A Permission-Dependent Type System for Secure Information Flow Analysis

Hongxu Chen
Nanyang Technological University, Singapore
hchen017@e.ntu.edu.sg

Alwen Tiu
Australian National University, Australia
alwen.tiu@anu.edu.au

Zhiwu Xu†
CSSE, Shenzhen University, China
xuzhiwu@szu.edu.cn

Yang Liu
Nanyang Technological University, Singapore
yangliu@ntu.edu.sg

***Abstract*—We introduce a novel type system for enforcing secure information flow in an imperative language. Our work is motivated by the problem of statically checking potential information leakage in Android applications. To this end, we design a lightweight type system featuring Android permission model, where the permissions are statically assigned to applications and are used to enforce access control in the applications. We take inspiration from a type system by Banerjee and Naumann to allow security types to be dependent on the permissions of the applications. A novel feature of our type system is a typing rule for conditional branching induced by permission testing, which introduces a merging operator on security types, allowing more precise security policies to be enforced. The soundness of our type system is proved with respect to non-interference. In addition, a type inference algorithm is presented for the underlying security type system, by reducing the inference problem to a constraint solving problem in the lattice of security types.**

I. Background and Introduction

Mobile security has become increasingly important for our daily life due to the pervasive use of mobile applications. Among the mobile devices that are currently in the market, Android devices account for the majority of them so analysis of their security has been of significant interests. There has been a large number of analyses on Android security ([1]–[5]) focusing on detecting potential security violations. Here we are interested instead in the problem of constructing secure applications, in particular, in providing guarantee of information flow security in the constructed applications.

We follow the language-based security approach whereby information flow is enforced through type systems [6]–[9]. In particular, we propose a design of a type system that guarantees non-interference property [8], i.e., typable programs are non-interferent. As shown in [10], non-interference provides a general and natural way to model information flow security. The type-based approach to non-interference requires assigning security labels to program variables and security policies to functions or procedures. Such policies are typically encoded as types, and typeability of the program implies that the runtime behavior of the program complies with the stated policies. Security labels form a lattice structure with an underlying partial order $\leq$, e.g., a lattice with two elements "high" (H) and "low" (L) where $L \leq H$. Typing rules can then be designed to prevent both explicit and implicit flow (through conditionals, e.g., if-then-else statements) from H to L. To prevent an *explicit* flow, the typing rule for an assignment statement such as *x := e* would require that $l(e) \leq l(x)$ where $l(.)$ denotes the security level of an expression. To prevent an *implicit* flow, e.g., *if (y = 0) then x:= 0 else x := 1*, most type systems for non-interference require that the assignments in *both* branches are given the same security level that is higher or at least equal to the security level of the condition (*y=0*). For example, if y is of type H and x is of type L, the statement would not be typable.

†Corresponding author.

```
String getContactNo(String name) {
    String number;
    if(checkPermission(READ_CONTACT))
       number = ... ; // query the phone number
    else number = "";
    return number;
}
```

Listing 1. Sample code for getting contact info with a permission check.

A. Motivating Examples

In designing an information flow type system for Android, we encounter a common pattern of conditionals that would not be typable using conventional type systems. Consider the pseudo-code in Listing 1. Such a code fragment could be part of a phone dialer or a social network service app such as Facebook, WhatsApp, where *getContactNo* provides a public interface to query the phone number associated with a name. The (implicit) security policy in this context is that contact information (the phone number) can only be released if the calling app has READ_CONTACT permission. The latter is enforced using the *checkPermission* API in Android. Suppose phone numbers are labelled with H, and the empty string is labelled with L. If the interface is invoked by an app that has the required permission, the phone number (H) is returned; otherwise an empty string (L) is returned. In both cases, no

data leakage happens: in the former case, the calling app is authorized; and in the latter case, no sensitive data is ever returned. By this informal reasoning, the function complies with the implicit security policy and it should be safe to be called in *any* context, regardless of the permissions the calling app has. However, in the traditional (non-value dependent) typing rule for the if-then-else construct, one would assign *the same* security level to both branches, and the return value of the function would be assigned level H. As a result, if this function is called from an app with *no* permission, assigning the return value to a variable with security level L has to be rejected by the type system even though no sensitive information is leaked. To cater for such a scenario, we need to make the security type of *getContactNo depend on* the permissions possessed by the caller.

Banerjee and Naumann [11] proposed a type system (which we shall refer to as BN system) that incorporates permissions into function types. Their type system was designed for an access control mechanism different from ours, but the basic principles are still applicable. In BN system, a Java class may be assigned a set of permissions which need to be *explicitly enabled* via an **enable** command for them to have any effect. We say a permission is *disabled* for an class if it is *not assigned* to the class, or it is *assigned* to the class but is *not explicitly enabled*. Depending on the permissions of the calling class (corresponding to an *app* in the above example), a function such as *getContactNo* can have a collection of types. In BN type system, the types of a function take the form $(l_1, \ldots, l_n) \xrightarrow{P} l$ where $l_1, \ldots, l_n$ denote security levels of the input, l denotes the security level of the output and P denotes a set of permissions that are disabled by the caller. The idea is that permissions are guards to sensitive values. Thus conservatively, one would type the return value of *getContactNo* as L only if one knows that the permission READ_CONTACT is disabled. In BN system, *getContactNo* admits the following types:

$$\textit{getContactNo} : L \xrightarrow{P} L \qquad \textit{getContactNo} : L \xrightarrow{\emptyset} H$$

where $P = \{\text{READ\_CONTACT}\}$. When typing a call to *getContactNo* by an app without permissions, the first type of *getContactNo* is used; otherwise the second type is used.

In BN system, the typing judgment is parameterized by a permission set Q containing the permissions that are currently known to be disabled. The set Q may or may not contain all disabled permissions. Their language features a command "**test**(P) c_1 **else** c_2", which means that if the permissions in the set P are *all enabled*, then the command behaves like c_1; otherwise it behaves like c_2. The typing rules for the *test* command (in a much simplified form) are:

$$\text{(R1)}\ \frac{Q \cap P = \emptyset \quad Q \vdash c_1 : \tau \quad Q \vdash c_2 : \tau}{Q \vdash \textbf{test}(P)\ c_1\ \textbf{else}\ c_2 : \tau}$$

$$\text{(R2)}\ \frac{Q \cap P \neq \emptyset \quad Q \vdash c_2 : \tau}{Q \vdash \textbf{test}(P)\ c_1\ \textbf{else}\ c_2 : \tau}$$

where Q is a set of permissions that are disabled. When $Q \cap P \neq \emptyset$, then at least one of the permissions in P is disabled, thus one can determine statically that "**test**(P)" would fail and only the *else* branch would be executed at runtime. This case is reflected in the typing rule R2. When $Q \cap P = \emptyset$, there can be two possible runtime scenarios. One scenario is that all permissions in P are enabled, so "**test**(P)" succeeds and c_1 is executed. The other is that some permissions in P are disabled, but are not accounted for in Q. So in this case, one cannot determine statically which branch of **test** will be taken at runtime. The typing rule R1 therefore conservatively considers typing both branches.

When adapting BN system to Android, R1 is still too strong in some scenarios, especially when it is desired that the *absence* of some permissions leads to the release of sensitive values. Consider for example an application that provides location tracking information related to a certain *advertising ID* (Listing 2), where the latter provides a unique ID for the purpose of anonymizing mobile users to be used for advertising (instead of relying on hardware device IDs such as IMEI numbers). If one can correlate an advertising ID with a unique hardware ID, it will defeat the purpose of the anonymizing service provided by the advertising ID. To prevent that, *getInfo* returns the location information for an advertising ID only if the caller *does not* have access to device ID. To simplify discussion, let us assume that the permissions to access IMEI and location information are denoted by p and q, respectively.

```
String getInfo() {
   String r = "";
   test(p) {
     test(q) r = loc; else r = "";
   } else {
     test(q) r = id++loc; else r = "";
   }
   return r;
}
```

Listing 2. An example about non-monotonic policy.

Here *id* denotes a unique advertising ID generated and stored by the app for the purpose of anonymizing user tracking and *loc* denotes location information. The function first tests whether the caller has access to IMEI number. If it does, and if it has access to location, then only the location information is returned. If the caller has no access to IMEI number, but can access location information, then the combination of advertising id and location *id++loc* is returned. In all the other cases, an empty string is returned. Let us consider a lattice with four elements ordered as: $L \leq l_1, l_2 \leq H$, where l_1 and l_2 are incomparable. We specify that empty string is of type L, *loc* is of type l_1, *id* is of type l_2, and the aggregate *id++loc* is of type H. Consider the case where the caller has permissions p and q and both are (explicitly) *enabled*. When applying BN system, the desired type of *getInfo* in this case is $() \xrightarrow{\emptyset} l_1$. This means that the type of *r* has to be at most l_1. Since no permissions are disabled, only R1 is applicable to type this program. This, however, will force both branches of **test**(p) to have the same type. As a result, *r* has to be typed as H so

that all four assignments in the program can be typed.

The issue with the example in Listing 2 is that the stated security policy is *non-monotonic* in the sense that an app with more permissions does not necessarily have access to information with higher level of security. The fact that BN system cannot precisely capture non-monotonic policies appears to be a design decision: they cited in [11] the lack of motivating examples for non-monotonic policies, and suggested that to accommodate such policies one might need to consider a notion of declassification. As we have seen, however, non-monotonic policies can arise naturally in mobile applications. In a study on Android malwares [1], Enck et. al. identify several combinations of permissions that are potentially 'dangerous', in the sense that they allow potentially unsafe information flow. An information flow policy that requires the *absence* of such combinations of permissions in information release would obviously be non-monotonic. In general, non-monotonic policies can be required to solve the *aggregation problem* studied in the information flow theory [12], where several pieces of low security level information may be pooled together to learn information at a higher security level.

We therefore designed a more precise type system for information flow under an access control model inspired by Android framework. Our type system solves the problem of typing non-monotonic policies without resorting to downgrading or declassifying information. It is done technically via a *merging* operator on security types, to keep information related to both branches of **test**. Additionally, there is a significant difference in the permission model used in traditional type systems such as BN system, where permissions are propagated across method invocations among apps. This is due to the fact that permissions in Android are relevant only during inter-process calls, while permissions are not inherited along the call chains across apps. As we shall see in Section II-E, this may give rise to a type of attack which we call "parameter laundering" attack if one adopts a naive typing rule for function calls. The soundness proof for our type system is significantly different from that for BN type system due to the difference in permission model and the new merging operator on types in our type system.

Due to space constraints, most proofs are omitted but they can be found in a technical report [13].

B. Contributions

The contributions of our work are three-fold.

1) We develop a lightweight type system in which security types are dependent on a permission-based access control mechanism, and prove its soundness with respect to non-interference (Section II). A novel feature of the type system is the type merging constructor, used for typing the conditional branch in permission checking, which allows us to model non-monotonic information flow policies.
2) We identify a problem of explicit flow through function calls in the setting where permissions are not propagated during function calls. This problem arises as a byproduct of Android's permission model, which is significantly different from that in JVM, and adopting a standard typing rule for function calls such as the one proposed for Java in [11] would lead to unsoundness. We call this problem the parameter laundering problem and we propose a typing rule for function calls that prevents it.
3) We show that the type inference is decidable for our type system, by reducing it to a constraint solving problem (Section III).

II. A Secure Information Flow Type System

In this section, we present the proposed information flow type system. Section II-A discusses informally a permission-based access control model, which is an abstraction of the permission mechanism used in Android. Section II-B and Section II-C give the operational semantics of a simple imperative language that includes permission checking constructs based on the abstract permission model. Section II-D and Section II-E describe the type system for our language and prove its soundness with respect to a notion of non-interference.

A. A model of permission-based access control

Instead of taking all the language features and the library dependencies of Android apps into account, we focus on the permission model used in inter-component communications within and across apps. Such permissions are used to regulate access to protected resources, such as device id, location information, contact information, etc.

In Android, an app specifies the permissions it needs at installation time via a manifest file. In recent versions of Android (since Android 6.0, API level 23), some of these permissions need to be granted by users at runtime. But at no point a permission request is allowed if it is not already specified in the manifest. For now, we assume a permission enforcement mechanism that applies to Android versions prior to version 6.0, so it does not account for permission granting at runtime [1]. Runtime permission granting [14] poses some problems in typing non-monotonic policies; we shall come back to this point later in Section V.

An Android app may provide services to other apps, or other components within the app itself. Such a service provider may impose permissions on other apps who want to access its services. Communication between apps is implemented through Binder IPC (inter-process communications) [15].

In our model, a program can be seen as a highly abstracted version of an app, and the intention is to show how one can reason about information flow in such a service provider when access control is imposed on the calling app. In the following we shall not model explicitly the IPC mechanism of Android, but will instead model it as a function call. Note that this abstraction is practical since it can be achieved by conventional data and control flow analyses, together with the modeling of Android IPC specific APIs. The feasibility

[1]To be specific, runtime permission request requires the compatible version specified in the manifest file to be greater than or equal to API level 23, and running OS should be at least Android 6.0.

has been demonstrated by frameworks like FlowDroid [3], Amandroid [4], IccTA[5], etc. [2]

One significant issue that has to be taken into account is that Android framework does not track IPC call chains between apps and permissions of an app are not propagated to the callee. That is, an app A calling another app B does not grant B the permissions assigned to A. This is different from the traditional type systems such as BN where permissions can potentially propagate along the call stacks. Note however that B can potentially have more permissions than A, leading to a potential privilege escalation, a known weakness in Android permission system [16]. Another consequence of lacking transitivity is that in designing the type system, one must be careful to avoid what we call a "parameter laundering" attack (see Section II-D).

B. A Language with Permission Checks

As mentioned earlier, we do not model directly all the language features of an Android app, but use a much simplified language to focus on the permission mechanism part. The language is a variant of the language considered in [8], extended with functions and an operator for permission checks.

We model an *app* as a collection of *functions* (*services*), together with a statically assigned permission set. A *system*, denoted by $\mathcal{S}$, consists of a set of apps. We use capital letters $A, B, \ldots$ to denote apps. A function f defined in an app A is denoted by $A.f$, and may be called independently of other functions in the same app. The intention is that a function models an application component (i.e., *Activity*, *Service*, *BroadCastReceiver*, and *ContentProvider*) in Android, which may be called from within the same app or other apps.

We assume that only one function is executed at a time, so we do not model concurrent executions of apps. We think that in the Android setting, considering sequential behavior only is not overly restrictive. This is because the communication between apps are (mostly) done via IPC. Shared states between apps, which is what contributes to the difficulty in concurrency handling, is mostly absent, apart from the very limited sharing of preferences. In such a setting, each invocation of a service can be treated independently as there is usually no synchronization needed between different invocations. Additionally, we assume functions in a system are not (mutually) recursive, so there is a finite chain of function calls from any given function. The absence of recursion is not a restriction, since our functions are supposed to model communications in Android, which are rarely recursive. We denote with $\mathbf{P}$ the finite set containing all permissions in the system. Each app is assigned a static set of permissions drawn from this set. The powerset of $\mathbf{P}$ is written as $\mathscr{P}$.

For simplicity, we consider only programs manipulating *integers*, so the expressions in our language all have the integer type. Boolean values are encoded as 0 (false) and any non-zero values (true). The grammar for expressions is given below:

$$e ::= n \mid x \mid e \ \mathbf{op}\ e$$

where n denotes an integer literal, x denotes a variable, and **op** denotes a binary operation. The commands of the language are given in the following grammar:

$$\begin{aligned} c \ ::= \ & x := e \mid \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c \mid \mathbf{while}\ e\ \mathbf{do}\ c \mid c; c \\ & \mid \mathbf{letvar}\ x = e\ \mathbf{in}\ c \mid x := \mathbf{call}\ A.f(\overline{e}) \mid \mathbf{test}(p)\ c\ \mathbf{else}\ c \end{aligned}$$

The first four constructs are respectively assignment, conditional, while-loop and sequential composition. The statement "**letvar** $x = e$ **in** c" is a local variable declaration statement. Here x is declared and initialized to e, and its scope is the command c. We require that x does not occur in e. The statement "$x :=$ **call** $A.f(\overline{e})$" denotes an assignment whose right hand side is a function call to $A.f$. The statement "**test**(p) c_1 **else** c_2" checks whether the calling app has permission p: if it does then c_1 is executed, otherwise c_2 is executed. This is similar to the **test** construct in BN system, except that we allow testing only one permission at a time. This is a not real restriction since both versions of the **test** can simulate one another.

A function declaration has the following syntax:

$$F ::= A.f(\overline{x})\big\{\mathbf{init}\ r = 0\ \mathbf{in}\ \{c; \mathbf{return}\ r\}\big\}$$

where $A.f$ is the name of the function, $\bar{x}$ are function parameters, c is a command and r is a local variable that holds the return value of the function. The variables $\overline{x}$ and r are bound variables with the command "c; **return** r" in their scopes. We consider only *closed functions*, i.e., the variables occurring in c are either introduced by **letvar** or from the set $\{\overline{x}, r\}$.

C. Operational Semantics

We assume that function definitions are stored in a table FD indexed by function names, and the permission sets assigned to apps are given by a table Θ indexed by app names.

An *evaluation environment* is a finite mapping from variables to values (i.e., integers). We denote with $EEnv$ the set of evaluation environments. Elements of $EEnv$ are ranged over by η. We use the notation $[x_1 \mapsto v_1, \cdots, x_n \mapsto v_n]$ to denote an evaluation environment mapping variable x_i to value v_i; this will sometimes be abbreviated as $[\overline{x} \mapsto \overline{v}]$. The domain of $\eta = [x_1 \mapsto v_1, \cdots, x_n \mapsto v_n]$ (i.e., $\{x_1, \ldots, x_n\}$) is denoted by $dom(\eta)$. Given two environments η_1 and η_2, we define $\eta_1\eta_2$ as an environment η such that $\eta(x) = \eta_2(x)$ if $x \in dom(\eta_2)$, otherwise $\eta(x) = \eta_1(x)$. For example, $\eta[x \mapsto v]$ maps x to v, and y to $\eta(y)$ for any $y \in dom(\eta)$ such that $y \neq x$. Given a mapping η and a variable x, we write $\eta - x$ to denote the mapping resulting from removing x from $dom(\eta)$.

The operational semantics for expressions and commands is given in Fig. 1. The evaluation judgment for expressions has the form $\eta \vdash e \rightsquigarrow v$, which states that expression e evaluates to value v when variables in e are interpreted in the evaluation environment η. We write $\eta \vdash \overline{e} \rightsquigarrow \overline{v}$, where $\overline{e} = e_1, \ldots, e_n$ and

[2] We have also been implementing a permission-dependent information flow analysis tool on top of Amandroid. The basic idea is similar to the one mentioned in this paper, however the focus is improving the precision of information leakage detection rather than non-interference certification.

$$\text{E-VAL}\frac{}{\eta \vdash v \rightsquigarrow v} \qquad \text{E-VAR}\frac{}{\eta \vdash x \rightsquigarrow \eta(x)}$$

$$\text{E-OP}\frac{\eta \vdash e_1 \rightsquigarrow v_1 \quad \eta \vdash e_2 \rightsquigarrow v_2}{\eta \vdash e_1 \ \textbf{op}\ e_2 \rightsquigarrow v_1 \ op\ v_2} \qquad \text{E-LETVAR}\frac{\eta \vdash e \rightsquigarrow v \quad \eta[x \mapsto v]; A; P \vdash c \rightsquigarrow \eta'}{\eta; A; P \vdash \textbf{letvar}\ x = e\ \textbf{in}\ c \rightsquigarrow \eta' - x}$$

$$\text{E-SEQ}\frac{\eta; A; P \vdash c_1 \rightsquigarrow \eta' \quad \eta'; A; P \vdash c_2 \rightsquigarrow \eta''}{\eta; A; P \vdash c_1; c_2 \rightsquigarrow \eta''} \qquad \text{E-ASS}\frac{\eta \vdash e \rightsquigarrow v}{\eta; A; P \vdash x := e \rightsquigarrow \eta[x \mapsto v]}$$

$$\text{E-IF-T}\frac{\eta \vdash e \rightsquigarrow v \quad v \neq 0 \quad \eta; A; P \vdash c_1 \rightsquigarrow \eta'}{\eta; A; P \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \rightsquigarrow \eta'} \qquad \text{E-IF-F}\frac{\eta \vdash e \rightsquigarrow v \quad v = 0 \quad \eta; A; P \vdash c_2 \rightsquigarrow \eta'}{\eta; A; P \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \rightsquigarrow \eta'}$$

$$\text{E-WHILE-T}\frac{\eta \vdash e \rightsquigarrow v \quad v \neq 0 \quad \eta; A; P \vdash c \rightsquigarrow \eta' \quad \eta'; A; P \vdash \textbf{while}\ e\ \textbf{do}\ c \rightsquigarrow \eta''}{\eta; A; P \vdash \textbf{while}\ e\ \textbf{do}\ c \rightsquigarrow \eta''} \qquad \text{E-WHILE-F}\frac{\eta \vdash e \rightsquigarrow v \qquad v = 0}{\eta; A; P \vdash \textbf{while}\ e\ \textbf{do}\ c \rightsquigarrow \eta}$$

$$\text{E-CP-T}\frac{p \in P \quad \eta; A; P \vdash c_1 \rightsquigarrow \eta'}{\eta; A; P \vdash \textbf{test}(p)\ c_1\ \textbf{else}\ c_2 \rightsquigarrow \eta'} \qquad \text{E-CP-F}\frac{p \notin P \quad \eta; A; P \vdash c_2 \rightsquigarrow \eta'}{\eta; A; P \vdash \textbf{test}(p)\ c_1\ \textbf{else}\ c_2 \rightsquigarrow \eta'}$$

$$\text{E-CALL}\frac{FD(B.f) = B.f(\overline{y})\{\textbf{init}\ r = 0\ \textbf{in}\ \{c; \textbf{return}\ r\}\} \qquad \eta \vdash \overline{e} \rightsquigarrow \overline{v} \quad [\overline{y} \mapsto \overline{v}, r \mapsto 0]; B; \Theta(A) \vdash c \rightsquigarrow \eta'}{\eta; A; P \vdash x := \textbf{call}\ B.f(\overline{e}) \rightsquigarrow \eta[x \mapsto \eta'(r)]}$$

Fig. 1. Evaluation rules for expressions and commands, given a function definition table FD and a permission assignment Θ.

$\overline{v} = v_1, \ldots, v_n$ for some n, to denote a sequence of judgments $\eta \vdash e_1 \rightsquigarrow v_1, \ldots, \eta \vdash e_n \rightsquigarrow v_n$.

The evaluation judgment for commands takes the form $\eta; A; P \vdash c \rightsquigarrow \eta'$ where η is an evaluation environment before the execution of the command c, and η' is the evaluation environment after the execution of c. Here A refers to the app to which the command c belongs. The permission set P denotes the *permission context*, i.e., it is the set of permissions of the app which invokes the function of A in which the command c resides. The caller app may be A itself (in which case the permission context will be the same as the permission set of A) but more often it is another app in the system.

The operational semantics of most commands are straightforward. We explain the semantics of the *test* primitive and the function call. Rules (E-CP-T) and (E-CP-F) capture the semantics of the **test** primitive. These are where the permission context P in the evaluation judgement is used. The semantics of function calls is given by (E-CALL). Notice that c inside the body of *callee* is executed under the permission context $\Theta(A)$, which is the permission set of A. The permission context P in the conclusion of that rule, which denotes the permission of the app that calls A, is not used in the premise. That is, the permission context of A is not inherited by the callee function $B.f$. This reflects the way permission contexts in Android are passed on during IPCs [15], [17], and is also a major difference between our permission model and that in BN type system, where permission contexts are inherited by successive function calls.

D. Security Types

In information flow type systems such as [8], it is common to adopt a lattice structure to encode security levels. Security types in this setting are just security levels. In our case, we generalize the security types to account for the dependency of security levels on permissions. So we shall distinguish security levels, given by a lattice structure which encodes sensitivity levels of information, and security types, which are mappings from permissions to security levels. We assume the security levels are given by a lattice $\mathscr{L}$, with a partial order $\leq_{\mathscr{L}}$. Security types are defined in the following.

Definition II.1 *A* base security type *(or* base type*)* t *is a mapping from* $\mathscr{P}$ *to* $\mathscr{L}$*. We denote with* $\mathcal{T}$ *the set of base types. Given two base types* s *and* t*, we say* $s = t$ *iff* $s(P) = t(P)$ *for all* $P \in \mathscr{P}$*. We define an ordering* $\leq_{\mathcal{T}}$ *on base types as follows:* $s \leq_{\mathcal{T}} t$ *iff* $\forall\ P \in \mathscr{P}$*,* $s(P) \leq_{\mathscr{L}} t(P)$*.*

As we shall see, if a variable is typed by a base type, the sensitivity of its content may depend on the permissions of the app which writes to the variable. In contrast, in traditional information flow type systems, a variable annotated with a security level has a fixed sensitivity level regardless of the permissions of the app that writes to the variable.

The set of base types with the order $\leq_{\mathcal{T}}$ forms a lattice. The join and meet of the lattice are defined as follows:

Definition II.2 *For* $s, t \in \mathcal{T}$*,* $s \sqcup t$ *and* $s \sqcap t$ *are defined as*

$$(s \sqcup t)(P) = s(P) \sqcup t(P), \forall P \in \mathscr{P}$$
$$(s \sqcap t)(P) = s(P) \sqcap t(P), \forall P \in \mathscr{P}$$

From now on, we shall drop the subscripts in $\leq_{\mathscr{L}}$ and $\leq_{\mathcal{T}}$ when no ambiguity arises.

Definition II.3 *Given a security level* l*, we define* $\hat{l}$ *as follows: for all* $P \in \mathscr{P}$*, we have* $\hat{l}(P) = l$*.*

Accordingly, a security level l can be lifted to the base type $\hat{l}$ that maps all permission sets to level l itself.

Definition II.4 *A* function type *has the form* $\overline{t} \to t$*, where* $\overline{t} = (t_1, \ldots, t_m)$*,* $m \geq 0$ *and* t, t_i *are base types. The types* $\overline{t}$ *are the types for the arguments of the function and* t *is the return type of the function.*

In our type system, security types of expressions (commands, functions, resp.) may be altered depending on the execution context. That is, when an expression is used in a context where a permission check has been performed (either successfully or unsuccessfully), its type may be adjusted to take into account the *presence* or *absence* of the checked permission. Such an adjustment is called a *promotion* or a *demotion*.

Definition II.5 *Given a permission p, the* promotion *and* demotion *of a base type t with respect to p are:*

$$(t\uparrow_p)(P) = t(P \cup \{p\}), \forall P \in \mathscr{P} \quad \text{(promotion)}$$
$$(t\downarrow_p)(P) = t(P \setminus \{p\}), \forall P \in \mathscr{P} \quad \text{(demotion)}$$

The promotion *and* demotion *of a function type $\bar{t} \to t$, where $\bar{t} = (t_1, \ldots, t_m)$, are respectively:*

$$(\bar{t} \to t)\uparrow_p = \bar{t}\uparrow_p \to t\uparrow_p, \text{ where } \bar{t}\uparrow_p = (t_1\uparrow_p, \ldots, t_m\uparrow_p),$$
$$(\bar{t} \to t)\downarrow_p = \bar{t}\downarrow_p \to t\downarrow_p, \text{ where } \bar{t}\downarrow_p = (t_1\downarrow_p, \ldots, t_m\downarrow_p).$$

E. Security Type System

We first define a couple of operations on security types and permissions that will be used later.

Definition II.6 *Given $t \in \mathcal{T}$ and $P \in \mathscr{P}$, the* projection *of t on a permission set P is a security type $\pi_P(t)$ defined as:*

$$\pi_P(t)(Q) = t(P), \ \forall Q \in \mathscr{P}.$$

Type projection of a list of types on P is then written as

$$\pi_P((t_1, \ldots, t_n)) = (\pi_P(t_1), \ldots, \pi_P(t_n)).$$

Definition II.7 *Given a permission p and two types t_1 and t_2, the* merging *of t_1 and t_2 along p, denoted as $t_1 \triangleright_p t_2$, is:*

$$(t_1 \triangleright_p t_2)(P) = \begin{cases} t_1(P) & p \in P \\ t_2(P) & p \notin P \end{cases} \quad \forall P \in \mathscr{P}$$

A *typing environment* is a finite mapping from variables to base types. We use the notation $[x_1 : t_1, \ldots, x_n : t_n]$ to enumerate a typing environment with domain $\{x_1, \ldots, x_n\}$. Typing environments are ranged over by Γ. Given Γ_1 and Γ_2 such that $dom(\Gamma_1) \cap dom(\Gamma_2) = \emptyset$, we write $\Gamma_1\Gamma_2$ to denote a typing environment that is the (disjoint) union of the mappings in Γ_1 and Γ_2.

Definition II.8 *Given a typing environment Γ, its* promotion *and* demotion *along p are typing environments $\Gamma\uparrow_p$ and $\Gamma\downarrow_p$, such that $(\Gamma\uparrow_p)(x) = \Gamma(x)\uparrow_p$ and $(\Gamma\downarrow_p)(x) = \Gamma(x)\downarrow_p$ for every $x \in dom(\Gamma)$. The projection of Γ on $P \in \mathscr{P}$ is a typing environment $\pi_P(\Gamma)$ such that $(\pi_P(\Gamma))(x) = \pi_P(\Gamma(x))$ for each $x \in dom(\Gamma)$.*

There are three typing judgments in our type system as explained below. All these judgments are implicitly parameterized by a function type table, FT, which maps all function names to function types, and a mapping Θ assigning permission sets to apps.

- Expression typing: $\Gamma \vdash e : t$. This says that under Γ, the expression e has a base type at most t.
- Command typing: $\Gamma; A \vdash c : t$. This means that the command c writes to variables with type at least t, when executed by app A, under the typing environment Γ.
- Function typing: The typing judgment takes the form:

$$\vdash B.f(\overline{x})\{\textbf{init } r = 0 \textbf{ in } \{c; \textbf{return } r\}\} : \bar{t} \to t'$$

  where $\overline{x} = (x_1, \ldots, x_n)$ and $\bar{t} = (t_1, \ldots, t_n)$ for some $n \geq 0$. Functions are polymorphic in the permissions of the caller. Intuitively, this means that each caller of the function above with permission set P "sees" the function as having type $\pi_P(\bar{t}) \to \pi_P(t')$. That is, if the function is called from another app with permission P, then it expects input of type up to $\pi_P(\bar{t})$ and a return value of type at most $\pi_P(t')$.

The typing rules are given in Fig. 2. Most of them are common to information flow type systems [8], [9], [11] except for T-CP and T-CALL. Note that in the subtyping rule for commands (T-SUB$_c$), the security type of the effect of the command can be safely downgraded, since typing for commands keeps track of a lower bound of the write effects of the command. This typing rule for command is standard, see, e.g., [8] for a more detailed discussion.

In T-CP, to type statement **test**(p) c_1 **else** c_2, we type c_1 in a promoted typing environment for a successful permission check on p, and c_2 in a demoted typing environment for a failed permission check on p. The challenge is how to combine the types of the two premises to obtain the type for the conclusion. One possibility is to force the type of the two premises and the conclusion to be identical (i.e., treat permission check the same as other if-then-else statements and apply T-IF). This, as we have seen in Section I, leads to a loss in precision of the type for **test** construct. Instead, we consider a more refined *merged* type $t_1 \triangleright_p t_2$ for the conclusion, where t_1 (t_2 resp.) is the type of the left (right resp.) premise. To understand the merged type, consider a scenario where the statement is executed in a context where permission p is *present*. Then the permission check succeeds and the statement **test**(p) c_1 **else** c_2 is equivalent to c_1. In this case, one would expect that the behavior of **test**(p) c_1 **else** c_2 would be equivalent to that of c_1. This is in fact captured by the equation $(t_1 \triangleright_p t_2)(P) = t_1(P)$ for all P such that $p \in P$, which holds by definition. A dual scenario arises when p is not in the permissions of the execution context.

In T-CALL, the callee function $B.f$ is assumed to be type checked beforehand and its type is given in the FT table. Here the function $B.f$ is called by A so the type of $B.f$ as seen by A should be a projection of the type given in $FT(B.f)$ on the permissions of A (given by $\Theta(A)$): $\pi_{\Theta(A)}(\bar{t}) \to \pi_{\Theta(A)}(t')$. Therefore the arguments for the function call should be typed as $\Gamma \vdash \bar{e} : \pi_{\Theta(A)}(\bar{t})$ and the return type (as viewed by A) should be dominated by the type of x, i.e., $\pi_{\Theta(A)}(t') \leq \Gamma(x)$.

Parameter laundering It is essential that in Rule T-CALL, the arguments $\bar{e}$ and the return value of the function call are typed according to the projection of $\bar{t}$ and t' on $\Theta(A)$. If they are instead typed with $\bar{t}$, then there is a potential implicit flow

$$\text{T-VAR}\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \text{T-OP}\frac{\Gamma \vdash e_1 : t \quad \Gamma; A \vdash e_2 : t}{\Gamma \vdash e_1 \ \textbf{op}\ e_2 : t} \qquad \text{T-SUB}_e\frac{\Gamma \vdash e : s \quad s \leq t}{\Gamma \vdash e : t} \qquad \text{T-SUB}_c\frac{\Gamma; A \vdash c : s \quad t \leq s}{\Gamma; A \vdash c : t}$$

$$\text{T-ASS}\frac{\Gamma \vdash e : \Gamma(x)}{\Gamma; A \vdash x := e : \Gamma(x)} \qquad \text{T-LETVAR}\frac{\Gamma \vdash e : s \quad \Gamma[x : s]; A \vdash c : t}{\Gamma; A \vdash \textbf{letvar}\ x = e\ \textbf{in}\ c : t} \qquad \text{T-IF}\frac{\Gamma \vdash e : t \quad \Gamma; A \vdash c_1 : t \quad \Gamma; A \vdash c_2 : t}{\Gamma; A \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : t}$$

$$\text{T-CP}\frac{\Gamma \uparrow_p; A \vdash c_1 : t_1 \quad \Gamma \downarrow_p; A \vdash c_2 : t_2}{\Gamma; A \vdash \textbf{test}(p)\ c_1\ \textbf{else}\ c_2 : t_1 \triangleright_p t_2} \qquad \text{T-WHILE}\frac{\Gamma \vdash e : t \quad \Gamma; A \vdash c : t}{\Gamma; A \vdash \textbf{while}\ e\ \textbf{do}\ c : t} \qquad \text{T-SEQ}\frac{\Gamma; A \vdash c_1 : t \quad \Gamma; A \vdash c_2 : t}{\Gamma; A \vdash c_1; c_2 : t}$$

$$\text{T-CALL}\frac{FT(B.f) = \overline{t} \to t' \quad \Gamma \vdash \overline{e} : \pi_{\Theta(A)}(\overline{t}) \quad \pi_{\Theta(A)}(t') \leq \Gamma(x)}{\Gamma; A \vdash x := \textbf{call}\ B.f(\overline{e}) : \Gamma(x)} \qquad \text{T-FUN}\frac{[\overline{x} : \overline{t}, r : t']; B \vdash c : s}{\vdash B.f(\overline{x})\{\textbf{init}\ r = 0\ \textbf{in}\ \{c; \textbf{return}\ r\}\} : \overline{t} \to t'}$$

Fig. 2. Typing rules for expressions, commands and functions.

```
A.f(x) {   // A does not have permission p
  init r = 0 in { r := call B.g(x); return r }
}

B.g(x) { // B does not have permission p
  init r = 0 in {
     test(p) r := 0 else r := x;
     return r
   }
}

C.getsecret() { // C has permission p
  init r = 0 in {
    test(p) r := P_INFO else r := 0;
    return r
  }
}

M.main() { // M has permission p
    init r = 0 in {
    letvar x_H = 0 in
    {x_H := C.getsecret(); r := call A.f(x_H)};
      return r
  }
}
```

Listing 3. An example illustrating the parameter laundering issue.

via a "parameter laundering" attack. To see why, consider the following alternative to T-CALL:

$$\text{T-CALL'}\frac{FT(B.f) = \overline{t} \to t' \quad \Gamma \vdash \overline{e} : \overline{t} \quad t' \leq \Gamma(x)}{\Gamma; A \vdash x := \textbf{call}\ B.f(\overline{e}) : \Gamma(x)}$$

Notice that the type of the argument $\overline{e}$ must match the type of the formal parameter of the function $B.f$. This is essentially what is adopted in BN system for method calls [11].

Let us consider the example in Listing 3. Let $\mathbf{P} = \{p\}$ and t be the base type $t = \{\emptyset \mapsto L, \{p\} \mapsto H\}$, where L and H are bottom and top levels respectively. Here we assume *P_INFO* is a sensitive value of security level H that needs to be protected, so function *C.getsecret* is required to have type $() \to t$. That is, only apps that have the required permission p may obtain the secret value. Suppose the permissions assigned to the apps are given by: $\Theta(A) = \Theta(B) = \emptyset, \Theta(C) = \Theta(M) = \{p\}$.

If we were to adopt the modified T-CALL' instead of T-CALL, then we can assign the following types to the above functions:

$$FT := \begin{cases} A.f & \mapsto \quad t \to \hat{L} \\ B.g & \mapsto \quad t \to \hat{L} \\ C.getsecret & \mapsto \quad () \to t \\ M.main & \mapsto \quad () \to \hat{L} \end{cases}$$

Notice that the return type of $M.main$ is $\hat{L}$ despite having a return value that contains sensitive value *P_INFO*. If we were to use T-CALL' in place of T-CALL, the above functions can be typed as shown in Fig. 3. Finally, still assuming T-CALL', a partial typing derivation for $M.main$ is given in Fig. 4.

As shown in Fig. 3, *B.g* can be given type $t \to \hat{L}$. Intuitively, it checks that the caller has permission p. If it does, then *B.g* returns 0 (non-sensitive), otherwise it returns the argument of the function (i.e., x). This is as expected and is sound, under the assumption that the security level of the content of x is dependent on the permissions of the caller. If the caller of *B.g* is the original creator of the content of x, then the assumption is trivially satisfied. The situation gets a bit tricky when the caller simply passes on the content it receives from another app to x. In our example, app A makes a call to *B.g*, and passes on the value of x it receives. In the run where *A.f* is called from *M.main*, the value of x is actually *sensitive* since it requires the permission p to acquire. However, when it goes through *A.f* to *B.g*, the value of x is perceived as *non-sensitive* by B, since the caller in this case (A) has no permissions. The use of the intermediary A in this case in effect launders the permissions associated with x. Therefore, if the rule T-CALL' is used in place of T-CALL, the call chain from *M.main* to *A.f* and finally to *B.g* can all be typed. This is correct in a setting where permissions are *propagated* along with calling context (e.g., [11]) however it is incorrect in the Android permission model II-A. To avoid the parameter laundering problem, our approach is to make sure that an app may only pass an argument to another function if the app itself is authorized to access the content of the argument in the first place, as formalized in the rule T-CALL.

With the correct typing rule for function calls, the function $A.f$ cannot be assigned type $t \to \hat{L}$, since that would require

$$\text{T-CALL'}\ \frac{FT(B.g) = t \to \hat{L} \qquad x : t, r : \hat{L} \vdash x : t \qquad \hat{L} \leq t}{x : t, r : \hat{L}; A \vdash r := \textbf{call}\ B.g(x) : \hat{L}}$$

$$\text{T-FUN}\ \frac{}{\vdash A.f(x)\{\textbf{init}\ r = 0\ \textbf{in}\ \{r := \textbf{call}\ B.g(x); \textbf{return}\ r\}\} : t \to \hat{L}}$$

$$\text{T-CP}\ \frac{x : t \uparrow_p, r : \hat{L} \uparrow_p \vdash r := 0 : \hat{L} \qquad x : t \downarrow_p, r : \hat{L} \downarrow_p \vdash r := x : \hat{L}}{x : t, r : \hat{L} \vdash \textbf{test}(p)\ r := 0\ \textbf{else}\ r := x : \hat{L} \triangleright_p \hat{L}}$$

$$\text{T-FUN}\ \frac{}{\vdash B.g(x)\{\textbf{init}\ r = 0\ \textbf{in}\ \{\textbf{test}(p)\ r := 0\ \textbf{else}\ r := x; \textbf{return}\ r\}\} : t \to \hat{L}}$$

Note that $t \uparrow_p = \hat{H}$, $t \downarrow_p = \hat{L} = \hat{L} \downarrow = \hat{L} \uparrow$ and $\hat{L} \triangleright_p \hat{L} = \hat{L}$.

$$\text{T-CP}\ \frac{r : t \uparrow_p \vdash r := \text{SECRET} : \hat{H} \qquad r : t \downarrow_p \vdash r := 0 : \hat{L}}{r : t \vdash \textbf{test}(p)\ r := \text{SECRET}\ \textbf{else}\ r := 0 : \hat{H} \triangleright_p \hat{L}}$$

$$\text{T-FUN}\ \frac{}{\vdash C.getsecret(\ )\{\textbf{init}\ r = 0\ \textbf{in}\ \{\textbf{test}(p)\ r := \text{SECRET}\ \textbf{else}\ r := 0; \textbf{return}\ r\}\} : () \to t}$$

Note that $\hat{H} \triangleright_p \hat{L} = t$.

Fig. 3. Typing derivations for functions A.f, B.g and C.getsecret

$$\text{T-LETVAR}\ \frac{r : \hat{L} \vdash 0 : t \qquad \text{T-SEQ}\ \dfrac{\Gamma; M \vdash x_H := \textbf{call}\ C.getsecret(\ ) : \hat{L} \qquad \Gamma; M \vdash r := \textbf{call}\ A.f(x_H) : \hat{L}}{r : \hat{L}, x_H : t; M \vdash x_H := \textbf{call}\ C.getsecret(\ ); r := \textbf{call}\ A.f(x_H) : \hat{L}}}{\text{T-FUN}\ \dfrac{r : \hat{L}\ ; M \vdash \textbf{letvar}\ x_H = 0\ \textbf{in}\ x_H := \textbf{call}\ C.getsecret(\ ); r := \textbf{call}\ A.f(x_H) : \hat{L}}{\vdash M.main(\ )\ \{\textbf{init}\ r = 0\ \textbf{in}\ \textbf{letvar}\ x_H = 0\ \textbf{in}\ \{\ x_H := \textbf{call}\ C.getsecret(\ );\ r := \textbf{call}\ A.f(x_H)\ \}\ \textbf{return}\ r\ \ \} : () \to \hat{L}}}$$

where $\Gamma = \{r : \hat{L}, x_H : t\}$ and the second and the third leaves are derived, respectively, as follows:

$$\text{T-CALL'}\ \frac{FT(C.getsecret) = () \to t \qquad \Gamma \vdash () : () \qquad t \leq \Gamma(x_H) = t}{\text{T-SUB}_c\ \dfrac{\Gamma; M \vdash x_H := \textbf{call}\ C.getsecret(\ ) : t}{\Gamma; M \vdash x_H := \textbf{call}\ C.getsecret(\ ) : \hat{L}}}$$

$$\text{T-CALL'}\ \frac{FT(A.f) = t \to \hat{L} \qquad \Gamma \vdash x_H : t \qquad \hat{L} \leq \Gamma(x_H) = t}{\Gamma; M \vdash r := \textbf{call}\ A.f(x_H) : \hat{L}}$$

Fig. 4. A typing derivation for function M.main

the instance of T-CALL (i.e., when making the call to $B.g$) in this case to satisfy the constraint:

$$x : t, r : \hat{L} \vdash x : \pi_{\Theta(A)}(t)$$

where $\pi_{\Theta(A)}(t) = \hat{L}$, which is impossible since $t \not\leq \hat{L}$. What this means is essentially that in our type system, information received by an app A from the parameters cannot be propagated by A to another app B, unless A is already authorized to access the information contained in the parameter. Note that this only restricts the propagation of such parameters to other apps; the app A can process the information internally without necessarily violating the typing constraints.

Finally, the reader may check that if we fix the type of $B.g$ to $t \to \hat{L}$ then $A.f$ can only be assigned type $\hat{L} \to \hat{L}$. In no circumstances can $M.main$ be typed, since the statement $x_H := C.getsecret()$ forces x_H to have type $\hat{H}$, and thus cannot be passed to $A.f$ as an argument.

F. Noninterference and Soundness

We first define an *indistinguishability* relation between evaluation environments. Such a definition typically assumes an observer who may observe values of variables at a certain security level. In the non-dependent setting, the security level of the observer is fixed, say at l_O, and valuations of variables at level l_O or below are required to be identical. In our setting, the security level of a variable in a function can vary depending on the permissions of the caller app (which may be the observer itself), so it may seem more natural to define indistinguishability in terms of the permission set assigned to the observer. However, we argue that such a definition is subsumed by the more traditional definition that is based on the security level of the observer. Assuming that the observer app is assigned a permission set P, then given two variables

$x : t$ and $y : t'$, the level of information that the observer can access through x and y is at most $t(P) \sqcap t'(P)$. In general the least upper bound of the security level that an observer with permission P has access to can be computed from the least upper bound of projections (along P) of the types of variables and the return types of functions in the system. In the following definition of indistinguishability, we simply assume that such an upper bound has been computed, and we will not refer explicitly to the permission set of the observer from which this upper bound is derived.

Definition II.9 *Given two evaluation environments η, η', a typing environment Γ, a security level $l_O \in \mathscr{L}$ of the observer, the* indistinguishability relation $=_\Gamma^{l_O}$ *is defined as:*

$$\eta =_\Gamma^{l_O} \eta' \textit{ iff} \forall x \in dom(\Gamma).\ \big(\Gamma(x) \leq \hat{l_O} \Rightarrow \eta(x) = \eta'(x)\big)$$

where $\eta(x) = \eta'(x)$ holds iff both sides of the equation are defined and equal, or both sides are undefined.

Note that in Definition II.9, η and η' may not have the same domain, but they must agree on their valuations for the variables in the domain of Γ. Note also that since base types are functions from permissions to security level, the security level l_O needs to be lifted to a base type in the comparison $\Gamma(x) \leq \hat{l_O}$. The latter implies that $\Gamma(x)(P) \leq l_O$ (in the latice $\mathscr{L}$) for every permission set P. If the base type of each variable assigns the same security level to every permission set (i.e., the security level is independent of the permissions), then our notion of indistinguishability coincides with the standard definition for the non-dependent setting.

We hereby give the definitions for well-typed property (Definition II.10) and non-interference for the type system (Defintion II.11 and Definition II.12), together with the final soundness conclusion (Theorem II.1). The detailed proofs are available in [13].

Definition II.10 *Let $\mathcal{S}$ be a system, and let FD, FT and Θ be its function declaration table, function type table, and permission assignments. We say $\mathcal{S}$ is* well-typed *iff for every function $A.f$, $\vdash FD(A.f) : FT(A.f)$ is derivable.*

Definition II.11 *A command c executed in app A is said to be* non-interferent *iff for all $\eta_1, \eta_1', \Gamma, P, l_O$, if $\eta_1 =_{\pi_P(\Gamma)}^{l_O} \eta_1'$, $\eta_1; A; P \vdash c \rightsquigarrow \eta_2$ and $\eta_1'; A; P \vdash c \rightsquigarrow \eta_2'$ then $\eta_2 =_{\pi_P(\Gamma)}^{l_O} \eta_2'$.*

Definition II.12 *Let $\mathcal{S}$ be a system. A function*

$$A.f(\overline{x})\big\{\textbf{\textit{init}}\ r = 0\ \textbf{\textit{in}}\ \{c; \textbf{\textit{return}}\ r\}\big\}$$

in $\mathcal{S}$ with $FT(A.f) = \bar{t} \to t'$ is non-interferent *if for all $\eta_1, \eta_1', P, v, l_O$, if the following hold:*

- $t'(P) \leq l_O$,
- $\eta_1 =_{\pi_P(\Gamma)}^{l_O} \eta_1'$, *where* $\Gamma = [\overline{x} : \bar{t}, r : t']$,
- $\eta_1; A; P \vdash c \rightsquigarrow \eta_2$, *and* $\eta_1'; A; P \vdash c \rightsquigarrow \eta_2'$,

then $\eta_2(r) = \eta_2'(r)$. The system $\mathcal{S}$ is non-interferent *iff all functions in $\mathcal{S}$ are non-interferent.*

Theorem II.1 *Well-typed systems are non-interferent.*

III. Type Inference

This section describes a decidable inference algorithm for the language in Section II-B. Section III-A firstly rewrites the typing rules (Fig. 2) in the form of permission trace rules (Fig. 5), then reduces the type inference into a constraint solving problem; Section III-B provides procedures to solve the generated constraints. Detailed definitions and proofs can be found in [13].

A. Constraint Generation

1) ***Permission Tracing****:* In an IPC between different apps (components), there may be multiple permission checks in a calling context. Therefore, to infer a security type for an expression, a command or a function, we need to track the applications of promotions $\Gamma \uparrow_p$ and demotions $\Gamma \downarrow_q$ in their typing derivations. To this end, we keep the applications symbolic and collect the promotions and demotions into a sequence. In other words, we treat them as a *sequence* of promotions $\uparrow_p$ and demotions $\downarrow_p$ applied on a typing environment Γ. For example, $(\Gamma \uparrow_p) \downarrow_q$ can be viewed as an *application* of the sequence $\uparrow_p \downarrow_q$ on Γ. The sequence of promotions and demotions is called a *permission trace* and denoted by Λ. The grammar of Λ is:

$$\Lambda\ ::=\ \oplus p :: \Lambda \mid \ominus p :: \Lambda \mid \epsilon \quad p \in \mathbf{P}$$

and its length, denoted by $len(\Lambda)$, is defined as:

$$len(\Lambda) = \begin{cases} 0 & \text{if } \Lambda = \epsilon \\ 1 + len(\Lambda') & \text{if } \Lambda = \circledcirc p :: \Lambda', \circledcirc \in \{\oplus, \ominus\} \end{cases}$$

Definition III.1 *Given a base type t and a permission trace Λ, the* application *of Λ to t, denoted by $t \cdot \Lambda$, is defined as:*

$$t \cdot \Lambda = \begin{cases} t & \text{if } \Lambda = \epsilon \\ (t \uparrow_p) \cdot \Lambda' & \text{if } \exists p, \Lambda', s.t.\ \Lambda = \oplus p :: \Lambda' \\ (t \downarrow_p) \cdot \Lambda' & \text{if } \exists p, \Lambda', s.t.\ \Lambda = \ominus p :: \Lambda' \end{cases}$$

We also extend the application of a permission trace Λ to a typing environment Γ (denoted by $\Gamma \cdot \Lambda$), such that $\forall x.\ (\Gamma \cdot \Lambda)(x) = \Gamma(x) \cdot \Lambda$. Based on permission traces, we give the definition of *partial subtyping relation*.

Definition III.2 *The* partial subtyping relation $\leq_\Lambda$, *which is the subtyping relation applied on the permission trace, is defined as $s \leq_\Lambda t$ iff. $s \cdot \Lambda \leq t \cdot \Lambda$.*

The application of permission traces to types preserves the subtyping relation.

Lemma III.1 *$\forall s, t \in \mathcal{T},\ s \leq t \implies s \leq_\Lambda t$ for all Λ.*

The following four lemmas discuss the impact of permission checking order on the same or different permissions.

Lemma III.2 *$\forall t \in \mathcal{T},\ p, q \in \mathbf{P}$ s.t. $p \neq q$, $t \cdot (\circledcirc p \circledast q) = t \cdot (\circledast q \circledcirc p)$, where $\circledcirc, \circledast, \in \{\oplus, \ominus\}$.*

Lemma III.3 *$\forall t \in \mathcal{T}, (t \cdot \circledcirc p) \cdot \Lambda = (t \cdot \Lambda) \cdot \circledcirc p$, where $\circledcirc \in \{\oplus, \ominus\}$ and $p \notin \Lambda$.*

Lemma III.4 *$\forall t \in \mathcal{T},\ p \in \mathbf{P}, (t \cdot \circledcirc p) \cdot \circledast p = t \cdot (\circledcirc p)$, where $\circledcirc, \circledast \in \{\oplus, \ominus\}$.*

Lemma III.5 $\forall t \in \mathcal{T},\ (t \cdot \Lambda) \cdot \Lambda = t \cdot \Lambda$.

Lemmas III.2 and III.3 state that the order of applications of promotions and demotions on *different* permissions does not affect the result. Lemmas III.4 and III.5 indicate that only the first application takes effect if there exist several (consecutive) applications of promotions and demotions on the *same* permission p. Therefore, we can safely keep only the first application, by removing the other applications on the same permission.

Let $occur(p, \Lambda)$ be the number of occurrences of p in Λ. We say Λ is *consistent* iff. $occur(p, \Lambda) \in \{0, 1\}$ for all $p \in \mathbf{P}$. In the remaining, we assume that all permission traces are consistent. Moreover, to ensure that the traces collected from the derivations of commands are consistent, we assume that in nested permission checks of a function definition, each permission is checked at most once.

*2) **Permission Trace Rules**:* We split the applications of the promotions and demotions into two parts (i.e., typing environments and permission traces), and move the subsumption rules (guarded by permission traces) for expressions and commands to where they are needed. This yields the syntax-directed typing rules, which we call the *permission trace rules* and are given in Fig. 5. The judgments of the trace rules are similar to those of typing rules, except that each trace rule is guarded by the permission trace Λ collected from the context, which keeps track of the adjustments of variables depending on the permission checks, and that the subtyping relation in the trace rules is the partial subtyping one $\leq_\Lambda$.

The next two lemmas show the trace rules are sound and complete with respect to the typing rules, i.e., an expression (command, function, resp.) is typable under the trace rules, if and only if it is typable under the typing rules.

Lemma III.6 *(a) If $\Gamma; \Lambda \vdash_t e : t$, then $\Gamma \cdot \Lambda \vdash e : (t \cdot \Lambda)$.*

(b) If $\Gamma; \Lambda; A \vdash_t c : t$, then $(\Gamma \cdot \Lambda); A \vdash c : (t \cdot \Lambda)$.

(c) If $\vdash_t B.f(\overline{x})\{\textbf{\textit{init}}\ r = 0\ \textbf{\textit{in}}\ \{c; \textbf{\textit{return}}\ r\}\} : \overline{t} \to t'$, then $\vdash B.f(\overline{x})\{\textbf{\textit{init}}\ r = 0\ \textbf{\textit{in}}\ \{c; \textbf{\textit{return}}\ r\}\} : \overline{t} \to t'$.

Lemma III.7 *(a) If $\Gamma \cdot \Lambda \vdash e : t \cdot \Lambda$, then there exists s such that $\Gamma; \Lambda \vdash_t e : s$ and $s \leq_\Lambda t$.*

(b) If $(\Gamma \cdot \Lambda); A \vdash c : t \cdot \Lambda$, then there exists s such that $\Gamma; \Lambda; A \vdash_t c : s$ and $t \leq_\Lambda s$.

(c) If $\vdash B.f(\overline{x})\{\textbf{\textit{init}}\ r = 0\ \textbf{\textit{in}}\ \{c; \textbf{\textit{return}}\ r\}\} : \overline{t} \to s$, then $\vdash_t B.f(\overline{x})\{\textbf{\textit{init}}\ r = 0\ \textbf{\textit{in}}\ \{c; \textbf{\textit{return}}\ r\}\} : \overline{t} \to s$.

*3) **Constraint Generation Rules**:* To infer types for functions in System $\mathcal{S}$, we assign a function type $\overline{\alpha} \to \beta$ for each function $A.f$ whose type is unknown and a type variable γ for each variable x with unknown type respectively, where $\overline{\alpha}, \beta, \gamma$ are fresh type variables. Then according to permission trace rules, we try to build a derivation for each function in $\mathcal{S}$, in which we collect the side conditions (*i.e.*, the partial subtyping relation $\leq_\Lambda$) needed by the rules. If the side conditions hold under a context, then $FD(A.f)$ is typed by $FT(A.f)$ under the same context for each function $A.f$ in $\mathcal{S}$.

To describe the side conditions (*i.e.*, $\leq_\Lambda$), we define the permission guarded constraints as follows:

$$\begin{array}{lcl} c & ::= & (\Lambda, t_l \leq t_r) \\ t_l & ::= & \alpha \mid t_g \mid t_l \sqcup t_l \mid \pi_P(t_l) \\ t_r & ::= & \alpha \mid t_g \mid t_r \sqcap t_r \mid t_r \rhd_p t_r \mid \pi_P(t_r) \end{array}$$

where Λ is a permission trace, α is a fresh type variable and t_g is a ground type.

A *type substitution* is a finite mapping from type variables to security types: $\theta ::= \epsilon \mid \alpha \mapsto t, \theta$

Definition III.3 *Given a constraint set C and a substitution θ, we say θ is a* solution *to C, denoted by $\theta \vDash C$, iff. for each $(\Lambda, t_l \leq t_r) \in C$, $t_l\theta \leq_\Lambda t_r\theta$ holds.*

The constraint generation rules are presented in Fig. 6, where FT_C is the extended function type table such that FT_C maps all function names to function types and their corresponding constraint sets. The judgments of the constraint rules are similar to those of trace rules, except that each rule generates a constraint set C, which consists of the side conditions needed by the typing derivation of $\mathcal{S}$. In addition, as the function call chains starting from a command are finite, the constraint generation will terminate.

The next two lemmas show the constraint rules are *sound* and *complete* with respect to permission trace rules, i.e., the constraint set generated by the derivation of an expression (command, function, resp.) under the constraint rules is solvable, if and only if an expression (command, function, resp.) is typable under trace rules.

Lemma III.8 *The following statements hold:*

(a) If $\Gamma; \Lambda \vdash_g e : t \rightsquigarrow C$ and $\theta \vDash C$, then $\Gamma\theta; \Lambda \vdash_t e : t\theta$.

(b) If $\Gamma; \Lambda; A \vdash_g c : t \rightsquigarrow C$ and $\theta \vDash C$, then $\Gamma\theta; \Lambda; A \vdash_t c : t\theta$.

(c) If $\vdash_g B.f(x)\{\textbf{\textit{init}}\ r = 0\ \textbf{\textit{in}}\ \{c; \textbf{\textit{return}}\ r\}\} : \overline{\alpha} \to \beta \rightsquigarrow C$ and $\theta \vDash C$, then

$$\vdash_t B.f(x)\{\textbf{\textit{init}}\ r = 0\ \textbf{\textit{in}}\ \{c; \textbf{\textit{return}}\ r\}\} : \overline{\theta(\alpha)} \to \theta(\beta).$$

Lemma III.9 *The following statements hold:*

(a) If $\Gamma; \Lambda \vdash_t e : t$, then there exist Γ', t', C, θ s.t. $\Gamma'; \Lambda \vdash_g e : t' \rightsquigarrow C$, $\theta \vDash C$, $\Gamma'\theta = \Gamma$ and $t'\theta = t$.

(b) If $\Gamma; \Lambda; A \vdash_t c : t$, then there exist Γ', t', C, θ s.t. $\Gamma'; \Lambda; A \vdash_g c : t' \rightsquigarrow C$, $\theta \vDash C$, $\Gamma'\theta = \Gamma$ and $t'\theta = t$.

(c) If $\vdash_t B.f(\overline{x})\{\textbf{\textit{init}}\ r = 0\ \textbf{\textit{in}}\ \{c; \textbf{\textit{return}}\ r\}\} : \overline{t_p} \to t_r$, then there exist α, β, C, θ s.t.

$$\vdash_g B.f(\overline{x})\{\textbf{\textit{init}}\ r = 0\ \textbf{\textit{in}}\ \{c; \textbf{\textit{return}}\ r\}\} : \overline{\alpha} \to \beta \rightsquigarrow C,$$

$\theta \vDash C$, and $(\overline{\alpha} \to \beta)\theta = \overline{t_p} \to t_r$, where α, β are fresh type variables.

Recall the function *getInfo* in Listing 2 and assume that *getInfo* is defined in app A (thus *A.getInfo*) and called by app B through the function *fun* (thus *B.fun*). The rephrased program is shown in Listing 4, where l_1, l_2 are the types for *loc* and *id* respectively, $\Theta(B) = \{q\}$, and $l_1 \sqcup l_2 = H$. Let us apply the constraint generation rules in Fig. 6 on each function,

$$\text{TT-VAR}\frac{}{\Gamma;\Lambda\vdash_t x:\Gamma(x)} \qquad \text{TT-OP}\frac{\Gamma;\Lambda\vdash_t e_1:t_1 \quad \Gamma;\Lambda\vdash_t e_2:t_2}{\Gamma;\Lambda\vdash_t e_1\ \mathbf{op}\ e_2:t_1\sqcup t_2} \qquad \text{TT-ASS}\frac{\Gamma;\Lambda\vdash_t e:t \quad t\leq_\Lambda \Gamma(x)}{\Gamma;\Lambda\vdash_t x:=e:\Gamma(x)}$$

$$\text{TT-IF}\frac{\Gamma;\Lambda\vdash_t e:t \quad \Gamma;\Lambda;A\vdash_t c_1:t_1 \quad \Gamma;\Lambda;A\vdash_t c_2:t_2 \quad t\leq_\Lambda t_1\sqcap t_2}{\Gamma;\Lambda;A\vdash_t \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2:t_1\sqcap t_2} \qquad \text{TT-SEQ}\frac{\Gamma;\Lambda;A\vdash_t c_1:t_1 \quad \Gamma;\Lambda;A\vdash_t c_2:t_2}{\Gamma;\Lambda;A\vdash_t c_1;c_2:t_1\sqcap t_2}$$

$$\text{TT-LETVAR}\frac{\Gamma;\Lambda\vdash_t e:s \quad \Gamma[x:s'];\Lambda;A\vdash_t c:t \quad s\leq_\Lambda s'}{\Gamma;\Lambda;A\vdash_t \mathbf{letvar}\ x=e\ \mathbf{in}\ c:t} \qquad \text{TT-WHILE}\frac{\Gamma;\Lambda\vdash_t e:s \quad \Gamma;\Lambda;A\vdash_t c:t \quad s\leq_\Lambda t}{\Gamma;\Lambda;A\vdash_t \mathbf{while}\ c\ \mathbf{do}\ e:t}$$

$$\text{TT-CALL}\frac{FT(B.f)=\overline{t}\to t' \quad \Gamma;\Lambda\vdash_t \overline{e}:\overline{s} \quad \overline{s}\leq_\Lambda \overline{\pi_{\Theta(A)}(t)} \quad \pi_{\Theta(A)}(t')\leq_\Lambda \Gamma(x)}{\Gamma;\Lambda;A\vdash_t x:=\mathbf{call}\ B.f(\overline{e}):\Gamma(x)}$$

$$\text{TT-CP}\frac{\Gamma;\Lambda::\oplus p;A\vdash_t c_1:t_1 \quad \Gamma;\Lambda::\ominus p;A\vdash_t c_2:t_2}{\Gamma;\Lambda;A\vdash_t \mathbf{test}(p)\ c_1\ \mathbf{else}\ c_2:t_1\triangleright_p t_2} \qquad \text{TT-FUN}\frac{[\overline{x}:\overline{t},r:t'];\epsilon;B\vdash_t c:s}{\vdash_t B.f(\overline{x})\{\mathbf{init}\ r=0\ \mathbf{in}\ \{c;\mathbf{return}\ r\}\}:\overline{t}\to t'}$$

Fig. 5. Permission trace rules for expressions, commands and functions

$$\text{TG-OP}\frac{\Gamma;\Lambda\vdash_g e_1:t_1\rightsquigarrow C_1 \quad \Gamma;\Lambda\vdash_g e_2:t_2\rightsquigarrow C_2}{\Gamma;\Lambda\vdash_g e_1\ \mathbf{op}\ e_2:t_1\sqcup t_2\rightsquigarrow C_1\cup C_2} \qquad \text{TG-ASS}\frac{\Gamma;\Lambda\vdash_g e:t\rightsquigarrow C}{\Gamma;\Lambda;A\vdash_g x:=e:\Gamma(x)\rightsquigarrow C\cup\{(\Lambda,t\leq\Gamma(x))\}}$$

$$\text{TG-VAR}\frac{}{\Gamma;\Lambda\vdash_g x:\Gamma(x)\rightsquigarrow\emptyset} \qquad \text{TG-LETVAR}\frac{\begin{array}{c}\Gamma;\Lambda\vdash_g e:s\rightsquigarrow C_1\\ \Gamma[x:\alpha];\Lambda;A\vdash_g c:t\rightsquigarrow C_2\\ C=C_1\cup C_2\cup\{(\Lambda,s\leq\alpha)\}\end{array}}{\Gamma;\Lambda;A\vdash_g \mathbf{letvar}\ x=e\ \mathbf{in}\ c:t\rightsquigarrow C}$$

$$\text{TG-WHILE}\frac{\begin{array}{c}\Gamma;\Lambda\vdash_g e:s\rightsquigarrow C\\ \Gamma;\Lambda;A\vdash_g c:t\rightsquigarrow C'\end{array}}{\Gamma;\Lambda;A\vdash_g \mathbf{while}\ e\ \mathbf{do}\ c:t\rightsquigarrow C\cup C'\cup\{(\Lambda,s\leq t)\}} \qquad \text{TG-SEQ}\frac{\begin{array}{c}\Gamma;\Lambda;A\vdash_g c_1:t_1\rightsquigarrow C_1\\ \Gamma;\Lambda;A\vdash_g c_2:t_2\rightsquigarrow C_2\end{array}}{\Gamma;\Lambda;A\vdash_g c_1;c_2:t_1\sqcap t_2\rightsquigarrow C_1\cup C_2}$$

$$\text{TG-IF}\frac{\begin{array}{c}\Gamma;\Lambda;A\vdash_g c_1:t_1\rightsquigarrow C_1\\ \Gamma;\Lambda;A\vdash_g c_2:t_2\rightsquigarrow C_2\\ \Gamma;\Lambda\vdash_g e:t\rightsquigarrow C_e\\ C=C_e\cup C_1\cup C_2\cup\{(\Lambda,t\leq t_1\sqcap t_2)\}\end{array}}{\Gamma;\Lambda;A\vdash_g \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2:t_1\sqcap t_2\rightsquigarrow C} \qquad \text{TG-CALL}\frac{\begin{array}{c}FT_C(B.f)=(\overline{t}\to t',C_f)\\ \Gamma;\Lambda\vdash_g \overline{e}:\overline{s}\rightsquigarrow\bigcup\overline{C_e}\\ C_a=\{(\Lambda,\overline{s}\leq\overline{\pi_{\Theta(A)}(t)}),(\Lambda,\pi_{\Theta(A)}(t')\leq\Gamma(x))\}\end{array}}{\Gamma;\Lambda;A\vdash_g x:=\mathbf{call}\ B.f(\overline{e}):\Gamma(x)\rightsquigarrow C_f\cup\bigcup\overline{C_e}\cup C_a}$$

$$\text{TG-CP}\frac{\begin{array}{c}\Gamma;\Lambda::\oplus p;A\vdash_g c_1:t_1\rightsquigarrow C_1\\ \Gamma;\Lambda::\ominus p;A\vdash_g c_2:t_2\rightsquigarrow C_2\end{array}}{\Gamma;\Lambda;A\vdash_g \mathbf{test}(p)\ c_1\ \mathbf{else}\ c_2:t_1\triangleright_p t_2\rightsquigarrow C_1\cup C_2} \qquad \text{TG-FUN}\frac{[\overline{x}:\overline{\alpha},r:\beta];\epsilon;B\vdash_g c:s\rightsquigarrow C}{\vdash_g B.f(x)\{\mathbf{init}\ r=0\ \mathbf{in}\ \{c;\mathbf{return}\ r\}\}:\overline{\alpha}\to\beta\rightsquigarrow C}$$

Fig. 6. Constraint generation rules for expressions, commands and functions, given function type table FT_C.

yielding the constraint sets C_A and C_B

$$\begin{array}{rcl} C_A & = & \{(\oplus p\oplus q,l_1\leq\alpha),(\oplus p\ominus q,L\leq\alpha),\\ & & (\ominus p\oplus q,H\leq\alpha),(\ominus p\ominus q,L\leq\alpha)\}\\ C_B & = & \{(\epsilon,L\leq\gamma),(\oplus p,L\leq\beta),(\ominus p,\pi_{\Theta(B)}(\alpha)\leq\gamma),\\ & & (\epsilon,L\leq\beta),(\epsilon,\gamma\leq\beta)\}\end{array}$$

and the types $t_A=()\to\alpha$ and $t_B=()\to\beta$ for the functions *getInfo* and *fun*[3] respectively. Thus, the constraint set C_{eg} for the whole program is $C_A\cup C_B$.

B. Constraint Solving

We now present an algorithm for solving the constraints generated by the rules in Fig. 6. For these constraints, both types appearing on the two sides of subtyping are guarded by the *same* permission trace. But during the process of solving these constraints, new constraints, whose two sides of subtyping are guarded by *different* traces, may be generated. Take the constraint $(\Lambda,\pi_P(t_l)\leq\pi_Q(\alpha))$ for example, t_l is indeed guarded by P while α is guarded by Q, where P and Q are different permission sets. So for constraint solving, we use a generalized version of the permission guarded constraints, allowing types on the two sides to be guarded by different permission traces: $((\Lambda_l,t_l)\leq(\Lambda_r,t_r))$ where $t_l\neq t_r$. Likewise, a *solution* to a generalized constraint set C is a substitution θ, denoted by $\theta\vDash C$, such that for each $((\Lambda_l,t_l)\leq(\Lambda_r,t_r))\in C$, $(t_l\theta\cdot\Lambda_l)\leq(t_r\theta\cdot\Lambda_r)$ holds.

It is easy to transform a permission guarded constraint set C into a generalized constraint set C': by rewriting each $(\Lambda,t_l\leq t_r)$ as $((\Lambda,t_l)\leq(\Lambda,t_r))$. Moreover, it is trivial that $\theta\vDash C\iff\theta\vDash C'$. Therefore, we focus on solving generalized constraints in the following. For example, the

[3]Indeed, the constraint set for *fun* is $C_A\cup C_B$, but here we focus on the constraints generated by the function itself.

```
A.getInfo() {
    init r in { // Γ(r) = α
        test(p) {
            test(q) r = loc; // (⊕p ⊕ q, l1 ≤ α)
            else r = ""; // (⊕p ⊖ q, L ≤ α)
        } else {
            test(q) r = id++loc; // (⊖p ⊕ q, H ≤ α)
            else r = ""; // (⊖p ⊖ q, L ≤ α)
        }
        return r;
    }
}
B.fun() { // B has permission q
    init r in { // Γ(r) = β
        letvar x = "" in { // Γ(x) = γ, (ϵ, L ≤ γ)
            test(p) r = 0; // (⊕p, L ≤ β)
            else x = call A.getInfo();
            //FT_C(A.getInfo) = () → α, (⊖p, π_Θ(B)(α) ≤ γ)
            if x == "" then r = 0; // (ϵ, L ≤ β)
            else r = 1; // (ϵ, L ≤ β), (ϵ, γ ≤ β)
        }
        return r;
    }
}
```

Listing 4. The example in Listing 2 in a calling context.

constraint set C_{eg} can be rewritten as

$$C_{eg} = \{((\epsilon, L) \leq (\epsilon, \gamma)), ((\ominus p, \pi_{\Theta(B)}(\alpha)) \leq (\ominus p, \gamma)),$$
$$((\oplus p, L) \leq (\oplus p, \beta)), ((\epsilon, L) \leq (\epsilon, \beta)), ((\epsilon, \gamma) \leq (\epsilon, \beta)),$$
$$((\oplus p \oplus q, l_1) \leq (\oplus p \oplus q, \alpha)), ((\oplus p \ominus q, L) \leq (\oplus p \ominus q, \alpha)),$$
$$((\ominus p \oplus q, H) \leq (\ominus p \oplus q, \alpha)), ((\ominus p \ominus q, L) \leq (\ominus p \ominus q, \alpha))\}$$

Given a permission set P and a permission trace Λ, we say P *entails* Λ, denoted by $P \vDash \Lambda$, iff. $\forall \oplus p \in \Lambda.\ p \in P$ and $\forall \ominus p \in \Lambda.\ p \notin P$. A permission trace Λ is *satisfiable*, denoted by $\Delta(\Lambda)$, iff. there exists a permission set P such that $P \vDash \Lambda$. We write Λ_P for the permission trace that only P can entail.

A permission trace Λ can be considered as a boolean logic formula on permissions, where $\oplus$ and $\ominus$ denote positive and negative respectively, and ϵ denotes $True$. In the remaining we shall use the logic connectives on permission traces freely. We also adopt the disjunctive normal form, i.e., a disjunction of conjunctive permissions, and denote it as $dnf(\cdot)$. For example, $dnf((\oplus p) \wedge \neg(\oplus q \wedge \ominus r)) = (\oplus p \wedge \ominus q) \vee (\oplus p \wedge \oplus r)$.

The constraint solving consists of three steps: 1) *decompose* types in constraints into ground types and type variables; 2) *saturate* the constraint set by the transitivity of the subtyping relation; 3) solve the final constraint set by *merging* the lower and upper bounds of same variables and *unifying* them to emit a solution.

1) ***Decomposition****:* The first step is to decompose the types into the simpler ones, i.e., type variables and ground types, according to their structures. This decomposition is defined via the function dec that takes a constraint $((\Lambda_l, t_l, \Lambda_r, t_r)$ for short) as input and generates a constraint set or $\perp$ (denoting unsastifiable):

$dec((\Lambda_l, t_l, \Lambda_r, t_r)) =$

if $t_l \cong t_l^1 \sqcup t_l^2$, **then return** $dec((\Lambda_l, t_l^1, \Lambda_r, t_r)) \cup dec((\Lambda_l, t_l^2, \Lambda_r, t_r))$

if $t_l \cong \pi_P(t)$, **then return** $dec((\Lambda_P, t, \Lambda_r, t_r))$

if $t_r \cong t_r^1 \sqcap t_r^2$, **then return** $dec((\Lambda_l, t_l, \Lambda_r, t_r^1)) \cup dec((\Lambda_l, t_l, \Lambda_r, t_r^2))$

if $t_r \cong \pi_P(t)$, **then return** $dec((\Lambda_l, t_l, \Lambda_P, t))$

if $t_r \cong t_r^1 \triangleright_p t_r^2$, **return** $dec((\Lambda_l :: \oplus p, t_l, \Lambda_r :: \oplus p, t_r^1)) \cup dec((\Lambda_l :: \ominus p, t_l, \Lambda_r :: \ominus p, t_r^2))$

if both t_l and t_r are ground, **return** $\emptyset$ if $t_l \cdot \Lambda_l \leq t_r \cdot \Lambda_r$ or $\perp$ otherwise

return $\{(\Lambda_l, t_l, \Lambda_r, t_r)\}$

After decomposition, constraints have one of the forms:

$$((\Lambda_l, \alpha) \leq (\Lambda_r, t_g)),\ ((\Lambda_l, t_g) \leq (\Lambda_r, \beta)),\ ((\Lambda_l, \alpha) \leq (\Lambda_r, \beta))$$

Considering the constraint set C_{eg}, only the constraint

$$((\ominus p, \pi_{\Theta(B)}(\alpha)) \leq (\ominus p, \gamma))$$

needs to be decomposed, yielding $((\ominus p \oplus q, \alpha) \leq (\ominus p, \gamma))$.

2) ***Saturation****:* Considering a variable α, to ensure any lower bound (e.g., $((\Lambda_l, t_l) \leq (\Lambda_1, \alpha))$) is "smaller" than any of its upper bound (e.g., $((\Lambda_2, \alpha) \leq (\Lambda_r, t_r))$, we need to saturate the constraint set by adding these conditions. However, since our constraints are guarded by permission traces, we need to consider lower-upper bound relations only when the traces of the variable α can be entailed by the same permission set, i.e., their intersection is satisfiable. In that case, we extend the traces of both the lower and upper bound constraints such that the traces of α are the same (*i.e.*, $\Lambda_1 \wedge \Lambda_2$), by adding the missing traces (*i.e.*, $\Lambda_1 \wedge \Lambda_2 - \Lambda_1$ for lower bound constraint while $\Lambda_1 \wedge \Lambda_2 - \Lambda_2$ for the upper one, where $-$ denotes set difference). This is done by the function sat defined as follows:

$sat((\Lambda_l, t_l, \Lambda_1, \alpha), (\Lambda_2, \alpha, \Lambda_r, t_r)) =$

if $\Lambda_1 \wedge \Lambda_2$ is satisfiable, **then** let $\Lambda_l' = \Lambda_l \wedge (\Lambda_1 \wedge \Lambda_2 - \Lambda_1)$ and $\Lambda_r' = \Lambda_r \wedge (\Lambda_1 \wedge \Lambda_2 - \Lambda_2)$ **in** $dec((\Lambda_l', t_l, \Lambda_r', t_r)))$

return $\emptyset$

Assume that there is an order $<$ on type variables and the smaller variable has a higher priority. If two variables α, β with $O(\alpha) < O(\beta)$ (the orderings) are in the same constraint $\beta \leq \alpha$, we consider the larger variable β is a bound for the smaller one α, but not vice-versa. There is a special case where both variables on two sides are the same, *e.g.*, $((\Lambda, \alpha) \leq (\Lambda', \alpha))$. In that case, we regroup all the trace of the variable α as the trace set $\{\Lambda_i \mid i \in I\}$ such that the set is full (*i.e.*, $\bigvee_{i \in I} \Lambda_i = \epsilon$) and disjoint (*i.e.*, $\forall i, j \in I. i \neq j \Rightarrow \neg\Delta(\Lambda_i \wedge \Lambda_j)$), and rewrite the constraints of α w.r.t. the set $\{\Lambda_i \mid i \in I\}$. Then we treat each (Λ_i, α) as different fresh variables α_i. Therefore, there are no loops like: $(\Lambda, \alpha) \leq \ldots \leq (\Lambda', \alpha)$, with the ordering.

Let us consider the constraint set C_{eg} and assume that the order on variables is $O(\alpha) < O(\gamma) < O(\beta)$. There are four lower bounds and one upper bounds for α. But only the lower bound $((\ominus p \oplus q, H) \leq (\ominus p \oplus q, \alpha))$ shares the same satisfiable trace with the upper bound. So we saturate the set with the constraint $((\ominus p \oplus q, H) \leq (\ominus p, \gamma))$. Likewise, there are two

lower bounds (one of which is newly generated) and one upper bound for γ. Each lower bound has a satisfiable intersected trace with the upper bound, which yields the following constraints $((\epsilon, L) \leq (\epsilon, \beta))$ and $((\ominus p \oplus q, H) \leq (\ominus p, \beta))$ (extended by $\ominus p$). While there are no upper bounds for β, so no constraints are generated. After saturation, the example set C_{eg} is

$$\{((\oplus p \oplus q, l_1) \leq (\oplus p \oplus q, \alpha)), ((\oplus p \ominus q, L) \leq (\oplus p \ominus q, \alpha)),$$
$$((\ominus p \oplus q, H) \leq (\ominus p \oplus q, \alpha)), ((\ominus p \ominus q, L) \leq (\ominus p \ominus q, \alpha)),$$
$$((\ominus p \oplus q, \alpha) \leq (\ominus p, \gamma)), ((\epsilon, L) \leq (\epsilon, \gamma)),$$
$$((\ominus p \oplus q, H) \leq (\ominus p, \gamma)), ((\epsilon, \gamma) \leq (\epsilon, \beta)),$$
$$((\oplus p, L) \leq (\oplus p, \beta)), ((\epsilon, L) \leq (\epsilon, \beta)), ((\ominus p \oplus q, H) \leq (\ominus p, \beta))\}$$

*3) **Unification**:* Since our constraints are guarded by permission traces, we need to consider the satisfiability of (any subset of) the permission traces of a variable α under any permission set when constructing a type for it. Let us consider a variable α and assume that the constraints on it to be solved are $\{((\Lambda_i^l, t_i^l) \leq (\Lambda_i, \alpha))\}_{i \in I}$ (*i.e.*, the lower bounds) and $\{((\Lambda_j, \alpha) \leq (\Lambda_j^r, t_j^r))\}_{j \in J}$ (*i.e.*, the upper bounds). This indicates that under a permission set P, α can take such a type t that is bigger than $t_i^l \cdot \Lambda_i^l$ if $P \vDash \Lambda_i$ and is smaller than $t_j^r \cdot \Lambda_j^r$ if $P \vDash \Lambda_j$. Consequently, t should be bigger than the union $\bigsqcup_{i \in I'} t_i^l \cdot \Lambda_i^l$ and smaller than the intersection $\bigsqcap_{j \in J'} t_j^r \cdot \Lambda_j^r$ if all the traces $\Lambda_i \in I'$ and $\Lambda_j \in J'$ are entailed by P. In other words, α can take any type ranging from $\bigsqcup_{i \in I'} t_i^l \cdot (\Lambda_i^l \wedge \Lambda_i')$ to $\bigsqcap_{j \in J'} t_j^r \cdot (\Lambda_j^r \wedge \Lambda_j')$ if only the intersection trace $\bigwedge_{i \in I'} \Lambda_i \wedge \bigwedge_{j \in J'} \Lambda_j$ is satisfiable, which indicates that α is equivalent to $(\bigsqcup_{i \in I'} t_i^l \cdot (\Lambda_i^l \wedge \Lambda_i') \sqcup \alpha') \sqcap \bigsqcap_{j \in J'} t_j^r \cdot (\Lambda_j^r \wedge \Lambda_j')$, where Λ_i' and Λ_j' are the missing traces to extend Λ_i and Λ_j to the intersection trace respectively, and α' is a fresh variable. The type above is exactly what we want. We define the construction of the type above via the function $merge$:

$merge(\{(\Lambda_i^l, t_i^l, \Lambda_i, \alpha)\}_{i \in I}, \{(\Lambda_j, \alpha, \Lambda_j^r, t_j^r)\}_{j \in J}) =$
  let $\phi(I', J') = \{\Lambda \in dnf(\bigwedge_{i \in I'} \Lambda_i \wedge \bigwedge_{j \in J'} \Lambda_j \wedge \bigwedge_{i \in I \setminus I'} \neg \Lambda_i \wedge \bigwedge_{j \in J \setminus J'} \neg \Lambda_j) \mid \Delta(\Lambda)\}$ **in**
  let $t^{\sqcup}_{I',\Lambda} = \bigsqcup_{i \in I'} t_i^l \cdot (\Lambda_i^l \wedge (\Lambda - \Lambda_i))$ **in**
  let $t^{\sqcap}_{J',\Lambda} = \bigsqcap_{j \in J'} t_j^r \cdot (\Lambda_j^r \wedge (\Lambda - \Lambda_j))$ **in**
  $\{\Lambda \mapsto (t^{\sqcup}_{I',\Lambda} \sqcup \alpha_\Lambda) \sqcap t^{\sqcap}_{J',\Lambda}\}_{I' \subseteq I, J' \subseteq J, \Lambda \in \phi(I', J')}$

(with the convention $t^{\sqcup}_{\emptyset,\Lambda} = L$ and $t^{\sqcap}_{\emptyset,\Lambda} = H$.)

Moreover, due to the absence of loops in constraints and that the variables are in order, we can solve the constraints in reverse order on variables by unification. The unification algorithm $unify$ is presented as follows.

$unify(C) =$
  let $subst\ \theta\ ((\Lambda_l, t_l, \Lambda_r, t_r)) = ((\Lambda_l, t_l\theta, \Lambda_r, t_r\theta))$ **in**
  select $\{(\Lambda_i^l, t_i^l, \Lambda_i, \alpha)\}_{i \in I}$ and $\{(\Lambda_j, \alpha, \Lambda_j^r, t_j^r)\}_{j \in J}$ for the maximum variable α **if** exists, $merge$ them as t_α
    let C' be the remaining constraints **in**
    let $C'' = List.map\ (subst\ [\alpha \mapsto t_\alpha])\ C'$ **in**
    let $\theta' = unify(C'')$ **in** $\theta'[\alpha \mapsto t_\alpha]$
  else return []

Let us consider the constraint set C_{eg} again and take the constraints on the maximum variable β, which are the following set without any upper bounds

$$\{((\oplus p, L) \leq (\oplus p, \beta)), ((\epsilon, L) \leq (\epsilon, \beta)), ((\ominus p \oplus q, H) \leq (\ominus p, \beta))\}$$

By applying the function $merge$, we construct for β the type $t_\beta = \{\oplus p \mapsto (L \sqcup \beta') \sqcap H, \ominus p \mapsto H\}$, where only the common traces (*i.e.*, $\oplus p$ and $\ominus p$) for the subsets $\{\epsilon, \oplus p\}$ and $\{\epsilon, \ominus p\}$ are satisfiable, and β' is a fresh variable. For simplicity, we pick the least upper bound as possible when constructing types. So we take $\{\oplus p \mapsto L, \ominus p \mapsto H\}$ as t_β instead. Next, we substitute t_β for all the occurrences of β in the remaining constraints and continue with the constraints on γ and α. Finally, the types constructed for γ and α are $t_\gamma = t_\beta$ and $t_\alpha = \{\oplus p \oplus q \mapsto l_1, \oplus p \ominus q \mapsto L, \ominus p \oplus q \mapsto H, \ominus p \ominus q \mapsto L\}$, respectively. Therefore, the types we infer for *A.getInfo* and *B.fun* are $() \to t_\alpha$ and $() \to t_\beta$, respectively.

Let sol be the function for the constraint solving algorithm, that is, $sol(C) = unify(sat(dec(C)))$. It is provable that the constraint solving algorithm is sound and complete.

To conclude, an expression (command, function, resp.) is typable, iff it is derivable under the constraint rules with a solvable constraint set by our algorithm. Therefore, our type inference system is sound and complete. Moreover, as the function call chains are finite, the constraint generation terminates with a finite constraint set, which can be solved by our algorithm in finite steps. Thus, our type inference system terminates.

Theorem III.1 *The type inference system is sound, complete and decidable.*

IV. Related work

There is a large body of work on language-based information flow security. We shall discuss only closely related work.

We have discussed extensively the work by Banerjee and Naumann [11] and highlights the major differences between our work and theirs in Section I.

Flow-sensitive and value-dependent information flow type systems provide a general treatment of security types that may depend on other program variables or execution contexts [18]–[36]. Hunt and Sands [36] proposed a flow-sensitive type system where order of execution is taken into account in the analysis, and demonstrated that the system is precise but can be simply described. Mantel et. al. [30] introduced a rely-guarantee style reasoning for information flow security in which the same variable can be assigned different security levels depending on whether some assumption is guaranteed, which is similar to our notion of permission-dependent security types. Li and Zhang [35] proposed both flow-sensitive and path-sensitive information flow analysis with program transformation techniques and dependent types. Information flow type systems that may depend on execution contexts have been considered in work on program synthesis [19] and dynamic information flow control [29]. Our permission context can be seen as a special instance of execution context, however, our intended applications and settings are different from [19], [29], and issues such as parameter laundering does not occur in their setting. Lourenço and Caires [26] provided a precise

dependent type system where security labels can be indexed by data structures, which can be used to encode the dependency of security labels on other values in the system. It may be possible to encode our notion of security types as a dependent type in their setting, by treating permission sets explicitly as an additional parameter to a function or a service, and to specify security levels of the output of the function as a type dependent on that parameter. Currently it is not yet clear to us how one could give a general construction of the index types in their type system that would correspond to our security types, and how the merge operator would translate to their dependent type constructors, among other things. We leave the exact correspondence to the future work.

Recent research on information flow has also been conducted to deal with Android security issues ([2], [16], [37]–[41]). SCandroid [2], [41] is a tool automating security certification of Android apps that focuses on typing communication between applications. Unlike our work, they do not consider implicit flows, and do not take into account access control in their type system. Ernst et al [37] proposed a verification model, SPARTA, for use in app stores to guarantee that apps are free of malicious information flows. Their approach requires the collaboration between software vendor and app store auditor and the additional modification of Android permission model to fit for their Information Flow Type-checker; soundness proof is also absent. Our work is done in the context of providing information flow security *certificates* for Android applications, following the Proof-Carrying-Code architecture by Necula and Lee [42] and does not require extra changes on existing Android application supply chain systems.

V. Conclusion and Future Work

We have provided a lightweight yet precise type system featuring Android permission model for enforcing secure information flow in an imperative language and proved its soundness with respect to non-interference. Compared to existing work, our type system can specify a broader range of security policies, including non-monotonic ones. We have also proposed a decidable type inference algorithm by reducing it to a constraint solving problem.

We next discuss briefly several directions for future work.

The immediate one is to extend our system to richer programming languages. We have been working on adding another security typing for global variables. The addition of global variables presents a potential side channel, i.e., when they are *written* and *read* by apps with different permission contexts, so they need to be treated differently than local variables. Other extensions include object-oriented feature (like [43]), exceptions (like [44]), etc.

We also plan to apply our type system to real Android applications to enforce permission-dependent information flow policies. A main challenge is to facilitate type inference so that a programmer does not need to type every variable and instead focuses only on policy specifications of a service. To enable this, we need to be able to extract all permissions relevant to an app and to identify all commands relevant to permission checking in an app. The former is straightforward since the permissions that can be granted to an app is statically specified in the app's manifest file. For the latter, the permission checking code segments (typically library function calls) can be located with pre-processed static analyses (e.g., [3], [4]).

Another interesting direction is in modeling runtime permission request. From Android 6.0 and above, several permissions are classified as *dangerous permissions* and granting of these permissions is subject to users' approval at runtime. This makes enforcing non-monotonic policies impossible in some cases, e.g., when a policy specifies the absence of a dangerous permission in releasing sensitive information. However, an app can only request for a permission it has explicitly declared in the manifest file, so to this extent, we can statically determine whether a permission request is definitely *not* going to be granted (as it is absent from the manifest), and whether it can *potentially* be granted. And fortunately (but unfortunately from a security perspective) the typical scenarios are that users grant all the requested permissions during runtime when requested (in order to gain a better user experience with the app). Therefore one can assume optimistically that all permissions in the manifest are finally granted. In the future, we plan to resolve this issue with weaker assumptions. One feasible approach is to model dangerous permissions in a typing environment separately and allow policies to be non-monotonic on non-dangerous permissions only.

Lastly, our eventual goal is to translate source code typing into Dalvik bytecode typing, following a similar approach done by Gilles Barthe et al [44]–[46] from Java source to JVM bytecode. The key idea that we describe in the paper, i.e., precise characterizations of security of IPC channels that depends on permission checks, can still be applied to richer type systems such as those used in the Cassandra project [39] or Gunadi's type system [40]. We envision our implementation can piggyback on, say, Cassandra system to improve the coverage of typable applications.

Acknowledgements

Zhiwu Xu was partially supported by National Natural Science Foundation of China (No. 61502308 and No. 61772347), Science and Technology Foundation of Shenzhen City (No. JCYJ20170302153712968).

References

[1] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *IEEE Security and Privacy*, vol. 7, no. 1, pp. 50–57, Jan. 2009.

[2] A. P. Fuchs, A. Chaudhuri, and J. Foster, "SCanDroid : Automated Security Certification of Android Applications," University of Maryland, Tech. Rep. CS-TR-4991, November 2009.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *SIGPLAN Not.*, vol. 49, no. 6, pp. 259–269, Jun. 2014.

[4] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 1329–1341.

[5] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 280–291.

[6] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, May 1976.

[7] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, Jul. 1977.

[8] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 2-3, pp. 167–187, Jan. 1996.

[9] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Sep. 2003.

[10] J. A. Goguen and J. Meseguer, "Security Policies and Security Models," in *SOSP*, 1982, pp. 11–20.

[11] A. Banerjee and D. A. Naumann, "Stack-based access control and secure information flow," *Journal of Functional Programming*, vol. 15, no. 2, pp. 131–177, Mar. 2005.

[12] J. Landauer and T. Redmond, "A lattice of information," in *6th IEEE Computer Security Foundations Workshop - CSFW'93, Franconia, New Hampshire, USA, June 15-17, 1993, Proceedings*. IEEE Computer Society, 1993, pp. 65–70.

[13] H. Chen, A. Tiu, Z. Xu, and Y. Liu, "A permission-dependent type system for secure information flow analysis," *CoRR*, vol. abs/1709.09623, 2017. [Online]. Available: http://arxiv.org/abs/1709.09623

[14] Android, "Requesting permissions at run time." [Online]. Available: https://developer.android.com/training/permissions/requesting.html

[15] A. Developers, "Binder," https://developer.android.com/reference/android/os/Binder.html, 2017, online, accessed on 07-July-2017.

[16] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '11, New York, NY, USA, 2011, pp. 239–252.

[17] A. Developers, "Permissionchecker | android developers," https://developer.android.com/reference/android/support/v4/content/PermissionChecker.html, 2017, online, accessed on 07-July-2017.

[18] T. M. , R. Sison, E. Pierzchalski, and C. Rizkallah, "Compositional verification and refinement of concurrent value-dependent noninterference," in *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, June 2016, pp. 417–431.

[19] N. Polikarpova, J. Yang, S. Itzhaky, and A. Solar-Lezama, "Type-driven repair for information flow security," *CoRR*, vol. abs/1607.03445, 2016. [Online]. Available: http://arxiv.org/abs/1607.03445

[20] T. Murray, R. Sison, E. Pierzchalski, and C. Rizkallah, "A dependent security type system for concurrent imperative programs," *Archive of Formal Proofs*, Jun. 2016, http://isa-afp.org/entries/Dependent_SIFUM_Type_Systems.shtml, Formal proof development.

[21] T. Murray, R. Sison, E. Pierzchalski, and C. Rizkallah, "Compositional security-preserving refinement for concurrent imperative programs," *Archive of Formal Proofs*, Jun. 2016, http://isa-afp.org/entries/Dependent_SIFUM_Refinement.shtml, Formal proof development.

[22] T. Murray, "Short Paper: On High-Assurance Information-Flow-Secure Programming Languages," in *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security*, 2015.

[23] X. Li, F. Nielson, and H. Riis Nielson, "Future-dependent Flow Policies with Prophetic Variables," in *the 2016 ACM Workshop*. New York, NY, USA: ACM Press, 2016, pp. 29–42.

[24] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A Hardware Design Language for Timing-Sensitive Information-Flow Security," in *the Twentieth International Conference*. New York, New York, USA: ACM Press, 2015, pp. 503–516.

[25] X. Li, F. Nielson, H. R. Nielson, and X. Feng, "Disjunctive Information Flow for Communicating Processes." *TGC*, 2015.

[26] L. Lourenço and L. Caires, "Dependent information flow types," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15, New York, NY, USA, 2015, pp. 317–328.

[27] L. Lourenço and L. Caires, "Information flow analysis for valued-indexed data security compartments," in *Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 8358. Springer, 2014, pp. 180–198.

[28] N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang, "Secure distributed programming with value-dependent types," *Journal of Functional Programming*, vol. 23, no. 4, pp. 402–451, 2013.

[29] J. Yang, K. Yessenov, and A. Solar-Lezama, "A language for automatically enforcing privacy policies," in *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012*. ACM, 2012, pp. 85–96. [Online]. Available: http://doi.acm.org/10.1145/2103656.2103669

[30] H. Mantel, D. Sands, and H. Sudbrock, "Assumptions and guarantees for compositional noninterference," in *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*. IEEE Computer Society, 2011, pp. 218–232. [Online]. Available: https://doi.org/10.1109/CSF.2011.22

[31] A. Nanevski, A. Banerjee, and D. Garg, "Verification of information flow and access control policies with dependent types," in *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*. IEEE Computer Society, 2011, pp. 165–179.

[32] N. Swamy, J. Chen, and R. Chugh, "Enforcing stateful authorization and information flow policies in fine," in *Programming Languages and Systems, 19th European Symposium on Programming (ESOP)*, ser. Lecture Notes in Computer Science, vol. 6012. Springer, 2010, pp. 529–549.

[33] L. Zheng and A. C. Myers, "Dynamic security labels and static information flow control." *International Journal of Information Security*, vol. 6, no. 2-3, pp. 67–84, 2007.

[34] S. Tse and S. Zdancewic, "Run-time principals in information-flow type systems," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 1, 2007.

[35] P. Li and D. Zhang, "Towards a flow- and path-sensitive information flow analysis," in *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, Aug 2017, pp. 53–67.

[36] S. Hunt and D. Sands, "On flow-sensitive security types," in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '06. New York, NY, USA: ACM, 2006, pp. 79–90.

[37] 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 *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 1092–1104.

[38] A. Nadkarni, B. Andow, W. Enck, and S. Jha, "Practical DIFC Enforcement on Android," *USENIX Security Symposium*, 2016.

[39] S. Lortz, H. Mantel, A. Starostin, T. Bahr, D. Schneider, and A. Weber, "Cassandra: Towards a certifying app store for android," in *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '14, New York, NY, USA, 2014, pp. 93–104.

[40] H. Gunadi, "Formal certification of non-interferent android bytecode (dex bytecode)," in *Proceedings of the 2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS)*, ser. ICECCS '15, Washington, DC, USA, 2015, pp. 202–205.

[41] A. Chaudhuri, "Language-based security on android," in *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, ser. PLAS '09, New York, NY, USA, 2009, pp. 1–7.

[42] G. C. Necula and P. Lee, "Proof-carrying code," School of Computer Science, Carnegie Mellon University, Tech. Rep., 1996, cMU-CS-96-165.

[43] Q. Sun, A. Banerjee, and D. A. Naumann, "Modular and constraint-based information flow inference for an object-oriented language," in *Static Analysis*, R. Giacobazzi, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 84–99.

[44] G. Barthe, T. Rezk, and D. A. Naumann, "Deriving an Information Flow Checker and Certifying Compiler for Java." *IEEE Symposium on Security and Privacy*, pp. 230–242, 2006.

[45] G. Barthe and T. Rezk, "Non-interference for a JVM-like language." *TLDI*, pp. 103–112, 2005.

[46] G. Barthe, D. Pichardie, and T. Rezk, "A certified lightweight non-interference java bytecode verifier," in *Proceedings of the 16th European Symposium on Programming*, ser. ESOP'07, Berlin, Heidelberg, 2007, pp. 125–140.