Automated Repair of Information Flow Security in Android Implicit Inter-App Communication[⋆]

Abhishek Tiwari(✉)[1], Jyoti Prakash[2], Zhen Dong[3], and Carlo A. Furia[1]

[1] Software Institute, USI Università della Svizzera italiana, Switzerland
abhishek.tiwari@usi.ch, bugcounting.net
[2] University of Passau, Germany jyotiprakash1@acm.org
[3] Fudan University, China zhendong@fudan.edu.cn

Abstract. Android's intents provide a form of inter-app communication with implicit, capability-based matching of senders and receivers. Such kind of implicit addressing provides some much-needed flexibility but also increases the risk of introducing information flow security bugs and vulnerabilities—as there is no standard way to specify what permissions are required to access the data sent through intents, so that it is handled properly.

To mitigate such risks of intent-based communication, this paper introduces IntentRepair, an automated technique to detect such information flow security leaks and to automatically repair them. IntentRepair first finds sender and receiver modules that may communicate via intents, and such that the sender sends sensitive information that the receiver forwards to a public channel. To prevent this flow, IntentRepair patches the sender so that it also includes information about the permissions needed to access the data; and the receiver so that it will only disclose the sensitive information if it possesses the required permissions.

We evaluated a prototype implementation of IntentRepair on 869 Android open-source apps, showing that it is effective in automatically detecting and repairing information flow security bugs that originate in implicit intent-based communication, introducing only a modest overhead in terms of patch size.

1 Introduction

Mobile applications ("apps") are often designed as a collection of specialized components that rely on each other to implement functionality for the end user. Thus, inter-app communication features prominently in their implementations, and mobile operating systems offer a variety of communication primitives that are sufficiently flexible to work in an open ecosystem of apps. Unfortunately, ease of communication also brings risks of introducing information flow security bugs and vulnerabilities that are hard to prevent, detect, and fix.

A concrete instance of this problem occurs in the popular Android mobile operating system, which provides *intents* for flexible, asynchronous inter-app coordination. An app that wants to delegate an operation (for example, opening a web page) to another app instantiates an intent object specifying the operation and the data needed to execute

[⋆] First and last author's work partially supported by SNF grant 200021-207919 (LastMile).

it (for example, the web page's URL), and registers the object with the Android operating system. Any other app that is capable of executing the operation (for example, a web browser) can *receive* that intent object from the system and handle its request. This kind of *implicit* communication between apps is suitable for programming in an open ecosystem, where the app that makes a request (instantiating the intent object) does not need to know which apps can handle it (receiving the intent object). However, it may also introduce unintended leaks of sensitive information [14, 17, 18, 22, 24, 26]: the sender has no way of specifying the sensitivity of the data packed within an intent, nor can it know in advance which apps will receive and how they will handle the intent. Conversely, the receivers do not know whether they are handling sensitive data, nor which privacy policies the sender app would like to enforce.

In this paper, we propose INTENTREPAIR: an automated technique to detect information leaks that originate in inter-app intent-based communication, and to automatically *repair* them by enforcing a preferred security policy. As we better discuss in Section 5, INTENTREPAIR's focus is quite novel: plenty of existing work [5, 6, 12, 17, 18, 24, 26] deals with detecting information-flow security violations in intent communication, but most of it focuses on intra-app communication. Furthermore, to our knowledge, no other work features the automated repair of such security flaws.

To *detect* leaks, INTENTREPAIR creates a summary of any app's usage of intents—whether the app sends or receives intent objects, for which operations, and the information flow of the intents' data. Then, it matches senders and receivers for the same operation to identify possible information leaks—when a sender's sensitive data is sent to a sink in the receiver. Unlike most existing approaches, INTENTREPAIR does not just detect information leaks but can also *automatically repair* them. The key idea is to repair both the senders—adding a sensitivity declaration to any data they add to intent objects—and the receivers—checking that the received data is handled according to the sender's preferred policy. To achieve high precision, INTENTREPAIR combines static analysis of Android bytecode with dynamic taint analysis, which validates whether certain information flow are actually possible at runtime.

We implemented the INTENTREPAIR technique in a tool with the same name. We evaluated it on 14 Android open-source apps from the DroidBench [1] and RepoDroid [21] curated collections, as well as on 855 larger open-source apps from the FDroid repository. The experimental evaluation demonstrates that INTENTREPAIR can analyze apps of realistic size, successfully detect scenarios of insecure intent-based inter-app communication, and automatically generates patches that avoid the information-flow security bugs.

In summary, this paper makes the following contributions:

- INTENTREPAIR: an automated technique to detect and repair information flow privacy leaks in Android apps.
- A prototype implementation of INTENTREPAIR [25].
- An experimental evaluation of INTENTREPAIR on 869 Android apps.

2 Preliminaries

This section provides an overview of inter-app communication and its challenges in Android apps. First, Sections 2.1 and 2.2 introduce the basics of Android apps and intent communication; then, Section 2.3 details the challenges in detecting and repairing the information flows via intents.

2.1 Android Basics

Android applications are usually written in Java or Kotlin, and consist of a collection of *components* of four kinds: activities, broadcast receivers, services, and content providers [4]. Activities usually implement user interfaces, such as a login screen. System and application events, such as boot-up notifications, are broadcasted to components registered as broadcast receivers. Services are active in the background and designed for lengthy or computationally intensive tasks, such as downloading a file in the background. Content providers shuffle data from one app to another by various means.

Each app contains a manifest file `AndroidManifest.xml`, which includes essential information, such as the app's name, its components, and any libraries it depends on. The manifest also specifies an app's *permissions*, that is the features of the Android operating system (and of the device that runs it) that the app may access.

2.2 How Intent Communication Works

Android provides *intents* as a flexible communication means between components. In a nutshell, intents implement a form of message-passing communication based on the component's capabilities (called "actions" in Android parlance).

Precisely, Android intents support two ways of *addressing*, that is of identifying the recipients of a message. With *explicit* intents, the sender explicitly specifies the component(s) that may receive the message; no other components are allowed to receive it. With *implicit* intents, the sender does not specify any explicit recipients, but rather an *action* (for example, opening a web page);[4] the Android system will dispatch the message to any components that support the action specified by the sender. In other words, implicit intents support a kind of implicit, capability-based addressing.

Explicit and implicit intents provide different trade offs between ease of communication and control over the recipients. Sending sensitive data via explicit intents is generally safe, in that the sender generally knows exactly who will receive that data (and how they will use it). In contrast, sending sensitive[5] data via implicit intents may be risky, since the sender of an implicit intent generally does not know exactly who will receive the data until when the app actually runs. Thus, enforcing privacy rules during app development is a challenge when using implicit intents; tackling this challenge is the main focus of the present work.

2.3 An Example of the Challenges of Implicit Intent Communication

Figure 1 illustrates the risks of implicit intent communication through a simple example. Two sender apps each create an intent object for custom action `"action_test"`: app S in Figure 1a includes some sensitive data in the object—the host mobile device's unique

[4] Android offers a number of predefined actions, but apps may also define new custom actions.

[5] As defined more rigorously in Section 3.2.6, one can associate a permission level to any piece of data; sending high-permission data to a low-permission channel violates information flow security—whereas one is always allowed to send low-permission data to a high-permission channel.

```
1  TelephonyManager tel = (TelephonyManager)
         getSystemService(TELEPHONY_SERVICE);
2  // sensitive data
3  String imei = tel.getDeviceId();
4  // create intent object
5  Intent i = new Intent("action_test");
6  // add sensitive data to intent
7  i.putExtra("data", imei);
8  // send intent object
9  startActivity(i);
```

(a) Sender app S: intent with sensitive data.

```
// non-sensitive data                          10
String ping = "Ping";                          11
// create intent object                        12
Intent i = new Intent("action_test");          13
// add non-sensitive data to intent            14
i.putExtra("data", ping);                      15
// send intent object                          16
startActivity(i);                              17
```

(b) Sender app N: intent with normal (non-sensitive) data.

```
18  // (This app's manifest specifies it handles "action_test")
19  // receive intent object for "action_test"
20  Intent i = getIntent();
21  // take data from intent
22  String data = i.getStringExtra("data");
23  // send data to public sink
24  smsManager.sendTextMessage("1234567890", null, data, null, null);
```

(c) Receiver app R, which handles intents `"action_test"`.

Fig. 1: Android code of sender and receiver apps communicating through implicit intents.

identifier (also known as IMEI number). App N in Figure 1b, instead, only includes information that is not sensitive.

Figure 1c shows the code of another app R, which is capable of *handling* action `"action_test"`.[6] In a system where all three apps S, N, and R operate, Android would dispatch the intent messages sent by S and N to R, which would then retrieve the data and re-send it through a public channel (i.e., in a text message—Line 24 in Figure 1c).

Such a scenario has two potential problems in terms of information-flow security. First, S is not aware that R sends its sensitive data to a public channel. Second, R may not even have the necessary permissions to receive that sensitive data. Both problems originate in the flexible nature of implicit intent-based communication: the sender of an implicit intent cannot specify the sensitive nature of the data it sends; and the receivers of an implicit intent may access its data even if it contains information that is beyond their permissions.

Addressing these problems when implementing apps S, N, and R would be infeasible or too expensive, and fundamentally at odds with the flexibility introduced by implicit intents. The receiver app R cannot know, in general, the sensitivity of the data received through intents. Considering a priori all potential sender apps is also practically impossible in an open ecosystem of apps like Android. To address these issues, we propose a novel automated repair approach that works at *app deployment time*, which we describe in Section 3.2.

3 Methodology

This section presents our approach to automatically detect and repair information flow security leaks that originate with implicit intent communication. First, Section 3.1 in-

[6] An app's manifest file specifies the actions it can handle.

Statement ::=	$i \in$ *Intent* := `createIntent`$(a \in$ *Action*$)$	Create intent object and assign it to i
|	`send`$(i \in$ *Intent*$)$	Launch intent i
|	$i \in$ *Intent* := `receive`$(a \in$ *Action*$)$	Receive intent and assign it to i
|	`put`$(i \in$ *Intent*$, k \in$ *Key*$, d \in$ *Data*$)$	Add data d under key k in intent i
|	$d \in$ *Data* := `get`$(i \in$ *Intent*$, k \in$ *Key*$)$	Assign to d data under key k in intent i
|	`sink`$(d \in$ *Data*$, p \in$ *Perm*$)$	Send data d to sink with security level p

Fig. 2: Tentative: An abstract model of intent programming.

```
i := createIntent("action_test")
put(i, "data", imei)
send(i)
```

(a) Intent sender S fragment, corresponding to lines 5–9 in Figure 1a.

```
i := receive("action_test")
data := get(i, "data")
sink(data, ⊤)
```

(b) Intent receiver R fragment, corresponding to lines 20–24 in Figure 1c.

Fig. 3: An example of intent communication in Tentative.

troduces an abstract model of implicit intent-based communication; then, Section 3.2 gives an overview of our intent repair framework, followed by a detailed presentation of how its components work.

3.1 An Abstract Model of Implicit Intents

Before delving into the details of our framework, we present an abstract model of implicit intents. As is, Android offers a rich API for intent communication [3]. For example, there are 25 operations to initialize an intent object, 30 operations to add data to it, and 42 operations to extract data from it.

In this paper, we only consider *implicit* intents, where the sender does not know precisely which components will receive an intent message, but only what *actions* the receivers can handle. Figure 2 shows the syntax of Tentative: an abstract, minimal model of implicit intent communication, which we'll use in the paper to simplify the presentation of the core technical concepts. Tentative provides statements to create an intent object for a certain action a (`createIntent`(a)), to add a key-value pair k, d to an intent object i (`put`(i, k, d)), to retrieve the data stored under k from an intent object i (`get`(i, k)), to send `send`(i) and receive `receive`(a) an intent object associated with action a, and to "sink" some information into a channel (`sink`(d, p))—public, or with some other security level p. In an Android app, the action associated with a receiver is declared in the receiver app's manifest; in Tentative, it is explicit in the call to `receive`. Figure 3 shows two snippets of Tentative code modeling a basic sender and receiver: the sender in Figure 3a captures the same behavior as Figure 1a's Android code; the receiver in Figure 3b captures the same behavior as Figure 1c's Android code, where `sink`$($`data`$, \top)$ denotes that the `data` is sent to a public sink.

3.2 How Intent Repair Works

Figure 4 pictures the overall workflow of our intent repair framework; Algorithm 1 presents its corresponding high-level algorithm. The input to the intent repair process is a set of Android apps—given as APK files—whose intent-based information flow communication will be analyzed. The first step is the *receiver analysis* (described in Section 3.2.1): for each app that receives implicit intent objects, we determine which

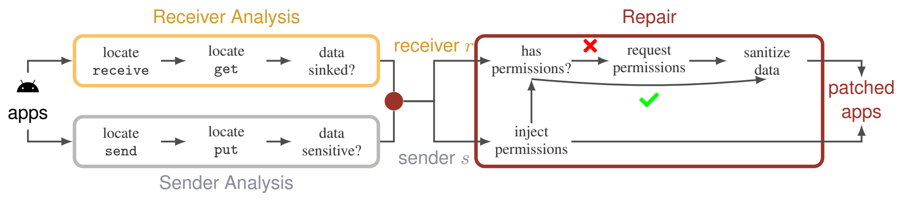


Fig. 4: An overview of how IntentRepair works.

Algorithm 1: IntentRepair's overall algorithm.

```
Input: a set of apps $A$
Output: a set of patches $P$
1  $R \leftarrow \varnothing$ // Receivers' summaries
2  $S \leftarrow \varnothing$ // Senders' summaries
3  foreach $a \in A$ do
4      $R \leftarrow R \cup \mathsf{ReceiverAnalysis}(a)$
5  if $R \neq \varnothing$ then
6      foreach $a \in A$ do
7          $S \leftarrow S \cup \mathsf{SenderAnalysis}(a)$
8  $P \leftarrow \varnothing$ // Patched apps
9  foreach $(s, r) \in \mathit{Match}(S, R)$ do
10     $s' \leftarrow \mathsf{InjectPermissions}(s)$
11     $r' \leftarrow \mathsf{CheckPermissions}(r)$
12     $P \leftarrow P \cup \{s', r'\}$
13 return $P$
```

actions it supports, and what it does with the data extracted from the intent objects—in particular, whether it leaks any of it to a sink. Assuming that at least one "potentially insecure" receiver exists, the next step is the *sender analysis* (described in Section 3.2.2), which summarizes the behavior of apps that send implicit intent objects—in particular, whether they include any sensitive data in the intents. The next step (described in Section 3.2.3) *matches* senders and receivers, identifying pairs (s, r) such that s sends sensitive information through implicit intents, r may receive such information and send it to a non-secure sink. For each such pair, the last step performs the actual *repair*: it patches the sender s so that it includes information about the permissions required to use the data it sends via implicit intents (as described in Section 3.2.4); and it patches the receiver r so that it retrieves this information and uses it to check that it has the necessary permissions to use the intent data (as described in Section 3.2.5).[7]

3.2.1 Receiver Analysis. Algorithm 2 outlines IntentRepair's receiver analysis. A receiver component is one that includes calls to the `receive` primitive (line 2).[8]

[7] In general, both the sender and receiver need repairing, as whether information flow security is enforced depends on how the receiver uses the data send by the sender. For example, a photo gallery app (sender) sends a private picture to a photo editing app (receiver); as long as the receiver does not make the private picture public, there is no privacy violation.

[8] As explained in Section 3.1, `receive` corresponds to any of the numerous Android API primitives to receive an implicit intent object, such as `getIntent()` in Figure 1c.

Algorithm 2: Analysis of receivers: ReceiverAnalysis.

```
Input: an app $A$
Output: a set of receivers' summaries $R$
1 $R \leftarrow \varnothing$ // Receivers' summaries
2 foreach $intent$ := receive($action$) $\in A$ do
3     foreach $value$ := get($intent, key$) $\in$ Taint($intent$) do
4         if sink($value$) $\in$ Taint($value$) then
5             $R \leftarrow R \cup \{\langle A, action, key \rangle\}$
6 return $R$
```

Algorithm 3: Analysis of senders: SenderAnalysis.

```
Input: an app $A$
Output: a set of senders' summaries $S$
1 $S \leftarrow \varnothing$ // Senders' summaries
2 foreach send($intent$) $\in class$ do
3     foreach $intent$ := createIntent($action$) $\in$ Slice($intent$) do
4         foreach put($intent, key, value$) $\in$ Slice($intent$) do
5             if Permission($value$) $\neq \top$ then
6                 $S \leftarrow S \cup \{\langle A, action, key, \text{Permission}(value) \rangle\}$
7 return $S$
```

IntentRepair *taints* the intent object for each such call to `receive`, in order to find program locations that extract data from the object (primitive `get`, line 3). Then, it also taints the data objects to determine if they flow into an insecure *sink* (line 4). If this is the case, all the collected information about the receiver is stored as the receiver summary (line 5).

3.2.2 Sender Analysis. Algorithm 3 outlines IntentRepair's sender analysis. A sender component is one that includes calls to the `send` primitive (line 2), which represents all variants of Android's `sendIntent` methods. For each such call to `send`, IntentRepair computes the inter-procedural backward slice using the sent intent object as slicing criterion; thus, the slice will include all calls to the `createIntent` and `put` primitives that involve the sent intent object. IntentRepair considers all pairs of `createIntent` (line 3) and `put` (line 4) in the slice that target the same *action*. By analyzing the data that is stored by each `put`, IntentRepair determines whether handling that data requires any non-trivial permission (line 5). If this is the case, all the collected information about the sender is stored as the sender summary (line 6).

3.2.3 Sender-Receiver Matching. Given a sender S's summary $\langle s, a_s, k_s, p \rangle$ and a receiver R's summary $\langle r, a_r, k_r \rangle$, *matching* them is straightforward: it amounts to determining if they send and receive intent objects associated with the same *action* ($a_s = a_r$), and exchange data using some shared *key* ($k_s = k_r$).

3.2.4 Sender Repair. If at least one matching pair of sender and receiver exists, it means there is sensitive information that may flow to a sink; in this case, the repair process begins. The first step is "repairing" the sender, which means providing means of communicating its security policies to the receiver site Android provides no built-in

```
 i := createIntent("action_test")
 put(i, "data", imei)
 // accessing imei requires
 // permission "read phone state"
+put(i, "perm:data", "READ_PHONE_STATE")
 send(i)
```

(a) Repairing Figure 1a's sender by injecting its intent object with information about the permissions that is required to access the intent data.

```
 i := receive("action_test")
 data := get(i, "data")
 // retrieve required permission
+perm := get(i, "perm:data")
 // try to acquire permission
+if (!request(perm)) exit()
 // sanitize data before sinking it
+data := sanitize(data, ⊤)
 sink(data, ⊤)
```

(b) Repairing Figure 3b's receiver by ensuring that it has the necessary permissions to access the intent data, and sanitizing the sensitive data before sending it to a non-secure sink.

Fig. 5: Abstract Repairs for the Senders and Receivers

mechanism to allow this kind of identification with intent objects—not even at runtime. To address this, we explicitly inject the intent object in the sender with additional data. The main idea is storing in the intent object pairs (k, p), where k is the *key* of a piece of data stored in the same object and p is the *permission* required to access that data.

Figure 5a illustrates this idea on the running example. Figure 1a's sender includes in intent object `i` sensitive data (an IMEI number) under key `"data"`. Android permission `READ_PHONE_STATE` is required to access this sensitive data; thus, IntentRepair injects the pair (`"data"`, `READ_PHONE_STATE`) in the sender's intent object, using a fresh, unique key `"perm:data"`.

3.2.5 Receiver Repair. As described in the previous section, IntentRepair injects intent objects on the sender's side, so that the receivers know the required permissions. Correspondingly, IntentRepair modifies all receiver apps so that they retrieve this information about permissions and use it appropriately.

First, the receiver should have the required permissions to handle the intent object. If this is not the case, IntentRepair patches the receiver so that it asks the app user to upgrade its permissions. If the user denies the request, the app is not allowed to continue and can only abort its operations.

Once the receiver has acquired the necessary permissions—either statically or dynamically—IntentRepair still has to sanitize the sensitive data it received through the intent object before dumping it into a public sink, so as not to violate any information flow security rules. To this end, IntentRepair provides a simple anonymization of the data (which could also be used, in a pinch, in the scenario where the client lacks the necessary permissions). Within the same general repair scheme, one could implement custom declassification policies for the nature of the sensitive data; for instance, if the sensitive data is location information, the receiver could replace the precise location with an approximation. IntentRepair supports customizing how receiver apps are repaired, so as to enforce the app developers' preferred policies and practices.

Figure 5b illustrates this idea on the running example. IntentRepair modifies Figure 1b's receiver so that it checks what permission `perm` is required to handle the data stored under key `"data"` in intent object `i`. If the receiver does not have nor cannot acquire permission `perm`, it simply terminates, so as to avoid any mishandling of sensitive information. Conversely, once it has acquired permission `perm`, it sanitizes the intent data before sending it to a public sink.

$$\begin{array}{rll}
\mathit{Repair} ::= & \texttt{exit()} & \text{Terminate the current activity} \\
\mid & w := \texttt{sanitize}(v \in \mathit{Data}, p \in \mathit{Perms}) & \text{Sanitize data } v \text{ to comply with permission } p \\
\mid & \texttt{request}(p \in \mathit{Perms}) & \text{Try to acquire permission } p
\end{array}$$

Fig. 6: Repair operations for Tentative intent programs.

$$\textsc{GET}\ \frac{P = (A, R) \qquad p = [\![\texttt{get(i, "perm:k")}]\!]}{\langle v \texttt{ := get(i, "k")}, P\rangle \to (A, R \cup [v \mapsto p])} \qquad \textsc{USE}\ \frac{P = (A, R) \qquad R(v) \in A}{\langle \mathit{stmt}[v], P\rangle \to P} \qquad \textsc{EXIT}\ \frac{}{\langle \texttt{exit()}, P\rangle \to \checkmark}$$

$$\textsc{SANITIZE}\ \frac{P = (A, R) \qquad R(v) \in A}{\langle w \texttt{ := sanitize}(v, p), (A, R)\rangle \to (A, R \cup [w \mapsto p])} \qquad \textsc{SINK}\ \frac{P = (A, R) \qquad R(w) \geq q}{\langle \texttt{sink}(w, q), P\rangle \to P}$$

$$\textsc{UPGRADE}\ \frac{P = (A, R) \qquad [\![\texttt{request}(p)]\!]}{\langle \texttt{request}(p), P\rangle \to (A \cup \{p\}, R)} \qquad \textsc{NO-UPGRADE}\ \frac{P = (A, R) \qquad \neg[\![\texttt{request}(p)]\!]}{\langle \texttt{request}(p), P\rangle \to P}$$

Fig. 7: Rules to check whether a Tentative program is information-flow secure.

3.2.6 Repair Correctness. To make the presentation of IntentRepair's repairs rigorous, let's extend Tentative with the set of *fix ingredients* shown in Figure 6: IntentRepair can avoid an information-flow security flaw in a receiver by terminating its execution (`exit`), sanitizing sensitive data (`sanitize`), or requesting a permission (`request`). As for the rest of Tentative, these operations generalize different Android library calls that can be used to change the permissions and a program's information-flow security—as demonstrated in Figure 5b's example.

Figure 7 shows the main rules that formalize what it means for a Tentative program to be information-flow secure. To this end, a *permission state* P keeps track of the permission as a program executes; P is a pair (A, R), where A is the set of permissions the running app currently has, whereas R maps each variable v to the permission $R(v)$ required to access that variable's content.[9] Each rule in Figure 7 has the form $\langle s, P\rangle \to P'$, which denotes that executing statement s when the permission state is P is successful and leads to permission state P' (or to termination if $P' = \checkmark$). A program is information-flow safe if we can successfully apply these rules to all its statements.

Rule GET models how the information about which permissions are needed to access which variables is retrieved by IntentRepair, which, in turn, relies on the sender repair algorithm described above. Rule USE indicates that, whenever a statement *stmt*$[v]$ accessing some variable v executes, the app must possess the necessary permission $R(v)$. Rule SINK deals with primitive `sink`, which is secure only if the output channel's security level q is not more restrictive than the permission $R(w)$ required to access the sinked data w. The program can always safely terminate, without requiring any special permission (rule EXIT). Sanitizing a variable's content may change (usually, reduce) the permission required to access it (rule SANITIZE). Conversely, successfully acquiring a permission extends the set of current permissions (rules NO-/UPGRADE).

With this formalization, we can support our claim that IntentRepair patches such as Figure 5b's are information-flow safe by construction: Line 4 retrieves the required

[9] Without loss of generality, we assume that all permissions form a complete lattice, with $\top$ being the least restrictive permission (i.e., public data).

permission; Line 6 tries to acquire it, and terminates if this is not possible; Line 8 sanitizes the data, so that Line 9 is allowed to sink it.

3.2.7 Sanitize Operations. IntentRepair can be customized and extended by providing different kinds of implementation of the `sanitize` primitive that achieve a desired trade off between security preservation and app functionality. We distinguish between *declassify* operations, which reduce the precision of the data, and pure *sanitize* operations, which completely replace sensitive data with dummy values. The latter are straightforward to implement using default values or encryption. In contrast, declassify operations depend on the nature of the data that should be declassified. For example, location data can be declassified by replacing a precise location with an approximate one. Table 1 lists several declassify and sanitize operations implemented in IntentRepair.

Table 1: IntentRepair's declassification (top) and sanitization (bottom) operations on several kinds and types of sensitive data d.

DATA d	TRANSFORMATION	DESCRIPTION
location	`coarse`(d)	approximate location d
device id	`substring`(d, n)	keep only first n characters of device id d
event	d.`location` = `coarse`(d.location)	approximate event d's location
step count	d + `random`($-10000, 10000$)	add random noise to number d of walked steps
person height	d ∗ `random`($0.5, 1.5$)	scale person height d by random factor
contacts	`filter`(d, name = person)	keep single person's contact data (instead of all contacts d)
`String`	`""`	sanitize string data
`int/Integer`	`0`	sanitize integer data
`T[]`	$\{t, t, \ldots, t\}$	sanitize array of `T`s, where t is T's sanitized default value

3.3 Implementation

We implemented our intent repair technique in a prototype tool also called IntentRepair. IntentRepair takes APK files or Java source code as input, which it analyzes as described above, and directly injects patches into the input files. IntentRepair is implemented in Java and comprises around three thousand lines of code. IntentRepair's static analysis uses the Wala framework [13] and APKTool [2] and works on the `.dex` and *Smali* intermediate representations. IntentRepair also uses Axplorer [7, 8] to detect sensitive sources based on an app's permissions. IntentRepair's implementation includes a few workarounds to handle unsupported operations of the Android framework. For example, whether the user grants the extra permissions needed by a repair is stored dynamically as a Boolean flag. If the patch were to be deployed officially, the extra permissions could be added to an app's manifest file.

4 Evaluation

We empirically evaluated the capabilities of IntentRepair in repairing information flow security bugs in Android apps. Our experiments answer the research questions:

RQ1 How *effective* is IntentRepair? ("Effectiveness" refers to how many information flow security bugs IntentRepair can detect and repair.)

RQ2 Is IntentRepair scalable? ("Scalability" refers to whether IntentRepair can be applied to realistic-size apps.)

All experiments described in this section ran on a MacBook with a 2.6 GHz 6-Core Intel Core i7 processor and 16 GB of RAM. For lack of space, we only present the main results and refer to the artifact package for details.

4.1 RQ1: Effectiveness of IntentRepair

4.1.1 Subjects. To assess IntentRepair's effectiveness, we selected apps from two widely used curated collections of open-source Android apps: DroidBench [1] and RepoDroid [21]. DroidBench was originally introduced to specifically benchmark taint analyses, whereas RepoDroid encompasses a broader selection of apps; both include a ground truth about which apps incur information-flow security leaks.

Starting from 45 apps (21 in DroidBench and 24 in RepoDroid), we selected all those that *i*) send sensitive information via implicit intents; or *ii*) receive information via implicit intents (regardless of whether they sink it or not). According to DroidBench's and RepoDroid's ground truths, only 14 apps (3 in DroidBench and 11 in RepoDroid) satisfy one or both these criteria; the selected apps are generally small, as each of them consists of only 50–100 lines of code over 2–3 classes. This is arguably due to the focus of DroidBench and RepoDroid, which were curated to mainly include apps that use (inter-app) communication mechanisms other than implicit intents. (RQ2 will demonstrate IntentRepair on a larger number of apps of realistic size.) Nevertheless, the 14 apps that we selected as experimental subjects do exhibit—in the small—significant patterns of information-flow exchanges. Figure 8 shows examples of code from some of these apps: Figure 8a is a sender that sends out the IMEI device identifier to receivers that support action `testaction`; Figures 8b and 8c are receivers supporting such action; Figure 8b sinks this sensitive data, whereas Figure 8c does not.

4.1.2 Sender/Receiver Analysis. Table 2 summarizes the outcome of IntentRepair's sender and receiver analysis on RQ1's 14 experimental subjects.

IntentRepair found 23 activities that can receive intent objects with 5 different actions; and 11 activities that can send intent objects with 4 different actions. This determines 36 potential instances of sender-receiver communication between activities. For example, app `icc_implicit_src_sink`'s `MainActivity` sends an intent object that stores Android device ids under key `"data"`; this data can be received by the `FooActivity` of the same app, as well as of the homonymous activity of apps `icc_implicit_nosrc_sink`, `icc_implicit_nosrc_nosink`, `icc_implicit_action`, and `icc_implicit_src_nosink`. We also confirmed that IntentRepair detected all information-flow security leaks in DroidBench's and RepoDroid's ground truth.

4.1.3 Repair. Out of the 36 sender-receiver communication pairs, IntentRepair reported 17 instances where the receiver may inappropriately send the intent information to a public sink. In the previous example, this happens when the receiver is app `icc_implicit_nosrc_sink`'s `FooActivity`. In all these cases, IntentRepair patched the sender-receiver pairs so as to avoid any information-flow security leaks. We manually validated these patches by running the communicating apps while monitoring the information sent to the sink, confirming that the leak occurs in the original app version (before applying the patch) and no longer occurs with IntentRepair's patch.

Table 2: INTENTREPAIR's sender and receiver summaries of the 14 apps analyzed for RQ1. For each APP and ACTIVITY, the table reports the actions it RECEIVES and/or SENDS; the KEY used to store the data in the intent object; the VALUE stored, and any PERMISSION needed to access the data. (Action `main` is `android.intent.action.MAIN`; action `test` is `amandroid.impliciticctest_action.testaction`; action `send` is `android.intent.action.SEND`; permission `phone` is `READ_PHONE_STATE`; `location` is `ACCESS_FINE_LOCATION`, `SMS` is `SEND_SMS`.)

APP	ACTIVITY	RECEIVES	SENDS	KEY	VALUE	PERMISSION
Echoer	MainActivity	send				
icc_implicit_action	MainActivity	main	test	"data"	*device id*	phone
	FooActivity	test				
icc_implicit_category	MainActivity	main	test	"data"	*device id*	phone
	FooActivity	test				
icc_implicit_data1	MainActivity	main		"data"	*device id*	phone
	FooActivity	test				
icc_implicit_data2	MainActivity	main	test/type	"data"	*device id*	phone
	FooActivity	test				
icc_implicit_mix1	MainActivity	main	test_action	"data"	*device id*	phone
	HookActivity	test_action2				
icc_implicit_mix2	MainActivity	main	test_action	"data"	*device id*	phone
	FooActivity	test_action				
icc_implicit_nosrc_nosink	MainActivity	main	test	"data"	"noSrc"	
	FooActivity	test				
icc_implicit_nosrc_sink	MainActivity	main	test	"data"	"noSrc"	
	FooActivity	test				
icc_implicit_src_sink	MainActivity	main	test	"data"	*device id*	phone
	FooActivity	test				
SendSms	MainActivity	main				phone, SMS
	Button1Listener		send	"secret"	*device id*	phone, SMS
WriteFile	MainActivity	main				location
	Button1Listener		send	"secret"	*location*	location
org.arguslab.icc_implicit_src_nosink	MainActivity	main	test	"data"	*device id*	phone
	FooActivity	test				

4.2 RQ2: Scalability of INTENTREPAIR

4.2.1 Subjects. To assess INTENTREPAIR's scalability, we selected apps from the well-known app hosting platform FDroid [19]. To only consider realistic-size apps, we first selected all apps greater than 5 MB in size (1301 apps); out of them, we further selected all those that use implicit intents (855 apps) as our experimental subjects. Table 3's left-hand half lists the five largest apps in this dataset, ranked by their size.

Although the intent communication API is very rich, Figure 9a shows that a small fraction of them dominate usage in our subjects. Two out of 25 methods to broadcast an intent object (abstracted by primitive `send` in Tentative) are used 80% of the time; `startActivity(Intent)` alone covers 65% of usages. Similarly, method `putExtra(String, String)` (out of all 30 methods to store data in an intent object) covers 40% of usages; integer values are stored in another 26% of usages.

4.2.2 Analysis and Repair. INTENTREPAIR found 98 app modules (in 83 apps) that send sensitive data (location, file system, . . .) through implicit intents; for 70 of them

```
// More code
TelephonyManager tel=(TelephonyManager) getSystemService(TELEPHONY_SERVICE);
String imei = tel.getDeviceId(); // source
Intent i = new Intent("amandroid.impliciticctest_action.testaction");
i.putExtra("data", imei);
startActivity(i); // sink
```

(a) Sender module `MainActivity` in app `icc_implicit_src_sink app`.

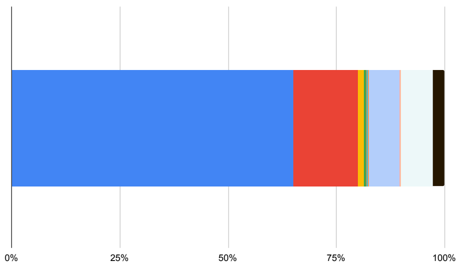
```
// More Code
Intent i = getIntent();
String imei = "" + i.getStringExtra("data");
Log.d("deviceid", imei); //sink
```

(b) Receiver module `FooActivity` in app `icc_implicit_src_sink`.

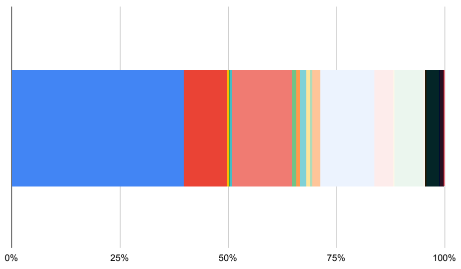
```
// More Code
Intent i = getIntent();
String v = i.getStringExtra("data");
v.trim(); //No Leak
```

(c) Receiver module `FooActivity` in app `icc_implicit_nosrc_nosink`.

Fig. 8: Snippets of code from Android apps that send sensitive information via implicit intents.



(a) Methods for sending an intent object (`send`).



(b) Methods for storing data in an intent object (`put`).

Fig. 9: Statistics about which intent API methods are used more frequently in RQ2's subjects.

(in 59 apps), IntentRepair could also determine the 10 different intents' *actions*:[10]

`android.intent.action.SEND`	`ch.blinkenlights.android.vanilla.action.LAUNCH_PLUGIN`
`android.intent.action.SENDTO`	`android.speech.action.RECOGNIZE_SPEECH`
`android.intent.action.RINGTONE_PICKER`	`android.intent.action.VIEW-URI`
`android.media.action.IMAGE_CAPTURE`	`de.azapps.mirakel.SHOW_TASK_FROM_WIDGET`
`android.app.action.ADD_DEVICE_ADMIN`	`android.intent.action.CREATE_DOCUMENT`

Table 3's right-hand half lists the five largest apps among these 59 apps, ranked by their size. IntentRepair also found 98 app modules that match some of these 70 senders; and 23 sender-receiver pairs where information-flow security leaks may happen. It successfully produced patches for all of these (validated as in RQ1).

4.2.3 Scalability.

In our experiments, IntentRepair ran for 10.5 seconds per app on average: this includes sender and receiver analysis (3.5 seconds), followed by sender-receiver matching and repair (7 seconds). This performance is reasonable for a prototype implementation, and shows that IntentRepair is also applicable to large apps.

4.3 Discussion

IntentRepair repairs information-flow leaks by simultaneously patching senders and receivers that may communicate; thus, applying the patches only to the receiving apps

[10] In the other 28 instances, the action was set dynamically and/or through complex string operations, which could not be resolved statically by IntentRepair.

Table 3: The five largest apps among all RQ2's 855 experimental subjects (left) and among the 59 of them that send sensitive data through implicit intents (right).

AMONG 855 IMPLICIT INTENT APPS			AMONG 59 SENDER APPS	
APP	SIZE (MB)	RANK	APP	SIZE (MB)
`org.openttd.fdroid`	227	1	`com.github.linwoodcloud.dev_doctor`	78
`org.olpc_france.sugarizer`	186	2	`org.dslul.openboard.inputmethod.latin`	53
`com.fr.laboussole.track`	174	3	`com.celzero.bravedns`	47
`network.mysterium.vpn`	153	4	`io.pslab`	40
`com.zhenxiang.superimage`	152	5	`org.kiwix.kiwixmobile`	40

does not suffice in general. In an ideal scenario, one may equip a receiver with the information obtained by sender analysis, and use that information at runtime to identify the sender's sensitive information (thus avoiding the need for patching the senders). Clearly, this would incur in all sorts of practical hurdles, as it is generally impossible to identify the sender apps with implicit intent communication.

Section 4's empirical evaluation of IntentRepair demonstrated that it is applicable to realistic apps, and that it generates repairs that are effective at removing the source of information-flow leaks. While we informally inspected the patched apps, and tried them out by running them, to gain some confidence that they remain usable and their overall behavior consistent, we did not perform a rigorous analysis of usability. IntentRepair repairs *senders* by simply injecting permission information in their intent objects; since the size of this information is negligible, we are fairly confident that these changes do not have any meaningful impact on the sender app's usability.

IntentRepair's repairs of *receivers* are potentially more invasive, as they may: *i*) request new permissions to the app user, and *ii*) terminate an activity to avoid an security leak. These actions are necessary, in general, to enforce information-flow security, but they may worsen the user experience. As we discussed elsewhere, IntentRepair's repair policies are customizable; thus, one may change it to achieve a different trade-off between usability and security (for example, by declassifying data instead of forcing app termination) depending on the practical application scenario.

5 Related Work

Previous work on automated repair of Android apps focused on bugs such as crashes, leaks, and configuration and compatibility issues. Tan et al. [23] describe how to repair null-pointer dereference crashes. Huang et al.'s technique [16] repairs inconsistent XML configuration files across Android versions. Zhao et al. [28] show how to generate fix templates for system- and device-compatibility issues. Guo et al. [15] detect and repair data losses that may occur when the user navigates from one UI component to another. Banerjee et al. [10] present a combined static and dynamic analysis technique to detect, validate, and repair energy bugs in Android apps. Xu et al. [27] tackle the problem of UI testing scripts becoming obsolete when an app's design changes; to this end, they propose a technique that can identify and remove obsolete testing scripts. Bhatt and Furia [11] present a static analysis tool that can detect and repair Android resource leaks. Unlike all these works, the present paper targets *information-flow* leaks in apps in accordance with the Android permission model; it combines static and dynamic analyses to automatically detect and repair such leaks.

To our knowledge, this paper is the first that can automatically *repair* information-flow security issues that occur in intent communication. In contrast, many approaches [5, 6, 9, 12, 17, 18, 24, 26] have been proposed to *detect* information-flow violations; however, most of them focus on intra-app intent communication [5, 17, 18], whereas our INTENTREPAIR fully supports inter-app detection (as we demonstrated in Section 4's evaluation). The few previous approaches that can deal with inter-app communication have limitations—such as they only work on older Android versions [12, 26], or rely on third-party slicers [24]—that restrict their effectiveness on realistic apps. Following a general-purpose approach, Mesecan [20] is a repair tool for information-flow bugs that is language- and system-agnostic, as it is based on genetic algorithms. In contrast to their work, our INTENTREPAIR is more specialized (on the Android permission model, and its intent-communication capabilities) and customizable, and is also capable of *detecting* information-flow bugs without requiring tests as input.

6 Conclusions and Future Work

In this paper, we presented INTENTREPAIR: the first automated framework to detect and repair information flow leaks that may occur in Android when apps communicate using implicit intents. To address the key issue that senders and receivers communicating via implicit intents do not have a standard way of identifying the sensitivity of the data they are sharing, INTENTREPAIR performs repair by *injecting* this information in the senders and processing it in the receivers, ensuring that their handling abides by the necessary permissions. We implemented INTENTREPAIR in a prototype tool with the same name. In a preliminary evaluation involving 14 apps from the popular DroidBench and RepoDroid benchmarks, and 855 larger apps from the FDroid repository, INTENTREPAIR showed promise, as it was able to precisely identify all known information flow security flaws in these apps, and to automatically fix them.

INTENTREPAIR is flexible, in that users can decide how to balance enforcing security and preserving app functionality when they deploy automatically generated repairs. INTENTREPAIR's analysis is also fine-grained, as it can identify different permissions for different pieces of data sent through intents. As future work, we'll systematically evaluate how to implement common declassification patterns that are found in mature apps. Note that Android does offer an API to send intent objects only to receivers with certain permissions; however, this mechanism is coarse-grained and thus inapplicable to many scenarios of implicit intent communication—as we have seen in our experiments, where we found several apps that send data associated with different actions and permissions. In the future, our approach to enable privacy-compliant intent communication could be the basis for an official Android API, or perhaps be provided as an extension of the Android framework—for example, providing the capability of annotating any piece of data with the required permissions. Finally, a rigorous evaluation of INTENTREPAIR's practical usability would also benefit from a user study.

Data availability: *Our prototype implementation of* INTENTREPAIR*, as well as the detailed experimental results, are available at* `https://doi.org/10.5281/zenodo.11957919`

References

1. Droidbench: A micro-benchmark suite to assess the stability of taint-analysis tools for android. Github, https://github.com/secure-software-engineering/DroidBench/tree/develop
2. Apktool: A tool for reverse engineering android apk files (01 2024), https://apktool.org
3. Intent communication in android (04 2024), https://developer.android.com/reference/android/content/Intent
4. Android platform architecture. https://developer.android.com/guide/platform/, accessed: 2024-01-15
5. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 259–269. PLDI '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2594291.2594299, https://doi.org/10.1145/2594291.2594299
6. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. SIGPLAN Not. **49**(6), 259–269 (jun 2014). https://doi.org/10.1145/2666356.2594299, https://doi.org/10.1145/2666356.2594299
7. Backes, M., Bugiel, S., Derr, E., McDaniel, P., Octeau, D., Weisgerber, S.: On demystifying the android application framework: Re-visiting android permission specification analysis. In: Proceedings of the 25th USENIX Conference on Security Symposium. pp. 1101–1118. SEC'16, USENIX Association, USA (2016)
8. Backes, M., Bugiel, S., Derr, E., McDaniel, P., Octeau, D., Weisgerber, S.: Github: Axplorer–android permission mappings. GitHub (Jan 2024), https://github.com/reddr/axplorer/tree/master
9. Bai, G., Ye, Q., Wu, Y., Botha, H., Sun, J., Liu, Y., Dong, J.S., Visser, W.: Towards model checking android applications. IEEE Transactions on Software Engineering **44**(6), 595–612 (2018). https://doi.org/10.1109/TSE.2017.2697848
10. Banerjee, A., Chong, L.K., Ballabriga, C., Roychoudhury, A.: Energypatch: Repairing resource leaks to improve energy-efficiency of android apps. IEEE Transactions on Software Engineering **44**(5), 470–490 (2018). https://doi.org/10.1109/TSE.2017.2689012
11. Bhatt, B.N., Furia, C.A.: Automated repair of resource leaks in android applications. Journal of Systems and Software **192**, 111417 (2022). https://doi.org/https://doi.org/10.1016/j.jss.2022.111417, https://www.sciencedirect.com/science/article/pii/S0164121222001273
12. Bosu, A., Liu, F., Yao, D.D., Wang, G.: Collusive data leak and more: Large-scale threat analysis of inter-app communications. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security. pp. 71–85. ASIA CCS '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3052973.3053004, https://doi.org/10.1145/3052973.3053004
13. Center, I.T.W.R.: Wala t. j. watson libraries for analysis. https://github.com/wala/WALA (2024 Jan)
14. Groß, S., Tiwari, A., Hammer, C.: Pianalyzer: A precise approach for pendingintent vulnerability analysis. In: Lopez, J., Zhou, J., Soriano, M. (eds.) Computer Security. pp. 41–59. Springer International Publishing, Cham (2018)
15. Guo, W., Dong, Z., Shen, L., Tian, W., Su, T., Peng, X.: Detecting and fixing data loss issues in android apps. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 605–616. ISSTA 2022, Association for Computing

Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3533767.3534402, https://doi.org/10.1145/3533767.3534402

16. Huang, H., Xu, C., Wen, M., Liu, Y., Cheung, S.: Conffix: Repairing configuration compatibility issues in android apps. ACM (2023). https://doi.org/10.1145/3597926, https://doi.org/10.1145/3597926
17. Klieber, W., Flynn, L., Bhosale, A., Jia, L., Bauer, L.: Android taint flow analysis for app sets. In: Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis. pp. 1–6. SOAP '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2614628.2614633, https://doi.org/10.1145/2614628.2614633
18. Li, L., Bartel, A., Bissyandé, T.F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., McDaniel, P.: Iccta: Detecting inter-component privacy leaks in android apps. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1. pp. 280–291. ICSE '15, IEEE Press (2015)
19. Limited, F.D., Contributors: F-droid (2023), https://f-droid.org
20. Mesecan, I., Blackwell, D., Clark, D., Cohen, M.B., Petke, J.: Hypergi: Automated detection and repair of information flow leakage. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 1358–1362 (2021). https://doi.org/10.1109/ASE51524.2021.9678758
21. Pauck, F.: Repodroid: Android benchmark reproduction framework. Github (Jan 2024), https://foellix.github.io/ReproDroid/
22. Romdhana, A., Merlo, A., Ceccato, M., Tonella, P.: Assessing the security of inter-app communications in android through reinforcement learning. Computers & Security **131**, 103311 (2023)
23. Tan, S.H., Dong, Z., Gao, X., Roychoudhury, A.: Repairing crashes in android apps. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). pp. 187–198. ACM (2018). https://doi.org/10.1145/3180155, https://doi.org/10.1145/3180155
24. Tiwari, A., Groß, S., Hammer, C.: Iifa: Modular inter-app intent information flow analysis of android applications. In: Chen, S., Choo, K.K.R., Fu, X., Lou, W., Mohaisen, A. (eds.) Security and Privacy in Communication Networks. pp. 335–349. Springer International Publishing, Cham (2019)
25. Tiwari, A., Prakash, J., Dong, Z., Furia, C.A.: Artifacts for automated repair of information flow security in android implicit inter-app communication. Zenodo (June 2024), https://doi.org/10.5281/zenodo.11957919
26. Wei, F., Roy, S., Ou, X., Robby: Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. ACM Trans. Priv. Secur. **21**(3) (apr 2018). https://doi.org/10.1145/3183575, https://doi.org/10.1145/3183575
27. Xu, T., Pan, M., Pei, Y., Li, G., Zeng, X., Zhang, T., Deng, Y., Li, X.: Guider: Gui structure and vision co-guided test script repair for android apps. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 191–203. ISSTA 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3460319.3464830, https://doi.org/10.1145/3460319.3464830
28. Zhao, Y., Li, L., Liu, K., Grundy, J.: Towards automatically repairing compatibility issues in published android apps. In: Proceedings of the 44th International Conference on Software Engineering. pp. 2142–2153. ICSE '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3510003.3510128, https://doi.org/10.1145/3510003.3510128