and prevent IPC-related vulnerabilities, none of the proposed approaches above tackled the problem from our perspective, i.e., instead of giving control to application developers, we must give control to users who are the real target for privacy violations by malicious programs.

In line with our perspective of giving control to users, *User-Driven Access Control* [9, 10] proposes the use of access control gadgets, predefined by the operating systems and embedded into applications' code, to limit what operation can be associated with a specific input event. Also, *AWare* [11] proposes to bind each operation request targeting sensitive sensors to an input event and to obtain explicit authorization from the user for each event-operation combination. Similarly, *ContexIoT* [16] is a context-based permission system for IoT platforms which leverages runtime prompts with rich context information including the program execution flow that allows users to identify how a sensitive operation is triggered. Unfortunately, all of these mechanisms only control how the input event is consumed by the program receiving the input event. The proposed mechanisms to enable mediation do not mediate inter-process communication and the operations resulting from such communication (e.g., *event delegations between programs*), which is necessary to prevent the attack vectors discussed in this paper. Also, differently from prior work on permission re-delegation [7], we do not rely on an over-restrictive defense mechanism that totally forbids programs from using their additional privileges. Such an over-restrictive defense would block necessary interactions between programs even when the interactions are benign and expected by users.

Prior work has also investigated the use of machine learning classifiers to analyze the contextuality behind user decisions to automatically grant access to sensors [14, 15]. Unfortunately, such classifiers only model the context relative to the single program that the user is currently interacting with, and the API calls that are made by such a program during the interaction. However, the context modeled by these classifiers does not account for inter-process communications, which allow programs to enlist other programs to perform sensor operations via input event delegation. Furthermore, the effectiveness of the learning depends on the accuracy of the user decisions used in training the learner. In other words, if the user's decisions suffer from inadequate information during the training phase, the learner will as well. Therefore, we firmly believe that an additional effort is necessary to support user decision making before the user decisions can be used to train a classifier.

Lastly, mechanisms based on taint analysis [29, 30, 31] or Decentralized Information Flow Control (DIFC) [13, 20] have been proposed by researchers to, respectively, track and control how sensitive data is used by or shared between programs. However, such mechanisms solve the orthogonal problem of controlling sensitive data leakage or accidental disclosure, rather than enabling users to control *how*, *when*, and *which* programs can access sensors for the collection of sensitive data.

10 Conclusion

While a collaborative model allows the creation of useful, rich, and creative applications, it also introduces new attack vectors that can be exploited by adversaries. We have shown that three well-studied attack vectors become critical, in operating systems supporting a cooperating program abstraction, and proposed the EnTrust authorization system to help mitigate them. EnTrust demonstrates that it is possible to prevent programs from abusing the collaborative model – in the attempt to perform delegated confused deputy, delegated Trojan horse, or delegated man-in-the-middle attacks – by binding together, input event, handoff events, and sensor operation requests made by programs, and by requiring an explicit user authorization for the constructed delegation path. Our results show that existing systems have room for improvement and permission-based systems, as well as machine learning classifiers, may significantly benefit from applying our methodology.

Acknowledgements

Thanks to our shepherd, Sascha Fahl, and the anonymous reviewers. The effort described in this article was partially sponsored by the U.S. Army Research Laboratory Cyber Security Collaborative Research Alliance under Contract Number W911NF-13-2-0045. The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation hereon. The research work of Jens Grossklags was supported by the German Institute for Trust and Safety on the Internet (DIVSI).

References

[1] Conger, K. Researchers: Uber's iOS app had secret permissions that allowed it to copy your phone screen. *Gizmodo*, (2017).

[2] Lieberman, E. Hackers are gunning for your personal data by tracking you. *The Daily Caller*, (2016).

[3] Sulleyman, A. Android apps secretly steal users' data by colluding with each other, finds research. *Independent*, (2017).

[4] Revel, T. Android apps share data between them without your permission. *NewScientist*, (2017).

[5] Norm, H., The Confused Deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, (1988).

[6] Petracca, G., Sun, Y., Jaeger, T., and Atamli, A. AuDroid: Preventing attacks on audio channels in mobile devices. In *ACSAC*, (2015), ACM.

[7] Felt, A. P., Wang, H., Moshchuk, A., Hanna, S., and Chin, E. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, (2011).

[8] Aafer, Y., Zhang, N., Zhang, Z., Zhang, X., Chen, K., Wang, X., Zhou, X., Du, W., and Grace, M. Hare hunting in the wild Android: A study on the threat of hanging attribute references. In *CCS*, (2015), ACM.

[9] Roesner, F., Kohno, T., Moshchuk, A., Parno, B., Wang, H., and Cowan, C. User-driven access control: Rethinking permission granting in modern operating systems. In *S&P*, (2012), IEEE.

[10] Ringer, T., Grossman, D., and Roesner, F. Audacious: User-driven access control with unmodified operating systems. In *CCS* (2016), ACM.

[11] Petracca, G., Reineh, A.-A., Sun, Y., Grossklags, J., and Jaeger, T. AWare: Preventing abuse of privacy-sensitive sensors via operation bindings. In *USENIX Security Symposium*, (2017).

[12] Onarlioglu, K., Robertson, W., and Kirda, E. Overhaul: Input-driven access control for better privacy on traditional operating systems. In *DSN*, (2016), IEEE/IFIP.

[13] Nadkarni, A., and Enck, W. Preventing accidental data disclosure in modern operating systems. In *CCS*, (2013), ACM.

[14] Wijesekera, P., Baokar, A., Tsai, L., Reardon, J., Egelman, S., Wagner, D., and Beznosov, K. The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences. In *S&P* (2017), IEEE.

[15] Olejnik, K., Dacosta, I., Machado, J.S., Huguenin, K., Khan, M.E., and Hubaux, J.P. Smarper: Context-aware and automatic runtime-permissions for mobile devices. In *S&P*, (2017), IEEE.

[16] Jia, Y. J., Chen, Q. A., Wang, S., Rahmati, A., Fernandes, E., Mao, Z. M., and Prakash, A. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *NDSS*, (2017).

[17] Acar, Y., Backes, M., Bugiel, S., Fahl, S., McDaniel, P. and Smith, M., Sok: Lessons learned from android security research for appified software platforms. In *S&P*, (2017), IEEE.

[18] Li, L., Bartel, A., Bissyandé, T. F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., and McDaniel, P. Iccta: Detecting inter-component privacy leaks in Android apps. In *ICSE*, (2015), IEEE.

[19] Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., and Le Traon, Y. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In *USENIX Security Symposium*, (2013).

[20] Nadkarni, A., Andow, B., Enck, W., and Jha, S. Practical DIFC enforcement on Android. In *USENIX Security Symposium*, (2016).

[21] Octeau, D., Luchaup, D., Dering, M., Jha, S., and McDaniel, P. Composite constant propagation: Application to Android inter-component communication analysis. In *ICSE*, (2015), IEEE.

[22] Octeau, D., Jha, S., Dering, M., McDaniel, P., Bartel, A., Li, L., Klein, J., and Le Traon, Y. Combining static analysis with probabilistic models to enable market-scale Android inter-component analysis. In *ACM SIGPLAN Notices*, (2016).

[23] Krohn, M.N., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M.F., Kohler, E., and Morris R. Information flow control for standard OS abstractions. In *SOSP*, (2007).

[24] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D. Making information flow explicit in HiStar. In *OSDI*, (2006).

[25] Chatterjee, R., Doerfler, P., Orgad, H., Havron, S., Palmer, J., Freed, D., Levy, K., Dell, N., McCoy, D., and Ristenpart, T. The Spyware Used in Intimate Partner Violence. In *S&P*, (2018), IEEE.

[26] Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. Analyzing inter-application communication in Android. In *MobiSys* (2011), ACM.

[27] Huang, L.-S., Moshchuk, A., Wang, H. J., Schecter, S., and Jackson, C. Clickjacking: Attacks and defenses. In *USENIX Security Symposium*, (2012).

[28] Luo, T., Jin, X., Ananthanarayanan, A., and Du, W. Touchjacking attacks on web in Android, iOS, and Windows phone. In *FPS* (2012).

[29] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX OSDI* (2010).

[30] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *ACM Sigplan Notices*, (2014), pp. 259–269.

[31] Tang, Y., Ames, P., Bhamidipati, S., Bijlani, A., Geambasu, R., and Sarda, N. CleanOS: Limiting mobile data exposure with idle eviction. In *USENIX OSDI* (2012).

[32] Sun, Y., Petracca, G., Ge, X., and Jaeger, T. Pileus: Protecting user resources from vulnerable cloud services. In *ACSAC*, (2016), ACM.

[33] Wijesekera, P., Baokar, A., Hosseini, A., Egelman, S., Wagner, D., and Beznosov, K. Android permissions remystified: A field study on contextual integrity. *USENIX Security Symposium*, (2015).

[34] Levy, H. M. *Capability-Based Computer Systems*. Digital Press. Available at http://www.cs.washington.edu/homes/levy/capabook/, (1984).

[35] Prevelakis, V., and Spinellis, D. Sandboxing applications. In *USENIX Annual Technical Conference, FREENIX Track*, (2001).

[36] Chang, F., Itzkovitz, A., and Karamcheti, V. User-level resource-constrained sandboxing. In *USENIX Windows Systems Symposium*, (2000).

[37] Smalley, S., Vance, C., and Salamon, W. Implementing SELinux as a Linux security module. *NAI Labs Report #01-043*, (2001).

[38] Smalley, S., and Craig, R. Security Enhanced (SE) Android: Bringing flexible MAC to Android. In *NDSS*, (2013).

[39] Ye, Z., Smith, S., and Anthony, D. Trusted paths for browsers. *ACM Transactions on Information and System Security*, (2005).

[40] Zhou, Z., Gligor, V., Newsome, J., and McCune, J. Building verifiable trusted path on commodity x86 computers. In *S&P*, (2012), IEEE.

[41] Shapiro, J., Vanderburgh, J., Northup, E., and Chizmadia, D. Design of the EROS trusted window system. In *USENIX Security Symposium*, (2004).

[42] Li, W., Ma, M., Han, J., Xia, Y., Zang, B., Chu, C.-K., and Li, T. Building trusted path on untrusted device drivers for mobile devices. In *Asia-Pacific Workshop on Systems*, (2014), ACM.

[43] Eugster, P., Felber, P., Guerraoui, R., and Kermarrec, A.-M. The many faces of publish/subscribe. *ACM Computing Surveys*, (2003).

[44] Mellor-Crummey, J. M., and Scott, M. L. Scalable reader-writer synchronization for shared-memory multiprocessors. *ACM SIGPLAN Notices*, (1991).

[45] Felt, A. P., Ha, E., Egelman, S., Haney, A., Chin, E., and Wagner, D. Android permissions: User attention, comprehension, and behavior. In *SOUPS*, (2012), ACM.

[46] Rivest, R., Shamir, A., and Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, (1978).

[47] Bianchi, A., Corbetta, J., Invernizzi, L., Fratantonio, Y., Kruegel, C., and Vigna, G. What the App is that? Deception and countermeasures in the Android user interface. *S&P*, (2015), IEEE.

[48] Cummings, P., Fullan, D.A., Goldstien, M.J., Gosse, M.J., Picciotto, J., Woodward, J.P., and Wynn, J. Compartmented Model Workstation: Results through prototyping. *S&P*, (1987), IEEE.

[49] Schechter, S. Common pitfalls in writing about security and privacy human subjects experiments, and how to avoid them. Microsoft Tech. Rep. (2013).

[50] Felt, A. P., Egelman, S., Finifter, M., Akhawe, D., and Wagner, D. How to ask for permission. In *USENIX Workshop on Hot Topics in Security* (2012).

[51] Sheehan, K.B. Toward a typology of Internet users and online privacy concerns. *The Information Society*, (2012).

[52] Debatin, B., Lovejoy, J.P., Horn, A.K., and Hughes, B.N. Facebook and online privacy: Attitudes, behaviors, and unintended consequences. *Journal of Computer-Mediated Communication*, (2009).

[53] Petracca, G., Atamli-Reineh, A., Sun, Y., Grossklags, J., and Jaeger, T. Aware: Controlling app access to I/O devices on mobile platforms. *CoRR abs/1604.02171*, (2016).

[54] Ongtang, M., McLaughlin, S., Enck, W., and McDaniel, P. Semantically rich application-centric security in Android. *Security and Communication Networks*, (2012).

[55] Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., and Wallach, D. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium* (2011).

[56] Sasse, M. A., Brostoff, S., and Weirich, D. Transforming the 'Weakest Link' — a Human/Computer Interaction Approach to Usable and Effective Security *BT Technology Journal*, (2001).

[57] Arce, I. The weakest link revisited [information security]. In *IEEE Security & Privacy*, (2003).

[58] Fahl, S., Harbach, M., Acar, Y., and Smith, M. On the Ecological Validity of a Password Study In *Ninth Symposium on Usable Privacy and Security*, (2013).

Appendices

Appendix A - Study Demographics: In total, from the 69 recruited subjects that completed our study, 34 (49%) were female; 36 (52%) were in the 18-25 years old range, 27 (39%) in the 26-50 range, and 6 (9%) were in above the 51 range; 33 (48%) were students from our Institution, 9 of them (13%) were undergraduate and 24 (35%) were graduate students, 2 (3%) were Computer Science Majors; 11 (16%) worked in Public Administration, 9 (13%) worked in Hospitality, 6 (9%) in Human Services, 6 (9%) in Manufacturing, and 4 (6%) worked in Science or Engineering. All participants reported being active smartphone users (1-5 hours/day). Also, 42 (61%) of the subjects were long-term Android users (3-5 years), others were long-term iOS users. For our laboratory and field studies, we redistributed the available participants as evenly as possible. Each lab group had 9 long-term Android users, the remaining 6 long-term Android users participated in our field study.

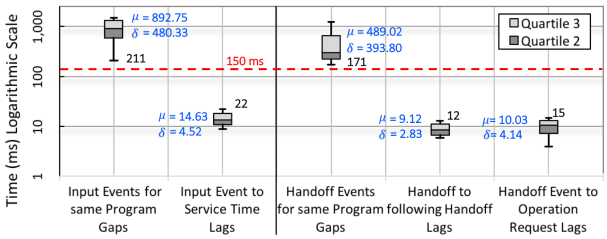

Figure 8: Time analysis used to study the possibility of ambiguous events delegation paths, as discussed in Section 4.2.

Appendix B - Time Constraints Analysis: We leveraged data collected via the field study to perform an analysis of time constraints for input events and action/operation requests to calibrate the time window

for the event ambiguity prevention mechanism (Section 4.2). Figure 8 reports the measurements of the gaps[15] between consecutive input events and consecutive handoff events, as well as the lags between each event and the corresponding response from the serving program. From the measurements, we observed: (1) the minimum gap between subsequent input events targeting the same program (211 ms) is an order of magnitude larger than the maximum lag required by the program to serve each incoming event (22 ms); and (2) the minimum gap (171 ms) between subsequent handoff events targeting the same program is an order of magnitude larger than the maximum lag required by the program to serve incoming requests (15 ms). Hence, to avoid ambiguity, we may set the time window to 150 ms to guarantee that the entire delegation path can be identified before the next event for the same program arrives. Lastly, we observed that 87% of the delegation paths had a total length of three edges (one input event, one handoff event, and one sensor operation request). The remaining 13% of the delegation paths had a maximum length of four edges (one additional handoff event), which further supports our claim that we can hold events without penalizing concurrency of such events.

Appendix C - Program Identification: To prove the programs' identity to users, EnTrust specifies both the programs' name and visual identity mark (e.g., icon) in every *delegation request* as shown in Figure 6. EnTrust retrieves programs' identity by accessing the `AndroidManifest.xml`, which must contain a unique name and a unique identity mark (e.g., icon) for the program package. EnTrust verifies programs' identity via the crypto-checksum[16] of the program's binary signed with the developer's private key and verifiable with the developer's public key [46], similarly to what proposed in prior work [47, 11].

Appendix D - Input Event Authentication: EnTrust leverages SEAndroid [38] to ensure that programs cannot inject input events by directly writing into input device files (i.e., `/dev/input/*`) corresponding to hardware and software input interfaces attached to the mobile platform. Hence, only device drivers can write into input device files and only the Android Input Manager, a trusted system service, can read such device files and dispatch input events to programs. Also, EnTrust leverages the Android screen overlay mechanism to block overlay of graphical user interface components and prevent hijacking of input events. Lastly, EnTrust accepts only voice commands that are processed by the Android System Voice Actions module.[17] EnTrust authenticates input events by leveraging sixteen mediation hooks placed inside the stock Android Input Manager and six mediation hooks placed inside the System Voice Actions module.

Appendix E - Handoff Event Mediation: Programs communicate with each other via Inter-Component Communication (ICC) that, in Android, is implemented as part of the Binder IPC mechanisms. The ICC includes both *intent* and *broadcast* messages that can be exchanged among programs. The Binder and the Activity Manager regulate messages exchanged among programs via the intent API.[18] Programs can also send intents to other programs or services by using the broadcast mechanism that allows sending intents as arguments in broadcast messages. The Activity Manager routes intents to broadcast receivers based on the information contained in the intents and the broadcast receivers that have registered their interest in the first place. To mediate intents and broadcast messages exchanged between programs completely, EnTrust leverages mediation hooks placed inside the Activity Manager and the Binder.

Notice that, other operating systems support mechanisms similar to Android's Intents. For instance, MacOS and iOS adopt the Segue mechanism, while Chrome OS supports Web Intents, thus EnTrust can be also implemented for other modern systems supporting the cooperating program abstraction.

Appendix F - Sensor Operation Mediation: Android uses the Hardware Abstraction Layer (HAL) interface to allow only system services and privileged processes to access system sensors indirectly via a well-defined API exposed by the kernel. Moreover, SEAndroid [38] is used to ensure that only system services can communicate with the HAL at runtime. Any other programs (e.g., apps) must interact with such system services to request execution of operations targeting sensors. EnTrust leverages such a mediation layer to identify operation requests generated by programs, by placing 12 hooks inside the stock Android Audio System, Media Server, Location Services, and Media Projection.

Notes

[1] In this paper, we use the term "delegate" to refer to the use of IPCs to request help in task processing, not the granting of permissions to other processes.

[2] One of the surveillance mobile apps available online (e.g., `flexispy`).

[3] Several banks are now offering these services to their clients.

[4] `https://source.android.com`

[5] The runtime permission mechanism enabled users to revoke permissions at any time.

[6] Source: `https://fortune.com`

[7] `https://dialogflow.com`

[8] `https://source.android.com/compatibility/cts/`

[9] `Android Open Source Project - https://source.android.com`

[10] This range was selected based on the size of the delegation graphs created during our experiments, which should be representative of real scenarios.

[11] `https://developer.android.com/studio/test/monkey.html`

[12] To stress test our system, we selected a lower bound that is considerably lower than the maximum speed at which a user can possibly keep tapping on the screen (~210 ms).

[13] Chosen among the most-downloaded Android apps from the Google Play Store and including all apps and system services shipped with the stock Android OS.

[14] Equivalent of IPCs for Android OS.

[15] Gaps higher than 1,500 ms were excluded because not relevant to the analysis.

[16] Android requires all apps and services to be signed by their developers.

[17] `https://developers.google.com/voice-actions/`

[18] `https://developer.android.com`