The Journal of Systems & Software 193 (2022) 111460

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Secure and flexible message-based communication for mobile apps within and across devices[☆]

Yin Liu [a,*], Breno Dantas Cruz [b], Eli Tilevich [c]

[a] Faculty of Information Technology, Beijing University of Technology, Beijing 100124, China
[b] Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, IA, USA
[c] Software Innovations Lab, Virginia Tech, 2202 Kraft Drive, Blacksburg, VA 24060, USA

ARTICLE INFO

Article history:
Received 26 September 2021
Received in revised form 31 May 2022
Accepted 25 July 2022
Available online 4 August 2022

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

ABSTRACT

In modern mobile platforms, message-based communication is afflicted by data leakage attacks, through which untrustworthy apps access the transferred message data. Existing defenses are overly restrictive, as they block all suspicious message exchanges, thus preventing any app from receiving messages. To better secure message-based communication, we present a model that strengthens security, while also allowing untrusted-but-not-malicious apps to execute their business logic. Our model, HTPD, introduces two novel mechanisms: hidden transmission and polymorphic delivery. Sensitive messages are transmitted hidden in an encrypted envelope. Their delivery is polymorphic: as determined by the destination's trustworthiness, it can be delivered no data, raw data, or encrypted data. To allow an untrusted destination to operate on encrypted data deliveries, HTPD integrates *homomorphic* and *convergent* encryption. We concretely realize HTPD as PoliCC, a plug-in replacement of Android Inter-Component Communication (ICC) middleware. PoliCC mitigates three classic Android data leakage attacks, while allowing untrusted apps to perform useful operations on delivered messages. Our evaluation shows that PoliCC supports secure message-based communication within and across devices by trading off performance costs, programming effort overheads, and security[1].

1. Introduction

An essential part of modern mobile platforms is inter-app communication,[2] 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, as has been documented both in the research literature (Lu et al., 2012; Xu et al., 2016; Bugiel et al., 2012; Fang et al., 2014; Bosu et al., 2017) and in vulnerability reporting repositories (e.g., CVE[3]) (CVE, 2018a,b; Blasco et al., 2016).

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 (Enck et al., 2014; Arzt et al., 2014), and (2) track call chains, as guided by a permission restriction policy for sending/receiving data (Dietz et al., 2011; Felt et al., 2011; Bugiel et al., 2011). Although these approaches[4] 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.

☆ Editor: W. Eric Wong.
* Corresponding author.
E-mail addresses: yinliu@bjut.edu.cn (Y. Liu), bdantasc@iastate.edu (B.D. Cruz), tilevich@cs.vt.edu (E. Tilevich).
1 This article is a revised and extended version of our prior paper, published in the 17th EAI International Conference on Security and Privacy in Communication Networks (SecureComm 2021) (Liu et al., 2021)
2 In Android, it is also called inter-component communication (ICC).

3 CVE stands for the Common Vulnerabilities and Exposures, which is a well-known vulnerability reporting website (CVE, 2022).
4 All of them target Android, due to its open-sourced codebase, which can be examined and modified.

https://doi.org/10.1016/j.jss.2022.111460
0164-1212/© 2022 Elsevier Inc. All rights reserved.

In this paper, we present HTPD, a novel model that improves the security of message-based communication. The name of HTPD is from its two combined 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 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 verify our model, we developed PoliCC, an Android middleware[5] 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. In addition, PoliCC applies HTPD to both inter-app and inter-device communication, so messages can be transmitted uniformly within and across devices, thus demonstrating HTPD's generality and extensibility.

When it comes to performance overheads, as compared to Android ICC, PoliCC's attack mitigation adds at most 40.4 ms (15.4 times from baseline) and 2 mW (11.1% from baseline). Based on the security/performance trade-offs revealed by our experiments, app developers can make an informed decision on whether to apply our security enhancements to their specific application scenarios.

This article 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, and reports its performance and programming effort overheads.

This article extends our earlier paper, presented at the 17th EAI International Conference on Security and Privacy in Communication Networks (SecureComm 2021) (Liu et al., 2021). In comparison to that conference publication (18-page, single-column), this article reports on additional unpublished research that extends our prior work as follows:

(a) PoliCC now also supports Device-to-Device data communication, unlike the previous version, which only worked within a single device;

(b) We report on the results of an empirical study of thousands of popular Android apps that reveals their ICC usage, including the prevalence of specific API methods and transmitted data types;

(c) We explain the specifics of PoliCC's programming interfaces, through which Android developers can enable untrusted-but-not-malicious apps to perform useful business operations with encrypted data.

Our experiences of designing, engineering, and evaluating our approach that supports exchanging data within and across devices should be of value and relevance to the audience of this journal.

The remainder of this paper is structured as follows. Section 2 discusses our threat model. Section 3 presents the HTPD model and gives an overview of our approach. Section 4 presents our empirical study of Intent API usage. Sections 5 and 6 detail PoliCC's design and implementation, respectively. Section 7 presents our evaluation results. Section 8 discusses related work. Section 9 discusses conclusions and future work directions.

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 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:. Fig. 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, this attack can target both within and across device communication.

(1) To communicate within the same device, Android provides the inter-component communication (ICC) mechanism (detailed

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

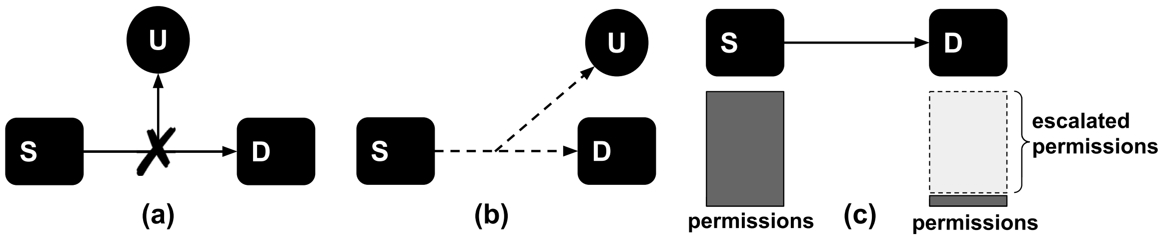

Fig. 1. Examples of attacks.

in 3.4). Using ICC, a source app sends an Intent[6] 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[7] can receive data. As defined in Common Attack Pattern Enumeration and Classification (CAPEC) (CAPEC, 2022), an interception attack occurs when malicious apps inappropriately receive an implicit Intent by declaring a certain Intent Filter (CAPEC, 2021). These attacks have been detected in many real-world Android apps (Chin et al., 2011), 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 (Intent Intercept, 2019).

(2) For device-to-device communication (i.e., across devices), consider a real attack reported in CVE, the `MensaMax` app-4.3 transmits sensitive information to a web server as cleartext, with the transmissions maliciously captured at the network packets level, resulting in data leakage (CVE, 2018b). Worse, attackers can tamper with the intercepted information to misroute them to malicious destinations.

Eavesdropping:. Fig. 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, Basic Service Set Identifier (BSSID), IP address, and DNS server). However, having declared the corresponding Intent Filter, any applications can receive this Intent and disclose the sensitive information (CVE, 2018a).

Permission escalation:. Fig. 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 (Blasco et al., 2016; Mimoso, 2016). 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 message 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 (discussed in Section 3.1).
- *Mobile devices have been paired by the user.* PoliCC transmits Intents across paired devices via Bluetooth.[8] However, the pairing procedure is outside of PoliCC's purview. Android users typically discover and pair with nearby mutually trusted devices using the built-in Bluetooth Settings component (i.e., Settings→Bluetooth).

(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)[9] that target data transmission (not data leakage), stealing data by breaking the system (not during message transmission), are out of scope.

Moreover, various attacks against key management can pilfer decryption keys. Although these attacks are a serious security

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

[8] The proliferation of wearable devices, including watches, fitness trackers, and smart-glasses, has standardized the Bluetooth technology as a foundation for connecting wearables with a smartphone. Our conceptual contribution is not bound to Bluetooth and would remain applicable for any device connection protocol (e.g., WiFi-Direct).
[9] Although DoS is not our focus, one of PoliCC's features mitigates them (see Section 7.5).

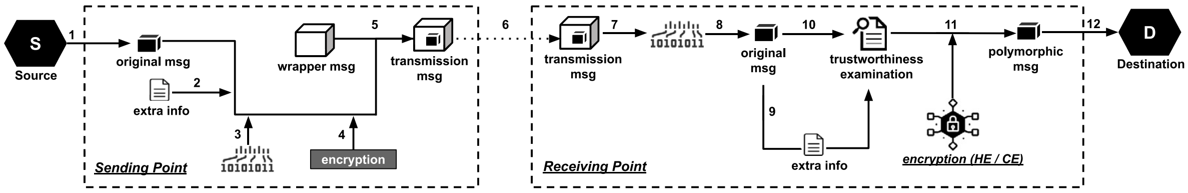

Fig. 2. HTPD Transmission Mechanisms.

challenge, mitigating these attacks is orthogonal to our approach. By relying on Android's private storage and straightforward data synchronization to manage the keys, the HTPD model's reference implementation does not exacerbate the key management vulnerabilities. Furthermore, as defenses against these vulnerabilities become more effective, they can be easily integrated with HTPD.

3. The HTPD Model

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

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 2.1. **(c)** reputation score. For example, if an app's reputation score in app markets (Google, 2018) 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).*

3.2. Transmission mechanisms

Fig. 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 verify HTPD, we developed PoliCC, a plug-in replacement of Android ICC. By mitigating Android ICC's data leakage vulnerabilities, PoliCC prevents the aforementioned attacks (Section 2.1). Following the definitions above, apps serve as message source/destination, whose sending/receiving points are managed by PoliCC (as shown 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.

To guide its polymorphic delivery, PoliCC computes the destination app's trustworthiness: (a) the delivered message's routing information,[10] 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.

Consider how PoliCC prevents the attacks described in 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 (case-1 in 2.1), or at point ❸ before the receiving point: U captures the network packets being transferred to an app on another device (case-2 in 2.1). 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 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[11] encrypted string. Since

[10] In Android ICC, routing information can be used for both data integrity and destination examinations (detailed in 6.2).
[11] Convergent encryption is applied to string data.

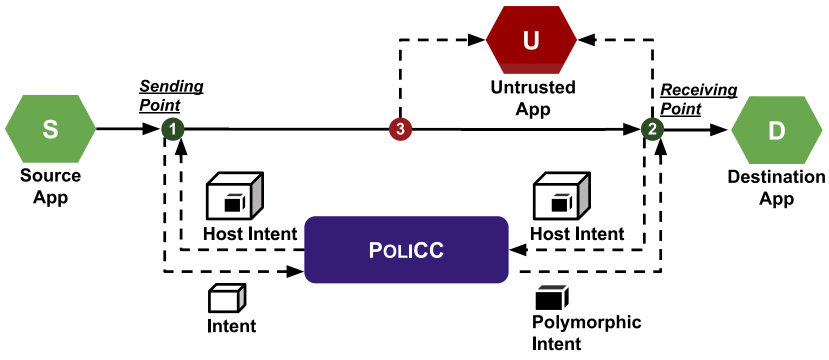

Fig. 3. PoLICC solution overview.

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 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[12] 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.

3.4. Enabling technologies

Intent and Android ICC. Intents carry data in their so-called "extended data field". To store/retrieve key–value data into/from an extended data field, apps call the "put/get" API methods (i.e., putExtra(), get*Extra(), etc.). To dispatch Intents, Android provides startActivity, which launches the Activity the user interacts with, and sendBroadcast, which launches the BroadcastReceiver that processes broadcast Intents. Before delivering an Intent, ICC must resolve it by examining which Intent Filters the installed apps declared. Upon matching Intent Filters, ICC delivers the resolved Intent to their corresponding destination apps. With multiple destinations, ICC prompts the user to select only one destination per Activity and allows all matched destinations to receive Intents broadcast using BroadcastReceiver.

Android Permission Scheme. To prevent the misuse and exploitation of sensitive data, the Android permission scheme restricts which sensitive data or API calls can be used by which apps: to obtain certain data (e.g., ACCESS_FINE_LOCATION to obtain geolocations), an app must have secured user permissions, either during installation or at runtime. Otherwise, invoking a protected call raises an exception and causes the execution to stop. Besides system-level permissions, an app can also require securing a custom permission to send/receive Intents. PoLICC uses Android application permissions: the relationship between the permission sets of the source/destination apps provides the trustworthiness measure that guides the polymorphic delivery of messages.

Homomorphic Encryption (HE) (Acar et al., 2018) encodes a numeric value, so its encrypted value (i.e., ciphertext) can be arithmetically operated on. HE preserves the invariant: the decrypted result of an arithmetic operation on ciphertexts is identical to the same operation's result on the cleartext version of the operands. Untrusted parties can operate on homomorphically encrypted ciphertext operands, while a trusted party can decrypt the encrypted result. PoLICC homomorphically encrypts message's numeric data, so its ciphertexts can still be computed without decrypting.

Convergent Encryption (CE) (Douceur et al., 2002) encodes string values, so the comparison order is preserved in their ciphertexts, as a given string (i.e., cleartext) is always encrypted into the same ciphertext. Computed from a cleartext, a hash code becomes the cleartext's encryption key. Finally, the hash code itself is encrypted with a user-provided key. Both researchers and practitioners have used CE to identify duplicated records in file systems (Duan, 2014). PoLICC convergently encrypts message's string data, so its ciphertexts can still be used in comparison to find identical data.

Xposed (Xposed, 2019) is a framework for interception the invocations of the Android system and app functions, without modifying their code. Injected custom code can be executed before or after the intercepted functions, using the beforeHookedMethod or afterHookedMethod interfaces, respectively. PoLICC use these interfaces to hook sending/receiving points.

4. An empirical study of ICC Intent

Since our solution is intended to plug-in replace Android ICC to secure message-based communication, it is necessary to study the status quo of ICC Intent usage in the Android app. Although several prior studies have analyzed how Android apps use ICC/Intent and provide a good baseline for understanding ICC scenarios, we aspired to comprehensively analyze a larger sample of apps, differentiated by their specific types (e.g., adware, spyware, trojan), with the goal of designing our solution to be able to prevent data leakage in diverse ICC scenarios.

In this section, we first summarize the findings reported by prior analyses. We then discuss the results of our analysis. These findings drive the design of our solution presented in Section 5, i.e., why we choose to keep Intents (findings 1 and 4), why we focus on Integer and String (findings 2 and 5), and why we focus on Activity and Broadcast (finding 3).

4.1. Summary of prior findings

Finding 1: Common use of extended data fields in Intents. Octeau et al. analyzed the use of ICC in Android app on (1) a "random sample" dataset with 350 apps randomly selected from over 200,000 apps in the Google Play Store, (2) a "popular app" dataset of 850 apps from top 25 most popular free apps from 34 categories in the Play store. 36% of Intents in the "random sample" dataset and 46% in the "popular apps" contain key–value data in the extended data fields (Octeau et al., 2013).

[12] Homomorphic encryption is applied to numeric data.

Finding 2: The Intent's API of operating Integer and String values (i.e., putString and putInt) are the top 10 used Android API methods (Li et al., 2016a).
Finding 3: startActivity is the most frequently used ICC method. 96% of the apps in the studied dataset (1023 malware samples collected by Zhou and Jiang (2012) and 1023 apps randomly selected from the Google Play Store) call this method, which accounts for 56% of all the ICC methods calls (Li et al., 2015).

4.2. Studying ICC usage in the wild

I. Study Methodology. We downloaded 1507 *safeware* apps (i.e., a complete set at the time of the study) from the online F-Droid repository (F-Droid, 2019), a repository of safe apps. We also analyzed over 9000 known *malware* apps from the RmvDroid repository (Wang et al., 2019), including adware, spyware, trojan, and other riskware. We analyzed these safeware and malware's distribution binaries as follows: (1) download the *.apk files for each Android app; (2) for each app, decompile its *.apk files into their `smali` (Gruver, 2015) intermediate representation, which is amenable to automated program analysis (Gruver, 2015); (3) examine the `smali` representation for the accesses to the data stored in Intent objects.
II. Major Findings.

Finding 4: More than one-third of safeware/malware apps put data into Intent's extended data fields, and three-quarters of the safeware apps and even more of the malware apps retrieve the received Intent data. Our study shows that 34% of *safeware*, 77% of *adware*, 50% of *spyware*, 50% of *trojan*, and 41% of other *riskware* invoke putExtra to place data into Intents, while 73% of *safeware*, 91% of *adware*, 72% of *spyware*, 88% of *trojan*, and 43% of other *riskware* invoke get*Extra to retrieve data from the received Intents.

Finding 5: The most common data types stored in Intent's extended data fields are strings and integers. Our findings show that 58% of *safeware*, 83% of *adware*, 62% of *spyware*, 77% of *trojan*, and 42% of other *riskware* retrieve String objects or arrays. Also, 17% of *safeware*, 63% of *adware*, 54% of *spyware*, 18% of *trojan*, and 5% of other *riskware* retrieve `int` values, arrays, or `Integer` objects. In terms of the operations that retrieve Intent data, about 65%, 59%, 54%, 80%, 79% of them return strings, in *safeware*, *adware*, *spyware*, *trojan*, and other *riskware*, respectively; while 21%, 33%, 35%, 11%, 17% of them return integers, in *safeware*, *adware*, *spyware*, *trojan*, and other *riskware*, respectively. That is, the data retrieval operations on strings and integers account for 86%, 92%, 89%, 91%, and 96% of the total retrievals, in *safeware*, *adware*, *spyware*, *trojan*, and other *riskware*, respectively.

III. Threats to Validity. As all static analysis techniques, ours is vulnerable to false positives: not all detected scenarios of Intent-based communication will be triggered in real app executions. Nevertheless, the detected numbers reasonably approximate the actual runtime occurrences of ICC.

5. PoliCCdesign

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

5.1. Design choices

PoliCC follows several design choices that we made by consulting both prior studies and our ICC usage studies above.

(1) Why keep Intents? *Finding 1—Intent's extended data fields are commonly used in storing data (Octeau et al., 2013).* Our study confirmed this finding: *Finding 4— About three-quarters of the safeware apps and even more of the malware apps access the received Intent data.*[13] Hence, as a plug-in replacement of Android ICC, PoliCC retains the Intent object as the message delivery vehicle, so existing apps could continue using Intents whose data would be protected from leakage.

(2) Why Focus on Integer and String? *Finding 2—Among the top 10 mostly used Android API methods are those that manipulate Integer and String values (i.e., putString and putInt) (Li et al., 2016a).* Our study confirmed this finding: *Finding 5—The most common data types stored in Intent's extended data fields are String and Integer.* Hence, our design supports operating on encrypted strings and integer values 6.3.

(3) Why Activity and Broadcast Communication? *Finding 3—The startActivity ICC method is the most frequently used (Zhou and Jiang, 2012).* Hence, PoliCC supports startActivity. To demonstrate HTPD's applicability, PoliCC also supports sendbroadcast, whose data flow allows multiple destinations.

(4) Why Focus on Communication both within and across Devices? When it comes to the communication both within and across devices, we decided to provide this facility in our system for two reasons: (a) Due to the growing popularity of wearable and IoT Android devices, an increasing number of common application scenarios now involve device-to-device communication. Hence, facilitating the implementation of cross-device communication is important. Furthermore, our approach to cross-device communication brings yet another option to the design space. (b) We see value in providing a unified programming interface for different communication types within and across devices. In lieu of a unified interface, app developers have to either implement vastly dissimilar communication mechanisms on their own, or learn a different set of programming interfaces for what is essentially the same communication functionality (e.g., Wearable Data Layer API Google, 2019a). These development practices incur unnecessary programmer effort and can potentially open new attack surfaces (e.g., ad-hoc implementations of communication mechanisms would likely be insecure). In addition, the time and the effort required to support these dissimilar communication mechanisms divert the resources from the difficult problem of preventing dangerous data leakage attacks: monitoring data flow or call chains across devices does require additional and dissimilar implementation strategies.

Hence, we design PoliCC to support uniform communication *within and across devices*. Not only does our reference implementation prove that the unified Intent interface can be applied to both within and across devices communication and prevent relevant data leakage attacks with our secure message-based communication mechanism, but it also identifies the performance overhead such a mechanism generates. We see this material as being conducive in potentially guiding both researchers and practitioners in determining the acceptable trade-offs between security and efficiency.

[13] For conciseness, the term "Intent data" refers to "the key–value data stored in an Intent's extended data field."

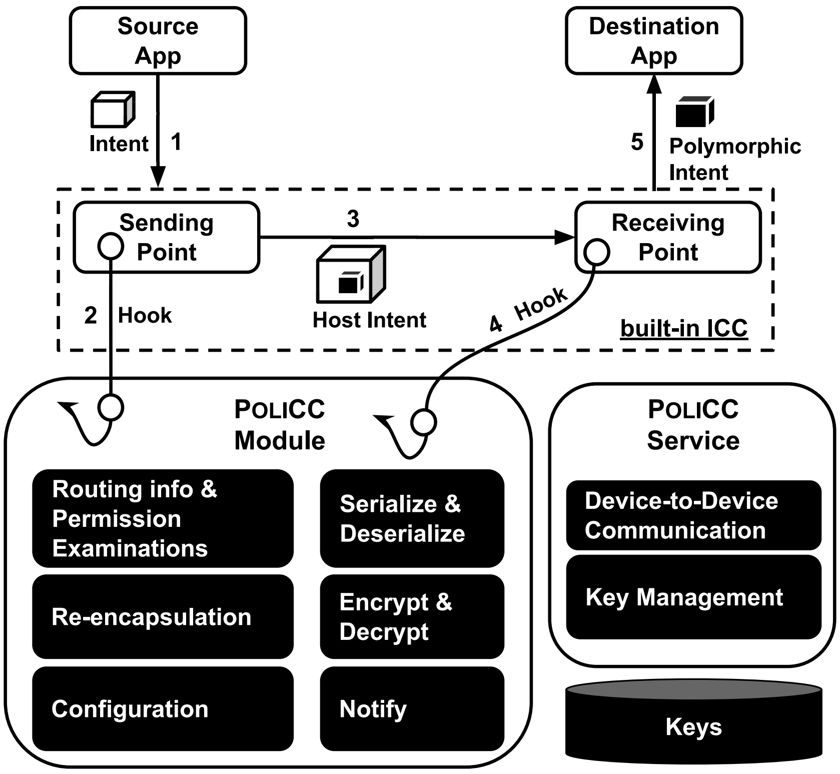

Fig. 4. PoliCC Architecture.

5.2. System architecture

Fig. 4 shows PoliCC's system architecture. As discussed in Sections 3.2 and 3.3, starting at a `Source App` (step 1), a regular Intent object is redirected to PoliCC `Module` twice (steps 2, 4), first becomes a Host Intent object (step 3), then a Polymorphic Intent object, finally being delivered to the `Destination App` (step 5). During this process, PoliCC utilizes Xposed hooks to redirect the Intent object into PoliCC `Module`. To secure the Intent object, PoliCC `Module` converts it into a Host Intent object then a Polymorphic Intent object via modules `Serialize & Deserialize`, `Re-encapsulation`, `Routing info & Permission Examination`, and `Encrypt & Decrypt`. To enable cross-device communication, PoliCC transmit Host Intents across devices via PoliCC `Service`'s `Device-to-Device Communication`. Users authorize which apps can send data across devices via `Configuration`, lest the transmitted Intents cannot cross the device's boundaries (discussed in Sections 6.1 and 6.2).

In addition, user-configured custom keys encrypt the decryption keys of homomorphic/convergent encryption, persisting them in `Keys`, private storage managed by `Key Management` (discussed in Section 6.4). Besides, 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.

With the design choices above, consider how our architecture secure the Android ICC while enabling untrusted data process. 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[14] (Damgård et al., 2003) 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` one become encrypted `String` objects.

5.3. Permission policies

As discussed in 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.[15] 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 or across devices. 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 (S and D can be installed on the same or different paired devices).

$P(S)_t$ 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 $\{I \text{ or } I_{EN} \mid (S \rightarrow D)_t\}$,

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

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

[15] Because the "no data" delivery is caused by failed data integrity checks rather than permissions, we detail it in 6.2.

(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}

6. Implementation

We describe PoliCC's hidden transmission and polymorphic delivery.

6.1. Hidden transmission

To seamlessly integrate hidden transmission into Android ICC, we had to determine: (1) where to place the sending/receiving points, (2) how to pack message data into its delivery vehicle, and (3) how to transmit messages uniformly within and across devices. We solve these problems as follows.

(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 API methods 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/receiving points by hooking into these API methods via the beforeHookedMethod and afterHookedMethod interfaces, respectively. PoliCC's custom code is injected to execute before or after the intercepted API methods, thus performing HTPD's transmission strategies. Note that we used the Hook mechanism to create a viable proof of concept to be able to evaluate PoliCC's security-enhancing properties. To deploy HTPD commercially, one should consider an implementation fully integrated with the system.

(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 piece 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 6.2). 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.

(3) Transmitting Intents. To control how apps exchange data with PoliCC, device users authorize apps to transmit data across devices. PoliCC maintains a list of the authorized apps to check each app encountered by a transmitted Intent. If all encountered apps are authorized, PoliCC continues transmitting the Intent within and across devices; otherwise, only within the same device.

Within the device, Intents are transmitted using the Android ICC channel. In contrast, to send/receive the Intent transmitted across devices, PoliCC service applies the Bluetooth technology. Launched at the device start time, the service acts as both a server and a client. As a client, to send Intents across devices, it transmits serialized Host Intents to the paired devices. As a server, at launch time, it starts a background monitoring thread to keep track of the data streams sent by remote clients. Upon receiving a sent stream, it deserializes the stream into a Host Intent, to be further transmitted by means of the Android ICC. By checking the source information, PoliCC confirms if an Intent arrived from a remote device. It forwards such Intents directly to the receiving point, then to follow the same steps as Intents transferred within the same device.

6.2. 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 5.3): 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 5.3), 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.[16] 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.

6.3. 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 apps to operate on ciphertext.

(1) Operating on Encrypted String Data. PoliCC convergently encrypts String value, so the ciphertext can be compared for equality. The encrypted string's hash code is computed via *SHA256* (Standard, 2002) to be used as the encryption key. This encryption (i.e., encrypt the string value with its hash code) uses *AES* (Miller et al., 2009) in the CTR mode.[17] The computed hash is encrypted with a user-provided key via *AES* in the CBC mode. Finally, PoliCC combines the encrypted String value and the encrypted hash, outputting the combined value as the ciphertext. Since the encrypted hash's size is fixed, the outputted ciphertext can be easily separated, so (a) the destination apps can obtain the encrypted String value to compare with their own convergently encrypted String values; (b) PoliCC can obtain the hash value to recover the raw String values.

Recall the threat model's scenario from 2.1, in which the system broadcasts an IP address, received by an untrusted app U (line 1 in the code snippet below). U wants to use the received IP address to verify its host IP. Since U's permission set is smaller than the system's, U receives a convergently encrypted IP address. To still perform its IP verification operation, U can first retrieve the received encrypted IP, by obtaining the combined ciphertext's substring that begins at the hash size's offset (line 2). Then, U can encrypt host IP in its stored collection (lines 3,4), comparing the received encrypted IP with its own encrypted IP (lines 5), and respond whether they matched (lines 6,7).

```
1  String combinedVal = intent.getStringExtra("IP");
2  String recvIP = combinedVal.substring(HASH_SIZE);
3  String hash = SHA256(hostIP);
4  String encryptedHostIP = AES(hash,hostIP);
5  if (encryptedHostIP.equals(recvIP))
6      return true;
7  return false;
```

(2) Operating on Encrypted Integer Data. HEInteger takes advantage of the *Variant of Elgamal* encryption (Damgård et al., 2003), a partial homomorphic encryption scheme that allows adding and multiplying the encrypted numeric data. HEInteger extends java.math.BigInteger and overrides the add, subtract and multiply methods to provide its own homomorphic versions of the addition, subtraction, and multiplication operations, respectively. Thus, HEInteger not only protects the contained int/Integer/BigInteger numeric data, but also supports mathematical operations on the contained encrypted data. Besides, HEInteger implements a marker interface HEType to allow instanceof queries in the apps to determine whether the retrieved integer is encrypted.

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

[17] CTR is used for performance reasons and can be replaced by other modes.

Recall the scenario from the threat model in 2.1, in which a source app with GPS permissions obtains and sends user geolocation information to a destination (e.g., a cloud or edge server) to estimate the distance of the user's movement.[18] In our case, the destination is an Android app U without these permissions. Suppose that every minute, U receives a polymorphic Intent, whose contained HEInteger object represents the new latitude to which the user moved during that minute. Having obtained the HEInteger object (lines 1,2 in the code snippet below), the app can obtain the difference between the latest value and the saved encrypted old latitude value (line 4), multiplying the result by 69 (i.e., 1 latitude $\approx$ 69 miles (US Geological Survey, 2019)) (line 5) to estimate the distance (in miles) by which the user has moved. The final result can be sent back to the source app without ever decrypting it.

```
1  Object newLatitude =
2    (Object)intent.getSerializableExtra("latitude");
3  if (newLatitude instanceof HEType) {
4    oldLatitude.subtract((HEInteger)newLatitude);
5    (HEInteger)oldLatitude.multiply(69); }}
```

6.4. Key management

In its operations, PoliCC manages three types of keys: public, private, and symmetric. The former two types are used for homomorphically encrypted data. The latter one is used for convergently encrypted data and encrypted Intent byte arrays inside Host Intents. To manage them, PoliCC's API method getHEIntegerKey() returns public keys that enable developers to construct their own homomorphically encrypted data for arithmetic operations. When it comes to the private and symmetric keys, PoliCC places them into its private storage, which can only be accessed by PoliCC itself. For device-to-device communication, we assume that the user has securely paired the communicating devices via Bluetooth (Section 2.3), so the PoliCC agents running on these devices are considered trustworthy. Once the devices are paired, the paired PoliCC agents directly synchronize their keys with each other. In the future, to exchange keys with untrustworthy devices, we plan to add new interfaces that configure PoliCC to use any available key exchange mechanism, such as Diffie–Hellman exchange (Rescorla, 1999).

7. 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?

7.1. 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 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 (Google, 2019b)) and 17.9% (the fourth highest

[18] GPS information can be used to estimate the movement, for example, one degree of latitude equals approximately 69 miles (364,000 ft) (US Geological Survey, 2019).

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.

7.2. Evaluation design

Q1. Effectiveness. As discussed in 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 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 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 messages exchanged by apps within the same device*, 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 messages exchanged by apps across devices*, we implemented another interceptor app `U2`, which intercepts the sent Intent objects before they leave the sender device's boundary, to misroute their delivery by tampering with their routing. 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 and device-to-device authorizations, we carried out each attack scenario with sufficient and insufficient permissions, with allowed and forbidden communication across devices.

More importantly, to illustrate that apps with insufficient permissions can still execute useful operations, we reused the `IP Verification` `V` and `distance estimator` (`DE`) apps from the attack scenarios (b) and (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` (Intent Intercept, 2019) (a debugging app), `Mylocation` (My Location, 2019) (a GPS app), and `QKSMS` (QKSMS, 2019) (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[19] 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 min, 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, including the hook points and device-to-device transmission (if applicable) 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 on the same or a paired device to (a) access the Intent data, and (b) retrieve and use homomorphically/convergently encrypted data.

7.3. Results

I. Effectiveness.

Reproducing Attacks

Table 1 summarizes the outcomes of reproducing each of the attacks:

(1) *For the interception attack on messages exchanged by apps within the same device* (i.e., row "Interception—within device"), 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 messages exchanged by apps across devices* (i.e., row "Interception—across devices"), 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.

(2) *For the eavesdropping attack*, whether the `IP Verification` `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").

(3) *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 estimator` (`DE`), so the

[19] We measure energy consumption with PowerTutor 1.4 (Mark Gordon and Tiwana, 2019).

Table 1
Effectiveness of PoliCC.

Attacks	Permission	Data retrieved		Successful defense		Operations		Authorization (across devices)	D-to-D
		ICC	PoliCC	ICC	PoliCC	ICC	PoliCC		
Interception —within device	insufficient	raw	**encrypted**	×	✓	–	–	forbidden	×
	sufficient	raw	**raw**	×	×	–	–		
Interception —across devices	–	–	**no data**	–	✓	–	–	allowed	✓
Eavesdropping	insufficient	raw	**encrypted**	×	✓	✓	✓	forbidden	×
	sufficient	raw	**raw**	×	×	✓	✓		
	insufficient	–	**encrypted**	–	✓	–	✓	allowed	✓
	sufficient	–	**raw**	–	×	–	✓		
Permission Escalation	insufficient	raw	**encrypted**	×	✓	✓	✓	forbidden	×
	sufficient	raw	**raw**	×	×	✓	✓		
	insufficient	–	**encrypted**	–	✓	–	✓	allowed	✓
	sufficient	–	**raw**	–	×	–	✓		

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 addition, PoliCC's across-devices transmission works as expected: with "allowed" authorization, the device-to-device Intent transmission always succeeds (column "D-to-D"); with "forbidden" authorization, the Intent cannot be transferred across the device boundary. Moreover, whenever the destination app is running on the same or different devices, PoliCC correctly delivers no data, raw data, or encrypted data, by examining the routing information and the permission relationships between the source/destination apps. In summary, the Android ICC leaves the data vulnerable to all three attacks, while PoliCC prevents these attacks in the cases of communication within and across devices, while still preserving 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 (49.6%), E_{app} by 0.8J (10.1%), and ΔE_{sys} by 1 mW (2.7%); sendBroadcast increases T by 40.4 ms (15.4 times), E_{app} by 0.5J (9.4%), ΔE_{sys} by 2mW(11.1%), 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. Not surprisingly, when communicating across paired devices, the majority of time is spent in the Bluetooth communication itself, which is a fixed cost.

Quantity-wise, 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., Krohn et al. (2007)'s performance overhead is ≈39 ms), with the total latency much lower than the Android response time limit (5000 ms Google, 2021). Also, since PoliCC increases E_{app} by 0.8J (8.7–7.9 in startActivity column) and ΔE_{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.

Percentage-wise, PoliCC increases E_{app} by 10.1% ($\frac{8.7-7.9}{7.9} * 100\%$ in startActivity) and ΔE_{sys} by 11.1% ($\frac{20-18}{18} * 100\%$ in sendBroadcast) at most, so the *energy consumption overheads* are negligible. In contrast, the *execution time overheads* are significant: PoliCC increases T by 15.4 times ($\frac{43.2}{2.8}$ in sendBroadcast column) and 49.6% ($\frac{85.3-57.0}{57.0} * 100\%$ in startActivity). We report our evaluation results from the perspectives of both absolute numbers and percentage increases, as our ultimate objective is to make it possible for developers to make informed decisions on the suitability of our solution in specific scenarios.

For cross-device communication, the highest performance and energy overheads are incurred by sending an Intent to start an Activity *across devices* (T of 539.9 ms, E_{app} of 11.4J, and ΔE_{sys} of 61 mW). The observed latency is in line with existing cross-device communication scenarios (Cruz and Tilevich, 2018; Kang et al., 2016; Le and Clyde, 2018), while the energy consumption overheads are negligible, particularly due to battery capacities soaring and fast charging technologies becoming mainstream.

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

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

ICCs	startActivity ($T/E_{app}/\Delta E_{sys}$)[a]	sendBroadcast ($T/E_{app}/\Delta E_{sys}$)[a]
Android ICC	57.0 ms/7.9 J/37 mW↑	2.8 ms/5.3 J/18 mW↑
PoliCC (within device)	85.3 ms/8.7 J/38 mW↑	43.2 ms/5.8 J/20 mW↑
PoliCC (across devices)	539.9 ms/11.4 J/61 mW↑	333.1 ms/9.2 J/55 mW↑

Operations	Sending point	Receiving point	Device-to-Device
startActivity	28.2 ms	13.5 ms	460.7 ms
sendBroadcast	28.8 ms	9.5 ms	202.9 ms

[a]T: execution time (ms); E_{app}: energy consumed by source/destination apps (J);
ΔE_{sys}: additional system energy consumed by ICCs (mW).

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

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 and 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").

7.4. Discussion

In this section, we first discuss our key evaluation results, and then present takeaways.

- *performance characteristics:* PoliCC applies the same security enhancement to both start- Activity and `sendBroadcast`, which in turn adds quite similar latency overheads: 28.3 ms for `startActivity` and 40.4 ms for `sendBroadcast`. However, percentage-wise, PoliCC increases the execution time of `sendBroadcast` by 1540%, while `startActivity` only by 49.6%. These differences is an artifact of how Android ICC implements these communication mechanisms. In the original form, `sendBroadcast` takes about 2.8 ms, while `startActivity` 57.0 ms, a 2000% difference. Adding the same overhead to two drastically different baselines explains the vast dissimilarities in the resulting overhead values. I.e.: `startActivity` increased by 49.6% (57.0 ms → 85.3 ms, added extra 28.3 ms), `sendBroadcast` by 1540% (2.8 ms → 43.2 ms, added extra 40.4 ms). In fact, even with this large performance overhead, the end user should not suffer from poor responsiveness: even 43.2 ms (as compared to the original 2.8 ms) is still less than 1/20th of a second. [20]

> *Takeaway-1*: To mitigate the performance overhead, one can integrate our solution with the Android system, which reduces the hooking process (Section 6.1) to increase execution speed. One can also refine the security mechanism for `sendBroadcast` (e.g., relatively weaken the encryption phase or use a faster encryption algorithm) in order to speed it up.

- *false positives/negatives:* PoliCC's HTPD implementation may suffer from false positives/negatives if app permissions are granted incorrectly: (1) sufficient permissions are granted to a malicious app (i.e., false-negatives); (2) insufficient permissions are granted to a benign app (i.e., false-positives).

> *Takeaway-2*: To mitigate such false positives/negatives, one can notify users of potential security attacks by using PoliCC's notification feature. As an extra feature 5.2, 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.

7.5. Limitation

As a reference implementation, PoliCC did not cover all the ICC Intent scenarios, which contained limitations below.

- *data type:* As discussed in Section 5, PoliCC only supports operating on string and integer values, including String, int, Integer, and BigInteger. However, it lacks support for other data types (e.g., string/int arrays, char, long, Parcelable, etc.) (Google, 2022). In fact, in order to allow untrusted apps to operate on ciphertext, PoliCC converts Intent's `String` data into convergently encrypted String objects, and `int`/`Integer`/`BigInteger` data into homomorphically encrypted `HEInteger` objects. Similarly, to enable other

[20] This total latency is still much lower than the Android response time limit (5000 ms Google, 2021).

types, we can either convert the data into our current homomorphic/convergent encryption objects, or create new homomorphic/convergent encryption classes for them. Such an extension would only take an additional engineering effort.

- *intercepting communication APIs:* As discussed in Section 5, PoliCC only intercepts startActivity and sendBroadcast rather than their relevant communication APIs (e.g., startActivityFromChild). We support "startActivity" because it is the most frequently used, and "sendBroadcast" because its data flow allows multiple destinations (discussed in Section 5.1-(3)). It should be possible to reuse much of the existing code for intercepting startActivity and sendBroadcast to extend this feature to other APIs.
- *hooking mechanism:* 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.

8. 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 (Enck et al., 2014) 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 (Arzt et al., 2014) applies static taint analysis to check if any app lifecycle contains data leaks. XManDroid (Bugiel et al., 2011) tracks and analyses the ICC data transferred in Intent objects at runtime to enforce the app's compliance with the defined permission policy. QUIRE (Dietz et al., 2011) enables users to examine and terminate the chain of requests associated with an ICC call. ComDroid (Chin et al., 2011) statically analyses *.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 Li et al., 2015, DINA Alhanahnah et al., 2019, DroidRA Li et al., 2016b, SEALANT Lee et al., 2017, IntentScope Jing et al., 2016) 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. (2016) use homomorphic encryption to preserve the privacy of cloud-based health data. Drosatos et al. (2012) 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. (2014) 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 (Bennett et al., 2002). Anderson et al. apply convergent encryption to securely de-duplicate the number of backup files (Anderson and Zhang, 2010). Wilcox-O'Hearn et al. apply convergent encryption to build a secure distributed storage (Wilcox-O'Hearn and Warner, 2008). In fact, many commercial systems used convergent encryption to enhance user data security and privacy: Bit (2019), Cip (2019), Fre (2019), flu (2019), and GNU (2019). PoliCC brings homomorphic/convergent encryption to mobile computing to secure message-based communication while enabling untrusted apps to execute useful operations.

Inter-Device Data Exchange. Several prior works focus on supporting Android devices to exchange data. RICCi (Cruz and Tilevich, 2018) makes it possible to exchange data peer-to-peer or over different networks by extending the built-in Intent API. SAMD (Lee et al., 2018) supports platform-level device-to-device collaboration, with the collaborating devices sharing code/resources and executing remote functionalities. Sip2Share (Canfora and Melillo, 2012) shares Services and Activities across devices by extending the Android SDK. With ShAir, distributed mobile apps share data P2P (Dubois et al., 2013). PoliCC enables devices to share data via a unified Intent API, while preserving the shared data's security.

9. 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 within and across devices, while also providing a uniform API for transmitting Intents. Our evaluation demonstrates that PoliCC mitigates interception, eavesdropping, and permission escalation attacks by trading off performance costs, programming effort overheads, and security. 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.

As future work directions, we plan to enable PoliCC to support more data types and communication APIs (e.g., startActivityFromResult) as well as further enhance its protection capabilities, for example, applying deep learning-based anomaly detection to identify the Intents that need to be protected, and utilizing Android Trusted Execution Environment (Trusty) to secure the sensitive operations.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors thank the anonymous reviewers, whose insightful comments helped improve this paper. This research is supported by National Science Foundation (NSF, USA) Grant #1717065 and Cisco Research (USA Grant CG# 1367161.

References

2019. Bitcasa. http://www.bitcasa.com/.
2019. Ciphertite. http://www.ciphertite.com.
2019. Flud. http://flud.org.
2019. Freenet. https://freenetproject.org/.
2019. Gnunet. http://gnunet.org.

Acar, A., Aksu, H., Uluagac, A.S., Conti, M., 2018. A survey on homomorphic encryption schemes: Theory and implementation. ACM Comput. Surv. 51 (4), 79.

Alhanahnah, M., Yan, Q., Bagheri, H., Zhou, H., Tsutano, Y., Srisa-an, W., Luo, X., 2019. Detecting vulnerable android inter-app communication in dynamically loaded code. In: IEEE INFOCOM 2019. IEEE, pp. 550–558.

Anderson, P., Zhang, L., 2010. Fast and secure laptop backups with encrypted de-duplication.. In: LISA, Vol. 10. p. 24th.

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P., 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. ACM SIGPLAN Notices.

Bennett, K., Grothoff, C., Horozov, T., Patrascu, I., 2002. Efficient sharing of encrypted data. In: ACISP. Springer.

Blasco, J., Chen, T.M., Muttik, I., Roggenbach, M., 2016. Wild android collusions. In: The VB Conference.

Bosu, A., Liu, F., Yao, D.D., Wang, G., 2017. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In: Asia Conference on Computer and Communications Security. ACM, pp. 71–85.

Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R., 2011. Xmandroid: A new Android Evolution to Mitigate Privilege Escalation Attacks. Technical Report TR-2011-04, Technische Universität Darmstadt.

Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R., Shastry, B., 2012. Towards taming privilege-escalation attacks on android. In: NDSS.

Canfora, G., Melillo, F., 2012. Sip2Share–a middleware for mobile peer-to-peer computing. ICSOFT 12, 445–450.

CAPEC, 2021. CAPEC-499: Android Intent Intercept. capec.mitre.org/data/definitions/499.html.

2022. CAPEC. http://capec.mitre.org/.

Carpov, S., Nguyen, T.H., Sirdey, R., Constantino, G., Martinelli, F., 2016. Practical privacy-preserving medical diagnosis using homomorphic encryption. In: Cloud Computing. IEEE, pp. 593–599.

Carter, H., Amrutkar, C., Dacosta, I., Traynor, P., 2014. For your phone only: custom protocols for efficient secure function evaluation on mobile devices. Secur. Commun. Netw. 7 (7), 1165–1176.

Chin, E., Felt, A.P., Greenwood, K., Wagner, D., 2011. Analyzing inter-application communication in android. In: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services. ACM, pp. 239–252.

Cruz, B.D., Tilevich, E., 2018. Intent to share: enhancing android inter-component communication for distributed devices. In: 2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft). IEEE, pp. 94–104.

2018. CVE-2018-9489. cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-9489.

2018. CVE-2018-15752. cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-15752.

2022. Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/.

Damgård, I., Groth, J., Salomonsen, G., 2003. The theory and implementation of an electronic voting system. In: Secure Electronic Voting. Springer, pp. 77–99.

Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., Wallach, D.S., 2011. Quire: Lightweight provenance for smart phone operating systems. In: USENIX Security Symposium, Vol. 31. p. 3.

Douceur, J.R., Adya, A., Bolosky, W.J., Simon, P., Theimer, M., 2002. Reclaiming space from duplicate files in a serverless distributed file system. In: Proceedings 22nd International Conference on Distributed Computing Systems. IEEE, pp. 617–624.

Drosatos, G., Efraimidis, P.S., Athanasiadis, I.N., D'Hondt, E., Stevens, M., 2012. A privacy-preserving cloud computing system for creating participatory noise maps. In: Computer Software and Applications Conference (COMPSAC), IEEE 36th Annual. IEEE, pp. 581–586.

Duan, Y., 2014. Distributed key generation for encrypted deduplication: Achieving the strongest privacy. In: Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security. ACM, pp. 57–68.

Dubois, D.J., Bando, Y., Watanabe, K., Holtzman, H., 2013. ShAir: Extensible middleware for mobile peer-to-peer resource sharing. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, pp. 687–690.

Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N., 2014. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. (TOCS) 32 (2), 5.

2019. F-droid. f-droid.org/.

Fang, Z., Han, W., Li, Y., 2014. Permission based android security: Issues and countermeasures. Comput. Secur. 43, 205–218.

Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E., 2011. Permission re-delegation: Attacks and defenses.. In: USENIX Security Symposium.

Google, 2018. Google play. play.google.com/store/apps?hl=en.

Google, 2019a. Send and sync data on wear. https://developer.android.com/training/wearables/data-layer.

Google, 2019b. Distribution dashboard. developer.android.com/about/dashboards.

Google, 2021. ANRs. developer.android.com/topic/performance/vitals/anr.

Google, 2022. Android developers : Intent API. developer.android.com/reference/android/content/Intent.

Gruver, B., 2015. Smali/baksmali tool. github.com/JesusFreke/smali.

2019. Intent Intercept - dev tool to view inter-app communication. f-droid.org/en/packages/de.k3b.android.intentintercept/.

Jing, Y., Ahn, G.-J., Doupé, A., Yi, J.H., 2016. Checking intent-based communication in android with intent space analysis. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security. ACM, pp. 735–746.

Kang, H., Jeong, K., Lee, K., Park, S., Kim, Y., 2016. Android RMI: a user-level remote method invocation mechanism between android devices. J. Supercomput. 72 (7), 2471–2487.

Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M.F., Kohler, E., Morris, R., 2007. Information flow control for standard OS abstractions. In: ACM SIGOPS Operating Systems Review, Vol. 41. ACM, pp. 321–334.

Le, M., Clyde, S.W., 2018. INGRIM: A middleware to enable remote method invocation routing in multiple group device-to-device networks. In: 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). IEEE, pp. 847–857.

Lee, J., Lee, H., Seo, B., Lee, Y.C., Han, H., Kang, S., 2018. SAMD: Fine-grained application sharing for mobile collaboration. In: 2018 IEEE International Conference on Pervasive Computing and Communications (PerCom). IEEE, pp. 1–10.

Lee, Y.K., Yoodee, P., Shahbazian, A., Nam, D., Medvidovic, N., 2017. 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. IEEE Press, pp. 883–888.

Li, L., Bartel, A., Bissyandé, T.F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., McDaniel, P., 2015. Iccta: Detecting inter-component privacy leaks in android apps. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1. IEEE Press, pp. 280–291.

Li, L., Bissyandé, T.F., Klein, J., Le Traon, Y., 2016a. 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.

Li, L., Bissyandé, T.F., Octeau, D., Klein, J., 2016b. Droidra: Taming reflection to support whole-program analysis of android apps. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM, pp. 318–329.

Liu, Y., Cruz, B.D., Tilevich, E., 2021. HTPD: Secure and flexible message-based communication for mobile apps. In: Proceedings of the 17th EAI International Conference on Security and Privacy in Communication Networks (SecureComm 2021).

Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G., 2012. Chex: statically vetting android apps for component hijacking vulnerabilities. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. ACM, pp. 229–240.

Mark Gordon, L.Z., Tiwana, B., 2019. A power monitor. ziyang.eecs.umich.edu/projects/powertutor/.

Miller, F.P., Vandome, A.F., McBrewster, J., 2009. Advanced encryption standard.

Mimoso, M., 2016. Mobile app collusion can bypass native android security. threatpost.com/mobile-app-collusion-can-bypass-native-android-security/121124/.

2019. My location. github.com/gjedeer/mylocation.

Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., Le Traon, Y., 2013. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In: USENIX Security.

2019. QKSMS. github.com/moezbhatti/qksms.

Rescorla, E., 1999. Diffie-Hellman Key Agreement Method. Tech. Rep..

Standard, Secure Hash, 2002. FIPS PUB 180-2. National Institute of Standards and Technology.

U.S. Geological Survey, 2019. How much distance does a degree, minute, and second cover on your maps?

Wang, H., Si, J., Li, H., Guo, Y., 2019. RmvDroid: towards a reliable android malware dataset with app metadata. In: Proceedings of the 16th International Conference on Mining Software Repositories. IEEE Press, pp. 404–408.

Wilcox-O'Hearn, Z., Warner, B., 2008. Tahoe: the least-authority filesystem. In: 4th ACM International Workshop on Storage Security and Survivability.

Xposed, 2019. Xposed framework API. https://api.xposed.info/reference/packages.html.

Xu, K., Li, Y., Deng, R.H., 2016. Iccdetector: Icc-based malware detection on android. IEEE Trans. Inf. Forensics Secur. 11 (6), 1252–1264.

Zhou, Y., Jiang, X., 2012. Dissecting android malware: Characterization and evolution. In: 2012 IEEE Symposium on Security and Privacy. IEEE, pp. 95–109.