A Permission-Dependent Type System for Secure Information Flow Analysis

Zhiwu Xu [a], Hongxu Chen [b], Alwen Tiu [c], Yang Liu [b], Kunal Sareen [c]
[a] *College of Computer Science and Software Engineering, Shenzhen University, China*
E-mail: xuzhiwu@szu.edu.cn
[b] *Nanyang Technological University, Singapore*
E-mails: hongxu.chen@ntu.edu.sg, yangliu@ntu.edu.sg
[c] *The Australian National University, Australia*
E-mails: alwen.tiu@anu.edu.au, kunal.sareen@anu.edu.au

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. A type inference algorithm is also presented for the underlying security type system, by reducing the inference problem to a constraint solving problem in the lattice of security types. In addition, a new way to represent our security types as reduced ordered binary decision diagrams is proposed.

Keywords: secure information flow, secure type system, non-interference, permission-dependent, Android

1. 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 analyses of their security have been of significant interest. 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 guarantees of information flow security in the constructed applications.

We follow the language-based security approach whereby information flow security is enforced through type systems [6–9]. In particular, we propose a design of a type system that guarantees a 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 a 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 $\leqslant$, e.g., a lattice with two elements "high" (H) and "low" (L)

where $L \leqslant H$. 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 $l(e) \leqslant l(x)$ 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.

1.1. 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 and 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 the permission READ_CONTACT. 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 a 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 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:

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

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.

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

Listing 1 Getting contact info with a permission check.

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

Listing 2 An example with a non-monotonic policy.

In BN system, each class is assigned a set of permissions, but before these permissions can be exercised, they must be explicitly enabled with an **enable** command. Initially all permissions are disabled by default. The typing judgment in BN system keeps track of the set of permissions (denoted by Q in the rules below) that are disabled at a particular point in the program. The language of BN system also 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:

$$\text{(R1)}\ \frac{Q \cap P = \emptyset \qquad Q \vdash c_1 : \tau \qquad Q \vdash c_2 : \tau}{Q \vdash \textbf{test}(P)\ c_1\ \textbf{else}\ c_2 : \tau} \qquad \text{(R2)}\ \frac{Q \cap P \neq \emptyset \qquad Q \vdash c_2 : \tau}{Q \vdash \textbf{test}(P)\ c_1\ \textbf{else}\ c_2 : \tau}$$

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. This could happen in BN system due to the way permissions are propagated and/or inherited across different classes, so it could happen that the context in which the program being typed occurs is not authorized to access some permissions in P. As noted in [11], in this case, one cannot determine statically, from the information available in the typing judgment, whether the test '**test**(P)' succeeds, so the typing rule R1 conservatively considers typing both branches.

In adapting BN system to Android, we can make some simplifications: permissions of an app are always enabled, and they are static.[1] So in this case, we can assume that $Q = \emptyset$, and only the rule R1 is relevant here. However, even with these simplifications, it is still not possible to determine statically whether 'test(P)' would succeed or not, in the context where the program being typed is part of a service that may be called by arbitrary apps, so their permissions cannot be determined statically. So it seems that a straightforward adaptation of BN system to Android would still keep the form of R1. However, R1 is still too strong in some scenarios, where the desired security policy requires the *absence* of some permissions for the release of sensitive information. We call such a policy a *non-monotonic policy*, in the sense that, viewing a policy as a function from permission sets to security labels, the possession of more permissions does not equal the ability to acquire more sensitive information.

[1] Some permissions in Android 6 and above require user approval at runtime, but for the purpose of typing the 'test' command, we make the assumption that these permissions are enabled as well.

For an example of an application for which a non-monotonic policy is desirable, consider for example an application that provides the location information related to a certain *advertising ID* (in Listing 2), where the latter provides a unique ID for the purpose of anonymizing mobile users to be used for personalized 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 the device ID. That is, if the caller of *getInfo* possess the permission to access the device ID, then the caller should not be able to obtain any information that contains the location information; so this policy is non-monotonic: a caller with no permissions is allowed to access the sensitive information (location), whereas another caller with *more* permission is not allowed to access that information.

To simplify the discussion, let us assume that the permissions to access the IMEI number and the location information are denoted by p and q, respectively; and *aid* denotes a unique advertising ID generated and stored by the app for the purpose of anonymizing user tracking and *loc* denotes the location information. The function *getInfo* first tests whether the caller has access to the IMEI number. If it does, and if it has access to the location information, then only the location information is returned. If the caller has no access to the IMEI number, but can access the location information, then the combination of the advertising id and the location information *aid*++*loc* is returned. In all other cases, the empty string is returned. Let us consider a lattice with four elements ordered as: $L \leqslant l_1, l_2 \leqslant H$, where l_1 and l_2 are incomparable. We specify that empty string is of type L, *loc* is of type l_1, *aid* is of type l_2, and the aggregate *aid*++*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 due to the inability to determine statically whether 'test(P)' succeeds, leading to a conservative choice in the rule R1 of assigning the same types to both branches of the test. As a result, BN system cannot precisely capture the non-monotonic security policy in our example above: when the caller has permissions p and q, the desired type of *getInfo* should be l_1, which is not necessary larger than the types when the caller has neither p nor q (*e.g.* the type for the enabled set $\{q\}$ is H). The choice of R1 taken in [11] 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 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 the Android framework. Our type system solves the problem of typing non-monotonic policies without resorting to downgrading or declassifying information. The technique we use is to keep information related to both branches of **test** via a *merging* operator on security types. Additionally, there

is a significant difference between the permission model of Android and that of BN system, where permissions are propagated across method invocations among apps. In Android, permissions are relevant only during inter-process or inter-component calls and are not inherited along the call chains across apps. As we shall see in Section 2.5, 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 system due to the difference in permission model and the new merging operator on types in our type system.

The contributions of our work are four-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 a notion of non-interference (Section 2). 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 of 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 3).
(4) We give a new way to represent our security types as reduced ordered binary decision diagrams, which generalizes boolean functions to functions mapping boolean formula to a multi-value set. We believe the representation will be efficient in practice.

This paper is an extension of [13]. It first extends the security type system with global variables and proves that it is still sound with respect to non-interference [2] (Section 2). It further revises type inference for global variables and contains the main lemmas for its decidability (Section 3). Finally, it also presents a representation for our types (Section 4).

The rest of the paper is organized as follows. Section 2 presents our security type system and its properties. Section 3 and Section 4 give the type inference for our security type system and the representation for our types, respectively. Section 5 presents related work. And Section 6 concludes the paper and discusses some future work.

2. A Secure Information Flow Type System

In this section, we present the proposed information flow type system. Section 2.1 informally discusses a permission-based access control model, which is an abstraction of the permission mechanism used in Android. Section 2.2 and Section 2.3 give the operational semantics of a simple imperative language that includes permission checking constructs based on the abstract permission model. Section 2.4 and Section 2.5 describe the type system for our language and prove its soundness with respect to a notion of non-interference.

[2] Due to space constraints, only the main proofs are presented here, and the other proofs, including the proofs for type inference and representation, can be found in the appendix.

2.1. 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 the 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 [3], so it does not account for permission granting at runtime. Runtime permission granting [14] poses some problems in typing non-monotonic policies; we shall come back to this point later in Section 6.

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

One significant issue that has to be taken into account is that the 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 2.4).

2.2. 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

[3] 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.

[4] 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.

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 $\mathbf{op}$ 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 "$\mathbf{letvar}\ x = e\ \mathbf{in}\ c$" 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 "$x := \mathbf{call}\ A.f(\overline{e})$" denotes an assignment whose right hand side is a function call to $A.f$. The statement "$\mathbf{test}(p)\ c_1\ \mathbf{else}\ c_2$" 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. The set of free variables occurring in an expression e or a command c is defined as follows:

$$\begin{aligned} fv(n) &= \emptyset & fv(\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) &= fv(e) \cup fv(c_1) \cup fv(c_2) \\ fv(x) &= \{x\} & fv(\mathbf{while}\ e\ \mathbf{do}\ c) &= fv(e) \cup fv(c) \\ fv(e_1\ \mathbf{op}\ e_2) &= fv(e_1) \cup fv(e_2) & fv(\mathbf{letvar}\ x = e\ \mathbf{in}\ c) &= (fv(e) \cup fv(c)) \setminus \{x\} \\ fv(x := e) &= \{x\} \cup fv(e) & fv(x := \mathbf{call}\ A.f(\overline{e})) &= \{x\} \cup \textstyle\bigcup_{e_i \in \overline{e}} fv(e_i) \\ fv(c_1; c_2) &= fv(c_1) \cup fv(c_2) & fv(\mathbf{test}(p)\ c_1\ \mathbf{else}\ c_2) &= fv(c_1) \cup fv(c_2) \end{aligned}$$

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, $\overline{x}$ 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. The set of free variables occurring a function f is defined as follows:

$$fv(A.f(\overline{x})\{\textbf{init } r = 0 \textbf{ in } \{c; \textbf{return } r\}\}) = fv(c) \setminus (\{x_i | x_i \in \overline{x}\} \cup \{r\})$$

Given a system $\mathcal{S}$, we call the free variables occurring in $\mathcal{S}$ as global variables, i.e., the variables that are neither introduced by **letvar** nor from the variable set $\{\overline{x}, r\}$ of any function in $\mathcal{S}$. Global variables can be used to model the shared preference between apps. Formally, the global variables of $\mathcal{S}$ is defined as follows:

$$gv(\mathcal{S}) = \bigcup_{A \in \mathcal{S}} fv(A) = \bigcup_{A \in \mathcal{S}} \bigcup_{A.f \in A} fv(A.f)$$

The others, i.e., the variables that are either introduced by **letvar** or from the variable set x, r of any function in $\mathcal{S}$, are called local variables. To simplify presentation, we assume that (1) variables in a system are named differently so there are no naming clashes between them; and (2) the variable x in $x :=$ **call** $A.f(\overline{e})$ is not a global one, since we can encode $x_g :=$ **call** $A.f(\overline{e})$ as **letvar** $x_l = 0$ **in** $x_l :=$ **call** $A.f(\overline{e}); x_g := x_l$, where x_g is a global variable and x_l is a fresh local variable.

2.3. Operational Semantics

Given a system $\mathcal{S}$, we assume that the permission sets assigned to the apps in $\mathcal{S}$ are given by a table Θ indexed by app names, and function definitions in $\mathcal{S}$ are stored in a table FD indexed by function 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 $dom(\eta)$. To simplify proofs of various properties, we split the evaluation environment into two parts (η^G, η): one (i.e., η^G) for the global variables and the other (i.e., η) for the non-global ones, and require that $dom(\eta^G) = gv(\mathcal{S})$.

The operational semantics for expressions and commands is given in Fig. 1. The evaluation judgment for expressions has the form $(\eta^G, \eta) \vdash e \rightsquigarrow v$, which states that expression e evaluates to value v when variables in e are interpreted in the evaluation environment (η^G, η). We write $(\eta^G, \eta) \vdash \overline{e} \rightsquigarrow \overline{v}$, where $\overline{e} = e_1, \ldots, e_n$ and $\overline{v} = v_1, \ldots, v_n$ for some n, to denote a sequence of judgments $(\eta^G, \eta) \vdash e_1 \rightsquigarrow v_1, \ldots, (\eta^G, \eta) \vdash e_n \rightsquigarrow v_n$.

The evaluation judgment for commands takes the form $(\eta^G, \eta); A; P \vdash c \rightsquigarrow (\eta_1^G, \eta_1)$, where (η^G, η) is an evaluation environment before the execution of the command c, and (η_1^G, η_1) 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.

$$\text{E-VAL}\frac{}{(\eta^G,\eta)\vdash v \rightsquigarrow v} \qquad \text{E-OP}\frac{(\eta^G,\eta)\vdash e_1 \rightsquigarrow v_1 \quad (\eta^G,\eta)\vdash e_2 \rightsquigarrow v_2}{(\eta^G,\eta)\vdash e_1 \ \mathbf{op}\ e_2 \rightsquigarrow v_1\ op\ v_2} \qquad \text{E-VAR-L}\frac{x\in dom(\eta)}{(\eta^G,\eta)\vdash x \rightsquigarrow \eta(x)}$$

$$\text{E-VAR-G}\frac{x\in dom(\eta^G)}{(\eta^G,\eta)\vdash x \rightsquigarrow \eta^G(x)} \qquad \text{E-LETVAR}\frac{(\eta^G,\eta)\vdash e \rightsquigarrow v \quad (\eta^G,\eta[x\mapsto v]);A;P\vdash c \rightsquigarrow (\eta_1^G,\eta_1)}{(\eta^G,\eta);A;P\vdash \mathbf{letvar}\ x=e\ \mathbf{in}\ c \rightsquigarrow (\eta_1^G,\eta_1-x)}$$

$$\text{E-ASS-L}\frac{(\eta^G,\eta)\vdash e \rightsquigarrow v \quad x\in dom(\eta)}{(\eta^G,\eta);A;P\vdash x:=e \rightsquigarrow (\eta^G,\eta[x\mapsto v])} \qquad \text{E-ASS-G}\frac{(\eta^G,\eta)\vdash e \rightsquigarrow v \quad x\in dom(\eta^G)}{(\eta^G,\eta);A;P\vdash x:=e \rightsquigarrow (\eta^G[x\mapsto v],\eta)}$$

$$\text{E-IF-T}\frac{(\eta^G,\eta)\vdash e \rightsquigarrow v \quad v\neq 0 \quad (\eta^G,\eta);A;P\vdash c_1 \rightsquigarrow (\eta_1^G,\eta_1)}{(\eta^G,\eta);A;P\vdash \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \rightsquigarrow (\eta_1^G,\eta_1)} \qquad \text{E-IF-F}\frac{(\eta^G,\eta)\vdash e \rightsquigarrow v \quad v=0 \quad (\eta^G,\eta);A;P\vdash c_2 \rightsquigarrow (\eta_1^G,\eta_1)}{(\eta^G,\eta);A;P\vdash \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \rightsquigarrow (\eta_1^G,\eta_1)}$$

$$\text{E-WHILE-T}\frac{(\eta^G,\eta)\vdash e \rightsquigarrow v \quad v\neq 0 \quad (\eta^G,\eta);A;P\vdash c \rightsquigarrow (\eta_1^G,\eta_1) \quad (\eta_1^G,\eta_1);A;P\vdash \mathbf{while}\ e\ \mathbf{do}\ c \rightsquigarrow (\eta_2^G,\eta_2)}{(\eta^G,\eta);A;P\vdash \mathbf{while}\ e\ \mathbf{do}\ c \rightsquigarrow (\eta_2^G,\eta_2)}$$

$$\text{E-WHILE-F}\frac{(\eta^G,\eta)\vdash e \rightsquigarrow v \qquad v=0}{(\eta^G,\eta);A;P\vdash \mathbf{while}\ e\ \mathbf{do}\ c \rightsquigarrow (\eta^G,\eta)} \qquad \text{E-SEQ}\frac{(\eta^G,\eta);A;P\vdash c_1 \rightsquigarrow (\eta_1^G,\eta_1) \quad (\eta_1^G,\eta_1);A;P\vdash c_2 \rightsquigarrow (\eta_2^G,\eta_2)}{(\eta^G,\eta);A;P\vdash c_1;c_2 \rightsquigarrow (\eta_2^G,\eta_2)}$$

$$\text{E-CP-T}\frac{p\in P \quad (\eta^G,\eta);A;P\vdash c_1 \rightsquigarrow (\eta_1^G,\eta_1)}{(\eta^G,\eta);A;P\vdash \mathbf{test}(p)\ c_1\ \mathbf{else}\ c_2 \rightsquigarrow (\eta_1^G,\eta_1)} \qquad \text{E-CP-F}\frac{p\notin P \quad (\eta^G,\eta);A;P\vdash c_2 \rightsquigarrow (\eta_1^G,\eta_1)}{(\eta^G,\eta);A;P\vdash \mathbf{test}(p)\ c_1\ \mathbf{else}\ c_2 \rightsquigarrow (\eta_1^G,\eta_1)}$$

$$\text{E-CALL}\frac{FD(B.f)=B.f(\overline{y})\{\mathbf{init}\ r=0\ \mathbf{in}\ \{c;\mathbf{return}\ r\}\} \quad (\eta^G,\eta)\vdash \overline{e} \rightsquigarrow \overline{v} \quad (\eta^G,[\overline{y}\mapsto\overline{v},r\mapsto 0]);B;\Theta(A)\vdash c \rightsquigarrow (\eta_1^G,\eta_1)}{(\eta^G,\eta);A;P\vdash x:=\mathbf{call}\ B.f(\overline{e}) \rightsquigarrow (\eta_1^G,\eta[x\mapsto\eta_1(r)])}$$

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

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 system, where permission contexts are inherited by successive function calls.

2.4. 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 $\leqslant_{\mathscr{L}}$. Security types are defined in the following.

Definition 2.1 *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 $\leqslant_{\mathcal{T}}$ on base types as follows: $s \leqslant_{\mathcal{T}} t$ iff $\forall\, P \in \mathscr{P}$, $s(P) \leqslant_{\mathscr{L}} t(P)$.*

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.

Next, we show that the set of base types with the order $\leqslant_{\mathcal{T}}$ forms a lattice.

Definition 2.2 *For $s, t \in \mathcal{T}$, $s \sqcup t$ and $s \sqcap t$ are defined as*

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

Lemma 2.1 *$\leqslant_{\mathcal{T}}$ is a partial order relation on $\mathcal{T}$.*

Lemma 2.2 *$(\mathcal{T}, \leqslant_{\mathcal{T}})$ forms a lattice.*

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

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

Definition 2.3 *Given a security level l, we define $\hat{l}$ as follows: for all $P \in \mathscr{P}$, we have $\hat{l}(P) = l$.*

Definition 2.4 *A* function type *has the form $\bar{t} \xrightarrow{s} t$, where $\bar{t} = (t_1, \ldots, t_m)$, $m \geqslant 0$ and t, s, t_i are base types. The types $\bar{t}$ are the types for the arguments of the function, t is the return type of the function, and s is the type for the body 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 2.5 *Given a permission p, the* promotion *and* demotion *of a base type t with respect to p are:*

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

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

$$(\bar{t} \xrightarrow{s} t) \uparrow_p = \bar{t} \uparrow_p \xrightarrow{s\uparrow_p} t \uparrow_p \qquad (\bar{t} \xrightarrow{s} t) \downarrow_p = \bar{t} \downarrow_p \xrightarrow{s\downarrow_p} t \downarrow_p$$

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

Lemma 2.3 *Given $P \in \mathscr{P}$ and $p \in \mathbf{P}$, (a) $(t \uparrow_p)(P) = t(P)$ if $p \in P$; and (b) $(t \downarrow_p)(P) = t(P)$ if $p \notin P$.*

2.5. Security Type System

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

$$\text{T-VAR-L}\frac{x \in dom(\Gamma)}{(\Gamma^G, \Gamma) \vdash x : \Gamma(x)} \qquad \text{T-VAR-G}\frac{x \in dom(\Gamma^G)}{(\Gamma^G, \Gamma) \vdash x : \Gamma^G(x)} \qquad \text{T-SUB}_e\frac{(\Gamma^G, \Gamma) \vdash e : s \quad s \leqslant t}{(\Gamma^G, \Gamma) \vdash e : t}$$

$$\text{T-OP}\frac{(\Gamma^G, \Gamma) \vdash e_1 : t \quad (\Gamma^G, \Gamma); A \vdash e_2 : t}{(\Gamma^G, \Gamma) \vdash e_1 \ \mathbf{op}\ e_2 : t} \qquad \text{T-SUB}_c\frac{(\Gamma^G, \Gamma); A \vdash c : s \quad t \leqslant s}{(\Gamma^G, \Gamma); A \vdash c : t}$$

$$\text{T-ASS-L}\frac{x \in dom(\Gamma) \quad (\Gamma^G, \Gamma) \vdash e : \Gamma(x)}{(\Gamma^G, \Gamma); A \vdash x := e : \Gamma(x)} \qquad \text{T-ASS-G}\frac{x \in dom(\Gamma^G) \quad (\Gamma^G, \Gamma) \vdash e : \Gamma^G(x)}{(\Gamma^G, \Gamma); A \vdash x := e : \Gamma^G(x)}$$

$$\text{T-LETVAR}\frac{(\Gamma^G, \Gamma) \vdash e : s \quad (\Gamma^G, \Gamma[x : s]); A \vdash c : t}{(\Gamma^G, \Gamma); A \vdash \mathbf{letvar}\ x = e\ \mathbf{in}\ c : t} \qquad \text{T-SEQ}\frac{(\Gamma^G, \Gamma); A \vdash c_1 : t \quad (\Gamma^G, \Gamma); A \vdash c_2 : t}{(\Gamma^G, \Gamma); A \vdash c_1; c_2 : t}$$

$$\text{T-IF}\frac{(\Gamma^G, \Gamma) \vdash e : t \quad (\Gamma^G, \Gamma); A \vdash c_1 : t \quad (\Gamma^G, \Gamma); A \vdash c_2 : t}{(\Gamma^G, \Gamma); A \vdash \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 : t}$$

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

$$\text{T-CALL}\frac{FT(B.f) = \bar{t} \xrightarrow{s} t' \quad (\Gamma^G, \Gamma) \vdash \bar{e} : \pi_{\Theta(A)}(\bar{t}) \quad \pi_{\Theta(A)}(t') \leqslant \Gamma(x)}{(\Gamma^G, \Gamma); A \vdash x := \mathbf{call}\ B.f(\bar{e}) : \Gamma(x) \sqcap \pi_{\Theta(A)}(s)}$$

$$\text{T-FUN}\frac{(\Gamma^G, [\overline{x} : \bar{t}, r : t']); B \vdash c : s}{\Gamma^G \vdash B.f(\overline{x})\{\mathbf{init}\ r = 0\ \mathbf{in}\ \{c; \mathbf{return}\ r\}\} : \bar{t} \xrightarrow{s} t'}$$

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

A function called by different callers may have different the permission contexts, so we define *type projection* to extract types according to the permission sets of callers.

Definition 2.6 *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 $\pi_P((t_1, \ldots, t_n)) = (\pi_P(t_1), \ldots, \pi_P(t_n))$.*

In order to type the permission check $\mathbf{test}(p)\ c_1\ \mathbf{else}\ c_2$ precisely, we need to construct a type t from the types t_1, t_2 respectively for its two branches c_1, c_2, such that t acts like t_1 (t_2 resp.) when p is enabled (disabled resp.).

Definition 2.7 *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. Accordingly, we split the typing environment into two parts Γ^G and Γ, where Γ^G denotes the part for the global variables and Γ denotes the other part for the non-global ones. And we also require that $dom(\Gamma^G) = gv(\mathcal{S})$.

Definition 2.8 *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: $(\Gamma^G, \Gamma) \vdash e : t$. This says that under (Γ^G, Γ), the expression e has a base type at most t.
- Command typing: $(\Gamma^G, \Gamma); A \vdash c : t$. This means that the command c writes to variables with type at least t, when executed by app A, under the typing environment (Γ^G, Γ).
- Function typing: The typing judgment takes the form:

$$\Gamma^G \vdash B.f(\overline{x})\{\textbf{init } r = 0 \textbf{ in } \{c; \textbf{return } r\}\} : \overline{t} \xrightarrow{s} t'$$

  where $\overline{x} = (x_1, \ldots, x_n)$ and $\overline{t} = (t_1, \ldots, t_n)$ for some $n \geqslant 0$. 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 $\pi_P(\overline{t}) \xrightarrow{\pi_P(s)} \pi_P(t')$. That is, if the function is called from another app with permission P, then it expects input of type up to $\pi_P(\overline{t})$, a return value of type at most $\pi_P(t')$, and the type for the function body at least $\pi_P(s)$.

When proving the soundness of our type system, we need to make sure that the system of apps are well-typed in the following sense:

Definition 2.9 *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 *under a global typing environment Γ^G iff for every function $A.f$, $\Gamma^G \vdash FD(A.f) : FT(A.f)$ is derivable.*

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 the types for constants depend on the constants themselves. Taking *loc* and *aid* in Section 1 for example, their types are respectively $\hat{l}_1$ and $\hat{l}_2$. Also 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 1, 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 $\textbf{test}(p)\ c_1\ \textbf{else}\ c_2$ is equivalent to c_1. In this case, one would expect that the behavior of $\textbf{test}(p)\ c_1\ \textbf{else}\ c_2$ 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 under the global typing environment Γ^G 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 $\Theta(A)$): $\pi_{\Theta(A)}(\bar{t}) \xrightarrow{\pi_{\Theta(A)}(s)} \pi_{\Theta(A)}(t')$. 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., $\pi_{\Theta(A)}(t') \leqslant \Gamma(x)$. Finally, the projection of the body type, i.e., the effect of the function call by A, should be propagated to the result type: one can think this is the sequential composition of the function body and the assignment. Due to global variables, we need to accumulate the type (i.e., effects) of the commands in the function body that involves global variables. But to make our presentation simple, we do not separate them from local ones. In addition, note that the execution of a function call depends on the permissions of the caller, but this does not mean the permissions of the caller is passed to the callee.

Attack on global variables. We require that the type of a global variable should be invariant for all permission sets, that is, the global typing environment Γ^G is form of $[x_1 : \hat{l_1}, \ldots, x_n : \hat{l_n}]$, where x_i is a global variable, l_i is a security level. The reason is that different from non-global variables, there is only one copy for global variables. In other words, all the apps share this unique evaluation environment η^G of global variables (*e.g.*, such as the shared preference files), which could easily lead to information leaks via global variables. For example, consider a global variable x with a non-constant type, e.g., $t_x = \{\emptyset \mapsto L, \{p\} \mapsto H\}$, and two apps A, B satisfying that A has permission p to access some security information but B does not. Suppose that A would like to get the security information and store in x. As A has p, when running on A, the type for x is $\hat{H}$. Therefore, the behavior of A is typeable. Consider another situation that B would like to get information from x and store in any variable typed of $\hat{L}$. Similarly, it is easy to check that the behavior of B is typeable as well. However, if the behavior of A happens before the behavior of B, then there is an information leak, but the whole system is typeable according to the discussion above.

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

$$\text{T-CALL'}\ \frac{FT(B.f) = \bar{t} \xrightarrow{s} t' \qquad (\Gamma^G, \Gamma) \vdash \bar{e} : \bar{t} \qquad t' \leqslant \Gamma(x)}{(\Gamma^G, \Gamma); A \vdash x := \textbf{call}\ B.f(\bar{e}) : \Gamma(x) \sqcap s}$$

Notice that the type of the argument $\bar{e}$ 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 $() \xrightarrow{t} t$. 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: $\Theta(A) = \Theta(B) = \emptyset, \Theta(C) = \Theta(M) = \{p\}$. As no global variables occur here, we omit the global environment Γ^G.

```
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 for parameter laundering issue.

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

$$FT := \{A.f \mapsto t \xrightarrow{\hat{L}} \hat{L};\quad B.g \mapsto t \xrightarrow{\hat{L}} \hat{L};\quad C.getsecret \mapsto () \xrightarrow{t} t;\quad M.main \mapsto () \xrightarrow{\hat{L}} \hat{L}\}$$

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 $t \xrightarrow{\hat{L}} \hat{L}$. 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 2.1. 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.

$$\text{T-FUN}\ \frac{\text{T-CALL'}\ \dfrac{FT(B.g) = t \xrightarrow{\hat{L}} \hat{L} \qquad x : t, r : \hat{L} \vdash x : t \qquad \hat{L} \leqslant t}{x : t, r : \hat{L}; A \vdash r := \textbf{call}\ B.g(x) : \hat{L} \sqcap \hat{L} = \hat{L}}}{\vdash A.f(x)\{\textbf{init}\ r = 0\ \textbf{in}\ \{r := \textbf{call}\ B.g(x); \textbf{return}\ r\}\} : t \xrightarrow{\hat{L}} \hat{L}}$$

$$\text{T-FUN}\ \frac{\text{T-CP}\ \dfrac{x : t \uparrow_p, r : \hat{L} \uparrow_p; B \vdash r := 0 : \hat{L} \qquad x : t \downarrow_p, r : \hat{L} \downarrow_p; B \vdash r := x : \hat{L}}{x : t, r : \hat{L}; B \vdash \textbf{test}(p)\ r := 0\ \textbf{else}\ r := x : \hat{L} \triangleright_p \hat{L}}}{\vdash B.g(x)\{\textbf{init}\ r = 0\ \textbf{in}\ \{\textbf{test}(p)\ r := 0\ \textbf{else}\ r := x; \textbf{return}\ r\}\} : t \xrightarrow{\hat{L}} \hat{L}}$$

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

$$\text{T-FUN}\ \frac{\text{T-CP}\ \dfrac{r : t \uparrow_p; C \vdash r := \text{SECRET} : \hat{H} \qquad r : t \downarrow_p; C \vdash r := 0 : \hat{L}}{r : t; C \vdash \textbf{test}(p)\ r := \text{SECRET}\ \textbf{else}\ r := 0 : \hat{H} \triangleright_p \hat{L}}}{\vdash C.getsecret(\,)\{\textbf{init}\ r = 0\ \textbf{in}\ \{\textbf{test}(p)\ r := \text{SECRET}\ \textbf{else}\ r := 0; \textbf{return}\ r\}\} : () \xrightarrow{t} t}$$

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

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

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

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

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

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

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

With the correct typing rule for function calls, the function $A.f$ cannot be assigned type $t \rightarrow \hat{L}$, since that would require 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 $t \not\leqslant \hat{L}$. 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 $t \xrightarrow{\hat{L}} \hat{L}$ then $A.f$ can only be assigned type $\hat{L} \xrightarrow{\hat{L}} \hat{L}$. 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.

Non-monotonic example. Let us consider the example about non-monotonic policy, that is, the function *getInfo* in Listing 2. Thanks to the rule T-CP, our system is able to capture the non-monotonic policy, so that the function *getInfo* can be precisely typed in our system, whose typing derivation is given in Fig. 5, where Γ^G is omitted for simplicity.

$$\text{T-FUN}\ \frac{\text{T-CP}\ \dfrac{\text{T-CP}\ \dfrac{r : \hat{l_1}; A \vdash r := loc : \hat{l_1} \qquad r : \hat{L}; A \vdash r := \text{“”} : \hat{L}}{r : t_1; A \vdash \textbf{test}(q)\ r := loc\ \textbf{else}\ r := \text{“”} : t_1} \qquad \text{T-CP}\ \dfrac{r : \hat{H}; A \vdash r := aid + loc : \hat{H} \qquad r : \hat{L}; A \vdash r := \text{“”} : \hat{L}}{r : t_2; A \vdash \textbf{test}(q)\ r := aid + loc\ \textbf{else}\ r := \text{“”} : t_2}}{r : t; A \vdash \textbf{test}(p)\ (\textbf{test}(q)\ r := loc\ \textbf{else}\ r := \text{“”})\ \textbf{else}\ (\textbf{test}(q)\ r := aid + loc\ \textbf{else}\ r := \text{“”}) : t}}{\vdash A.getInfo(\,) : () \xrightarrow{t} t}$$

where $t = \{\emptyset \mapsto L, \{p\} \mapsto L, \{q\} \mapsto H, \{p,q\} \mapsto l_1\}$, $t_1 = \{\emptyset \mapsto L, \{q\} \mapsto l_1\} = t \uparrow_p$, and $t_2 = \{\emptyset \mapsto L, \{q\} \mapsto H\} = t \downarrow_p$.

Fig. 5. Typing derivation for functions A.getInfo in Listing 2

2.6. 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 caller app. 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 $t(P) \sqcup t'(P)$. 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 2.10 *Given two evaluation environments η, η', a typing environment Γ, a security level $l_O \in \mathscr{L}$ of the observer, the* indistinguishability relation $=^{l_O}_{\Gamma}$ *is defined as:*

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

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 2.10, 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

$\Gamma(x) \leqslant \hat{l_O}$. The latter implies that $\Gamma(x)(P) \leqslant l_O$ (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.

Lemma 2.4 *$=^{lo}_{\Gamma}$ is an equivalence relation on EEnv.*

Recall that we assume no (mutual) recursions, so every function call chain in a well-typed system is finite; this is formalized via the rank function below. We will use this as a measure in our soundness proof (Lemma 2.9).

$$\begin{aligned}
\mathbf{r}(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2) &= max(\mathbf{r}(c_1), \mathbf{r}(c_2)) & \mathbf{r}(x := e) &= 0\\
\mathbf{r}(c_1; c_2) &= max(\mathbf{r}(c_1), \mathbf{r}(c_2)) & \mathbf{r}(\textbf{while } e \textbf{ do } c) &= \mathbf{r}(c)\\
\mathbf{r}(\textbf{test}(p)\ c_1 \textbf{ else } c_2) &= max(\mathbf{r}(c_1), \mathbf{r}(c_2)) & \mathbf{r}(\textbf{letvar } x = e \textbf{ in } c) &= \mathbf{r}(c)\\
\mathbf{r}(x := \textbf{call } A.f(\overline{e})) &= \mathbf{r}(FD(A.f)) + 1 & \mathbf{r}(A.f(\overline{x})\{\textbf{init } r = 0 \textbf{ in } \{c; \textbf{return } r\}\}) &= \mathbf{r}(c)
\end{aligned}$$

The next two lemmas relate projection, promotion/demotion and the indistinguishability relation.

Lemma 2.5 *If $p \in P$, then $\eta =^{lo}_{\pi_P(\Gamma)} \eta'$ iff $\eta =^{lo}_{\pi_P(\Gamma\uparrow_p)} \eta'$.*

Lemma 2.6 *If $p \notin P$, then $\eta =^{lo}_{\pi_P(\Gamma)} \eta' \iff \eta =^{lo}_{\pi_P(\Gamma\downarrow_p)} \eta'$.*

The key to the soundness proof is the following two lemmas, which are the analogs to the simple security property and the confinement property in [8].

Lemma 2.7 *Suppose $(\Gamma^G, \Gamma) \vdash e : t$. For $P \in \mathscr{P}$, if $t(P) \leqslant l_O$ and $\eta_1 =^{lo}_{\pi_P(\Gamma)} \eta_2$, $\eta_1^G =^{lo}_{\Gamma^G} \eta_2^G$, $(\eta_1^G, \eta_1) \vdash e \rightsquigarrow v_1$ and $(\eta_2^G, \eta_2) \vdash e \rightsquigarrow v_2$, then $v_1 = v_2$.*

Proof. The proof proceeds by induction on the derivation of $(\Gamma^G, \Gamma) \vdash e : t$.

T-VAR-L We have $(\Gamma^G, \Gamma) \vdash x : \Gamma(x) = t$ and x is non-global. Since $t(P) \leqslant l_O$ and $\eta_1 =^{lo}_{\pi_P(\Gamma)} \eta_2$, it is deducible that $v_1 = \eta_1(x) = \eta_2(x) = v_2$.

T-VAR-G We have $(\Gamma^G, \Gamma) \vdash x : \Gamma^G(x) = t$ and x is global. Since $t(P) \leqslant l_O$ and $\Gamma^G(x)$ is invariant for all permission sets, we have $\Gamma^G(x) \leqslant \hat{l_O}$. Then from $\eta_1^G =^{lo}_{\Gamma^G} \eta_2^G$, we get $v_1 = \eta_1^G(x) = \eta_2^G(x) = v_2$.

T-OP We have the typing derivation

$$\frac{(\Gamma^G, \Gamma) \vdash e_1 : t \qquad (\Gamma^G, \Gamma) \vdash e_2 : t}{(\Gamma^G, \Gamma) \vdash e_1 \textbf{ op } e_2 : t}$$

and the evaluations

$$\frac{(\eta_1^G, \eta_1) \vdash e_i \rightsquigarrow v_{1i}}{(\eta_1^G, \eta_1) \vdash e_1 \textbf{ op } e_2 \rightsquigarrow v_{11}\ op\ v_{12}} \qquad \frac{(\eta_2^G, \eta_2) \vdash e_i \rightsquigarrow v_{2i}}{(\eta_2^G, \eta_2) \vdash e_1 \textbf{ op } e_2 \rightsquigarrow v_{21}\ op\ v_{22}}$$

By induction on e_i, we can get $v_{1i} = v_{2i}$. Therefore $v_1 = v_2$.

T-SUB$_e$ we have

$$\frac{(\Gamma^G, \Gamma) \vdash e : s \qquad s \leqslant t}{(\Gamma^G, \Gamma) \vdash e : t}$$

since $s(P) \leqslant t(P)$ and $t(P) \leqslant l_O$, then $s(P) \leqslant l_O$ as well, thus the result follows by induction on $(\Gamma^G, \Gamma) \vdash e : s$.

□

Lemma 2.8 *Suppose* $(\Gamma^G,\Gamma);A \vdash c : t$. *Then for any* $P \in \mathscr{P}$, *if* $t(P) \not\leq l_O$ *and* $(\eta^G,\eta);A;P \vdash c \rightsquigarrow (\eta_1^G,\eta_1)$, *then* $\eta =^{l_O}_{\pi_P(\Gamma)} \eta_1$ *and* $\eta^G =^{l_O}_{\Gamma^G} \eta_1^G$.

Proof. By induction on the derivation of $(\Gamma^G,\Gamma);A \vdash c : t$, with subinduction on the derivation of $(\eta^G,\eta);A;P \vdash c \rightsquigarrow (\eta_1^G,\eta_1)$.

T-ASS-L In this case x is non-global, $t = \Gamma(x)$ and the typing derivation has the form:

$$\frac{(\Gamma^G,\Gamma) \vdash e : \Gamma(x)}{(\Gamma^G,\Gamma);A \vdash x := e : \Gamma(x)}$$

and the evaluation under (η^G,η) takes the form:

$$\frac{(\eta^G,\eta) \vdash e \rightsquigarrow v}{(\eta^G,\eta);A;P \vdash x := e \rightsquigarrow (\eta^G,\eta[x \mapsto v])}$$

That is, $\eta_1 = \eta[x \mapsto v]$ and $\eta_1^G = \eta^G$. So η and η_1 differ possibly only in the mapping of x. Since $\Gamma(x)(P) = t(P) \not\leq l_O$, that is $\pi_P(\Gamma)(x) \not\leq \hat{l_O}$, the difference in the valuation of x is not observable at level l_O. It then follows from Definition 2.10 that $\eta =^{l_O}_{\pi_P(\Gamma)} \eta_1$.

T-ASS-G In this case x is global, $t = \Gamma^G(x)$ and the typing derivation has the form:

$$\frac{(\Gamma^G,\Gamma) \vdash e : \Gamma^G(x)}{(\Gamma^G,\Gamma);A \vdash x := e : \Gamma^G(x)}$$

and the evaluation under (η^G,η) takes the form:

$$\frac{(\eta^G,\eta) \vdash e \rightsquigarrow v}{(\eta^G,\eta);A;P \vdash x := e \rightsquigarrow (\eta^G[x \mapsto v],\eta)}$$

That is, $\eta_1 = \eta$ and $\eta_1^G = \eta^G[x \mapsto v]$. So η^G and η_1^G differ possibly only in the mapping of x. Since $\Gamma^G(x)(P) = t(P) \not\leq l_O$ and $\Gamma^G(x)$ is invariant for all permission sets, we have $\Gamma^G(x) \not\leq \hat{l_O}$. Then the difference in the valuation of x is not observable at level l_O. It then follows from Definition 2.10 that $\eta^G =^{l_O}_{\Gamma^G} \eta_1^G$.

T-CALL In this case c has the form $x :=$ **call** $B.f(\overline{e})$ and the typing derivation takes the form:

$$\frac{(\Gamma^G,\Gamma) \vdash \overline{e} : \pi_{\Theta(A)}(\overline{s}) \qquad FT(B.f) = \overline{s} \xrightarrow{s_b} s' \qquad \pi_{\Theta(A)}(s') \leqslant \Gamma(x)}{(\Gamma^G,\Gamma);A \vdash x := \textbf{call } B.f(\overline{e}) : \Gamma(x) \sqcap \pi_{\Theta(A)}(s_b)}$$

and we have that $t = \Gamma(x) \sqcap \pi_{\Theta(A)}(s_b)$. The evaluation under (η^G,η) is derived as follows:

$$\frac{\begin{array}{c} FD(B.f) = B.f(\overline{y})\{\textbf{init } r = 0 \textbf{ in } \{c_1; \textbf{return } r\}\} \qquad (\eta^G,\eta) \vdash \overline{e} \rightsquigarrow \overline{v} \\ (\mathbf{T_1})\ (\eta^G,[\overline{y} \mapsto \overline{v}, r \mapsto 0]);B;\Theta(A) \vdash c_1 \rightsquigarrow (\eta_2^G,\eta_2) \end{array}}{(\eta^G,\eta);A;P \vdash x := \textbf{call } B.f(\overline{e}) \rightsquigarrow (\eta_2^G,\eta[x \mapsto \eta_2(r)])}$$

where $\eta_1 = \eta[x \mapsto \eta_2(r)]$ and $\eta_1^G = \eta_2^G$. Since $t(P) \not\leqslant l_O$, we have $\Gamma(x)(P) \not\leqslant l_O$ and therefore $\Gamma(x) \not\leqslant l_O$ and $\eta =^{l_O}_{\pi_P(\Gamma)} \eta[x \mapsto \eta_2(r)]$, that is, $\eta =^{l_O}_{\pi_P(\Gamma)} \eta_1$.

The remaining is to prove $\eta^G =^{l_O}_{\Gamma^G} \eta_1^G$. As we consider only well-typed systems, the function $FD(B.f)$ is also typable under Γ^G:

$$\frac{(\mathbf{T_2})\ (\Gamma^G, [\overline{y} : \overline{s}, r : s']); B \vdash c_1 : s_b}{\Gamma^G \vdash B.f(\overline{y})\{\textbf{init } r = 0 \textbf{ in } \{c_1; \textbf{return } r\}\} : \overline{s} \xrightarrow{s_b} s'}$$

From $t(P) \not\leqslant l_O$, we also deduce $\pi_{\Theta(A)}(s_b) \not\leqslant l_O$. By induction on $(\mathbf{T_2})$ and $(\mathbf{T_1})$, we get $[\overline{y} \mapsto \overline{v}, r \mapsto 0] =_{\pi_{\Theta(A)}([\overline{y}:\overline{s},r:s'])} \eta_2$ and $\eta^G =^{l_O}_{\Gamma^G} \eta_1^G$. The latter implies $\eta^G =^{l_O}_{\Gamma^G} \eta_2^G$.

T-IF This follows straightforwardly from the induction hypothesis.

T-WHILE We look at the case where the condition of the while loop evaluates to true, otherwise it is trivial. In this case the typing derivation is

$$\frac{(\Gamma^G, \Gamma) \vdash e : t \qquad (\Gamma^G, \Gamma); A \vdash c : t}{(\Gamma^G, \Gamma); A \vdash \textbf{while } e \textbf{ do } c : t}$$

and the evaluation derivation is

$$\frac{(\eta^G, \eta) \vdash e \rightsquigarrow v \quad v \neq 0 \qquad (\eta^G, \eta); A; P \vdash c \rightsquigarrow (\eta_2^G, \eta_2) \qquad (\eta_2^G, \eta_2); A; P \vdash \textbf{while } e \textbf{ do } c \rightsquigarrow (\eta_1^G, \eta_1)}{(\eta^G, \eta); A; P \vdash \textbf{while } e \textbf{ do } c \rightsquigarrow (\eta_1^G, \eta_1)}$$

Applying the induction hypothesis (on typing derivation) and the inner induction hypothesis (on the evaluation derivation) we get $\eta =^{l_O}_{\pi_P(\Gamma)} \eta_2$ and $\eta^G =^{l_O}_{\Gamma^G} \eta_2^G$, and then $\eta_2 =^{l_O}_{\pi_P(\Gamma)} \eta_1$ and $\eta_2^G =^{l_O}_{\Gamma^G} \eta_1^G$; by transitivity of $=^{l_O}_{\pi_P(\Gamma)}$ the result follows.

T-SEQ This follows from the induction hypothesis and transitivity of the indistinguishability relation.

T-LETVAR This case follows from the induction hypothesis and the fact that we can choose fresh variables for local variables, and that the local variables are not visible outside the scope of letvar.

T-CP We have:

$$\frac{(\Gamma^G, \Gamma \uparrow_p); A \vdash c_1 : t_1 \qquad (\Gamma^G, \Gamma \downarrow_p); A \vdash c_2 : t_2 \qquad t = t_1 \triangleright_p t_2}{(\Gamma^G, \Gamma); A \vdash \textbf{test}(p)\ c_1 \textbf{ else } c_2 : t}$$

There are two possible derivations for the evaluation. In one case, we have

$$\frac{p \in P \quad (\eta^G, \eta); A; P \vdash c_1 \rightsquigarrow (\eta_1^G, \eta_1)}{(\eta^G, \eta); A; P \vdash \textbf{test}(p)\ c_1 \textbf{ else } c_2 \rightsquigarrow (\eta_1^G, \eta_1)}$$

Since $t(P) \not\leqslant l_O$ and $p \in P$, by Definiton 2.7, we have $t_1(P) \not\leqslant l_O$. By induction hypothesis, we have $\eta =^{l_O}_{\pi_P(\Gamma\uparrow_p)} \eta_1$ and $\eta^G =^{l_O}_{\Gamma^G} \eta_1^G$. by Lemma 2.5, we have $\eta =^{l_O}_{\pi_P(\Gamma)} \eta_1$.

The case where $p \notin P$ can be handled similarly, making use of Lemma 2.6.

T-SUB$_c$ Straightforward by induction.

□

Before we define the notion of non-interference for a system, we need to define what it means for a command to be non-interferent.

Definition 2.11 *A command c executed in app A is said to be* non-interferent *iff for all* η_1, η_2, η_1^G, η_2^G, Γ, Γ^G, P, l_O, *if* $\eta_1 =^{l_O}_{\pi_P(\Gamma)} \eta_2$, $\eta_1^G =^{l_O}_{\Gamma^G} \eta_2^G$, $(\eta_1^G, \eta_1); A; P \vdash c \rightsquigarrow (\eta_3^G, \eta_3)$ *and* $(\eta_2^G, \eta_2); A; P \vdash c \rightsquigarrow (\eta_4^G, \eta_4)$ *then* $\eta_3 =^{l_O}_{\pi_P(\Gamma)} \eta_4$ *and* $\eta_3^G =^{l_O}_{\Gamma^G} \eta_4^G$.

The main technical lemma is that well-typed commands are non-interferent.

Lemma 2.9 *Suppose* $(\Gamma^G, \Gamma); A \vdash c : t$, *for any* $P \in \mathscr{P}$, *if* $\eta_1 =^{l_O}_{\pi_P(\Gamma)} \eta_2$, $\eta_1^G =^{l_O}_{\Gamma^G} \eta_2^G$ $(\eta_1^G, \eta_1); A; P \vdash c \rightsquigarrow (\eta_3^G, \eta_3)$, *and* $(\eta_2^G, \eta_2); A; P \vdash c \rightsquigarrow (\eta_4^G, \eta_4)$, *then* $\eta_3 =^{l_O}_{\pi_P(\Gamma)} \eta_4$ *and* $\eta_3^G =^{l_O}_{\Gamma^G} \eta_4^G$.

Proof. If $t(P) \not\leqslant l_O$, then according to Lemma 2.8, we have

$$\eta_3 =^{l_O}_{\pi_P(\Gamma)} \eta_1 =^{l_O}_{\pi_P(\Gamma)} \eta_2 =^{l_O}_{\pi_P(\Gamma)} \eta_4 \qquad \text{and} \qquad \eta_3^G =^{l_O}_{\eta^G} \eta_1^G =^{l_O}_{\Gamma^G} \eta_2^G =^{l_O}_{\Gamma^G} \eta_4^G$$

and thus the lemma follows. In the following, we assume that $t(P) \leqslant l_O$. The proof proceeds by induction on **r**(c), with subinduction on the derivations of $(\Gamma^G, \Gamma); A \vdash c : t$ and $(\eta_1^G, \eta_1); A; P \vdash c \rightsquigarrow (\eta_3^G, \eta_3)$. In the following, we shall omit the superscript l_O from $=^{l_O}_{\pi_P(\Gamma)}$ to simplify presentation.

T-ASS-L In this case, $c \equiv x := e$, x is non-global, and the typing derivation takes the form:

$$\frac{(\Gamma^G, \Gamma) \vdash e : \Gamma(x)}{(\Gamma^G, \Gamma); A \vdash x := e : \Gamma(x)}$$

where $t = \Gamma(x)$, and suppose the two executions of c are derived as follows:

$$\frac{(\eta_1^G, \eta_1) \vdash e \rightsquigarrow v_1}{(\eta_1^G, \eta_1); A; P \vdash x := e \rightsquigarrow (\eta_1^G, \eta_1[x \mapsto v_1])} \qquad \frac{(\eta_2^G, \eta_2) \vdash e \rightsquigarrow v_2}{(\eta_2^G, \eta_2); A; P \vdash x := e \rightsquigarrow (\eta_2^G, \eta_2[x \mapsto v_2])}$$

So $\eta_3 = \eta_1[x \mapsto v_1]$, $\eta_4 = \eta_2[x \mapsto v_2]$, $\eta_3^G = \eta_1^G$ and $\eta_4^G = \eta_2^G$. Note that $\eta_3^G =_{\Gamma^G} \eta_4^G$ holds trivially. Let us consider η_3 and η_4. As $t(P) \leqslant l_O$, applying Lemma 2.7 to $(\eta_1^G, \eta_1) \vdash e \rightsquigarrow v_1$ and $(\eta_2^G, \eta_2) \vdash e \rightsquigarrow v_2$ we get $v_1 = v_2$, so it then follows that $\eta_3 =_{\pi_P(\Gamma)} \eta_4$.

T-ASS-G In this case, $c \equiv x := e$, x is global, and the typing derivation takes the form:

$$\frac{(\Gamma^G, \Gamma) \vdash e : \Gamma^G(x)}{(\Gamma^G, \Gamma); A \vdash x := e : \Gamma^G(x)}$$

where $t = \Gamma^G(x)$, and suppose the two executions of c are derived as follows:

$$\frac{(\eta_1^G, \eta_1) \vdash e \rightsquigarrow v_1}{(\eta_1^G, \eta_1); A; P \vdash x := e \rightsquigarrow (\eta_1^G[x \mapsto v_1], \eta_1)} \qquad \frac{(\eta_2^G, \eta_2) \vdash e \rightsquigarrow v_2}{(\eta_2^G, \eta_2); A; P \vdash x := e \rightsquigarrow (\eta_2^G[x \mapsto v_2], \eta_2)}$$

So $\eta_3 = \eta_1$, $\eta_4 = \eta_2$, $\eta_3^G = \eta_1^G[x \mapsto v_1]$ and $\eta_4^G = \eta_2^G[x \mapsto v_2]$. Note that $\eta_3 =_{\pi_P(\Gamma)} \eta_4$ holds trivially. Let us consider η_3^G and η_4^G. As $t(P) \leqslant l_O$, applying Lemma 2.7 to $(\eta_1^G, \eta_1) \vdash e \rightsquigarrow v_1$ and $(\eta_2^G, \eta_2) \vdash e \rightsquigarrow v_2$ we get $v_1 = v_2$, so it then follows that $\eta_3^G =_{\Gamma^G} \eta_4^G$.

T-IF In this case $c \equiv \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2$ and we have

$$\frac{(\Gamma^G,\Gamma) \vdash e : t \qquad (\Gamma^G,\Gamma); A \vdash c_1 : t \qquad (\Gamma^G,\Gamma); A \vdash c_2 : t}{(\Gamma^G,\Gamma); A \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 : t}$$

The evaluation derivation under (η_1^G, η_1) takes either one of the following forms:

$$\frac{\begin{array}{c}(\eta_1^G,\eta_1) \vdash e \rightsquigarrow v \qquad v \neq 0 \\ (\eta_1^G,\eta_1); A; P \vdash c_1 \rightsquigarrow (\eta_3^G,\eta_3)\end{array}}{(\eta_1^G,\eta_1); A; P \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \rightsquigarrow (\eta_3^G,\eta_3)} \qquad \frac{\begin{array}{c}(\eta_1^G,\eta_1) \vdash e \rightsquigarrow v \qquad v = 0 \\ (\eta_1^G,\eta_1); A; P \vdash c_2 \rightsquigarrow (\eta_3^G,\eta_3)\end{array}}{(\eta_1^G,\eta_1); A; P \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \rightsquigarrow (\eta_3^G,\eta_3)}$$

We consider here only the case where $v \neq 0$; the case with $v = 0$ can be dealt with similarly. We first need to show that the evaluation of c under (η_2^G, η_2) would take the same if-branch. That is, suppose $(\eta_2^G,\eta_2) \vdash e \rightsquigarrow v'$. Since $t(P) \leqslant l_O$, we can apply Lemma 2.7 to conclude that $v' = v \neq 0$, hence the evaluation of c under (η_2^G, η_2) takes the form:

$$\frac{(\eta_2^G,\eta_2) \vdash e \rightsquigarrow v' \qquad v' \neq 0 \qquad (\eta_2^G,\eta_2); A; P \vdash c_1 \rightsquigarrow (\eta_4^G,\eta_4)}{(\eta_2^G,\eta_2); A; P \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \rightsquigarrow (\eta_4^G,\eta_4)}$$

The lemma then follows straightforwardly from the induction hypothesis.

T-WHILE $c \equiv \textbf{while } e \textbf{ do } c_b$ and we have

$$\frac{(\Gamma^G,\Gamma) \vdash e : t \qquad (\Gamma^G,\Gamma); A \vdash c_b : t}{(\Gamma^G,\Gamma); A \vdash \textbf{while } e \textbf{ do } c_b : t}$$

According to Lemma 2.7, if $(\eta_1^G,\eta_1) \vdash e \rightsquigarrow v_1$ and $(\eta_2^G,\eta_2) \vdash e \rightsquigarrow v_2$, then $v_1 = v_2$. If both are 0 then the conclusion holds according to (E-WHILE-F). Otherwise, we have

$$\frac{\begin{array}{c}(\eta_1^G,\eta_1) \vdash e \rightsquigarrow v_1 \quad v_1 \neq 0 \\ (\eta_1^G,\eta_1); A; P \vdash c_b \rightsquigarrow (\eta_5^G,\eta_5) \\ (\eta_5^G,\eta_5); A; P \vdash \textbf{while } e \textbf{ do } c_b \rightsquigarrow (\eta_3^G,\eta_3)\end{array}}{(\eta_1^G,\eta_1); A; P \vdash \textbf{while } e \textbf{ do } c_b \rightsquigarrow (\eta_3^G,\eta_3)} \qquad \frac{\begin{array}{c}(\eta_2^G,\eta_2) \vdash e \rightsquigarrow v_2 \quad v_2 \neq 0 \\ (\eta_2^G,\eta_2); A; P \vdash c_b \rightsquigarrow (\eta_6^G,\eta_6) \\ (\eta_6^G,\eta_6); A; P \vdash \textbf{while } e \textbf{ do } c_b \rightsquigarrow (\eta_4^G,\eta_4)\end{array}}{(\eta_2^G,\eta_2); A; P \vdash \textbf{while } e \textbf{ do } c_b \rightsquigarrow (\eta_4^G,\eta_4)}$$

Applying the induction hypothesis to $(\Gamma^G,\Gamma); A \vdash c_b : t$, $(\eta_1^G,\eta_1); A; P \vdash c_b \rightsquigarrow (\eta_5^G,\eta_5)$ and $(\eta_2^G,\eta_2); A; P \vdash c_b \rightsquigarrow (\eta_6^G,\eta_6)$, we obtain $\eta_5 =_{\pi_P(\Gamma)} \eta_6$ and $\eta_5^G =_{\Gamma^G} \eta_6^G$. Then applying the inner induction hypothesis to $(\eta_5^G,\eta_5); A; P \vdash \textbf{while } e \textbf{ do } c_b \rightsquigarrow (\eta_3^G,\eta_3)$ and $(\eta_6^G,\eta_6); A; P \vdash \textbf{while } e \textbf{ do } c_b \rightsquigarrow (\eta_4^G,\eta_4)$, we obtain $\eta_3 =_{\pi_P(\Gamma)} \eta_4$ and $\eta_3^G =_{\Gamma^G} \eta_4^G$.

T-SEQ In this case we have $c \equiv c_1; c_2$ and $(\Gamma^G,\Gamma); A \vdash c : t$. It holds by induction on c_1 and c_2.

T-LETVAR In this case we have $c \equiv \textbf{letvar } x = e \textbf{ in } c_b$. This case follows from the induction hypothesis and the fact that the mapping for the local variable x is removed in η_2 and η_2'.

T-CALL In this case, c has the form $x := \textbf{call } B.f(\overline{e})$. Suppose the typing derivation is the following (where we label the premises for ease of reference later):

$$\frac{FT(B.f) = \overline{s} \xrightarrow{s_b} s' \qquad (\mathbf{T_1})\ (\Gamma^G,\Gamma) \vdash \overline{e} : \pi_{\Theta(A)}(\overline{s}) \qquad (\mathbf{T_2})\ \pi_{\Theta(A)}(s') \leqslant \Gamma(x)}{(\Gamma^G,\Gamma); A \vdash x := \textbf{call } B.f(\overline{e}) : \Gamma(x) \sqcap \pi_{\Theta(A)}(s_b)}$$

where $t = \Gamma(x) \sqcap \pi_{\Theta(A)}(s_b)$, and the executions under (η_1^G, η_1) and (η_2^G, η_2) are derived, respectively, as follows:

$$\frac{(\mathbf{E_1})\ (\eta_1^G, \eta_1) \vdash \overline{e} \rightsquigarrow \overline{v_1} \qquad (\mathbf{E_2})\ (\eta_1^G, [\overline{y} \mapsto \overline{v_1}, r \mapsto 0]); B; \Theta(A) \vdash c_1 \rightsquigarrow (\eta_5^G, \eta_5)}{(\eta_1^G, \eta_1); A; P \vdash x := \textbf{call}\ B.f(\overline{e}) \rightsquigarrow (\eta_5^G, \eta_1[x \mapsto \eta_5(r)])}$$

$$\frac{(\mathbf{E'_1})\ (\eta_2^G, \eta_2) \vdash \overline{e} \rightsquigarrow \overline{v_2} \qquad (\mathbf{E'_2})\ (\eta_2^G, [\overline{y} \mapsto \overline{v_2}, r \mapsto 0]); B; \Theta(A) \vdash c_1 \rightsquigarrow (\eta_6^G, \eta_6)}{(\eta_2^G, \eta_2); A; P \vdash x := \textbf{call}\ B.f(\overline{e}) \rightsquigarrow (\eta_6^G, \eta_2[x \mapsto \eta_6(r)])}$$

where $FD(B.f) = B.f(\overline{x})\big\{\textbf{init}\ r = 0\ \textbf{in}\ \{c_1; \textbf{return}\ r\}\big\}$, $\eta_3 = \eta_1[x \mapsto \eta_5(r)]$, $\eta_4 = \eta_2[x \mapsto \eta_6(r)]$, $\eta_3^G = \eta_5^G$ and $\eta_4^G = \eta_6^G$.
Moreover, since we consider only well-typed systems, the function $FD(B.f)$ is also typable:

$$\frac{(\mathbf{T_3})\ (\Gamma^G, [\overline{y} : \overline{s}, r : s']); B \vdash c_1 : s_b}{\Gamma^G \vdash B.f(\overline{y})\big\{\textbf{init}\ r = 0\ \textbf{in}\ \{c_1; \textbf{return}\ r\}\big\} : \overline{s} \xrightarrow{s_b} s'}$$

Let $\Gamma' = \pi_{\Theta(A)}([\overline{y} : \overline{s}, r : s'])$. We first prove several claims:

- Claim 1: $[\overline{y} \mapsto \overline{v_1}, r \mapsto 0] =_{\Gamma'} [\overline{y} \mapsto \overline{v_2}, r \mapsto 0]$.
  Proof: Let $\rho = [\overline{y} \mapsto \overline{v_1}, r \mapsto 0]$ and $\rho' = [\overline{y} \mapsto \overline{v_2}, r \mapsto 0]$. We only need to check that the two mappings agree on mappings of $\overline{y}$ that are of type $\leqslant \hat{l}_O$. Suppose y_u is such a variable, i.e., $\Gamma'(y_u) = u \leqslant \hat{l}_O$, and suppose $\rho(y_u) = v_u$ and $\rho'(y_u) = v'_u$ for some $y_u \in \overline{y}$. From $(\mathbf{E_1})$ we have $(\eta_1^G, \eta_1) \vdash e_u \rightsquigarrow v_u$ and from $(\mathbf{E_2})$ we have $(\eta_2^G, \eta_2) \vdash e_u \rightsquigarrow v'_u$, and from $(\mathbf{T_1})$ we have $(\Gamma^G, \Gamma) \vdash e_u : u$. Since $u \leqslant \hat{l}_O$, applying Lemma 2.7, we get $v_u = v'_u$.
- Claim 2: $\eta_5 =_{\Gamma'} \eta_6$ and $\eta_5^G =_{\Gamma^G} \eta_6^G$.
  Proof: From Claim 1, we know that $[\overline{y} \mapsto \overline{v_1}, r \mapsto 0] =_{\Gamma'} [\overline{y} \mapsto \overline{v_2}, r \mapsto 0]$.
  If $s_b(\Theta(A)) \not\leqslant l_O$, similarly, the results follows according to Lemma 2.8. Let us assume $s_b(\Theta(A)) \leqslant l_O$. Since $\mathbf{r}(c_1) < \mathbf{r}(c)$, we can apply the outer induction hypothesis to $(\mathbf{E_2})$, $(\mathbf{E'_2})$ and $(\mathbf{T_3})$ to obtain $\eta_5 =_{\Gamma'} \eta_6$ and $\eta_5^G =_{\Gamma^G} \eta_6^G$.
- Claim 3: $\eta_1[x \mapsto \eta_5(r)] =_{\pi_P(\Gamma)} \eta_2[x \mapsto \eta_6(r)]$
  Proof: We first note that if $\Gamma(x)(P) \not\leqslant l_O$, then x is not observable at level l_O, and thus the result follows from $\eta_1 =_{\pi_P(\Gamma)} \eta_2$. Assume that $\Gamma(x)(P) \leqslant l_O$. From $(\mathbf{T_2})$, we get $(\pi_{\Theta(A)}(s'))(P) \leqslant l_O$. The latter, by Definition 2.6, implies that $\pi_{\Theta(A)}(s') \leqslant \hat{l_O}$. Since $r \in dom(\Gamma')$, it is obvious that $\eta_5(r) = \eta_6(r)$ From Claim 2.

The statement we are trying to prove, i.e., $\eta_3 =_{\pi_P(\Gamma)} \eta_4$ and $\eta_3^G =_{\Gamma^G} \eta_4^G$, follows immediately from Claim 2 and 3 above.

T-CP $c \equiv \textbf{test}(p)\ c_1\ \textbf{else}\ c_2$ and we have

$$\frac{(\Gamma^G, \Gamma \uparrow_p); A \vdash c_1 : t_1 \qquad (\Gamma^G, \Gamma \downarrow_p); A \vdash c_2 : t_2 \qquad t = t_1 \triangleright_p t_2}{(\Gamma^G, \Gamma); A \vdash \textbf{test}(p)\ c_1\ \textbf{else}\ c_2 : t}$$

We need to consider two cases, one where $p \in P$ and the other where $p \notin P$.

Assume that $p \in P$. Then the evaluation of c under (η_1^G, η_1) and (η_2^G, η_2) are respectively:

$$\frac{p \in P \quad (\eta_1^G, \eta_1); A; P \vdash c_1 \rightsquigarrow (\eta_3^G, \eta_3)}{(\eta_1^G, \eta_1); A; P \vdash \textbf{test}(p)\ c_1\ \textbf{else}\ c_2 \rightsquigarrow (\eta_3^G, \eta_3)} \qquad \frac{p \in P \quad (\eta_2^G, \eta_2); A; P \vdash c_1 \rightsquigarrow (\eta_4^G, \eta_4)}{(\eta_2^G, \eta_2); A; P \vdash \textbf{test}(p)\ c_1\ \textbf{else}\ c_2 \rightsquigarrow (\eta_4^G, \eta_4)}$$

since $p \in P$, we have $t_1(P) = t(P) \leqslant l_O$. Moreover, since $\eta_1 =_{\pi_P(\Gamma)} \eta_2$, by Lemma 2.5, we have $\eta_1 =_{\pi_P(\Gamma\uparrow_p)} \eta_2$. Therefore by the induction hypothesis applied to c_1, we obtain $\eta_3 =_{\pi_P(\Gamma\uparrow_p)} \eta_4$ and $\eta_3^G =_{\Gamma^G} \eta_4^G$. And by Lemma 2.5, we get $\eta_3 =_{\pi_P(\Gamma)} \eta_4$.
For the case where $p \notin P$, we apply a similar reasoning as above, but using Lemma 2.6 in place of Lemma 2.5.

□

Definition 2.12 *Let $\mathcal{S}$ be a system with the global typing environment Γ^G. 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 $FT(A.f) = \bar{t} \xrightarrow{s} t'$ is non-interferent *iff. for all η_1, η_2, η_1^G, η_2^G, P, v, l_O, if the following hold:*

- $t'(P) \leqslant l_O$,
- $\eta_1 =^{l_O}_{\pi_P(\Gamma)} \eta_2$ *and* $\eta_1^G =^{l_O}_{\Gamma^G} \eta_2^G$, *where* $\Gamma = [\overline{x} : \bar{t}, r : t']$,
- $(\eta_1^G, \eta_1); A; P \vdash c \rightsquigarrow (\eta_3^G, \eta_3)$, *and* $(\eta_2^G, \eta_2); A; P \vdash c \rightsquigarrow (\eta_4^G, \eta_4)$,

then $\eta_3(r) = \eta_4(r)$ and $\eta_3^G =^{l_O}_{\eta^G} \eta_4^G$. The system $\mathcal{S}$ is non-interferent *iff all functions in $\mathcal{S}$ are non-interferent.*

Theorem 2.1 *Well-typed systems are non-interferent.*

Proof. Follows from Lemma 2.9. □

3. Type Inference

This section describes a decidable inference algorithm for the language in Section 2.2. Section 3.1 firstly rewrites the typing rules (Fig. 2) in the form of permission trace rules (Fig. 6), then reduces the type inference into a constraint solving problem; Section 3.2 provides procedures to solve the generated constraints.

3.1. Constraint Generation

*3.1.1. **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 3.1 *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 3.2 *The* partial subtyping relation $\leqslant_\Lambda$*, which is the subtyping relation applied on the permission trace, is defined as $s \leqslant_\Lambda t$ iff. $s \cdot \Lambda \leqslant t \cdot \Lambda$.*

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

Lemma 3.1 *$\forall s, t \in \mathcal{T},\ s \leqslant t \implies s \leqslant_\Lambda t$ for all Λ.*

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

Lemma 3.2 *$\forall t \in \mathcal{T},\ p, q \in \mathbf{P}$ s.t. $p \neq q$, $t \cdot (\circledcirc p \circledast q) = t \cdot (\circledast q \circledcirc p)$, where $\circledcirc, \circledast, \in \{\oplus, \ominus\}$.*

Lemma 3.3 *$\forall t \in \mathcal{T}, (t \cdot \circledcirc p) \cdot \Lambda = (t \cdot \Lambda) \cdot \circledcirc p$, where $\circledcirc \in \{\oplus, \ominus\}$ and $p \notin \Lambda$.*

Lemma 3.4 *$\forall t \in \mathcal{T},\ p \in \mathbf{P}, (t \cdot \circledcirc p) \cdot \circledast p = t \cdot (\circledcirc p)$, where $\circledcirc, \circledast \in \{\oplus, \ominus\}$.*

Lemma 3.5 *$\forall t \in \mathcal{T},\ (t \cdot \Lambda) \cdot \Lambda = t \cdot \Lambda$.*

Lemmas 3.2 and 3.3 state that the order of applications of promotions and demotions on *different* permissions does not affect the result, which enables us to solve the constraints in any order (see Section 3.2). Lemmas 3.4 and 3.5 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 $p \in \mathbf{P}$. According to Lemmas 3.4 and 3.5, we can safely assume that all permission traces are consistent; we do so from now on. 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.

Lemma 3.6 *$\forall s, t \in \mathcal{T}.\ \forall p \in \mathbf{P}. (s \triangleright_p t) \cdot \Lambda = (s \cdot \Lambda) \triangleright_p (t \cdot \Lambda)$, where $p \notin \Lambda$.*

$$\text{TT-VAR-L}\frac{x \in dom(\Gamma)}{\Gamma^G;\Gamma;\Lambda \vdash_{tr} x : \Gamma(x)} \qquad \text{TT-ASS-L}\frac{x \in dom(\Gamma) \quad \Gamma^G;\Gamma;\Lambda \vdash_{tr} e : t \quad t \leqslant_\Lambda \Gamma(x)}{\Gamma^G;\Gamma;\Lambda \vdash_{tr} x := e : \Gamma(x)}$$

$$\text{TT-VAR-G}\frac{x \in dom(\Gamma^G)}{\Gamma^G;\Gamma;\Lambda \vdash_{tr} x : \Gamma^G(x)} \qquad \text{TT-ASS-G}\frac{x \in dom(\Gamma^G) \quad \Gamma^G;\Gamma;\Lambda \vdash_{tr} e : t \quad t \leqslant_\Lambda \Gamma^G(x)}{\Gamma^G;\Gamma;\Lambda \vdash_{tr} x := e : \Gamma^G(x)}$$

$$\text{TT-OP}\frac{\Gamma^G;\Gamma;\Lambda \vdash_{tr} e_1 : t_1 \quad \Gamma^G;\Gamma;\Lambda \vdash_{tr} e_2 : t_2}{\Gamma^G;\Gamma;\Lambda \vdash_{tr} e_1 \ \mathbf{op}\ e_2 : t_1 \sqcup t_2} \qquad \text{TT-SEQ}\frac{\Gamma^G;\Gamma;\Lambda;A \vdash_{tr} c_1 : t_1 \quad \Gamma^G;\Gamma;\Lambda;A \vdash_{tr} c_2 : t_2}{\Gamma^G;\Gamma;\Lambda;A \vdash_{tr} c_1;c_2 : t_1 \sqcap t_2}$$

$$\text{TT-IF}\frac{\Gamma^G;\Gamma;\Lambda \vdash_{tr} e : t \quad \Gamma^G;\Gamma;\Lambda;A \vdash_{tr} c_1 : t_1 \quad \Gamma^G;\Gamma;\Lambda;A \vdash_{tr} c_2 : t_2 \quad t \leqslant_\Lambda t_1 \sqcap t_2}{\Gamma^G;\Gamma;\Lambda;A \vdash_{tr} \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 : t_1 \sqcap t_2}$$

$$\text{TT-WHILE}\frac{\Gamma^G;\Gamma;\Lambda \vdash_{tr} e : s \quad \Gamma^G;\Gamma;\Lambda;A \vdash_{tr} c : t \quad s \leqslant_\Lambda t}{\Gamma^G;\Gamma;\Lambda;A \vdash_{tr} \mathbf{while}\ e\ \mathbf{do}\ c : t}$$

$$\text{TT-LETVAR}\frac{\Gamma^G;\Gamma;\Lambda \vdash_{tr} e : s \quad \Gamma^G;\Gamma[x : s'];\Lambda;A \vdash_{tr} c : t \quad s \leqslant_\Lambda s'}{\Gamma^G;\Gamma;\Lambda;A \vdash_{tr} \mathbf{letvar}\ x = e\ \mathbf{in}\ c : t}$$

$$\text{TT-CALL}\frac{FT(B.f) = \bar{t} \xrightarrow{s_b} t' \quad \Gamma^G;\Gamma;\Lambda \vdash_{tr} \bar{e} : \bar{s} \quad \bar{s} \leqslant_\Lambda \overline{\pi_{\Theta(A)}(t)} \quad \pi_{\Theta(A)}(t') \leqslant_\Lambda \Gamma(x)}{\Gamma^G;\Gamma;\Lambda;A \vdash_{tr} x := \mathbf{call}\ B.f(\bar{e}) : \Gamma(x) \sqcap \pi_{\Theta(A)}(s_b)}$$

$$\text{TT-CP}\frac{\Gamma^G;\Gamma;\Lambda :: \oplus p;A \vdash_{tr} c_1 : t_1 \quad \Gamma^G;\Gamma;\Lambda :: \ominus p;A \vdash_{tr} c_2 : t_2}{\Gamma^G;\Gamma;\Lambda;A \vdash_{tr} \mathbf{test}(p)\ c_1\ \mathbf{else}\ c_2 : t_1 \triangleright_p t_2}$$

$$\text{TT-FUN}\frac{\Gamma^G;[\bar{x} : \bar{t}, r : t'];\epsilon;B \vdash_{tr} c : s_b \quad s \leqslant s_b}{\Gamma^G \vdash_{tr} B.f(\bar{x})\{\mathbf{init}\ r = 0\ \mathbf{in}\ \{c;\mathbf{return}\ r\}\} : \bar{t} \xrightarrow{s} t'}$$

Fig. 6. Permission trace rules for expressions, commands and functions

Lemma 3.6 states the application of trace Λ and the merging along p are orthogonal if $p \notin \Lambda$. And the following lemma (Lemma 3.7) indicates the subtyping checking can be divided into two cases: one applied by $\oplus p$ and the other applied by $\ominus p$.

Lemma 3.7 *$\forall s,t \in \mathcal{T}.\ \forall p \in \mathbf{P}.\ s \leqslant t \iff s \cdot \oplus p \leqslant t \cdot \oplus p$ and $s \cdot \ominus p \leqslant t \cdot \ominus p$.*

3.1.2. ***Permission Trace Rules***

We keep the applications of the promotions and demotions symbolically (*i.e.*, representing the applications as combinations of 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 given in Fig. 6, which we call the *permission trace rules* and mark with a subscript *tr* (*i.e.*, $\vdash_{tr}$). 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 non-global variables depending on the permission checks, and that the subtyping relation in the trace rules is the partial subtyping one $\leqslant_\Lambda$.

Next, we show the trace rules are sound and complete with respect to the typing rules, that is, an expression (command, function, resp.) is typeable under the trace rules, if and only if it is typeable under the typing rules.

Lemma 3.8 *(a) If* $\Gamma^G;\Gamma;\Lambda \vdash_{tr} e : t$*, then* $(\Gamma^G,\Gamma\cdot\Lambda)\vdash e:(t\cdot\Lambda)$.

(b) If $\Gamma^G;\Gamma;\Lambda;A \vdash_{tr} c : t$*, then* $(\Gamma^G,\Gamma\cdot\Lambda);A\vdash c:(t\cdot\Lambda)$.

(c) If $\Gamma^G \vdash_{tr} B.f(\overline{x})\{\textbf{\textit{init}}\ r=0\ \textbf{\textit{in}}\ \{c;\textbf{\textit{return}}\ r\}\} : \bar{t}\xrightarrow{s} t'$*, then* $\Gamma^G\vdash B.f(\overline{x})\{\textbf{\textit{init}}\ r=0\ \textbf{\textit{in}}\ \{c;\textbf{\textit{return}}\ r\}\} : \bar{t}\xrightarrow{s} t'$.

Lemma 3.9 *(a) If* $(\Gamma^G,\Gamma\cdot\Lambda)\vdash e:t\cdot\Lambda$*, then there exists s such that* $\Gamma^G;\Gamma;\Lambda\vdash_{tr} e : s$ *and* $s\leqslant_\Lambda t$.

(b) If $(\Gamma^G,\Gamma\cdot\Lambda);A\vdash c:t\cdot\Lambda$*, then there exists s such that* $\Gamma^G;\Gamma;\Lambda;A\vdash_{tr} c : s$ *and* $t\leqslant_\Lambda s$.

(c) If $\Gamma^G\vdash B.f(\overline{x})\{\textbf{\textit{init}}\ r=0\ \textbf{\textit{in}}\ \{c;\textbf{\textit{return}}\ r\}\} : \bar{t}\xrightarrow{s_b} s$*, then* $\Gamma^G\vdash_{tr} B.f(\overline{x})\{\textbf{\textit{init}}\ r=0\ \textbf{\textit{in}}\ \{c;\textbf{\textit{return}}\ r\}\} : \bar{t}\xrightarrow{s_b} s$.

3.1.3. *Constraint Generation Rules*

To infer types for functions in System $\mathcal{S}$, we assign a function type $\overline{\alpha}\xrightarrow{\gamma}\beta$ for each function $A.f$ whose type is unknown and a type variable γ_x for each variable x with unknown type respectively, where $\overline{\alpha},\beta,\gamma,\gamma_x$ are fresh type variables. And we mark the type variables for global variables with superscripts. 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 $\leqslant_\Lambda$) 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.*, $\leqslant_\Lambda$), we define the permission guarded constraints as follows:

$$
\begin{array}{ll}
c & ::= (\Lambda, LHS \leqslant RHS)\\
LHS & ::= \alpha \mid \alpha^G \mid t_g \mid LHS \sqcup LHS \mid \pi_P(LHS)\\
RHS & ::= \alpha \mid \alpha^G \mid t_g \mid RHS \sqcap RHS \mid RHS \triangleright_p RHS \mid \pi_P(RHS)
\end{array}
$$

where Λ is a permission trace, α,α^G are fresh type variables for non-global and global variables respectively, and t_g is a ground type.

A *type substitution* is a finite mapping from type variables to security types:

$$\theta ::= \epsilon \mid \alpha\mapsto t,\theta \mid \alpha^G\mapsto \hat{l},\theta$$

Definition 3.3 *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* $(\Lambda, t_l\leqslant t_r)\in C$*,* $t_l\theta\leqslant_\Lambda t_r\theta$ *holds.*

The constraint generation rules are presented in Fig. 7, where each rule is marked with a subscript *cg* (*i.e.*, $\vdash_{cg}$), and 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.

Next, we show the constraint rules are *sound* and *complete* with respect to permission trace rules, that is, the constraint set generated by the derivation of an expression (command, function, resp.) under the

$$\text{TG-VAR-L}\frac{x \in dom(\Gamma)}{\Gamma^G;\Gamma;\Lambda \vdash_{cg} x : \Gamma(x) \rightsquigarrow \emptyset} \qquad \text{TG-ASS-L}\frac{x \in dom(\Gamma) \quad \Gamma^G;\Gamma;\Lambda \vdash_{cg} e : t \rightsquigarrow C}{\Gamma^G;\Gamma;\Lambda;A \vdash_{cg} x := e : \Gamma(x) \rightsquigarrow C \cup \{(\Lambda, t \leqslant \Gamma(x))\}}$$

$$\text{TG-VAR-G}\frac{x \in dom(\Gamma^G)}{\Gamma^G;\Gamma;\Lambda \vdash_{cg} x : \Gamma^G(x) \rightsquigarrow \emptyset} \qquad \text{TG-ASS-G}\frac{x \in dom(\Gamma^G) \quad \Gamma^G;\Gamma;\Lambda \vdash_{cg} e : t \rightsquigarrow C}{\Gamma^G;\Gamma;\Lambda;A \vdash_{cg} x := e : \Gamma^G(x) \rightsquigarrow C \cup \{(\Lambda, t \leqslant \Gamma^G(x))\}}$$

$$\text{TG-OP}\frac{\Gamma^G;\Gamma;\Lambda \vdash_{cg} e_1 : t_1 \rightsquigarrow C_1 \quad \Gamma^G;\Gamma;\Lambda \vdash_{cg} e_2 : t_2 \rightsquigarrow C_2}{\Gamma^G;\Gamma;\Lambda \vdash_{cg} e_1 \ \mathbf{op}\ e_2 : t_1 \sqcup t_2 \rightsquigarrow C_1 \cup C_2}$$

$$\text{TG-SEQ}\frac{\Gamma^G;\Gamma;\Lambda;A \vdash_{cg} c_1 : t_1 \rightsquigarrow C_1 \quad \Gamma^G;\Gamma;\Lambda;A \vdash_{cg} c_2 : t_2 \rightsquigarrow C_2}{\Gamma^G;\Gamma;\Lambda;A \vdash_{cg} c_1;c_2 : t_1 \sqcap t_2 \rightsquigarrow C_1 \cup C_2}$$

$$\text{TG-IF}\frac{\begin{array}{c}\Gamma^G;\Gamma;\Lambda;A \vdash_{cg} c_1 : t_1 \rightsquigarrow C_1 \qquad \Gamma^G;\Gamma;\Lambda;A \vdash_{cg} c_2 : t_2 \rightsquigarrow C_2 \\ \Gamma^G;\Gamma;\Lambda \vdash_{cg} e : t \rightsquigarrow C_e \quad C = C_e \cup C_1 \cup C_2 \cup \{(\Lambda, t \leqslant t_1 \sqcap t_2)\}\end{array}}{\Gamma^G;\Gamma;\Lambda;A \vdash_{cg} \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 : t_1 \sqcap t_2 \rightsquigarrow C}$$

$$\text{TG-WHILE}\frac{\Gamma^G;\Gamma;\Lambda \vdash_{cg} e : s \rightsquigarrow C \quad \Gamma^G;\Gamma;\Lambda;A \vdash_{cg} c : t \rightsquigarrow C'}{\Gamma^G;\Gamma;\Lambda;A \vdash_{cg} \mathbf{while}\ e\ \mathbf{do}\ c : t \rightsquigarrow C \cup C' \cup \{(\Lambda, s \leqslant t)\}}$$

$$\text{TG-LETVAR}\frac{\Gamma^G;\Gamma;\Lambda \vdash_{cg} e : s \rightsquigarrow C_1 \quad \Gamma^G;\Gamma[x:\alpha];\Lambda;A \vdash_{cg} c : t \rightsquigarrow C_2 \quad C = C_1 \cup C_2 \cup \{(\Lambda, s \leqslant \alpha)\}}{\Gamma^G;\Gamma;\Lambda;A \vdash_{cg} \mathbf{letvar}\ x = e\ \mathbf{in}\ c : t \rightsquigarrow C}$$

$$\text{TG-CALL}\frac{\begin{array}{c}FT_C(B.f) = (\overline{t} \xrightarrow{s_b} t', C_f) \quad \Gamma^G;\Gamma;\Lambda \vdash_{cg} \overline{e} : \overline{s} \rightsquigarrow \bigcup \overline{C_e} \\ C_a = \{(\Lambda, \overline{s} \leqslant \overline{\pi_{\Theta(A)}(t)}), (\Lambda, \pi_{\Theta(A)}(t') \leqslant \Gamma(x))\}\end{array}}{\Gamma^G;\Gamma;\Lambda;A \vdash_{cg} x := \mathbf{call}\ B.f(\overline{e}) : \Gamma(x) \sqcap \pi_{\Theta(A)}(s_b) \rightsquigarrow C_f \cup \bigcup \overline{C_e} \cup C_a}$$

$$\text{TG-CP}\frac{\Gamma^G;\Gamma;\Lambda :: \oplus p;A \vdash_{cg} c_1 : t_1 \rightsquigarrow C_1 \quad \Gamma^G;\Gamma;\Lambda :: \ominus p;A \vdash_{cg} c_2 : t_2 \rightsquigarrow C_2}{\Gamma^G;\Gamma;\Lambda;A \vdash_{cg} \mathbf{test}(p)\ c_1\ \mathbf{else}\ c_2 : t_1 \triangleright_p t_2 \rightsquigarrow C_1 \cup C_2}$$

$$\text{TG-FUN}\frac{\Gamma^G;[\overline{x}:\overline{\alpha}, r:\beta];\epsilon;B \vdash_{cg} c : s \rightsquigarrow C}{\Gamma^G \vdash_{cg} B.f(x)\{\mathbf{init}\ r = 0\ \mathbf{in}\ \{c;\mathbf{return}\ r\}\} : \overline{\alpha} \xrightarrow{\gamma} \beta \rightsquigarrow C \cup \{(\epsilon, \gamma \leqslant s)\}}$$

Fig. 7. Constraint generation rules for expressions, commands and functions, given function type table FT_C.

constraint rules is solvable, if and only if an expression (command, function, resp.) is typable under trace rules.

Lemma 3.10 *The following statements hold:*

(a) If $\Gamma^G;\Gamma;\Lambda \vdash_{cg} e : t \rightsquigarrow C$ *and* $\theta \vDash C$, *then* $\Gamma^G\theta;\Gamma\theta;\Lambda \vdash_{tr} e : t\theta$.

(b) If $\Gamma^G;\Gamma;\Lambda;A \vdash_{cg} c : t \rightsquigarrow C$ *and* $\theta \vDash C$, *then* $\Gamma^G\theta;\Gamma\theta;\Lambda;A \vdash_{tr} c : t\theta$.

(c) If $\Gamma^G \vdash_{cg} B.f(x)\{\textbf{\textit{init}}\ r = 0\ \textbf{\textit{in}}\ \{c;\textbf{\textit{return}}\ r\}\} : \overline{\alpha} \xrightarrow{\gamma} \beta \rightsquigarrow C$ *and* $\theta \vDash C$, *then* $\Gamma^G\theta \vdash_{tr} B.f(x)\{\textbf{\textit{init}}\ r = 0\ \textbf{\textit{in}}\ \{c;\textbf{\textit{return}}\ r\}\} : \overline{\theta(\alpha)} \xrightarrow{\theta(\gamma)} \theta(\beta)$.

```
A.getInfo() {
    init r in { // Γ(r) = α
        test(p) {
            test(q) r = loc; // (⊕p ⊕ q, l̂1 ⩽ α)
            else r = ""; // (⊕p ⊖ q, L̂ ⩽ α)
        } else {
            test(q) r = aid++loc; // (⊖p ⊕ q, Ĥ ⩽ α)
            else r = ""; // (⊖p ⊖ q, L̂ ⩽ α)
        }
        //(ϵ, αb ⩽ (α ▷q α) ▷p (α ▷q α))
        return r;
    }
}
B.fun() { // B has permission q
    init r in { // Γ(r) = β
        letvar x = "" in { // Γ(x) = γ, (ϵ, L̂ ⩽ γ)
            test(p) r = 0; // (⊕p, L̂ ⩽ β)
            else x = call A.getInfo();
            //FT_C(A.getInfo) = () --αb--> α,
            //(⊖p, π_Θ(B)(α) ⩽ γ)
            if x == "" then r = 0; // (ϵ, L̂ ⩽ β)
            else r = 1; // (ϵ, L̂ ⩽ β), (ϵ, γ ⩽ β ⊓ β)
        }
        //(ϵ, βb ⩽ (β ▷p (γ ⊓ π_Θ(B)(αb))) ⊓ (β ⊓ β))
        return r;
    }
}
```

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

Lemma 3.11 *The following statements hold:*

(a) If $\Gamma^G;\Gamma;\Lambda \vdash_{tr} e : t$, *then there exist* $\Gamma_1, \Gamma_1^G, s, C, \theta$ *such that* $\Gamma_1^G;\Gamma_1;\Lambda \vdash_{cg} e : s \rightsquigarrow C$, $\theta \vDash C$, $\Gamma_1\theta = \Gamma$, $\Gamma_1^G\theta = \Gamma^G$ *and* $s\theta = t$.

(b) If $\Gamma^G;\Gamma;\Lambda;A \vdash_{tr} c : t$, *then there exist* $\Gamma_1, \Gamma_1^G, s, C, \theta$ *such that* $\Gamma_1^G;\Gamma_1;\Lambda;A \vdash_{cg} c : s \rightsquigarrow C$, $\theta \vDash C$, $\Gamma_1\theta = \Gamma$, $\Gamma_1^G = \Gamma^G$, *and* $s\theta = t$.

(c) If $\Gamma^G \vdash_{tr} B.f(\overline{x})\{\textbf{\textit{init}}\ r = 0\ \textbf{\textit{in}}\ \{c; \textbf{\textit{return}}\ r\}\} : \overline{t_p} \xrightarrow{t_b} t_r$, *then there exist* $\Gamma_1^G, \overline{\alpha}, \beta, \gamma, C, \theta$ *such that* $\Gamma_1^G \vdash_{cg} B.f(\overline{x})\{\textbf{\textit{init}}\ r = 0\ \textbf{\textit{in}}\ \{c; \textbf{\textit{return}}\ r\}\} : \overline{\alpha} \xrightarrow{\gamma} \beta \rightsquigarrow C$, $\theta \vDash C$, $\Gamma_1^G\theta = \Gamma^G$, *and* $(\overline{\alpha} \xrightarrow{\gamma} \beta)\theta = \overline{t_p} \xrightarrow{t_b} t_r$, *where* α, β, γ *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 *aid* respectively, $\Theta(B) = \{q\}$, and $l_1 \sqcup l_2 = H$. Let us apply the constraint generation rules in Fig. 7 on each function, yielding the constraint sets C_A and C_B[5]

$$C_A = \{(\oplus p \oplus q, \hat{l_1} \leqslant \alpha), (\oplus p \ominus q, \hat{L} \leqslant \alpha), (\ominus p \oplus q, \hat{H} \leqslant \alpha), (\ominus p \ominus q, \hat{L} \leqslant \alpha), (\epsilon, \alpha_b \leqslant (\alpha \triangleright_q \alpha) \triangleright_p (\alpha \triangleright_q \alpha))\}$$
$$C_B = \{(\epsilon, \hat{L} \leqslant \gamma), (\oplus p, \hat{L} \leqslant \beta), (\ominus p, \pi_{\Theta(B)}(\alpha) \leqslant \gamma), (\epsilon, \hat{L} \leqslant \beta), (\epsilon, \gamma \leqslant \beta \sqcap \beta),$$
$$(\epsilon, \beta_b \leqslant (\beta \triangleright_p (\gamma \sqcap \pi_{\Theta(B)}(\alpha_b))) \sqcap (\beta \sqcap \beta))\}$$

[5] If we split the types for commands into global ones and non-global ones, we could have simpler constraints.

and the types $t_A = () \xrightarrow{\alpha_b} \alpha$ and $t_B = () \xrightarrow{\beta_b} \beta$ for the functions *getInfo* and *fun*[6] respectively. Thus, the constraint set C_{eg} for the whole program is $C_A \cup C_B$.

3.2. Constraint Solving

To start with, we present the interpretation and the difference on traces that are needed for constraint solving. A permission trace Λ is a property on permission sets and can be interpreted as a set of permission sets. Formally, the *interpretation* of Λ is

$$\mathcal{I}(\Lambda) = \{P \mid \forall \oplus p \in \Lambda.\ p \in P \text{ and } \forall \ominus p \in \Lambda.\ p \notin P\}$$

We said a permission trace Λ is *satisfiable*, denoted by $\Delta(\Lambda)$, iff. $\mathcal{I}(\Lambda) \neq \emptyset$; and a permission set P *entails* Λ, denoted by $P \vDash \Lambda$, iff. $P \in \mathcal{I}(\Lambda)$. We write Λ_P for the permission trace that only P can entail (*i.e.*, $\mathcal{I}(\Lambda_P) = \{P\}$). Given two traces Λ_1, Λ_2, the *difference* of Λ_1 from Λ_2, denoted by $dif(\Lambda_1,\ \Lambda_2)$, is the trace consisting of the promotions and demotions in Λ_1 but not in Λ_2. For example, $dif(\oplus p \ominus q \oplus r,\ \ominus q \oplus l) = \oplus p \oplus r$.

We now present an algorithm for solving the constraints generated by the rules in Fig. 7. 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 $(\Lambda, \pi_P(t_l) \leqslant \pi_Q(\alpha))$ for example, t_l is indeed guarded by Λ_P while α is guarded by Λ_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: $((\Lambda_l, t_l) \leqslant (\Lambda_r, t_r))$, 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 $((\Lambda_l, t_l) \leqslant (\Lambda_r, t_r)) \in C$, $(t_l\theta \cdot \Lambda_l) \leqslant (t_r\theta \cdot \Lambda_r)$ holds.

It is easy to transform a permission guarded constraint set C into a generalized constraint set C': by rewriting each $(\Lambda, t_l \leqslant t_r)$ as $((\Lambda, t_l) \leqslant (\Lambda, t_r))$. 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 constraint set C_{eg} can be rewritten as follows, where the first two lines are from C_A while the last two lines from C_B:

$$\begin{aligned} C_{eg} = \{&((\oplus p \oplus q, \hat{l_1}) \leqslant (\oplus p \oplus q, \alpha)), ((\oplus p \ominus q, \hat{L}) \leqslant (\oplus p \ominus q, \alpha)), ((\ominus p \oplus q, \hat{H}) \leqslant (\ominus p \oplus q, \alpha)), \\ &((\ominus p \ominus q, \hat{L}) \leqslant (\ominus p \ominus q, \alpha)), ((\epsilon, \alpha_b) \leqslant (\epsilon, (\alpha \triangleright_q \alpha) \triangleright_p (\alpha \triangleright_q \alpha))), \\ &((\epsilon, \hat{L}) \leqslant (\epsilon, \gamma)), ((\oplus p, \hat{L}) \leqslant (\oplus p, \beta)), ((\ominus p, \pi_{\Theta(B)}(\alpha)) \leqslant (\ominus p, \gamma)), ((\epsilon, \hat{L}) \leqslant (\epsilon, \beta)), \\ &((\epsilon, \gamma) \leqslant (\epsilon, \beta \sqcap \beta)), ((\epsilon, \beta_b) \leqslant (\epsilon, (\beta \triangleright_p (\gamma \sqcap \pi_{\Theta(B)}(\alpha_b))) \sqcap (\beta \sqcap \beta)))\} \end{aligned}$$

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.

3.2.1. ***Decomposition***

The first step is to decompose the types into the simpler ones, namely, type variables and ground types, according to their structures. This decomposition is formalized as the decomposing rules, which are given in Fig. 8. Rules (CD-CUP), (CD-CAP), (CD-SVAR) and (CD-MEGER$_0$)[7] are trivial. Rule

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

[7] Rule (CD-MEGER$_0$) is not necessary but can simplify the constraints as shown in the illustrated example.

$$\text{CD-CUP}\ \frac{}{C \cup \{((\Lambda, t_1 \sqcup t_2) \leqslant (\Lambda', t))\} \rightsquigarrow_d C \cup \{((\Lambda, t_1) \leqslant (\Lambda', t)), ((\Lambda, t_2) \leqslant (\Lambda', t))\}}$$

$$\text{CD-CAP}\ \frac{}{C \cup \{((\Lambda, t) \leqslant (\Lambda', t_1 \sqcap t_2))\} \rightsquigarrow_d C \cup \{((\Lambda, t) \leqslant (\Lambda', t_1)), ((\Lambda, t) \leqslant (\Lambda', t_2))\}}$$

$$\text{CD-SVAR}\ \frac{}{C \cup \{((\Lambda, \alpha) \leqslant (\Lambda, \alpha))\} \rightsquigarrow_d C}$$

$$\text{CD-LAPP}\ \frac{}{C \cup \{((\Lambda, \pi_P(t)) \leqslant (\Lambda', t'))\} \rightsquigarrow_d C \cup \{((\Lambda_P, t) \leqslant (\Lambda', t'))\}}$$

$$\text{CD-RAPP}\ \frac{}{C \cup \{((\Lambda, t) \leqslant (\Lambda', \pi_P(t')))\} \rightsquigarrow_d C \cup \{((\Lambda, t) \leqslant (\Lambda_P, t'))\}}$$

$$\text{CD-MERGE}_0\ \frac{t_1 \equiv t_2}{C \cup \{((\Lambda, t) \leqslant (\Lambda', t_1 \triangleright_p t_2))\} \rightsquigarrow_d C \cup \{((\Lambda, t) \leqslant (\Lambda', t_1))\}}$$

$$\text{CD-MERGE}_1\ \frac{t_1 \not\equiv t_2 \qquad C' = \{((\Lambda :: \oplus p, t) \leqslant (\Lambda' :: \oplus p, t_1)), ((\Lambda :: \ominus p, t) \leqslant (\Lambda' :: \ominus p, t_2))\}}{C \cup \{((\Lambda, t) \leqslant (\Lambda', t_1 \triangleright_p t_2))\} \rightsquigarrow_d C \cup C'}$$

$$\text{CD-SUB}_0\ \frac{t_g \cdot \Lambda \leqslant s_g \cdot \Lambda'}{C \cup \{((\Lambda, t_g) \leqslant (\Lambda', s_g))\} \rightsquigarrow_d C} \qquad \text{CD-SUB}_1\ \frac{t_g \cdot \Lambda \not\leqslant s_g \cdot \Lambda'}{C \cup \{((\Lambda, t_g) \leqslant (\Lambda', s_g))\} \rightsquigarrow_d \bot}$$

$$\text{CS-LU}\ \frac{\Delta(\Lambda_1 :: \Lambda_2) \qquad \{((\Lambda_l :: \mathit{dif}(\Lambda_1 :: \Lambda_2,\ \Lambda_1), t_l) \leqslant (\Lambda_r :: \mathit{dif}(\Lambda_1 :: \Lambda_2,\ \Lambda_2), t_r))\} \rightsquigarrow_d C' \qquad C' \not\subseteq C}{\{((\Lambda_l, t_l) \leqslant (\Lambda_1, \alpha)), ((\Lambda_2, \alpha) \leqslant (\Lambda_r, t_r))\} \subseteq C \rightsquigarrow_s C \cup C'}$$

Fig. 8. Constraint solving rules, including *decomposition* (**CD-**) and *saturation* (**CS-**).

(CD-MEGER$_1$) states that two p-merged types satisfy the relation if and only if both their p-promotions and p-demotions satisfy the relation, where t can be viewed as a p-merged type $t \triangleright_p t$. The projection of types yields a "monomorphic type" such that any successive trace application makes no changes, therefore we have (CD-LAPP) and (CD-RAPP). Rules (CD-SUB$_0$) and (CD-SUB$_1$) handle the constraints on ground types, where $\bot$ denotes the failure case. Let *dec* be the procedure for the constraint decomposition.

After decomposition, constraints have one of the forms:

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

Considering the constraint set C_{eg}, these are four constraints that need to be decomposed. Take the constraint $((\epsilon, \beta_b) \leqslant (\epsilon, (\beta \triangleright_p (\gamma \sqcap \pi_{\Theta(B)}(\alpha_b))) \sqcap (\beta \sqcap \beta)))$ for example, where $\Theta(B) = \{q\}$. Rules (CD-CAP), (CD-MEGER$_1$), and (CD-RAPP) are performed, yielding $\{((\epsilon, \beta_b) \leqslant (\epsilon, \beta)), ((\ominus p, \beta_b) \leqslant$

$(\ominus p,\gamma)),((\ominus p,\beta_b) \leqslant (\ominus p \oplus q,\alpha_b)),((\oplus p,\beta_b) \leqslant (\oplus p,\beta))\}$. After decomposing, C_{eg} becomes

$$\{((\oplus p \oplus q,\hat{l_1}) \leqslant (\oplus p \oplus q,\alpha)),((\oplus p \ominus q,\hat{L}) \leqslant (\oplus p \ominus q,\alpha)),((\ominus p \oplus q,\hat{H}) \leqslant (\ominus p \oplus q,\alpha)),$$
$$((\ominus p \ominus q,\hat{L}) \leqslant (\ominus p \ominus q,\alpha)),((\epsilon,\alpha_b) \leqslant (\epsilon,\alpha)),((\epsilon,\hat{L}) \leqslant (\epsilon,\gamma)),((\oplus p,\hat{L}) \leqslant (\oplus p,\beta)),$$
$$((\ominus p \oplus q,\alpha) \leqslant (\ominus p,\gamma)),((\epsilon,\hat{L}) \leqslant (\epsilon,\beta)),((\epsilon,\gamma) \leqslant (\epsilon,\beta)),((\epsilon,\beta_b) \leqslant (\epsilon,\beta)),$$
$$((\ominus p,\beta_b) \leqslant (\ominus p,\gamma)),((\ominus p,\beta_b) \leqslant (\ominus p \oplus q,\alpha_b)),((\oplus p,\beta_b) \leqslant (\oplus p,\beta))\}$$

where the constraints in blue are generated via decomposition.

3.2.2. *Saturation*

Considering a variable α, to ensure any lower bound (e.g., $((\Lambda_l,t_l) \leqslant (\Lambda_1,\alpha))$) is "smaller" than any of its upper bound (e.g., $((\Lambda_2,\alpha) \leqslant (\Lambda_r,t_r))$), 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, namely, the intersection of their interpretations are not empty. 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.*, $\Lambda_1 :: \Lambda_2$[8]), by adding the missing traces (*i.e.*, $dif(\Lambda_1 :: \Lambda_2,\ \Lambda_1)$ for lower bound constraint while $dif(\Lambda_1 :: \Lambda_2,\ \Lambda_2)$ for the upper one). This is done by the saturation rule (*i.e.*, CS-LU in Fig. 8). Let *sat* be the procedure for the constraint saturation.

Assume that there is an order $<$ on type variables and the smaller variable has a higher priority. If two variables α,β with $\alpha < \beta$ are in the same constraint $\beta \leqslant \alpha$ (or $\alpha \leqslant \beta$), we consider the variable β with the larger order is a bound for the variable α with the smaller order during constraint solving, but not vice-versa. There is a special case where both variables on two sides are the same, *e.g.*, $((\Lambda,\alpha) \leqslant (\Lambda',\alpha))$. In that case, we regroup all the traces of the variable α as the trace set $\{\Lambda_i \mid i \in I\}$ such that the set is full (*i.e.*, $\bigcup_{i\in I}\mathcal{I}(\Lambda_i) = \mathbf{P}$) and disjoint (*i.e.*, $\forall i,j \in I.i \neq j \Rightarrow \mathcal{I}(\Lambda_i) \cap \mathcal{I}(\Lambda_j) = \emptyset$), and rewrite the constraints of α w.r.t. the set $\{\Lambda_i \mid i \in I\}$. A possible trace set is the combination of the promotions and demotions on the permissions related to the variable α. For example, the combination on the permission set $\{p,q\}$ is $\{\oplus p :: \oplus q,\oplus p :: \ominus q,\ominus p :: \oplus q,\ominus p :: \ominus q\}$. Then we treat each (Λ_i,α) as different fresh variables α_i. Therefore, with the ordering, there are no loops like: $(\Lambda,\alpha) \leqslant \ldots \leqslant (\Lambda',\alpha)$. Note that different ordering could yield different results.

Let us consider the constraint set C_{eg} and assume that the order on variables is $\beta_b < \alpha_b < \alpha < \gamma < \beta$. There are four lower bounds and one upper bound for α. But only the lower bound $((\ominus p \oplus q,\hat{H}) \leqslant (\ominus p \oplus q,\alpha))$ shares the same satisfiable trace with the upper bound $((\ominus p \oplus q,\alpha) \leqslant (\ominus p,\gamma))$. So we saturate the set with the constraint $((\ominus p \oplus q,\hat{H}) \leqslant (\ominus p,\gamma))$. Note that the constraint $((\epsilon,\alpha_b) \leqslant (\epsilon,\alpha))$ is not considered as a lower bound for α due to $\alpha_b < \alpha$. Likewise, there are two lower bounds (*i.e.*, $((\epsilon,\hat{L}) \leqslant (\epsilon,\gamma))$ and the one newly generated above) and one upper bound (*i.e.*, $((\epsilon,\gamma) \leqslant (\epsilon,\beta))$) for γ. Each lower bound has a satisfiable intersected trace with the upper bound, which yields the following constraints $((\epsilon,\hat{L}) \leqslant (\epsilon,\beta))$ (already existing in C_{eg}) and $((\ominus p \oplus q,\hat{H}) \leqslant (\ominus p,\beta))$ (extended by $\ominus p$). While there are no upper bounds for β and no lower bounds for α_b and β_b, so no constraints are generated.

[8]It should be $\Lambda_1 :: dif(\Lambda_1,\Lambda_2)$ or $\Lambda_2 :: dif(\Lambda_2,\Lambda_1)$. But thanks to Lemmas 3.4 and 3.2, we assume the duplicated promotions and demotions can be removed implicitly and the remaining promotions and demotions can be reordered if needed. Here we write $\Lambda_1 :: \Lambda_2$ for short.

After saturation, the example set C_{eg} is

$$\{((\epsilon,\beta_b) \leqslant (\epsilon,\beta)), ((\ominus p,\beta_b) \leqslant (\ominus p,\gamma)), ((\ominus p,\beta_b) \leqslant (\ominus p \oplus q,\alpha_b)), ((\oplus p,\beta_b) \leqslant (\oplus p,\beta)),$$
$$((\epsilon,\alpha_b) \leqslant (\epsilon,\alpha)), ((\oplus p \oplus q,\hat{l}_1) \leqslant (\oplus p \oplus q,\alpha)), ((\oplus p \ominus q,\hat{L}) \leqslant (\oplus p \ominus q,\alpha)), ((\ominus p \oplus q,\hat{H}) \leqslant (\ominus p \oplus q,\alpha)),$$
$$((\ominus p \ominus q,\hat{L}) \leqslant (\ominus p \ominus q,\alpha)), ((\ominus p \oplus q,\alpha) \leqslant (\ominus p,\gamma)), ((\ominus p \oplus q,\hat{H}) \leqslant (\ominus p,\gamma)), ((\epsilon,\hat{L}) \leqslant (\epsilon,\gamma)),$$
$$((\epsilon,\gamma) \leqslant (\epsilon,\beta)), ((\ominus p \oplus q,\hat{H}) \leqslant (\ominus p,\beta)), ((\epsilon,\hat{L}) \leqslant (\epsilon,\beta)), ((\oplus p,\hat{L}) \leqslant (\oplus p,\beta))\}$$

where the constraints are rearranged via the ordering and the ones in blue are newly added via saturation.

Given two constraint sets C_1, C_2, we say C_1 entails C_2, denoted as $C_1 \vDash C_2$, iff. for any substitution θ, if $\theta \vDash C_1$, then $\theta \vDash C_2$. We proved that the constraint solving rules are sound and complete, that is, the original constraint set entails the converted set (obtained by decomposition and saturation), and vice-versa.

Lemma 3.12 *If $C \rightsquigarrow_r C'$, then $C \vDash C'$ and $C' \vDash C$, where $r \in \{d, s\}$.*

*3.2.3. **Unification***

There are two kinds of variables in our setting: global variables and non-global ones, and the constraints for them are all guarded by permission traces. Since the types for non-global variables are dependent on the permission sets, we need to consider the satisfiability of (any subset of) the permission traces of a non-global variable α under any permission set when constructing a type for it. For that, we consider the evaluation of a type t guarded by a permission trace Λ along a permission set P. If $P \vDash \Lambda$, according to Definition 2.5 and Lemma 2.3, then the security level along P is preserved after the application of Λ. So clearly $(t \cdot \Lambda)(P) = t(P)$. While for $P \nvDash \Lambda$, the security level along P is no longer preserved. Let us consider the simple case $P \nvDash \ominus p$ (*i.e.*, $p \in P$). According to Definition 2.5, we have $(t \cdot \ominus p)(P) = (t \downarrow_p)(P) = t(P \setminus \{p\})$. That is, the security level along a set P containing p is updated as the one along $P \setminus \{p\}$. Similar to $\oplus p$. Therefore, for a trace Λ, we have $(t \cdot \Lambda)(P) = t(P \cdot \Lambda)$, where the application $P \cdot \Lambda$ is defined as $P \cdot (\oplus p :: \Lambda') = (P \cup \{p\}) \cdot \Lambda'$, $P \cdot (\ominus p :: \Lambda') = (P \setminus \{p\}) \cdot \Lambda'$, and $P \cdot \epsilon = P$. It is easy to show that $P \cdot \Lambda = P$ if $P \vDash \Lambda$. So we can uniform the above two cases into $(t \cdot \Lambda)(P) = t(P \cdot \Lambda)$.

Let us consider a non-global variable α and assume that the constraints on it to be solved are $\{((\Lambda_i^l, t_i^l) \leqslant (\Lambda_i, \alpha))\}_{i \in I}$ (*i.e.*, the lower bounds) and $\{((\Lambda_j, \alpha) \leqslant (\Lambda_j^r, t_j^r))\}_{j \in J}$ (*i.e.*, the upper bounds). According the above discussion, for any permission set P, α can take a type t such that $(t \cdot \Lambda_i)(P)$ (*i.e.*, $t(P \cdot \Lambda_i)$) is bigger than $(t_i^l \cdot \Lambda_i^l)(P)$ for any Λ_i and that $(t \cdot \Lambda_j)(P)$ (*i.e.*, $t(P \cdot \Lambda_j)$) is smaller than $(t_j^r \cdot \Lambda_j^r)(P)$ for any Λ_j. Consequently, $t(P)$ should be bigger than the union $\bigsqcup_{P' \cdot \Lambda_i = P}(t_i^l \cdot \Lambda_i^l)(P')$ and smaller than the intersection $\bigsqcap_{P' \cdot \Lambda_j = P}(t_j^r \cdot \Lambda_j^r)(P')$. In other words, $t(P)$ is equivalent to $((\bigsqcup_{P' \cdot \Lambda_i = P} t_i^l \cdot \Lambda_i^l)(P') \sqcup \alpha'(P)) \sqcap (\bigsqcap_{P' \cdot \Lambda_j = P} t_j^r \cdot \Lambda_j^r)(P')$, where α' is a fresh type variable. Note that the constraint contributes to $t(P)$ if P entails its guarded permission trace Λ. The type above is exactly what we want. We define the construction of the above type via the function *toType*:

$$\begin{array}{l} \mathit{toType}(\{(\Lambda_i^l, t_i^l, \Lambda_i, \alpha)\}_{i \in I}, \{(\Lambda_j, \alpha, \Lambda_j^r, t_j^r)\}_{j \in J}) = \\ \quad \textbf{let } S_I^P = \{(i, P') \mid P' \cdot \Lambda_i = P, P' \subseteq \mathbf{P}, i \in I\} \textbf{ and } S_J^P = \{(j, P') \mid P' \cdot \Lambda_j = P, P' \subseteq \mathbf{P}, j \in J\} \textbf{ in} \\ \quad \{P \mapsto (\bigsqcup_{(i,P') \in S_I^P}(t_i^l \cdot \Lambda_i^l)(P') \sqcup \alpha'(P)) \sqcap \bigsqcap_{(j,P') \in S_J^P}(t_j^r \cdot \Lambda_j^r)(P') \mid P \subseteq \mathbf{P}\} \end{array}$$

(with the convention $\bigsqcup_{(i,P') \in \emptyset} t_i(P') = L$ and $\bigsqcap_{(j,P') \in \emptyset} t_j(P') = H$.) Note that the fresh variables enable us to get a principal type [18] such that every possible type can be obtained from it via subsumption or instantiation. In practice, we take the least possible type.

Since the types for global variables are invariant for all permission sets, that is, the type for a global variable α^G should be a level type, e.g., $\hat{l}$. So the level type $\hat{l}$ should be bigger than all its lower bounds

and smaller than all its upper bounds. Let us consider a global variable α^G and assume that the constraints on it to be solved are $\{((\Lambda_i^l, t_i^l) \leqslant (\Lambda_i, \alpha^G))\}_{i\in I}$ (*i.e.*, the lower bounds) and $\{((\Lambda_j, \alpha^G) \leqslant (\Lambda_j^r, t_j^r))\}_{j\in J}$ (*i.e.*, the upper bounds). Given a type t, let $maxlevel(t)$ and $minlevel(t)$ denote the maximum level and the minimum level in type t, respectively. So the level l should be bigger than any maximum level of all its lower bounds (i.e., $l \geqslant maxlevel(t_i^l \cdot \Lambda_i^l)$) and smaller than any minimum level of all its upper bounds (i.e., $l \leqslant minlevel(t_j^r \cdot \Lambda_j^r)$). In other words, l can take any level ranging from $\bigsqcup_{i\in I} maxlevel(t_i^l \cdot \Lambda_i^l)$ to $\bigsqcap_{j\in J} minlevel(t_j^r \cdot \Lambda_j^r)$, which indicates that l is equivalent to $(\bigsqcup_{i\in I} maxlevel(t_i^l \cdot \Lambda_i^l) \sqcup l_{\alpha^G}) \sqcap \bigsqcap_{j\in J} minlevel(t_j^r \cdot \Lambda_j^r)$, where l_{α^G} is a fresh level variable. Likewise, we define the construction of the type above via the function $toType^G$:

$toType^G(\{(\Lambda_i^l, t_i^l, \Lambda_i, \alpha^G)\}_{i\in I}, \{(\Lambda_j, \alpha^G, \Lambda_j^r, t_j^r)\}_{j\in J}) =$

let $l_l = \bigsqcup_{i\in I} maxlevel(t_i^l \cdot \Lambda_i^l)$ **and** $l_r = \bigsqcap_{j\in J} minlevel(t_j^r \cdot \Lambda_j^r)$ **in** $(l_l \sqcup \widehat{l_{\alpha^G}}) \sqcap l_r$

(with the convention $\bigsqcup_{i\in\emptyset} l_i = L$ and $\bigsqcap_{j\in\emptyset} l_j = H$.)

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* θ $((\Lambda_l, t_l, \Lambda_r, t_r)) = ((\Lambda_l, t_l\theta, \Lambda_r, t_r\theta))$ **in**
  if the maximum variable α exists in C
    construct a type t_α for α via the function $toType$ or $toType^G$
    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 []

Consider the constraint set C_{eg} again and assume the set $\mathbf{P}$ of all permissions is $\{p, q\}$. Firstly, let us take the constraints on the maximum variable β, which are the following set without any upper bounds

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

For the security level along the permission set $\emptyset$, there are three combinations contributing to it: $(\ominus p, \emptyset)$, $(\ominus p, \{p\})$ and $(\epsilon, \emptyset)$. So the level for $\emptyset$ is $((\hat{H} \cdot \ominus p \oplus q)(\emptyset) \sqcup (\hat{H} \cdot \ominus p \oplus q)(\{p\}) \sqcup (\hat{L} \cdot \epsilon)(\emptyset) \sqcup \beta'(\emptyset)) \sqcap H$, which is equivalent to H. Similar to the other sets. So by applying the function $toType$, we construct for β the type $t_\beta = \{\emptyset \mapsto H, \{p\} \mapsto (L \sqcup \beta'(\{p\})) \sqcap H, \{q\} \mapsto H, \{p, q\} \mapsto (L \sqcup \beta'(\{p, q\})) \sqcap H\}$, where β' is a fresh variable. For simplicity, we pick the least possible upper bound when constructing types. So we take $\{\emptyset \mapsto H, \{p\} \mapsto L, \{q\} \mapsto H, \{p, q\} \mapsto L\}$ 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, α, α_b, and β_b. Finally, the types constructed for these type variables are $t_\gamma = t_\beta$ and $t_\alpha = \{\emptyset \mapsto L, \{p\} \mapsto L, \{q\} \mapsto H, \{p, q\} \mapsto l_1\}$, $t_{\alpha_b} = t_\alpha$, $t_{\beta_b} = t_\beta$, respectively. Therefore, the types we infer for *A.getInfo* and *B.fun* are $() \xrightarrow{t_\alpha} t_\alpha$ and $() \xrightarrow{t_\beta} t_\beta$, respectively.

Next, we show that our unification is sound and complete.

Lemma 3.13 *If* $unify(C) = \theta$, *then* $\theta \vDash C$.

Lemma 3.14 *If* $\theta \vDash C$, *then there exist* θ' *and* θ'' *such that* $unify(C) = \theta'$ *and* $\theta = \theta'\theta''$.

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.

Lemma 3.15 *If $sol(C) = \theta$, then $\theta \vDash C$.*

Proof. By Lemma 3.12 and Lemma 3.13. □

Lemma 3.16 *If $\theta \vDash C$, then there exist θ' and θ'' such that $sol(C) = \theta'$ and $\theta = \theta'\theta''$.*

Proof. By Lemma 3.12 and Lemma 3.14. □

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 3.1 *The type inference system is sound, complete and decidable.*

Proof.
- **sound**: By Lemma 3.8, Lemma3.10, and Lemma 3.15.
- **complete**: By Lemma 3.9, Lemma3.11, and Lemma 3.16.
- **decidable**: as the function call chains are finite, the constraint generation terminates with a finite constraint set, which can be solved by *sol* in finite steps. Thus, the type inference system terminates.

□

4. A Representation of Types

As given in Definition 2.1, our types are defined as mappings from the power set $\mathscr{P}$ of the permission set $\mathbf{P}$ to the lattice $\mathscr{L}$ of security levels. Naively encoding types as permission sets leads to an exponential increase in time as well as size required to process and manipulate them due to the cardinality of the power set $\mathscr{P}$ growing as 2^n, where n is the number of permissions. Android, itself, has around 200 permissions[9] which can lead to performance decreases due to the time taken to process permissions. In practice, we can expect a security type to consist only of a few 'interesting' mappings for some combinations of permissions, and a 'default' mapping for the rest of the combinations of permissions, so the actual size of security types used in practice should be much smaller.

There are a number of choices one can make as to how to encode the 'default' mappings; for example, one could use propositional logic, first order logic, or some ad hoc data structures. Two main considerations are the space complexity of the chosen representation (in the average case) and the space/time complexity of the type-related operations on the representation. Here we choose a representation of types that builds on the concept of reduced ordered binary decision diagrams (ROBDDs) [20, 21]. ROBDD has been well-studied and has been applied successfully in areas such as model checking [22], so we think it is a good choice to build our representation on.

The principle idea behind choosing ROBDDs as the underlying data structure for representing types lies in the fact that any permission set can be encoded as *bit vectors* (after fixing an order on these permissions), namely, the presence or absence of permissions can be represented by a sequence of 1s and 0s. Taking the permission set $\mathbf{P} = \{p, q\}$ from Listing 2 as example, all the possible permission sets

[9] https://developer.android.com/reference/android/Manifest.permission.html

can be represented as the bit vectors $\{00, 01, 10, 11\}$. Moreover, as shown in Section 3.2, the permission traces are interpreted as sets of permission sets and can be considered as boolean logic formulae on permission sets, where $\oplus$ and $\ominus$ respectively denote the positive and negative literals, corresponding to the presence or absence of permissions. Thus the encoding of ROBDD enables us to infer our types directly on the permission traces via the logic connectives. The key is to generate a set of permission traces $\{\Lambda_i | i \in I\}$ for each variable such that they are full (*i.e.*, $\bigcup_{i\in I} \mathcal{I}(\Lambda_i) = \mathbf{P}$) and disjoint (*i.e.*, $\forall i, j \in I. i \neq j \Rightarrow \mathcal{I}(\Lambda_i) \cap \mathcal{I}(\Lambda_j) = \emptyset$). For example, we can infer for β in Listing 4 the type $t_\beta = \{\ominus p \mapsto H, \oplus p \mapsto (L \sqcup l_\beta^{\oplus p}) \sqcap H\}$ (no matter what $\mathbf{P}$ is), which is a variant of ROBDD, rather than $\{\emptyset \mapsto H, \{p\} \mapsto (L \sqcup l_\beta^{\{p\}}) \sqcap H, \{q\} \mapsto H, \{p,q\} \mapsto (L \sqcup l_\beta^{\{p,q\}}) \sqcap H\}$ ($\mathbf{P} = \{p, q\}$).

For simplicity, we shall refer to ROBDDs as just binary decision diagrams (BDDs). BDDs have some important and useful properties: (i) *canonicity*, that is to say for a fixed boolean variable ordering, each boolean function has a *canonical* (i.e. *unique*) representation (up to isomorphism) [20, 21]; and (ii) operations on BDDs are efficient. With proper ordering, the time complexity of operations usually depends linearly on the number of boolean variables [20]. Note that a security type with a two-element lattice ('High' and 'Low') is essentially a boolean function. So the worst case space complexity of the BDD representation (or any known representation of boolean functions, for that matter) is exponential [20].

4.1. Types as Binary Decision Diagrams

In this section, we give the definition of a binary decision diagram that fits our types, which is a modification[10] of the one in [20]. Note that in all our definitions, we assume that a *static global order* has been defined on the permissions. This serves two purposes: (i) the concept of representing permission sets as bit vectors would be meaningless otherwise; and (ii) it helps in the construction of binary decision diagrams.

To start with, let us consider a general function $f : \{0,1\}^n \to S$, where S is an arbitrary finite set. Clearly, we can represent it as a *truth table*. More specifically, we can represent the function $f(x_1, \ldots, x_n)$ as a 2^n-bit string of values, where each value is some $s \in S$. This string starts with the function value $f(0, \ldots, 0)$ and continues on with $f(0, \ldots, 0, 1), \ldots, f(1, \ldots, 1, 1)$. As an example, the truth table for the boolean function $g = x_1 \wedge x_2$ would be $g(0,0)g(0,1)g(1,0)g(1,1) = FFFT$.

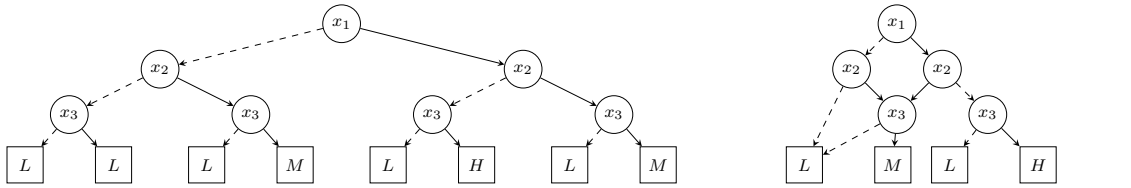
Definition 4.1 *Given an arbitrary finite set S and a natural number n, a* truth table *of order n is a string $\alpha \in S^*$ of length 2^n. A* bead *of order n is a truth table α of order n that is not square, that is, α can not be represented as $\beta\beta$ for any string $\beta \in S^*$ of length 2^{n-1}.*

Given a finite set S, there are $|S|$ beads of order 0; and $|S|^2 - |S|$ beads of order 1. In general, there are $|S|^{2^n} - |S|^{2^{n-1}}$ beads of order n. This is derived simply by removing elements on the diagonal of a square matrix of order 2^{n-1}, that is, by removing all combinations $\alpha\beta$ with $\alpha = \beta$, where α, β are truth tables of order $n-1$.

Our modified definition of a binary decision diagram to represent the function $f : \{0,1\}^n \to S$ is given below.

Definition 4.2 *A* binary decision diagram *(BDD) is a rooted, directed graph with vertex set V containing two types of vertices. A* nonterminal *vertex (*node*) m has as attributes an argument index $m.v \in \{1, 2, \cdots, n\}$; and two children $m.l, m.h \in V$ (referred to as* low *and* high *respectively). A* terminal *vertex (*sink *or* leaf*) m has as attribute a value $m.val \in S$, where S is an arbitrary finite set.*

[10] Our modification generalizes the set of outputs from binary to an arbitrary finite set.



(a) A complete binary decision diagram (b) A reduced binary decision diagram

Fig. 9. Examples of binary decision diagram

Furthermore, a strict ordering restriction is imposed on nonterminal vertices. For any nonterminal vertex m, if m.l is also nonterminal, then it must be the case that m.v < m.l.v. Similarly, if m.h is nonterminal, then it must be the case that m.v < m.h.v.

A BDD is *reduced* if no two nodes are allowed to have the same triple of values (v, l, h). Therefore, no node m in a reduced BDD is allowed to have $m.l = m.h$, which indicates that the output does not depend on this particular node m. Fig. 9 gives two examples of BDDs that represent the same function f_e given in Table 1, wherein $n = 3$ and $S = \{L, M, H\}$. Note that, in this figure and others henceforth, the dashed lines represent the '0' or low branch. We can see that the first example (*i.e.*, Fig. 9a) is not reduced, because (*i*) the leftmost node of x_3 has two equal children, and (*ii*) both the high children of the nodes x_2 are equal. While the second example (*i.e.*, Fig. 9b) is reduced.

Table 1
The truth table of function f_e

x_1	x_2	x_3	$f_e(x_1,x_2,x_3)$	x_1	x_2	x_3	$f_e(x_1,x_2,x_3)$
0	0	0	L	1	0	0	L
0	0	1	L	1	0	1	H
0	1	0	L	1	1	0	L
0	1	1	M	1	1	1	M

We now show that our modified definition of a BDD still holds the property of *canonicity*. To that end, we modify a proof for canonicity as described by Knuth [21].

Theorem 4.1 *Given an arbitrary finite set S, any function of the form $f : \{0,1\}^n \to S$ has a unique (up to isomorphism) reduced binary decision diagram, where n is the number of inputs.*

Proof. We argue that every truth table τ can be represented as a power of a unique bead. Let τ be a truth table of order n that is not a bead. Then, by Definition 4.1, it is the square of a truth table τ' of order $n-1$. By induction on the order of τ, we have that $\tau' = \beta^k$, where k is a power of 2, and β is a bead of order $\leqslant n-1$. Hence, we have that τ can be represented as $\tau = \beta^{2k}$. We call this β the *root* of τ (and τ'). Therefore, by the principle of induction, every truth table τ is a power of a unique bead.

A truth table τ of order $n > 0$ can be represented as $\tau_0\tau_1$, where τ_0 and τ_1 are truth tables of order $n-1$. By construction, τ represents the function $f(x_1, x_2, \ldots, x_n)$ if and only if τ_0 represents $f(0, x_2, \ldots, x_n)$ and τ_1 represents $f(1, x_2, \ldots, x_n)$. The functions $f(0, x_2, \ldots, x_n)$ and $f(1, x_2, \ldots, x_n)$ are called *subfunctions* of f; and τ_0 and τ_1 are called *subtables* of τ. The *beads* of such a function f are the subtables of its truth table that are beads.

Now we come to the crux of the argument: *the nodes of a binary decision diagram for a function f (as described above) are in bijection with the beads of f*. A function's truth tables of order $n + 1 - k$

(with $1 \leqslant k \leqslant n$) correspond to its subfunctions $f(c_1, \ldots, c_{k-1}, x_k, \ldots, x_n)$, where c_i denotes a binary value. Hence, the beads of order $n+1-k$ correspond to those subfunctions that depend on their first boolean variable x_k. Therefore, every such bead corresponds to a node $\textcircled{k}$ in the BDD. Let the truth table corresponding to this node be $\tau' = \tau'_0\tau'_1$, then its l and h attributes point respectively to the nodes that correspond to the roots of τ'_0 and τ'_1. □

Considering the function f_e in Table 1, the truth table τ_{f_e} for f_e is $LLLMLHLM$ and hence, all possible subtables of τ_{f_e} are $\{LLLMLHLM, LLLM, LHLM, LL, LM, LH, L, M, H\}$. Given this, the beads of f_e then are $\{LLLMLHLM, LLLM, LHLM, LM, LH, L, M, H\}$. Note that the subtable LL is not present since it is not a bead. Fig. 10 showcases the BDD formed by the beads of f_e as laid out in Theorem 4.1. Note the isomorphism between Fig. 9b and 10.

Let us return to our types. As mentioned above, permission sets can be encoded as bit vectors, where '0' represents the absence of a permission and '1' represents the presence of a permission. Moreover, operations on sets such as $\cup, \cap, \setminus$, etc. can easily be transcribed into operations on bit vectors. Accordingly, we redefine our types as folllows:

Definition 4.3 *A* base security type *t is a mapping from the set of all possible bit vectors representing permission sets to the lattice of security levels, i.e. $t : \mathscr{P} \to \mathscr{L}$. The set of base types is denoted with $\mathcal{T}$.*

Using Theorem 4.1, the new definition of a type, and the definition of a binary decision diagram, we define the representation of types as follows:

Definition 4.4 *Let t be a base type. The* representation *of t, denoted by $\mathcal{R}(t)$, is a reduced ordered binary decision diagram, where each nonterminal node m represents (the position in the static order of) a permission $p \in \{1, 2, \cdots |\mathbf{P}|\}$; the* low *and high branches of m (*i.e.*, $m.l$ and $m.h$) represent the absence and the presence of p, respectively; and each terminal (or sink) node represents an output security level $l \in \mathscr{L}$.*

In order to evaluate a permission set for the *representation* of a given type, one can simply move down the diagram by choosing the branch that corresponds to the 0-1 value in the bit vector for that particular permission. If a node for a permission p does not exist, then one can simply skip that permission as the output security level does not depend on p's value. Formally, the evaluation of the *representation* r along (a bit vector representing) a permission set P can be defined as

$$r(P) = \begin{cases} r.val & r \text{ is sink} \\ r.l(P) & P[r.v] = 0 \\ r.h(P) & P[r.v] = 1 \end{cases}$$

Clearly, it is easy to get that our representation is correct, as shown in Lemma 4.1.

Lemma 4.1 *Let t be a type. Then for any permission set P, we have* (1) *$t(P) = \mathcal{R}(t)(P)$; and* (2)*$\mathcal{R}(t)(P) = \mathcal{R}(t)(P \cup \{p\}) = \mathcal{R}(t)(P \setminus \{p\})$ for any permission $p \notin \mathcal{R}(t)$.*

To illustrate, let us consider the type t that maps the sets of permissions $\{p_1\}$ and $\{p_2, p_3\}$ to H, and any other subset of $\{p_1, p_2, p_3\}$ to L. Fig. 11 show the BDD of t, where the bit vector is represented as $p_1p_2p_3$. Following the paths to H in the diagram gives us the following bit vectors: $\{100, 011\}$. On the other hand, following the paths to L in the diagram gives us the following bit vectors: $\{00x, 010, 101, 11x\}$, where 'x' represents permissions whose values do not affect the output security level. These values encompass all elements of the powerset $\mathscr{P}$ and accurately represents the mapping in type t as required. Moreover, the permissions whose values do not affect the output security level can be

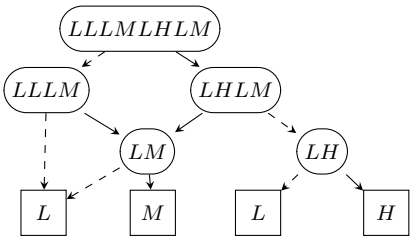


Fig. 10. BDD formed by beads of f_e

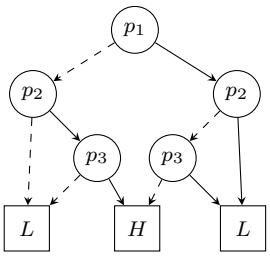


Fig. 11. Representation of type t

removed in BDD, which yields the small numbers (*i.e.*, 5 and 6, resp.) of nodes and of paths to the sinks. In comparison, a binary decision tree would have $2^3 - 1$ nodes and 2^3 different paths to sinks.

An example in practice is the type $t = \{\emptyset \mapsto L, \{p\} \mapsto L, \{q\} \mapsto H, \{p, q\} \mapsto l_1\}$ used in Listing 2. Assuming the order is q, p, the paths to L in the corresponding BDD could be merged, resulting in fewer nodes and paths. Moreover, many apps or services require more than two permissions, such as the facial recognition service requiring the permissions for camera and storage (2 permissions for the storage group) and the video calling service requiring the permissions[11] for Wi-Fi, camera, audio, and storage. These services terminate immediately if any permission is not granted[12]. So the type for these services maps the set containing all the required permissions to a non-low security level depending on the services and the other sets to L, whose representation is given in Figure 12, where the dashed rectangle denotes a group of permissions, for example, the permission group for storage may contain *READ_EXTERNAL_STORAGE* and *WRITE_EXTERNAL_STORAGE*. Figure 12 shows that the representation requires $|P|$ nodes and $|P| + 1$ paths to sinks, rather than $2^{|P|} - 1$ nodes and $2^{|P|}$ paths (even worse if all the permissions are considered), where P is the set of all required permissions. In addition, some services take different actions depending the granted permissions. For example, the location service determines the current location from GPS sensor (requiring the permissions for location), networks (requiring the permissions for Wi-Fi), or the last known location (requiring the permissions for storage). The type for the location service would map the set containing any group of required permissions to a non-low security level for the location and the other sets to L, whose representation is given in Figure 13, wherein a group is not granted if any one of its permissions is not granted. Similarly, the corresponding BDD has $|P|$ nodes and at least $|P| + 1$ paths to sinks. So we believe the representation will be efficient in practice.

4.2. Operations on types

Unlike BDDs in [20], the outputs of our BDDs are a multi-value lattice. It is quite difficult to come up with a general framework describing operations on types as directly working on the Boolean functions (*i.e.*, BDDs on two-value lattice). Therefore, we directly provide algorithms for operations on types mentioned in Section 2.4, via reasoning on the structure of the diagram. Here we focus on the operations only for our types, namely, *projection*, *promotion*, *demotion*, and *merging*. While the others, namely, the intersection $s \sqcap t$, the union $s \sqcup t$, and the subtyping $s \leqslant t$, could be obtained directly via the APPLY function as described in [20, 21].

[11] https://developers.connectycube.com/android/videocalling.

[12] Such a multiple permission checking refers to https://codingmitra.com/multiple-permission-run-time-in-android/.

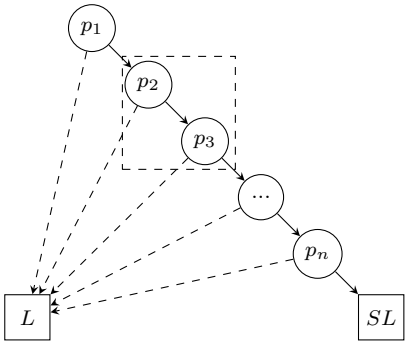


Fig. 12. Representation of type for video calling

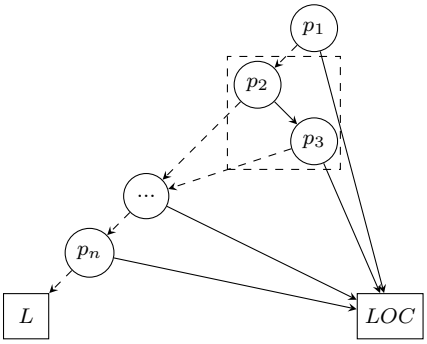


Fig. 13. Representation of type for location

Firstly, with respect to the definition of type presentations, the properties for *promotion*, *demotion*, *projection*, and *merging* defined in Definition 2.5, 2.6, and 2.7 still hold.

Lemma 4.2 *Given a base type t, a permission p, and a permission set $P \in \mathscr{P}$, then* (1) $\mathcal{R}(t \uparrow_p)(P) = \mathcal{R}(t)(P \cup \{p\})$ *and* (2) $\mathcal{R}(t \downarrow_p)(P) = \mathcal{R}(t)(P \setminus \{p\})$.

Lemma 4.3 *Given a type t and a permission set $P \in \mathscr{P}$, then for any permission set $Q \in \mathscr{P}$,* $\mathcal{R}(\pi_P(t))(Q) = \mathcal{R}(t)(P)$.

Lemma 4.4 *Given two types t_1, t_2 and a permission p, then for any permission set $P \in \mathscr{P}$,*

$$\mathcal{R}(t_1 \triangleright_p t_2)(P) = \begin{cases} \mathcal{R}(t_1)(P) & p \in P \\ \mathcal{R}(t_2)(P) & p \notin P \end{cases}$$

According to Lemma 4.3, we argue that we can simply represent the *projection* as a binary decision diagram with a single vertex, being a sink node.

Lemma 4.5 *Given a base type t and a permission set $P \in \mathscr{P}$, the* projection *of t on P can be represented as a constant terminal vertex m with $m.val = t(P)$.*

The case for *promotion* and *demotion* is slightly more complicated. Recalling the Definition 2.5, the *promotion* of a base type t with respect to a permission p involves removing all branches where $p = 0$ and redirecting them to where $p = 1$. In other words, p is present in all permission sets regardless. We formalize this intuitive explanation into an algorithm as described in Fig. 14a. Likewise, the algorithm for *demotion* is given in Fig. 14b, where all branches with $p = 1$ are removed and redirected the one with $p = 0$ instead.

The function INDEX(p) is a helper function that outputs the position of the permission p in the static global order; and REDUCE( ) is a procedure which restores the reduced property of a binary decision diagram [20, 21]. Hence, given a base type t, and a permission p, the *promotion* operation is performed *in situ* by calling PROMOTION$(\mathcal{R}(t), p)$.

Fig. 15 shows the promotion of the type t in Fig. 9 with respect to p_2, where the dashed lines in red denote the modifications needed by the promotion.

The correctness of the algorithms for *promotion* and *demotion* is given in Lemma 4.6.

Lemma 4.6 *Given a base type t and a permission p, then* (a) PROMOTION$(\mathcal{R}(t), p) = \mathcal{R}(t \uparrow_p)$ *and* (b) DEMOTION$(\mathcal{R}(t), p) = \mathcal{R}(t \downarrow_p)$.

One can see that the number of operations required for promoting a type depends on the number of nodes in $\mathcal{R}(t)$, or more accurately, the number of nodes m such that $m.v \leqslant$ INDEX(p). Note that since

```
procedure PROMOTION-RECURSE(r, p)
    if r is a sink then
        return
    else if r.v > INDEX(p) then
        return
    else if r.v = INDEX(p) then
        r.l ← r.h
        return
    end if          ▷ when r.v < INDEX(p)
    PROMOTION-RECURSE(r.l, p)
    PROMOTION-RECURSE(r.h, p)
end procedure
procedure PROMOTION(r, p)
    REDUCE(PROMOTION-
RECURSE(r, p))
    return
end procedure
```

(a) Pseudo-code for promotion

```
procedure DEMOTION-RECURSE(r, p)
    if r is a sink then
        return
    else if r.v > INDEX(p) then
        return
    else if r.v = INDEX(p) then
        r.h ← r.l
        return
    end if          ▷ when r.v < INDEX(p)
    DEMOTION-RECURSE(r.l, p)
    DEMOTION-RECURSE(r.h, p)
end procedure
procedure DEMOTION(r, p)
    REDUCE(DEMOTION-
RECURSE(r, p))
    return
end procedure
```

(b) Pseudo-code for demotion

Fig. 14. Algorithms for promotion and demotion

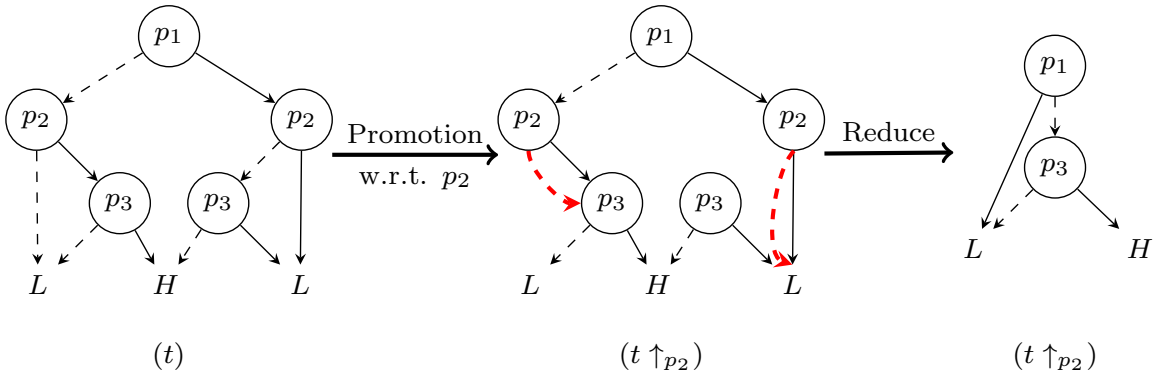


Fig. 15. Example for promotion (the dashed lines in red denote the modifications needed by the promotion)

the procedure is performed *in situ*, we do not actually introduce any new nodes, and in fact our output will have fewer nodes (than initially) after reduction. Also note that the time complexity of REDUCE( ), as defined in [21], is $\mathcal{O}(N+n)$, where N is the size of the input BDD, and n is the number of boolean variables. Given the above, we can say that the time complexity of PROMOTION is $\mathcal{O}(N+n+N) = \mathcal{O}(2N+n)$. A similar argument can be constructed for the case of DEMOTION as well.

Finally, let us describe how the *merge* operation could be implemented. Recall the definition: the *merging* of two types t_1 and t_2 along the permission p is:

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

```
function MERGE-RECURSE(r1, r2, p)
    if r1, r2 are sink then
        return (p, r2, r1)
    else if r1 is sink or r1.v > r2.v then
        if r2.v > INDEX(p) then
            return (p, r2, r1)
        else if r2.v = INDEX(p) then
            return (r2.v, r2.l, r1)
        else                                              ▷ r2.v < INDEX(p)
            return (r2.v, MERGE-RECURSE(r1, r2.l, p), MERGE-RECURSE(r1, r2.h, p))
        end if
    else if r2 is sink or r1.v < r2.v then
        if r1.v > INDEX(p) then
            return (p, r2, r1)
        else if r1.v = INDEX(p) then
            return (r1.v, r2, r1.h)
        else                                              ▷ r1.v < INDEX(p)
            return (r1.v, MERGE-RECURSE(r1.l, r2, p), MERGE-RECURSE(r1.h, r2, p))
        end if
    else                                                  ▷ r1.v = r2.v
        if r1.v > INDEX(p) then
            return (p, r2, r1)
        else if r1.v = INDEX(p) then                      ▷ Base Case
            return (r1.v, r2.l, r1.h)
        else                                              ▷ r1.v < INDEX(p)
            return (r1.v, MERGE-RECURSE(r1.l, r2.l, p), MERGE-RECURSE(r1.h, r2.h, p))
        end if
    end if
end function
function MERGE(r1, r2, p)
    return REDUCE(MERGE-RECURSE(r1, r2, p))
end function
```

Fig. 16. Algorithm for merge

That is, the node of p in the merged diagram is formed by the low branch of the node of p in $\mathcal{R}(t_2)$ and the high branch of the node of p in $\mathcal{R}(t_1)$. Due to the reducing, the permission p may not occur in $\mathcal{R}(t_1)$ or $\mathcal{R}(t_2)$. So we divide the problem into two cases: (i) p occurs in neither $\mathcal{R}(t_1)$ nor $\mathcal{R}(t_2)$ (*e.g.*, both $\mathcal{R}(t_1)$ and $\mathcal{R}(t_2)$ are sink or both $\mathcal{R}(t_1).v$ and $\mathcal{R}(t_2).v$ are greater than INDEX(p)); and (ii) p occurs in at least one of $\mathcal{R}(t_1)$ and $\mathcal{R}(t_2)$. The first case is intuitively easy to reason for: we simply create a node for p and let the low branch and the high branch point to (the roots of) $\mathcal{R}(t_2)$ and $\mathcal{R}(t_1)$, respectively.

The second case is slightly more complex. Let us assume p occurs in both $\mathcal{R}(t_1)$ and $\mathcal{R}(t_2)$ and further divide it into two subcases: (i) both representations are rooted by p; and (ii) one of the representations is not rooted by p. For the first subcase, we simply redirect the low branch (i.e. l or '0' branch) of (the root of) $\mathcal{R}(t_1)$ to the low branch of (the root of) $\mathcal{R}(t_2)$. And for the second subcase, we recursively traverse $\mathcal{R}(t_1)$ and $\mathcal{R}(t_2)$ until we reach a point where both the representations are rooted by p, and then merge

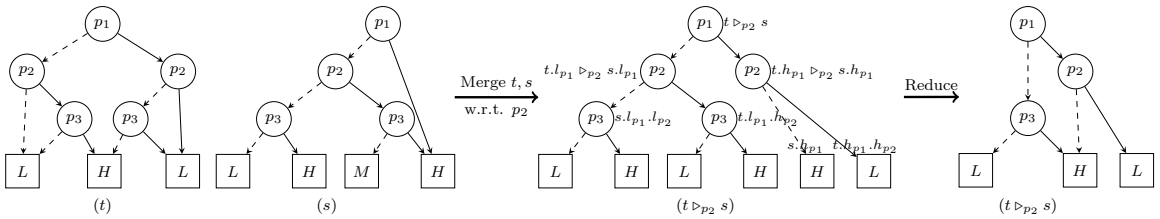


Fig. 17. Example for merge

as per the first subcase at that point. Moreover, if p does not occur in $\mathcal{R}(t_1)$ (resp. $\mathcal{R}(t_2)$), then we can imagine that there is a dummy node for p, where both its low branch and high branch are the smallest node in $\mathcal{R}(t_1)$ (resp. $\mathcal{R}(t_2)$) that is greater than $\text{INDEX}(p)$. The algorithm is a slightly modified version of the APPLY function as described in [20, 21]. The intuition behind our change lies in, besides the sinks, all nodes with index no smaller than p are treated as a terminal (i.e. as a base case). Fig. 16 gives the details for the MERGE( ) algorithm.

An example of the merge of the type t in Fig. 9 and another type s along p_2 is given in Fig. 17. The following lemma shows the correctness of the MERGE( ) algorithm.

Lemma 4.7 *Given two types t_1, t_2 and a permission p, then* $\text{MERGE}(\mathcal{R}(t_1),\ \mathcal{R}(t_2),\ p) = \mathcal{R}(t_1 \triangleright_p t_2)$.

Note that unlike PROMOTION-RECURSE( ) and DEMOTION-RECURSE( ), MERGE-RECURSE( ) constructs an entire new diagram such that the p nodes of both diagrams have been merged. Hence, the space complexity of this operation drastically increases. The time complexity of MERGE-RECURSE( ) has order of $|\mathcal{R}(t_1)| \cdot |\mathcal{R}(t_2)|$ as the *merge* operation essentially creates ordered pairs of nodes of the two diagrams. Moreover, we can represent the above as a matrix of values. Then we can reduce the time complexity of MERGE-RECURSE( ) to $|\mathcal{R}(t_1 \triangleright_p t_2)|$, by only filling entries in the matrix that are reachable by $t_1 \triangleright_p t_2$.

5. Related work

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

We have extensively discussed the work by Banerjee and Naumann [11] and highlighted the major differences between our work and theirs in Section 1.

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 [23–41]. Hunt and Sands [41] 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 the system is simpler than ours. Mantel et. al. [35] 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 [40] 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 [24] and dynamic information flow control [34]. Our permission context can be seen as a special instance of execution context, however, our intended applications and settings are different from [24, 34], and issues such as parameter laundering does not occur in their setting. Lourenço and Caires [31] 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, 42–46]). SCandroid [2, 46] 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 [42] 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 [47] and does not require extra changes on existing Android application supply chain systems. Cassandra [44] performs the security analysis of apps on a server, which employs the proof-carrying code paradigm such that the server's analysis result can be validated on the client. H. Gunadi [45] presented a type system for DEX bytecode and prove the soundness of the type system with respect to a notion of non-interference. Both [44] and [45] do not consider permission-dependent. ComDroid [16] detects communication vulnerabilities in Android applications from the perspectives of Intent senders and Intent recipients. Weir [43] is a practical DIFC system for Android, allowing data owner applications to set secrecy policies and control the export of their data to the network. And a number of security analysis tools, such as TaintDroid [48], DroidSafe [49] and TaintART [50], perform a data flow analysis on Android apps to track information-flow. But there is no soundness result for these analyses.

A few inference algorithms [51, 52] have been proposed to infer dependent labels. Lifty [51] employed refinement types to encode information flow security, and proposed an inference algorithm based on the inference engine of liquid types [53]. While this is a neat solution for a two-level security lattice, the encoding does not apply to applications that require multiple security labels. Li and Zhang [52] proposed a general framework for designing and checking label inference algorithms for information flow analysis with dependent security labels. Based on the framework, novel inference algorithms that are both sound and complete are developed. The predicated constraint they proposed can express predicate on program state, and thus is a general version of ours. Limited to our setting, their one-shot algorithm is equivalent to ours. However, although efficient, the algorithms, except the one-shot one, may be over-conservative. For example, the early-accept algorithm and hybrid algorithm would infer $\hat{H}$ for the motivation example *getInfo* listed in 2, which is clearly imprecise. Moreover, Li and Zhang did not provide a principal solution for their algorithms. This indicates that the algorithms, except the one-shot one, may not be complete in the sense that every possible solution can be obtained by the algorithm, such as the precise type for *getInfo* can't be obtained from $\hat{H}$ via subsumption or instantiation.

6. 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, as well as a new way to represent our security types as reduced ordered binary decision diagrams.

We next discuss briefly several directions for future work.

The immediate one is to extend our system to richer programming languages, including object-oriented features (like [54]), exceptions (like [55]), etc. Another extension is to incorporate the efficient type representation in the inference algorithm. A proposed solution is to encode our permission guarded constraints as a special BDD, where the outputs are the constraints without guards (*i.e.*, constraints on the output labels).

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. [55–57] 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 [44] or Gunadi's type system [45]. We envision our implementation can piggyback on, say, Cassandra system to improve the coverage of typable applications.

References

[1] W. Enck, M. Ongtang and P. McDaniel, Understanding Android Security, *IEEE Security and Privacy* **7**(1) (2009), 50–57.
[2] A.P. Fuchs, A. Chaudhuri and J. Foster, SCanDroid : Automated Security Certification of Android Applications, Technical Report, CS-TR-4991, University of Maryland, 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.* **49**(6) (2014), 259–269.

[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*, CCS '14, ACM, New York, NY, USA, 2014, pp. 1329–1341. ISBN 978-1-4503-2957-6.
[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*, ICSE '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 280–291. ISBN 978-1-4799-1934-5.
[6] D.E. Denning, A Lattice Model of Secure Information Flow, *Communications of the ACM* **19**(5) (1976), 236–243.
[7] D.E. Denning and P.J. Denning, Certification of Programs for Secure Information Flow, *Communications of the ACM* **20**(7) (1977), 504–513.
[8] D. Volpano, C. Irvine and G. Smith, A Sound Type System for Secure Flow Analysis, *Journal of Computer Security* **4**(2–3) (1996), 167–187.
[9] A. Sabelfeld and A.C. Myers, Language-based Information-flow Security, *IEEE Journal on Selected Areas in Communications* **21**(1) (2003), 5–19.
[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* **15**(2) (2005), 131–177.
[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, in: *31st IEEE Computer Security Foundations Symposium, CSF*, IEEE Computer Society, Oxford, United Kingdom, 2018, pp. 218–232.
[14] Android, Requesting Permissions at Run Time. https://developer.android.com/training/permissions/requesting.html.
[15] A. Developers, Binder, 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*, MobiSys '11, New York, NY, USA, 2011, pp. 239–252.
[17] A. Developers, PermissionChecker | Android Developers, 2017, online, accessed on 07-July-2017.
[18] B.C. Pierce, *Types and programming languages*, MIT Press, 2002. ISBN 978-0-262-16209-8.
[19] A.O.S. Project, Manifest.permission, Android Open Source Project, [n. d.]. https://developer.android.com/reference/android/Manifest.permission.html.
[20] R.E. Bryant, Graph-Based Algorithms for Boolean Function Manipulation, *IEEE Transactions on Computers* **35**(8) (1986), 677–691. doi:10.1109/TC.1986.1676819.
[21] D.E. Knuth, *The Art of Computer Programming*, Vol. 4. Fascicle 1, 1st edn, Addison Wesley Longman Publishing Co., Inc., United States, 2009.
[22] E.M. Clarke, T.A. Henzinger, H. Veith and R. Bloem (eds), *Handbook of Model Checking*, Springer, 2018. ISBN 978-3-319-10574-1. doi:10.1007/978-3-319-10575-8.
[23] 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)*, 2016, pp. 417–431.
[24] N. Polikarpova, J. Yang, S. Itzhaky and A. Solar-Lezama, Type-Driven Repair for Information Flow Security, *CoRR* **abs/1607.03445** (2016). http://arxiv.org/abs/1607.03445.
[25] T. Murray, R. Sison, E. Pierzchalski and C. Rizkallah, A Dependent Security Type System for Concurrent Imperative Programs, *Archive of Formal Proofs* (2016), http://isa-afp.org/entries/Dependent_SIFUM_Type_Systems.shtml, Formal proof development.
[26] T. Murray, R. Sison, E. Pierzchalski and C. Rizkallah, Compositional Security-Preserving Refinement for Concurrent Imperative Programs, *Archive of Formal Proofs* (2016), http://isa-afp.org/entries/Dependent_SIFUM_Refinement.shtml, Formal proof development.
[27] 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.
[28] X. Li, F. Nielson and H. Riis Nielson, Future-dependent Flow Policies with Prophetic Variables, in: *the 2016 ACM Workshop*, ACM Press, New York, NY, USA, 2016, pp. 29–42.
[29] 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*, ACM Press, New York, New York, USA, 2015, pp. 503–516.
[30] X. Li, F. Nielson, H.R. Nielson and X. Feng, Disjunctive Information Flow for Communicating Processes., *TGC* (2015).
[31] L. Lourenço and L. Caires, Dependent Information Flow Types, in: *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, New York, NY, USA, 2015, pp. 317–328.

[32] 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*, Lecture Notes in Computer Science, Vol. 8358, Springer, 2014, pp. 180–198.
[33] N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan and J. Yang, Secure distributed programming with value-dependent types, *Journal of Functional Programming* **23**(4) (2013), 402–451.
[34] 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. doi:10.1145/2103656.2103669.
[35] 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. doi:10.1109/CSF.2011.22.
[36] 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.
[37] 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)*, Lecture Notes in Computer Science, Vol. 6012, Springer, 2010, pp. 529–549.
[38] L. Zheng and A.C. Myers, Dynamic security labels and static information flow control., *International Journal of Information Security* **6**(2–3) (2007), 67–84.
[39] S. Tse and S. Zdancewic, Run-time principals in information-flow type systems, *ACM Trans. Program. Lang. Syst.* **30**(1) (2007).
[40] P. Li and D. Zhang, Towards a Flow- and Path-Sensitive Information Flow Analysis, in: *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, 2017, pp. 53–67.
[41] 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*, POPL '06, ACM, New York, NY, USA, 2006, pp. 79–90. ISBN 1-59593-027-2.
[42] 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*, CCS '14, ACM, New York, NY, USA, 2014, pp. 1092–1104. ISBN 978-1-4503-2957-6.
[43] A. Nadkarni, B. Andow, W. Enck and S. Jha, Practical DIFC Enforcement on Android, *USENIX Security Symposium* (2016).
[44] 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*, SPSM '14, New York, NY, USA, 2014, pp. 93–104.
[45] 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)*, ICECCS '15, Washington, DC, USA, 2015, pp. 202–205.
[46] A. Chaudhuri, Language-based Security on Android, in: *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, New York, NY, USA, 2009, pp. 1–7.
[47] G.C. Necula and P. Lee, Proof-Carrying Code, Technical Report, School of Computer Science, Carnegie Mellon University, 1996, CMU-CS-96-165.
[48] W. Enck, P. Gilbert, B. Chun, L.P. Cox, J. Jung, P.D. McDaniel and A. Sheth, TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones, in: *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2010)*, 2010, pp. 393–407.
[49] M.I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen and M. Rinard, Information-Flow Analysis of Android Applications in DroidSafe, in: *22nd Annual Network and Distributed System Security Symposium (NDSS 2015)*, 2015.
[50] M. Sun, T. Wei and J.C.S. Lui, TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime, in: *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, CCS'16, 2016.
[51] N. Polikarpova, J. Yang, S. Itzhaky, T. Hance and A. Solar-Lezama, Enforcing Information Flow Policies with Type-Targeted Program Synthesis, *arXiv preprint arXiv:1607.03445v2* (2018).
[52] P. Li and D. Zhang, A derivation framework for dependent security label inference, *Proc. ACM Program. Lang.* **2**(OOPSLA) (2018), 115:1–115:26.
[53] P.M. Rondon, M. Kawaguchi and R. Jhala, Liquid types, in: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, R. Gupta and S.P. Amarasinghe, eds, ACM, 2008, pp. 159–169. doi:10.1145/1375581.1375602.

[54] 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., Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 84–99. ISBN 978-3-540-27864-1.

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

[56] G. Barthe and T. Rezk, Non-interference for a JVM-like language., *TLDI* (2005), 103–112.

[57] G. Barthe, D. Pichardie and T. Rezk, A Certified Lightweight Non-interference Java Bytecode Verifier, in: *Proceedings of the 16th European Symposium on Programming*, ESOP'07, Berlin, Heidelberg, 2007, pp. 125–140.

[58] H. Chen, A. Tiu, Z. Xu and Y. Liu, A Permission-Dependent Type System for Secure Information Flow Analysis, *CoRR* **abs/1709.09623** (2017). http://arxiv.org/abs/1709.09623.

[59] G. Winskel, *The Formal Semantics of Programming Languages: An Introduction*, MIT Press, Cambridge, MA, USA, 1993. ISBN 0-262-23169-7.

[60] J. van Rest, D. Boonstra, M. Everts, M. van Rijn and R. van Paassen, *Designing Privacy-by-Design*, in: *Privacy Technologies and Policy: First Annual Privacy Forum, APF 2012, Limassol, Cyprus, October 10-11, 2012, Revised Selected Papers*, B. Preneel and D. Ikonomou, eds, Berlin, Heidelberg, 2014, pp. 55–72.

[61] D.F.C. Brewer and M.J. Nash, The Chinese Wall security policy, in: *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, 1989, pp. 206–214.

[62] Y. Takata and H. Seki, *Automatic Generation of History-Based Access Control from Information Flow Specification*, in: *Automated Technology for Verification and Analysis: 8th International Symposium, ATVA 2010, Singapore, September 21-24, 2010. Proceedings*, A. Bouajjani and W.-N. Chin, eds, Berlin, Heidelberg, 2010, pp. 259–275.

[63] D. Liu, Bytecode Verification for Enhanced JVM Access Control, in: *Proceedings of the The Second International Conference on Availability, Reliability and Security*, ARES '07, Washington, DC, USA, 2007, pp. 162–172.

[64] A. Banerjee and D.A. Naumann, A Simple Semantics and Static Analysis for Stack Inspection (2013). doi:10.4204/EPTCS.129.17.

[65] M. Pistoia, A. Banerjee and D.A. Naumann, Beyond Stack Inspection: A Unified Access-Control and Information-Flow Security Model, in: *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, Washington, DC, USA, 2007, pp. 149–163.

[66] T.F. Lunt, Aggregation and inference: facts and fallacies, in: *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, 1989, pp. 102–109.

[67] M. Kennedy and R. Sulaiman, Following the Wi-Fi breadcrumbs: Network based mobile application privacy threats, in: *Electrical Engineering and Informatics (ICEEI), 2015 International Conference on*, 2015, pp. 265–270.

[68] C. Marforio, H. Ritzdorf, A. Francillon and S. Capkun, Analysis of the Communication Between Colluding Applications on Modern Smartphones, in: *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, New York, NY, USA, 2012, pp. 51–60.

[69] A. Nanevski, A. Banerjee and D. Garg, Dependent Type Theory for Verification of Information Flow and Access Control Policies, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **35**(2) (2013), 6:1–6:41.

[70] A.C. Myers and B. Liskov, A Decentralized Model for Information Flow Control, *SIGOPS Oper. Syst. Rev.* **31**(5) (1997), 129–142.

[71] O. Arden, M.D. George, J. Liu, K. Vikram, A. Askarov and A.C. Myers, Sharing Mobile Code Securely with Information Flow Control, *IEEE Symposium on Security and Privacy* (2012), 191–205.

[72] A.C. Myers, JFlow: Practical Mostly-static Information Flow Control, in: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, New York, NY, USA, 1999, pp. 228–241.

[73] Z. Fang, W. Han and Y. Li, Permission based Android security: Issues and countermeasures, *Computers & Security* **43**(C) (2014), 205–218.

[74] C. Marforio, A. Francillon and S. Capkun, *Application collusion attack on the permission-based security model and its implications for modern smartphone systems*, 2011.

[75] 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*, ICSE '15, Piscataway, NJ, USA, 2015, pp. 280–291. ISBN 978-1-4799-1934-5.

[76] A.P. Felt, E. Chin, S. Hanna, D. Song and D. Wagner, Android permissions demystified (2016), 1–11.

[77] A. Sabelfeld and D. Sands, Declassification: Dimensions and principles, *Journal of Computer Security* **17**(5) (2009), 517–548.

[78] W.J. Bowman and A. Ahmed, Noninterference for Free, in: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, ACM, New York, NY, USA, 2015, pp. 101–113. ISBN 978-1-4503-3669-7.

[79] Y. Shao, Q.A. Chen, Z.M. Mao, J. Ott and Z. Qian, Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework, in: *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, The Internet Society, 2016.

[80] D.E. Denning, A Lattice Model of Secure Information Flow, *Communications of the ACM* **19**(5) (1976), 236–243. doi:10.1145/360051.360056.
[81] B.A. Davey and H.A. Priestley, *Introduction to Lattices and Order, Second Edition*, Cambridge University Press, Cambridge, United Kingdom, 2002. ISBN 978-0-521-78451-1. doi:10.1017/CBO9780511809088.
[82] C. Smith, Google isn't fixing a serious Android security flaw for months. http://bgr.com/2017/05/09/android-permissions-security-flaw-android-o/.
[83] Android, Manifest.permission. https://developer.android.com/reference/android/Manifest.permission.html.
[84] E. Duan, DressCode and its potential impact for enterprise, September 2016. http://blog.trendmicro.com/trendlabs-security-intelligence/dresscode-potential-impact-enterprises/.
[85] M. Kumar, Beware! New Android Malware Infected 2 Million Google Play Store Users. http://thehackernews.com/2017/04/android-malware-playstore.html.

Appendix A. Proofs for Soundness

Proof for **Lemma 2.1**:

Proof. **Reflectivity** $\forall P, t(P) \leqslant_{\mathscr{L}} t(P)$ therefore $t \leqslant_{\mathcal{T}} t$.
Antisymmetry $t \leqslant_{\mathcal{T}} s \wedge s \leqslant_{\mathcal{T}} t \iff \forall P, t(P) \leqslant_{\mathscr{L}} s(P) \wedge s(P) \leqslant_{\mathscr{L}} t(P)$ therefore $\forall P, t(P) = s(P)$, which means that $s = t$.
Transitivity if $r \leqslant_{\mathcal{T}} s$ and $s \leqslant_{\mathcal{T}} t$, $\forall P, r(P) \leqslant_{\mathscr{L}} s(P)$ and $s(P) \leqslant_{\mathscr{L}} t(P)$ therefore $r(P) \leqslant_{\mathscr{L}} t(P)$, which means that $r \leqslant_{\mathcal{T}} t$.
□

Lemma A.1 *Given two base types s and t, it follows that*

(a) $s \leqslant_{\mathcal{T}} s \sqcup t$ *and* $t \leqslant_{\mathcal{T}} s \sqcup t$.
(b) $s \sqcap t \leqslant_{\mathcal{T}} s$ *and* $s \sqcap t \leqslant_{\mathcal{T}} t$.

Proof. Immediately from Definition 2.1. □

Proof for **Lemma 2.2**:

Proof. $\forall s, t \in \mathscr{P}$, according to Lemma A.1, $s \sqcup t$ is their upper bound. Suppose r is another upper bound of them, i.e., $s \leqslant_{\mathcal{T}} r$ and $t \leqslant_{\mathcal{T}} r$, which means $\forall P \in \mathscr{P}, (s \sqcup t)(P) = s(P) \sqcup t(P) \leqslant_{\mathscr{L}} r(P)$, so $s \sqcup t \leqslant r$. Therefore $s \sqcup t$ is the least upper bound of $\{s, t\}$. Similarly, $s \sqcap t$ is s and t's greatest lower bound. This makes $(\mathcal{T}, \leqslant_{\mathcal{T}})$ a lattice. □

Proof for **Lemma 2.3**:

Proof. If $p \in P$, $P \cup \{p\} = P$, therefore $(t \uparrow_p)(P) = t(P \cup \{p\}) = t(P)$.
If $p \notin P$, $P \setminus \{p\} = P$, therefore $(t \downarrow_p)(P) = t(P \setminus \{p\}) = t(P)$. □

Lemma A.2 *If* $s \leqslant t$, *then* $s \uparrow_p \leqslant t \uparrow_p$ *and* $s \downarrow_p \leqslant t \downarrow_p$.

Proof. For $P \in \mathbf{P}$, since $s \leqslant t$, $s(P \cup \{p\}) \leqslant t(P \cup \{p\})$ and $s(P \setminus \{p\}) \leqslant t(P \setminus \{p\})$, according to Definition 2.1 and Definition 2.5, the conclusion follows. □

Proof for **Lemma 2.4**:

Proof. **Reflexivity** Obviously $\eta =^{lo}_{\Gamma} \eta$.
Symmetry Since $\forall x \in dom(\Gamma).(\Gamma(x) \leqslant \hat{l}_O \Rightarrow \eta' =^{lo}_{\Gamma} \eta)$.
Transitivity If $\eta_1 =^{lo}_{\Gamma} \eta_2$ and $\eta_2 =^{lo}_{\Gamma} \eta_3$, for a given $x \in dom(\Gamma)$, when $\Gamma(x) \leqslant \hat{l_O}$, we have $\eta_1(x) = \eta_2(x)$ and $\eta_2(x) = \eta_3(x)$.

(1) If $\eta_1(x) \neq \bot$, then $\eta_2(x) \neq \bot$ by the first equation, which in return requires $\eta_3(x) \neq \bot$ by the second equation; by transitivity $\eta_1(x) = \eta_3(x) (\neq \bot)$.
(2) If $\eta_1(x) = \bot$, the first equation requires that $\eta_2(x) = \bot$, which makes $\eta_3(x) = \bot$, therefore both $\eta_1(x) = \eta_3(x) (= \bot)$.

Therefore $\eta_1(x) = \eta_3(x)$.

□

Lemma A.3 *If $\eta =^{lo}_{\Gamma} \eta'$, then for each $P \in \mathscr{P}$, $\eta =^{lo}_{\pi_P(\Gamma)} \eta'$.*

Proof. $\forall x \in dom(\Gamma)$, we need to prove that when $\pi_P(\Gamma)(x) \leqslant l_O$ then $\eta(x) = \eta'(x)$. But $\pi_P(\Gamma)(x) = \pi_P(\Gamma(x)) = t(P)$, from the definition of $\eta(x) = \eta'(x)$, the conclusion holds. □

Proof for **Lemma 2.5**:

Proof. We first note that $dom(\pi_P(\Gamma)) = dom(\pi_P(\Gamma \uparrow_p))$ since both promotion and projection do not change the domain of a typing environment. We then show below that $\pi_P(\Gamma) = \pi_P(\Gamma \uparrow_p)$, from which the lemma follows immediately. Given any $x \in dom(\pi_P(\Gamma \uparrow_p))$, for any $Q \in \mathscr{P}$, we have

$$\begin{array}{ll} & \pi_P(\Gamma \uparrow_p)(x)(Q) \\ = & (\pi_P(\Gamma \uparrow_p (x)))(Q) \quad \text{by Def.2.8} \\ = & (\Gamma \uparrow_p (x))(P) \quad \text{by Def. 2.6} \\ = & \Gamma(x)(P \cup \{p\}) \quad \text{by Def. 2.5} \\ = & \Gamma(x)(P) \quad \text{by assumption } p \in P \\ = & (\pi_P(\Gamma(x)))(Q) \quad \text{by Def. 2.6} \\ = & \pi_P(\Gamma)(x)(Q) \quad \text{by Def. 2.8} \end{array}$$

Since this holds for arbitrary Q,it follows that $\pi_P(\Gamma \uparrow_p) = \pi_P(\Gamma)$. □

Proof for **Lemma 2.6**:

Proof. Similar to the proof of Lemma 2.5. □

Appendix B. Proofs for Type Inference

Proof for **Lemma 3.1**:

Proof. By induction on $len(\Lambda)$.

- $\Lambda = \epsilon$. Since $s \cdot \Lambda = s$ and $t \cdot \Lambda = t$, the conclusion holds trivially.
- $\Lambda = \oplus p :: \Lambda'$ or $\Lambda = \ominus p :: \Lambda'$. Assume it is the former case, the latter is similar. By the hypothesis and Lemma A.2, $s \uparrow_p \leqslant t \uparrow_p$. Then by induction, $s \uparrow_p \cdot \Lambda' \leqslant t \uparrow_p \cdot \Lambda'$, that is, $s \cdot \Lambda \leqslant t \cdot \Lambda$.

□

Proof for **Lemma 3.2**:

Proof. We only prove $t \cdot (\ominus p \oplus q) = t \cdot (\oplus q \ominus p)$, the other cases are similar. Consider any $P \in \mathscr{P}$,

$$t \cdot (\ominus p \oplus q)(P) = ((t \downarrow_p) \uparrow_q)(P) = (t \downarrow_p (P \cup \{q\})) = t((P \cup \{q\}) \setminus \{p\})$$

and

$$t\cdot(\oplus q\ominus p)(P) = ((t\uparrow_q)\downarrow_p)(P) = t((P\backslash\{p\})\cup\{q\}) = t((P\cup\{q\})\backslash(\{p\}\backslash\{q\})) = t((P\cup\{q\})\backslash\{p\})$$

Therefore, $t\cdot(\ominus p\oplus q) = t\cdot(\oplus q\ominus p)$. □

Proof for **Lemma 3.3**:

Proof. By induction on $len(\Lambda)$. The conclusion holds when $len(\Lambda)=0$ and $len(\Lambda)=1$ by Lemma 3.2. Suppose $len(\Lambda)>1$, there exists Λ' and q such that $\Lambda = \circledast q :: \Lambda'$ where $\circledast \in \{\oplus,\ominus\}$.

$$\begin{aligned}
(t\cdot\circledcirc p)\cdot\Lambda &= ((t\cdot\circledcirc p)\cdot\circledast q)\cdot\Lambda' && \text{(by Definition 3.1)}\\
&= (t\cdot(\circledcirc p\circledast q))\cdot\Lambda' && \text{(by Definition 3.1)}\\
&= (t\cdot(\circledast q\circledcirc p))\cdot\Lambda' && \text{(by Lemma 3.2)}\\
&= ((t\cdot\circledast q)\cdot\circledcirc p)\cdot\Lambda' && \text{(by Definition 3.1)}\\
&= ((t\cdot\circledast q)\cdot\Lambda')\cdot\circledcirc p && \text{(by induction hypothesis)}\\
&= (t\cdot(\circledast q :: \Lambda'))\cdot\circledcirc p && \text{(by Definition 3.1)}\\
&= (t\cdot\Lambda)\cdot\circledcirc p
\end{aligned}$$

□

Proof for **Lemma 3.4**:

Proof. By case analysis.

- $((t\cdot\oplus p)\cdot\oplus p)(P) = t(P\cup\{p\}\cup\{p\}) = t(P\cup\{p\}) = t\cdot(\oplus p)$ for each $P\in\mathscr{P}$.
- $((t\cdot\oplus p)\cdot\ominus p)(P) = t((P\setminus\{p\})\cup\{p\}) = t(P\cup\{p\}) = t\cdot(\oplus p)$ for each $P\in\mathscr{P}$.
- $((t\cdot\ominus p)\cdot\oplus p)(P) = t((P\cup\{p\})\setminus\{p\}) = t(P\setminus\{p\}) = t\cdot(\ominus p)$ for each $P\in\mathscr{P}$.
- $((t\cdot\ominus p)\cdot\ominus p)(P) = t((P\setminus\{p\})\setminus\{p\}) = t(P\setminus\{p\}) = t\cdot(\ominus p)$ for each $P\in\mathscr{P}$.

□

Proof for **Lemma 3.5**:

Proof. By induction on $len(\Lambda)$. The conclusion holds trivially for $len(\Lambda)=0$ and for $len(\Lambda)=1$ by Lemma 3.4 . When $len(\Lambda)>1$, without loss of generality, assume $\Lambda = \oplus p :: \Lambda'$.

$$\begin{aligned}
t\cdot\Lambda\cdot\Lambda &= (t\cdot(\oplus p\cdot\Lambda'))\cdot(\oplus p :: \Lambda')\\
&= (t\cdot(\Lambda'\cdot\oplus p))\cdot(\oplus p :: \Lambda') && \text{(by Lemma 3.3)}\\
&= (((t\cdot\Lambda')\cdot\oplus p)\cdot\oplus p)\cdot\Lambda' && \text{(by Definition 3.1)}\\
&= ((t\cdot\Lambda')\cdot\oplus p)\cdot\Lambda' && \text{(by Lemma 3.4)}\\
&= ((t\cdot\oplus p)\cdot\Lambda')\cdot\Lambda' && \text{(by Lemma 3.3)}\\
&= (t\cdot\oplus p)\cdot\Lambda' && \text{(by induction hypothesis)}\\
&= t\cdot(\oplus p :: \Lambda') && \text{(by Definition 3.1)}\\
&= t\cdot\Lambda
\end{aligned}$$

□

Proof for **Lemma 3.6**:

Proof. By induction on $len(\Lambda)$.
$len(\Lambda) = 0$**:** Trivially.
$len(\Lambda) > 0$**:** In this case we have $\Lambda = \Lambda' :: \oplus q$ or $\Lambda' :: \ominus q$, where $len(\Lambda') \geqslant 0$, $p \notin \Lambda'$, $q \neq p$. We only prove $\Lambda' :: \oplus q$, the other case is similar. Consider any P, we have

$$\begin{aligned}
&((s \triangleright_p t) \cdot (\Lambda' :: \oplus q))(P) \\
=\ &((s \triangleright_p t) \cdot \Lambda')(P \cup \{q\}) \\
=\ &((s \cdot \Lambda') \triangleright_p (t \cdot \Lambda'))(P \cup \{q\}) \text{ (By induction)} \\
=\ &\begin{cases} (s \cdot \Lambda')(P \cup \{q\}) & p \in P \\ (t \cdot \Lambda')(P \cup \{q\}) & p \notin P \end{cases} \\
=\ &\begin{cases} (s \cdot (\Lambda' :: \oplus q))(P) & p \in P \\ (t \cdot (\Lambda' :: \oplus q))(P) & p \notin P \end{cases} \\
=\ &((s \cdot (\Lambda' :: \oplus q)) \triangleright_p (t \cdot (\Lambda' :: \oplus q)))(P)
\end{aligned}$$

□

Proof for **Lemma 3.7**:

Proof. ($\Rightarrow$) by applying Lemma 3.1 with $\Lambda = \oplus p$ and $\Lambda = \ominus p$ respectively.
($\Leftarrow$) $\forall P \in \mathscr{P}$,

(1) If $p \in P$, by Lemma 2.3 $s(P) = (s \uparrow_p)(P) = (s \cdot \oplus p)(P)$ and $t(P) = (t \uparrow_p)(P) = (t \cdot \oplus p)(P)$, since $s \cdot \oplus p \leqslant t \cdot \oplus p$, then $s(P) \leqslant t(P)$.
(2) If $p \notin P$, by Lemma 2.3, $s(P) = (s \downarrow_p)(P) = (s \cdot \ominus p)(P)$ and $t(P) = (t \downarrow_p)(P) = (t \cdot \ominus p)(P)$, since $s \cdot \ominus p \leqslant t \cdot \ominus p$, then $s(P) \leqslant t(P)$.

This indicates that $s \leqslant t$. □

Proof for **Lemma 3.8**:

Proof. The proof of (a), by induction on $\Gamma^G; \Gamma; \Lambda \vdash_{tr} e : t$.
TT-VAR-L/G Trivially.
TT-OP In this case we have $e \cong e_1$ **op** e_2 and the following derivation

$$\frac{\Gamma^G; \Gamma; \Lambda \vdash_{tr} e_1 : t_1 \qquad \Gamma^G; \Gamma; \Lambda \vdash_{tr} e_2 : t_2}{\Gamma^G; \Gamma; \Lambda \vdash_{tr} e_1 \textbf{ op } e_2 : t_1 \sqcup t_2}$$

By induction on e_i, we can get $(\Gamma^G, \Gamma \cdot \Lambda) \vdash e_i : (t_i \cdot \Lambda)$. By Lemma 3.1 and $t_i \leqslant t_1 \sqcup t_2$, we have $t_i \cdot \Lambda \leqslant (t_1 \sqcup t_2) \cdot \Lambda$. By subsumption, we have $(\Gamma^G, \Gamma \cdot \Lambda) \vdash e_i : ((t_1 \sqcup t_2) \cdot \Lambda)$. Finally, by applying Rule (T-OP), we get $(\Gamma^G, \Gamma \cdot \Lambda) \vdash e_1$ **op** $e_2 : ((t_1 \sqcup t_2) \cdot \Lambda)$.
The proof of (b):
TT-ASS-L In this case we have $c \cong x := e$, x is non-global, and the following derivation

$$\frac{(\mathbf{T_1})\ \Gamma^G; \Gamma; \Lambda \vdash_{tr} e : t \qquad (\mathbf{T_2})\ t \leqslant_\Lambda \Gamma(x)}{\Gamma^G; \Gamma; \Lambda; A \vdash_{tr} x := e : \Gamma(x)}$$

By (a) on $(\mathbf{T_1})$, $(\Gamma^G, \Gamma \cdot \Lambda) \vdash e : (t \cdot \Lambda)$. From $(\mathbf{T_2})$, we get $t \cdot \Lambda \leqslant (\Gamma \cdot \Lambda)(x)$. So by subsumption, $(\Gamma^G, \Gamma \cdot \Lambda) \vdash e : (\Gamma \cdot \Lambda)(x)$. Finally, by Rule (T-ASS), the result follows.

TT-ASS-G Similar to the case of (TT-ASS-L).

TT-LETVAR In this case we have $c \cong$ **letvar** $x = e$ **in** c' and the following derivation

$$\frac{(\mathbf{T_1})\ \Gamma^G; \Gamma; \Lambda \vdash_{tr} e : s \qquad (\mathbf{T_2})\ s \leqslant_\Lambda s' \qquad (\mathbf{T_3})\ \Gamma^G; \Gamma[x : s']; \Lambda; A \vdash_{tr} c' : t}{\Gamma^G; \Gamma; \Lambda; A \vdash_{tr} \textbf{letvar}\ x = e\ \textbf{in}\ c' : t}$$

By (a) on $(\mathbf{T_1})$, $(\Gamma^G, \Gamma \cdot \Lambda) \vdash e : (s \cdot \Lambda)$. From $(\mathbf{T_2})$, we get $s \cdot \Lambda \leqslant s' \cdot \Lambda$. So by subsumption, $(\Gamma^G, \Gamma \cdot \Lambda) \vdash e : s' \cdot \Lambda$. By induction on $(\mathbf{T_3})$, we have $(\Gamma^G, \Gamma[x : s'] \cdot \Lambda); A \vdash c' : t \cdot \Lambda$, that is $(\Gamma^G, (\Gamma \cdot \Lambda)[x : s' \cdot \Lambda]); A \vdash c' : t \cdot \Lambda$. Finally, by Rule (T-LETVAR), the result follows.

TT-SEQ By induction and Rules (T-SUB$_c$), (T-SEQ).

TT-IF By induction and Rules (T-SUB$_c$), (T-IF).

TT-WHILE By induction and Rules (T-SUB$_c$), (T-WHILE).

TT-CALL In this case we have $c \cong x :=$ **call** $B.f(\overline{e})$ and the following derivation

$$\frac{(\mathbf{T_1})\ FT(B.f) = \overline{t} \xrightarrow{s_b} t' \qquad (\mathbf{T_2})\ \Gamma^G; \Gamma; \Lambda \vdash_{tr} \overline{e} : \overline{s} \qquad (\mathbf{T_3})\ \overline{s} \leqslant_\Lambda \overline{\pi_{\Theta(A)}(t)} \qquad (\mathbf{T_4})\ \pi_{\Theta(A)}(t') \leqslant_\Lambda \Gamma(x)}{\Gamma^G; \Gamma; \Lambda; A \vdash_{tr} x := \textbf{call}\ B.f(\overline{e}) : \Gamma(x) \sqcap \pi_{\Theta(A)}(s_b)}$$

By (c) on $(\mathbf{T_1})$, $\Gamma^G \vdash B.f : \overline{t} \xrightarrow{s_b} t'$. By (a) on $(\mathbf{T_2})$, $(\Gamma^G, \Gamma \cdot \Lambda) \vdash \overline{e} : \overline{s \cdot \Lambda}$. From $(\mathbf{T_3})$, we get $\overline{s \cdot \Lambda} \leqslant \overline{(\pi_{\Theta(A)}(t) \cdot \Lambda)} = \overline{\pi_{\Theta(A)}(t)}$. So by subsumption, $(\Gamma^G, \Gamma \cdot \Lambda) \vdash \overline{e} : \overline{\pi_{\Theta(A)}(t)}$. Similarly, from $(\mathbf{T_4})$, we get $\pi_{\Theta(A)}(t') \leqslant (\Gamma \cdot \Lambda)(x)$. Then by Rule (T-CALL), $(\Gamma^G, \Gamma \cdot \Lambda); A \vdash x :=$ **call** $B.f(\overline{e}) : (\Gamma \cdot \Lambda)(x) \sqcap \pi_{\Theta(A)}(s_b)$. Finally, it is easy to check that $(\Gamma \cdot \Lambda)(x) \sqcap \pi_{\Theta(A)}(s_b) = ((\Gamma)(x) \sqcap \pi_{\Theta(A)}(s_b)) \cdot \Lambda$, the result follows.

TT-CP In this case we have $c \cong$ **test**(p) c_1 **else** c_2 and the following derivation

$$\frac{\Gamma^G; \Gamma; \Lambda :: \oplus p; A \vdash_{tr} c_1 : t_1 \qquad \Gamma^G; \Gamma; \Lambda :: \ominus p; A \vdash_{tr} c_2 : t_2}{\Gamma^G; \Gamma; \Lambda; A \vdash_{tr} \textbf{test}(p)\ c_1\ \textbf{else}\ c_2 : t_1 \triangleright_p t_2}$$

By induction on c_i, we have

$$(\Gamma^G, \Gamma \cdot (\Lambda :: \oplus p)); A \vdash c_1 : (t_1 \cdot (\Lambda :: \oplus p)) \quad (\Gamma^G, \Gamma \cdot (\Lambda :: \ominus p)); A \vdash c_2 : (t_2 \cdot (\Lambda :: \ominus p))$$

which is equivalent to

$$(\Gamma^G, (\Gamma \cdot \Lambda) \uparrow_p); A \vdash c_1 : (t_1 \cdot \Lambda) \uparrow_p \quad (\Gamma^G, (\Gamma \cdot \Lambda) \downarrow_p); A \vdash c_2 : (t_2 \cdot \Lambda) \downarrow_p$$

Let t' be $(t_1 \cdot \Lambda) \uparrow_p \triangleright_p (t_2 \cdot \Lambda) \downarrow_p$. By Rule (T-CP), we have

$$(\Gamma^G, \Gamma \cdot \Lambda); A \vdash \textbf{test}(p)\ c_1\ \textbf{else}\ c_2 : t'.$$

Since Λ is collected from the context of c and there are no nested checks of p, we have $p \notin \Lambda$. Let's consider any P. If $p \in P$, then

$$\begin{aligned} t'(P) &= (t_1 \cdot \Lambda) \uparrow_p (P) && (p \in P) \\ &= (t_1 \cdot \Lambda)(P) && (\text{Lemma 2.3}) \\ &= ((t_1 \cdot \Lambda) \triangleright_p (t_2 \cdot \Lambda))(P) && (p \in P) \\ &= ((t_1 \triangleright_p t_2) \cdot \Lambda)(P) && (\text{Lemma 3.6}) \end{aligned}$$

Similarly, if $p \notin P$, $t'(P) = ((t_1 \triangleright_p t_2) \cdot \Lambda)(P)$. Therefore, $t' = (t_1 \triangleright_p t_2) \cdot \Lambda$ and thus the result follows.

The proof of (c) :
Clearly, we have

$$\frac{\Gamma^G; [\overline{x} : \overline{t}, r : t']; \epsilon; B \vdash_{tr} c : s_b \qquad s \leqslant s_b}{\Gamma^G \vdash_{tr} B.f(\overline{x})\{\textbf{init } r = 0 \textbf{ in } \{c; \textbf{return } r\}\} : \overline{t} \xrightarrow{s} t'}$$

Applying (b) on c, $(\Gamma^G, [\overline{x} : \overline{t}, r : t']); B \vdash c : s_b$. By subsumption, we also have $(\Gamma^G, [\overline{x} : \overline{t}, r : t']); B \vdash c : s$. Finally, by Rule (T-FUN), the result follows. □

To prove the completeness, we need to prove some auxiliary lemmas.

Lemma B.1 *Let Λ be the permission trace collected from the context of e or c.*
(a) If $(\Gamma^G, \Gamma) \vdash e : t$, then $(\Gamma^G, \Gamma \cdot \Lambda) \vdash e : (t \cdot \Lambda)$
(b) If $(\Gamma^G, \Gamma); A \vdash c : t$, then $(\Gamma^G, \Gamma \cdot \Lambda); A \vdash c : (t \cdot \Lambda)$.

Proof. The proof of (a) : by induction on $(\Gamma^G, \Gamma) \vdash e : t$.
T-VAR-L/G Trivially.
T-OP In this case we have $e \cong e_1 \textbf{ op } e_2$ and the following derivation

$$\frac{(\Gamma^G, \Gamma) \vdash e_1 : t \qquad (\Gamma^G, \Gamma) \vdash e_2 : t}{(\Gamma^G, \Gamma) \vdash e_1 \textbf{ op } e_2 : t}$$

By induction on e_i, $(\Gamma^G, \Gamma \cdot \Lambda) \vdash e_i : (t \cdot \Lambda)$. By Rule (T-OP), we have $(\Gamma^G, \Gamma \cdot \Lambda) \vdash e_1 \textbf{ op } e_2 : (t \cdot \Lambda)$.
T-SUB$_e$ In this case we have the following derivation

$$\frac{(\textbf{T1})\ (\Gamma^G, \Gamma) \vdash e : s \qquad (\textbf{T2})\ s \leqslant t}{(\Gamma^G, \Gamma) \vdash e : t}$$

By induction on (**T1**), $(\Gamma^G, \Gamma \cdot \Lambda) \vdash_{tr} e : (s \cdot \Lambda)$. From (**T2**) and Lemma 3.1, we get $s \cdot \Lambda \leqslant t \cdot \Lambda$. So by subsumption, the result follows.

The proof of (b): by induction on $(\Gamma^G, \Gamma); A \vdash c : t$.
T-ASS-L In this case we have $c \cong x := e$, x is non-global, and the following derivation

$$\frac{(\Gamma^G, \Gamma) \vdash e : \Gamma(x)}{(\Gamma^G, \Gamma); A \vdash x := e : \Gamma(x)}$$

By (a) on e, $(\Gamma^G, \Gamma \cdot \Lambda) \vdash e : (\Gamma \cdot \Lambda)(x)$. Then by Rule (T-ASS-L), the result follows.

T-ASS-G similarly to the case of (T-ASS-L).

T-LETVAR In this case we have $c \cong \textbf{letvar}\ x = e\ \textbf{in}\ c'$ and the following derivation

$$\frac{(\Gamma^G, \Gamma) \vdash e : s \qquad (\Gamma^G, \Gamma[x : s]); A \vdash c' : t}{(\Gamma^G, \Gamma); A \vdash \textbf{letvar}\ x = e\ \textbf{in}\ c' : t}$$

By (a) on e, $(\Gamma^G, \Gamma \cdot \Lambda) \vdash e : (s \cdot \Lambda)$. By induction on c, $(\Gamma^G, (\Gamma[x : s]) \cdot \Lambda); A \vdash c' : (t \cdot \Lambda)$, that is $(\Gamma^G, (\Gamma \cdot \Lambda)[x : s \cdot \Lambda]); A \vdash c' : (t \cdot \Lambda)$. Then by Rule (T-LETVAR), the result follows.

T-SEQ By induction and Rule (T-SEQ).

T-IF By induction and Rule (T-IF).

T-WHILE By induction and Rule (T-WHILE).

T-CALL in this case we have $c \cong x := \textbf{call}\ B.f(\overline{e})$ and the following derivation

$$\frac{(\textbf{T1})\ (\Gamma^G, \Gamma) \vdash \overline{e} : \overline{\pi_{\Theta(A)}(t)} \qquad FT(B.f) = \overline{t} \xrightarrow{s} t' \qquad (\textbf{T2})\ \pi_{\Theta(A)}(t') \leqslant \Gamma(x)}{(\Gamma^G, \Gamma); A \vdash x := \textbf{call}\ B.f(\overline{e}) : \Gamma(x) \sqcap \pi_{\Theta(A)}(s)}$$

By (a) on (**T1**), $(\Gamma^G, \Gamma \cdot \Lambda) \vdash \overline{e} : \overline{\pi_{\Theta(A)}(t) \cdot \Lambda}$, that is, $(\Gamma^G, \Gamma \cdot \Lambda) \vdash \overline{e} : \overline{\pi_{\Theta(A)}(t)}$. From (**T2**) and Lemma 3.1, $\pi_{\Theta(A)}(t') \cdot \Lambda \leqslant (\Gamma \cdot \Lambda)(x)$, that is, $\pi_{\Theta(A)}(t') \leqslant (\Gamma \cdot \Lambda)(x)$. Then by Rule (T-CALL), $(\Gamma^G, \Gamma \cdot \Lambda); A \vdash x := \textbf{call}\ B.f(\overline{e}) : (\Gamma \cdot \Lambda)(x) \sqcap \pi_{\Theta(A)}(s)$. Finally, it is each to check that $(\Gamma \cdot \Lambda)(x) \sqcap \pi_{\Theta(A)}(s) = (\Gamma(x) \sqcap \pi_{\Theta(A)}(s)) \cdot \Lambda$, then the result follows.

T-CP In this case we have $c \cong \textbf{test}(p)\ c_1\ \textbf{else}\ c_2$ and the following derivation

$$\frac{(\Gamma^G, \Gamma \uparrow_p); A \vdash c_1 : t_1 \qquad (\Gamma^G, \Gamma \downarrow_p); A \vdash c_2 : t_2}{(\Gamma^G, \Gamma); A \vdash \textbf{test}(p)\ c_1\ \textbf{else}\ c_2 : t_1 \triangleright_p t_2}$$

By induction on c_1 and c_2, we have

$$(\Gamma^G, (\Gamma \uparrow_p) \cdot \Lambda); A \vdash c_1 : t_1 \cdot \Lambda \quad (\Gamma^G, (\Gamma \downarrow_p) \cdot \Lambda); A \vdash c_2 : t_2 \cdot \Lambda$$

Since Λ is collected from the context of c and there are no nested checks of p, we have $p \notin \Lambda$. By Lemma 3.3, $(\Gamma \uparrow_p) \cdot \Lambda = (\Gamma \cdot \Lambda) \uparrow_p$ and $(\Gamma \downarrow_p) \cdot \Lambda = (\Gamma \cdot \Lambda) \downarrow_p$. Then by Rule (T-CP), we have

$$(\Gamma^G, \Gamma \cdot \Lambda); A \vdash \textbf{test}(p)\ c_1\ \textbf{else}\ c_2 : (t_1 \cdot \Lambda) \triangleright_p (t_2 \cdot \Lambda).$$

Finally, by Lemma 3.6, $(t_1 \triangleright_p t_2) \cdot \Lambda = (t_1 \cdot \Lambda) \triangleright_p (t_2 \cdot \Lambda)$, and thus the result follows.

T-SUB$_c$ In this case we have the following derivation

$$\frac{(\textbf{T1})\ (\Gamma^G, \Gamma); A \vdash c : s \qquad (\textbf{T2})\ t \leqslant s}{(\Gamma^G, \Gamma); A \vdash c : t}$$

By induction on (**T1**), $(\Gamma^G, \Gamma \cdot \Lambda); A \vdash c : (s \cdot \Lambda)$. From (**T2**) and by Lemma 3.1, $t \cdot \Lambda \leqslant s \cdot \Lambda$. Then by subsumption, the result follows.

□

Lemma B.2 *(a) If* $(\Gamma^G, \Gamma \cdot \Lambda) \vdash e : t$, *then* $(\Gamma^G, \Gamma \cdot \Lambda) \vdash e : (t \cdot \Lambda)$
(b) If $(\Gamma^G, \Gamma \cdot \Lambda); A \vdash c : t$, *then* $(\Gamma^G, \Gamma \cdot \Lambda); A \vdash c : (t \cdot \Lambda)$.

Proof. By Lemma B.1 and Lemma 3.5. □

Proof for **Lemma 3.9**:

Proof. The proof of (a) : by induction on $(\Gamma^G, \Gamma \cdot \Lambda) \vdash e : t \cdot \Lambda$.
T-VAR-L Trivially with $s = \Gamma(x)$.
T-VAR-G In this case we have $e \cong x$, x is global, and the following derivation

$$\frac{(\Gamma^G, \Gamma \cdot \Lambda) \vdash x : \Gamma^G(x) \qquad \Gamma^G(x) \leqslant t \cdot \Lambda}{(\Gamma^G, \Gamma \cdot \Lambda) \vdash x : t \cdot \Lambda}$$

Take s as $\Gamma^G(x)$ and the result follows trivially.
T-OP In this case we have $e \cong e_1$ **op** e_2 and the following derivation

$$\frac{(\Gamma^G, \Gamma \cdot \Lambda) \vdash e_1 : t \cdot \Lambda \qquad (\Gamma^G, \Gamma \cdot \Lambda) \vdash e_2 : t \cdot \Lambda}{(\Gamma^G, \Gamma \cdot \Lambda) \vdash e_1 \ \mathbf{op}\ e_2 : t \cdot \Lambda}$$

By induction on e_i, there exists s_i such that $\Gamma^G; \Gamma; \Lambda \vdash_{tr} e_i : s_i$ and $s_i \leqslant_\Lambda t$. By Rule (TT-OP), we have $\Gamma^G; \Gamma; \Lambda \vdash_{tr} e_1$ **op** $e_2 : s_1 \sqcup s_2$. Moreover, it is clear that $s_1 \sqcup s_2 \leqslant_\Lambda t$. Therefore, the result follows.
T-SUB$_e$ In this case we have the following derivation

$$\frac{(\mathbf{T1})\ (\Gamma^G, \Gamma \cdot \Lambda) \vdash e : s \qquad (\mathbf{T2})\ s \leqslant t \cdot \Lambda}{(\Gamma^G, \Gamma \cdot \Lambda) \vdash e : t \cdot \Lambda}$$

Applying Lemma B.2 on (**T1**), $(\Gamma^G, \Gamma \cdot \Lambda) \vdash e : s \cdot \Lambda$. Then by induction, there exists s' such that $\Gamma^G; \Gamma; \Lambda \vdash_{tr} e : s'$ and $s' \leqslant_\Lambda s$. From (**T2**) and Lemma 3.5, we can get $s' \leqslant_\Lambda t \cdot \Lambda \leqslant_\Lambda t$. Thus the result follows.
The proof of (b): by induction on $(\Gamma^G, \Gamma \cdot \Lambda); A \vdash c : t \cdot \Lambda$.
T-ASS-L In this case we have $c \cong x := e$, x is non-global, and the following derivation

$$\frac{(\Gamma^G, \Gamma \cdot \Lambda) \vdash e : (\Gamma \cdot \Lambda)(x)}{(\Gamma^G, \Gamma \cdot \Lambda); A \vdash x := e : (\Gamma \cdot \Lambda)(x)}$$

By (a) on e, there exists s such that $\Gamma^G; \Gamma; \Lambda \vdash_{tr} e : s$ and $s \leqslant_\Lambda \Gamma(x)$. Then by Rule (TT-ASS-L), $\Gamma^G; \Gamma; \Lambda; A \vdash_{tr} x := e : \Gamma(x)$, and thus the result follows.
T-ASS-G In this case we have $c \cong x := e$, x is global, and the following derivation

$$\frac{\dfrac{(\mathbf{T_1})\ (\Gamma^G, \Gamma \cdot \Lambda) \vdash e : \Gamma^G(x)}{(\Gamma^G, \Gamma \cdot \Lambda); A \vdash x := e : \Gamma^G(x)} \qquad (\mathbf{T_2})\ t \cdot \Lambda \leqslant \Gamma^G(x)}{(\Gamma^G, \Gamma \cdot \Lambda); A \vdash x := e : t \cdot \Lambda}$$

Applying Lemma B.2 on $(\mathbf{T1})$, $(\Gamma^G, \Gamma \cdot \Lambda) \vdash e : \Gamma^G(x) \cdot \Lambda$. And by (a) on e, there exists s such that $\Gamma^G; \Gamma; \Lambda \vdash_{tr} e : s$ and $s \leqslant_\Lambda \Gamma^G(x)$. They by Rule (TT-ASS-G), $\Gamma^G; \Gamma; \Lambda; A \vdash_{tr} x := e : \Gamma^G(x)$. Finally, from $(\mathbf{T_2})$, we have $t \leqslant_\Lambda \Gamma^G(x)$.

T-LETVAR In this case we have $c \cong$ **letvar** $x = e$ **in** c' and the following derivation

$$\frac{(\mathbf{T_1})\ (\Gamma^G, \Gamma \cdot \Lambda) \vdash e : s \qquad (\mathbf{T_2})\ (\Gamma^G, (\Gamma \cdot \Lambda)[x : s]); A \vdash c' : t \cdot \Lambda}{(\Gamma^G, \Gamma \cdot \Lambda); A \vdash \textbf{letvar}\ x = e\ \textbf{in}\ c' : t \cdot \Lambda}$$

Applying Lemma B.2 on $(\mathbf{T_1})$, $(\Gamma^G, \Gamma \cdot \Lambda) \vdash e : s \cdot \Lambda$. Then by (a), there exists s' such that $\Gamma^G; \Gamma; \Lambda \vdash_{tr} e : s'$ and $s' \leqslant_\Lambda s$. Applying Lemma B.1 on $(\mathbf{T_2})$, $(\Gamma^G, (\Gamma \cdot \Lambda)[x : s]) \cdot \Lambda); A \vdash c' : (t \cdot \Lambda) \cdot \Lambda$. And by Lemma 3.5, we have $(\Gamma^G, \Gamma[x : s] \cdot \Lambda); A \vdash c' : (t \cdot \Lambda)$. Then by induction, there exists t' such that $\Gamma^G; \Gamma[x : s]; \Lambda; A \vdash_{tr} c' : t'$ and $t \leqslant_\Lambda t'$. Finally, by Rule (TT-LETVAR), we have $\Gamma^G; \Gamma; \Lambda; A \vdash_{tr}$ **letvar** $x = e$ **in** $c' : t'$.

T-IF By induction and Rule (TT-IF).

T-WHILE By induction and Rule (TT-WHILE).

T-SEQ By induction and Rule (TT-SEQ).

T-CALL In this case we have $c \cong x :=$ **call** $B.f(\overline{e})$ and the following derivation

$$\frac{\dfrac{FT(B.f) = \overline{t_p} \xrightarrow{t_b} t_r \qquad (\mathbf{T1})\ (\Gamma^G, \Gamma \cdot \Lambda) \vdash \overline{e} : \overline{\pi_{\Theta(A)}(t_p)} \qquad (\mathbf{T2})\ \pi_{\Theta(A)}(t_r) \leqslant (\Gamma \cdot \Lambda)(x)}{(\Gamma^G, \Gamma \cdot \Lambda); A \vdash x := \textbf{call}\ B.f(\overline{e}) : t'}\ (\mathbf{T3})}{(\Gamma^G, \Gamma \cdot \Lambda); A \vdash x := \textbf{call}\ B.f(\overline{e}) : t \cdot \Lambda}$$

where $t' = (\Gamma \cdot \Lambda)(x) \sqcap \pi_{\Theta(A)}(t_b)$ and $(\mathbf{T3})$ is $t \cdot \Lambda \leqslant t'$. From $(\mathbf{T1})$, we have $(\Gamma^G, \Gamma \cdot \Lambda) \vdash \overline{e} : \overline{(\pi_{\Theta(A)}(t_p) \cdot \Lambda)}$. Then by (a), there exists $\overline{t_e}$ such that $\Gamma^G; \Gamma; \Lambda \vdash_{tr} \overline{e} : \overline{t_e}$ and $\overline{t_e} \leqslant_\Lambda \overline{\pi_{\Theta(A)}(t_p)}$. From $(\mathbf{T2})$, we have $\pi_{\Theta(A)}(t_r) \leqslant_\Lambda \Gamma(x)$. Then by Rule (TT-CALL), $\Gamma; \Lambda; A \vdash x :=$ **call** $B.f(\overline{e}) : \Gamma(x) \sqcap \pi_{\Theta(A)}(t_b)$. Finally, it is easy to check $t \leqslant_\Lambda \Gamma(x) \sqcap \pi_{\Theta(A)}(t_b)$.

T-CP In this case we have $c \cong$ **test**(p) c_1 **else** c_2 and the following derivation

$$\frac{\dfrac{(\Gamma^G, (\Gamma \cdot \Lambda) \uparrow_p); A \vdash c_1 : t_1 \qquad (\Gamma^G, (\Gamma \cdot \Lambda) \downarrow_p); A \vdash c_2 : t_2}{(\Gamma^G, \Gamma \cdot \Lambda); A \vdash \textbf{test}(p)\ c_1\ \textbf{else}\ c_2 : t_1 \triangleright_p t_2} \qquad (\mathbf{T1})\ t \cdot \Lambda \leqslant t_1 \triangleright_p t_2}{(\Gamma^G, \Gamma \cdot \Lambda); A \vdash \textbf{test}(p)\ c_1\ \textbf{else}\ c_2 : t \cdot \Lambda}$$

Clearly, $(\Gamma \cdot \Lambda) \uparrow_p = \Gamma \cdot (\Lambda :: \oplus p)$ and $(\Gamma \cdot \Lambda) \downarrow_p = \Gamma \cdot (\Lambda :: \ominus p)$. Applying Lemma B.2 on c_1 and c_2 with $\Lambda :: \oplus p$ and $\Lambda :: \ominus p$ respectively, we have

$$(\Gamma^G, \Gamma \cdot (\Lambda :: \oplus p)); A \vdash c_1 : t_1 \cdot (\Lambda :: \oplus p) \quad (\Gamma^G, \Gamma \cdot (\Lambda :: \ominus p)); A \vdash c_2 : t_2 \cdot (\Lambda :: \ominus p)$$

By induction, there exist s_1 and s_2 such that

$$\Gamma^G; \Gamma; (\Lambda :: \oplus p); A \vdash_{tr} c_1 : s_1 \text{ with } t_1 \leqslant_{\Lambda :: \oplus p} s_1 \quad \Gamma^G; \Gamma; (\Lambda :: \ominus p); A \vdash_{tr} c_2 : s_2 \text{ with } t_2 \leqslant_{\Lambda :: \ominus p} s_2$$

Then by Rule (TT-CP), we have

$$\Gamma^G; \Gamma; \Lambda; A \vdash_{tr} \textbf{test}(p)\ c_1\ \textbf{else}\ c_2 : s_1 \triangleright_p s_2.$$

The remaining is to prove $t \cdot \Lambda \leqslant (s_1 \triangleright_p s_2) \cdot \Lambda$. Since Λ is collected from the context of c and there are no nested checks of p, we have $p \notin \Lambda$. Consider any P. If $p \in P$, then

$$\begin{aligned}
((s_1 \triangleright_p s_2) \cdot \Lambda)(P) &= ((s_1 \cdot \Lambda) \triangleright_p (s_2 \cdot \Lambda))(P) && \text{(Lemma 3.6)} \\
&= (s_1 \cdot \Lambda)(P) && (p \in P) \\
&= (s_1 \cdot (\Lambda :: \oplus p))(P) && \text{(Lemma 2.3)} \\
&\geqslant (t_1 \cdot (\Lambda :: \oplus p))(P) && (t_1 \leqslant_{\Lambda::\oplus p} s_1) \\
&= (t_1 \cdot \Lambda)(P) && \text{(Lemma 2.3)} \\
&= ((t_1 \cdot \Lambda) \triangleright_p (t_2 \cdot \Lambda))(P) && (p \in P) \\
&= ((t_1 \triangleright_p t_2) \cdot \Lambda)(P) && \text{(Lemma 3.6)} \\
&\geqslant ((t \cdot \Lambda) \cdot \Lambda)(P) && (\mathbf{T1}) \\
&= (t \cdot \Lambda)(P) && \text{(Lemma 3.5)}
\end{aligned}$$

Similarly, if $p \notin P$, we also have $(t \cdot \Lambda)(P) \leqslant ((s_1 \triangleright_p s_2) \cdot \Lambda)(P)$. Therefore, the result follows.

T-SUB$_c$ In this case we have the following derivation

$$\frac{(\mathbf{T1})\ (\Gamma^G, \Gamma \cdot \Lambda); A \vdash c : s \qquad (\mathbf{T2})\ t \cdot \Lambda \leqslant s}{(\Gamma^G, \Gamma \cdot \Lambda); A \vdash c : t \cdot \Lambda}$$

where $(\mathbf{T1})$ does not end of (T-ASS-G), (T-CALL) or (T-CP). Applying Lemma B.2 on $(\mathbf{T1})$, $(\Gamma^G, \Gamma \cdot \Lambda); A \vdash c : s \cdot \Lambda$. Then by induction, there exists s' such that $\Gamma^G; \Gamma; \Lambda; A \vdash_{tr} c : s'$ and $s \leqslant_\Lambda s'$. From $(\mathbf{T2})$ and by Lemma 3.1, we can get $t \leqslant_\Lambda s \leqslant_\Lambda s'$. Thus the result follows.

The proof of (c):
Clearly, we have

$$\frac{(\Gamma^G, [\overline{x} : \overline{t}, r : t']); B \vdash c : s_b}{\Gamma^G \vdash B.f(\overline{x})\{\textbf{init } r = 0 \textbf{ in } \{c; \textbf{return } r\}\} : \overline{t} \xrightarrow{s_b} t'}$$

By (b) on c, there exists t_b such that $\Gamma^G; [\overline{x} : \overline{t}, r : t']; \epsilon; B \vdash_{tr} c : t_b$ and $s_b \leqslant_\epsilon t_b$. Thus by Rule (TT-FUN),

$$\Gamma^G \vdash_{tr} B.f(\overline{x})\{\textbf{init } r = 0 \textbf{ in } \{c; \textbf{return } r\}\} : \overline{t} \xrightarrow{s_b} t'$$

□

Proof for **Lemma 3.10**:

Proof. The proof of (a): by induction on the derivation of $\Gamma^G; \Gamma; \Lambda \vdash_{cg} e : t \rightsquigarrow C$.

TG-VAR-L In this case we have x is non-global and the following derivation

$$\frac{}{\Gamma^G; \Gamma; \Lambda \vdash_{cg} x : \Gamma(x) \rightsquigarrow \emptyset}$$

Clearly, for any θ, $\theta \vDash \emptyset$. By Rule (TT-VAR), we have

$$\frac{}{\Gamma^G\theta; \Gamma\theta; \Lambda; A \vdash_{tr} x : \Gamma\theta(x)}$$

TG-VAR-G Similarly to the case of (TG-VAR-L).

TG-OP In this case we have $e \cong e_1 \ \mathbf{op}\ e_2$ and the following derivation

$$\frac{\Gamma^G;\Gamma;\Lambda \vdash_{cg} e_1 : t_1 \rightsquigarrow C_1 \qquad \Gamma^G;\Gamma;\Lambda \vdash_{cg} e_2 : t_2 \rightsquigarrow C_2}{\Gamma^G;\Gamma;\Lambda \vdash_{cg} e_1 \ \mathbf{op}\ e_2 : t_1 \sqcup t_2 \rightsquigarrow C_1 \cup C_2}$$

Since $\theta \vDash C_1 \cup C_2$, $\theta \vDash C_1$ and $\theta \vDash C_2$. Then by induction on e_i, $\Gamma^G\theta;\Gamma\theta;\Lambda \vdash_{tr} e_i : t_i\theta$. So by Rule (TT-OP), we have $\Gamma^G\theta;\Gamma\theta;\Lambda \vdash_{tr} e_1 \ \mathbf{op}\ e_2 : t_1\theta \sqcup t_2\theta$. Clearly, $t_1\theta \sqcup t_2\theta = (t_1 \sqcup t_2)\theta$.

The proof of (b):

TG-ASS-L In this case we have $c \cong x := e$, x is non-global, and the following derivation

$$\frac{\Gamma^G;\Gamma;\Lambda \vdash_{cg} e : t \rightsquigarrow C_e}{\Gamma^G;\Gamma;\Lambda;A \vdash_{cg} x := e : \Gamma(x) \rightsquigarrow C_e \cup \{(\Lambda, t \leqslant \Gamma(x))\}}$$

Since $\theta \vDash C_e \cup \{(\Lambda, t \leqslant \Gamma(x))\}$, $\theta \vDash C_e$ and $t\theta \leqslant_\Lambda \Gamma(x)\theta$. By (a) on e, $\Gamma^G\theta;\Gamma\theta;\Lambda \vdash_{tr} e : t\theta$. By Rule (TT-ASS), $\Gamma^G\theta;\Gamma\theta;\Lambda;A \vdash_{tr} x := e : \Gamma\theta(x)$.

TG-ASS-L Similarly to the case of (TG-ASS-G).

TG-LETVAR In this case we have $c \cong \mathbf{letvar}\ x = e\ \mathbf{in}\ c'$ and the following derivation

$$\frac{\Gamma^G;\Gamma;\Lambda \vdash_{cg} e : s \rightsquigarrow C_1 \qquad \Gamma^G;\Gamma[x:\alpha];\Lambda;A \vdash_{cg} c' : t \rightsquigarrow C_2 \qquad C = C_1 \cup C_2 \cup \{(\Lambda, s \leqslant \alpha)\}}{\Gamma^G;\Gamma;\Lambda;A \vdash_{cg} \mathbf{letvar}\ x = e\ \mathbf{in}\ c : t \rightsquigarrow C}$$

Since $\theta \vDash C$, $\theta \vDash C_1$, $\theta \vDash C_2$, and $s\theta \leqslant_\Lambda \theta(\alpha)$. By (a) on e, $\Gamma^G\theta;\Gamma\theta;\Lambda \vdash_{tr} e : s\theta$. By induction on c', $\Gamma^G\theta;\Gamma\theta[x:\theta(\alpha)];\Lambda;A \vdash_{tr} c' : t\theta$. Finally, by Rule (TT-LETVAR), $\Gamma^G\theta;\Gamma\theta;\Lambda;A \vdash_{tr}$ $\mathbf{letvar}\ x = e\ \mathbf{in}\ c : t\theta$.

TG-CALL In this case we have $c \cong x := \mathbf{call}\ B.f(\overline{e})$ and the following derivation

$$\frac{\begin{array}{cc} FT_C(B.f) = (\overline{t} \xrightarrow{s_b} t', C_f) & \Gamma^G;\Gamma;\Lambda \vdash_{cg} \overline{e} : \overline{s} \rightsquigarrow \bigcup \overline{C_e} \\ C_a = \{(\Lambda, \overline{s} \leqslant \overline{\pi_{\Theta(A)}(t)}), (\Lambda, \pi_{\Theta(A)}(t') \leqslant \Gamma(x))\} & C = C_f \cup \bigcup \overline{C_e} \cup C_a \end{array}}{\Gamma^G;\Gamma;\Lambda;A \vdash_{cg} x := \mathbf{call}\ B.f(\overline{e}) : \Gamma(x) \sqcap \pi_{\Theta(A)}(s_b) \rightsquigarrow C}$$

Since $\theta \vDash C$, then $\theta \vDash C_f$, $\theta \vDash \overline{C_e}$, $\overline{s\theta} \leqslant_\Lambda \overline{\pi_{\Theta(A)}(t\theta)}$, and $\pi_{\Theta(A)}(t'\theta) \leqslant_\Lambda \Gamma\theta(x))$. By (c) on $B.f$, we have

$$\Gamma^G\theta \vdash_{tr} B.f(x)\{\mathbf{init}\ r = 0\ \mathbf{in}\ \{c; \mathbf{return}\ r\}\} : \overline{t\theta} \xrightarrow{s_b\theta} t'\theta$$

that is, $FT(B.f) = \overline{t\theta} \xrightarrow{s_b\theta} t'\theta$. By (a) on $\overline{e}$,

$$\Gamma^G\theta;\Gamma\theta;\Lambda;A \vdash_{tr} \overline{e} : \overline{s\theta}.$$

Finally, by Rule (TT-CALL),

$$\Gamma^G\theta;\Gamma\theta;\Lambda;A \vdash_{tr} x := \mathbf{call}\ B.f(\overline{e}) : \Gamma\theta(x) \sqcap \pi_{\Theta(A)}(s_b\theta)$$

TG-CP In this case we have $c \cong \textbf{test}(p)\ c_1\ \textbf{else}\ c_2$ and the following derivation

$$\frac{\Gamma^G;\Gamma;\Lambda :: \oplus p; A \vdash_{cg} c_1 : t_1 \rightsquigarrow C_1 \qquad \Gamma^G;\Gamma;\Lambda :: \ominus p; A \vdash_{cg} c_2 : t_2 \rightsquigarrow C_2}{\Gamma;\Lambda; A \vdash_{cg} \textbf{test}(p)\ c_1\ \textbf{else}\ c_2 : t_1 \triangleright_p t_2 \rightsquigarrow C_1 \cup C_2}$$

Since $\theta \vDash C_1 \cup C_2$, then $\theta \vDash C_1$ and $\theta \vDash C_2$. By induction on c_1 and c_2, we get

$$\Gamma^G\theta;\Gamma\theta;(\Lambda :: \oplus p); A \vdash_{tr} c_1 : t_1\theta \qquad \Gamma^G\theta;\Gamma\theta;(\Lambda :: \ominus p); A \vdash_{tr} c_2 : t_2\theta$$

By Rule (TT-CP), we have

$$\Gamma^G\theta;\Gamma\theta;\Lambda; A \vdash_{tr} \textbf{test}(p)\ c_1\ \textbf{else}\ c_2 : t_1\theta \triangleright_p t_2\theta$$

Moreover, it is clear that $(t_1 \triangleright_p t_2)\theta = (t_1\theta) \triangleright_p (t_2\theta)$.

others By induction.

The proof of (c):

$$\frac{\Gamma^G;[\overline{x} : \overline{\alpha}, r : \beta];\epsilon; B \vdash_{cg} c : s \rightsquigarrow C_b \qquad C = C_b \cup \{(\epsilon, \gamma \leqslant s)\}}{\Gamma^G;\vdash_{cg} B.f(x)\big\{\textbf{init}\ r = 0\ \textbf{in}\ \{c; \textbf{return}\ r\}\big\} : \overline{\alpha} \xrightarrow{\gamma} \beta \rightsquigarrow C}$$

As $\theta \vDash C$, we have $\theta \vDash C_b$ and $\theta(\gamma) \leqslant_\epsilon s\theta$. By (b) on c, we have

$$\Gamma^G\theta;[\overline{x} : \overline{\theta(\alpha)}, r : \theta(\beta)];\epsilon; B \vdash_{tr} c : s\theta$$

Finally, by Rule (TT-FUN), we get

$$\Gamma^G\theta \vdash_{tr} B.f(\overline{x})\big\{\textbf{init}\ r = 0\ \textbf{in}\ \{c; \textbf{return}\ r\}\big\} : \overline{\theta(\alpha)} \xrightarrow{\theta(\gamma)} \theta(\beta)$$

□

Proof for **Lemma 3.11**:

Proof. The proof of (a): Let $\Gamma_0 = \{x \mapsto \alpha_x \mid x \in dom(\Gamma)\}$, $\Gamma_0^G = \{x \mapsto \alpha_x \mid x \in dom(\Gamma^G)\}$, and $\theta_0 = \{\alpha_x \mapsto \Gamma(x) \mid x \in dom(\Gamma)\} \cup \{\alpha_x \mapsto \Gamma^G(x) \mid x \in dom(\Gamma^G)\}$, where α_xs are fresh type variables. Clearly, we have $\Gamma_0\theta_0 = \Gamma$ and $\Gamma_0^G\theta_0 = \Gamma^G$. The remaining is to prove that

$$\exists C, s.\ \Gamma_0^G;\Gamma_0;\Lambda \vdash_{cg} e : s \rightsquigarrow C,\ \theta_0 \vDash C \text{ and } s\theta_0 = t \qquad (1)$$

TT-VAR-L In this case we have $e \cong x$, x is non-global, and the following derivation

$$\frac{x \in dom(\Gamma)}{\Gamma^G;\Gamma;\Lambda \vdash_{tr} x : \Gamma(x)}$$

By Rule (TG-VAR), we have $\Gamma_0^G;\Gamma_0;\Lambda \vdash_{cg} x : \Gamma_0(x) \rightsquigarrow \emptyset$. Clearly, $\theta_0 \vDash \emptyset$ and $\Gamma_0\theta_0(x) = \Gamma(x)$.

TT-VAR-G Similarly to the case of (TT-VAR-L).

TT-OP In this case we have $e \cong e_1$ **op** e_2 and the following derivation

$$\frac{\Gamma^G;\Gamma;\Lambda \vdash_{tr} e_1 : t_1 \qquad \Gamma^G;\Gamma;\Lambda \vdash_{tr} e_2 : t_2}{\Gamma^G;\Gamma;\Lambda \vdash_{tr} e_1 \textbf{ op } e_2 : t_1 \sqcup t_2}$$

By induction on e_i, there exist C_i, s_i such that

$$\Gamma_0^G;\Gamma_0;\Lambda \vdash_{cg} e_i : s_i \rightsquigarrow C_i,\ \theta_0 \vDash C_i \text{ and } s_i\theta_0 = t_i.$$

By Rule (TG-OP), we have

$$\Gamma_0^G;\Gamma_0;\Lambda \vdash_{cg} e_1 \textbf{ op } e_2 : s_1 \sqcup s_2 \rightsquigarrow C_1 \cup C_2.$$

Moreover, it is clear that $\theta_0 \vDash C_1 \cup C_2$ and $(s_1 \sqcup s_2)\theta_0 = t_1 \sqcup t_2$.

The proof of (b). Let $\Gamma_0 = \{x \mapsto \alpha_x \mid x \in dom(\Gamma)\}$, $\Gamma_0^G = \{x \mapsto \alpha_x \mid x \in dom(\Gamma^G)\}$, and $\theta_0 = \{\alpha_x \mapsto \Gamma(x) \mid x \in dom(\Gamma)\} \cup \{\alpha_x \mapsto \Gamma^G(x) \mid x \in dom(\Gamma^G)\}$, where α_xs are fresh type variables. Clearly, we have $\Gamma_0\theta_0 = \Gamma$ and $\Gamma_0^G\theta_0 = \Gamma^G$. Assume that different parameters of different functions have different names, and let $V_1 \sharp V_2$ denote $V_1 \cap V_2 = \emptyset$. In the remaining we prove the following statement:

$\exists C, s, \theta.\ \Gamma_0^G;\Gamma_0;\Lambda;A \vdash_{cg} c : s \rightsquigarrow C,\ dom(\theta_0) \sharp dom(\theta), \theta_0 \cup \theta \vDash C,$ and $s(\theta_0 \cup \theta) = t$ (2)

TT-ASS-L In this case we have $c \cong x := e$, x is non-global, and the following derivation

$$\frac{x \in dom(\Gamma) \quad \Gamma^G;\Gamma;\Lambda \vdash_{tr} e : t' \quad t' \leqslant_\Lambda \Gamma(x)}{\Gamma;\Lambda;A \vdash_{tr} x := e : \Gamma(x)}$$

By (1) on e, there exist C, s' such that $\Gamma_0^G;\Gamma_0;\Lambda \vdash_{cg} e : s' \rightsquigarrow C$, $\theta_0 \vDash C$ and $s'\theta_0 = t'$. By Rule (TG-ASS), we get

$$\Gamma_0^G;\Gamma_0;\Lambda;A \vdash_{cg} x := e : \Gamma_0(x) \rightsquigarrow C \cup \{(\Lambda, t' \leqslant \Gamma_0(x))\}.$$

Take $\theta = \emptyset$. Since $s'\theta_0 = t' \leqslant_\Lambda \Gamma(x) = \Gamma_0\theta_0(x)$, then $\theta_0 \vDash \{(\Lambda, t' \leqslant \Gamma_0(x))\}$, and thus $\theta_0 \vDash C \cup \{(\Lambda, t' \leqslant \Gamma_0(x))\}$.

TT-ASS-G Similarly to the case of (TT-ASS-L).

TT-LETVAR In this case we have $c \cong$ **letvar** $x = e$ **in** c' and the following derivation

$$\frac{\Gamma^G;\Gamma;\Lambda \vdash_{tr} e : t_e \qquad t_e \leqslant_\Lambda t'_e \qquad \Gamma^G;\Gamma[x : t'_e];\Lambda;A \vdash_{tr} c' : t}{\Gamma^G;\Gamma;\Lambda;A \vdash_{tr} \textbf{letvar } x = e \textbf{ in } c' : t}$$

By (1) on e, there exist C_e, s_e such that $\Gamma_0^G;\Gamma_0;\Lambda \vdash_{cg} e : s_e \rightsquigarrow C_e$, $\theta_0 \vDash C_e$ and $s_e\theta_0 = t_e$.

Let $\Gamma'_0 = \Gamma_0[x : \alpha_x]$ and $\theta'_0 = \theta_0 \cup \{\alpha_x \mapsto t'_e\}$, where α_x is fresh. By induction on c', there exist C', t', θ such that $\Gamma_0^G;\Gamma'_0;\Lambda;A \vdash_{cg} c' : t' \rightsquigarrow C'$, $dom(\theta'_0) \sharp dom(\theta)$, $\theta'_0 \cup \theta \vDash C'$, and $t'(\theta'_0 \cup \theta) = t$.

By Rule (TG-LETVAR), we have

$$\Gamma_0^G;\Gamma_0;\Lambda;A \vdash_{cg} \textbf{letvar } x = e \textbf{ in } c : t' \rightsquigarrow C$$

where $C = C_e \cup C' \cup \{(\Lambda, s_e \leqslant \alpha_x)\}$. Let $\theta' = \theta \cup \{\alpha_x \mapsto t'_e\}$. It is clear that $dom(\theta_0) \sharp dom(\theta')$ and $\theta_0 \cup \theta' = \theta'_0 \cup \theta$. From the construction of θ' (*i.e.*, the proof of (2) and (3)), the type variables in $dom(\theta')$ are collected from the types of the functions and local variables, which are fresh. So we also have $\theta_0 \cup \theta' \vDash C_e$, $\Gamma_0(\theta_0 \cup \theta') = \Gamma_0\theta_0 = \Gamma$ and $\Gamma_0^G(\theta_0 \cup \theta') = \Gamma_0^G\theta_0 = \Gamma^G$. Moreover, $s_e(\theta_0 \cup \theta') = s_e\theta_0 = t_e \leqslant_\Lambda t'_e = \alpha_x(\theta_0 \cup \theta')$. Therefore, $\theta_0 \cup \theta' \vDash C$.

TT-CALL In this case we have $c \cong x := \textbf{call } B.f(\overline{e})$ and the following derivation

$$\frac{FT(B.f) = \overline{t_p} \xrightarrow{t_b} t_r \qquad \Gamma^G;\Gamma;\Lambda \vdash_{tr} \overline{e} : \overline{s_e} \qquad \overline{s_e} \leqslant_\Lambda \overline{\pi_{\Theta(A)}(t_p)} \qquad \pi_{\Theta(A)}(t_r) \leqslant_\Lambda \Gamma(x)}{\Gamma^G;\Gamma;\Lambda;A \vdash_{tr} x := \textbf{call } B.f(\overline{e}) : \Gamma(x) \sqcap \pi_{\Theta(A)}(t_b)}$$

By (3) on B.f, there exist $\overline{\alpha}, \beta, \gamma, C_f, \theta_f$ such that

$$\Gamma_0^G \vdash_{cg} B.f(\overline{x})\{\textbf{init } r = 0 \textbf{ in } \{c; \textbf{return } r\}\} : \overline{\alpha} \xrightarrow{\gamma} \beta \rightsquigarrow C_f,$$

$\theta_f \vDash C_f$ and $(\overline{\alpha} \xrightarrow{\gamma} \beta)\theta_f = \overline{t_p} \xrightarrow{t_b} t_r$. So we have $FT_C(B.f) = (\overline{\alpha} \xrightarrow{\gamma} \beta, C_f)$. Since α_xs are fresh, we can safely get $dom(\theta_0) \sharp dom(\theta_f)$. Therefore, $\theta_0 \cup \theta_f \vDash C_f$ and $(\overline{\alpha} \xrightarrow{\gamma} \beta)(\theta_0 \cup \theta_f) = \overline{t_p} \xrightarrow{t_b} t_r$.
By (1) on $\overline{e}$, there exist $\overline{s'_e}, \overline{C_e}$ such that $\Gamma_0^G;\Gamma_0;\Lambda;A \vdash_{cg} \overline{e} : \overline{s'_e} \rightsquigarrow \overline{C_e}$, $\theta_0 \vDash \overline{C_e}$, and $\overline{s'_e(\theta_0)} = \overline{s_e}$.
From the construction of θ_f (*i.e.*, the proof of (2) and (3)), the type variables in $dom(\theta_f)$ are collected from the types of functions and local variables, which are fresh. So we also have $\theta_0 \cup \theta_f \vDash \overline{C_e}$, $\overline{s'_e(\theta_0 \cup \theta_f)} = \overline{s_e}$, $\Gamma_0(\theta_0 \cup \theta_f) = \Gamma$ and $\Gamma_0^G(\theta_0 \cup \theta_f) = \Gamma^G$.
By Rule (TG-CALL), we have

$$\Gamma_0^G;\Gamma_0;\Lambda;A \vdash_{cg} x := \textbf{call } B.f(\overline{e}) : \Gamma_0(x) \sqcap \pi_{\Theta(A)}(\gamma) \rightsquigarrow C$$

where $C' = \{(\Lambda, \overline{s'_e} \leqslant \overline{\pi_{\Theta(A)}(\alpha)}), (\Lambda, \pi_{\Theta(A)}(\beta) \leqslant \Gamma_0(x))\}$ and $C = C_f \cup \bigcup \overline{C_e} \cup C'$. Since $dom(\theta_f)$ are fresh, we have

$$\begin{array}{l}\overline{s'_e(\theta_0 \cup \theta_f)} = \overline{s_e} \leqslant_\Lambda \overline{\pi_{\Theta(A)}(t_p)} = \overline{\pi_{\Theta(A)}(\alpha(\theta_0 \cup \theta_f))} \\ \pi_{\Theta(A)}(\beta(\theta_0 \cup \theta_f)) = \pi_{\Theta(A)}(t_r) \leqslant_\Lambda \Gamma(x) = \Gamma_0(\theta_0 \cup \theta_f)(x)\end{array}$$

So $(\theta_0 \cup \theta_f) \vDash C'$, and thus $(\theta_0 \cup \theta_f) \vDash C$. Thus the result follows.

TT-CP In this case we have $c \cong \textbf{test}(p)\ c_1\ \textbf{else}\ c_2$ and the following derivation

$$\frac{\Gamma^G;\Gamma;\Lambda :: \oplus p;A \vdash_{tr} c_1 : t_1 \qquad \Gamma^G;\Gamma;\Lambda :: \ominus p;A \vdash_{tr} c_2 : t_2}{\Gamma^G;\Gamma;\Lambda;A \vdash_{tr} \textbf{test}(p)\ c_1\ \textbf{else}\ c_2 : t_1 \triangleright_p t_2}$$

By induction on c_i, we have

$$\begin{array}{l}\exists C_1, s_1, \theta_1.\ \Gamma_0^G;\Gamma_0;\Lambda :: \oplus p;A \vdash_{cg} c_1 : s_1 \rightsquigarrow C_1, \theta_0 \cup \theta_1 \vDash C_1,\ dom(\theta_0) \sharp dom(\theta_1) \text{ and } s_1(\theta_0 \cup \theta_1) = t_1 \\ \exists C_2, s_2, \theta_2.\ \Gamma_0^G;\Gamma_0;\Lambda :: \ominus p;A \vdash_{cg} c_2 : s_2 \rightsquigarrow C_2, \theta_0 \cup \theta_2 \vDash C_2,\ dom(\theta_0) \sharp dom(\theta_2) \text{ and } s_2(\theta_0 \cup \theta_2) = t_2\end{array}$$

By Rule (TG-CP), we get

$$\Gamma_0^G;\Gamma_0;\Lambda;A \vdash_{cg} \textbf{test}(p)\ c_1\ \textbf{else}\ c_2 : s_1 \triangleright_p s_2 \rightsquigarrow C_1 \cup C_2$$

From the construction of θ_1 and θ_2 (*i.e.*, the proof of (2) and (3)), the type variables in $dom(\theta_1)$ and $dom(\theta_2)$ are collected from the types of functions and local variables, which are fresh. Moreover, if $\theta_1 \cap \theta_2 \neq \emptyset$, that is, they share some functions, then $\theta_1(\alpha_x) = \theta_2(\alpha_x)$ for all $\alpha_x \in dom(\theta_1) \cap dom(\theta_2)$. So we can safely get $(\theta_0 \cup \theta_1 \cup \theta_2) \vDash C_i$ and $s_i(\theta_0 \cup \theta_1 \cup \theta_2) = t_i$, $\Gamma_0(\theta_0 \cup \theta_1 \cup \theta_2) = \Gamma$, and $\Gamma_0^G(\theta_0 \cup \theta_1 \cup \theta_2) = \Gamma^G$. Therefore, $(\theta_0 \cup \theta_1 \cup \theta_2) \vDash C_1 \cup C_2$ and $(s_1 \triangleright_p s_2)(\theta_0 \cup \theta_1 \cup \theta_2) = t_1 \triangleright_p t_2$.

others By induction.

The proof of (c):

$$\frac{\Gamma^G; [\overline{x} : \overline{t_p}, r : t_r]; \epsilon; B \vdash_{tr} c : s \qquad t_b \leqslant s}{\Gamma^G \vdash_{tr} B.f(\overline{x})\{\textbf{init } r = 0 \textbf{ in } \{c; \textbf{return } r\}\} : \overline{t_p} \xrightarrow{t_b} t_r}$$

Let $\Gamma_0 = \{\overline{x} \mapsto \overline{\alpha}, r \mapsto \beta\}$, $\Gamma_0^G = \{x \mapsto \alpha_x \mid x \in dom(\Gamma^G)\}$ and $\theta_0 = \{\overline{\alpha} \mapsto \overline{t_p}, \beta \mapsto t_r, \gamma \mapsto t_b\} \cup \{\alpha_x \mapsto \Gamma^G(x) \mid x \in dom(\Gamma^G)\}$, where $\overline{\alpha}, \beta, \gamma, \alpha_x$ are fresh. By (2) on c, we have

$$\exists t, C_f, \theta.\ \Gamma_0^G; \Gamma_0; \epsilon; B \vdash_{cg} c : t \rightsquigarrow C_f,\ (\theta_0 \cup \theta) \vDash C_f, dom(\theta_0) \nsharp dom(\theta) \text{ and } t(\theta_0 \cup \theta) = s.$$

By Rule (TG-FUN), we have

$$\Gamma_0^G \vdash_{cg} B.f(\overline{x})\{\textbf{init } r = 0 \textbf{ in } \{c; \textbf{return } r\}\} : \overline{\alpha} \xrightarrow{\gamma} \beta \rightsquigarrow C_f$$

Finally, it is clear that $(\overline{\alpha} \xrightarrow{\gamma} \beta)(\theta_0 \cup \theta) = \overline{t_p} \xrightarrow{t_b} t_r$. □

Proof for **Lemma 3.12**:

Proof. By case analysis.

CD-CUP, CD-CAP Trivial.

CD-LAPP, CD-RAPP By definition of projection.

CD-SVAR, CD-SUB$_0$, CD-SUB$_1$ Trivial.

CD-MERGE$_0$, CD-MERGE$_1$ By Lemma 3.7.

CS-LU It is clear that $C \cup C' \vDash C$. For the other direction, we only need to prove $\{((\Lambda_l, t_l) \leqslant (\Lambda_1, \alpha)), ((\Lambda_2, \alpha) \leqslant (\Lambda_r, t_r))\} \vDash \{((\Lambda_l :: dif(\Lambda_1 :: \Lambda_2,\ \Lambda_1), t_l) \leqslant (\Lambda_r :: dif(\Lambda_1 :: \Lambda_2,\ \Lambda_2), t_r))\}$ if $\Delta(\Lambda_1 :: \Lambda_2)$. Assume that $\theta \vDash \{((\Lambda_l, t_l) \leqslant (\Lambda_1, \alpha)), ((\Lambda_2, \alpha) \leqslant (\Lambda_r, t_r))\}$, that is,

$$(t_l\theta) \cdot \Lambda_l \leqslant \theta(\alpha) \cdot \Lambda_1 \qquad \theta(\alpha) \cdot \Lambda_2 \leqslant (t_r\theta) \cdot \Lambda_r$$

Since $\Delta(\Lambda_1 :: \Lambda_2)$, by Lemmas 3.1 and 3.2, we have

$$(t_l\theta) \cdot (\Lambda_l :: dif(\Lambda_1 :: \Lambda_2,\ \Lambda_1)) \leqslant \theta(\alpha) \cdot (\Lambda_1 :: \Lambda_2) \quad \theta(\alpha) \cdot \Lambda_1 :: \Lambda_2 \leqslant (t_r\theta) \cdot (\Lambda_r :: dif(\Lambda_1 :: \Lambda_2,\ \Lambda_2))$$

which deduces

$$(t_l\theta) \cdot (\Lambda_l :: dif(\Lambda_1 :: \Lambda_2,\ \Lambda_1)) \leqslant (t_r\theta) \cdot (\Lambda_r :: dif(\Lambda_1 :: \Lambda_2,\ \Lambda_2))$$

that is, $\theta \vDash \{((\Lambda_l :: dif(\Lambda_1 :: \Lambda_2,\ \Lambda_1), t_l) \leqslant (\Lambda_r :: dif(\Lambda_1 :: \Lambda_2,\ \Lambda_2), t_r))\}$.

□

Before proving our unification is sound and complete, we show the *toType* is correct.

Lemma B.3 *Consider a type variable α with $C^L_\alpha = \{((\Lambda^l_i, t^l_i) \leqslant (\Lambda_i, \alpha))\}_{i\in I}$ and $C^U_\alpha = \{((\Lambda_j, \alpha) \leqslant (\Lambda^r_j, t^r_j))\}_{j\in J}$. Let t_α be the type constructed from $toType(C^L_\alpha, C^U_\alpha)$ if α is local or $toType^G(C^L_\alpha, C^U_\alpha)$ otherwise.*

(a) $\{\alpha \mapsto t_\alpha\} \vDash C^L_\alpha \cup C^U_\alpha$.

(b) If $\theta \vDash C^L_\alpha \cup C^U_\alpha$, *then there exists* θ' *such that* $dom(\theta') \sharp dom(\theta)$ *and* $(\theta \cup \theta')(\alpha) = t_\alpha(\theta \cup \theta')$.

Proof. The proof of (a): Let us consider any lower bound $((\Lambda^l_i, t^l_i) \leqslant (\Lambda_i, \alpha))$ and any permission set P:

$$\begin{aligned}
(t_\alpha \cdot \Lambda_i)(P) &= t_\alpha(P \cdot \Lambda_i) \\
&= \{P \mapsto (\bigsqcup_{(i,P')\in S^P_I} (t^l_i \cdot \Lambda^l_i)(P') \sqcup \alpha'(P)) \sqcap \bigsqcap_{(j,P')\in S^P_J} (t^r_j \cdot \Lambda^r_j)(P') \mid P \subseteq \mathbf{P}\}(P \cdot \Lambda_i) \\
&= (\bigsqcup_{(i,P')\in S^{P\cdot\Lambda_i}_I} (t^l_i \cdot \Lambda^l_i)(P') \sqcup \alpha'(P \cdot \Lambda_i)) \sqcap \bigsqcap_{(j,P')\in S^{P\cdot\Lambda_i}_J} (t^r_j \cdot \Lambda^r_j)(P') \\
&\geqslant \bigsqcup_{(i,P')\in S^{P\cdot\Lambda_i}_I} (t^l_i \cdot \Lambda^l_i)(P') \\
&\geqslant (t^l_i \cdot \Lambda^l_i)(P) \text{ (as } (i,P) \in S^{P\cdot\Lambda_i}_I)
\end{aligned}$$

So $\{\alpha \mapsto t_\alpha\} \vDash C^L_\alpha$. Similar to $\{\alpha \mapsto t_\alpha\} \vDash C^U_\alpha$.

The proof of (b): Let $\theta_0 = \{\alpha' \mapsto \theta(\alpha)\}$, where α' is a fresh variable introduced by *toType*. Clearly, we have $dom(\theta_0) \sharp dom(\theta)$. Since $\theta \vDash C^L_\alpha \cup C^U_\alpha$, we have $(t^l_i\theta) \cdot \Lambda^l_i \leqslant \theta(\alpha) \cdot \Lambda_i$ and $\theta(\alpha) \cdot \Lambda_j \leqslant (t^r_j\theta) \cdot \Lambda^r_j$ for all $i \in I$ and $j \in J$. So we have $((t^l_i\theta) \cdot \Lambda^l_i)(P) \leqslant (\theta(\alpha) \cdot \Lambda_i)(P)$ and $(\theta(\alpha) \cdot \Lambda_j)(P) \leqslant ((t^r_j\theta) \cdot \Lambda^r_j)(P)$ for any permission set P. In other words, for any permission set P, we have $((t^l_i\theta) \cdot \Lambda^l_i)(P') \leqslant \theta(\alpha)(P)$ if $P' \cdot \Lambda_i = P$ and $\theta(\alpha)(P) \leqslant ((t^r_j\theta) \cdot \Lambda^r_j)(P')$ if $P' \cdot \Lambda_j = P$. Moreover,

$$\begin{aligned}
t_\alpha(\theta \cup \theta_0)(P) &= \{P \mapsto (\bigsqcup_{(i,P')\in S^P_I} (t^l_i \cdot \Lambda^l_i)(P') \sqcup \alpha'(P)) \sqcap \bigsqcap_{(j,P')\in S^P_J} (t^r_j \cdot \Lambda^r_j)(P') \mid P \subseteq \mathbf{P}\}(\theta \cup \theta_0)(P) \\
&= (\bigsqcup_{(i,P')\in S^P_I} (t^l_i(\theta \cup \theta_0) \cdot \Lambda^l_i)(P') \sqcup ((\theta \cup \theta_0)(\alpha'))(P)) \sqcap \bigsqcap_{(j,P')\in S^P_J} (t^r_j(\theta \cup \theta_0) \cdot \Lambda^r_j)(P') \\
&= (\bigsqcup_{(i,P')\in S^P_I} (t^l_i\theta \cdot \Lambda^l_i)(P') \sqcup \theta(\alpha)(P)) \sqcap \bigsqcap_{(j,P')\in S^P_J} (t^r_j\theta \cdot \Lambda^r_j)(P') \\
&= \theta(\alpha)(P) \sqcap \bigsqcap_{(j,P')\in S^P_J} (t^r_j\theta \cdot \Lambda^r_j)(P') \text{ (as } (t^l_i\theta \cdot \Lambda^l_i)(P') \leqslant \theta(\alpha)(P) \text{ for } (i,P') \in S^P_I) \\
&= \theta(\alpha)(P) \text{ (as } \theta(\alpha)(P) \leqslant ((t^r_j\theta) \cdot \Lambda^r_j)(P') \text{ for } (j,P') \in S^P_J) \\
&= (\theta \cup \theta_0)(\alpha) \cdot \Lambda
\end{aligned}$$

Otherwise, α is global. The proof is similar to the case above and based on the fact that the types for global variables are invariance for all permission sets. □

Proof for **Lemma 3.13**:

Proof. By induction on $|C|$.

$|C| = 0$ Trivial.

$|C| > 0$ In this case we have $C = \{((\Lambda^l_i, t^l_i) \leqslant (\Lambda_i, \alpha))\}_{i\in I} \cup \{((\Lambda_j, \alpha) \leqslant (\Lambda^r_j, t^r_j))\}_{j\in J} \cup C'$, where $O(\alpha)$ is the greatest. Let t_α be the type constructed from the constraints on α (denoted as C_α) and C'' be the constraint set obtained by replacing in C' every occurrence of α by t_α. Assume assume $unify(C'') = \theta'$. Thus $\theta = \theta' \cup \{\alpha \mapsto t_\alpha\}$. First, by Lemma B.3, we have $\{\alpha \mapsto t_\alpha\} \vDash C_\alpha$, and thus $\theta \vDash C_\alpha$. Next, by induction, we have $\theta' \vDash C''$. Let's consider the constraints $\{((\Lambda^l_i, s^l_i) \leqslant$

$(\Lambda_i,\beta))\}_{i\in I}\cup\{((\Lambda_j,\beta)\leqslant(\Lambda_j^r,s_j^r))\}_{j\in J}\subseteq C'$ of any other variable β. Then we have $\{((\Lambda_i^l,s_i^l\{\alpha\mapsto t_\alpha\})\leqslant(\Lambda_i,\beta))\}_{i\in I}\cup\{((\Lambda_j,\beta)\leqslant(\Lambda_j^r,s_j^r\{\alpha\mapsto t_\alpha\}))\}_{j\in J}\subseteq C''$.

$$\begin{aligned}\theta(\beta)\cdot\Lambda_i &= \theta'(\beta)\cdot\Lambda_i && \text{(Apply }\theta\text{)}\\ &\geqslant((s_i^l\{\alpha\mapsto t_\alpha\})\theta')\cdot\Lambda_i^l && (\theta'\vDash C'')\\ &=(s_i^l(\theta'\cup\{\alpha\mapsto t_\alpha\}))\cdot\Lambda_i^l\end{aligned}$$

and

$$\begin{aligned}\theta(\beta)\cdot\Lambda_j &= \theta'(\beta)\cdot\Lambda_j && \text{(Apply }\theta\text{)}\\ &\leqslant((s_j^r\{\alpha\mapsto t_\alpha\})\theta')\cdot\Lambda_j^r && (\theta'\vDash C'')\\ &=(s_j^r(\theta'\cup\{\alpha\mapsto t_\alpha\}))\cdot\Lambda_j^r\end{aligned}$$

Therefore, the result follows.

□

Proof for **Lemma 3.14**:

Proof. We prove that the statement $\theta=\theta'\theta$, which deduces the result. The statement holds trivially when $|C|=0$.
When $|C|>0$, we have $C=\{((\Lambda_i^l,t_i^l)\leqslant(\Lambda_i,\alpha))\}_{i\in I}\cup\{((\Lambda_j,\alpha)\leqslant(\Lambda_j^r,t_j^r))\}_{j\in J}\cup C'$, where $O(\alpha)$ is the greatest. Let t_α be the type constructed from the constraints on α (denoted as C_α) and C_0 be the constraint set obtained by replacing in C' every occurrence of α by t_α. Since $\theta\vDash C$, we have $\theta\vDash C'$ and $\theta\vDash C_\alpha$, and thus $\theta\vDash C_0$. By induction on C_0, there exists θ_0' such that $unify(C_0)=\theta_0'$ and $\theta=\theta_0'\theta$. According to the function $unify$, we get $unify(C)=\theta_0'\cup\{\alpha\mapsto t_\alpha\}=\theta'$. For any $\beta\notin dom(\theta')$, clearly $\beta(\theta'\theta)=\beta\theta$. Considering α, since $\theta\vDash C_\alpha$, by Lemma B.3, we have $\alpha\theta=t_\alpha\theta$, and thus $\alpha\theta=\alpha(\theta'\theta)$. While for any other variable $\beta\in dom(\theta')$, we have $\beta(\theta'\theta)=\beta((\theta_0'\cup\{\alpha\mapsto t_\alpha\})\theta)=\beta(\theta_0'\theta)=\beta\theta$. □

Appendix C. Proofs for Representation

Proof for **Lemma 4.1**:

Proof. Straightforward from Definition 4.4. □

Proof for **Lemma 4.2**:

Proof. For the proof of (1), according to Lemma 4.1, we have

$$\mathcal{R}(t\uparrow_p)(P)=t\uparrow_p(P)=t(P\cup\{p\})=\mathcal{R}(t)(P\cup\{p\}).$$

Similar to the proof of (2). □

Proof for **Lemma 4.3**:

Proof. Similar to Lemma 4.2 □

Proof for **Lemma 4.4**:

Proof. Similar to Lemma 4.2 □

Proof for **Lemma 4.5**:

Proof. Straightforward From Lemma 4.3. □

Proof for **Lemma 4.6**:

Proof. For the proof of (a), we first prove the statement: for any type presentation r, any permission p, and any permission set P, PROMOTION-RECURSE$(r,p)(P) = r(P \cup \{p\})$. The proof proceeds by induction on r.

r **is sink** In this case, PROMOTION-RECURSE$(r,p) = r$ and $r(P) = r(P \cup \{p\}) = r.val$. So the result holds.

$r.v =$ **INDEX**(p) In this case, PROMOTION-RECURSE$(r,p) = (r.v, r.h, r.h)$ and $r.h.v >$ INDEX(p). Then no matter $p \in P$ or not, we have

$$\begin{aligned} \text{PROMOTION-RECURSE}(r,p)(P) &= r.h(P) \\ &= r.h(P \cup \{p\}) \quad (p \text{ not affect the output}) \\ &= r(P \cup \{p\}) \end{aligned}$$

$r.v >$ **INDEX**(p) In this case, PROMOTION-RECURSE$(r,p) = r$ and p does not affect the output. Therefore, PROMOTION$(r,p)(P) = r(P) = r(P \cup \{p\})$.

$r.v <$ **INDEX**(p) In this case, PROMOTION-RECURSE$(r,p) = (r.v, l, h)$, where $l =$ PROMOTION-RECURSE$(r.l,p)$ and $h =$ PROMOTION-RECURSE$(r.h,p)$. Assume $P[r.v] = 0$, then we have

$$\begin{aligned} \text{PROMOTION-RECURSE}(r,p)(P) &= \text{PROMOTION-RECURSE}(r.l,p)(P) \quad (P[r.v] = 0) \\ &= r.l(P \cup \{p\}) \quad (\text{by induction on } r.l) \\ &= r(P \cup \{p\}) \quad ((P \cup \{p\})[r.v] = 0) \end{aligned}$$

Similar for $P[r.v] = 1$, the result holds as well.

From existing work on BDD [20, 21], we can get the procedure REDUCE( ) is correct. Considering any permission set P, we have

$$\begin{aligned} \text{PROMOTION}(\mathcal{R}(t),p)(P) &= \text{REDUCE}(\text{PROMOTION-RECURSE}(\mathcal{R}(t),p))(P) \\ &= \text{PROMOTION-RECURSE}(\mathcal{R}(t),p)(P) \\ &= \mathcal{R}(t)(P \cup \{p\}) \\ &= \mathcal{R}(t \uparrow_p) \quad (\textit{Lemma } 4.2) \end{aligned}$$

Similar to the proof of (b). □

Proof for **Lemma 4.7**:

Proof. we first prove the statement: given two type presentations r_1, r_2, a permission p, and a permission set P, if $p \in P$, then MERGE-RECURSE$(r_1, r_2, p)(P) = r_1(P)$. The proof proceeds by induction on r_1 and r_2.

r_1, r_2 **are sink** In this case, MERGE-RECURSE$(r_1, r_2, p) = (p, r_2, r_1)$. As $p \in P$, we have $(p, r_2, r_1)(P) = r_1(P)$. So the result holds.

r_1 **is sink or** $r_1.v > r_2.v$ There are three subcases.

Subcase (1) where $r_2.v >$ INDEX(p): we have MERGE-RECURSE$(r_1, r_2, p) = (p, r_2, r_1)$. Similar to the first case.

Subcase (2) where $r_2.v =$ INDEX(p): we have MERGE-RECURSE$(r_1, r_2, p) = (r_2.v, r_2.l, r_1)$. Similar to the first case.

Subcase (3) where $r_2.v <$ INDEX(p): we have MERGE-RECURSE$(r_1, r_2, p) = (r_2.v, l, h)$, where $l =$ MERGE-RECURSE$(r_1, r_2.l, p)$ and $h =$ MERGE-RECURSE$(r_1, r_2.h, p)$. By induction on r_1 and $r_2.h$, we have $h(P) = r_1(P)$. Therefore, $(r_2.v, l, h)(P) = h(P) = r_1(P)$. So the result holds.

r_2 **is sink or** $r_1.v < r_2.v$ There are three subcases.

Subcase (1) where $r_1.v >$ INDEX(p): we have MERGE-RECURSE$(r_1, r_2, p) = (p, r_2, r_1)$. Similar to the first case.

Subcase (2) where $r_1.v =$ INDEX(p): we have MERGE-RECURSE$(r_1, r_2, p) = (r_1.v, r_2, r_1.h)$. As $p \in P$, namely, $P[r_1.v] = 1$, we have $(r_1.v, r_2, r_1.h)(P) = r_1.h(P) = r_1(P)$.

Subcase (3) where $r_1.v <$ INDEX(p): we have MERGE-RECURSE$(r_1, r_2, p) = (r_1.v, l, h)$, where $l =$ MERGE-RECURSE$(r_1.l, r_2, p)$ and $h =$ MERGE-RECURSE$(r_1.h, r_2, p)$. By induction on $r_1.h$ and r_2, we have $h(P) = r_1.h(P)$. Moreover, as $p \in P$, that is, $P[r_1.v] = 1$, we have $(r_1.v, l, h)(P) = h(P) = r_1.h(P) = r_1(P)$.

$r_1.v = r_2.v$ There are three subcases.

Subcase (1) where $r_1.v >$ INDEX(p): we have MERGE-RECURSE$(r_1, r_2, p) = (p, r_2, r_1)$. Similar to the first case.

Subcase (2) where $r_1.v =$ INDEX(p): we have MERGE-RECURSE$(r_1, r_2, p) = (r_1.v, r_2.l, r_1.h)$. Similar to Subcase (2) of the third case.

Subcase (3) where $r_1.v <$ INDEX(p): we have MERGE-RECURSE$(r_1, r_2, p) = (r_1.v, l, h)$, where $l =$ MERGE-RECURSE$(r_1.l, r_2.l, p)$ and $h =$ MERGE-RECURSE$(r_1.h, r_2.h, p)$. Similar to Subcase (3) of the third case.

From existing work on BDD [20, 21], we can get the procedure REDUCE( ) is correct. Considering any permission set P such that $p \in P$, we have

$$\begin{aligned}
\text{MERGE}(\mathcal{R}(t_1), \mathcal{R}(t_2), p)(P) &= \text{REDUCE}(\text{MERGE-RECURSE}(\mathcal{R}(t_1), \mathcal{R}(t_2), p))(P) \\
&= \text{MERGE-RECURSE}(\mathcal{R}(t_1), \mathcal{R}(t_2), p)(P) \\
&= \mathcal{R}(t_1)(P) \\
&= \mathcal{R}(t_1 \triangleright_p t_2)(P) \quad (\textit{Lemma } 4.4)
\end{aligned}$$

Likewise, we can get MERGE$(\mathcal{R}(t_1), \mathcal{R}(t_2), p)(P) = \mathcal{R}(t_1 \triangleright_p t_2)(P)$ if $p \notin P$. Therefore, the result follows. □