

RESEARCH ARTICLE

TySA: Enforcing Security Policies for Safeguarding Against Permission-Induced Attacks in Android Applications

XINWEN HU 

Hunan Normal University, Changsha, Hunan 410081, China

e-mail: huxinwen@hunnu.edu.cn

This work was supported in part by the Young Scientists Fund of the National Natural Science Foundation of China under Grant 62402179, in part by the Natural Science Foundation of Hunan Province of China under Grant 2024JJ6321, and in part by Hunan Normal University Talent Special Project under Grant 2023053003.

ABSTRACT Android applications (apps) are ubiquitous in complex environments. Although permission-based access control mechanism in Android apps can ensure the proper access of information to a certain extent, it cannot enforce security policies on information propagation, which may cause permission-induced attacks. Our work is motivated by the problem of enforcing security policies on explicit and implicit information flows under the premise of the permission checks/requests at runtime in the Inter-Component Communication (ICC). To this end, we design a formal calculus for reasoning operations (e.g., permission checks and permission requests at runtime) and interactions (e.g., ICC) in Android apps. In addition, we introduce a novel type system based on the proposed formal calculus for checking secure information flow to prevent permission-induced attacks in Android apps. A soundness theorem of this type system is also proved with respect to non-interference property in Android apps. Finally, we realize a prototype Tysa based on the K-framework (\mathbb{K}) and illustrate its effectiveness.

INDEX TERMS Android apps, permission-induced attacks, type systems, soundness proofs, K-framework.


I. INTRODUCTION

Android apps, managing personal data, interacting on-line to share information and accessing sensitive services [1], have become ubiquitous and exponentially increased in number recently [2]. According to the latest statistics [3], [4], the number of Android apps in the Android Google Play Store so far has exceeded 3 million and is growing at an average rate of more than 30,000 apps per month. As apps become more and more sophisticated in function, ensuring their security has become a pressing concern [5], [6], [7]. Although permission-based access control is one of the major Android security mechanisms to prevent apps from privacy data leakage or trusted data tampering to a certain extent [8], it is intriguing that a large proportion of apps are still extremely vulnerable to adversarial attacks, especially

permission/induced attacks involving multiple apps [9], [10], [11], [12], [13], [14], [15], [16], [17], [18].

To tackle this problem, many dynamic and static analysis techniques of program analysis have been proposed for Android apps [19]. In the dynamic analysis methods, the dynamic behaviors (system calls, API calls, etc.) and responses of the apps are analyzed via inputting a set of required test cases during the execution of the apps [5], [19], [20], [21], [22], [23], [24], [25]. In the static analysis methods, apps can be extracted for some patterns (e.g., permissions, intents, data flow, control flow), without being executed [5], [16], [19], [20], [26], [27], [28], [29], [30]. Both methods have their own strengths and weaknesses. But in terms of program analysis, as the number of Android apps being analyzed increases, the costs of program analysis methods increase exponentially [5], [20].

As a formal method to verify the type constraints of the program, type checking can not only complement program analysis to tackle the scalability issue, but also

The associate editor coordinating the review of this manuscript and approving it for publication was Leandros Maglaras .

enforce information flow policy on information propagation to develop Android apps in security critical scenarios [5], [9], [30], [31], [32], [33], [34], [35]. However, with the development of the Android system, the existing type checker cannot automatically enforce security policies on the information flows (explicit or implicit) under the premise of the permission checks/requests at runtime in the ICC, so as to prevent permission/induced attacks among multiple apps [9], [31], [32], [33], [34], [35].

The aforementioned challenges underline the importance of developing a sound and automatic type checker for Android apps. To summarize, we made the following contributions:

- We design a formal language including syntax and semantics for Android apps that can particularly reason about ICC commands and permission checks/requests at runtime.
- We present a lightweight type system for Android apps based on the formal language and prove soundness of the type system with respect to the non/interference property in apps.
- We implement a prototype of TySA based on the \mathbb{K} [36], [37] (which has inherent advantages for defining the type system) and demonstrate the effectiveness of our methodology.

The next section motivates our approach by four permission/induced attacks. In Sect. III, we present a core language as a simplified abstract programming model for security-critical Android apps. Sect. IV introduces a sound security type system for Android apps. In Sect. V, we present implementation details of the prototype Tysa. In Sect. VI, the related work and some discussions are given. We present the conclusion in Sect. VII.

II. MOTIVATING EXAMPLES

Permission/based access control, including permission check and permission request, is the key security mechanism in Android apps [2], [9], [31], [38]. The permission check, enforced by the API *checkPermission*, controls which calling apps/components can invoke the called app/component where the permission check is located, and the permission request, enforced by the API *requestPermission*, controls which called apps/components can be invoked by the calling app/component where the permission request is located [38], [39]. In this paper, we use *caller* to refer to the calling component in the calling app, and use *callee* to refer to the called component in the called app.

Consider the pseudo/code shown in Figure 1. It consists of a component *getContactNo_Activity*, which can get contact number if the requested permission (READ_CONTACT) is granted successfully at runtime (line 6). In addition, as a *callee*, this component also checks whether *caller* is granted with permission READ_CONTACT successfully at runtime to access the

```

1  String getContactNo_Activity (String name) {
2  String number;
3  /* this component is now guarded with READ_CONTACT
   permission*/
4  if (checkPermission(READ_CONTACT) ==
   PERMISSION_GRANTED)
5  /* this component is now granted with READ_CONTACT
   permission*/
6  if (requestPermission(READ_CONTACT) ==
   PERMISSION_GRANTED)
7  number = xH;
8  ...
9  }

```

FIGURE 1. Sample code for getting contact number with permission check and permission request at runtime.

protected data in *callee* (line 4). That is, this component is currently guarded and granted by READ_CONTACT.

To some extent, permission/based access control can prevent leakage of sensitive information or tampering with trusted information on a single Android app, but it has no restriction on information propagation. In other words, permission/based access control does not involve the enforcement of secure information flow policies [1]. However, its combination with typing techniques may avoid information leakage and tampering due to the collusion of multiple Android apps. Starting from Android 6.0 (API level 23), Android apps need to request dangerous permissions at runtime, and allow users to grant/revoke dangerous permissions at any time [40]. This change results in the traditional type system not being applicable to the permission/based commands.

This section describes four kinds of permission/induced attacks caused by the collusion of multi-Android apps, which are identified in prior research [2], to motivate the problem. In this paper, we introduce these attacks based on two Android apps: *app₁* and *app₂*, and we name the component *comp* declared in the app in the form of *app.comp*. The declaration of *app.comp* includes the following parts: the type¹ of the component, the commands of the component, the input parameters and the return variable of the component. e.g., we declare *app.comp* of *Activity* type with command *c*, input parameter *par_{in}* and return variable *par_{out}* as follows:

```

activity app.comp (parin)
  {letvar parout := 0 in{c return parout;}}

```

Based on the assumption-guarantee heuristic in the literature [41], we also introduce four annotations for expressing permission checks and permission requests at runtime in Android apps. e.g., (**chk**(*p*, **t**)) represents “check permission *p* at runtime successfully” and (**chk**(*p*, **f**)) represents “fail to check permission *p* at runtime”. Similarly, (**req**(*p*, **t**)) represents “request permission *p* at runtime successfully” and (**req**(*p*, **f**)) represents “fail to request permission *p* at runtime”. In our examples, we use *x_H*

¹We use **activity**, **service**, **receiver** and **provider** to represent *Activity*, *Service*, *BroadCastReceiver* and *ContentProvider*, respectively.

to denote a sensitive variable and use x_U to represent an untrusted variable. We also assume that $x_H \neq 1$ and $x_U \neq 1$.

```

1  activity app1.comp1 (x1){
2    /*app1.comp1 is now granted without p1*/
3    (req(p1, f))
4    x1:=1;
5    call(app2.comp1, x1);
6  }

```

(a) Sample code for app₁

```

1  activity app2.comp1 (x2){
2    /*app2.comp1 is now guarded without p1*/
3    (chk(p1, f))
4    /*app2.comp1 is now granted with p1*/
5    (req(p1, t))
6    if(x2==1)
7      bind(app2.comp2, x2);
8    else
9      skip;
10 }
11 service app2.comp2 (x3){
12  /*app2.comp2 is now guarded with p1*/
13  (chk(p1, t))
14  /*app2.comp2 is now granted with p1*/
15  (req(p1, t))
16  if(x3==1)
17    /*out(xH) operation is now guarded with p1*/
18    out(xH);
19  else
20    skip;
21 }

```

(b) Sample code for app₂

FIGURE 2. Sample code for privilege escalation attack.

A. PRIVILEGE ESCALATION ATTACK

Privilege escalation attack, resulting in malicious component granted with low-level permission set can directly or indirectly execute the high-level operation, always occurs in the following scenario: A caller granted with low-level permission set is not restricted to invoke an exposed component (i.e., a component is guarded without any permissions) to indirectly invoke operations of callee guarded with high-level permission set in a different app, without requesting for corresponding permissions successfully at runtime [2], [11], [12], [13], [14], [18], [38].

Sample code for privilege escalation attack, consisting of 3 components app₁.comp₁, app₂.comp₁ and app₂.comp₂, is shown as Figure 2. In subfigure 2b, app₂.comp₁ receives parameter x_2 and sends it to app₂.comp₂, which depends on the content of x_2 to decide whether to send out the variable x_H with high security level or not. Here we use **out**(x) command to represent an output operation that sends the value of x out of apps. And in order to express the ICC [16], we introduce different invocation forms for different component types. e.g., if the caller invokes the callee of the **service** type in the invocation, the ICC command in caller adopts the form of **bind**(callee, x), where x is the variable that is sent to callee. Similarly, for the callee of **activity** or **receiver** type, **call**(callee, x) or **send**(callee, x) form is used, respectively. **provider**-type components are slightly different from

the other three types of components. They are kind of component that can encapsulate data to help an app manage access to data stored by itself [42]. We use **query**(callee, x) and **update**(callee, x) to represent reading from and writing to the **provider**-type component, respectively.

From the perspective of permission checks and requests, because app₂.comp₁ requests permission p_1 successfully, there is no information leakage when app₂.comp₁ invokes operation **out**(x_H) of app₂.comp₂ guarded by p_1 . Privilege escalation will not occur if there only has app₂. However, as shown in subfigure 2a, app₁.comp₁ fails to request p_1 and still invokes app₂.comp₁ guarded without p_1 to indirectly invoke **out**(x_H), which leads to leakage of high-level operation.

B. PARAMETER LAUNDERING ATTACK

Parameter laundering attack, laundering the high-level data with less privileged protection and further causing data leakage, has a common attack scenario: A caller granted with high-level permission set is not restricted to send high-level data to an exposed callee and retrieve this data without any protections [2], [9].

```

1  activity app1.comp2 (x1){
2    /*app1.comp2 is now guarded with p2*/
3    (chk(p2, t))
4    /*app1.comp2 is now granted with p2*/
5    (req(p2, t))
6    x1:=bind(app2.comp3, xH);
7  }

```

(a) Sample code for app₁

```

1  service app2.comp3 (x2){
2    letvar x := 0 in {
3    /*app2.comp3 is now guarded without p2*/
4    (chk(p2, f))
5    /*app2.comp3 is now granted without p2*/
6    (req(p2, f))
7    x:=x2;
8    return x;
9  }
10 }

```

(b) Sample code for app₂

FIGURE 3. Sample code for parameter laundering attack.

Sample code for parameter laundering attack, consisting of app₁.comp₂ and app₂.comp₃, is shown as Figure 3. In subfigure 3b, app₂.comp₃ receives parameter x_2 and returns it.

From the perspective of permission checks and requests, app₂.comp₃ is not guarded or granted by p_2 and does not handle any high-level data or operations, so app₂.comp₃ does not result in information disclosure. As shown in subfigure 3a, app₁.comp₂ is guarded and granted with p_2 , which enables it to handle high-level x_H . However, app₁.comp₂ sends x_H to app₂.comp₃ and retrieves it without p_2 protection, resulting in x_H can be accessible to any component that has been granted with arbitrary permissions.

C. PASSIVE CONTENT LEAKAGE ATTACK

Passive content leakage attack results in information leakage mainly due to the unprotected **provider**-type component in the vulnerable app [15]. It has a common attack scenario as follows: A caller granted with high-level permission set is not restricted to update high-level data to a **provider**-type component guarded without any permissions, which may passively leaks various types of sensitive in-app data [2].

```

1  activity app1.comp3 (x1){
2      /*app1.comp3 is now granted with p3*/
3      (req(p3,t))
4      update(app1.comp4, xH);
5  }
6  provider app1.comp4 (x2){
7      letvar x := 0 in {
8          /*app1.comp4 is now guarded without p3*/
9          (chk(p3,f))
10         /*app1.comp4 is now granted without p3*/
11         (req(p3,f))
12         if(x2==1)
13             x:=get(DB);
14         else
15             put(x2,DB);
16         return x;
17     }
18 }

```

(a) Sample code for app₁

```

1  activity app2.comp4 (x1){
2      /*app2.comp4 is now granted without p3*/
3      (req(p3,f))
4      x1:=query(app1.comp4, 1);
5  }

```

(b) Sample code for app₂

FIGURE 4. Sample code for passive content leakage attack.

Sample code in Figure 4, consisting of three components, shows an example of passive content leakage. In subfigure 4a, as a **provider**-type component, app₁.comp₄ can store input parameter x₂ in database (i.e., DB) or fetch data from DB based on the value of x₂. Here we use **get**(DB) to specify a get operation that a **provider**-type component fetches data from DB, and use **put**(x, DB) to specify a put operation that a **provider**-type component stores x in DB. Thus, app₁.comp₃ updates x_H to app₁.comp₄, which stores x_H in DB.

From the perspective of permission checks and requests, app₁.comp₄ is not guarded or granted by p₃ and does not handle any high-level data/operations, so app₁.comp₄ does not result in information disclosure. However, the following collusion of two apps can lead to information leakage: app₁.comp₃ updates x_H to app₁.comp₄, and app₂.comp₄ queries data from app₁.comp₄ (as shown in subfigure 4b), which results in x_H can be accessible to any component that has been granted with arbitrary permissions.

D. PASSIVE CONTENT POLLUTION ATTACK

The main reason for information pollution caused by passive content pollution attack is the unprotected **provider** in the vulnerable app [15]. There is a common attack scenario: A caller granted with low-level permission set is not

restricted to query data from a **provider**-type component guarded without any permissions, which may inadvertently manipulates certain trusted in-app settings or configurations and subsequently causes serious system-wide side effects [2].

In Figure 5, sample code of passive content pollution is given. It consists of three components app₁.comp₅, app₁.comp₆ and app₂.comp₅. In subfigure 5a, app₁.comp₅ queries data from app₁.comp₆, which fetches data from DB.

From the perspective of permission checks and requests, app₁.comp₆ is not guarded or granted by p₄ and does not handle any untrusted (high-level)² data or operations, so app₁ does not result in information tampering. However, the following collusion of two apps can lead to information tampering: app₂.comp₅ updates x_U to app₁.comp₆ (as shown in subfigure 5b) and app₁.comp₅ queries data from app₁.comp₆, which results in x_U tampering with trusted in-app settings or configurations in app₁.comp₅.

```

1  activity app1.comp5 (x1){
2      /*app1.comp5 is now granted without p4*/
3      (req(p4,f))
4      x1:=query(app1.comp6, 1);
5  }
6  provider app1.comp6 (x2){
7      letvar x := 0 in {
8          /*app1.comp6 is now guarded without p4*/
9          (chk(p4,f))
10         /*app1.comp6 is now granted without p4*/
11         (req(p4,f))
12         if(x2==1)
13             x:=get(DB);
14         else
15             put(x2,DB);
16         return x;
17     }
18 }

```

(a) Sample code for app₁

```

1  activity app2.comp5 (x1){
2      /*app2.comp5 is now granted with p4*/
3      (req(p4,t))
4      update(app1.comp6, xU);
5  }

```

(b) Sample code for app₂

FIGURE 5. Sample code for passive content pollution attack.

In order to prevent these permission/induced attacks in Android apps, we not only need to avoid information disclosure and tampering in the process of information release with the aid of permission/based access control, but also need to utilize type system to enforce the security information flow policies on the information propagations. Therefore, we propose a sound type system TySA for permission/based Android apps. The rest of this paper focuses on the examples in this subsection to illustrate the effectiveness of TySA.

²With reference to the integrity property settings in literature [2] and [43], the security level of untrusted information in this paper is higher than that of trusted information.

III. CORE LANGUAGE

In the Android model, an app is always composed of a finite set of components, which can be instantiated and invoked by other apps/components [44]. Since the communications between components are mostly done by Binder Inter-Process Communications (IPC) [45] and shared states between apps are mostly absent [2], [9], we only consider sequential execution of apps in this paper. For the ICC within and across apps, permission checks restrict the ability to access `callee`, and permission requests restrict the ability of the `caller` to access certain high-level data/operations in the `callee` [38]. Therefore, we not only need to model permission checks to describe which permissions a component is currently guarded with, but also need to model permission requests to denote which permissions a component is currently granted with.

In literature [9], authors use a conditional command with a conditional branching induced by permission testing to indicate permission check. But they have not yet considered the permission-based access control at runtime. Inspired by the assumption/guarantee heuristic of the literature [41], we introduce permission-based annotations to model permission checks and permission requests at runtime to specify their modification of the set of permissions used to guard the component and the set of permissions used to grant the component, respectively. This approach can not only improve the treatment of ICC (i.e., method calls) [41], but also improve the reasoning about how permission checks and permission requests at runtime affect the security of information in the apps when ICC occurs within or across apps, leading for instance to the possibility of a precise and security type system for permission-based Android apps.

A. A SIMPLE SYNTAX OF ANDROID APPS

We now proceed to design a core formal language to describe Android apps, which have many language features and library dependencies [44]. To narrow our focus, as shown in our examples in Section II, we mainly extend the core syntax of the language considered in literature [35] with the ICC commands and the annotations for permission checks and permission requests at runtime in this paper.

In our language, a system, denoted by \mathcal{S} , consists of a finite set of apps. We use app_i to represent the i -th app in \mathcal{S} . A component $comp_j$ defined in app_i is denoted by $app_i.comp_j$, which has one of the following `cTypes` (component types): **activity**, **service**, **receiver** and **provider**. We denote with \mathcal{C} the finite set consisting of all components in \mathcal{S} , and denote with P the finite set containing all permissions in \mathcal{S} . Each component is guarded with a dynamic set of permissions (denoted as P_{gu}) drawn from P due to the permission check at runtime. Similarly, each component is granted with a dynamic set of permissions (denoted as P_{gr}) drawn from P due to the permission request at runtime. The power set of P is written as \mathcal{P} .

The phrases in our language are either expressions e or commands c :

$$(Phrases) \quad ph := e \mid c$$

The syntax of the expressions is given below:

$$(Expressions) \quad e := v \mid x \mid e_1 \mathbf{op} e_2$$

where v denotes an integer value,³ x denotes a variable and $e_1 \mathbf{op} e_2$ denotes an arithmetic expression that join subexpressions through binary operator **op** (e.g., “=” or “<=”).

The commands are given in the following syntax:

$$\begin{aligned} (Commands) \quad c &:= \text{simpleCmd}; \mid \text{blockCmd} \mid c_1 c_2 \\ \text{simpleCmd} &:= x := e \mid \mathbf{skip} \mid \text{ICC} \mid \mathbf{out}(e) \\ &\quad \mid x_1 := \mathbf{get}(x_2) \mid \mathbf{put}(x_1, x_2) \\ \text{ICC} &:= \text{Invoke}(app_i.comp_j, \bar{e}) \\ &\quad \mid x := \text{Invoke}(app_i.comp_j, \bar{e}) \\ \text{Invoke} &:= \mathbf{call} \mid \mathbf{bind} \mid \mathbf{send} \mid \mathbf{query} \mid \mathbf{update} \\ \text{blockCmd} &:= \mathbf{letvar} \ x := e \ \mathbf{in} \ c \\ &\quad \mid \mathbf{if} \ (e) \ c_1 \ \mathbf{else} \ c_2 \\ &\quad \mid \mathbf{while} \ (e) \ \mathbf{do} \ c \\ \text{ann} &:= \mathbf{ann} \ c \\ \text{ann} &:= (\mathbf{chk} \ (p, \mathbf{t})) \mid (\mathbf{chk} \ (p, \mathbf{f})) \\ &\quad \mid (\mathbf{req} \ (p, \mathbf{t})) \mid (\mathbf{req} \ (p, \mathbf{f})) \\ &\quad \mid \text{ann}_1 \ \text{ann}_2 \end{aligned}$$

where $p \in P$. `simpleCmd` stands for the simple command. The first two constructs of `simpleCmd` are the $x := e$ assignment and the **skip** command, where **skip** is also used as a symbol to define formal semantics of our language. The command `ICC` is a inter-component communication command, which has two different constructs: 1) the `Invoke` ($app_i.comp_j, \bar{e}$) command denotes an invocation from `caller` to `callee`, where $app_i.comp_j$ is the `callee` and \bar{e} represents the parameters passed to the `callee`. 2) the $x := \text{Invoke}(app_i.comp_j, \bar{e})$ command denotes an assignment, where x receives the value returned by an invocation. The command **out**(e) denotes an out operation, where e is the value sent out of the Android apps (e.g., via the Internet). The command $x_1 := \mathbf{get}(x_2)$ denotes an assignment, where **get**(x_2) represents a get operation that the component (of **provider** type) fetches data from its database named x_2 and x_1 represents a variable that receives the value returned by the get operation. The command **put**(x_1, x_2) denotes a put operation that the component (of **provider** type) stores the value of x_1 to its database named x_2 . `blockCmd` represents the block command, which has three common constructs: local variable definition, if-conditional and while-loop commands. “ $c_1 c_2$ ” represents the sequential composition.

³This paper only considers integers, because boolean values can be encoded as 0 to represent false and non-zero values to represent true.

The syntax also supports annotations to specify which permissions are checked to guard the component and which permissions are granted to the component by the user at runtime. Annotations ann are enclosed in brackets “(...)”, which affect the permission context of the component and do not contribute to the runtime behavior of the Android app. For instance, the annotation $\mathbf{chk}(p, \mathbf{t})$ indicates that the component is currently guarded with permission p , and the annotation $\mathbf{chk}(p, \mathbf{f})$ indicates that the component is currently guarded without p . Similarly, the two annotations $\mathbf{req}(p, \mathbf{t})$ and $\mathbf{req}(p, \mathbf{f})$ indicate that the component is currently granted with and without p , respectively. The command c , annotated with (multiple of) these annotations, is denoted by $\text{ann}c$.

A component declaration has the following syntax:

```
COMP := cType appi.compj ( $\overline{\text{par}}_{\text{in}}$ )
      {letvar parout := 0 in {annc return parout; }}
cType := activity | service | receiver | provider
```

where $cType$ is the type of the component, $\overline{\text{par}}_{\text{in}}$ are the parameters for starting the component, $\text{ann}c$ is the command with annotations and par_{out} is the return variable of the component. Notice that when a component does not return a value, we simply omit its return variable in this declaration. In this paper, we assume that all variables in $COMP$ are introduced by **letvar** or $\{\overline{\text{par}}_{\text{in}}, \text{par}_{\text{out}}\}$.

B. EXECUTION MODEL

As shown in the component declaration, we declare commands with annotations for each component to describe the permission context (state) at runtime. This approach borrows from the idea of assumption-guarantee modes in concurrent programs, where a variable is assigned modes representing how it may be accessed [41]. Similar to a thread assigning multiple modes to a variable, we allow multiple annotations to be assigned to each component, which forms dynamic P_{gu} and P_{gr} for each component.

In this paper, we only consider sequential execution of the Android apps in the permission state \mathbb{P} , which consists of two parts: the permission state \mathbb{P}_{gu} and the permission state \mathbb{P}_{gr} . The permission state \mathbb{P}_{gu} models a snapshot of P_{gu} for each component during the execution of Android apps, while the permission state \mathbb{P}_{gr} models a snapshot of P_{gr} for each component during the execution of Android apps. The definition of the permission state \mathbb{P} is as follows:

Definition 1: A permission state \mathbb{P} is a two-tuple $\langle \mathbb{P}_{gu}, \mathbb{P}_{gr} \rangle$, where permission state \mathbb{P}_{gu} and permission state \mathbb{P}_{gr} both are finite mappings from \mathcal{C} to \mathcal{P} . We use the notation $[\text{comp}_1 \mapsto P_{gu_1}, \dots, \text{comp}_n \mapsto P_{gu_n}]$ to denote a permission state \mathbb{P}_{gu} mapping each $\text{comp}_i \in \mathcal{C}$ to its own $\mathbb{P}_{gu}[\text{comp}_i] = P_{gu_i} \in \mathcal{P}$. Similarly, we use the notation $[\text{comp}_1 \mapsto P_{gr_1}, \dots, \text{comp}_n \mapsto P_{gr_n}]$ to denote a permission state \mathbb{P}_{gr} mapping each $\text{comp}_i \in \mathcal{C}$ to its own $\mathbb{P}_{gr}[\text{comp}_i] = P_{gr_i} \in \mathcal{P}$.

We assume that no annotation is assigned to components at the beginning of the Android app’s execution, but that all

annotations are assigned dynamically at runtime. Thus, the initial permission state \mathbb{P}_{gu_0} is defined by $\mathbb{P}_{gu}[\text{comp}] = \{\}$ for all $\text{comp} \in \mathcal{C}$, and so does the initial \mathbb{P}_{gr_0} .

Except the permission state, the execution of Android apps will also affect the value of each variable. The definition of memory state is given as follows:

Definition 2: A memory state m is a finite mapping from variables to values. We use the notation $[x_1 \mapsto v_1, \dots, x_n \mapsto v_n]$ to denote an memory state mapping x_i to its value v_i , where $1 \leq i \leq n$.

Based on the above definitions, a configuration of the execution model in system \mathcal{S} is given as a unique four-tuple

$$\langle c, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle$$

that models a snapshot during the execution of the \mathcal{S} , where $c \in \text{Commands}$ is the command that remains to be executed in \mathcal{S} , $\langle \mathbb{P}_{gu}, \mathbb{P}_{gr} \rangle$ (i.e., \mathbb{P}) is the permission state modeling the current permission context of the \mathcal{S} , and m is the memory state modeling the current memory of \mathcal{S} .

C. OPERATIONAL SEMANTICS

The soundness of a type system is established with respect to a formal semantics of the language [43]. We now present a formal operational semantics for our language based on the following assumptions: 1) Component definitions are stored in a table **CD** indexed by component names. 2) If a **provider**-type component has a database named DB , then there is a variable denoted as $\llbracket DB \rrbracket$, indicating an element that can be fetched or written in a first-in-first-out order in the database. Notice that this variable can only be manipulated via the **provider**-type component.

There are two evaluation judgements for our language.

- 1) The evaluation judgement of the expressions is given in the form of $\langle e, m \rangle \Downarrow v$, where expression e evaluates to value v in the memory state m .
- 2) The operational semantics for the commands is defined by a transition relation on configurations of the execution model, denoted \rightsquigarrow . i.e., the evaluation judgement of the commands takes the form

$$\text{caller} \vdash \langle c, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle \rightsquigarrow \langle \mathbf{skip}, \mathbb{P}'_{gu}, \mathbb{P}'_{gr}, m' \rangle,$$

where $\langle \mathbb{P}_{gu}, \mathbb{P}_{gr} \rangle$ and m are the permission state and the memory state before the execution of the command c , and $\langle \mathbb{P}'_{gu}, \mathbb{P}'_{gr} \rangle$ and m' are the permission state and the memory state after the execution of c . **skip**, representing the end of a command execution, cannot make a transition in any configuration. Here caller refers to the component where the command c is located. When caller has no ambiguity, we omit the caller in the operational semantics to save space.

As shown in Figure 6, the operational semantics for expressions such as rules (**E-Val**), (**E-Var**), and (**E-Arith**) are given. Intuitively, the operational semantics of expressions does not cause any side effects.

The operational semantics for commands is given in Figure 6, most of which are straightforward. We mainly

$$\begin{array}{c}
\frac{}{\langle v, m \rangle \Downarrow v} \text{ (E-Val)} \quad \frac{}{\langle x, m \rangle \Downarrow m(x)} \text{ (E-Var)} \quad \frac{\langle e_1, m \rangle \Downarrow v_1 \quad \langle e_2, m \rangle \Downarrow v_2}{\langle e_1 \text{ op } e_2, m \rangle \Downarrow v_1 \text{ op } v_2} \text{ (E-Arith)} \quad \frac{\langle e, m \rangle \Downarrow v}{\langle x := e, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m[x \mapsto v] \rangle} \text{ (E-Assign)} \\
\frac{}{\langle \bar{e}, m \rangle \Downarrow \bar{v}} \text{ (E-Sk)} \quad \frac{\langle \text{skip } c, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle \rightsquigarrow \langle c, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle}{\text{CD}(\text{callee}) = \text{activity } \text{app}_i.\text{comp}_j(\overline{\text{par}}_{in})\{\text{annc}\} \quad (\mathbb{P}'_{gu}[\text{callee}] \cup \mathbb{P}'_{gr}[\text{callee}] \subseteq \mathbb{P}_{gr}[\text{caller}])} \text{ (E-IccC)} \\
\frac{\langle \bar{e}, m \rangle \Downarrow \bar{v} \quad \text{callee} \vdash \langle \text{annc}, \mathbb{P}_{gu}, \mathbb{P}_{gr}, [\overline{\text{par}}_{in} \mapsto \bar{v}] \rangle \rightsquigarrow \langle c, \mathbb{P}_{gu}, \mathbb{P}_{gr}, [\overline{\text{par}}_{in} \mapsto \bar{v}] \rangle}{\text{caller} \vdash \langle \text{call}(\text{callee}, \bar{e}), \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}'_{gu}, \mathbb{P}'_{gr}, m \rangle} \text{ (E-IccC)} \\
\frac{\text{CD}(\text{callee}) = \text{provider } \text{app}_i.\text{comp}_j(\overline{\text{par}}_{in})\{\text{letvar } \text{par}_{out} := 0 \text{ in } \{\text{annc return } \text{par}_{out};\}\} \quad \text{callee} \vdash \langle \text{annc}, \mathbb{P}_{gu}, \mathbb{P}_{gr}, [\overline{\text{par}}_{in} \mapsto \bar{v}, \text{par}_{out} \mapsto 0] \rangle \rightsquigarrow \langle c, \mathbb{P}'_{gu}, \mathbb{P}'_{gr}, [\overline{\text{par}}_{in} \mapsto \bar{v}, \text{par}_{out} \mapsto 0] \rangle}{\frac{(\mathbb{P}'_{gu}[\text{callee}] \cup \mathbb{P}'_{gr}[\text{callee}] \subseteq \mathbb{P}_{gr}[\text{caller}])}{\langle \bar{e}, m \rangle \Downarrow \bar{v} \quad \text{callee} \vdash \langle c, \mathbb{P}'_{gu}, \mathbb{P}'_{gr}, [\overline{\text{par}}_{in} \mapsto \bar{v}, \text{par}_{out} \mapsto 0] \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}'_{gu}, \mathbb{P}'_{gr}, m' \rangle} \text{ (E=-IccQ)} \\
\text{caller} \vdash \langle x := \text{query}(\text{callee}, \bar{e}), \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}'_{gu}, \mathbb{P}'_{gr}, m[x \mapsto m'(\text{par}_{out})] \rangle} \\
\frac{\langle [x_2], m \rangle \Downarrow v}{\langle x_1 := \text{get}(x_2), \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m[x_1 \mapsto v] \rangle} \text{ (E-Get)} \quad \frac{\langle x_1, m \rangle \Downarrow v}{\langle \text{put}(x_1, x_2), \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m[[x_2] \mapsto v] \rangle} \text{ (E-Put)} \\
\frac{}{\langle \text{out}(e), \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle} \text{ (E-Out)} \quad \frac{\langle e, m \rangle \Downarrow v \quad \langle c, \mathbb{P}_{gu}, \mathbb{P}_{gr}, \text{mem}[x \mapsto v] \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m' \rangle}{\langle \text{letvar } x := e \text{ in } c, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m' - x \rangle} \text{ (E-LetVar)} \\
\frac{\langle e, m \rangle \Downarrow v \quad v \neq 0 \quad \langle c_1, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m' \rangle}{\langle \text{if}(e) c_1 \text{ else } c_2, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m' \rangle} \text{ (E-IfT)} \quad \frac{\langle e, m \rangle \Downarrow v \quad v = 0 \quad \langle c_2, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m' \rangle}{\langle \text{if}(e) c_1 \text{ else } c_2, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m' \rangle} \text{ (E-IfF)} \\
\frac{\langle e, m \rangle \Downarrow v \quad v \neq 0 \quad \langle c_1, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m' \rangle \quad \langle \text{while}(e) \text{ do } c, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m' \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m'' \rangle}{\langle \text{while}(e) \text{ do } c, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m'' \rangle} \text{ (E-WhT)} \\
\frac{\langle e, m \rangle \Downarrow v \quad v = 0 \quad \langle c_1, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m' \rangle \quad \langle c_2, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m' \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m'' \rangle}{\langle \text{while}(e) \text{ do } c, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m'' \rangle} \text{ (E-WhF)} \quad \frac{\langle c_1 c_2, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m' \rangle}{\langle c_1 c_2, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m' \rangle} \text{ (E-Seq)} \\
\frac{\mathbb{P}'_{gu}[\text{caller}] = \mathbb{P}_{gu}[\text{caller}] \cup \{p\}}{\text{caller} \vdash \langle (\text{chk}(p, t)) c, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle \rightsquigarrow \langle c, \mathbb{P}'_{gu}, \mathbb{P}_{gr}, m \rangle} \text{ (E-AnCT)} \quad \frac{\mathbb{P}'_{gu}[\text{caller}] = \mathbb{P}_{gu}[\text{caller}] \setminus \{p\}}{\text{caller} \vdash \langle (\text{chk}(p, f)) c, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle \rightsquigarrow \langle c, \mathbb{P}'_{gu}, \mathbb{P}_{gr}, m \rangle} \text{ (E-AnCF)} \\
\frac{\mathbb{P}'_{gr}[\text{caller}] = \mathbb{P}_{gr}[\text{caller}] \cup \{p\}}{\text{caller} \vdash \langle (\text{req}(p, t)) c, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle \rightsquigarrow \langle c, \mathbb{P}_{gu}, \mathbb{P}'_{gr}, m \rangle} \text{ (E-AnRT)} \quad \frac{\mathbb{P}'_{gr}[\text{caller}] = \mathbb{P}_{gr}[\text{caller}] \setminus \{p\}}{\text{caller} \vdash \langle (\text{req}(p, f)) c, \mathbb{P}_{gu}, \mathbb{P}_{gr}, m \rangle \rightsquigarrow \langle c, \mathbb{P}_{gu}, \mathbb{P}'_{gr}, m \rangle} \text{ (E-AnRF)}
\end{array}$$

FIGURE 6. Formal operational semantics of the core language in Android apps.

explain the operational semantics of the `ann`, `ICC`, `x1 := get(x2)`, `put(x1, x2)` and `out(e)` constructs. Rule (E-An_{CT}) captures the semantics of `(chk(p, t))`: `p` is successfully checked at runtime. Thus, this rule adds permission `p` to P_{gu} of `caller`. Rule (E-An_{CF}) captures the semantics of `(chk(p, f))`: checking `p` at runtime fails. Thus, this rule deletes permission `p` from P_{gu} of the `caller`. Similarly, rule (E-An_{RT}) captures the semantics of `(req(p, t))`: `p` is successfully requested at runtime. Thus, this rule adds permission `p` to P_{gr} of the `caller`. Rule (E-An_{RF}) captures the semantics of `(req(p, f))`: requesting `p` at runtime fails. Thus, this rule deletes permission `p` from P_{gr} of the `caller`. These four rules, analyzing in which permission context the command `c` in the `caller` should be executed, only affect the permission state and neither the memory state nor the runtime behavior of the Android apps.

Invoke(`appi.compj, \bar{e}`) has five different evaluation rules depending on the type of `callee` being invoked: such as rules (E-Icc_C), (E-Icc_B), (E-Icc_S), (E-Icc_Q) and (E-Icc_U) (The subscript of these rules indicates the first letter of the Invoke type in uppercase). e.g., rule (E-Icc_C) shows the semantics of `call(appi.compj, \bar{e})`, and the `appi.compj` (i.e. `callee`) is of type **activity**. This rule says that if the `callee` has `annc`, then an updated permission state obtained by `annc` in the context of the `callee` is also the

updated permission state for the invocation. Notice that we need to make sure that both the permissions `callee` was granted and guarded are contained in the set of permissions granted to `caller`, otherwise this will result in privilege escalation. Moreover, intuitively, since this invocation does not change any variable's value in `caller`, it may only affect \mathbb{P} . Similarly, `x := Invoke(appi.compj, \bar{e})` also has five different evaluation rules: (E=-Icc_C), (E=-Icc_B), (E=-Icc_S), (E=-Icc_Q) and (E=-Icc_U). Take rule (E=-Icc_Q) as an example: It specifies the operational semantics of `x := query(callee, \bar{e})`, and the `callee` is of type **provider**. The primary difference between `x := query(callee, \bar{e})` and `query(callee, \bar{e})` is that the former changes the value of `x` in the `caller`. Thus, in addition to updating the permission state like the rule (E-Icc_Q), `x` is also updated with the return variable of the `callee`.

Rule (E-Get) captures the operational semantics of `x1 := get(x2)`, indicating that if the variable `[x2]` has the value `v` in the memory of the database `x2`, then `x1` is assigned with `v`. Rule (E-Put) shows the operational semantics of `put(x1, x2)`, specifying that if the variable `x1` has the value `v` in the memory, then the variable `[x2]` in the database `x2` is assigned with `v`. Rule (E-Out) indicates the operational semantics of `out(e)`, showing that `e` is sent out of the Android apps.

IV. A SOUND SECURITY TYPE SYSTEM

In the information flow type system, it is very common to use the lattice model [46] to encode security levels [9], [43]. According to different security properties, different lattice models can be used to design corresponding security information flow policies [43]. For instance, security information flow policies can be defined by a lattice structure (SL, \leq) , where SL is a finite set of security levels partially ordered by \leq . SL may include \perp (Low), H (High) for secrecy property, or may include T (Trusted), U (Untrusted) for integrity property, where $\perp \leq H$ and $T \leq U$ [43].

The lattice model makes it possible and intuitive to enforce security information flow policies on explicit/implicit flows via simple attribute grammars [43]. For example, we can use simply attribute constraints: “ y ’s security level is less than or equal to x ’s” to enforce security information flow policy: “No high-level information can interference low-level information” on explicit information flows, such as $x := y$, or implicit information flows, like **if** ($y == 0$) **then** $x := 0$ **else** $x := 1$. In this paper, we simply use the security levels to set the security types to construct security type system. The security types of our phrases are defined as follows:

Definition 3: A base security type τ is a security level and forms a security lattice, with the partial order \leq . We denote with \mathcal{T} the set of base types, where the highest base type is labeled as \top and the lowest base type is labeled as \perp . Given two base types $\tau_1, \tau_2 \in \mathcal{T}$, $\tau_1 \leq \tau_2$ means that τ_1 is lower than or equal to τ_2 , $\tau_1 \not\leq \tau_2$ means that τ_1 is higher than τ_2 , $\tau_1 = \tau_2$ means that the two types are equal, $\tau_1 \sqcup \tau_2$ is the least upper bound of τ_1 and τ_2 , and $\tau_1 \sqcap \tau_2$ is the greatest lower bound of τ_1 and τ_2 .

Since the security type system is a logical system consisting of a set of security inference rules and axioms that derive security typing judgements for various constructs of a program (e.g., expressions, commands and components in this paper) based on the security types of subprograms (e.g., variables in this paper) [43], [47], we not only define the security typing environment for mapping variables to security types, but also define the security type of the component in the Android apps, as follows:

Definition 4: A typing environment Γ is a finite mapping from variables to \mathcal{T} . We use the notation $[x_1 : \tau_1, \dots, x_n : \tau_n]$ to enumerate a typing environment mapping x_i to its security type τ_i , where $1 \leq i \leq n$.

Definition 5: A component type has the form $(\overline{\tau_{in}}, \tau_{gu}, \tau_{gr}, \tau_{out})$, where $\overline{\tau_{in}} = (\tau_{in_1}, \dots, \tau_{in_m})$ ($1 \leq j \leq m$), and $\tau_{gu}, \tau_{gr}, \tau_{out}, \tau_{in_j} \in \mathcal{T}$. $\overline{\tau_{in}}$ is the security type tuple of the input parameters of the component, τ_{gu} is the security type that guards the component, τ_{gr} is the security type that grants the component, and τ_{out} is the security type of the return variable of the component. For the components without return variable, the type of the return variable can be omitted, i.e., $(\overline{\tau_{in}}, \tau_{gu}, \tau_{gr}, ())$.

A. SECURITY TYPING RULES

We now present a type system for our language based on the following assumptions: 1) The security levels are stored in a table \mathbf{PL} indexed by the set of permissions. In particular, we assume that the security level corresponding to the set without permission is the lowest, i.e., $\mathbf{PL}() = \perp$. 2) The component types are stored in a table \mathbf{CT} indexed by component names.

In TySA, security types of expressions (commands, components, resp.) may be altered depending on the permission context. For instance, when an expression is used in the execution context where a permission request has been performed, its type may be adjusted depending on whether the permission p is requested successfully. For another example, when the component is used in the execution context of checking permissions, its type may be adjusted according to whether the permission p is checked successfully. Such an adjustment, called a *promotion* or a *demotion*, can be used with respect to not only a base type, but also a typing environment. The corresponding definitions are as follows.

Definition 6: Given a permission p , the promotion and demotion of a base type τ with respect to p are:

$$\tau \oplus p = \tau \sqcup \mathbf{PL}(p) \quad (\text{promotion})$$

$$\tau \ominus p = \tau \sqcap \mathbf{PL}(p) \quad (\text{demotion})$$

Definition 7: Given a typing environment Γ , its promotion and demotion along a permission p are typing environments $\Gamma \oplus p$ and $\Gamma \ominus p$, such that $(\Gamma \oplus p)(x) = \Gamma(x) \oplus p$, $(\Gamma \ominus p)(x) = \Gamma(x) \ominus p$ for every $x \in \text{dom}(\Gamma)$.

There are three typing judgements in our type system explained below.

- 1) Expression typing has the form: $\Gamma \vdash e : \tau$, which denotes that the e has a type at most τ in Γ .
- 2) Command typing has the form: $\Gamma; (\tau_{gu}^{\text{caller}}, \tau_{gr}^{\text{caller}}) \vdash c : \tau$, which denotes that the command c can write to variables with type at least τ in Γ . Moreover, it specifies that c is currently executed in *caller*, which has types $\tau_{gu}^{\text{caller}}$ and $\tau_{gr}^{\text{caller}}$ representing the security type that guards the grants the *caller*, respectively.
- 3) Component typing has the form: $\vdash \text{COMP} : (\overline{\tau_{in}}, \tau_{gu}, \tau_{gr}, \tau_{out})$. Intuitively, this denotes the input parameters and return variable of the component have a type at most $\overline{\tau_{in}}$ and τ_{out} in Γ , respectively. Meanwhile, it means that the component can be invoked from other components that are granted at least τ_{gu} , and that the component can invoke other components that are guarded up to τ_{gr} .

The typing rules are shown in Figure 7, most of which are straightforward [9], [43], [48]. We explain the typing rules for the `ann`, `ICC`, `$x_1 := \text{get}(x_2)$` , `$\text{put}(x_1, x_2)$` , `$\text{out}(e)$` and `COMP` constructs.

We first give the typing rules for `ann` as follows: Rule $(\mathbf{T-An}_{CT})$ types c in Γ for a successful permission check on p , and promotes $\tau_{gu}^{\text{caller}}$ with respect to p for the *caller*. On the contrary, rule $(\mathbf{T-An}_{CF})$ types c in Γ for a failed

$$\begin{array}{c}
\frac{}{\Gamma \vdash v : \tau} \text{(T-Val)} \quad \frac{}{\Gamma \vdash x : \Gamma(x)} \text{(T-Var)} \quad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \text{ op } e_2 : \tau} \text{(T-Arith)} \quad \frac{\Gamma \vdash e : \tau_2 \quad \tau_2 \leq \tau_1}{\Gamma \vdash e : \tau_1} \text{(T-Sub}_E\text{)} \quad \frac{\Gamma \vdash c_1 : \tau \quad \Gamma \vdash c_2 : \tau}{\Gamma \vdash c_1 \ c_2 : \tau} \text{(T-Seq)} \\
\frac{\Gamma \vdash e : \Gamma(x)}{\Gamma \vdash x := e : \Gamma(x)} \text{(T-Assign)} \quad \frac{}{\Gamma \vdash \text{skip} : \top} \text{(T-Sk)} \quad \frac{\text{CT}(\text{app}_i.\text{comp}_j) = (\overline{\tau}_{\text{in}}, \tau_{\text{gu}}, \tau_{\text{gr}}, \tau_{\text{out}}) \quad \Gamma \vdash \overline{e} : \overline{\tau}_{\text{in}} \quad (\tau_{\text{gu}} \sqcup \tau_{\text{gr}}) \leq \tau_{\text{gr}}^{\text{caller}}}{\Gamma; (\tau_{\text{gu}}^{\text{caller}}, \tau_{\text{gr}}^{\text{caller}}) \vdash \text{call}(\text{app}_i.\text{comp}_j, \overline{e}) : \top} \text{(T-Icc}_C\text{)} \\
\frac{\Gamma \vdash c : \tau_2 \quad \tau_1 \leq \tau_2}{\Gamma \vdash c : \tau_1} \text{(T-Sub}_C\text{)} \quad \frac{\text{CT}(\text{app}_i.\text{comp}_j) = (\overline{\tau}_{\text{in}}, \tau_{\text{gu}}, \tau_{\text{gr}}, \tau_{\text{out}}) \quad \Gamma \vdash \overline{e} : \overline{\tau}_{\text{in}} \quad (\tau_{\text{gu}} \sqcup \tau_{\text{gr}}) \leq \tau_{\text{gr}}^{\text{caller}} \quad \tau_{\text{out}} \leq \Gamma(x)}{\Gamma; (\tau_{\text{gu}}^{\text{caller}}, \tau_{\text{gr}}^{\text{caller}}) \vdash x := \text{query}(\text{app}_i.\text{comp}_j, \overline{e}) : \Gamma(x)} \text{(T=-Icc}_O\text{)} \\
\frac{\Gamma \vdash \llbracket x_2 \rrbracket : \Gamma(x_1)}{\Gamma \vdash x_1 := \text{get}(x_2) : \Gamma(x_1)} \text{(T-Get)} \quad \frac{\Gamma \vdash x_1 : \Gamma(\llbracket x_2 \rrbracket)}{\Gamma \vdash \text{put}(x_1, x_2) : \Gamma(\llbracket x_2 \rrbracket)} \text{(T-Put)} \quad \frac{\Gamma \vdash e : \tau_{\text{gr}}}{\Gamma; (\tau_{\text{gu}}, \tau_{\text{gr}}) \vdash \text{out}(e) : \top} \text{(T-Out)} \quad \frac{\Gamma \vdash e : \tau_e \quad \Gamma[x : \tau_e] \vdash c : \tau}{\Gamma \vdash \text{letvar } x := e \text{ in } c : \tau} \text{(T-LetVar)} \\
\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash c_1 : \tau \quad \Gamma \vdash c_2 : \tau}{\Gamma \vdash \text{if}(e) \ c_1 \ \text{else} \ c_2 : \tau} \text{(T-If)} \quad \frac{\Gamma; (\tau_{\text{gu}}^{\text{caller}} \oplus p, \tau_{\text{gr}}^{\text{caller}}) \vdash c : \tau}{\Gamma; (\tau_{\text{gu}}^{\text{caller}} \oplus p, \tau_{\text{gr}}^{\text{caller}}) \vdash (\text{chk}(p, t)) \ c : \tau} \text{(T-An}_{CR}\text{)} \quad \frac{\Gamma; (\tau_{\text{gu}}^{\text{caller}} \oplus p, \tau_{\text{gr}}^{\text{caller}}) \vdash c : \tau}{\Gamma; (\tau_{\text{gu}}^{\text{caller}} \oplus p, \tau_{\text{gr}}^{\text{caller}}) \vdash (\text{chk}(p, f)) \ c : \tau} \text{(T-An}_{CF}\text{)} \\
\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash c : \tau}{\Gamma \vdash \text{while}(e) \ \text{do} \ c : \tau} \text{(T-Wh)} \quad \frac{\Gamma \oplus p; (\tau_{\text{gu}}^{\text{caller}}, \tau_{\text{gr}}^{\text{caller}} \oplus p) \vdash c : \tau}{\Gamma \oplus p; (\tau_{\text{gu}}^{\text{caller}}, \tau_{\text{gr}}^{\text{caller}} \oplus p) \vdash (\text{req}(p, t)) \ c : \tau} \text{(T-An}_{RT}\text{)} \quad \frac{\Gamma \oplus p; (\tau_{\text{gu}}^{\text{caller}}, \tau_{\text{gr}}^{\text{caller}} \oplus p) \vdash c : \tau}{\Gamma \oplus p; (\tau_{\text{gu}}^{\text{caller}}, \tau_{\text{gr}}^{\text{caller}} \oplus p) \vdash (\text{req}(p, f)) \ c : \tau} \text{(T-An}_{RF}\text{)} \\
\frac{[\overline{\text{par}}_{\text{in}} : \overline{\tau}_{\text{in}}, \overline{\text{par}}_{\text{out}} : \tau_{\text{out}}]; (\tau_{\text{gu}}, \tau_{\text{gr}}) \vdash \text{ann}_c : \tau}{\vdash \text{cType } \text{app}_i.\text{comp}_j(\overline{\text{par}}_{\text{in}})\{\text{letvar } \overline{\text{par}}_{\text{out}} := 0 \text{ in } \{\text{ann}_c \ \text{return } \overline{\text{par}}_{\text{out}};\}\} : (\overline{\tau}_{\text{in}}, \tau_{\text{gu}}, \tau_{\text{gr}}, \tau_{\text{out}})} \text{(T-Comp)}
\end{array}$$

FIGURE 7. Security typing rules of the core language in Android apps.

permission check on p , and demotes $\tau_{\text{gu}}^{\text{caller}}$ with respect to p for the caller. Different from the permission check, when the permission request modifies the permissions in P_{gr} of the caller, it will essentially change the security type of the information in the entire system observed by the caller. Therefore, rule (T-An_{RT}) types c in a promoted Γ for a successful permission request on p , and promotes $\tau_{\text{gr}}^{\text{caller}}$ with respect to p for the caller. On the contrary, rule (T-An_{RF}) types c in a demoted Γ for a failed permission request on p , and demotes $\tau_{\text{gr}}^{\text{caller}}$ with respect to p for the caller.

Invoke ($\text{app}_i.\text{comp}_j, \overline{e}$) has different typing rules depending on the type of component being invoked: such as rules (T-Icc_C), (T-Icc_B), (T-Icc_S), (T-Icc_O) and (T-Icc_I) (The subscript of these rules indicates the first letter of the Invoke type in uppercase). For instance, rule (T-Icc_C) shows the typing rule of $\text{call}(\text{callee}, \overline{e})$, and the callee is of type **activity**. Since $\text{call}(\text{callee}, \overline{e})$ does not modify variables in the caller, rule (T-Icc_C) types it as \top . Here we assume that the security type of the callee is checked beforehand and stored in the CT table. Since only the arguments \overline{e} are passed between caller and callee, \overline{e} should be typed as $\overline{\tau}_{\text{in}}$. To avoid privilege escalation, it is worth noting that both τ_{gu} and τ_{gr} in the security type of callee should be dominated by $\tau_{\text{gu}}^{\text{caller}}$. Similarly, $x := \text{Invoke}(\text{app}_i.\text{comp}_j, \overline{e})$ also has five different typing rules: (T=-Icc_C), (T=-Icc_B), (T=-Icc_S), (T=-Icc_O) and (T=-Icc_I). Take rule (T=-Icc_O) as an example to illustrate: It specifies the typing rule of $x := \text{query}(\text{callee}, \overline{e})$, and the callee is of type **provider**. Since the primary difference between $x := \text{query}(\text{callee}, \overline{e})$ and $\text{query}(\text{callee}, \overline{e})$ is that the former changes the value of x in the caller, the rule (T=-Icc_O) types it as $\Gamma(x)$. In addition, the return variable of callee is also passed back to caller, so τ_{out} should be dominated by $\Gamma(x)$.

In both $x_1 := \text{get}(x_2)$ and $\text{put}(x_1, x_2)$, the essence is still an assignment command. Therefore, rules (T-Get)

and (T-Put) are similar to the rule (T-Assign). Rule (T-Out) types the $\text{out}(e)$ as \top because it does not change any variables in caller. It is worth noting that in order to prevent the caller from sending data that it cannot access to the outside world, the e should be typed as τ_{gr} . Rule (T-Comp) is obtained naturally based on component definition and component type.

To avoid the label creeping [49], we also introduce subtyping rules (T-Sub_E) and (T-Sub_C) [9], [41], [43] for expressions and commands, respectively. In our rules, $(\tau_{\text{gu}}^{\text{caller}}, \tau_{\text{gr}}^{\text{caller}})$ is omitted in command typings when caller has no ambiguity, such as rules (T-Seq), (T-Assign), etc.

In TySA, to facilitate typing derivation, we define the well-typed property of Android apps as follows:

Definition 8: Suppose \mathcal{S} is a system consists of finite Android apps, and let **CD**, **CT** and **PL** be the component declaration table, component type table and permission level table of \mathcal{S} , respectively. Then \mathcal{S} is well-typed iff every component $\text{app}_i.\text{comp}_j$ in \mathcal{S} has component type, i.e., $\vdash \text{CD}(\text{app}_i.\text{comp}_j) : \text{CT}(\text{app}_i.\text{comp}_j)$ is derivable.

In order to prevent a potential implicit flow via the privilege escalation attack, it is essential that both τ_{gu} and τ_{gr} of the callee are constrained to be less than or equal to $\tau_{\text{gr}}^{\text{caller}}$ in typing rules for the ICC construct. To illustrate the necessity of this constraint, consider the following alternative rule (T-Icc'_C). If only τ_{gu} of the callee is constrained to be less than or equal to $\tau_{\text{gr}}^{\text{caller}}$, then the typing rule (T-Icc'_C) for Invoke (callee, \overline{e}) construct, where the callee is of type activity, is as follows:

$$\frac{\text{CT}(\text{callee}) = (\overline{\tau}_{\text{in}}, \tau_{\text{gu}}, \tau_{\text{gr}}, \tau_{\text{out}}) \quad \Gamma \vdash \overline{e} : \overline{\tau}_{\text{in}} \quad \tau_{\text{gu}} \leq \tau_{\text{gr}}^{\text{caller}}}{\Gamma; (\tau_{\text{gu}}^{\text{caller}}, \tau_{\text{gr}}^{\text{caller}}) \vdash \text{call}(\text{callee}, \overline{e}) : \top} \text{(T-Icc}'_C\text{)}$$

Let us consider the example in Figure 2. Let $P = \{p_1\}$, and we assume that $\text{PL}(p_1) = \text{H}$ and $\text{PL}() = \text{L}$, where H and L are

the top and bottom type respectively. If we use rule (**T-Icc'**) instead of (**T-Icc**), sample code in Figure 2 is well-typed and we can get the following component types:

$$\mathbf{CT} := \begin{cases} \text{app}_1.\text{comp}_1 \mapsto (L, L, L, ()) \\ \text{app}_2.\text{comp}_1 \mapsto (H, L, H, ()) \\ \text{app}_2.\text{comp}_2 \mapsto (H, H, H, ()) \end{cases}$$

The error in this type derivation is that the rule (**T-Icc'**) does not consider that the `callee` can still be granted permissions at runtime without permission protection, so that the `caller` can access data or operations that it has no permission to access through the exposed `callee`, which is also the main cause of privilege escalation attacks. In the rule (**T-Icc**), $\tau_{\text{gu}} \sqcup \tau_{\text{gr}} \leq \tau_{\text{gr}}^{\text{caller}}$ can ensure that the permissions granted to the `callee` are not passed to the `caller`, which can avoid privilege escalation attacks.

With the correct typing rule for ICC, typing derivations for the sample code in Figure 2 are shown in Figure 8, where the red part indicates that the typing judgement does not follow the rule (**T-Icc**), meaning that the type of component `app1.comp1` cannot be obtained and the sample code is not well-typed. The typing derivations for examples in Figure 3, 4, and 5 can be found in Appendix.

B. NON-INTERFERENCE AND SOUNDNESS

Proving the soundness of a security type system is essentially proving that the system enforcing non/interference property, which can be defined informally as “the value of the high-level variable do not interfere with (affect) the value of the low-level variable” [43], [47], [50].

Based on the semantics in Section III-C, the non/interference property of the command (Definition 9), the non/interference property of the component (Definition 10) and the non/interference property of the system (Definition 11) are defined as follows.

Definition 9: A command or an annotated command $(\text{ann})\ c$ executed in `caller` is non/interference iff for all $m_1, m_2, \mathbb{P}_{\text{gu}}, \mathbb{P}_{\text{gr}}, \Gamma, \tau$, suppose that $\Gamma; (\tau_{\text{gu}}^{\text{caller}}, \tau_{\text{gr}}^{\text{caller}}) \vdash (\text{ann})\ c : \tau_c$, $\text{caller} \vdash \langle (\text{ann})\ c, \mathbb{P}_{\text{gu}}, \mathbb{P}_{\text{gr}}, m_1 \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}'_{\text{gu}}, \mathbb{P}'_{\text{gr}}, m_1' \rangle$, $\text{caller} \vdash \langle (\text{ann})\ c, \mathbb{P}_{\text{gu}}, \mathbb{P}_{\text{gr}}, m_2 \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}'_{\text{gu}}, \mathbb{P}'_{\text{gr}}, m_2' \rangle$, $\text{dom}(m_1) = \text{dom}(m_2) = \text{dom}(\Gamma)$, and $m_1(e_p) = m_2(e_p)$ for all e_p such that $\Gamma(e_p) \leq \tau$, then $m_1'(e_q) = m_2'(e_q)$ for all e_q such that $\Gamma(e_q) \leq \tau$.

Definition 10: A component `caller` defined as follows:

```
cType caller ( $\overline{\text{par}}_{\text{in}}$ )
{letvar  $\text{par}_{\text{out}} := 0$  in {annc return  $\text{par}_{\text{out}}$ ;}}
```

with component type $(\overline{\tau}_{\text{in}}, \tau_{\text{gu}}, \tau_{\text{gr}}, \tau_{\text{out}})$ is non/interference iff for all $m_1, m_2, \mathbb{P}_{\text{gu}}, \mathbb{P}_{\text{gr}}, \tau$, suppose that $\Gamma = [\overline{\text{par}}_{\text{in}} : \overline{\tau}_{\text{in}}, \text{par}_{\text{out}} : \tau_{\text{out}}]$, $\text{caller} \vdash \langle \text{annc}, \mathbb{P}_{\text{gu}}, \mathbb{P}_{\text{gr}}, m_1 \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}'_{\text{gu}}, \mathbb{P}'_{\text{gr}}, m_1' \rangle$, $\text{caller} \vdash \langle \text{annc}, \mathbb{P}_{\text{gu}}, \mathbb{P}_{\text{gr}}, m_2 \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}'_{\text{gu}}, \mathbb{P}'_{\text{gr}}, m_2' \rangle$, and $m_1(e_p) = m_2(e_p)$ for all e_p such that $\Gamma(e_p) \leq \tau$, then $m_1'(\text{par}_{\text{out}}) = m_2'(\text{par}_{\text{out}})$.

Definition 11: A system \mathcal{S} is non/interference iff all components in \mathcal{S} are non/interference.

Based on these definitions, we establish the following lemmas and theorem for proving that the TySA is sound. Lemma 1 and Lemma 2 are two properties of the type system based on the semantics of Android apps in Section III-C. Lemma 1 applies to expressions, and it indicates that no expression has a type higher than τ in e can be read when e has a type at most τ in Γ . Lemma 2 applies to commands, and it specifies that no expression has a type lower than τ in $(\text{ann})\ c$ can be written when $(\text{ann})\ c$ has a type at least τ in Γ . Lemma 1 and Lemma 2 can be used for the proof of the Lemma 3, where soundness is formulated as the non/interference property of command. Finally, based on these lemmas, Theorem 1 is given to show that the well-typed program satisfies the non/interference property of the system.

Lemma 1 (No Read Up): If $\Gamma \vdash e : \tau$, then for every e_i in e ($1 \leq i \leq n$), $\Gamma(e_i) \leq \tau$.

Due to the limited space, we hereby give the proof of lemma 1 for rule (**T-Arith**).

Proof of Lemma 1: By induction on the structure of e . Suppose $\Gamma \vdash e_1 \ \mathbf{op} \ e_2 : \tau$ by rule (**T-Arith**). Then we have $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$. By induction on e_1 and e_2 , we have $\Gamma \vdash e_{1_{i'}} : \tau$ for every $e_{1_{i'}} (1 \leq i' \leq n_1)$ in e_1 and $\Gamma \vdash e_{2_{i''}} : \tau$ for every $e_{2_{i''}} (1 \leq i'' \leq n_2)$ in e_2 . So $\Gamma(e_i) \leq \tau$ for every $e_i (1 \leq i \leq 2)$ in $e_1 \ \mathbf{op} \ e_2$.

Lemma 2 (No Write Down): If $\Gamma; (\tau_{\text{gu}}^{\text{caller}}, \tau_{\text{gr}}^{\text{caller}}) \vdash (\text{ann})\ c : \tau$, then for every e_i that is assigned in $(\text{ann})\ c$ ($1 \leq i \leq n$), $\tau \leq \Gamma(e_i)$.

Due to the limited space, We hereby give the proof of lemma 2 for rule (**T=-Icc_Q**).

Proof of Lemma 2: By induction on the structure of c . Suppose that $\Gamma; (\tau_{\text{gu}}^{\text{caller}}, \tau_{\text{gr}}^{\text{caller}}) \vdash x := \mathbf{query}(\text{app}_i.\text{comp}_j, \bar{e}) : \Gamma(x) = \tau$ by rule (**T=-Icc_Q**). Then we have $\tau \leq \Gamma(x)$. So for every $e_i (1 \leq i \leq n)$ that is assigned in $x := \mathbf{query}(\text{app}_i.\text{comp}_j, \bar{e})$, we have $\tau \leq \Gamma(e_i)$.

Lemma 3 (Soundness of Command): Suppose that

- 1) $\Gamma; (\tau_{\text{gu}}^{\text{caller}}, \tau_{\text{gr}}^{\text{caller}}) \vdash (\text{ann})\ c : \tau_c$,
- 2) $\text{caller} \vdash \langle (\text{ann})\ c, \mathbb{P}_{\text{gu}}, \mathbb{P}_{\text{gr}}, m_1 \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}'_{\text{gu}}, \mathbb{P}'_{\text{gr}}, m_1' \rangle$,
- 3) $\text{caller} \vdash \langle (\text{ann})\ c, \mathbb{P}_{\text{gu}}, \mathbb{P}_{\text{gr}}, m_2 \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}'_{\text{gu}}, \mathbb{P}'_{\text{gr}}, m_2' \rangle$,
- 4) $\text{dom}(m_1) = \text{dom}(m_2) = \text{dom}(\Gamma)$, and
- 5) $m_1(e_p) = m_2(e_p)$ for all e_p such that $\Gamma(e_p) \leq \tau$.

Then $m_1'(e_q) = m_2'(e_q)$ for all e_q such that $\Gamma(e_q) \leq \tau$.

Due to the limited space, we hereby give the proof of lemma 3 for rule (**T=-Icc_Q**).

Proof of Lemma 3: By induction on the structure of the derivation of $\text{caller} \vdash \langle c, \mathbb{P}_{\text{gu}}, \mathbb{P}_{\text{gr}}, m \rangle \rightsquigarrow \langle \text{skip}, \mathbb{P}'_{\text{gu}}, \mathbb{P}'_{\text{gr}}, m' \rangle$, suppose that $\Gamma; (\tau_{\text{gu}}^{\text{caller}}, \tau_{\text{gr}}^{\text{caller}}) \vdash x := \mathbf{query}(\text{app}_i.\text{comp}_j, \bar{e}) : \Gamma(x) = \tau_c$ by rule (**T=-Icc_Q**).

- 1) If $\tau_c \not\leq \tau$, by lemma 2, for x that is assigned in $x := \mathbf{query}(\text{app}_i.\text{comp}_j, \bar{e})$, we have $\tau_c \leq \Gamma(x)$. Then we have $\Gamma(x) \not\leq \tau$. Then by hypothesis e), we have $m_1'(e_q) = m_2'(e_q)$ for all e_q such that $\Gamma(e_q) \leq \tau$.
- 2) If $\tau_c \leq \tau$, we prove it by the following steps:

$$\begin{array}{c}
\frac{\Gamma_1 \oplus p_1 \vdash x_1 : L \quad \text{CT}(\text{app}_2.\text{comp}_1) = (H, L, H, ()) \quad \Gamma_1 \oplus p_1 \vdash x_1 : L \quad \Gamma_1 \oplus p_1 \vdash x_1 : H \quad (L \cup H = H) \not\leq (\tau_{gr}^{\text{caller}} \oplus p_1 = L)}{\Gamma_1 \oplus p_1 \vdash x_1 := 1 : L} \text{(T-Assign)} \quad \frac{\Gamma_1 \oplus p_1; (L, \tau_{gr}^{\text{caller}} \oplus p_1) \vdash \text{call}(\text{app}_2.\text{comp}_1, x_1) : T \quad \Gamma_1 \oplus p_1; (L, \tau_{gr}^{\text{caller}} \oplus p_1) \vdash \text{call}(\text{app}_2.\text{comp}_1, x_1) : L}{\Gamma_1 \oplus p_1; (L, \tau_{gr}^{\text{caller}} \oplus p_1) \vdash x_1 := 1; \text{call}(\text{app}_2.\text{comp}_1, x_1) : L} \text{(T-Subc)} \quad \frac{\Gamma_1 \oplus p_1; (L, \tau_{gr}^{\text{caller}} \oplus p_1) \vdash x_1 := 1; \text{call}(\text{app}_2.\text{comp}_1, x_1) : L}{[x_1 : L]; (L, L) \vdash (\text{req}(p_1, \mathbf{f}) \quad x_1 := 1; \text{call}(\text{app}_2.\text{comp}_1, x_1) : L)} \text{(T-Angr)} \quad \frac{\Gamma_1 \oplus p_1; (L, \tau_{gr}^{\text{caller}} \oplus p_1) \vdash x_1 := 1; \text{call}(\text{app}_2.\text{comp}_1, x_1) : L}{\vdash \text{activity app}_1.\text{comp}_1(x_1) \{ (\text{req}(p_1, \mathbf{f}) \quad x_1 := 1; \text{call}(\text{app}_2.\text{comp}_1, x_1) : L) \} : (L, L, L, ())} \text{(T-Comp)} \\
\frac{\Gamma_2 \oplus p_1 \vdash x_2 : H \quad \Gamma_2 \oplus p_1 \vdash 1 : H \quad \text{CT}(\text{app}_2.\text{comp}_2) = (H, H, H, ()) \quad \Gamma_2 \oplus p_1 \vdash x_2 : H \quad (H \cup H) \leq (\tau_{gr}^{\text{caller}} \oplus p_1 = H)}{\Gamma_2 \oplus p_1 \vdash (x_2 := 1) : H} \text{(T-Arith)} \quad \frac{\Gamma_2 \oplus p_1; (\tau_{gr}^{\text{caller}} \oplus p_1, \tau_{gr}^{\text{caller}} \oplus p_1) \vdash \text{bind}(\text{app}_2.\text{comp}_2, x_2) : H}{\Gamma_2 \oplus p_1 \vdash \text{skip} : H} \text{(T-Iccg)} \quad \frac{\Gamma_2 \oplus p_1; (\tau_{gr}^{\text{caller}} \oplus p_1, \tau_{gr}^{\text{caller}} \oplus p_1) \vdash \text{if} (x_2 = 1) \text{bind}(\text{app}_2.\text{comp}_2, x_2) ; \text{else skip} : H}{[x_2 : H]; (L, H) \vdash (\text{chk}(p_1, \mathbf{f}) \quad (\text{req}(p_1, \mathbf{t}) \quad \text{if} (x_2 = 1) \text{bind}(\text{app}_2.\text{comp}_2, x_2) ; \text{else skip} : H)} \text{(T-AnCr)} \text{(T-Angr)} \quad \frac{\Gamma_2 \oplus p_1; (\tau_{gr}^{\text{caller}} \oplus p_1, \tau_{gr}^{\text{caller}} \oplus p_1) \vdash \text{if} (x_2 = 1) \text{bind}(\text{app}_2.\text{comp}_2, x_2) ; \text{else skip} : H}{\vdash \text{activity app}_2.\text{comp}_1(x_2) \{ (\text{chk}(p_1, \mathbf{f}) \quad (\text{req}(p_1, \mathbf{t}) \quad \text{if} (x_2 = 1) \text{bind}(\text{app}_2.\text{comp}_2, x_2) ; \text{else skip} : H) \} : (H, L, H, ())} \text{(T-Comp)} \\
\frac{\Gamma_3 \oplus p_1 \vdash x_3 : H \quad \Gamma_3 \oplus p_1 \vdash 1 : H \quad \text{CT}(\text{app}_2.\text{comp}_3) = (H, H, H, ()) \quad \Gamma_3 \oplus p_1 \vdash x_H : H}{\Gamma_3 \oplus p_1 \vdash (x_3 = 1) : H} \text{(T-Arith)} \quad \frac{\Gamma_3 \oplus p_1; (\tau_{gr}^{\text{caller}} \oplus p_1, \tau_{gr}^{\text{caller}} \oplus p_1) \vdash \text{out}(x_H) : H}{\Gamma_3 \oplus p_1 \vdash \text{skip} : H} \text{(T-Out)} \quad \frac{\Gamma_3 \oplus p_1; (\tau_{gr}^{\text{caller}} \oplus p_1, \tau_{gr}^{\text{caller}} \oplus p_1) \vdash \text{if} (x_3 = 1) \text{out}(x_H) ; \text{else skip} : H}{[x_3 : H]; (H, H) \vdash (\text{chk}(p_1, \mathbf{t}) \quad (\text{req}(p_1, \mathbf{t}) \quad \text{if} (x_3 = 1) \text{out}(x_H) ; \text{else skip} : H)} \text{(T-AnCr)} \text{(T-Angr)} \quad \frac{\Gamma_3 \oplus p_1; (\tau_{gr}^{\text{caller}} \oplus p_1, \tau_{gr}^{\text{caller}} \oplus p_1) \vdash \text{if} (x_3 = 1) \text{out}(x_H) ; \text{else skip} : H}{\vdash \text{service app}_2.\text{comp}_2(x_3) \{ (\text{chk}(p_1, \mathbf{t}) \quad (\text{req}(p_1, \mathbf{t}) \quad \text{if} (x_3 = 1) \text{out}(x_H) ; \text{else skip} : H) \} : (H, H, H, ())} \text{(T-Comp)}
\end{array}$$

FIGURE 8. Typing derivation for privilege escalation.

- Since we have $\langle \bar{e}, m_1 \rangle \Downarrow \bar{v}_1$, callee $\vdash \langle c, p'_{gu}, p'_{gr}, [\overline{\text{par}}_{in} \mapsto \bar{v}_1, \text{par}_{out} \mapsto 0] \rangle \rightsquigarrow \langle \text{skip}, p'_{gu}, p'_{gr}, m'_1 \rangle$ and $\langle \bar{e}, m_2 \rangle \Downarrow \bar{v}_2$, callee $\vdash \langle c, p'_{gu}, p'_{gr}, [\overline{\text{par}}_{in} \mapsto \bar{v}_2, \text{par}_{out} \mapsto 0] \rangle \rightsquigarrow \langle \text{skip}, p'_{gu}, p'_{gr}, m'_2 \rangle$ by rule **(E-=-Icc₀)**, we only need to check $[\overline{\text{par}}_{in} \mapsto \bar{v}_1, \text{par}_{out} \mapsto 0](e_k) = [\overline{\text{par}}_{in} \mapsto \bar{v}_2, \text{par}_{out} \mapsto 0](e_k)$ for all e_k such that $\Gamma(e_k) \leq \tau$. Let $\rho_1 = [\overline{\text{par}}_{in} \mapsto \bar{v}_1, \text{par}_{out} \mapsto 0]$ and $\rho_2 = [\overline{\text{par}}_{in} \mapsto \bar{v}_2, \text{par}_{out} \mapsto 0]$, suppose par_u is a variable in $\overline{\text{par}}_{in}$ and $\Gamma(\text{par}_u) \leq \tau$, and suppose $\rho_1(\text{par}_u) = v_u$ and $\rho_2(\text{par}_u) = v'_u$. By lemma 1 and hypothesis e), we have $v_u = v'_u$. Therefore, we have $\rho_1(e_k) = \rho_2(e_k)$ for all e_k such that $\Gamma(e_k) \leq \tau$.
- By applying the outer induction hypothesis, we have $m'_1(e_k) = m'_2(e_k)$ for all e_k such that $\Gamma(e_k) \leq \tau$.
- Since $\tau_{out} \leq \Gamma(x) = \tau_c \leq \tau$, by lemma 1, we have $[x \mapsto m'_1(\text{par}_{out})](e_k) = [x \mapsto m'_2(\text{par}_{out})](e_k)$ for all e_k such that $\Gamma(e_k) \leq \tau$.

By hypothesis e), it is deducible that $m'_1(e_q) = m_1[x \mapsto m'_1(\text{par}_{out})](e_q) = m_2[x \mapsto m'_2(\text{par}_{out})](e_q) = m_2(e_q)$ for all e_q such that $\Gamma(e_q) \leq \tau$.

Theorem 1 (Soundness of System): If a system \mathcal{S} is well-typed, then it is non/interference.

Proof of Theorem 1: The non/interference property of well-typed systems follows from Lemma 3.

V. IMPLEMENTATION

Since the \mathbb{K} [36] is a rewrite-based executable semantic framework, which is suitable for defining type systems via *configurations*, *computations* and *rules* [37], we have implemented the prototype (TySA)⁴ of the TySA based on \mathbb{K} . TySA consists of three parts: 1) Configurations, which organize the program state of Android apps in units called *cells*, which are denoted by “ $\{ \}$ ” and can be nested. 2) Computations, which sequentially the abstract syntax

⁴The TySA can be found in <https://sites.google.com/view/the-type-checker-for-apps>

trees of Android apps into a list of computation tasks, with the list connector “ \curvearrowright ” and the empty computation “ \cdot ”. 3) Rules, which are essentially a set of rewriting rules used to trigger transitions on configurations.

In our actual implementation, the prototype has approximately 2000 lines of K code. For illustrative purposes, we demonstrate the three parts of TySA for the command

$$x := \text{query}(\text{app}_1.\text{comp}_1, y);$$

in Android apps as follows.

Configurations. For each cell in the configuration, the right subscript of the $\{ \}$ indicates the name of the cell, and the content of the $\{ \}$ is the current value of the cell. In \mathbb{K} , a dot followed by any type indicates an empty set of that type [36], [51], [52]. For instance, `.List` represents an empty list, `.Map` represents an empty map. In particular, `.K` represents an empty set of any specific type defined in \mathbb{K} .

The initial configuration (state) of Android apps shown in Figure 9 has three main cells in the whole configuration cell T and they are k , *callstate* and *Apps*. Each cell is initialized with the value wrapped in $\{ \}$. For instance, the cell k is initialized with the source program of Android apps, denoted by `Pgm`. The function of this cell is to record the remaining source code that needs to be analyzed. If all the code is analyzed, only a dot is left in the cell to indicate that the cell is empty. Since the source code of the Android apps in the initial state has not yet been analyzed, we initialize the remaining cells with empty sets.

The cell *callstate* records the *callerApp*, *callerComp*, *callerGuard*, *callerGrant*, *calleeApp*, *calleeComp*, *calleeGuard*, *calleeGrant*, *currentCmd* and *secTypeOfCmd*. To be specific, *currentCmd* records the command currently being analyzed and *secTypeOfCmd* stores its security type. *callerApp*, *callerComp*, *callerGuard* and *callerGrant* keep track of the app, component, P_{gu} and P_{gr} of the caller to which *currentCmd* belongs, respectively. The remaining cells in *callstate* record the related properties of the callee.

In the cell *Apps*, a set of app definitions is stored. Each cell *App* represents an app definition. In *App*, *nameOfApp* records the app name and *secTypeOfApp* stores whether the app is non/interference (distinguish by `Untype` and

Type). Similarly, the cell *Comps* stores a set of component definitions. Each cell *Comp* represents a component definition. For each *Comp* definition, *nameOfComp* records the component name, *typeOfComp* records the type of the component, and *secTypeOfComp* stores whether the component is non/interference. *guardOfComp* (*grantOfComp*) keeps track of the list of permissions in P_{gu} (P_{gr}), and *guard Level* (*grantLevel*) records the security level corresponding to P_{gu} (P_{gr}). In addition, component parameters, including input parameters and return parameter, are recorded in the cell *paraOfComp*. We also store the security typing environment in the cell *secEnvOfVar*.

Computations. First, we give the following simple syntax for type checking command $x := \text{query}(\text{app}_1.\text{comp}_1, y)$; based on the K definition. In K Definition, everything inside double quotes is fixed, and if the code currently being analyzed does not conform to this syntax, the K tool embedded in the \mathbb{K} stops. The main difference between our syntax in Section III-A and these K definitions is that these definitions add the *KResult* definition and the strictness attribute, which is annotated by the keyword **strict**.

```

syntax Cmd ::= SimpleCmd";"
              | #cmd(SimpleCmd)   [strict]
syntax SimpleCmd ::= InvokeCmd
              | AssignCmd
syntax InvokeCmd ::=
              Invoke("Id"."Id"," Id")[strict(4)]
syntax Invoke ::= "call" | "query"
              | "listen" | "send" | "update"
syntax AssignCmd ::= Exp" := " Exp   [strict]
syntax Exp ::= Id | InvokeCmd
syntax SecType ::= "High" | "Low"
              | "Untype" | "Type"
syntax KResult ::= SecType

```

In \mathbb{K} , it is convenient to use *KResult* to distinguish syntactically between unfinished computations and finished computations [36]. That is, the *KResult* definition can tell the K tool which expressions are meant to be results of computations, so that the K tool will not attempt to evaluate them anymore. In our computations, we only focus on the security type, so we define *KResult* as *SecType*. The strictness attribute can state the evaluation strategy of some language constructs, which means that all specified arguments must be evaluated before evaluating the construct itself [36]. For instance, *InvokeCmd* **[strict(4)]** means that the fourth argument *Id*⁵ of *InvokeCmd* need to be evaluated on its security type before evaluating the whole *InvokeCmd* construct. *AssignCmd* **[strict]** means that all the arguments of *AssignCmd* need to be evaluated on its security type before evaluating the whole *AssignCmd* construct.

⁵*Id* term is the built-in identifier in \mathbb{K} .

Based on the K definitions that contain these evaluation policies, the command $x := \text{query}(\text{app}_1.\text{comp}_1, y)$; can be sequentialized in 3 tasks as follows. It means that in order to compute $x := \text{query}(\text{app}_1.\text{comp}_1, y)$; we need to: (1) compute the security type of *y*, (2) compute the security type of *x*, and use the security type of *y* to compute the security type of *query* ($\text{app}_1.\text{comp}_1, \text{secType}(y)$), (3) compute the security type of $\text{secType}(x) := \text{secType}(\text{query}(\text{app}_1.\text{comp}_1, \text{secType}(y)))$.

```

Y
  ↪ x, query(app1.comp1, secType(y))
  ↪ secType(x) := secType(query
    (app1.comp1, secType(y)))

```

Rules. \mathbb{K} rules usually have the rewrite part, and some also have the matching part. The rewriting part divides the cell of the configuration into two parts by a horizontal line, the upper part is rewritten to the lower part when triggered. The matching part is composed of cells, which are the conditions that trigger the rewriting part. In the \mathbb{K} rules, ... represents the content that is not concerned in the cell, and _ stands for anything in the cell but we don't care what it is specifically.

In order to realize the concrete computation of the command $x := \text{query}(\text{app}_1.\text{comp}_1, y)$; we present some assumptions here: 1) The security type of both *x* and *y* is High. 2) The P_{gr} of the caller to which command $x := \text{query}(\text{app}_1.\text{comp}_1, y)$; belongs corresponds to a High security level. 3) The $\text{app}_1.\text{comp}_1$ component is well-typed. 4) The P_{gr} and P_{gu} of the $\text{app}_1.\text{comp}_1$ component both correspond to a Low security level. On the basis of these assumptions, we introduce the following five rules for the command $x := \text{query}(\text{app}_1.\text{comp}_1, y)$;

From the syntax in **Computations**, *InvokeCmd* can be either a *SimpleCmd* or an *Exp*. To make the distinction easier, we add *#cmd(SimpleCmd)* to the syntax. The rule *SimpleCmd* simply rewrites command $x := \text{query}(\text{app}_1.\text{comp}_1, y)$; to *#cmd(x := query (app1.comp1, y))*.

RULE SimpleCmd

$$\left\langle \frac{SC:\text{SimpleCmd}; \dots}{\#cmd(SC)} \dots \right\rangle_k$$

In the rule *Var*, we find the security type of the variable in the corresponding component in the *Apps* cell according to the properties of the caller recorded by the *callstate* cell. Then, we get the security types of *x* and *y*.

RULE Var

$$\left\langle \frac{\frac{I:\text{Id}}{ST:\text{SecType}} \dots}{\{A\} \text{callerApp} \{C\} \text{callerComp} \dots \} \text{callstate}}{\left\langle \left\langle \left\langle \dots \left\langle \dots \left\langle \dots \left\langle \dots \left\langle \dots \left\langle \dots \right\rangle_{Comp} \right\rangle_{App} \right\rangle_{App} \right\rangle_{App} \right\rangle_{App} \right\rangle_{App} \right\rangle_{App} \right\rangle_{App} \right\rangle_{App} \right\rangle_{App} \right\rangle_{App}$$

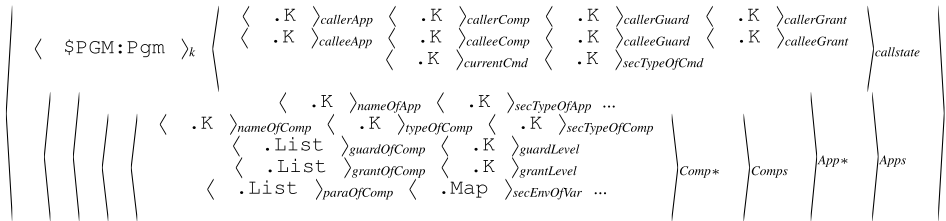


FIGURE 9. Initial Configuration of Android Apps.

According to rule Untype_InvokeCmd, we can calculate the security type of $query(app_1.comp_1, secType(y))$. Because the security type of y is higher than τ_{in} of the callee, $query(app_1.comp_1, secType(y))$ is rewritten with Untype.

$$\begin{array}{c}
 \text{RULE Untype_InvokeCmd} \\
 \left\langle \frac{I: \text{Invoke}(I1:Id, I2:Id, High) \dots}{\text{Untype}} \dots \right\rangle_k \\
 \left\langle \left\langle \begin{array}{l} \{ A \} \text{callerApp} \{ C \} \text{callerComp} \\ \{ High \} \text{callerGrant} \dots \end{array} \right\rangle \text{callstate} \right. \\
 \left. \left\langle \left\langle \frac{-}{I1} \right\rangle \text{calleeApp} \left\langle \frac{-}{I2} \right\rangle \text{calleeComp} \right\rangle \right. \\
 \left. \left\langle \begin{array}{l} \{ I1 \} \text{nameOfApp} \{ Type \} \text{secTypeOfApp} \\ \{ I2 \} \text{nameOfComp} \end{array} \right\rangle \right. \\
 \left. \left\langle \left\langle \begin{array}{l} \{ provider \} \text{typeOfComp} \\ \{ Type \} \text{secTypeOfComp} \end{array} \right\rangle \text{Comp} \dots \right\rangle \right. \\
 \left. \left\langle \left\langle \begin{array}{l} \{ Low \} \text{guardLevel} \\ \{ Low \} \text{grantLevel} \dots \end{array} \right\rangle \right\rangle \right. \\
 \left. \right\rangle \text{App}
 \end{array}$$

the rule Untype_Assign gives one of the rules for an assignment command. When the right side of an assignment command is known as Untype, the command is rewritten as Untype regardless of the security type on the left. Therefore, the security type of the assignment command $secType(x) := secType(query(app_1.comp_1, secType(y)))$ is Untype.

$$\text{RULE Untype_Assign} \\
 \left\langle \frac{- := \text{Untype} \dots}{\text{Untype}} \dots \right\rangle_k$$

For all SimpleCmd with security type Untype, we rewrite it as empty computation, as shown in the rule Untype_#cmd. Moreover, we record Untype in the cell Apps for the corresponding app and component to indicate that neither of them met non/interference property. Therefore, the caller to which command $x := query(app_1.comp_1, y);$ belongs is not well-typed.

$$\begin{array}{c}
 \text{RULE Untype_#cmd} \\
 \left\langle \frac{\#cmd(\text{Untype}) \dots}{\dots} \dots \right\rangle_k \\
 \left\langle \left\langle \begin{array}{l} \{ A \} \text{callerApp} \{ C \} \text{callerComp} \\ \left\langle \frac{-}{\text{Untype}} \right\rangle \text{secTypeOfCmd} \dots \end{array} \right\rangle \text{callstate} \right. \\
 \left. \left\langle \begin{array}{l} \{ A \} \text{nameOfApp} \left\langle \frac{-}{\text{Untype}} \right\rangle \text{secTypeOfApp} \\ \left\langle \begin{array}{l} \{ C \} \text{nameOfComp} \dots \\ \left\langle \frac{-}{\text{Untype}} \right\rangle \text{secTypeOfComp} \end{array} \right\rangle \text{Comp} \dots \end{array} \right\rangle \right. \\
 \left. \right\rangle \text{App}
 \end{array}$$

$$\begin{array}{c}
 \text{CT}(app_2.comp_1) = (L, L, L, L) \quad \Gamma @ p_2 \vdash x_H : H \neq L \quad (L \cup L) \leq (\tau_{in}^{caller} @ p_2 = H) \quad L \leq (\Gamma(x_1) = H) \quad (\text{T-!ecr}) \\
 \frac{\Gamma @ p_2: (\tau_{in}^{caller} @ p_2, \tau_{in}^{callee} @ p_2) \vdash x_1 := \text{bind}(app_2.comp_1, x_H) : H}{[x_1 : H]; (H, H) \vdash (\text{chk}(p_2, \mathbf{t})) (\text{req}(p_2, \mathbf{t})) x_1 := \text{bind}(app_2.comp_1, x_H) : H} \quad (\text{T-AnCr})(\text{T-AnGr}) \\
 \vdash \text{activity } app_2.comp_2(x_1) \{ (\text{chk}(p_2, \mathbf{t})) (\text{req}(p_2, \mathbf{t})) x_1 := \text{bind}(app_2.comp_1, x_H) \} : (H, H, H, \emptyset) \quad (\text{T-Comp}) \\
 \frac{\Gamma @ p_2 \vdash x_2 : L}{\Gamma @ p_2: (\tau_{in}^{caller} @ p_2, \tau_{in}^{callee} @ p_2) \vdash x := x_2 : L} \quad (\text{T-Assign}) \\
 \frac{[x_2 : L, x : L]; (L, L) \vdash (\text{chk}(p_2, \mathbf{f})) (\text{req}(p_2, \mathbf{f})) x := x_2 : L}{\vdash \text{service } app_2.comp_3(x_2) \{ \text{letvar } x := 0 \text{ in } \{ (\text{chk}(p_2, \mathbf{f})) (\text{req}(p_2, \mathbf{f})) x := x_2; \text{return } x \} \} : (L, L, L, \emptyset)} \quad (\text{T-Comp})
 \end{array}$$

FIGURE 10. Typing derivation for parameter laundering.

VI. RELATED WORKS AND DISCUSSIONS

Android apps have been plagued by security threats in a number of attacks, especially permission/induced attacks [2], [17]. For instance, privilege escalation attack authorizes unprivileged users access to privileged information [2], [11], [12], [13], [14], [17], [18], resulting in *Elevation of Privilege* [5] on Android apps. Parameter laundering attack [2], [9] and passive content leakage attack [2], [15] expose the sensitive and protected information to untrusted environments, causing *Information Disclosure* [5] on Android apps. Passive content pollution attack [2], [15] causes internal databases in vulnerable apps to be manipulated by other apps, resulting in *Tampering* [5] on Android apps.

To address these issues, a number of program analysis methods, type of which could be *dynamic* or *static*, have been proposed for Android apps in security domain [5], [5], [20], [21], [22], [23], [24], [25]. Each approach has its own strengths and weaknesses. Dynamic analysis executes the Android apps to observe its actual behaviors at runtime [5], [20], [21], [22], [23], [24], [25]. For instance, TaintDroid and TaintART monitor the Android apps at runtime and track multiple sources of sensitive data so as to detect the privacy leakage [21], [22]. In addition to taint tracking, testing is also a commonly used technique in dynamic analysis [23], [24], [25]. For example, IntentFuzzer detects capability leak vulnerabilities of Android apps by dynamically sending test intents to the components [23]. Daze fully-automated extracts components in apps and fuzzes all interfaces in apps to identify ICC vulnerabilities [25]. IntentDroid tests 8 kinds of ICC integrity vulnerabilities caused by unsafe handling of incoming ICC messages [24]. What these dynamic approaches have in common is that they all require a set of input data to execute Android apps. Since the test cases provided are often unlikely to be complete, they may not cover certain parts of the app's code and its behavior. This may lead to missing vulnerabilities or malicious behaviors, namely false negatives in security analysis [5].

$$\begin{array}{c}
\frac{\text{CT}(\text{app}_1, \text{comp}_4) = (L, L, L, L) \quad \Gamma \oplus \text{P}_3 \vdash x_H : H \neq L \quad (L \cup L) \leq (\tau_{\text{gr}}^{\text{collier}} \oplus \text{P}_3 = H)}{\text{(T-icc}_v\text{)}} \\
\frac{\Gamma \oplus \text{P}_3; (L, \tau_{\text{gr}}^{\text{collier}} \oplus \text{P}_3) \vdash \text{update}(\text{app}_1, \text{comp}_4, x_H) : H}{\text{(T-Angr)}} \\
\frac{[x_1 : H]; (L, H) \vdash (\text{req}(\text{p}_3, \mathbf{t})) \quad \text{update}(\text{app}_1, \text{comp}_4, x_H) : H}{\vdash \text{activity app}_1, \text{comp}_4(x_1) \{ (\text{req}(\text{p}_3, \mathbf{t})) \quad \text{update}(\text{app}_1, \text{comp}_4, x_H) \} : (H, L, H, ())} \text{(T-Comp)} \\
\\
\frac{\Gamma \oplus \text{P}_3 \vdash x_2 : L \quad \Gamma \oplus \text{P}_3 \vdash 1 : L}{\Gamma \oplus \text{P}_3 \vdash (x_2=1) : L} \text{(T-Arith)} \quad \frac{\Gamma \oplus \text{P}_3 \vdash \llbracket \text{DB} \rrbracket : L}{\Gamma \oplus \text{P}_3 \vdash x : \text{get}(\text{DB}); : L} \text{(T-Get)} \quad \frac{\Gamma \oplus \text{P}_3 \vdash x_2 : \Gamma \llbracket \text{DB} \rrbracket = L}{\Gamma \oplus \text{P}_3 \vdash \text{put}(x_2, \text{DB}) : L} \text{(T-Put)} \\
\frac{\Gamma \oplus \text{P}_3; (\tau_{\text{gr}}^{\text{collier}} \oplus \text{P}_3, \tau_{\text{gr}}^{\text{collier}} \oplus \text{P}_3) \vdash \text{if}(x_2=1) \quad x : \text{get}(\text{DB}); \quad \text{else put}(x_2, \text{DB}); : L}{\text{(T-If)}} \\
\frac{[x_2 : L, x : L]; (L, L) \vdash (\text{chk}(\text{p}_3, \mathbf{f})) \quad (\text{req}(\text{p}_3, \mathbf{f})) \quad \text{if}(x_2=1) \quad x : \text{get}(\text{DB}); \quad \text{else put}(x_2, \text{DB}); : L}{\text{(T-AnCr)(T-Angr)}} \\
\frac{\vdash \text{provider app}_1, \text{comp}_4(x_2) \{\text{letvar } x : \text{=} 0 \quad \text{in}(\text{chk}(\text{p}_3, \mathbf{f})) \quad (\text{req}(\text{p}_3, \mathbf{f})) \quad \text{if}(x_2=1) \quad x : \text{get}(\text{DB}); \quad \text{else put}(x_2, \text{DB}); \quad \text{return } x; \} : (L, L, L, ())}{\text{(T-Comp)}}
\end{array}$$

FIGURE 11. Typing derivation for passive content leakage.

$$\begin{array}{c}
\text{CT}(\text{app}_1, \text{comp}_4) = (L, L, L, L) \quad \Gamma \oplus \text{P}_3 \vdash 1 : L \quad (L \cup L) \leq (\tau_{\text{gr}}^{\text{collier}} \oplus \text{P}_3 = L) \quad L \leq \Gamma(x_1) \\
\frac{\Gamma \oplus \text{P}_3; (L, \tau_{\text{gr}}^{\text{collier}} \oplus \text{P}_3) \vdash x_1 : \text{query}(\text{app}_1, \text{comp}_4, 1); : L}{\text{(T-icc}_v\text{)}} \\
\frac{[x_1 : L]; (L, L) \vdash (\text{req}(\text{p}_3, \mathbf{f})) \quad x_1 : \text{query}(\text{app}_1, \text{comp}_4, 1); : L}{\text{(T-Angr)}} \\
\frac{\vdash \text{activity app}_2, \text{comp}_4(x_1) \{ (\text{req}(\text{p}_3, \mathbf{f})) \quad x_1 : \text{query}(\text{app}_1, \text{comp}_4, 1); \} : (L, L, L, ())}{\text{(T-Comp)}} \\
\\
\text{CT}(\text{app}_1, \text{comp}_6) = (T, T, T, T) \quad \Gamma \oplus \text{P}_4 \vdash 1 : T \quad (T \cup T) \leq (\tau_{\text{gr}}^{\text{collier}} \oplus \text{P}_4 = T) \quad T \leq \Gamma(x_1) = T \\
\frac{\Gamma \oplus \text{P}_4; (T, \tau_{\text{gr}}^{\text{collier}} \oplus \text{P}_4) \vdash x_1 : \text{query}(\text{app}_1, \text{comp}_6, 1); : T}{\text{(T-icc}_v\text{)}} \\
\frac{[x_1 : T]; (T, T) \vdash (\text{req}(\text{p}_4, \mathbf{f})) \quad x_1 : \text{query}(\text{app}_1, \text{comp}_6, 1); : T}{\text{(T-Angr)}} \\
\frac{\vdash \text{activity app}_1, \text{comp}_6(x_1) \{ (\text{req}(\text{p}_4, \mathbf{f})) \quad x_1 : \text{query}(\text{app}_1, \text{comp}_6, 1); \} : (T, T, T, ())}{\text{(T-Comp)}} \\
\\
\frac{\Gamma \oplus \text{P}_4 \vdash x_2 : T \quad \Gamma \oplus \text{P}_4 \vdash 1 : T}{\Gamma \oplus \text{P}_4 \vdash (x_2=1) : T} \text{(T-Arith)} \quad \frac{\Gamma \oplus \text{P}_4 \vdash \llbracket \text{DB} \rrbracket : T}{\Gamma \oplus \text{P}_4 \vdash x : \text{get}(\text{DB}); : T} \text{(T-Get)} \quad \frac{\Gamma \oplus \text{P}_4 \vdash x_2 : \Gamma \llbracket \text{DB} \rrbracket = T}{\Gamma \oplus \text{P}_4 \vdash \text{put}(x_2, \text{DB}) : T} \text{(T-Put)} \\
\frac{\Gamma \oplus \text{P}_4; (\tau_{\text{gr}}^{\text{collier}} \oplus \text{P}_4, \tau_{\text{gr}}^{\text{collier}} \oplus \text{P}_4) \vdash \text{if}(x_2=1) \quad x : \text{get}(\text{DB}); \quad \text{else put}(x_2, \text{DB}); : T}{\text{(T-If)}} \\
\frac{[x_2 : T, x : T]; (T, T) \vdash (\text{chk}(\text{p}_4, \mathbf{f})) \quad (\text{req}(\text{p}_4, \mathbf{f})) \quad \text{if}(x_2=1) \quad x : \text{get}(\text{DB}); \quad \text{else put}(x_2, \text{DB}); : T}{\text{(T-AnCr)(T-Angr)}} \\
\frac{\vdash \text{provider app}_1, \text{comp}_6(x_2) \{\text{letvar } x : \text{=} 0 \quad \text{in}(\text{chk}(\text{p}_4, \mathbf{f})) \quad (\text{req}(\text{p}_4, \mathbf{f})) \quad \text{if}(x_2=1) \quad x : \text{get}(\text{DB}); \quad \text{else put}(x_2, \text{DB}); \quad \text{return } x; \} : (T, T, T, ())}{\text{(T-Comp)}} \\
\\
\text{CT}(\text{app}_1, \text{comp}_6) = (T, T, T, T) \quad \Gamma \oplus \text{P}_4 \vdash x_U : U \neq T \quad (T \cup T) \leq (\tau_{\text{gr}}^{\text{collier}} \oplus \text{P}_4 = U) \\
\frac{\Gamma \oplus \text{P}_4; (T, \tau_{\text{gr}}^{\text{collier}} \oplus \text{P}_4) \vdash \text{update}(\text{app}_1, \text{comp}_6, x_U) : U}{\text{(T-icc}_v\text{)}} \\
\frac{[x_1 : U]; (T, U) \vdash (\text{req}(\text{p}_4, \mathbf{t})) \quad \text{update}(\text{app}_1, \text{comp}_6, x_U) : U}{\text{(T-Angr)}} \\
\frac{\vdash \text{activity app}_2, \text{comp}_6(x_1) \{ (\text{req}(\text{p}_4, \mathbf{t})) \quad \text{update}(\text{app}_1, \text{comp}_6, x_U) \} : (U, T, U, ())}{\text{(T-Comp)}}
\end{array}$$

FIGURE 12. Typing derivation for passive content pollution.

Dynamic analysis is precise yet unsound, while static analysis, examining the program structure of Android apps to reason about its potential behaviors, is considered to be conservative and sound [5], [16], [20], [26], [27], [28], [29], [30]. Some fundamental techniques are mainly used in static analysis methods for Android apps, such as taint analysis [16], [27], program slicing [28], [29], etc [30]. For instance, [27] presents FlowDroid, a novel and highly precise static taint-analysis tool for components in an Android app. [29] proposes Hopper for performing precise inter-event analysis of Android apps. However, with the increase in the number of Android apps, the costs of these program analysis methods grow exponentially, that is, these program analysis approaches may suffer from scalability issues [5], [20].

To tackle the scalability issue, formal analysis techniques are leveraged to complement program analysis [5], [30]. As a formal method to verify the type constraints of the program, type checking is commonly used in program analysis [9], [30], [31], [32], [33], [34], [35]. For instance, [31], [32] designs a formal language to describe Android apps and proposes a program analysis tool, SCandroid, to automatically type communications between apps. But they do not consider access control and implicit flow security in the type system. Reference [33] presents Cassandra, a security analysis tool to type check whether the Android apps comply with user-specified security requirements even before installing these apps. However, this method does not consider that permission checks and permission requests at runtime in ICC will affect the security level of information. Reference [34] presents a type system for Android apps, which checks the

information flow type qualifiers and guarantees that apps are free of malicious information flows. However, this method not only requires the participation of users (such as software vendors, app store auditors, etc) and other modifications to the Android permission model to adapt to its type checker, but also lacks the soundness proof. Reference [9] introduces a novel type system for enforcing secure information flow in Android apps. This literature mainly focuses on preventing the parameter laundering problem, but it neither considers the permission-based access control at runtime nor implements a usable tool. Reference [35] combines static information flow control with runtime checking to prevent the collusive information leak in Android apps. However, this approach mainly focuses on leak-freedom property in inter/process communication rather than non/interference property.

VII. CONCLUSION

In this paper, we introduce a formal calculus to reason about the operations and interactions in Android apps, especially reason about ICC and permission checks/requests at runtime. Moreover, we present a sound type system, namely TySA, for Android apps based on the proposed formal calculus to prevent permission-induced attacks. Finally, we demonstrate the effectiveness of TySA by developing a prototype tool based on the \mathbb{K} to implement our analysis.

As part of our future work, we want to consider flow-sensitive and path-sensitive dependent type system to further develop the TySA. On the practical side, we will also extend our tool to detect more types of vulnerabilities in Android apps.

APPENDIX TYPING DERIVATIONS

The typing derivations for examples in Figure 3, 4, and 5 are shown in Figure 10, 11 and 12, respectively. In these figures, the red parts indicate that the typing judgements does not follow the TySA, meaning that the sample codes are not well-typed.

REFERENCES

- [1] M. Bugliesi, S. Calzavara, and A. Spanò, "Lintent: Towards security type-checking of Android applications," in *Formal Techniques for Distributed Systems*, vol. 7892, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. P. Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, D. Beyer, and M. Boreale, Eds., Berlin, Germany: Springer, 2013, pp. 289–304.
- [2] X. Hu and Y. Zhuang, "PHRiMA: A permission-based hybrid risk management framework for Android apps," *Comput. Secur.*, vol. 94, Jul. 2020, Art. no. 101791.
- [3] A. A. G. P. S. AppBrain. *Android and Google Play Statistics, Development Resources and Intelligence*. [Online]. Available: <https://www.appbrain.com/stats>
- [4] G. P. S. Statista. *Google Play Store: Number of Apps*. [Online]. Available: <https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/>
- [5] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek, "A taxonomy and qualitative comparison of program analysis techniques for security assessment of Android software," *IEEE Trans. Softw. Eng.*, vol. 43, no. 6, pp. 492–530, Jun. 2017.
- [6] B. Rashidi, C. Fung, and E. Bertino, "Android resource usage risk assessment using hidden Markov model and online learning," *Comput. Secur.*, vol. 65, pp. 90–107, Mar. 2017.
- [7] P. H. Nguyen, S. Ali, and T. Yue, "Model-based security engineering for cyber-physical systems: A systematic mapping study," *Inf. Softw. Technol.*, vol. 83, pp. 116–135, Mar. 2017.
- [8] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang, "Exploring permission-induced risk in Android applications for malicious application detection," *IEEE Trans. Inf. Forensics Security*, vol. 9, no. 11, pp. 1869–1882, Nov. 2014.
- [9] H. Chen, A. Tiu, Z. Xu, and Y. Liu, "A permission-dependent type system for secure information flow analysis," in *Proc. IEEE 31st Comput. Secur. Found. Symp. (CSF)*, Jul. 2018, pp. 218–232.
- [10] M. A. El-Zawawy, E. Losiouk, and M. Conti, "Do not let next-intent vulnerability be your next nightmare: Type system-based approach to detect it in Android apps," *Int. J. Inf. Secur.*, vol. 20, no. 1, pp. 39–58, Feb. 2021.
- [11] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proc. 9th Int. Conf. Mobile Syst., Appl., Services*, Jun. 2011, p. 239.
- [12] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on Android," in *Information Security*, vol. 6531, M. Burmester, G. Tsudik, S. Magliveras, and I. Ilić, Eds., Berlin, Germany: Springer, 2011, pp. 346–360.
- [13] K. O. Elish, H. Cai, D. Barton, D. Yao, and B. G. Ryder, "Identifying mobile inter-app communication risks," *IEEE Trans. Mobile Comput.*, vol. 19, no. 1, pp. 90–102, Jan. 2020.
- [14] Z. Fang, W. Han, and Y. Li, "Permission based Android security: Issues and countermeasures," *Comput. Secur.*, vol. 43, pp. 205–218, Jun. 2014.
- [15] Y. Zhou and X. Jiang, "Detecting passive content leaks and pollution in Android applications," in *Proc. NDSS*, 2013.
- [16] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Oceau, and P. McDaniel, "IccTA: Detecting inter-component privacy leaks in Android apps," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 1, May 2015, pp. 280–291.
- [17] A. Sadeghi, R. Jabbarvand, N. Ghorbani, H. Bagheri, and S. Malek, "A temporal permission analysis and enforcement framework for Android," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng. (ICSE)*, May 2018, pp. 846–857.
- [18] J. Wu, T. Cui, T. Ban, S. Guo, and L. Cui, "PaddyFrog: Systematically detecting confused deputy vulnerability in Android applications: PaddyFrog: Systematically detecting confused deputy vulnerability in Android applications," *Secur. Commun. Netw.*, vol. 8, no. 13, pp. 2338–2349, Sep. 2015.
- [19] S. K. Sahay and A. Sharma, "A survey on the detection of Android malicious apps," in *Advances in Intelligent Systems and Computing*, S. K. Bhatia, S. Tiwari, K. K. Mishra, and M. C. Trivedi, Eds. Singapore: Springer, 2019, pp. 437–446.
- [20] Y.-D. Bromberg and L. Gitzinger, "DroidAutoML: A microservice architecture to automate the evaluation of Android machine learning detection systems," in *Distributed Applications and Interoperable Systems*, A. Remke and V. Schiavoni, Eds., Cham, Switzerland: Springer, 2020, pp. 148–165.
- [21] W. Enck, P. Gilbert, S. Han, V. Tendulkar, 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," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 1–29, Jun. 2014.
- [22] M. Sun, T. Wei, and J. C. S. Lui, "TaintART: A practical multi-level information-flow tracking system for Android RunTime," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Vienna, Austria, Oct. 2016, pp. 331–342.
- [23] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan, "IntentFuzzer: Detecting capability leaks of Android applications," in *Proc. 9th ACM Symp. Inf. Comput. Commun. Secur.*, vol. 18, Jun. 2014, pp. 531–536.
- [24] R. Hay, O. Tripp, and M. Pistoia, "Dynamic detection of inter-application communication vulnerabilities in Android," in *Proc. Int. Symp. Softw. Test. Anal.*, Baltimore, MD, USA, Jul. 2015, pp. 118–128.
- [25] R. Johnson, M. Elsabagh, A. Stavrou, and J. Offutt, "Dazed droids: A longitudinal study of Android inter-app vulnerabilities," in *Proc. Asia Conf. Comput. Commun. Secur.*, vol. 30, May 2018, pp. 777–791.
- [26] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *Proc. WODA Workshop Dyn. Anal.*, Portland, OR, USA, May 2003, pp. 24–27.
- [27] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Oceau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, 2014.
- [28] D. Titze and J. Schütte, "Apparecium: Revealing data flows in Android applications," in *Proc. IEEE 29th Int. Conf. Adv. Inf. Netw. Appl.*, South Korea, Mar. 2015, pp. 579–586.
- [29] S. Blackshear, B.-Y.-E. Chang, and M. Sridharan, "Selective control-flow abstraction via jumping," in *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program., Syst., Lang., Appl.*, vol. 3, Pittsburgh, PA, USA, Oct. 2015, pp. 163–182.
- [30] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Oceau, J. Klein, and Y. Le Traon, "Static analysis of Android apps: A systematic literature review," *Inf. Softw. Technol.*, vol. 88, pp. 67–95, Aug. 2017.
- [31] A. Chaudhuri, "Language-based security on Android," in *Proc. ACM SIGPLAN 4th Workshop Program. Lang. Anal. Secur.*, Jun. 2009, pp. 1–7.
- [32] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "ScanDroid: Automated security certification of Android applications," *Tech. Rep.*, 2009.
- [33] S. Lortz, H. Mantel, A. Starostin, T. Bähr, D. Schneider, and A. Weber, "Cassandra: Towards a certifying app store for Android," in *Proc. 4th ACM Workshop Secur. Privacy Smartphones Mobile Devices*, Scottsdale, AZ, USA, Nov. 2014, pp. 93–104.
- [34] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. B. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu, "Collaborative verification of information flow for a high-assurance app store," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Scottsdale, AZ, USA, Nov. 2014, pp. 1092–1104.
- [35] Z.-P. Zhang, M. Fu, and X.-Y. Feng, "A lightweight dynamic enforcement of privacy protection for Android," *J. Comput. Sci. Technol.*, vol. 34, no. 4, pp. 901–923, Jul. 2019.
- [36] G. Rosu and T. F. Serbănuță, "An overview of the k semantic framework," *J. Log. Algebr. Program.*, vol. 79, no. 6, pp. 397–434, Aug. 2010.
- [37] K. F. K Team. *K Framework*. [Online]. Available: <http://www.kframework.org/index.php>
- [38] X. Hu, Y. Zhuang, and F. Zhang, "A security modeling and verification method of embedded software based on Z and MARTE," *Comput. Secur.*, vol. 88, Jan. 2020, Art. no. 101615.
- [39] P. O. Android Developers. *Permissions Overview*. [Online]. Available: <https://developer.android.com/guide/topics/permissions>

- [40] R. A. P. Android Developers. *Request App Permissions*. [Online]. Available: <https://developer.android.com/training/permissions>
- [41] H. Mantel, D. Sands, and H. Sudbrock, "Assumptions and guarantees for compositional noninterference," in *Proc. IEEE 24th Comput. Secur. Found. Symp.*, Jun. 2011, pp. 218–232.
- [42] C. P. Android Developers. *Content Providers*. [Online]. Available: <https://developer.android.com/guide/topics/providers>
- [43] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *J. Comput. Secur.*, vol. 4, nos. 2–3, pp. 167–187, Apr. 1996.
- [44] A. F. Android Developers. *Application Fundamentals*. [Online]. Available: <https://developer.android.com/guide/components>
- [45] B. Android Developers. *Binder*. [Online]. Available: <https://developer.android.com/reference>
- [46] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Commun. ACM*, vol. 20, no. 7, pp. 504–513, Jul. 1977.
- [47] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 1, pp. 5–19, Jan. 2003.
- [48] A. Banerjee and D. A. Naumann, "Stack-based access control and secure information flow," *J. Funct. Program.*, vol. 15, no. 2, pp. 131–177, Mar. 2005.
- [49] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, May 1976.
- [50] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proc. IEEE Symp. Secur. Privacy*, Oakland, CA, USA, Apr. 1982, p. 11.
- [51] S. Kan, Z. Chen, D. Sanan, S.-W. Lin, and Y. Liu, "An executable operational semantics for rust with the formalization of ownership and borrowing," 2018, *arXiv:1804.07608*.
- [52] J. Jiao, S. Kan, S.-W. Lin, D. Sanan, Y. Liu, and J. Sun, "Semantic understanding of smart contracts: Executable operational semantics of solidity," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 1695–1712.



XINWEN HU received the B.E. and Ph.D. degrees in computer science and technology from Nanjing University of Aeronautics and Astronautics, China, in 2016 and 2021, respectively. Since 2021, she has been a Lecturer with the School of Journalism and Communication, Hunan Normal University, China. Her research interests include developing effective methodologies, techniques, and tools to improve software security.

• • •