MuTent: Dynamic Android Intent Protection with Ownership-Based Key Distribution and Security Contracts

Pradeep Kumar D S[1], Jaejong Baek[2], Tiffany Bao[2], Yan Shoshitaishvili[2], Adam L Doupe[2], Ruoyu Wang[2], and Gail-Joon Ahn[2]
Arizona State University
[1]{pradeepkumarst}@gmail.com
[2]{jbaek7, tbao, yans, doupe, fishw, gahn}@asu.edu

Abstract

Intents are the plain-text based message object used for ICC by the Android framework. Hence the framework essentially lacks an inbuilt security mechanism to protect the visibility, accessibility, and integrity of Intent's data that facilitates adversaries to intercept or manipulate the data.

In this work, we investigate the Intent protection mechanism and propose a security-enhanced Intent library µTent that allows Android apps to securely exchange sensitive data during ICC. Differently from the existing mechanism, µTent provides accessibility and visibility of Intent data by validating the receiver's capability and provides integrity by using encryption and the Arc security contract code. Especially, ICC is initiated by exchanging µTent and follows a novel ownership-based key distribution model, that restricts the malware apps without permission from deciphering data. Through the evaluation, we show that µTent can improve the security for popular Android apps with minimal performance overheads, demonstrated using F-Droid apps.

1. Introduction

Intents are messaging object in the Android platform used to coordinate an activity, start a service, or as payload carrying sensitive data, operations, and actions to be performed by the target component (for example, the BroadcastReceiver or IntentService). Utilizing Intents without well-defined security mechanism inevitably exposes the system to threats like backdoors, which attackers can use to intercept sensitive information. For example, CVE-2018-9489 [1] reports an unintentional information leak, where the Android's WiFi module leaks the MAC address of the device to all the registered component's without validating the receiver's permissions.

There are mechanisms to prevent a malware component from receiving the Intent data. These mechanisms prevent Inter-Component Communication (ICC) attacks by statically analyzing the DEX files and dynamically monitoring to detect security vulnerabilities [2, 3]. SALMA [4] builds a Multiple-Domain-Matrix (MDM), and incrementally analyzes the App security in response to incremental system changes. Barros et al. [5] present a static program analysis where the developer can annotate the Intent data with security levels, thereby restricting data flow across incompatible source and sink.

With these filtering techniques, we can protect Intent flow across apps, however, protecting Intents dynamically from content pollution and information confidentiality through encryption of data have not been fully covered for practical use. For example, an intermediate component may pollute the Intent data by direct mutation or through reflection, thereby all the information that is used and forwarded by that subsequent component will be corrupted. The result is an epidemic propagation of a corrupted Intent which will be multiplied upon further propagation.

In terms of the confidentiality issue, EventGuard [6] proposes a mechanism to guard the information in a pub-sub environment through encryption. It uses a meta-service layer that connects the subscriber and publisher based on the filter. However, EventGuard treats every subscriber as equal with the same capability. This may not be applicable for Android, since in Android every app (subscriber) should be treated differently based on the capability of application (i.e. security properties defined in their AndroidManifest.xml and acquired at runtime by user granting/revoking permissions). Thus, any changes in App's permissions make an impact on the capability of corresponding App's component's. Overall, the lack of a holistic security framework having both integrity and confidentiality is a primary challenge in

Table 1: Types of Android Intent Attack. [**Note:** T_r = *Benign;* M_r = *Malware;* A_a=*Active Attack;* P_a=*Passive Attack;* C_b = *Channel Based;* I_b = *Intent Based*]

	Receiver's	**Type**	**Mode**	**Vulnerability Type**
UIR	M_r	Leak	P_a	(C_b, I_b)
PE	M_r	Leak	(A_a, P_a)	(C_b, I_b)
IM	T_r or M_r	Mutation	A_a	$(\_ , I_b)$

Android system to protect the Intents during ICC. In this paper, we propose µTent, a secure and pluggable Intent library supporting encryption of Intent data, an ownership-based key generation and distribution mechanism, and dynamic verification of receiver's capability and access privileges.

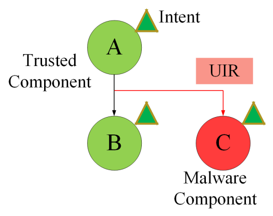
µTent has three enhanced security features: (1) dynamically verifying the sender and receiver capabilities for subset property, i.e. the source (sender or publisher) component's capability must have a subset relationship with the sink (receiver or subscriber) component's capability; (2) µTents carry *security contract code written in our own domain specific language called Arc explained in §3.3*, that will be verified dynamically before granting/revoking access to the receiver; (3) µTents are encrypted and the key is withheld by the µTent's owner, receiver component must pass conditions (1) and (2) to receive the key.

In summary, µTent provides the following advantages: firstly, capability-based access guarantees that µTent is secure wherever it goes; secondly, the Arc contracts (called simply as contract) can control not only the legal information flow in ICC but also the legal operations such as accessibility and visibility that are allowed on the Intent data; finally, every µTent object is dynamically encrypted by its owner, and makes any access satisfy the above two conditions to fetch the decryption key from µTent's owner. To the best of our knowledge, µTent is the first holistic security framework that handles dynamic Intent mutation and information confidentiality.

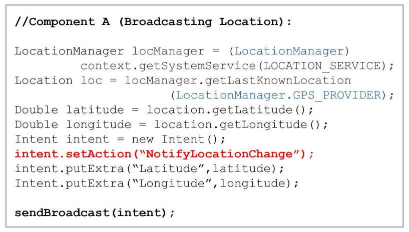
As a proof of our mechanism, we have implemented µTent as a pluggable Java Library, which can be used by developer's at the application layer by replacing Android's Intent library.

2. Overview

Intent-based communication may not trust the publishers and subscribers, and may not trust the transit path as well. A study of 7,406 applications by Ali Feizollah et.al. [7] found that 91% of the applications are using Android Intents to communicate



(a)



```
//Component A (Broadcasting Location):

LocationManager locManager = (LocationManager)
         context.getSystemService(LOCATION_SERVICE);
Location loc = locManager.getLastKnownLocation
                     (LocationManager.GPS_PROVIDER);
Double latitude = location.getLatitude();
Double longitude = location.getLongitude();
Intent intent = new Intent();
intent.setAction("NotifyLocationChange");
intent.putExtra("Latitude",latitude);
Intent.putExtra("Longitude",longitude);

sendBroadcast(intent);
```

(b) Code.

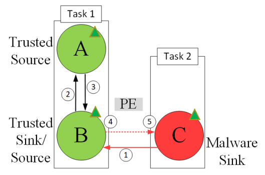
Figure 1: Unauthorized Intent Receipt Attack.

across components, among which 28.78% are implicit Intents, where the receiver component is decided by the Android framework, thereby creating a potential risk of leaking information to malicious components, and 71.22% apps uses explicit Intents that may introduce mutation attacks where the receiver component can mutate the Intent data consciously or fortuitously.

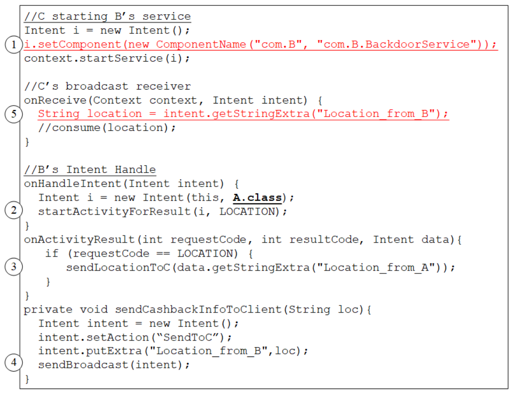
We present three types of common Intent-based attacks (see Table 1), that actively or passively confronts the Intent data security, two Intent leak attacks and one Intent mutation attack that impair the confidentiality and privacy of Android's ICC. Unauthorized Intent Receipt (UIR) and Privilege Escalation (PE) are the passive attack model where the malicious receiver (M_r) remains stealthy sneaking Intent information directly or indirectly from the source. For example, the Android WiFi module (tested on AOSP v8.1.0_r70) leaks the device sensitive information such as (*the WiFi network name, BSSID, local IP addresses, DNS server information and the MAC address*) to all registered applications running on that user device irrespective of the application's capability (permissions declared in AndroidManifest.xml and actual permissions granted by the user) (CVE-2018-9489 [1]). In privilege escalation, a malware component acquires sensitive information indirectly through a privilege component by using its loophole. In active mutation attack, the attacker (T_r) intercepts the transit Intent data and modifies the information that affects the behavior of the components further.

2.1. Motivating Example

We demonstrate our examples with the following four components: benign component A (acts as source)



(a)



```
//C starting B's service
Intent i = new Intent();
① i.setComponent(new ComponentName("com.B", "com.B.BackdoorService"));
context.startService(i);

//C's broadcast receiver
onReceive(Context context, Intent intent) {
⑤   String location = intent.getStringExtra("Location_from_B");
    //consume(location);
}

//B's Intent Handle
onHandleIntent(Intent intent) {
    Intent i = new Intent(this, A.class);
②   startActivityForResult(i, LOCATION);
}
onActivityResult(int requestCode, int resultCode, Intent data){
     if (requestCode == LOCATION) {
③        sendLocationToC(data.getStringExtra("Location_from_A"));
     }
}
private void sendCashbackInfoToClient(String loc){
    Intent intent = new Intent();
    intent.setAction("SendToC");
    intent.putExtra("Location_from_B",loc);
④   sendBroadcast(intent);
}
```

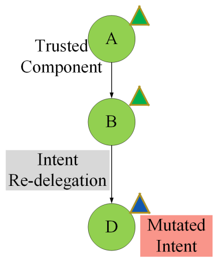
(b) Code.
Figure 2: Privilege Escalation Attack.

with capability C_a= (LOCATION, READ_CONTACTS), a crafted component B with capability C_b= (LOCATION, READ_CONTACTS), that can act either as benign (2.1.1) or as malware (2.1.2), the malware component C with capability C_c= (INTERNET), and the benign component D with capability C_d= (LOCATION, READ_CONTACTS).

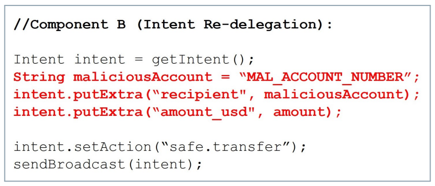
2.1.1. Intent Leak Attacks. **UIR** - As shown in Figure 1 A broadcasts the Intent with sensitive information (i.e., device locations) to B, (Figure 1b), using action ("NotifyLocationChange"). Now by registering to the action ("NotifyLocationChange") the malware component C sniffs the transit Intent from A (shown in Figure 1a), thereby gaining access to the sensitive information.

PE - In this case C without location permission indirectly accesses the location data through the benign B (Figure 2b ①, ②). B receives the location information from A (Figure 2b ③) and broadcasts it to the C (Figure 2b ④, ⑤).

2.1.2. Mutation Attack. This is a type of inside attack, where the receiver is a trusted component performing an active attack on the intent by modifying the data consciously or fortuitously and passing to other components further. We demonstrate a similar example in Figure 3a, (1) A receives the following details - sender and receiver account numbers, amount, and bank code, (2) B validates the sender and receiver account numbers, bank code, and the amount before initiating sending process, and (3) D processes the money transfer



(a)



```
//Component B (Intent Re-delegation):

Intent intent = getIntent();
String maliciousAccount = "MAL_ACCOUNT_NUMBER";
intent.putExtra("recipient", maliciousAccount);
intent.putExtra("amount_usd", amount);

intent.setAction("safe.transfer");
sendBroadcast(intent);
```

(b) Code.
Figure 3: Intent Mutation Attack.

from sender to receiver account. Here, B is performing mutation-attack by modifying the account number and the amount using the hard-coded values or inputs taken from other side-channels [8], Figure3b. Now D processes money transfer on the mutated data. This type of attack introduces unpredictable security issues that are difficult to capture using application capability or static analysis. Failure of data integrity and data security creates attacks such as man-in-the-middle attacks [9, 10] where the attacker can modify the transit data that may lead to a catastrophe in the banking applications.

2.1.3. Challenges. The Intent flow defines the source component from where the Intent originates and the sink component of Intent. Android initiates ICC by creating the Intent inside the receiver context. Since we focus on providing Intent-based ICC filtering, it becomes challenging to restrict ICC from Intents before creating Intent inside the receiver's context. Once Intent is created within receiver's context the receiver obtains complete control over the Intent object by gaining access the Intent object through direct API or through reflection, i.e., data once copied into the memory of an untrusted application can be exported thereby failed to achieve data secrecy [11].

The channel-based attacks are handled by filtering the receiver's through capability subset mechanism (§3.4), i.e. source's components capability (C_s) should be subset of the subscriber's capability (C_r) (i.e. $C_s \subseteq C_r$)). This approach ensures the confidentiality and accessibility, however the guarantee for integrity is not addressed here.

On the other hand, the Intent-based mutation attack happens even if the receiver is benign. This problem requires limiting the receiver from performing

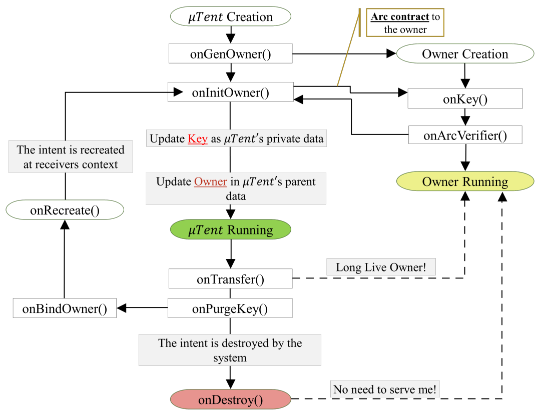


Figure 4: μTent Life Cycle.

malicious operations on the Intent data like mutating the information.

To this end, we identified three major challenges in protection against Intent-based attacks to ensure confidentiality, accessibility and integrity: (1) encrypting the Intent object so that the malicious receiver cannot eavesdrop the Intent bypassing the validation process, (2) dynamically validating the capability within the receiver's context, (3) limiting mutation on Intent data.

2.1.4. Approach and Assumptions. UIR in Figure 1 and PE in Figure 2, the component A (publisher) has a capability subset relationship with B (the subscriber) (i.e. $C_a \subseteq C_b$), therefore, B being the superset of A has all capabilities that A has and hence we recognize B as benign. Being benign B can access the Intent data from A, although the operations on Intent data (i.e. mutation) are not guaranteed. On the other hand, the component C does not have superset relationship with component A or B (i.e. $(C_a \vee C_b) \nsubseteq C_c$). Therefore component C cannot even access the Intent, thus we recognize C as malware.

The Intent-based mutation attack requires the developer to send a runtime Arc contract along with the Intent specifying its mutable or immutable nature during the transit. The Arc contract imposes the permitted operations and visibility restrictions of the receiver within the receiver's context. The contract is added during Intent creation and cannot be modified later.

Currently, our model has limits to handle the following types of leaks: 1) log leaks - leaking the data extracted from the Intent object to the log channels; 2) creating a new Intent object by copying the data from the original Intent and distributing it further along the transit path. μTent protects the Intent at object level using encryption and dynamic contracts rather than at the object's data level.

3. μTent

In this section, we present the design and architecture of μTent. μTent is a secure and pluggable Intent library supporting encryption of Intent data with an ownership-based key generation and distribution mechanism and dynamic verification of receiver's capability and access privileges. μTent encrypts the extras and data attributes, however, does not encrypt action, category, type, component that are used by Android framework for ICC, and the owner's Unique Resource Identifier (*URI*) and Arc contract.

3.1. μTent Primitives

Each μTent has two parts, (1) the μTent owner (for encryption key generation and distribution, and receiver capability validation) and (2) the Arc contract defines additional access restrictions/capabilities imposed on the receiver. The owner [12] provides a logical encapsulation boundary to the μTent object. Before initializing the μTent at the receiver's context, the receiver's context should be authorized by the μTent's owner by validating the receiver's capability and the Arc contract (if given).

Formally, the relationship between μTent and its corresponding owner (O) can be represented as $\mathtt{\mu Tent} \preceq O$. The symbol $\preceq$ is known as within (or dominated by) relationship, says that the owner (O) encapsulates the μTent with a capability-based logical boundary, protecting μTent from any unauthorized access from outside the ownership boundary. Thereby, all access to the μTent must pass through its corresponding owner (O). This ownership property ensures that possession of a μTent object by a receiver does not imply permission to access the information of that μTent. Additionally, the Arc contracts control's the legal operations allowed on the Intent data.

3.2. μTent Life Cycle

ICC initiates μTent's lifecycle by exchanging μTent across component's. Figure 4 shows various states in the μTent's lifecycle from the creation to the destruction of the μTent.

3.2.1. Creating μTent Owner. The pseudo code is given in Algorithm 1. It takes the μTent current context, the Arc contract, and the current timestamp from the environment. First the creation process verifies the correctness of the Arc version (mismatch in version throws an exception). Then the owner's unique ID is generated UUID, and the timestamp (t_s). The owner

Algorithm 1: Creating µTent Owner

```
Input: $\mu_{ctx}$: µTent Context Instantiated
Input: ARC: Arc Contract
Input: $t_s$: timestamp
1 ArcVersion ← checkArcVersion(ARC)
2 if ¬ArcVersion then
3     throw version_error() //version mismatch
4 UUID ← genRandomUUID()
5 OID ← genOwnerID($t_s$, UUID)
6 $\mu_o$ ← createOwner(OID)
7 $\mu_o$ ← initOwner(ARC)
8 $P_k$ ← genSecretKey() ∈ $\mu_o$
9 ($\mu_k \in \mu_{ctx}$) ← $P_k$
Output: $\mu_o$: µTent Owner Created
Output: $\mu_k$: µTent Secret Key from $\mu_o$
Output: $\mu_{obj}$: µTent owned-by $\mu_o$
```

then initiate the call to generate a secret key (P_k). The generated key (P_k) is stored into the owner μ_o.

3.2.2. Accessing µTent Object. The pseudo code is given in Algorithm 2. When µTent receives request for get/set methods of Intent data, the µTent validates the current context capability and privileges. On successful verification, the µTent will request the owner for the secret key (P_k). This shared secret key is used by the µTent to decrypt the Intent data.

Algorithm 2: Accessing µTent Object

```
Input: $\mu_{obj}$: µTent Object
Input: ARC: µTent's Arc Contract
Input: $t_s$: timestamp
1 $\mu_o$ ← $\mu_{obj}$.owner()
2 VERIFY ← $\mu_o$.verify($\mu_{ctx}$, ARC, $t_s$)
3 if ¬VERIFY then
4     throw Permission_Exception() //no permission
5 $\mu_{obj}$.invoke()
6 VAL ← $P_k$() ← $\mu_o$.KEY()
Output: VAL: Decrypted/Encrypted Value
```

3.2.3. Migrating µTent Object. The pseudo code is given in Algorithm 3. Migration process gets initiated during ICC: startActivity, sendBroadcast, startService, startActivityForResult, etc. Migration consists of three steps: (1) the µTent sends a signal to its corresponding owner to wait till it validates the receiver's context before recreation. (2) The µTent's data will be encrypted with the secret key, (3) The secret key is purged from the μ_{obj}, (4) followed that, the owner ID is added to μ_{obj}, (5) finally, μ_{obj} is parceled and migrated from the current context to the receiver's context.

3.2.4. Recreating µTent Object. The pseudo code is given in Algorithm 4. Inside the receiver's context, first the µTent communicates to its source owner, by sending the Arc contract, receiver's capability and time stamp. Upon successful verification, the owner will send the secret key to the µTent. On verification failure, the µTent will not get its decryption key.

Algorithm 3: Migrating µTent Objects'

```
Input: $\mu_{obj}$: µTent Object
Input: ARC: µTent's Arc Contract
Input: $\mu_{ctx}$.parcel() : Initialize Migration
Input: $t_s$: timestamp
1 $\mu_o$ ← $\mu_{obj}$.owner()
2 VERIFY ← $\mu_o$.verify(ARC, $t_s$)
3 if ¬VERIFY then
4     throw Migration_Exception() //no permission
5 INIT-BROADCAST ← $\mu_{obj}$.onBroadcast($\mu_o$)
6 ENCRYPT ← $\mu_{obj}$.encrypt()← $P_k$ ← $\mu_o$.KEY() ←
    $\mu_o$.encrypt()
7 PURGE ← $\mu_{obj}$.purgeKey()← $\mu_o$.purgeKey()
8 BIND-OWNER ← $\mu_{obj}$.bindOwner($\mu_o$)
9 PARCEL ← $\mu_{obj}$.parcel()
Output: PARCEL: Parceled µTent object
```

Algorithm 4: Recreating µTent Object

```
Input: PARCEL: Parceled µTent object
Input: $\mu_{obj}$: µTent Object
Input: $\mu_{rctx}$: µTent Receiver Context
Input: ARC: µTent's Arc Contract
Input: $t_s$: timestamp
1 $\mu_o$ ← $\mu_{obj}$.owner()
2 VERIFY ← $\mu_o$.verify($\mu_{rctx}$, ARC, $t_s$)
3 if ¬VERIFY then
4     throw Unbox_Exception() //no permission to unbox
5 $\mu_{obj}$.KEY ← $P_k$ ← $\mu_o$.KEY()
Output: $\mu_{obj}$.KEY: Key from Owner
Output: VAL: Decrypted/Encrypted Value
```

3.2.5. Destroying µTent Object. µTent destruction state pseudo code is given in Algorithm 5. Here, the µTent sends an notify signal to its owner. Followed by, purging the secret key, and µTent's destroy signal to its corresponding owner. The destroy signal removes the µTent's dependency from its owner, and now the owner can self-destruct itself.

Algorithm 5: Destroying µTent Object

```
Input: $\mu_{obj}$: µTent Object
Input: $\mu_o$: µTent Owner
Input: $\mu_{obj}$.destroy($\mu_o$) : Initialize Destruction
Input: $t_s$: timestamp
1 SIGNAL-OWNER ← $\mu_o$.notify($t_s$)
2 PURGE ← $\mu_{obj}$.purgeKey()← $\mu_o$.purgeKey()
3 KILL-OWNER ← $\mu_{obj}$.destroy($\mu_o$)
Output: KILL: µTent object is removed
Output: $KILL_{opt}$: $\mu_o$ removed iff no dependent $\mu_{obj}$
```

3.3. Arc Language Specifications

The Arc is a design by contract language, with basic constructs for Android security. The Arc follows annotation such as syntax similar to JML [13]. The current Arc implementation supports eight basic constructs: (1) @sameProcess; (2) @sameTask; (3) @read; (4) @write; (5) $\wedge$; (6) $\vee$; (7) $\neg$; (8) $\xrightarrow{impl}$.

The Arc grammar is given in Figure 5. Constraints describe the receiver's context, and the **access** keyword is used to specify the accessibility. The meta variable **P** range over the contracts. The meta variable **OP** describe the logical operators.

@sameProcess: It limits µTent's visibility and accessibility within the same process (source) that created the µTent, thereby component's from different processes cannot access the µTent information. By default, µTent is accessible across processes.

@sameTask: It ensures that the receiver's task ID and task affinity (in case of BroadcastReceiver and IntentService) is same as the source's task ID and task affinity. By default, µTent is accessible across task.

@read: It protects the µTent data from any future (unintended) modifications. Negation of read specifies that the µTent cannot be read by the receiver. For example the Arc contract "$\neg\texttt{@sameTask} \xrightarrow{\text{impl}} \neg\texttt{@read}$" specifies **no read constraint** to the receiver's' belongs to different task. That is receiver's from different task cannot read the data. Under this condition the receiver can only forward/drop the µTent and cannot consume any information from it as they are protected with encryption.

@write: It enables µTent data to get modified, by default µTent is writable. Negation of @write (i.e. $\neg$@write) means that the µTent is read-only.

$\wedge \mid \vee \mid \neg$: The logical represents general math logic for **AND | OR | NOT** operation respectively.

$\xrightarrow{\text{impl}}$: The implication operation can be expressed as **(X $\xrightarrow{\textbf{impl}}$ Y)**, where Y becomes the resultant for the operations defined in (X). Y will be evaluated iff the operation defined in (X) results ***true***.

3.4. µTent's Subset Properties

µTent follows subset property to classify the component's as either (1) Benign, or (2) Malware. By default µTent source is labeled as benign. The receiver is labeled as benign iff the sender's capability (C_s) forms a subset of the receiver's capability (C_r), i.e. $C_s \subseteq C_r$, else the receiver is labeled as malware, i.e. $C_s \nsubseteq C_r$. By default, empty permission (i.e. when an application doesn't define any permissions in AndroidManifest.xml) is considered as a special permission. Therefore, if the receiver's has an empty capability ($C_r = \{\emptyset\}$) and the sender has read contacts capability ($C_s = \{\text{READ\_CONTACTS}\}$) then receiver's capability is not a subset of the sender's capability ($C_r \nsubseteq C_s$).

$$
\begin{array}{lcl}
\mathtt{P} & ::= & (\mathtt{contract}\ (\xrightarrow{\text{impl}} \mathtt{access})_{\mathtt{opt}})^{*} \\
\mathtt{contract} & ::= & \mathtt{constraint}\ (\mathtt{OP}\ \mathtt{constraint})^{*} \\
\mathtt{constraint} & ::= & \mathtt{@sameProcess} \\
 & \mid & \mathtt{@sameTask} \\
\mathtt{access} & ::= & \mathtt{@read} \\
 & \mid & \mathtt{@write} \\
\mathtt{OP} & ::= & \wedge \mid \vee \mid \neg
\end{array}
$$

Figure 5: Arc Grammar

4. Evaluation

We have implemented µTent as a pluggable Java Library for Android (8.1+). µTent library contains 11 files and approximately 2,000 lines of code. We present our evaluation report for µTent tested using the following hardware: Android 8.1.0 (Oreo, SDK Version = 27), OnePlus 3T device with Snapdragon 821 CPU and 6GB RAM.

4.1. Data Set

Our benchmark application consists of the following component's: (1) $\mathtt{Compo}^{(n,*)}$ can be either activity or broadcast receiver's or Services, where n represents process/task/task-affinity to which the component belongs and $*$ represents component capability (Benign (P) or Malware (LP)).

We evaluated µTent's accuracy by creating a repository of 288 applications (shown in Table 3) with total 276 vulnerable paths taken from existing benchmarks [14,15] and added our own ICC applications with the following programming practices such as: data mutation, read-only, no-read, and encryption. In addition to the above benchmark, we have also tested µTent with the SEALANT benchmark, which consists of 135 apps with total 59 vulnerable paths.

In our test, we fixed the Arc contract (i.e. $\mathtt{DEFAULT} \wedge \mathtt{@read}$) for all the examples presented in Table 2. The rule DEFAULT represents the permission subset property (§3.4), encryption property (§3.2) of the µTent and "@read" constraint enforce the read-only property on the µTent within the receiver's context.

The application set from (1-6) in Table 2 consists of a total 72 apps demonstrating the scenario of sniffing between component's across different process and task. In the application set (3,6), the receiver being benign performs mutation attack on µTent data. Though such attacks are generally assumed safe, in certain scenarios like (§2.1) it is required to limit mutation in order to ensure integrity of the data. Such attacks are classified as tricky ICC which µTent handles by using "@read" Arc contract limiting mutation of Intent data.

The application set (7-12) in Table 2 consists of 162 apps with 162 vulnerable paths. In total we have 210 vulnerable paths demonstrating the scenario of UIR and

Table 2: Detailed Comparison

-	Name[Tot.Apps]	Covered Attacks	Model (Component$^{(1,*)} \xrightarrow[L_n]{}$ Component$^{(2,*)}$)	Arc Contract	$\mu T^{\ddagger}$	$SE^{\S}$	FP	FN
1	*sameP1*[6]	*UIR,PE*	$Compo^{(1,P)} \xrightarrow[L_{k1}]{} Compo^{(2,P)}$	arc_c	●	●		
2	*sameP2*[24]	*UIR,PE*	$Compo^{(1,P)} \xrightarrow[L_{k1}]{} Compo^{(2,LP)}$	arc_c	●	●		
3	*sameP3*[6]	*IM*	$Compo^{(1,P)} \xrightarrow[L_{k1}]{} Compo^{(2,P)} \looparrowright [I_m]$	arc_c	●	○		⊘
4	*sameT1*[6]	*UIR,PE*	$Compo^{(1,P)} \xrightarrow[L_{k1}]{} Compo^{(2,P)}$	arc_c	●	●		
5	*sameT2*[24]	*UIR,PE*	$Compo^{(1,P)} \xrightarrow[L_{k1}]{} Compo^{(2,LP)}$	arc_c	●	●		
6	*sameT3*[6]	*IM*	$Compo^{(1,P)} \xrightarrow[L_{k1}]{} Compo^{(2,P)} \looparrowright [I_m]$	arc_c	●	○		⊘
7	*redel1*[27]	*UIR,PE*	$Compo^{(1,P)} \xrightarrow[L_1]{} Compo^{(1,P)} \xrightarrow[L_{k2}]{} Compo^{(2,LP)}$	arc_c	●	●		
8	*redel2*[27]	*UIR,PE*	$Compo^{(1,P)} \xrightarrow[L_1]{} Compo^{(1,P)} \xrightarrow[L_2]{} Compo^{(2,P)}$	arc_c	●	●		
9	*redel3*[27]	*UIR,PE*	$Compo^{(1,P)} \xrightarrow[L_1]{} Compo^{(2,P)} \xrightarrow[L_2]{} Compo^{(3,P)}$	arc_c	●	●		
10	*redel4*[27]	*UIR,PE*	$Compo^{(1,P)} \xrightarrow[L_1]{} Compo^{(2,P)} \xrightarrow[L_{k2}]{} Compo^{(3,LP)}$	arc_c	●	●		
11	*redel5*[27]	*UIR,PE*	$Compo^{(1,P)} \xrightarrow[L_{k1}]{} Compo^{(2,LP)} \xrightarrow[L_{k2}]{} Compo^{(3,P)}$	arc_c	●	●		
12	*redel6*[27]	*UIR,PE*	$Compo^{(1,P)} \xrightarrow[L_{k1}]{} Compo^{(2,LP)} \xrightarrow[L_{k2}]{} Compo^{(3,LP)}$	arc_c	●	●		
13	*InMut1*[27]	*IM*	$Compo^{(1,P)} \xrightarrow[L_{k2}]{} Compo^{(2,LP)} \Rightarrow [I_m] \xrightarrow[L_1]{} Compo^{(3,P)}$	arc_c	●	●		
14	*InMut2*[27]	*IM*	$Compo^{(1,P)} \xrightarrow[L_{k2}]{} Compo^{(2,P)} \looparrowright [I_m] \xrightarrow[L_1]{} Compo^{(3,P)}$	arc_c	●	○		⊘

Note:
‡=MuTent; §=SEALANT; ● = provides property; ○ = does not provide property; L_n = safe paths;
L_{kn} = vulnerable paths; $\looparrowright$ = tricky intent operations $\Rightarrow$ = direct/indirect intent operations I_m = Intent Mutation Attack;
⊖ = False Negative (μTent); ⊘ = False Negative (SEALANT) arc_c = [DEFAULT $\wedge$ @read]

Table 3: μTent Evaluation.

Attacks	**Tot.Apps**	**Tot. Vulnerable Paths**	**Blocked** (μT)	**Blocked** (SE)
			(no.,%)	(no.,%)
UIR	111	105	105(100%)	105(100%)
PE	111	105	105(100%)	105(100%)
IM	66	66	66(100%)	27(40.91%)
Total	288	276	276(100%)	237(85.87%)

Table 4: μTent Overall Impact (%).

Apps	**Intent Creation Time(ms)**	**μTent Creation Time(ms)**	**μTent Overall Dif.(ms)**	**μTent Overall Impact(%)**
Ads Droid	0.45	14.2	13.75	0.0046
Activity Diary	0.77	14.0	13.23	0.0044
Face Slim	0.75	9.75	9.0	0.0030

PE. μTent blocks all the 210 vulnerable path by using its "DEFAULT" property of validating the sender and the receiver's capability as discussed in §3.4.

Finally, we have added a collections of 54 apps to demonstrate mutation-attacks (13,14). The attack scenario (13) has been filtered easily by μTent by validating the capability subset property, thereby the malware receiver $Compo^{2,LP}$ cannot access the data. However, in attack scenario (14) the benign component is mutating the data. In order to handle such class of attacks, we used the "@read" Arc contract that specifies read-only operation to the receiver ($Compo^{2,P}$), thereby limiting the receiver from modifying the data.

4.2. Effectiveness

Table 2 presents the detailed comparison between μTent and SEALANT [2]. The attack scenarios (3, 6, 14) demonstrates tricky operations performed by trusted component's, where the transit ICC path consists of only benign component's, which creates difficulty to restrict access based on receiver's capability and hence filtering only based on the component's capability may result in a False Negative outcome.

The other similar tools/methodology available in literature are XmanDroid [16,17] (a policy based model) and Sparta [5] (a static tool to identify the taint-flow). XmanDroid uses application level policies to block vulnerable paths. It demands the user to specify the policy defining the vulnerable ICC path. Absence of policy makes XmanDroid to allow every ICC, that is without such policy all the component's are assumed benign by default.

μTent differs from XmanDroid by performing permission subset validation of the receiver by default (without requiring an explicit policy specification), and protecting the payload with an ownership-based encryption model. μTent considers the following application-level covert channels [18] such as:

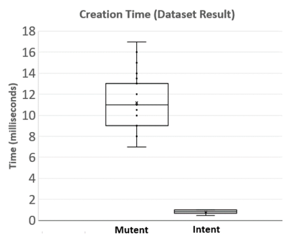


Figure 6: Overall Creation Time Overhead

broadcasts, system services, or content providers that uses Android's Intent sharing as communication medium and not any other covert channels. Also in this paper we are not considering the other operating-system level or hardware-level covert channels [19, 20] such as: file system, database, log, cache memory etc. µTent uses dynamic verification of the receiver based on receiver's capability and the Arc contract, rather than static taint analysis of source-code such as Sparta, or static apk analysis for vulnerable paths such as SEALANT.

4.3. Performance

Table 3 shows the overall impact of the µTent. We evaluate by analyzing the number of vulnerable paths present and the number of paths µTent's can identify as vulnerable, and can block by throwing exception. The test result is from the outcome of evaluating the µTent benchmark consisting of 288 total apps (§4.1). The benchmark consists of a total 276 vulnerable paths, out-of which µTent can identify and block 276 paths (i.e. 100% success), and SEALANT has 85.87% accuracy by blocking 237 paths.

Additionally, we tested µTent with the SEALANT benchmark, which consists of 135 apps with total 59 vulnerable paths. In this test, µTent was able to identify and block 112 apps (with an accuracy of 82.96%) and 48 vulnerable paths (with an accuracy of 81.36%) that consists of redelegation attacks and Intent data access.

4.4. Creation Time Overhead

Since µTent object requires an additional dynamic owner creation along with a complex algorithm for encryption and key distribution, it becomes necessary for us to calculate the time taken to create a µTent so that we can calculate the overall impact µTent can impose on an application. Hence in this section we discuss the average creation time and its impact ratio based on our current µTent library implementation. We tested µTent by modifying three open source projects taken from F-Droid, and modified the Intent with µTent library (shown in Table 4). On an average µTent took ~11.5

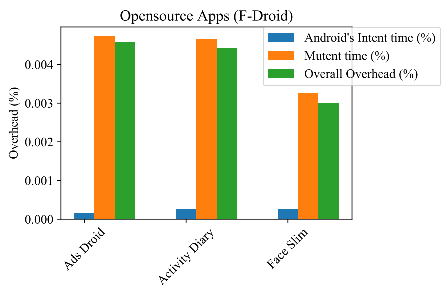


Figure 7: µTent Implementation Overhead

milliseconds (Figure 7) more than Android's Intent that is approximately equivalent to 0.005(%) overhead in a 300,000 milliseconds of an application runtime as shown in Figure 6.

We tested µTent using two approaches: (1) using Culebra - a record and play test environment [21] where user interactions can be recorded as python script and can be played automatically, and (2) manual testing - by giving the applications to three student volunteers from our lab and asked them to interact with the application randomly (without any pre-defined screen interaction order). Even though µTent's creation time is higher than Android's Intent, the overall impact percentage added by µTent to an applications execution is trivial (i.e. ~0.005%). Thus with this nominal impact on creation time, during our testing the users have not felt any latency in the screen response and screen navigation.

5. Related Work

Detecting data-leaks and malicious component's during Inter Component Communication (ICC) have already been explored for source code and DEX file. These approaches combines static analysis and runtime policies at the OS level (i.e., requires modifying the framework or kernel code). However, µTent is designed and implemented as a pluggable Java Intent library to filter the data-leaks and malicious component's during ICC. Encrypting ICC payload is generally treated as a OS layer functionality, which µTent introduce as Intent property.

Static Analysis: SEALANT [2] provides a technique that combines static analysis of app code infering vulnerable communication channels and runtime monitoring of inter-app communication through those channels to prevent attacks. SALMA [4], presents self-protecting Android system using a Multiple-Domain-Matrix (a knowledge base), that will be updated incrementally and thereby enabling incremental analyzes upon changes to the system. FlowDroid [22] a static taint-analysis reduces false alarm by detecting the data-leaks based on the context, flow, field and object-sensitivity. Amandroid [23] statically detects inter-component control and data flows among multiple component's from either the same or different apps. Since µTent's objective is to prevent Intent-based attacks by creating an Intent library we do not perform static analysis like the above models; however µTent's dynamic mechanism for adding contract to the Intent helps to enable data integrity and to control accessibility and visibility of the data. ComDroid [24] analyzes the DEX files to detect Intent-based attacks such as Intent spoofing, UIR and Intent vulnerabilities by performing flow-sensitive and intra-procedural static analysis. CHEX [25] approaches the detection problems such as leak of private data or component vulnerability as data-flow problem that helps to identify the potential risk from the existence of flow pattern. AnFlo [26] proposes a mechanism to detect anomalous sensitive data flows in Android apps by grouping trusted apps based on the categories identified from the apps functional descriptions. TERMINATOR [27] analyzes the permission misuse by considering the order of events that require the time of an event and the security vulnerability in the Android's custom permission model. In µTent every ICC call must have a valid capability to access the Intent data, thereby the flow graph varies in par with the users permission decision for that application.

Policies & Filters: Kirin [28] detects the malicious behavior by certifying an app using kirin security rules at install time. DREBIN [29] proposed a method for detection of Android malware by uses dynamic inferring of security patterns rather than manually crafted patterns [28]. TaintDroid [30], a data-leak detection tool, suggested a fine-grained tracking of private information and how it is actually used. µTent's context-based privacy using dynamic contracts was inspired from the above works; during ICC every µTent is controlled by a unique Trusted Computing Base (TCB) that is created per object. Permission re-delegation introduces a risk when an application with permissions performs a privileged task for an application without permissions. The permission re-delegation paper [3] proposed IPC inspection at OS level, that reduces a deputy's (application with permissions) privileges after receiving communication from a malware application (app without privileges). Consequently, the Android's permission system can deny a privileged API call from the deputy if any application in the chain of influence lacks the appropriate permission(s). DroidCap [31] uses Binder handles as capability tokens by associating Android's permissions to the handle. DroidCap facilitates privilege separation between app's component's through compartmentalization of component's into a logical app with a subset of privileges. Maxoid [32] confines the delegates from leaking secrets by creating different views of the initiator's state. The delegates may update initiator private state or public state which the initiator can selectively commit or discard to prevent unwanted modifications by delegates. This may not be directly applied to Intent data protection, since Intent communication creates a uni-directional data distribution, where the shared Intent data cannot be controlled by the initiator unless the data is represented as a URI. Aquifer [33] defines UI workflow policies where each participating application define its export list, a required list, and a workflow filter. Aquifier tracks the workflow policies to control the interactions across different applications who are participating to complete the task. Similarly, in µTent we compare the capability of an application before allowing the receiver to access the Intent data, since Intent's are initiated within the receiver's context, we provide an additional encryption mechanism to limit the receiver from accessing the data through reflection.

Encryption & Key Distribution: Srivatsa, et al. [6] provided routable attribute protection using content-based routing where the publisher uses event key K(e) to encrypt, and subscriber with filter (f) uses key K(f) to decrypt, where K(f) is capability of a subscriber and hence treated equally. On the other hand, the µTent handles subscriber's based on the capability and the Arc contract, rather based on subscription.

6. Conclusion

We have presented µTent, the secure and pluggable Android Intent Library written in Java, to securely exchange intent data across component's during ICC. µTent is the first holistic model to provide the dynamic control of intents inside the subscriber's context. It differs from the earlier model in terms of providing dynamic encryption, ownership based key distribution, and verification of the receiver's context using user defined security contracts and receiver's capability. Thereby with µTent, we enable application component's to sensibly retain the control of their intent data after it has been shared with the receiver during ICC.

References

[1] "Vulnerability Details : CVE-2018-9489.." `https://www.cvedetails.com/cve/CVE-2018-9489`.

[2] Y. K. Lee, J. andg Bang, G. Safi, A. Shahbazian, Y. Zhao, and N. Medvidovic., "A SEALANT for Inter-app Security Holes in Android.," in *Proceedings of the 39th ICSE*, pp. 312–323, 2017.

[3] F. A.P., W. H.J., M. A., H. S., and C. E., "Permission Re-Delegation: Attacks and Defenses.," in *USENIX Symposium*, p. 22, 2011.

[4] M. Hammad, J. Garcia, and S. Malek, "Self-Protection of Android Systems from Inter-Component Communication Attacks.," in *33rd ACM/IEEE ASE'18*, p. 726–737, 2018.

[5] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. d'Amorim, , and M. D. Ernst., "Static analysis of implicit control flow: Resolving Java reflection and Android intents.," in *Proceedings of 30th ASE'15*, p. 669–679, 2015.

[6] M. Srivatsa, L. Liu, and A. Iyengar., "EventGuard: A System Architecture for Securing Publish-Subscribe Networks.," in *ACM TOCS, vol. 29*, 2011.

[7] A. Feizollah, N. B. Anuar, R. Salleh, G. Suarez-Tangil, and S. Furnell., "AndroDialysis: Analysis of Android Intent Effectiveness in Malware Detection.," in *Elsevier Computers & Security*, pp. 121–134, 2017.

[8] M. Heiderich, J. Schwenk, T. Frosch, J. Magazinius, and E. Z. Yang., "mXSS attacks: attacking well-secured web-applications by using innerHTML mutations.," in *ACM SIGSAC CCS'13*, pp. 777–788, 2013.

[9] C. M. Stone, T. Chothia, and F. D. Garcia., "Spinner: Semi-Automatic Detection of Pinning without Hostname Verification.," in *33rd ACSAC'17*, pp. 176–188, 2017.

[10] "Man-in-the-Middle Flaw in Major Banking, VPN Apps Exposes Millions.." `https://www.darkreading.com/mobile/man-in-the-middle-flaw-in-major-banking-vpn-apps-exposes-millions/d/d-id/1330586`.

[11] A. Nadkarni, B. Andow, W. Enck, and S. Jha, "Practical DIFC Enforcement on Android," in *25th USENIX'16*, pp. 1119–1136, 2016.

[12] D. Clarke and S. Drossopoulou., "Ownership, encapsulation and the disjointness of type and effect.," in *17th ACM OOPSLA'02*, pp. 292–310, 2002.

[13] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T.Leavens, K. R. M. Leino, and E. Poll, "An Overview of JML Tools and Applications.," in *FMICS'03*, pp. 75–91, 2003.

[14] "ICC-Bench: Benchmark apps for static analyzing inter-component data leakage problem of Android apps.." `https://github.com/fgwei/ICC-Bench`.

[15] "DroidBench 2.0: A micro-benchmark suite to assess the stability of taint-analysis tools for Android.." `https://github.com/secure-software-engineering/DroidBench`.

[16] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi., "XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks.," in *Technische Universität Darmstadt, TR-2011-04*, 2011.

[17] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry., "Towards Taming Privilege-Escalation Attacks on Android.," in *NDSS Symposium*, 2012.

[18] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun., "Analysis of the communication between colluding applications on modern smartphones.," in *Proceedings of the 28th ACSAC'12*, pp. 51–60, 2012.

[19] H. Okhravi, S. Bak, and S. T. King., "Design, Implementation and Evaluation of Covert Channel Attacks.," in *IEEE International Conference on Technologies for Homeland Security (HST), Waltham, MA*, pp. 481–487, 2010.

[20] H. Ritzdorf, "Analyzing Covert Channels on Mobile Devices.," in *Master Thesis. ETH Zürich, Department of Computer Science*, 2012.

[21] "Culebra: Generating Ready-to-Execute Scripts for Black box testing.." `https://github.com/dtmilano/AndroidViewClient/wiki/culebra`.

[22] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel., "FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps.," in *PLDI'14*, pp. 259–269, 2014.

[23] F. Wei, S. R. X. O., and Robby., "Amandroid: A Precise and General Inter-Component Data Flow Analysis Framework for Security Vetting of Android Apps.," in *Proceedings of ACM CCS'14*, pp. 1329–1341, 2014.

[24] E. Chin, A. Porter, F. K. Greenwood, and D. Wagner., "Analyzing Inter-Application Communication in Android.," in *Proceedings of the 9th MobiSys*, pp. 239–252, 2011.

[25] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: statically vetting Android apps for component hijacking vulnerabilities.," in *Proceedings of the ACM CCS*, p. 229–240, 2018.

[26] B. F. Demissie, M. Ceccato, and L. K. Shar., "AnFlo: detecting anomalous sensitive information Flows in Android apps.," in *Proceedings of MOBILESoft*, p. 24–34, 2018.

[27] A. Sadeghi, R. Jabbarvand, N. Ghorbani, H. Bagheri, and S. Malek., "A temporal permission analysis and enforcement framework for Android.," in *Proceedings of the 40th ICSE*, pp. 846–857, 2018.

[28] W. Enck, M. Ongtang, and P. McDaniel, "On Lightweight Mobile Phone Application Certification.," in *Proceedings of the ACM CCS*, p. 235–245, 2009.

[29] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket.," in *NDSS'14*, pp. 23–26, 2014.

[30] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. Mcdaniel, , and A. N. Sheth., "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones.," in *9th USENIX (OSDI'10)*, pp. 393–407, 2010.

[31] A. Dawoud and S. Bugiel., "DroidCap: OS Support for Capability-based Permissions in Android.," in *NDSS Symposium*, 2019.

[32] Y. Xu and E. Witchel., "Maxoid: Transparently confining mobile applications with custom views of state," in *EuroSys'15*, pp. 1–16, 2015.

[33] L. Jia, J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, and Y. Miyake., "Run-time enforcement of information-flow properties on Android," in *European Symposium on Research in Computer Security*, pp. 775–792, 2013.