
# HTPD: Secure and Flexible Message-Based Communication for Mobile Apps

Yin Liu(✉), Breno Dantas Cruz, and Eli Tilevich

Software Innovations Lab, Virginia Tech, Blacksburg, USA
{yinliu,bdantasc,tilevich}@cs.vt.edu

**Abstract.** In modern mobile message-based communication, malicious apps can illicitly access transferred messages via data leakage attacks. Existing defenses are overly restrictive, as they block all suspicious apps, malicious or not, from receiving messages. As a solution, we present a communication model that allows untrusted-but-not-malicious apps to receive messages. Our model—hidden transmission and polymorphic delivery (HTPD)—transmits sensitive messages in an encrypted envelope and delivers them polymorphically. Depending on the destination's trustworthiness, HTPD delivers either no data, raw data, or encrypted data. *Homomorphic* and *convergent* encryption allows untrusted destinations to securely operate on encrypted data deliveries. We realize HTPD as PoliCC, a plug-in replacement of Android Inter-Component Communication middleware. PoliCC mitigates three classic Android data leakage attacks, and allows untrusted apps to operate on delivered messages. Our evaluation shows that PoliCC enables mobile apps to securely and flexibly exchange communication messages, with low performance and programming effort overheads.

**Keywords:** Mobile security · Message-based communication · Secure inter-component communication

## 1 Introduction

An essential part of modern mobile platforms is inter-app communication[1], which is typically message-based: apps send and receive various kinds of messages, some of which may contain sensitive data. When a malicious app accesses sensitive data, *data leakage* occurs. To prevent data leakage, modern mobile platforms (e.g., Android and iOS) customize their communication models to control how apps access message data. However, these models remain vulnerable to data leakage, commonly exploited by attacks that include interception, eavesdropping, and permission escalation. These attacks leak volumes of sensitive data,

[1] In Android, it is also called inter-component communication (ICC).


J. Garcia-Alfaro et al. (Eds.): SecureComm 2021, LNICST 399, pp. 273–294, 2021.
https://doi.org/10.1007/978-3-030-90022-9_14

as has been documented both in the research literature [11,13,21,34,40] and in vulnerability reporting repositories (e.g., CVE) [2,3,10].

To prevent data leakage, state-of-the-art approaches fall into two general categories: (1) taint message data to track and analyze its data flow [8,20], and (2) track call chains, as guided by a permission restriction policy for sending/receiving data [12,18,22]. Although these approaches[2] strengthen the security of message-based communication, their high false positive rates often render them impractical for realistic communication scenarios. Once any app in a call chain or data flow is identified as "malicious," even as a false positive, they can no longer receive any messages. Although "untrusted" may not be "malicious", these data flow monitoring approaches block all *untrusted-but-not-malicious* destinations. In addition, mobile users may change app permissions at any point, thus also causing false positives. With high false-positive rates, these prior approaches lack flexibility required to secure message-based communication, without blocking untrusted-but-not-malicious destinations from operating on delivered messages.

In this paper, we present HTPD, a novel model that improves the security of message-based communication. The model combines two key mechanisms: (1) hidden transmission of messages and (2) their polymorphic delivery.

Mechanism (1) serializes a message object with additional information (e.g., data integrity or routing information) as an encrypted binary stream, and then hides the resulting stream as the data field of another message used for transmission. Intercepting the transmitted message would not leak its hidden content to interceptors. In the meantime, it cannot be tampered with undetectably either: before delivering the message to a destination, the model retrieves the message's hidden content, using it to verify the message's integrity and destination.

Mechanism (2) steps away from the standard message delivery, in which the delivered message data is presented identically to all destinations, having so-called *monomorphic* semantic. Instead, depending on the destination's trustworthiness at runtime, the delivered message data is presented either in no form, raw form, or encrypted form, thus having *polymorphic* semantic. No data is presented for misrouted messages or when the message's integrity cannot be verified. Raw data is presented to destinations whose trustworthiness can be established. Encrypted data is presented to all other destinations. However, the received encrypted data can still be used in limited computational scenarios, due to homomorphic encryption (HE) and convergent encryption (CE), which preserve certain arithmetic and comparison properties of ciphertext, respectively.

To the best of our knowledge, our approach is the first to apply HE and CE to the design of message-based communication models. Homomorphic and convergent operations on sensitive data provide the middle ground between permitting access to raw data and denying access altogether. The primary barrier to widespread adoption of HE and CE has been their heavy performance overhead. The resulting escalation in execution time has rendered these encryption techniques a poor fit for intensive computational workloads of large statistical

[2] All of them target Android, due to its open-sourced codebase, which can be examined and modified.

analyses and machine learning. In contrast, our work demonstrates that HE and CE can effectively solve long-standing problems in the design of mobile message-based communication. Because mobile communication rarely involves large computational workloads, the inclusion of HE and CE provides the required security and flexibility benefits, without noticeable deterioration in user experience.

To reify our model, we developed PoliCC, an Android middleware[3] that plug-in replaces Android inter-component communication (ICC). PoliCC mitigates interception, eavesdropping, and permission escalation attacks, without preventing untrusted-but-not-malicious apps from operating on delivered message data. Although our design trades performance for security, our evaluation shows that, PoliCC effectively mitigates these attacks while adding at most 40.4 ms and 2 mW overheads, as compared to the Android ICC counterpart.

This paper contributes:

**(1)** HTPD—a novel model that strengthens the security of message-based communication via hidden transmission and polymorphic delivery. This model retains the protection of prior models, but eliminates their unnecessary restrictions, so untrusted-but-not-malicious destinations can perform useful operations on the delivered message data.

**(2)** The first successful application of homomorphic and convergent encryption to the design of mobile message-based communication, offering operations on encrypted sensitive data as the middle ground between permitting access to raw data and denying access altogether.

**(3)** PoliCC—a reification of HTPD that plug-in replaces Android ICC, mitigating interception, eavesdropping, and permission escalation attacks, without preventing untrusted-but-not-malicious apps from operating on delivered data. Through its plug-and-play integration with the Android system, PoliCC requires only minimal changes to existing apps.

**(4)** An experimental evaluation that shows how PoliCC prevents the aforementioned attacks carried out against benchmarks and real apps, while incurring low performance and programming effort overheads.

## 2 Threat Model

By following our model, message-based communication can prevent data leakage, so it would not be exploited by *interception*, *eavesdropping*, and *permission*

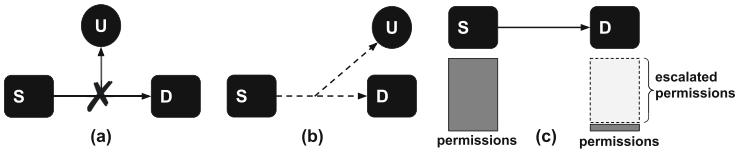


**Fig. 1.** Examples of attacks

[3] Similarly to prior works, we target Android as the dominant open-source platform.

*escalation* attacks. Our model can strengthen any message-based communication, but our reference implementation is Android-specific. We generally define each of the aforementioned attacks, and present examples of their real-world occurrences in Android apps.

### 2.1 Examples of Data Leakage Attacks

**Interception:** Figure 1-a demonstrates an interception attack: source S is transferring data to destination D, with U intercepting the transferred data. Another common name of this attack is *man-in-the-middle*.

In Android, apps can communicate with each other via inter-component communication (ICC). Using ICC, a source app sends an Intent[4] to user-permitted destination apps: with *explicit* Intents, only specific destinations can receive data; with *implicit* Intents, any destination that registers a certain Intent Filter[5] can receive data. As defined in Common Attack Pattern Enumeration and Classification (CAPEC) [1], an interception attack occurs when malicious apps inappropriately receive an implicit Intent by declaring a certain Intent Filter [5]. These attacks have been detected in many real-world Android apps [16], which even be used in some developer tools for benign purposes. For example, a developer tool, `Intent Intercept`, intercepts the transferred Intent to help developers in debugging ICC-based communication [4].

**Eavesdropping:** Figure 1-b demonstrates an eavesdropping attack: source S broadcasts data to destination D, but U (i.e., eavesdropper) can also receive the data. In Android, when an app broadcasts Intents, any app can receive them by declaring a certain Intent Filter. Consider a real eavesdropping attack reported in CVE, when WiFi is switched, the Android system broadcasts an Intent that contains detailed WiFi network information (e.g., network name, BSSID, IP, and DNS server). However, having declared the corresponding Intent Filter, any applications can receive this Intent and disclose the sensitive information[3].

**Permission Escalation:** Figure 1-c demonstrates a permission escalation attack: source S has been granted sufficient permissions to access sensitive data, but destination D has not. When S sends its sensitive data to D, D's permission is escalated. In Android, to access sensitive user data (e.g., GPS, contacts, and SMS), apps must secure the required permissions. As previously reported, attackers can force apps, with dissimilar permissions, to communicate sensitive data to other apps, thus leaking it to the destinations [10,36]. For example, if an app has GPS permissions, it can send its obtained user geolocation information to any app that has no such permissions, which may cause sensitive data leakage.

### 2.2 Untrusted Data Processing

The aforementioned attacks that exploit data leakage share the same root cause: a destination illicitly accesses and discloses or tampers with sensitive mes-

[4] In ICC, Intent objects serve as data delivery vehicles.
[5] Intent Filter declares expected Intent properties (action/category).

sage data. However, blocking all suspicious message transmissions may paralyze apps' legitimate operations. Consider the scenario from the eavesdropping attack above: if, having received a message containing the device's IP address, an untrusted app uses the received IP only for legitimate operations (e.g., host IP verification), is it reasonable to block all such message transmissions to strengthen security?

A more flexible solution could use homomorphic encryption (HE) and convergent encryption (CE), currently most commonly used for sending sensitive data to the cloud for processing by untrusted providers. Using HE/CE schemes, data owners encrypt and send their data to the cloud server. The cloud server operates on and returns the encrypted results to the data owner. Only the data owner, possessing the secret key, can decrypt the results. In the case above, an untrusted app can still receive the IP address's encrypted version to verify its host address, without accessing the raw IP address. By means of HE/CE, HTPD enables untrusted apps to operate on sensitive data without data leakage.

### 2.3 Assumptions and Scope

To counteract the threats, our design is subject to the following constraints:

**(1) Assumptions: Trustworthiness.** Since HTPD relies on apps' trustworthiness to determine whether to expose no data, raw data, or encrypted data, we assume that application trustworthiness can be reliably configured or calculated. Further, attackers cannot change the involved apps' trustworthiness. As an abstract metric of how to expose the data, the trustworthiness can be represented as many specific forms, such as permissions in the above example attacks (Sect. 3.1).

**(2) Scope: Message-based data leakage vulnerabilities.** HTPD's focus is message-based data leakage vulnerabilities. That is, the vulnerabilities should (a) cause data leakage, and (b) occur during message transmission. Hence, other attacks, such as denial of service (DoS)[6] that target data transmission (not data leakage), stealing data by breaking the system (not during message transmission), are out of scope. Also, since attacking key management to obtain decryption keys is an orthogonal issue, we consider it out of our scope. To mitigate this issue, HTPD should be used with proven key management schemes/systems.

## 3 The HTPD Model

We present the HTPD model and its application to Android ICC in turn next.

[6] Although DoS is not our focus, one of PoliCC's features mitigates them (see Sect. 6.4).

### 3.1 Definitions

(1) *Source/Destination.* In message-based communication, a *source* sends messages, and a *destination* receives messages. In mobile platforms, apps can be both source and destination.

(2) *Sending and Receiving Points.* We use the term *a sending point* to describe an API function, invoked by a *source* and passed message data, that starts transmitting messages. *A receiving point* is a callback API function, through which a *destination* retrieves the transferred message data.

(3) *Trustworthiness.* Trustworthiness measures the degree to which an app can be trusted. We use this metric to define how an app can access message data. An app whose trustworthiness is established can access raw message data; otherwise, encrypted data or no data. Trustworthiness can be measured in different ways: **(a)** data integrity (i.e., to detect data tampering) and destination examinations (i.e., to detect misrouting). For example, if the received message fails such examinations, the destination app should not access the raw data. **(b)** apps' permissions and the relationship between apps' permission sets. For example, if a source app's permission set is larger than that of a destination app, messages transmitted between them may become vulnerable to permission escalation attacks, causing data leakage (Sect. 2.1). **(c)** reputation score. For example, if an app's reputation score in app markets [26] is low, then allowing it to access raw data may cause data leakage. HTPD *can be parameterized with various measures of trustworthiness, as required for a given scenario of message-based communication. In particular, to determine an app's trustworthiness, our* HTPD*'s reification uses both (a) and (b).*

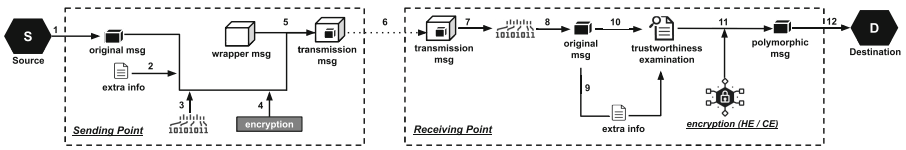


**Fig. 2.** HTPD transmission mechanisms

### 3.2 Transmission Mechanisms

Figure 2 depicts HTPD's hidden transmission and polymorphic delivery mechanisms. When a source starts transmitting a message (step 1), the message's data field is inserted with some extra information (e.g., custom routing) (step 2). After that, the message is serialized (step 3) and encrypted (step 4) to a binary stream, which becomes the data field of a newly created wrapper message. Hence, the original message is hidden within the wrapper message, which becomes *the transmission message* (step 5).

Next, the transmission message is dispatched via the system's standard communication channel (step 6). Once the transmission message arrives to its receiving point, its data field is extracted, decrypted, and deserialized into the original message (steps 7, 8). Then, the extracted extra information is used to examine the destination's trustworthiness (e.g., data integrity, app permissions), which determines in which form the message data can be accessed. In the cases of failed integrity checks or misrouted deliveries, HTPD would not disclose any data. If the destination's trustworthiness is established, HTPD reveals the raw form of the original message's data; otherwise, it encrypts the data into its homomorphically or convergently encrypted form (step 11). Due to its polymorphic delivery, the final received message is referred to as "polymorphic message" (step 12).

### 3.3 HTPD in Practice

To reify HTPD, we developed PoliCC, a plug-in replacement of Android ICC. By mitigating Android ICC's data leakage vulnerabilities, PoliCC prevents the aforementioned attacks (Sect. 2.1). Following the definitions above, apps serve as message source/destination, whose sending/receiving points are managed by PoliCC (in Fig. 3). PoliCC retains *Intent objects* as ICC transmission vehicles, but hides the original Intent object within a so-called *Host Intent* object, whose data field stores the original Intent object's encrypted serialized version.

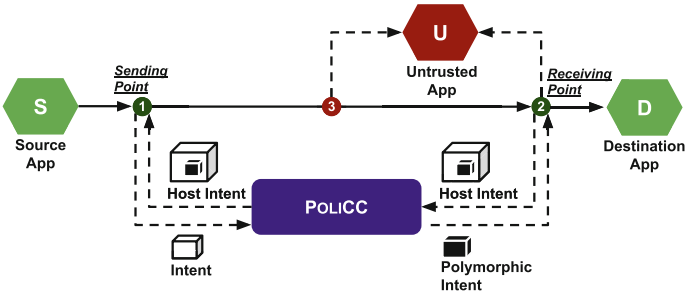


**Fig. 3.** PoliCC solution overview

To guide its polymorphic delivery, PoliCC computes the destination app's trustworthiness: (a) the delivered message's routing information[7], and (b) how the permission sets of the source and destination apps relate to each other. PoliCC delivers no data from messages identified as tampered with or misrouted; it delivers raw data to destinations whose permission sets are equal to or exceed those of source apps; and it delivers homomorphically or convergently encrypted data to all other destinations.

[7] In Android ICC, routing information can be used for both data integrity and destination examinations (detailed in Sect. 5.2).

Consider how PoliCC prevents the attacks described in Sect. 2.1: **Preventing Interception Attacks.** As shown in Fig. 3, an interception attack can be carried out at the receiving point ❷: an untrusted app U becomes the final delivery destination by declaring a certain Intent Filter, or at point ❸ before the receiving point: with Android ICC, U intercepts the transferred data.

In the first case, having received the transferred Intent, U would be able to retrieve the contained raw data only if U's permission set equals to or exceeds that of the source app. Otherwise, the received data would be homomorphically or convergently encrypted, so it would not be leaked. In the second case, U would receive a Host Intent, containing only the encrypted and thus inaccessible original Intent. At this point, tampering with the Host Intent's routing information would be easily detected through data integrity and destination examinations at ❷.

**Preventing Eavesdropping Attacks.** As discussed in Sect. 2.1, an eavesdropping attack occurs when the Android system broadcasts an IP address (i.e., string value), received by both a trusted app D and an untrusted app U. Without sufficient permissions, U would receive the IP address as a convergently[8] encrypted string. Since convergent encryption makes it possible to compare encrypted values for equality, U can verify the host address by convergently encrypting the host address and comparing the result with the received data.

**Preventing Permission Escalation Attacks.** As discussed in Sect. 2.1, an escalation attack occurs when a source app with GPS permissions obtains and sends user geolocation information to a destination without these permissions. Without the geolocation permissions, destination D would be delivered geolocation (i.e., numeric value) as a homomorphically[9] encrypted numeric value, so no permissions would be escalated. If D forwards the delivered ciphertext to malware M to decrypt the ciphertext, M's permission set would have to be equal or greater than the union of permission sets of the source app and D, a hard-to-satisfy requirement for the granted permissions to access raw geolocations.

## 4 PoliCC Design

We next explain PoliCC's design and then describe its architecture and permission policies.

### 4.1 Design Choices

PoliCC follows several design choices that we made by consulting prior studies.

**(1) Why Intents?.** As its message delivery vehicle, PoliCC retains Intents to facilitate integration with existing apps, protecting their Intents from leakage. *Prior Finding: Apps commonly store data in Intent's extended data field* [38].

[8] Convergent encryption is applied to string data.
[9] Homomorphic encryption is applied to numeric data.

**(2) Why Focus on Integer and String?.** PoliCC supports operating on encrypted strings and integer values. *Prior Finding:Android API methods that manipulate Integer and String values (i.e., `putString` and `putInt`) are among the top 10 mostly used* [32].

**(3) Why Activity and Broadcast Communication?.** PoliCC supports `startActivity`. *Prior Finding: The `startActivity` ICC method is the most frequently used* [41]. Further, to demonstrate HTPD's applicability, PoliCC also supports `sendBroadcast`, whose data flow allows multiple destinations.

### 4.2 System Architecture

Figure 4 shows PoliCC's architecture. Via the Android ICC, a `Source App` sends a regular Intent object (step 1). The sending flow is redirected via an Xposed hook that forwards (step 2) the sent Intent object to PoliCC `Module`, which initiates a protection procedure (i.e., the sending point). The forwarded Intent is inserted with the source app's package name as another extended data field (i.e., its transmission history is recorded for permission examinations Sect. 5.2), and is then serialized and encrypted (via AES) into a byte array, with the result placed in a Host Intent object, thereby concealing the original Intent. It is the Host Intent object that continues its transmission through the Android ICC (step 3).

Right before delivering the Host Intent (i.e., the receiving point), PoliCC's another Xposed hook redirects (step 4) the Intent to PoliCC `Module`, which extracts the original Intent data and reconstructs the original Intent (`Re-encapsulation`). PoliCC then collects the routing information and the permission sets for both source and destination apps, passing them to the `Routing info & Permission Examination`. The routing information determines, via `Routing info Examination`, whether to return no data. The relationship between the permissions of the source and destination apps determines, via `Permission Examination`, whether to homomorphically or convergently encrypt the Intent's data. Finally, PoliCC delivers the `Polymorphic Intent` object to the `Destination App` (step 5). There is no separate `Polymorphic Intent` type. Rather, these entities are normal Intent objects, whose content has been protected.

User-configured (via `Configuration`) custom keys encrypt the decryption keys of homomorphic/convergent encryption, persisting them in Android private storage [27]. Note that, by relying on Android's private storage, we treat *key management* as orthogonal to our design, which can straightforwardly integrate more elaborate key management schemes as system services. In addition, by configuring PoliCC to notify of the transmission information (e.g., source/destination, permissions, data types, actions) (`Notify`), the user can stop the delivery of any PoliCC Intents.

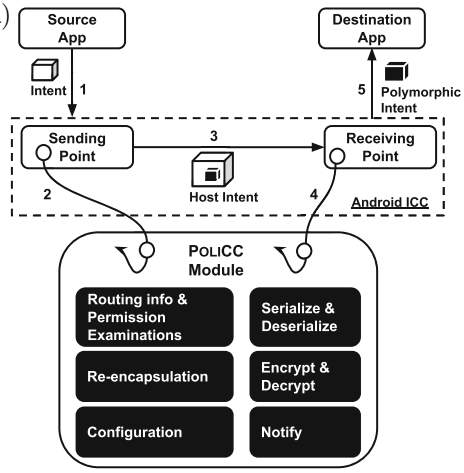


**Fig. 4.** PoliCC Architecture

With the design choices above, consider how our architecture secures the Android ICC while enabling untrusted data processes. To transmit messages securely, PoliCC provides Host and Polymorphic Intents for ICC's data transmission flow of both Activity and Broadcast. The Host Intent acts as a transmission vehicle over the Android ICC; it hides the original Intent's data and routing information. The Polymorphic Intent delivers data polymorphically: only destination apps with sufficient permissions can access raw Intent data. To allow untrusted apps to operate on sensitive data securely, PoliCC provides arithmetic and comparison operations on ciphertext, enabled by homomorphic and convergent encryption: *Variant of Elgamal* encryption[10] [17] for `int`/`Integer`/`BigInteger` values and convergent encryption (combines SHA256 with AES) for `String` values. So `int`/`Integer`/`BigInteger` variables become `HEInteger` objects, and `String` ones become encrypted `String` objects.

### 4.3 Permission Policies

As discussed in Sect. 3.3, PoliCC uses Android app permissions and the relationship between the permission sets of the source/destination apps as the trustworthiness measure that determines whether to deliver raw data or encrypted data[11]. Hence, we design PoliCC as a policy-based middleware: an extensible set of policies governs data access and Intent routing. $\rightarrow$ indicates the *From-To* Intent transmission relationship within the device. E.g., $\{I \mid (S \rightarrow D)_t\}$ indicates that the source app $S$ sends the Intent $I$ to the destination app $D$ at time $t$.

[10] As fully homomorphic encryption is slow, its partial variant achieves a practical performance security tradeoff.

[11] Because the "no data" delivery is caused by failed data integrity checks rather than permissions, we detail it in Sect. 5.2.

$P(S)_t$ denotes the permission set of app $S$ at time $t$. If the user changes $P(S)$ at runtime, $P(S)_t \neq P(S)_{t+1}$. Hence, PoliCC always dynamically analyzes permissions, reading the latest permissions for all apps. Further, $I$ denotes the original Intent, and $I_{EN}$ denotes that its data has been encrypted. We define the PoliCC policies as follows:

**(1) Encryption & Decryption Policies.**
When $D$ receives $I$ (or $I_{EN}$) from $S$, if the permission set of $S$ is a subset of or equal to that of $D$, $I$'s (or $I_{EN}$'s) data remains unencrypted; encrypted otherwise:
If $\{I$ or $I_{EN} \mid (S \rightarrow D)_t\}$,

- iff $P(S)_t \subseteq P(D)_t$, return $I$
- otherwise, return $I_{EN}$

**(2) Permission Transitivity.** Whether a destination app receives $I$ or $I_{EN}$ is determined by the transitive closure of the permission relationships between the encountered apps in that Intent's transmission chain:
If $\{I \mid ((S \rightarrow D1)_t \rightarrow D2)_{t+1}\}$,

- iff $(P(S)_{t+1} \cup P(D1)_{t+1}) \subseteq P(D2)_{t+1}$, return $I$
- otherwise, return $I_{EN}$

## 5 Implementation

We describe PoliCC's hidden transmission and polymorphic delivery.

### 5.1 Hidden Transmission

To seamlessly integrate hidden transmission into Android ICC, we had to determine: (1) where to place the sending/receiving points, and (2) how to pack message data into its delivery vehicle.

**(1) Hook Mechanism.** For PoliCC to take control over the delivery of Intent objects, the hook mechanism taps into the Android ICC. ICC commences by invoking the standard Android APIs, `startActivity` to start an activity and `sendBroadcast` to send a broadcast, so we use them as "sending points." Similarly, ICC ends up the final delivery by invoking `performLaunchActivity` for the activity and `deliverToRegisteredReceiverLocked` for the broadcast, so we use them as "receiving points." PoliCC intercepts the sending and receiving points by hooking into these API methods. Then, PoliCC's custom code is injected to execute before or after the intercepted API methods, thus performing HTPD's transmission strategies.

**(2) Host Intent.** A Host Intent is derived from an original Intent by retaining the routing information (e.g., action, category) but removing the extended data (i.e., the data inserted via `putExtra`). Instead, the only pieces of extended data in Host Intent are serialized and encrypted representations of the original Intent. This implementation strategy is non-intrusive, thus requiring no changes to the source app's Intent API. Specifically, our implementation intercepts the built-in Intent transmission procedure at the points right before an Intent is dispatched (i.e., sending points) and delivered (i.e., receiving points). At the sending point, a Host Intent is constructed, replacing the original Intent; at the receiving point, the Host Intent's content is extracted, decrypted, and deserialized into the original Intent, which is then polymorphically delivered to the destination app (see Sect. 5.2). Notice that this strategy makes it possible to transmit Host Intents through the built-in Intent transmission channels. Because these two interception points cannot be bypassed, the Host Intents would always be constructed at the sending point, and the original Intent would always be reconstructed at the receiving point. In essence, PoliCC can straightforwardly detect any tampering with the routing information of a Host Intent.

### 5.2 Polymorphic Delivery

To seamlessly integrate polymorphic delivery into Android ICC, we had to determine: (1) how to link trustworthiness (i.e., routing info and permission relationships) to delivery strategies (i.e., no data, raw data, or encrypted data), and (2) if it is possible to bypass our secure delivery mechanism and how to defend against it. We solve these problems as follows.

**(1) Examining Routing Info and Permissions.** As described above, in the sending point, PoliCC re-encapsulates the original Intent object, retaining its routing information, and inserts the source app's information, which is checked as follows:

*a) to check the routing information*, having intercepted the Intent in the receiving point, PoliCC extracts the routing information from both the Host and original Intents and compares them for equality (i.e., integrity check). Then, it checks whether the current destination is reachable through the original Intent's routing information (i.e., destination examination). If any of these checks fails, the Intent object may have been tampered with, causing PoliCC to deliver no data to the destination app.

*b) to check the permission relationships between the source and destination apps*, from the original Intent, PoliCC extracts the inserted source app information. Note that, before sending or forwarding an Intent (i.e., the sending point), PoliCC appends the current source app's package name into the Intent's data field, thus keeping track of the Intent's transmission history. At the receiving point, PoliCC computes the union of the permissions granted to all source apps, through which the Intent has passed

in that transmission. Next, PoliCC obtains the destination app's permissions via the Android API. The results are compared based on the *permission transitivity policy* (see Sect. 4.3): if the destination app's permissions are not equal to or exceed the union of the source permissions set, PoliCC delivers the homomorphically/convergently encrypted data to the destination app.

**(2) Defense against *Encryption Bypassing* Attack.** When forwarding a polymorphic Intent with encrypted data to a sufficiently permitted destination app (see policies in Sect. 4.3), PoliCC decrypts the contained data. To that end, a special field, `isEncrypted`, reflects whether the extended data of a Polymorphic Intent has been encrypted, so as to prevent the encryption of ciphertext. However, malware can attempt to bypass the encryption process by maliciously setting the `isEncrypted` field of an unencrypted Intent to "true," an occurrence that we call an *encryption bypassing attack*.

To defend against this attack, PoliCC provides a simple but effective defense: when the `isEncrypted` field is set to "true", PoliCC first decrypts the data and then encrypts it again. However, decrypting unencrypted data produces an unusable value, out of which the original Intent data cannot be recovered[12]. Hence, this design effectively defends against the attacks that tamper with the `isEncrypted` field, albeit rendering the transferred data unusable as a result of invalid data attacks. Our design contends with the possibility of Intent data becoming damaged in such cases, as the main objective is to defend against data leakage attacks.

### 5.3 Computing with Encrypted Data

As discussed above, Intent's `String` data become convergently encrypted `String` objects, and `int`/`Integer`/`BigInteger` data become homomorphically encrypted `HEInteger` objects in order to allow untrusted-but-not-malicious apps to operate on ciphertext.

## 6 Evaluation

we seek to answer the following questions: **Q1. Effectiveness**: How effectively does PoliCC reduce the threats? **Q2. Cost**: What is the performance overhead of PoliCC on top of the Android ICC? **Q3. Effort:** How much additional programming effort is required to use PoliCC instead of Android ICC?

### 6.1 Environment Setup

Because PoliCC is implemented on top of the Xposed framework, our evaluation uses this framework's latest version (XposedBridge version-82 and Xposed

[12] With PoliCC's encryption implementation, decrypting unencypted data destroys the original data, which may not be the case for other encryption implementations.

Installer-3.1.5). Besides, to make use of as many Android latest features as possible, while guaranteeing the compatibility of Android apps, we use the Android Nougat (7.x) and Lollipop (5.x), currently run by 28.2% (the first highest percentage among the 8 most popular Android versions [25]) and 17.9% (the fourth highest percentage) of Android devices. These Android releases as well as the latest one are vulnerable to all the aforementioned attacks. In all experiments, the devices are: Nexus 6 with Android 7.1.1, Moto X with Android 5.1, and Moto G2 with Android 5.1.1.

### 6.2 Evaluation Design

**Q1. Effectiveness.** As discussed in Sect. 2, PoliCC plug-in replaces ICC to secure its message-based communication against interception, eavesdropping, and permission escalation attacks. To evaluate PoliCC's security mechanisms, we simulated the attacks, discussed in Sect. 2.1, and conducted a case study with three real-world apps, discussed in turn next:

***(1) Reproducing Attacks.*** To test how effectively PoliCC defends apps against the attacks described in Sect. 2.1, we had to reproduce these attacks with real apps. Unfortunately, several of the apps, mentioned in the CVE entries describing the attacks, are not open-sourced, while the target attacks would be impossible to trigger in a black-box fashion. Hence, we had to recreate the described apps on our own.

*(a)* *To reproduce the interception attack on message-based communication at the receiving point*, we created: source app `S`, destination app `D`, and interceptor app `U`. `S` invokes `startActivity(Intent)` to send an implicit Intent to `D`. However, by registering the same `Intent Filter`, the untrusted app `U` receives the same Intent object as well. *To reproduce the interception attack on message-based communication before the receiving point*, we implemented another interceptor app `U2`, which intercepts the sent Intent objects before they arrive at the receiving point, to misroute their delivery by tampering with their routing information. Without loss of generality, we assumed that the end-user had designated the destination apps as our untrusted apps `U` and `U2`.

*(b)* *To reproduce the eavesdropping attack*, we created source and destination apps (i.e., `S` and `D`), with the same system-level permissions, and `IP Verification` app `V` with no such permissions. `S` invokes `sendBroadcast(Intent)` to broadcast an Intent to be received by `D`. The Intent object contains an IP address. However, by registering the same `Intent Filter`, `V` can receive the same Intent object as well.

*(c)* *To reproduce the permission escalation attack*, we created `GC`, a `geolocation collecting` app, permitted to obtain geolocations from the GPS sensor, and `DE`, a `distance estimating` app, forbidden to read geolocations. To process the obtained geolocations, `GC` invokes `startActivity(Intent)` to directly send an explicit Intent to `DE`. `DE` receives the Intent, retrieves the contained geolocation, and estimates the distance of user movement.

We simulated the above attacks using Android ICC and PoliCC, and then compared the respective outcomes. To determine whether PoliCC's polymorphic delivery correctly responds to changes in app permissions, we carried out each attack scenario with sufficient and insufficient permissions. More importantly, to illustrate that apps with insufficient permissions can still execute useful operations, we reused the `IP Verification V` and `distance estimater` (`DE`) apps from the attack scenarios (b)(c) above to check if their original operations (i.e., verify host IP–`V`, estimate distance of movement–`DE`) can still be executed.

*(2) **A case study with real-world apps.*** We also evaluated how effective PoliCC was at mitigating the aforementioned attacks in three open-source, real-world apps: `Intent Intercept` [4] (a debugging app), `Mylocation` [23] (a GPS app), and `QKSMS` [37] (a messaging app). By registering numerous Intent Filters, `Intent Intercept` intercepts implicit Intents and examines their data fields. Having the geolocation permissions (`ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION`), `Mylocation` can obtain the user's geolocation and share it with other apps via an implicit Intent with the `ACTION_SEND` action. Using its `Intent Filter` for the `ACTION_SEND` action, `QKSMS` can receive the Intents containing this action. However, `QKSMS` has no geolocation permissions. In our case studies, we always used `Mylocation` as the source and `QKSMS` as the destination.

**Q2. Cost.** To determine whether PoliCC's performance overhead is acceptable, we compare the respective execution time and energy consumption[13] taken to deliver Intent data from the source to the destination app by PoliCC and the Android ICC. Our measurements (a) exclude prompting the user to approve the Intent transmission; (b) fix the length of intent data items (32 bytes for the `String` objects); (c) repeat all executions 20 times and then compute the average execution time; (d) trigger `startActivity`/`startBroadcast` 100 times in 5 minutes, measuring the amount of energy consumed by the participating apps and the system; and (e) fix the experimental device (i.e., Moto G2) to compare PoliCC with the Android ICC. Besides, we isolate the time PoliCC takes to deliver Intents to identify the performance bottlenecks.

**Q3. Effort.** To confirm PoliCC's portability, we test it on combinations of devices that run the Lolipop and Nougat Android framework versions. To estimate PoliCC's programming effort, we measure the uncommented lines of code (ULOC) required to modify the original source app's ICC code that sends an Intent to a destination app to (a) access the Intent data, and (b) retrieve and use homomorphically/convergently encrypted data.

[13] We measure energy consumption with PowerTutor 1.4 [35].

### 6.3 Results

**Q1. Effectiveness.**

**Table 1.** Effectiveness of PoliCC

| Attacks | Permission | Data retrieved | | Successful defense | | Operations | |
|---|---|---|---|---|---|---|---|
| | | ICC | **PoliCC** | ICC | **PoliCC** | ICC | **PoliCC** |
| Interception —at receiving point | insufficient | raw | **encrypted** | × | ✓ | - | - |
| | sufficient | raw | **raw** | × | × | - | - |
| Interception —before receiving point | - | raw | **no data** | × | ✓ | - | - |
| Eavesdropping | insufficient | raw | **encrypted** | × | ✓ | ✓ | ✓ |
| | sufficient | raw | **raw** | × | × | ✓ | ✓ |
| Permission Escalation | insufficient | raw | **encrypted** | × | ✓ | ✓ | ✓ |
| | sufficient | raw | **raw** | × | × | ✓ | ✓ |

***(1) Reproducing Attacks.*** Table 1 summarizes the outcomes of reproducing each of the attacks: *(a) For the interception attack on message-based communication at the receiving point* (i.e., row "Interception—at receiving point"), whether the interceptor app `U`'s permissions are sufficient or not, Android ICC always delivers the Intent's raw data to `U`, thus leaking sensitive data to an untrusted party. In contrast, if `U`'s permission set is smaller than that of the source app (i.e., insufficient permissions), PoliCC delivers encrypted Intent data, thus successfully preventing the attack. *For the interception attack on message-based communication before the receiving point* (i.e., row "Interception—before receiving point"), after `U2` tampers with the Host Intent's routing information, its examination fails causing PoliCC to deliver no data to the destination app, thus repelling the attack.

*(b) For the eavesdropping attack*, whether the `IP Verifi- cation V`'s permissions are sufficient or not, the Android ICC always delivers a raw IP address, thus leaking the data to `V`. In contrast, when `V`'s permission set is smaller than that of the source app, PoliCC delivers convergently encrypted IP address. Although `V` cannot access the raw data, it can still validate the host IP using the received encrypted IP address (column "Operations").

*(c) For the permission escalation attack*, similar to the attacks above, the Android ICC always delivers an explicit Intent with a raw geolocation to the `distance estimater` (`DE`), so the attack succeeds in exfiltrating the sensitive geolocation. In contrast, PoliCC Intent data's encryption status is determined by the source/destination permission relationship. When `DE` has insufficient permissions, PoliCC delivers homomorphically encrypted longitude and latitude values, so their raw values are not leaked. More importantly, `DE` can still perform its distance estimation operation to approximate the distance by computing with the encrypted values.

In summary, the Android ICC leaves the data vulnerable to all three attacks, while PoliCC prevents these attacks and still preserves the ability of untrusted destination apps to operate on the received encrypted message data.

***(2) Case study with real-world apps.***
*Case 1 (interception):* (a) `QKSMS` acts as the malicious app that intercepts the implicit Intents sent by `Mylocation`. In the original setup, `QKSMS` always obtains the raw geolocation value. With PoliCC, since `QKSMS` lacks the geolocation permissions, it obtains only a homomorphically encrypted geolocation. With the geolocation permissions added to `QKSMS`'s manifest file, it is the end-user who determines the app's data access by granting or declining the geolocation permissions, so `QKSMS` obtains the raw or encrypted geolocation values, respectively.

(b) `Intent Intercept` acts as the malicious app that intercepts the implicit Intents. The app is configured to always obtain the implicit Intents sent by `Mylocation`. However, as it lacks GPS permissions, `Intent Intercept` can only access geolocation data that is homomorphically encrypted, so the raw geolocations are never leaked.

*Case 2 (eavesdropping):* To execute an eavesdropping attack, `Mylocation` sends the same Intent as in Case 1 via an added `sendBroadcast`. `QKSMS` receives this Intent via an added broadcast receiver, registered for the `ACTION_SEND` action. In the original setup, `QKSMS` always obtains the raw geolocation, irrespective of whether the end-user grants/declines the geolocation permissions. With PoliCC, it is the end-user who determines the app's data access by granting or declining the geolocation permissions, so `QKSMS` obtains the raw or encrypted geolocation values, respectively.

*Case 3 (permission escalation):* To execute a permission escalation attack, `Mylocation` creates an explicit Intent containing a geolocation, sending it to an added `Activity` in `QKSMS`. In the original setup, this permission escalation attack always succeeds. With PoliCC, the attack always fails, as long as `QKSMS` has no geolocation permissions.

**Q2. Cost.** Table 2 (at the top) shows PoliCC's overheads. Specifically, PoliCC's `startActivity` increases $T$ by 28.3 ms, $E_{app}$ by 0.8J, and $\Delta E_{sys}$ by 1mW; `send- Broadcast` increases $T$ by 40.4 ms, $E_{app}$ by 0.5J, $\Delta E_{sys}$ by 2 mW, as compared to the Android ICC counterparts. Table 2 (at the bottom) breaks down the execution time per each PoliCC procedure. In both `startActivity` and `sendBroadcast`, the Sending/Receiving Points perform similarly: these procedures' hook and re-encapsulation mechanisms are fixed for all operations.

Since PoliCC increases $T$ by 40.4 ms (43.2 - 2.8 in `sendBroadcast` column) at most, its *execution time overheads* are in line with other related solutions (e.g., [29]'s performance overhead is ≈39 ms), with the total latency much lower than the Android response time limit (5000 ms [24]). Also, since PoliCC increases $E_{app}$ by 0.8J (8.7 - 7.9 in `sendActivity` column) and $\Delta E_{sys}$ by 2 mW (20 - 18) at most, its *energy consumption overheads* are negligible. It is PoliCC's protection mechanisms (i.e., re-encapsulation, encryption/decryption) that incur these performance and energy overheads.

**Table 2.** PoliCC's Overheads (milliseconds–ms, Joules–J, milliwatt–mW)

| ICCs | startActivity ($T$ / $E_{app}$ / $\Delta E_{sys}$)* | sendBroadcast ($T$ / $E_{app}$ / $\Delta E_{sys}$)* |
|---|---|---|
| Android ICC | 57.0 ms / 7.9 J / 37 mW↑ | 2.8 ms / 5.3 J / 18 mW↑ |
| PoliCC | 85.3 ms / 8.7 J / 38 mW↑ | 43.2 ms / 5.8 J / 20 mW↑ |

| Operations | Sending Point | Receiving Point |
|---|---|---|
| startActivity | 28.2 ms | 13.5 ms |
| sendBroadcast | 28.8 ms | 9.5 ms |

* $T$: execution time (ms); $E_{app}$: energy consumed by source/destination apps (J); $\Delta E_{sys}$: additional system energy consumed by ICCs (mW).

**Q3. Effort.** We first confirm PoliCC's portability by testing its operations on three Android devices/versions: Nexus 6/Android 7.1.1, Moto X/Android 5.1, and Moto G2/Android 5.1.1 by running our subject apps on these devices in different combinations. This test has not revealed any deployment and operational issues. For source apps, the PoliCC API is indistinguishable from that of the Android ICC as well as for sufficiently permitted destination apps, as the delivered Polymorphic Intents return raw data. With insufficient permissions, additional code is required in destination apps to handle the delivered homomorphically/convergently encrypted data.

Nevertheless, the extra programming effort is small, as Table 3 demonstrates: in the source apps, PoliCC requires no deviation from the familiar Android ICC API (column "Send"). In the destination apps, the code for retrieving, using, and creating `int`/`Integer`/`BigInteger` and `String` objects require extra lines of code (columns "Retrieve", "Use", and "Create"): (a) For `BigInteger` and `String` objects (columns "BigInt." and "Str."), due to the inheritance hierarchies of their operations, the code for retrieving them is indistinguishable between the Android ICC and PoliCC (column "Retrieve"). To use the retrieved `String` object, 1 extra LOC is required to separate its convergently encrypted value. To use the retrieved `BigInteger` object, 1 extra LOC is required to check whether the data is homomorphically encrypted (column "Use"). (b) For `int`/`Integer` values (columns "int"/"Int."), their original receiving and operating methods are replaced with `HEInteger`'s methods (`add`, `subtract`, and `multiply`), taking 4 extra LOCs at most (columns "Retrieve" and "Use"). Finally, it takes 3 extra LOC to create homomorphically and convergently encrypted `int`/`Integer`/`BigInteger` and `String` values (column "Create").

**Table 3.** PoliCC's Extra Prog. Effort (ULOC)

| Send | Retrieve | | Use | | Create | |
|---|---|---|---|---|---|---|
| | int/Int./BigInt | Str. | int/Int./BigInt | Str | int/Int./BigInt | Str. |
| 0 | 1/1/0 | 0 | 4/4/1 | 1 | 3/3/3 | 3 |

### 6.4 Discussion

PoliCC's HTPD implementation may suffer from false positives/negatives if app permissions are granted incorrectly. This limitation can be mitigated by notifying users of potential security attacks. As an extra feature (Sect. 4.2), PoliCC's `Notify` module can be configured to report the transmission information (e.g., source/destination, permissions, data types, actions) to the user, who can then stop the delivery of any PoliCC Intents. Further, the `Notify` module can also mitigate the denial of service attacks: it can detect and block notifications floods from any source app. PoliCC performs Hook mechanism via Xposed. Note that we use Xposed to create a viable proof of concept to be able to evaluate PoliCC's security-enhancing properties. To commercially deploy HTPD, one can fully integrate it and its mechanisms with any Android release and other mobile platforms, despite the peculiarities of our reference implementation.

## 7 Related Work

**Data Flow & ICC Calls Monitoring.** Most of the existing solutions counteract data leakage attacks by monitoring the data flow or ICC calls. TaintDroid [20] traces data flows by labeling sensitive data and transitively applying labels as the data propagates through program variables, files, and interprocess messages. If any tainted data is to leave the system via a sink (e.g., network interface), the system notifies the user about the coming data leakage. FlowDroid [8] applies static taint analysis to check if any app lifecycle contains data leaks. XManDroid [12] tracks and analyzes the ICC data transferred in Intent objects at runtime to enforce the app's compliance with the defined permission policy. QUIRE [18] enables users to examine and terminate the chain of requests associated with an ICC call. ComDroid [16] statically analyzes *.dex binaries of Android apps to log potential component vulnerabilities. Besides, ComDroid tracks how an Intent object changes moving from its source to destination. Other state-of-the-art Intent vulnerabilities detectors (e.g., IccTa [31], DINA [6], DroidRA [33], SEALANT [30], IntentScope [28]) further improve the above methods for monitoring data flows & ICC calls. However, their reliance on overly restrictive policies prevents them from supporting Android application-specific data flows, also causing false-positives when data flows change unpredictably. In contrast, HTPD model systematically defends against data leakage attacks, requiring neither data flow nor call chain tracking.

**Encryption.** *Homomorphic encryption* enables computational operations on ciphertext, with some prior applications to mobile cloud computing. Carpov et al. [14] use homomorphic encryption to preserve the privacy of cloud-based health data. Drosatos et al. [19] use homomorphic encryption to preserve the privacy of crowd-sourced data accessed via the cloud. Besides, homomorphic encryption also can be used to compute the proximity of users in mobile social

networks: Carter et al. [15] use homomorphic encryption to find common locations and friends via private set intersection operations that preserve user privacy. *Convergently encrypted* ciphertext can be compared, so this encryption can securely identify duplicated records. Bennett et al.'s convergent encryption-based encoding scheme allows boolean searches on ciphertext [9]. Anderson et al. apply convergent encryption to securely de-duplicate the number of backup files [7]. Wilcox-O'Hearn et al. apply convergent encryption to build a secure distributed storage [39]. PoliCC brings homomorphic/convergent encryption to mobile computing to secure message-based communication while enabling untrusted apps to execute useful operations.

## 8 Conclusions

We have presented HTPD, hidden transmission and polymorphic delivery, a novel message-based communication model that secures message-based communication while allowing untrusted apps to operate on the received message data. As a reference implementation of HTPD, PoliCC plug-in replaces and extends Android ICC to defend against common data leakage attacks, while also providing a uniform API for transmitting Intents. Our evaluation confirms that PoliCC effectively prevents interception, eavesdropping, and permission escalation attacks, with low performance costs and programming effort overheads. In addition, we hope that our work would lead to HE and CE becoming widely accepted in the design space of mobile message-based communication.

**Acknowledgements.** The authors thank the anonymous reviewers, whose insightful comments helped improve this paper. NSF supported this research through the grant #1717065.

## References

1. Common Attack Pattern Enumeration and Classification. capec.mitre.org/
2. CVE-2018-15752. cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-15752
3. CVE-2018-9489. cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-9489
4. Dev tool to view inter-app communication (2019). f-droid.org/en/packages/de.k3b.android.intentintercept/
5. Intent Intercept (2019). capec.mitre.org/data/definitions/499.html
6. Alhanahnah, M., et al.: Detecting vulnerable Android inter-app communication in dynamically loaded code. In: IEEE INFOCOM 2019, pp. 550–558. IEEE (2019)
7. Anderson, P., Zhang, L.: Fast and secure laptop backups with encrypted de-duplication. In: LISA, vol. 10, p. 24th (2010)
8. Arzt, S., et al.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. Acm Sigplan Notices (2014)
9. Bennett, K., Grothoff, C., Horozov, T., Patrascu, I.: Efficient sharing of encrypted data. In: Batten, L., Seberry, J. (eds.) ACISP 2002. LNCS, vol. 2384, pp. 107–120. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45450-0_8
10. Blasco, J., Chen, T.M., Muttik, I., Roggenbach, M.: Wild android collusions (2016)

11. Bosu, A., Liu, F., Yao, D.D., Wang, G.: Collusive data leak and more: large-scale threat analysis of inter-app communications. In: Asia Conference on Computer and Communications Security, pp. 71–85. ACM (2017)
12. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R.: Xmandroid: A new Android evolution to mitigate privilege escalation attacks. Technische Universität Darmstadt, Technical Report TR-2011-04 (2011)
13. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R., Shastry, B.: Towards taming privilege-escalation attacks on Android. In: NDSS (2012)
14. Carpov, S., Nguyen, T.H., Sirdey, R., Constantino, G., Martinelli, F.: Practical privacy-preserving medical diagnosis using homomorphic encryption. In: Cloud Computing, pp. 593–599. IEEE (2016)
15. Carter, H., Amrutkar, C., Dacosta, I., Traynor, P.: For your phone only: custom protocols for efficient secure function evaluation on mobile devices. Secur. Commun. Networks **7**(7), 1165–1176 (2014)
16. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, pp. 239–252. ACM (2011)
17. Damgård, I., Groth, J., Salomonsen, G.: The theory and implementation of an electronic voting system. In: Secure Electronic Voting, pp. 77–99. Springer, Boston (2003). https://doi.org/10.1007/978-1-4615-0239-5_6
18. Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., Wallach, D.S.: Quire: lightweight provenance for smart phone operating systems. In: USENIX Security Symposium, vol. 31, p. 3 (2011)
19. Drosatos, G., Efraimidis, P.S., Athanasiadis, I.N., D'Hondt, E., Stevens, M.: A privacy-preserving cloud computing system for creating participatory noise maps. In: Computer Software and Applications Conference (COMPSAC), IEEE 36th Annual, pp. 581–586. IEEE (2012)
20. Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. (TOCS) **32**(2), 5 (2014)
21. Fang, Z., Han, W., Li, Y., Permission based Android security: Issues and countermeasures. Comput. Secur. **43**, 205–218 (2014)
22. Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission redelegation: attacks and defenses. In: USENIX Security Symposium (2011)
23. GDR!: My location (2019). https://tinyurl.com/yh9c8qok
24. Google: ANRs. developer.android.com/topic/performance/vitals/anr
25. Google: Distribution dashboard. developer.android.com/about/dashboards
26. Google: Google play (2018). play.google.com/store/apps?hl=en
27. Google: Data and file storage (2019). https://tinyurl.com/t6hr6t4
28. Jing, Y., Ahn, G.J., Doupé, A., Yi, J.H.: Checking intent-based communication in Android with intent space analysis. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, pp. 735–746. ACM (2016)
29. Krohn, M., et al.: Information flow control for standard OS abstractions. In: ACM SIGOPS Operating Systems Review, vol. 41, pp. 321–334. ACM (2007)
30. Lee, Y.K., Yoodee, P., Shahbazian, A., Nam, D., Medvidovic, N.: SEALANT: a detection and visualization tool for inter-app security vulnerabilities in Android. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, pp. 883–888. IEEE Press (2017)

31. Li, L., et al.: Iccta: Detecting inter-component privacy leaks in Android apps. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1, pp. 280–291. IEEE Press (2015)
32. Li, L., Bissyandé, T.F., Klein, J., Le Traon, Y.: Parameter values of android apis: a preliminary study on 100,000 apps. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, vol. 1, pp. 584–588 (2016)
33. Li, L., Bissyandé, T.F., Octeau, D., Klein, J.: Droidra: taming reflection to support whole-program analysis of Android apps. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, pp. 318–329. ACM (2016)
34. Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: Chex: statically vetting android apps for component hijacking vulnerabilities. In: Proceedings of the 2012 ACM conference on Computer and communications Security, pp. 229–240. ACM (2012)
35. Mark Gordon, L.Z., Tiwana, B.: A power monitor (2019). ziyang.eecs.umich.edu/projects/powertutor/
36. Mimoso, M.: Mobile app collusion can bypass native android security (2016). https://tinyurl.com/jpndk7g
37. Moez BhattiCommunication: Qksms (2019). https://tinyurl.com/k8dd4u2
38. Octeau, D., et al.: Effective inter-component communication mapping in Android: An essential step towards holistic security analysis. In: USENIX Security (2013)
39. Wilcox-O'Hearn, Z., Warner, B.: Tahoe: the least-authority filesystem. In: 4th ACM international workshop on Storage security and survivability (2008)
40. Xu, K., Li, Y., Deng, R.H.: Iccdetector: Icc-based malware detection on Android. IEEE Trans. Inf. Forensics Secur. **11**(6), 1252–1264 (2016)
41. Zhou, Y., Jiang, X.: Dissecting Android malware: characterization and evolution. In: 2012 IEEE Symposium on Security and Privacy, pp. 95–109. IEEE (2012)