Zhang ZP, Fu M, Feng XY. A lightweight dynamic enforcement of privacy protection for Android. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 34(4): 901–923 July 2019. DOI 10.1007/s11390-019-1949-1

A Lightweight Dynamic Enforcement of Privacy Protection for Android

Zi-Peng Zhang[1], Ming Fu[2], and Xin-Yu Feng[3,*], *Member*, *CCF*, *ACM*

[1]*School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China*
[2]*OS Kernel Laboratory, Huawei Technologies Co., Ltd., Shanghai 200135, China*
[3]*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China*

E-mail: zhangzpp@mail.ustc.edu.cn; ming.fu@huawei.com; xyfeng@nju.edu.cn

Received August 29, 2018; revised April 15, 2019.

Abstract Inter-process communication (IPC) provides a message passing mechanism for information exchange between applications. It has been long believed that IPCs can be abused by malware writers to launch collusive information leak using two or more applications. Much work on privacy protection focuses on the simple information leak caused by the individual applications and lacks effective approaches to preventing the collusive information leak caused by IPCs between multiple processes. In this paper, we propose a hybrid approach to prevent the collusive information leak based on information flow control. Our approach combines static information flow analysis and dynamic runtime checking together. Information leak caused by individual processes is prevented through static information flow control, and dynamic checking is done at runtime to prevent the collusive information leak. Such a combination may effectively reduce the runtime overhead of pure dynamic checking, and reduce false-alarms in pure static analysis. We develop this approach based on an abstract and simplified programming model, and formalize a novel definition of the leak-freedom property as our target security property. A simulation-based proof technique is used to prove that our approach is able to guarantee leak-freedom. All proofs are mechanized in Coq.

Keywords privacy protection, dynamic runtime checking, static information flow control, Android, verification

1 Introduction

Sensitive information stored in mobile devices such as mobile phones is under the threat of being stolen. Android, as one of the most popular mobile operating systems, provides a permission-based access control mechanism for privacy protection. Each application requires the permissions to access the corresponding information when it is installed. The information can be accessed in the subsequent execution only if the permission is granted by the user at the installation time.

There are several problems with this access control mechanism. First, it is sometimes too coarse-grained to use in practice. An untrusted application may need certain private information for some useful functionality. It should be safe to grant the corresponding permission to the application if the information is used only locally and is never sent out of the device. One such example is the input method applications, which may need to read the contact book to generate better hints of names for user inputs. The access to the contact book should be permitted as long as it cannot be sent out. However, based on the existing mechanism, the users have to either deny the permission and lose the improved name hint functionality, or simply grant the permission and have no further control on how the contact book is propagated.

Second, it is well known that access control cannot effectively prevent information leak, especially the leak through the collusion of multiple applications. Consider the program shown in Fig.1. It consists of two processes `t0` and `t1`. Process `t0` sends a non-confidential constant

Regular Paper
This work was supported in part by the National Natural Science Foundation of China under Grant No. 61632005.
*Corresponding Author
©2019 Springer Science + Business Media, LLC & Science Press, China

value 1 to `t1`. Depending on the content of the incoming message, `t1` decides whether to send out the contact book (i.e., CB) or not. Here we use *out*(y) to represent an output operation that sends the value of y out of the device. The *send* and *recv* primitives are used for inter-process communication (IPC). In this case, although `t0` may have no permissions to access and send out the contact book, it could do so by sending a message to `t1` who has the permissions.

```
                      | t1. recv(t,x);
                      |     if (x==1)
t0. x := 1;           |     then y := get(CB);
    send(t1,x);       |          out(y);
                      |     else skip;
```

Fig.1. Example of information leak via collusion of processes.

To address the first problem, we refine the permissions into two kinds: the access permissions and the send permissions. The access permissions are similar to the permissions of the Android system, which specify the information that the application is allowed to access. The send permissions specify the information that the application can send out. In the example of the input method application, if the user wants to use the name hint functionality but does not want to leak his or her contact book, the application should be granted the access permission of {contactbook} only and the send permission should be empty (i.e., {}). Because two applications are allowed to communicate with each other, to enforce the send permission, the security mechanism should prevent the application not only from sending out the privacy without the proper send permission by itself, but also from sending it out with the help of another authorized application.

The second problem, however, is much more challenging to solve. To prevent the privacy leak caused by the inter-process communication, the traditional static analysis[1−8] regards each message-sending point as a leak point, which is overly restrictive and prevents benign cooperation among the applications as well. Consider the program shown in Fig.2. Process `t0` sends the privacy contact book (i.e., CB) to process `t1` and process `t1` does not send it out. Therefore this communication is benign and should not be prevented. The work[9] analyzes the possible communications among applications statically, and allows the communications that will definitely not lead to the privacy leak. But it is still conservative when one of the receivers may leak the privacy. Since the concrete receiver which may receive the message cannot be determined statically, one safe method is to prevent the communications with all receivers.

```
t0. x := get(CB);  |
    send(t1,x);    | t1. recv(t,y);
```

Fig.2. Example of benign cooperation between processes.

The work[2] based on the dynamic taint analysis considers the cooperation of applications, and tracks all the information flow for each instruction at runtime, including the communications among processes. But it only prevents confidential contents from being transmitted between the applications. It does not prevent the leak shown in Fig.1, where the message contains only a non-confidential constant value. Dynamic checking also introduces high runtime overhead, especially if we have to track even non-confidential IPC messages to prevent this kind of leak.

We propose a hybrid approach that combines static analysis and dynamic checking to achieve less false positives (than the overly restrictive static approaches) and lower runtime overhead (than the purely dynamic approaches). To analyze communications more precisely, we check each communication at runtime and detect information leak through collusion of multiple processes. To reduce the runtime overhead, the individual process is checked statically, where privacy leak caused by the process directly is prevented. The communications are the only program points checked dynamically.

The overall architecture of our approach is shown in Fig.3. The application developer provides the Android package (APK) and the security policy consisting of the access permissions (*gap*) and the send permissions (*gsp*). The application package (APK) and the security policy are sent to the static checker to detect information leak and to generate the security labels for the outgoing IPC messages. The generated labels record the privacy contained in the messages, and they are fed to the translation phase to rewrite the message sending primitives for dynamic checking. The runtime checking codes are instrumented in the rewritten primitives and determine whether the communications cause the privacy leak based on the message labels and the policies of both sides.

To prove that our approach prevents the privacy leak effectively, especially the collusive information leak, we formally study the correctness of our hybrid mechanism based on a simplified programming language, and give a sound proof. The previous non-interference properties for concurrent programs[10−15] cannot be used as the security property for Android

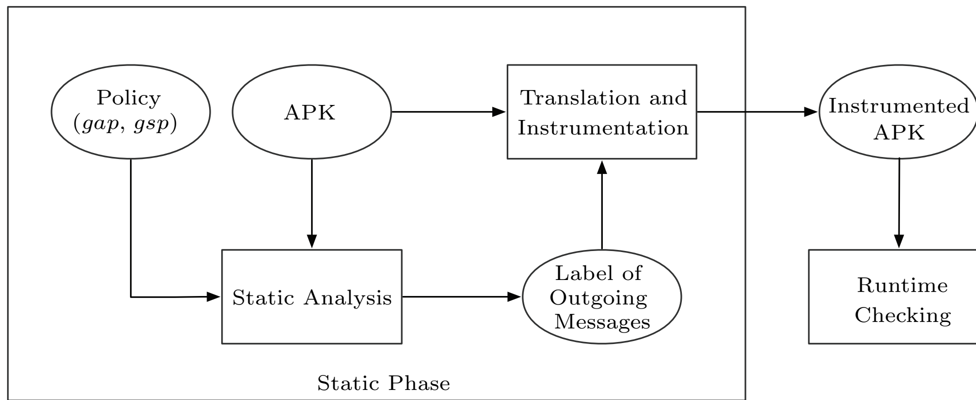


Fig.3. Overall architecture of proposed approach.

applications. Consider the example in Fig.1. Assume that process `t0` is not permitted to send any privacy out. The traditional non-interference properties classify `t0` as a secure program, because it does not output any privacy. But process `t0` is insecure, leaking privacy with the help of process `t1`. We propose the leak-freedom property as the security property to prevent the privacy leak caused by communications among applications.

Our work makes the following contributions.

• We propose a novel method to prevent collusive information leak caused by IPCs between multiple processes. A new class of permissions (i.e., the send permissions) is introduced into the traditional permission-based access control mechanism for effectively preventing collusive information leak. It also combines static information flow control and dynamic checking; therefore it may potentially reduce the number of false alarms in purely static approaches, and reduce the runtime overhead of purely dynamic approaches.

• We propose an abstract core language for the programming model with IPC mechanisms, which can be applied in modelling Android and other systems using IPCs. The small-step operational semantics given to the language serves as the basis to formally study the soundness of our approach.

• We introduce a new security property, leak-freedom, a non-interference-like property[16,17] that guarantees that no information is leaked through application collusion. The soundness is formally defined and proved in Coq①, and we have written around 15 500 lines of Coq script for the soundness proof in Coq 8.7.0, which are available online②. The correctness of the proof can be checked by the Coq proof checker, and it guarantees that no error is introduced.

The rest of this paper is organized as follows. We first analyse the possible information leak under the current access control mechanism in Android in Section 2. In Section 3, we present a simplified core language as an abstract programming model for Android applications. In Section 4 and Section 5 we show the static analysis, the program translation, and the dynamic checking to enforce the policy. We give an input method application③ as an example in Section 6 to illustrate how our mechanism works. In Section 7, we give the definition of the leak-freedom property. In Section 8, we formalize the soundness of our approach. In Section 9, we give some details about the implementation. Finally, we compare our method with related work in Section 10 and conclude the paper in Section 11.

2 Possible Information Leak in Android

There are two types of information leak in Android systems, the direct leak and the cooperation leak. The direct leak means that an application reads the private information and then sends it out by itself, and the cooperation leak means that an application sends the private data out through another application.

Fig.4(a) shows the direct leak. The application App requests the permissions to access both the contact book and the Internet. It can read the contact book from the contact book provider and then forwards it to

①The Coq Proof Assistant. https://coq.inria.fr, May 2019.

②https://formal-android-security.github.io, May 2019.

③Input Method Wiki. https://en.wikipedia.org/wiki/Input_method, May 2019.

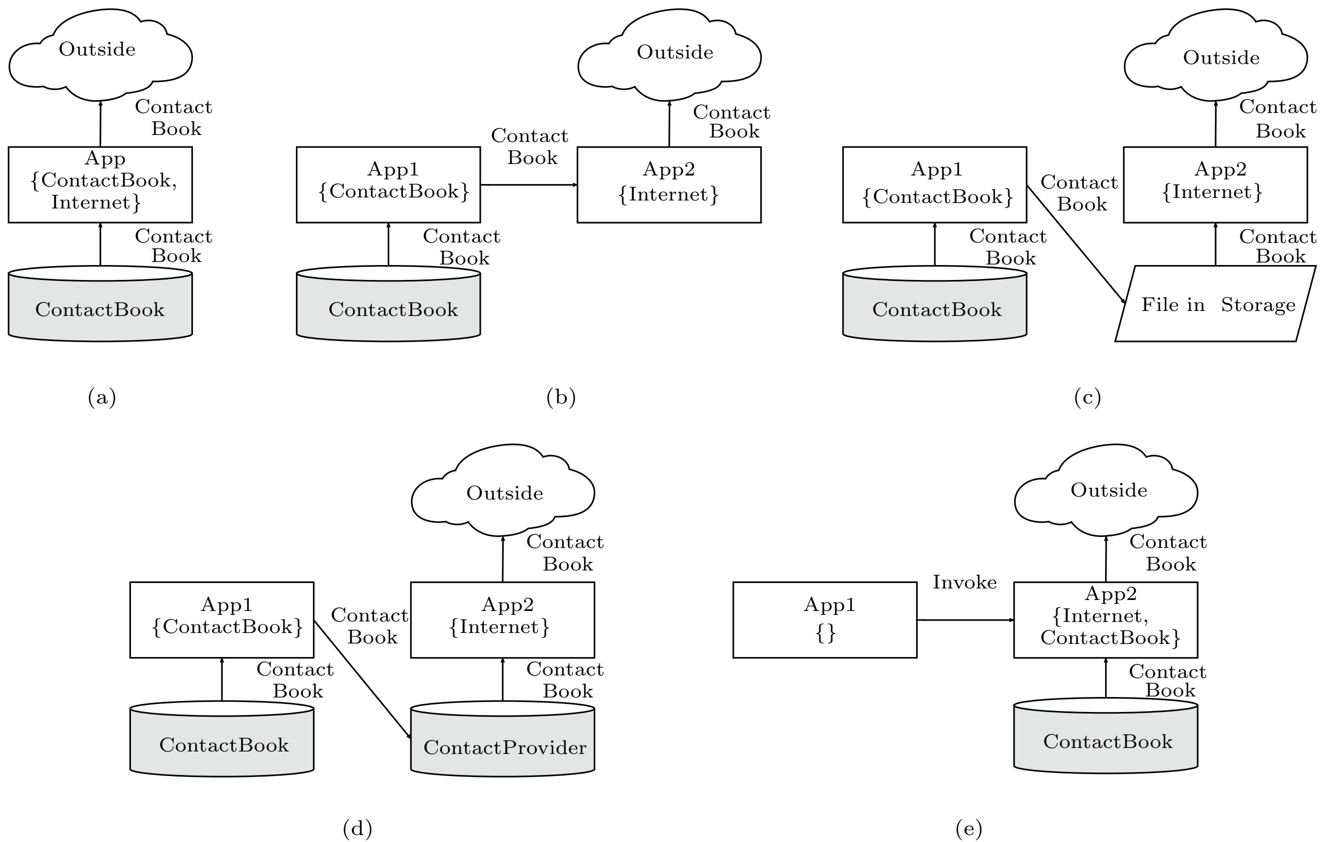


Fig.4. Possible information leak. (a) Individual leak. (b) Collusive leak by IPC. (c) Collusive leak by files. (d) Collusive leak by content provider. (d) Collusive by invocation.

the Internet. The following program shows this case.

```
x := get(CB);
out(x);
```

Here we use the *get* and *put* primitives to model the system calls to read and update the shared resources, respectively. The *out* primitive is used to model the system calls to send the information outside of the device, such as the Internet access. As a concrete example for this case, an input method application may need the contact book to generate hints for name inputs. It may also access the Internet to update its word database to add the latest popular terms. Therefore it requires permissions to access both resources, which then could enable it to send the contact book to the Internet.

Figs.4(b)–4(e) demonstrate the cooperation leak. In Fig.4(b), App1 requests the permission to access the contact book. App2 requests the permission to access the Internet. Each application alone cannot leak anything about the contact book, but App1 and App2 can work together to send the contact book out. For example, App1 reads the contact book first and then sends it to App2. App2 receives the message containing the contact book, and then sends it to the Internet. The following program shows this case.

```
t0. x := get(CB);    |  t1. recv(t,x);
    send(t1,x);      |      out(x);
```

Here the *send* and *recv* primitives are used to model the systems calls, which performs the inter-process communication. “`t0` ‖ `t1`” means that processes `t0` and `t1` run in parallel.

Similar to Fig.4(b), in Fig.4(c) and Fig.4(d), App1 reads the contact book first and then stores it into a file in the storage (in Fig.4(c)) or into a content provider (in Fig.4(d)), where a content provider in Android can be viewed as a database owned by some applications and can be shared with others. App2 reads the file or the content provider, and then sends it to the Internet. The following program shows this case.

```
t0. x := get(CB);    |  t1. y := get(s);
    put(s,x);        |      out(y);
```

Here s is a shared variable which can be accessed by all the processes and it is used to model the file in the

storage or the content provider. Process `t1` sends out whatever it reads from s.

In Fig.4(e), App2 is a trusted application and it can send the contact book to the Internet freely. App1 is untrusted and it has no permissions to access the contact book or the Internet. However, App1 can leak the contact book out by invoking the service of App2, if App2 exposes an interface for such a service. The example program has already been shown in Fig.1.

3 Core Language

In this section, we define a core language as a simplified abstract programming model for Android. The language serves as the basis to formally study the soundness of our approach.

3.1 Permissions

The permissions of our model extend those for Android. In addition to permissions to access the confidential resources, a new class of permissions called *send* permissions is introduced. It is used to specify the set of private information which can be sent outside. Compared with Android, we only care about permissions to access private information, such as the contact book, text messages and location coordinates. The permissions to perform other critical actions, such as accessing the Internet, modifying the system setting and performing I/O operations over NFC, are beyond the scope of the discussion.

The confidential information in Android can be accessed by any applications with permissions. Therefore we use the shared variables to model the privacy, each representing a kind of privacy. The permissions are represented as a set of shared variables that an application can access or send out. The permissions are specified by the programmers in the form of security labels. A label L, shown in Fig.5, is a set of shared variables. Here $\mathcal{P}(S)$ represents the powerset of the set S. The set of labels forms a security lattice, with the top element $\top$ being the whole set of *Vars* and the partial order $\sqsubseteq$. $L_1 \sqsubseteq L_2$ means that the label L_1 is lower than L_2. $L_1 \sqcup L_2$ is the least upper bound of L_1 and L_2. Two labels are equal, i.e., $L_1 = L_2$, if and only if L_1 is less than L_2 and L_2 is less than L_1.

Every process is associated with a security policy θ, which is specified by the programmer and consists of two labels, *gap* and *gsp*. The access permission *gap* specifies the set of shared variables that can be accessed by the process. It is similar to the permissions granted to Android applications. The access permission *gap* is checked for *get* and *put* primitives.

The *gap* permission alone, however, is too coarse-grained to use in practice. As shown in Fig.4(a) in Section 2, if an input method application requires permissions to access both the contact book and the Internet, then *gap* alone cannot prevent the application from sending the contact book out. The send permission *gsp* is introduced to address this problem. It specifies the set of shared variables that can be sent out using the *out* primitive. For the input method application in Fig.4(a), we can prevent the leak of the contact book by setting *gsp* to empty (i.e., $\{\}$).

For each process, its send permission *gsp* is usually less than or equal to its access permission *gap*, i.e., $gsp \sqsubseteq gap$, that is, a process needs to first have the permission to access a variable if it wants to send it out.

3.2 Programming Language

The syntax of the language is given in Fig.6. A program p consists of multiple processes, each with a policy θ declared by the programmer. Here we assume that all variables are local, except those accessed by $get(x)$ and $put(x, e)$ primitives, which read and update the shared variable x respectively. Note that the variables in e in the $put(x, e)$ primitive need to be local.

Inter-process communication is done using the send and receive primitives. The send primitive $lsend(e_1, e_2)$ sends a message e_2 to the process identified by e_1. Primitive $lrecv(x, y)$ receives a message from another process, and stores the identifier of the sender and the message into variables x and y respectively. Here the

(Labels) $L, gap, gsp \in \mathcal{P}(Vars)$ (Policy) $\theta ::= (gap, gsp)$

$$L_1 \sqsubseteq L_2 \text{ iff } L_1 \subseteq L_2 \vee L_2 = \top, \qquad L_1 \not\sqsubseteq L_2 \text{ iff } L_1 \not\subseteq L_2 \wedge L_2 \neq \top,$$

$$L_1 \sqcup L_2 = \begin{cases} \top, & \text{if } L_1 = \top \text{ or } L_2 = \top, \\ L_1 \cup L_2, & \text{otherwise.} \end{cases} \qquad L_1 = L_2 \text{ iff } L_1 \sqsubseteq L_2 \wedge L_2 \sqsubseteq L_1.$$

Fig.5. Label and permission.

$$
\begin{array}{rcl}
(Expr)\ e & ::= & x \mid n \mid e+e \mid \ldots \\
(Bexp)\ b & ::= & \text{true} \mid \text{false} \mid e=e \mid e<e \mid \text{not}\ b \mid \ldots \\
(Prims)\ o & ::= & x := e \mid lsend(e,e) \mid lrecv(x,y) \mid out(e) \mid hrecv(x,y) \mid x := get(y) \mid put(x,e) \\
& \mid & skip \\
(Cmd)\ c & ::= & o \mid hsend(e,e) \mid c_1;\ c_2 \mid while\ b\ do\ c \mid if\ b\ then\ c_1\ else\ c_2 \\
(Prog)\ p & ::= & \theta_1; c_1\ \parallel\ \ldots\ \parallel\ \theta_n; c_n \\
(AnnoCmd)\ \mathcal{C} & ::= & o \mid end \mid hsend(e,e,L) \mid \mathcal{C}_1;\ \mathcal{C}_2 \mid while\ b\ do\ \mathcal{C} \mid if\ b\ then\ \mathcal{C}_1\ else\ \mathcal{C}_2 \\
(AnnoProg)\ \mathcal{P} & ::= & \mathcal{C}_1\ \parallel\ \ldots\ \parallel\ \mathcal{C}_n
\end{array}
$$

Fig.6. Abstract language for the application.

messages cannot contain any confidential information; otherwise they need to be sent using a different pair of primitives called *hsend* and *hrecv*. Note that the *hsend* primitive is defined separately from all the other primitives o in Fig.6. This is because we may first rewrite the $hsend(e_1, e_2)$ primitive into the form of $hsend(e_1, e_2, L)$ before executing the code, where L is the security label inferred for the send operation in the static analysis phase. The code rewriting will be explained later in Subsection 4.4.2. The annotated commands and programs are represented as $\mathcal{C}$ and $\mathcal{P}$ respectively. Note that the send/receive primitives are asynchronous (i.e., non-blocking).

The $out(x)$ primitive sends the value of x to the external world (e.g., the Internet). It is different from the send and receive primitives for IPC.

The *end* command is an auxiliary command to help the proof of the soundness, which marks the finish boundaries of conditional blocks, such as the blocks of *if* and *while* (explained in Subsection 4.4.2). The semantics of *end* is explained in Section 8 in detail.

How Is the Language Related to Android Programming Model? We use the core language as a simplified model for Android programming. Each Android application can be represented as a process in the program. Shared resources, such as the contact book, can be modeled as shared variables, and the *get* and *put* primitives can be viewed as the system calls to access the shared resources.

The primary mechanism for Android component interaction is an intent, which is simply a message object containing a destination component address and data. Intent can be used to transfer data to another application, and to invoke a service provided by another application, etc. We use the send/receive message passing primitives to model the intent-based interaction. As we explain above, we provide two sets of primitives for non-confidential and confidential messages respectively. The *hsend* and *hrecv* primitives are checked more conservatively. In our implementation for Android, inter-process communication is modeled as *hsend*/*hrecv* by default, but the programmers can provide annotations in programs to indicate that only non-confidential messages are sent and the *lsend*/*lrecv* primitives should be used instead.

The *out* primitive models all kinds of output system calls and services in Android, such as Internet access or sending of text messages. It is the only way to leak information outside of the mobile devices.

How Is the Language Related to the Architecture in Fig.3? We use the command c to model the Android application and θ to represent the policy. The whole program p contains both APK and the policy. The annotated command $\mathcal{C}$ represents the instrumented APK. In the implementation, the dynamic checking is performed by instrumented codes. But for the sake of convenience of discussion, we integrate the dynamic checking into the operational semantic of the annotated command $\mathcal{C}$ in our model.

3.3 Operational Semantics

We now present a small-step operational semantics for our language. The state model is shown in Fig.7. The program state Σ consists of the set δ of the private store s and a shared state σ. Each store s_i can be accessed exclusively by the corresponding thread t_i, and we model the private store s as a mapping from private variables to their values. The shared state σ can be accessed by any process, and it contains a shared store $\mathbb{s}$, two message queues, Q and $\mathbb{Q}$, for confidential and non-confidential message respectively, and an output state o. The shared store $\mathbb{s}$ maps shared variables to their values. The output state o is a sequence of integers sent by the *out* primitive.

We give some key operational semantics rules in Fig.8. The execution of the program follows the standard interleaving semantics, as shown by the PARALLEL rule. The relation $\Theta \vdash (\mathcal{C}_1, (s_1, \sigma_1)) \rightsquigarrow_t (\mathcal{C}_2, (s_2, \sigma_2))$

(PId)	t	$\in$ Nat		$(Store)$	$s, \mathbb{s}$	$\in$ $Vars \to Nat$
$(PrivStateSet)$	δ	$::= \{t_1 \rightsquigarrow s_1, \ldots, t_n \rightsquigarrow s_n\}$		$(MsgQueue)$	$q, \mathbb{q}$	$::= \text{nil} \mid (t, n) :: q$
$(MsgQueueSet)$	$Q, \mathbb{Q}$	$::= \{t_1 \rightsquigarrow q_1, \ldots, t_n \rightsquigarrow q_n\}$		$(OutState)$	o	$::= \text{nil} \mid n :: o$
$(SharedState)$	σ	$::= (\mathbb{s}, Q, \mathbb{Q}, o)$		$(State)$	Σ	$::= (\delta, \sigma)$
$(PolicySet)$	Θ	$::= \{t_1 \rightsquigarrow \theta_1, \ldots, t_n \rightsquigarrow \theta_n\}$				

Fig.7. State model.

$$\frac{\delta(i) = s \quad \Theta \vdash (\mathcal{C}_i, (s, \sigma)) \rightsquigarrow_i (\mathcal{C}_i', (s', \sigma'))}{\Theta \vdash (\mathcal{C}_1 \parallel \cdots \parallel \mathcal{C}_i \parallel \cdots \parallel \mathcal{C}_n, (\delta, \sigma)) \to (\mathcal{C}_1 \parallel \cdots \parallel \mathcal{C}_i' \parallel \cdots \parallel \mathcal{C}_n, (\delta\{i \rightsquigarrow s'\}, \sigma'))} \text{ (PARALLEL)}$$

$$\frac{\begin{array}{c}[\![e_1]\!]_s = i \quad [\![e_2]\!]_s = n \quad \sigma = (\mathbb{s}, Q, \mathbb{Q}, o) \\ q_i = Q(i) \quad q_i' = q_i +\!\!+ [(t, n)] \quad Q' = Q\{i \rightsquigarrow q_i'\} \quad \sigma' = (\mathbb{s}, Q', \mathbb{Q}, o) \\ L \sqsubseteq \Theta(i).gap \quad \Theta(i).gsp \sqsubseteq \Theta(t).gsp\end{array}}{\Theta \vdash (hsend(e_1, e_2, L), (s, \sigma)) \rightsquigarrow_t (skip, (s, \sigma'))} \text{ (HSEND-SUCC)}$$

$$\frac{[\![e_1]\!]_s = i \quad \sigma = (\mathbb{s}, Q, \mathbb{Q}, o) \quad i \notin \text{dom}(Q)}{\Theta \vdash (hsend(e_1, e_2, L), (s, \sigma)) \rightsquigarrow_t (skip, (s, \sigma))} \text{ (HSEND-FAIL)}$$

$$\frac{\begin{array}{c}[\![e_1]\!]_s = i \quad [\![e_2]\!]_s = n \quad \sigma = (\mathbb{s}, Q, \mathbb{Q}, o) \\ \mathbb{q}_i = \mathbb{Q}(i) \quad \mathbb{q}_i' = \mathbb{q}_i +\!\!+ [(t, n)] \quad \mathbb{Q}' = \mathbb{Q}\{i \rightsquigarrow \mathbb{q}'\} \quad \sigma' = (\mathbb{s}, Q, \mathbb{Q}', o) \\ \Theta(i).gsp \sqsubseteq \Theta(t).gsp\end{array}}{\Theta \vdash (lsend(e_1, e_2), (s, \sigma)) \rightsquigarrow_t (skip, (s, \sigma'))} \text{ (LSEND-SUCC)}$$

$$\frac{[\![e_1]\!]_s = i \quad \sigma = (\mathbb{s}, Q, \mathbb{Q}, o) \quad i \notin \text{dom}(\mathbb{Q})}{\Theta \vdash (lsend(e_1, e_2), (s, \sigma)) \rightsquigarrow_t (skip, (s, \sigma))} \text{ (LSEND-FAIL)}$$

$$\frac{\begin{array}{c}\sigma = (\mathbb{s}, Q, \mathbb{Q}, o) \quad Q(t) = (i, n) :: q' \\ s' = s\{x \rightsquigarrow i, y \rightsquigarrow n\} \quad Q' = Q\{t \rightsquigarrow q'\} \quad \sigma' = (\mathbb{s}, Q', \mathbb{Q}, o)\end{array}}{\Theta \vdash (hrecv(x, y), (s, \sigma)) \rightsquigarrow_t (skip, (s', \sigma'))} \text{ (HRECV-SUCC)}$$

$$\frac{\sigma = (\mathbb{s}, Q, \mathbb{Q}, o) \quad Q(t) = \text{nil} \quad s' = s\{x \rightsquigarrow 0\}\{y \rightsquigarrow 0\}}{\Theta \vdash (hrecv(x, y), (s, \sigma)) \rightsquigarrow_t (skip, (s', \sigma))} \text{ (HRECV-FAIL)}$$

$$\frac{\sigma = (\mathbb{s}, Q, \mathbb{Q}, o) \quad \mathbb{s}(y) = n \quad s' = s\{x \rightsquigarrow n\}}{\Theta \vdash (x := get(y), (s, \sigma)) \rightsquigarrow_t (skip, (s', \sigma))} \text{ (GET)}$$

$$\frac{\sigma = (\mathbb{s}, Q, \mathbb{Q}, o) \quad [\![e]\!]_s = n \quad \mathbb{s}' = \mathbb{s}\{x \rightsquigarrow n\} \quad \sigma' = (\mathbb{s}', Q, \mathbb{Q}, o)}{\Theta \vdash (put(x, e), (s, \sigma)) \rightsquigarrow_t (skip, (s, \sigma'))} \text{ (PUT)}$$

$$\frac{\sigma = (\mathbb{s}, Q, \mathbb{Q}, o) \quad [\![e]\!]_s = n \quad o' = n :: o \quad \sigma' = (\mathbb{s}, Q, \mathbb{Q}, o')}{\Theta \vdash (out(e), (s, \sigma)) \rightsquigarrow_t (skip, (s, \sigma'))} \text{ (OUT)}$$

Fig.8. Selected rules for operational semantics.

denotes a small step in the sequential operational semantics where the annotated command $\mathcal{C}_1$ with identifier t steps to $\mathcal{C}_2$ under the security policy Θ. Note that the code of each process has been translated into $\mathcal{C}$, where each $hsend(e_1, e_2)$ command is converted to the form of $hsend(e_1, e_2, L)$, with an annotation of the security label of the command. The security policy set Θ is unchanged during the execution and is used only in the dynamic checking of *hsend* and *lsend*. $[\![e]\!]_s$ and $[\![b]\!]_s$ return the values of the expression e and boolean expression b respectively under the given private store s. As shown by the HSEND-SUCC rule, $hsend(e_1, e_2, L)$ appends the message directly to the confidential message queue q_i of the target thread identified by e_1, if it exists. The confidential message queue q is represented as a list, and the operation "++" denotes list concatenation. The premises of the third line show the runtime checking with respect to the policy set Θ and the security label L, which we will explain in Section 5. If there is no thread identified by e_1, it does nothing (the

HSEND-FAIL rule).

The LSEND-SUCC rule and the LSEND-FAIL rule are similar, except that the non-confidential message queue $\mathbb{q}$ instead of q is used. The LSEND-SUCC rule also performs the dynamic checking, though much simpler than the checking for *hsend*. Again, we postpone the explanation of the runtime checking to Section 5.

The *hsend* and *lsend* commands are the only commands that require the runtime checking.

The $hrecv(x, y)$ command dequeues the message from the confidential message queue q if it is not empty, and assigns the identifier of the sender and the message content to x and y respectively, as shown by the HRECV-SUCC rule. If q is empty, it simply returns and sets x and y to be 0 (the HRECV-FAIL rule). $lrecv(x, y)$ is similar, except that the non-confidential message queue $\mathbb{q}$ is used. Its rules are omitted here.

Rules for *get*, *put* and *out* commands are mostly straightforward. Primitive $x := get(y)$ reads the shared variable y into the local variable x. Primitive $put(x, e)$ updates the shared variable x with the value of e. Primitive $out(e)$ appends the value of e to the output trace. Note that the variables in e of command $put(x, e)$ and $out(e)$ are local variables.

4 Static Checking and Program Translation

In this section we propose a type system for static information flow control. It checks the possible information leak inside a process. It also generates the security label L for each *hsend* command, which is used in the program translation phase to generate the $hsend(e_1, e_2, L)$ command for dynamic checking. The security label L represents the privacy contained in message e_2, and it is recorded in the code by program translation.

4.1 Type System

The typing judgment for commands has the form of

$$\Psi, \theta, \Gamma_s \vdash \{\Gamma_p\}\ c\ \{\Gamma'_p\}\,.$$

The program counter (pc) label Ψ represents the security label of the program counter, which is used to check the implicit information channel dependent on control flows. θ is the security policy of the current process, as explained in Subsection 3.3.1. Γ is the type environment that maps variable names to the corresponding labels:

$$\Gamma \in \mathit{Vars} \to \mathit{Labels}\,.$$

Similar to *Label*, we define the order relation between two type environments

$$\Gamma_1 \sqsubseteq \Gamma_2 \quad \text{iff} \quad \forall x.\, \Gamma_1(x) \sqsubseteq \Gamma_2(x)\,,$$

and the least upper bound of two type environments

$$\Gamma_1 \sqcup \Gamma_2 = \{x \rightsquigarrow (\Gamma_1(x) \sqcup \Gamma_2(x)) \mid \text{for all } x \in \mathit{Vars}\}\,.$$

Γ_s assigns labels to "shared" variables, which is usually in the form of

$$\{x \rightsquigarrow \{x\} \mid \text{for all shared variable } x\}\,.$$

Γ_s is never changed during the analysis. Γ_p assigns labels to the process-local variables, which can be different at different program points. Therefore our type system is flow-sensitive.

The judgment $\Gamma_p \vdash e : L$ means that the expression e has the label L under the type environment Γ_p. The label of an expression e is defined simply by taking the least upper bound of labels of its variables.

$$\Gamma_p \vdash e : L \text{ iff } L = \bigsqcup_{x \in vars(e)} \Gamma_p(x),$$

where $vars(e)$ represents the set of variables in e. The judgment $\Gamma_p \vdash b : L$ for the boolean expression b is defined similarly.

Typing of the Send/Receive Primitives. The typing rules are shown in Fig.9. The HRECV rule says that the identifier x of the sender and the message content y returned by the *hrecv* command are assigned to the label $gap \sqcup \Psi$. Since it is difficult to track precisely the label of the message (which requires the knowledge at the sending point in a different process, challenging for compositional analysis), we first set the security labels of x and y to be the receiver's *gap* permission. Because in the dynamic checking we require at the sending point that the security label associated with the send operation should be lower than or equal to the receiver's *gap*, *gap* is a safe but conservative estimation at the receiver side for x and y. The label *gap* also needs to be joined with the pc label Ψ, to record implicit information channel depending on the control flow.

However, this conservative approach sometimes may lead to too many false positives. Consider the following example.

```
t0. {}              | t1. {CB}
    {}              |     {}
    hsend(t1,1);    |     hrecv(t,x);
                    |     out(x);
```

$$\frac{\Psi \sqsubseteq gap}{\Psi, (gap, gsp), \Gamma_s \vdash \{\Gamma_p\}\ hrecv(x, y)\ \{\Gamma_p[y \mapsto gap \sqcup \Psi][x \mapsto gap \sqcup \Psi]\}}\ (\text{HRecv})$$

$$\frac{\Psi = \{\}}{\Psi, \theta, \Gamma_s \vdash \{\Gamma_p\}\ lrecv(x, y)\ \{\Gamma_p[y \mapsto \Psi][x \mapsto \Psi]\}}\ (\text{LRecv})$$

$$\frac{}{\Psi, \theta, \Gamma_s \vdash \{\Gamma_p\}\ hsend(e_1, e_2)\ \{\Gamma_p\}}\ (\text{HSend}) \qquad \frac{\Gamma_p \vdash e_1 : \{\} \quad \Gamma_p \vdash e_2 : \{\} \quad \Psi = \{\}}{\Psi, \theta, \Gamma_s \vdash \{\Gamma_p\}\ lsend(e_1, e_2)\ \{\Gamma_p\}}\ (\text{LSend})$$

$$\frac{\Gamma_p \vdash e : L_e \quad \Psi \sqcup L_e \sqsubseteq gsp}{\Psi, (gap, gsp), \Gamma_s \vdash \{\Gamma_p\}\ out(e)\ \{\Gamma_p\}}\ (\text{Out}) \qquad \frac{\Gamma_s \vdash y : L_y \quad L_y \sqsubseteq gap}{\Psi, (gap, gsp), \Gamma_s \vdash \{\Gamma_p\}\ x := get(y)\ \{\Gamma_p[x \mapsto \Psi \sqcup L_y]\}}\ (\text{Get})$$

$$\frac{\Gamma_p \vdash e : L_e \quad \Gamma_s \vdash x : L_x \quad L_e \sqcup \Psi \sqsubseteq L_x \quad L_x \sqsubseteq gap}{\Psi, (gap, gsp), \Gamma_s \vdash \{\Gamma_p\}\ put(x, e)\ \{\Gamma_p\}}\ (\text{Put}) \qquad \frac{\Gamma_p \vdash e : L_e}{\Psi, \theta, \Gamma_s \vdash \{\Gamma_p\}\ x := e\ \{\Gamma_p[x \mapsto \Psi \sqcup L_e]\}}\ (\text{Assign})$$

$$\frac{\Psi, \theta, \Gamma_s \vdash \{\Gamma_p\}\ c_1\ \{\Gamma'_p\} \quad \Psi, \theta, \Gamma_s \vdash \{\Gamma'_p\}\ c_2\ \{\Gamma''_p\}}{\Psi, \theta, \Gamma_s \vdash \{\Gamma_p\}\ c_1;\ c_2\ \{\Gamma''_p\}}\ (\text{Seq}) \qquad \frac{\Gamma_p \vdash b : L_b \quad \Psi \sqcup L_b, \theta, \Gamma_s \vdash \{\Gamma_p\}\ c\ \{\Gamma_p\}}{\Psi, \theta, \Gamma_s \vdash \{\Gamma_p\}\ while\ b\ do\ c\ \{\Gamma_p\}}\ (\text{While})$$

$$\frac{\Gamma_p \vdash b : L_b \qquad \Psi \sqcup L_b, \theta, \Gamma_s \vdash \{\Gamma_p\}\ \{c_i\}\ \{\Gamma'_p\} \quad i = 1, 2}{\Psi, \theta, \Gamma_s \vdash \{\Gamma_p\}\ if\ b\ then\ c_1\ else\ c_2\ \{\Gamma'_p\}}\ (\text{If})$$

$$\frac{\Gamma_p \sqsubseteq \Gamma''_p \qquad \Psi, \theta, \Gamma_s \vdash \{\Gamma''_p\}\ c\ \{\Gamma'''_p\} \qquad \Gamma'''_p \sqsubseteq \Gamma'_p}{\Psi, \theta, \Gamma_s \vdash \{\Gamma_p\}\ c\ \{\Gamma'_p\}}\ (\text{Conseq})$$

Fig.9. Flow-sensitive typing rules.

Process `t0` sends a constant 1 to `t1`, and then process `t1` sends out the message it receives directly. The first two lines of each process show the *gap* and *gsp* of the process.

While in this particular example, the information that the *gap* of `t0` is $\{\}$, could be used to conclude that x has label $\{\}$. The type system cannot in general determine at compile time the process identifier of the sender; hence it conservatively sets it to the *gap* of the current process. The program does not leak any privacy. However, the HRecv rule will assign the label `{CB}` to x in this case (note that *gap* of `t1` is `{CB}`). Since the following *out* command sends the value of x out, the output will be forbidden by the static checking for the *out* primitive (see the Out rule explained below).

To address this problem, we introduce another pair of primitives, *lsend*/*lrecv*, for sending and receiving non-confidential messages. In the LRecv rule in Fig.9, the labels for x and y are simply Ψ, knowing the message does not contain any confidential information. As a result, the above example will pass the type checking if we replace *hrecv* with *lrecv*. Note that Ψ is required to be $\{\}$, because the value of the program counter can be inferred by observing the value of the low message queue, i.e., testing whether the low message queue is empty. The following program is an example.

```
h = get(CB)
if h == 0
then lrecv(x, y)
else skip

lrecv(x1, y1)
if x1 == 0 // test whether the low
then out(0) // message queue is empty
else out(1)
```

Assume that the low message queue contains only one message before the execution of the program. If the output is 0, then we know that `CB` is equal to 0. Otherwise, `CB` is not equal to 0. Due to the similar reason, the HRecv rules require that Ψ should be less than the access permission *gap*. *gap* can be viewed as the label of the high message queue.

Correspondingly, at the side of the sender, we need to ensure that *lsend* indeed does not send any confidential information. Therefore the LSend rule for $lsend(e_1, e_2)$ requires that the labels for e_1, e_2 and pc label are all $\{\}$.

The HSend rule does not do any checking. Instead, the checking is done at runtime, which we will explain in Section 5.

Output Primitive. The Out rule says that the label of e and the Ψ label should be less than *gsp*, i.e., $L_e \sqcup \Psi \sqsubseteq gsp$. The process can only send out the value whose label L_e is less than *gsp*.

Access to Shared Variables. The GET rule requires that the process has the permission to access the shared variable y. At the end of the command, the label of the local variable x becomes the union of L_y and the Ψ label.

The PUT rule also requires that the process has the permission to access the shared variable x. Also the label of the expression e and the Ψ label should be lower than that of L_x. The security label of the shared variable x is specified by Γ_s and never changed. This requirement prevents the flow from the higher privacy to x. The command does not update local variables; therefore Γ_p does not change at the end.

Note that the data which can be accessed by multiple processes are modeled as the shared variables, such as files, databases. Each of this kind of data has a fixed label (i.e., L) and can only store some known privacy. No privacy that is not described in L can be stored in the shared variables. For example, the labels of the files are set as empty by default; therefore no privacy should be stored in the files. Since we adopt the conservative approach, even when the privacy is encrypted, it is still regarded as privacy. Therefore, the information leak caused by the intermediate forms of data can be prevented.

Other Rules. The rest of the rules are mostly standard. The ASSIGN, SEQ, IF and WHILE rules simply track the information flow, as in normal information flow control. In the IF and WHILE rules, the Ψ labels for the conditional branches (of the *if* commands) or the loop body (of the *while* commands) are joined with the label of the condition b, following the standard approach to track the implicit information channel dependent on the control flow.

The CONSEQ rule says that, to derive $\Psi, \theta, \Gamma_s \vdash \{\Gamma_p\}\ c\ \{\Gamma'_p\}$, one can use a higher initial environment Γ''_p at the beginning of c, as long as the resulting environment Γ'''_p is no higher than Γ'_p. It is similar to the standard consequence rule in Hoare logic.

4.2 Translation of the Program

Based on the static checking, we can infer the security label L at each $hsend(e_1, e_2)$ command. Then we rewrite the send command into the form of $hsend(e_1, e_2, L)$, so that L can be used in the dynamic checking at runtime. In Fig.10, we present the translation from the program p and the command c to their annotated versions $\mathcal{P}$ and $\mathcal{C}$.

The translation of the program is denoted as "$\mathrm{TP}(p, \Psi, \Gamma_s, \Delta) \twoheadrightarrow (\mathcal{P}, \Theta)$". In addition to the original program p, it also takes the set of pc (program counter) labels Ψ and the environments for shared and local variables, i.e., Γ_s and Δ, as inputs. The outputs are the resulted annotated program $\mathcal{P}$ and the security policy Θ extracted from p. Here Ψ maps process identifiers to their pc labels, i.e., $\Psi ::= \{t_1 \rightsquigarrow \Psi_1, \ldots, t_n \rightsquigarrow \Psi_n\}$ and Δ maps process identifiers to their environments, i.e., $\Delta ::= \{t_1 \rightsquigarrow \Gamma_1, \ldots, t_n \rightsquigarrow \Gamma_n\}$. The TPROG rule in Fig.10 translates the original program "$\theta_1; c_1 \parallel \cdots \parallel \theta_n; c_n$" into the policy set "$\{1 \rightsquigarrow \theta_1, \ldots, n \rightsquigarrow \theta_n\}$"

$$\frac{\mathrm{TC}(c_i, \Psi(i), \theta_i, \Gamma_s, \Delta(i)) \rightsquigarrow (\mathcal{C}_i, _) \quad \text{for all } 1 \leqslant i \leqslant n}{\mathrm{TP}(\theta_1; c_1 \parallel \cdots \parallel \theta_n; c_n, \Psi, \Gamma_s, \Delta) \twoheadrightarrow (\mathcal{C}_1 \parallel \cdots \parallel \mathcal{C}_n, \{1 \rightsquigarrow \theta_1, \ldots, n \rightsquigarrow \theta_n\})} \ (\text{TPROG})$$

$$\frac{\Gamma_p \vdash e_1 : L_1 \quad \Gamma_p \vdash e_2 : L_2}{\mathrm{TC}(hsend(e_1, e_2), \Psi, \theta, \Gamma_s, \Gamma_p) \rightsquigarrow (hsend(e_1, e_2, L_1 \sqcup L_2 \sqcup \Psi), \Gamma_p)} \ (\text{THSEND})$$

$$\frac{\mathrm{TC}(c_1, \Psi, \theta, \Gamma_s, \Gamma_p) \rightsquigarrow (\mathcal{C}_1, \Gamma'_p) \quad \mathrm{TC}(c_2, \Psi, \theta, \Gamma_s, \Gamma'_p) \rightsquigarrow (\mathcal{C}_2, \Gamma''_p)}{\mathrm{TC}(c_1;\ c_2, \Psi, \theta, \Gamma_s, \Gamma_p) \rightsquigarrow (\mathcal{C}_1;\ \mathcal{C}_2, \Gamma''_p)} \ (\text{TSEQ}) \qquad \frac{\Psi, \theta, \Gamma_s \vdash \{\Gamma_p\}\ o\ \{\Gamma'_p\}}{\mathrm{TC}(o, \Psi, \theta, \Gamma_s, \Gamma_p) \rightsquigarrow (o, \Gamma'_p)} \ (\text{TPRIM})$$

$$\frac{\Gamma_p \vdash b : L \quad \mathrm{TC}(c_1, \Psi \sqcup L, \theta, \Gamma_s, \Gamma_p) \rightsquigarrow (\mathcal{C}_1, \Gamma^1_p) \quad \mathrm{TC}(c_2, \Psi \sqcup L, \theta, \Gamma_s, \Gamma_p) \rightsquigarrow (\mathcal{C}_2, \Gamma^2_p)}{\mathrm{TC}(\mathit{if}\ b\ \mathit{then}\ c_1\ \mathit{else}\ c_2, \Psi, \theta, \Gamma_s, \Gamma_p) \rightsquigarrow (\mathit{if}\ b\ \mathit{then}\ (\mathcal{C}_1;\ \mathit{end})\ \mathit{else}\ (\mathcal{C}_2;\ \mathit{end}), \Gamma^1_p \sqcup \Gamma^2_p)} \ (\text{TIF})$$

$$\frac{\Gamma_p \vdash b : L \quad \mathrm{TC}(c, \Psi \sqcup L, \theta, \Gamma_s, \Gamma_p) \rightsquigarrow (\mathcal{C}, \Gamma'_p) \quad \Gamma_p = \Gamma'_p}{\mathrm{TC}(\mathit{while}\ b\ \mathit{do}\ c, \Psi, \theta, \Gamma_s, \Gamma_p) \rightsquigarrow (\mathit{while}\ b\ \mathit{do}\ (\mathcal{C};\ \mathit{end}), \Gamma'_p)} \ (\text{TWHILE1})$$

$$\frac{\begin{array}{c}\Gamma_p \vdash b : L \quad \mathrm{TC}(c, \Psi \sqcup L, \theta, \Gamma_s, \Gamma_p) \rightsquigarrow (_, \Gamma''_p) \quad \Gamma_p \neq \Gamma''_p \\ \mathrm{TC}(\mathit{while}\ b\ \mathit{do}\ c, \Psi, \theta, \Gamma_s, \Gamma_p \sqcup \Gamma''_p) \rightsquigarrow (\mathcal{C}, \Gamma'_p)\end{array}}{\mathrm{TC}(\mathit{while}\ b\ \mathit{do}\ c, \Psi, \theta, \Gamma_s, \Gamma_p) \rightsquigarrow (\mathcal{C}, \Gamma'_p)} \ (\text{TWHILE2})$$

Fig.10. Translation rules.

and the annotated program "$\mathcal{C}_1 \parallel \cdots \parallel \mathcal{C}_n$" by translating the command of each process with the command translation rules. Note that "_" is a placeholder which means that there exists an element such that the current proposition holds.

The judgement for command translation is denoted as "$\mathrm{TC}(c, \Psi, \theta, \Gamma_s, \Gamma_p) \rightsquigarrow (\mathcal{C}, \Gamma'_p)$", which takes the original command c, the Ψ label, the security policy θ, and the environments for shared and local variables as inputs. The output is the resulted annotated command $\mathcal{C}$ together with a new local environment Γ'_p, which can be used for translation of the subsequent commands. The only primitive command that needs to be translated is $hsend(e_1, e_2)$. As the THSEND rule in Fig.10 shows, the resulting command is $hsend(e_1, e_2, L)$ where L is the union of the Ψ label and the security labels of e_1 and e_2. For other primitive commands o, the command is unchanged and the resulting local environment Γ'_p is the same with the one generated in the static analysis, as shown by the TPRIM rule. For the branch command "*if* b *then* c_1 *else* c_2", the transformed branches are the translation of the branches, terminated with an explicit *end* command, as shown by the TIF rule. Similarly, for the loop command "*while* b *do* c", as shown by the TWHILE1 and TWHILE2 rules, the transformed body is the translation of the body, appending with an *end* command. The *end* command marks the end of nearest enclosing *if* and *while* commands.

Fig.10 also shows the algorithm to calculate the resulting local environment Γ_p of the sequential, conditional and loop commands. For the branch command "*if* b *then* c_1 *else* c_2", the resulting local variable environment is the union of the resulting local variable environments for c_1 and c_2. For the loop command "*while* b *do* c", the environment should be the loop invariant. We start from an initial Γ_p and calculate Γ'_p at the end of the loop body. If Γ'_p is not equal to Γ_p, we use $\Gamma_p \sqcup \Gamma'_p$ as the initial environment and repeat this process, until the resulting environment never changes. It is easy to see that the process will eventually terminate because the resulting environment Γ'_p is always higher than or equal to the initial one Γ_p (recall the definition of $\Gamma \sqsubseteq \Gamma'$ in Subsection 4.4.1). Since we only have a finite number of shared variables, Γ'_p cannot keep increasing. Eventually we will have $\Gamma'_p = \Gamma_p$.

For the translation of each process, we set the initial local environment with the empty set, i.e., $\Gamma_p^{\mathrm{init}} = \{x \rightsquigarrow \{\} \mid \text{for all local } x\}$. The shared environment is initialized as $\Gamma_s^{\mathrm{init}} = \{x \rightsquigarrow \{x\} \mid \text{for all shared } x\}$, and the initial pc label Ψ_0 is set to be $\{\}$.

5 Dynamic Checking

In the type system we do not do any static checking for *hsend* and do only limited checking for *lsend*. This is because the checking involves the permissions of both the sender and the receiver. However, it is very difficult to determine statically the identifier of the receiver, because applications (modeled as processes in our language) can be dynamically installed or removed, and the argument for the receiver's identifier in the send primitives may be given as a general expression instead of a constant. To address these problems, we rewrite $hsend(e_1, e_2)$ into $hsend(e_1, e_2, L)$ in the phase of program translation, as explained in Subsection 4.4.2, and do runtime checking for the annotated version of the program. The dynamic checking takes the security label of the message L and the policies of both the sender and the receiver as inputs, and checks whether the communication causes privacy leak.

Firstly, we explain the runtime checking for the primitive $hsend(e_1, e_2, L)$ based on the two cases shown in Fig.11. Recall that L is the union of the Ψ label and the labels for e_1 and e_2, as shown in Fig.10. Let (gap_a, gsp_a) and (gap_b, gsp_b) be the permissions of the processes A and B respectively.

Sending the contact book from A to B in Fig.11(a) should be forbidden, because B does not have the permission to access the contact book. Therefore the first constraint for $hsend(e_1, e_2, L)$ is

$$L \sqsubseteq gap_b\,.$$

In the static type system, we take the access permission *gap* of the current process as a conservative estimation for the labels of the incoming messages. This requirement enforces it.

Fig.11(b) should be forbidden because A does not have the permission to send contact book out while B does. The message passing may allow A to send the contact book out through the help of B. To prevent this case, the second constraint is

$$gsp_b \sqsubseteq gsp_a\,.$$

It means that if A does not have the permission to send a confidential data, then it cannot be permitted to communicate with B which does have the permission.

In summary, the constraints for $hsend(e_1, e_2, L)$ are

$$L \sqsubseteq gap_b \wedge\ gsp_b \sqsubseteq gsp_a\,.$$

This is what we need to check at runtime, as the operational semantics rule HSEND-SUCC shown in Fig.8.

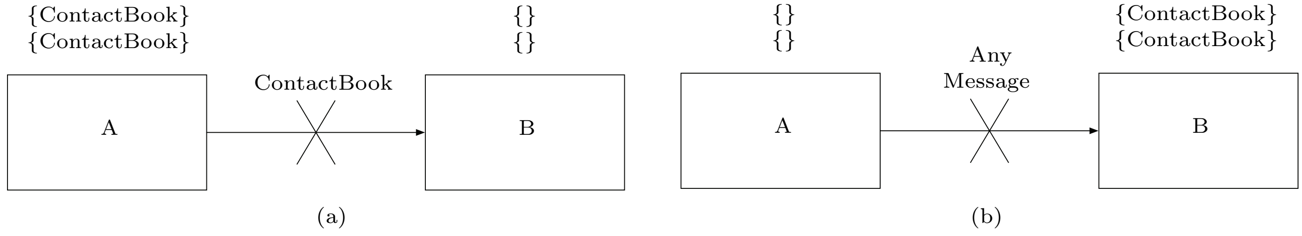


Fig.11. Cases of forbidden send. (a) Private data is forbidden to send to the application without the access permission. (b) An application without the send permission is forbidden to send any message to an application it has.

For the $lsend(e_1, e_2)$ primitive for non-confidential messages, since the static checking (the LSEND rule) already requires that $L_{e_1} \sqcup L_{e_2} \sqcup \Psi = \{\}$, the above condition downgrades to $gsp_b \sqsubseteq gsp_a$, as shown by the LSEND-SUCC operational semantics rule in Fig.8.

6 One More Example

Now we show a simplified input method application as a case study. We only consider two functionalities: 1) generating candidate words based on the (incomplete) user inputs, and 2) updating the word database.

The input method needs to access the contact book to enlarge its word database to help generate hints for names, but it is not supposed to send the contact book out. The security policy ({CB},{}) allows the input method to access the contact book, but prevents the input method from sending it over the Internet. The program shown in Fig.12 is used to model the input method.

To explain the security mechanism more clearly, the input method's functionality is split into two processes, *InputDaemon* and *UpdateDaemon*. The *InputDaemon* reads the contact book, searches the word database for potential matches, and sends the candidates to the requesting process. The *UpdateDaemon* updates the word database. It should never access any confidential data. It only needs to send some non-confidential information about the current database, such as the version number, to the Internet, through the `updatedb` function. We also show the code of the *Requester*, which sends the user inputs `v` to *InputDaemon* and gets the candidates from it.

InputDaemon cannot send the contact book, i.e., `CB`, outside directly by itself, because its send permission is {}. InputDaemon cannot send `CB` outside via UpdateDaemon, because UpdateDaemon's access permission is {}, which will break the dynamic checking for *hsend* (see Section 5). InputDaemon cannot send `CB` outside via InputRequester either. Although InputRequester can access `CB`, its send permission is {}, thereby it cannot send `CB` out.

Note that InputRequester must have the access permission for the contact book `CB`; otherwise the *hsend* primitive in InputDaemon cannot pass the dynamic checking. The function `addIntoDB` adds `CB` into the database `wordsdb` (line 3 in Fig.12). Therefore the value returned by function `candidates` will contain the value of `CB`. Therefore the message `z` will be sent to InputRequester (line 7 in Fig.12). If InputRequester does not have the permission to access `CB`, it cannot receive the message containing `CB`. Therefore the message-passing will fail.

7 Leak-Freedom

An apparent way to specify the leak-freedom property is to require that each output generated by each process i must be permitted by its send permission gsp_i. However, this requirement is too weak because it can-

InputDaemon:	UpdateDaemon:	InputRequester:
1. {CB} {}	9. {} {}	13. {CB} {}
2. y := get(CB);	10. while(true) do {	14. lsend(InputDaemon,v);
3. addIntoDB(y,wordsdb);	11. updatedb(wordsdb);	15. hrecv(t2,w);
4. while (true) do {	12. }	
5. lrecv(t1,x);		
6. z := candidates(x,wordsdb);		
7. hsend(t1,z);		
8. }		

Fig.12. Input method example.

not prevent the information leak caused by process collusion. For instance, for Figs.4(b)–4(e), App1 trivially satisfies this requirement because it does not generate any output at all. However, it may cause the leak of the contact book with the help of App2.

The program shown in Fig.1 is such a concrete example. Assume that process `t0` is untrusted and process `t1` is trusted, e.g., the policies of processes `t0` and `t1` are ({}, {}) and ({`CB`}, {`CB`}) respectively. Process `t0` does not output any privacy by itself, but it controls whether process `t1` outputs the privacy through the interaction with process `t1`. This may potentially lead to the privacy leak, e.g., privacy is sent to an address controlled by the developer of process `t0` who is untrusted. The previous non-interference properties for concurrent programs[10–15] do not prevent this kind of leak. For process `t0`, the send permission is {}; therefore the non-interference property requires that outputs of process `t0` should not contain any privacy. This requirement is satisfied by this program, because process `t0` outputs nothing. For process `t1`, the send permission is {`CB`}; therefore the non-interference property requires that outputs of process `t1` should not contain any privacy other than `CB`. This requirement is satisfied by this program too, because the only information output by `t1` is `CB`.

To address this problem, we propose a new non-interference-like property, leak-freedom property, which requires that for any output generated by any process, as long as the output is influenced by the process i, it does not contain any information unpermitted by gsp_i. In Figs.4(b) and Fig.4(e), the outputs of the contact book by App2 are all influenced by App1. Then we have to define what we mean by saying an output is influenced by a process i. Below we first give an instrumented operational semantics to the core language. Then we formally define the leak-freedom property. Note that the instrumented semantics is used to define leak-freedom property (Definition 5) only. It does not have to be implemented at runtime and does not introduce any extra runtime behaviors.

7.1 Instrumented Operational Semantics

Fig.4(b) and Fig.4(e) show the collusive leak caused by IPC. The following program is an example of Fig.4(b).

```
t0. {CB}                 |  t1. {CB}
    {}                   |      {CB}
    x := get(CB);        |      hrecv(t,x);
    hsend(t1,x);         |      out(x);
```

Process `t0` does not have the permission to send `CB` out, but it sends `CB` to process `t1` directly and `t1` sends it out for `t0`. To forbid this kind of leak, we need to know whether the output is influenced by messages from processes that do not have permissions. Therefore each value in the output is instrumented with a process identifier set, which records the processes influencing the value.

It is not enough to just record the processes affecting the value sent out, i.e., the processes affecting e in the $out(e)$ primitive. Consider the following example of Fig.4(e).

```
                     |  t1. {CB}
t0. {}               |      {CB}
    {}               |      lrecv(t,x);
    x := 1;          |      if (x==1)
    lsend(t1,x);     |      then y := get(CB);
                     |           out(y);
                     |      else skip;
```

Process `t0` does not have the permission to send `CB` out, but its execution leads to `CB` sent. Therefore processes influencing the program counter should be recorded in the output as well.

In our language, a process A may influence an output if

1) A generates the output itself;

2) A interacts with another process B through IPC, which causes B to influence the output.

It is obvious that each output is influenced by at least one or possibly multiple processes.

There are two kinds of process sets to influence the output recorded in the outstate o as the $out(e)$ primitive command executes:

1) the process set influencing the value of e;

2) the process set influencing the program counter.

To trace the set of processes that may influence an output, in the instrumented semantics we record for each variable the set of processes T that may affect its value. When a message is received by primitive $lrecv(x, y)$ or $hrecv(x, y)$, the process identifier of the sender is stored in the values of variables x and y in the private store s. The process identifiers influencing the value of the message should be stored in the value of x and y too. Therefore each element in the message queues q and $\mathbb{q}$ is instrumented with a process set. When the program continues to execute, the set of processes that influences each variable is traced. For example, the assignment $x := e$ assigns the union of

the sets of processes that influence the variables in e to the process set influencing x. To distinguish the outputs affected by different processes, each element in the output history o is instrumented with a process set.

The new state model is shown in Fig.13. Compared with the state model in Fig.7, a process set T is introduced for each variable in private store s, for each message in the message queues q and $\mathbb{q}$, and for each output in the output state o. We also add the process stack Ω that may affect the program counter into the private state of each process, so that the influence of the control flow can be caught. The pc process stack Ω is a stack of the process set. Each time when the process enters a conditional or a loop, a process set that influences the control flow is pushed on the pc process stack. When a conditional or a loop is exited, the top element of each stack is popped. Since the exit points of conditionals and loops are unknown during the execution, we insert the *end* command at the end of conditionals and loops to mark the exit points by program translation (shown in Fig.10). When the *end* command is executed, the top of pc process stack is popped.

(*PIdSet*)	T	$\in$	$\mathcal{P}(PId)$
(*Stores*)	$s, \mathbb{s}$	$\in$	$Vars \rightarrow PIdSet \times Nat$
(*Private State Set*)	δ	$::=$	$\{t_1 \rightsquigarrow \rho_1, \ldots, t_n \rightsquigarrow \rho_n\}$
(*Msg Queue Set*)	$Q, \mathbb{Q}$	$::=$	$\{t_1 \rightsquigarrow q_1, \ldots, t_n \rightsquigarrow q_n\}$
(*Shared State*)	σ	$::=$	$(\mathbb{s}, Q, \mathbb{Q}, o)$
(*Policy Set*)	Θ	$::=$	$\{t_1 \rightsquigarrow \theta_1, \ldots, t_n \rightsquigarrow \theta_n\}$
(*PC PIdSet*)	Ω	$::=$	$\text{nil} \mid T :: \Omega$
(*Private State*)	ρ	$::=$	(Ω, s)
(*Msg Queue*)	$q, \mathbb{q}$	$::=$	$\text{nil} \mid (t, T, n) :: q$
(*OutState*)	o	$::=$	$\text{nil} \mid (T, n) :: o$
(*State*)	Σ	$::=$	(δ, σ)

Fig.13. Instrumented state model.

Selected rules of the instrumented semantics are shown in Fig.14. Note that Ω is a stack of the process set and $\hat{\Omega}$ flattens the stack Ω into a single set, i.e., $\hat{\Omega} = \bigcup_{T \in \Omega} T$. The PARALLEL rule follows the standard interleaving semantics. The *lsend* primitive appends the message, along with the union of process identifiers influencing the message T, process identifiers influencing the program counter $\hat{\Omega}$ and the process identifier of the sender t, to the non-confidential message queue, if the target exists (the LSEND-SUCC rule). Note that $[\![e]\!]_s$ returns a pair (T, i), where T represents the process set influencing the value of e (i.e., the union of the process sets influencing the variables in e) and i is the value of e under the private store s. As shown in the example below, the message queue cannot record only the identifier of the sender, or the following leak cannot be captured.

```
   t0. {}
       {}
       x := 1;
       lsend(t1,x);
|| t1. {CB}
       {CB}
       lrecv(t,x);
       lsend(t2,x);
|| t2. {CB}
       {CB}
       lrecv(t,x);
       if (x==1)
       then y := get(CB);
            out(y);
       else skip;
```

Here process `t0` does not have the permission to send `CB` out. But since `t1` sends to `t2` whatever it receives from `t0`, `t0` leads to the output of `CB` indirectly. The process identifiers influencing the program counter $\hat{\Omega}$ are recorded to prevent the implicit information channel depending on the control flow.

If the target of *lsend* primitive does not exist, it simply returns (see the LSEND-FAIL rule). Correspondingly, the *lrecv*(x, y) command sets the process identifiers influencing y to be the union of process identifiers influencing the non-confidential message T, process identifiers influencing the program counter Ω and the current process identifier t, if the non-confidential message queue is not empty (see the LRECV-SUCC rule). If the non-confidential message queue is empty, the process identifiers influencing y are set to be the union of process identifiers influencing the program counter Ω and the current process identifier t (the LRECV-FAIL rule). No matter whether the non-confidential message is empty, the process identifiers influencing x are set to be the union of process identifiers influencing the program counter Ω and the current process identifier t. The rules for *hsend*/*hrecv* primitives are similar, except that the confidential message queue is used.

The *put* primitive does not record the process identifiers influencing the value of the shared variables. Compared with the inter-process communication, the processes which have written the shared variables are not of concern (e.g., the writers of the shared variables are not recorded in the shared variables in Android), but the sender of the received message should be considered (e.g., the sender of the message is recorded along

$$\frac{\rho = \delta(i) \quad \Theta \vdash (\mathcal{C}_i, (\rho, \sigma)) \hookrightarrow_i (\mathcal{C}_i', (\rho', \sigma'))}{\Theta \vdash (\mathcal{C}_1 \parallel \cdots \parallel \mathcal{C}_i \parallel \cdots \parallel \mathcal{C}_n, (\delta, \sigma)) \longrightarrow (\mathcal{C}_1 \parallel \cdots \parallel \mathcal{C}_i' \parallel \cdots \parallel \mathcal{C}_n, (\delta\{i \rightsquigarrow \rho'\}, \sigma'))} \ (\text{PARALLEL})$$

$$\frac{\begin{array}{c}\rho = (\Omega, s) \quad [\![e_1]\!]_s = (T_i, i) \quad [\![e_2]\!]_s = (T, n) \quad \sigma = (\mathbb{s}, Q, \mathbb{Q}, o) \\ \mathbb{q}_i = \mathbb{Q}(i) \quad \mathbb{q}_i' = \mathbb{q}_i{+\!\!+}[(t, T \cup \hat{\Omega} \cup \{t\}, n)] \quad \mathbb{Q}' = \mathbb{Q}\{i \rightsquigarrow \mathbb{q}_i'\} \quad \sigma' = (\mathbb{s}, Q, \mathbb{Q}', o) \\ \Theta(i).gsp \sqsubseteq \Theta(t).gsp\end{array}}{\Theta \vdash (\mathit{lsend}(e_1, e_2), (\rho, \sigma)) \hookrightarrow_t (\mathit{skip}, (\rho, \sigma'))} \ (\text{LSEND-SUCC})$$

$$\frac{\rho = (\Omega, s) \quad [\![e_1]\!]_s = (T_i, i) \quad \sigma = (\mathbb{s}, Q, \mathbb{Q}, o) \quad \mathbb{Q}(i) = \text{undefined}}{\Theta \vdash (\mathit{lsend}(e_1, e_2), (\rho, \sigma)) \hookrightarrow_t (\mathit{skip}, (\rho, \sigma))} \ (\text{LSEND-FAIL})$$

$$\frac{\begin{array}{c}\rho = (\Omega, s) \quad \sigma = (\mathbb{s}, Q, \mathbb{Q}, o) \quad \mathbb{Q}(t) = (i, (T, n)) :: \mathbb{q}' \\ s' = s\{x \rightsquigarrow (\hat{\Omega} \cup \{t\}, i), y \rightsquigarrow (T \cup \hat{\Omega} \cup \{t\}, n)\} \quad \rho' = (\Omega, s') \\ \mathbb{Q}' = \mathbb{Q}\{t \rightsquigarrow \mathbb{q}'\} \quad \sigma' = (\mathbb{s}, Q, \mathbb{Q}', o)\end{array}}{\Theta \vdash (\mathit{lrecv}(x, y), (\rho, \sigma)) \hookrightarrow_t (\mathit{skip}, (\rho', \sigma'))} \ (\text{LRECV-SUCC})$$

$$\frac{\begin{array}{c}\rho = (\Omega, s) \quad \sigma = (\mathbb{s}, Q, \mathbb{Q}, o) \quad \mathbb{Q}(t) = \text{undefined} \\ s' = s\{x \rightsquigarrow (\hat{\Omega} \cup \{t\}, 0), y \rightsquigarrow (\hat{\Omega} \cup \{t\}, 0)\} \quad \rho' = (\Omega, s')\end{array}}{\Theta \vdash (\mathit{lrecv}(x, y), (\rho, \sigma)) \hookrightarrow_t (\mathit{skip}, (\rho', \sigma))} \ (\text{LRECV-FAIL})$$

$$\frac{\rho = (\Omega, s) \quad [\![e]\!]_s = (T, n) \quad \sigma = (\mathbb{s}, Q, \mathbb{Q}, o) \quad \mathbb{s}' = \mathbb{s}\{x \rightsquigarrow (\{\}, n)\} \quad \sigma' = (\mathbb{s}', Q, \mathbb{Q}, o)}{\Theta \vdash (\mathit{put}(x, e), (\rho, \sigma)) \hookrightarrow_t (\mathit{skip}, (\rho, \sigma'))} \ (\text{PUT})$$

$$\frac{\rho = (\Omega, s) \quad \sigma = (\mathbb{s}, Q, \mathbb{Q}, o) \quad \mathbb{s}(y) = (T, n) \quad s' = s\{x \rightsquigarrow \hat{\Omega} \cup \{t\}, n\} \quad \rho' = (\Omega, s')}{\Theta \vdash (x := \mathit{get}(y), (\rho, \sigma)) \hookrightarrow_t (\mathit{skip}, (\rho', \sigma))} \ (\text{GET})$$

$$\frac{\rho = (\Omega, s) \quad [\![e]\!]_s = (T, n) \quad \sigma = (\mathbb{s}, Q, \mathbb{Q}, o) \quad o' = (\hat{\Omega} \cup \{t\} \cup T, n) :: o \quad \sigma' = (\mathbb{s}, Q, \mathbb{Q}, o')}{\Theta \vdash (\mathit{out}(e), (\rho, \sigma)) \hookrightarrow_t (\mathit{skip}, (\rho, \sigma'))} \ (\text{OUT})$$

$$\frac{\rho = (\Omega, s) \quad [\![b]\!]_s = (T, \text{true}) \quad \rho' = (T :: \Omega, s)}{\Theta \vdash (\mathit{if}\ b\ \mathit{then}\ \mathcal{C}_1\ \mathit{else}\ \mathcal{C}_2, (\rho, \sigma)) \hookrightarrow_t (\mathcal{C}_1, (\rho', \sigma))} \ (\text{IF-TRUE})$$

$$\frac{\rho = (\Omega, s) \quad [\![b]\!]_s = (T, \text{false}) \quad \rho' = (T :: \Omega, s)}{\Theta \vdash (\mathit{if}\ b\ \mathit{then}\ \mathcal{C}_1\ \mathit{else}\ \mathcal{C}_2, (\rho, \sigma)) \hookrightarrow_t (\mathcal{C}_2, (\rho', \sigma))} \ (\text{IF-FALSE})$$

$$\frac{\rho = (\Omega, s) \quad [\![b]\!]_s = (T, \text{true}) \quad \rho' = (T :: \Omega, s)}{\Theta \vdash (\mathit{while}\ b\ \mathit{do}\ \mathcal{C}, (\rho, \sigma)) \hookrightarrow_t (\mathcal{C};\ \mathit{while}\ b\ \mathit{do}\ \mathcal{C}, (\rho', \sigma))} \ (\text{WHILE-TRUE})$$

$$\frac{\rho = (T :: \Omega, s) \quad \rho' = (\Omega, s)}{\Theta \vdash (\mathit{end}, (\rho, \sigma)) \hookrightarrow_t (\mathit{skip}, (\rho', \sigma))} \ (\text{END})$$

Fig.14. Selected rules of the instrumented semantics.

with the content of the message in Android) (see the PUT rule). Correspondingly, x is not considered to be affected by the process identifiers influencing y in $x := \mathit{get}(y)$ command. The process identifiers influencing x are set to be the union of process identifiers influencing the program counter Ω and the current process identifier t (see the GET rule).

The *out* primitive sets the process identifiers influencing the output to be the union of process identifiers influencing the value sent out T, process identifiers influencing the program counter Ω and the current process identifier t (see the OUT rule).

The process set of the condition expression b is pushed into Ω when *if* b *then* $\mathcal{C}_1$ *else* $\mathcal{C}_2$ or *while* b *do* $\mathcal{C}$ commands are executed (the IF-TRUE, IF-FALSE and WHILE-TRUE rules), and the top of Ω is popped when an *end* command is executed (see the END rule). Therefore the influence of the control flow is restricted inside the scope of the *if* and *while* commands.

Compared with the semantics shown in Section 3, the instrumented semantics only does some extra bookkeeping during the execution, which would not affect

the control flow and the original state of the program. Therefore if a program that runs on the new semantics does not leak privacy (i.e., violating its *gsp* permission), then its execution on the original semantics does not leak privacy either. Below we specify the leak-freedom property based on the instrumented semantics.

7.2 Formalization of Leak-Freedom

A program $\mathcal{P}$ does not leak any confidential information, if for each process i with the send permissions gsp_i, the information in the shared variables which are not declared in gsp_i should not flow to the output influenced by process i. Following the idea of the non-interference, we define a similar leak-freedom property, that is, for each process i, if the low-level parts of the initial state, i.e., the parts whose labels are lower than gsp_i, stay unchanged, then the outputs influenced by i remain the same.

To formally define the leak-freedom property, we first define the equivalence of two states with respect to the given security label L (defined in Definition 1), which says that two states are L-equal if they agree on the low level parts. Then we give an invariant property (defined in Definition 2) that needs to be satisfied by the initial states and throughout the program execution. It describes the relation between the permissions of the process which the state belongs to and the permissions of other processes which influence the value of the state. Next we define the equivalence of two annotated programs in Definition 4. The non-interference property can be viewed as the equivalence between a program and itself running under equivalent states. Finally our target security property leak-freedom is defined in Definition 5 based on the program equivalence.

Definition 1 (L-Equal States). *The program states Σ_1 and Σ_2 are L-equal on process i over Θ, Γ_s, Ψ and Δ, denoted as $\Sigma_1 \approx_L^{\Theta,\Gamma_s,\Psi,\Delta,i} \Sigma_2$, iff for all δ_1, σ_1, δ_2 and σ_2, if $\Sigma_1 = (\delta_1, \sigma_1)$ and $\Sigma_2 = (\delta_2, \sigma_2)$, then we require that both private state sets and shared states are L-equal.*

1) *L-Equal Private States. $\delta_1 \approx_L^{\Psi,\Delta} \delta_2$, iff for all t, Ω_1, s_1, Ω_2, s_2, Ψ and Γ, if $\Psi(t) = \Psi$, $\Delta(t) = \Gamma$, $\delta_1(t) = (\Omega_1, s_1)$ and $\delta_2(t) = (\Omega_2, s_2)$, then:*

a) *L-equal pc process sets: $\Omega_1 \approx_L^{\Psi} \Omega_2$, iff if $\Psi \sqsubseteq L$, then $\Omega_1 = \Omega_2$;*

b) *L-equal private stores: $s_1 \approx_L^{\Gamma} s_2$, iff for all x, if $\Gamma(x) \sqsubseteq L$, then $s_1(x) = s_2(x)$.*

2) *L-Equal Shared States. $\sigma_1 \approx_L^{\Theta,\Gamma_s,i} \sigma_2$, iff for all $\mathbb{s}_1, Q_1, \mathbb{Q}_1, o_1$, $\mathbb{s}_2, Q_2, \mathbb{Q}_2, o_2$, if $\sigma_1 = (\mathbb{s}_1, Q_1, \mathbb{Q}_1, o_1)$, $\sigma_2 = (\mathbb{s}_2, Q_2, \mathbb{Q}_2, o_2)$, then we require that four components in the shared state are L-equal:*

a) *L-equal shared stores: $\mathbb{s}_1 \approx_L^{\Gamma_s} \mathbb{s}_2$, iff for all x, if $\Gamma_s(x) \sqsubseteq L$, then $\mathbb{s}_1(x) = \mathbb{s}_2(x)$;*

b) *L-equal high message queues: $Q_1 \approx_L^{\Theta} Q_2$, iff for all j and gap, if $\Theta(j) = (gap, _)$ and $gap \sqsubseteq L$, then $Q_1(j) = Q_2(j)$;*

c) *equal low message queues: $\mathbb{Q}_1 = \mathbb{Q}_2$;*

d) *equal outputs: $o_1 \approx^i o_2$, iff $o_1|_i = o_2|_i$, where $o|_i$ is inductively defined as below:*

$$nil|_i = nil,$$
$$((T,n) :: o)|_i = \begin{cases} n :: o|_i, & \text{if } i \in T,\\ o|_i, & \text{otherwise.}\end{cases}$$

Informally, two states are L-equal, if their low-level parts are the same, i.e., the parts whose labels are lower than L. We define two L-equal states in Definition 1. It says that two states $\Sigma_1 = (\delta_1, \sigma_1)$ and $\Sigma_2 = (\delta_2, \sigma_2)$ are L-equal, i.e., $\Sigma_1 \approx_L^{\Theta,\Gamma_s,\Psi,\Delta,i} \Sigma_2$, if and only if both the private and the shared parts are L-equal respectively, i.e., $\delta_1 \approx_L^{\Psi,\Delta} \delta_2$ and $\sigma_1 \approx_L^{\Theta,\Gamma_s,i} \sigma_2$. The L-equality of two private states $\delta_1 \approx_L^{\Psi,\Delta} \delta_2$ requires that both pc and private stores are L-equal. The L-equality of two shared states requires that shared stores and high message queues are L-equal and they have the same low message queues and output for each process. Here we use $o|_i$ to generate the subsequence of the output influenced by the process i.

Definition 2 (Invariant Property). $\mathsf{INV}(\Sigma, \Theta)$ *iff for all δ and σ, if $\Sigma = (\delta, \sigma)$, then we require that all private state sets and shared states satisfy some certain properties respectively.*

1) *For all $t \in dom(\delta)$ such that $\mathsf{INVP}(\delta(t), t, \Theta)$ which is defined below.*

Private State Invariant: $\mathsf{INVP}(\rho, t, \Theta)$, requires that for all Ω and s, if $\rho = (\Omega, s)$, then the following holds:

a) *for all i and T, if $T \in \Omega$ and $i \in T$, then $\Theta(t).gsp \sqsubseteq \Theta(i).gsp$;*

b) *for all i, x, T and n, if $s(x) = (T, n)$ and $i \in T$, then $\Theta(t).gsp \sqsubseteq \Theta(i).gsp$.*

2) *Shared State Invariant: $\mathsf{INVS}(\sigma, \Theta)$, requires that for all t, $\mathbb{s}$, Q, $\mathbb{Q}$, o, q and $\mathbb{q}$, if $\sigma = (\mathbb{s}, Q, \mathbb{Q}, o)$, $q = Q(t)$ and $\mathbb{q} = \mathbb{Q}(t)$, then the following holds:*

a) *for all i, i', T and n, if $(i', T, n) \in q$ and $i = i' \vee i \in T$, then $\Theta(t).gsp \sqsubseteq \Theta(i).gsp$;*

b) *for all i, i', T and n, if $(i', T, n) \in \mathbb{q}$ and $i = i' \vee i \in T$, then $\Theta(t).gsp \sqsubseteq \Theta(i).gsp$.*

The above invariant property states that for each process t, the process i that influences the values of

process t's private state and shared state must have a higher send permission than the process t, i.e., $gsp_t \sqsubseteq gsp_i$. Since a process can be influenced by other processes only through IPC, the shared state invariant only considers the high and low message queue sets. The property holds during the execution of the program; therefore it is an invariant. This invariant catches the fact that the behaviors of process t, including the *out* command, can be only affected by processes which have a higher send permission. Therefore processes cannot send out the confidential information for others which do not have sufficient permissions. Note that Ω is a stack of the process identifier set (see Subsection 7.7.1), and $T \in \Omega$ means that T is an element in the stack Ω.

We only consider the equivalence of two programs under the assumption of the program termination, which is defined in Definition 3.

Definition 3 (Program Termination). *A program* $\mathcal{P} = (\mathcal{C}_1 \parallel \cdots \parallel \mathcal{C}_n)$ *terminates from the initial private state set* δ *under the policy set* Θ*, i.e.,* $\mathsf{terminate}(\mathcal{P}, \delta, \Theta)$*, iff each process* t *terminates with an arbitrary shared state* σ*, denoted as* $\mathsf{processterm}(t, \mathcal{C}_t, \delta(t), \sigma, \Theta)$*, where* $\mathsf{processterm}(t, \mathcal{C}, \rho, \sigma, \Theta)$ *is inductively defined as below*:

$$\frac{}{\mathsf{processterm}(t, \mathit{skip}, \rho, \sigma, \Theta)}$$

$$\frac{\Theta \vdash (\mathcal{C}, (\rho, \sigma)) \hookrightarrow_t (\mathcal{C}', (\rho', \sigma')) \quad \mathsf{processterm}(t, \mathcal{C}', \rho', \sigma'', \Theta)}{\mathsf{processterm}(t, \mathcal{C}, \rho, \sigma, \Theta)}.$$

When a command $\mathcal{C}$ executes, other processes may modify the shared state. Therefore, a process t always terminates from the configuration $(\mathcal{C}, (\rho, \sigma))$ with the policy set Θ, if it can execute to *skip*, regardless of the changes of the shared state.

The requirement for the termination is reasonable. Consider the program shown in Fig.15[10].

`PIN`, `mask`, `trigger0`, `trigger1`, `maintrigger` and `result` are shared variables. Processes `t0` and `t1` are allowed to output the value of `result`. If initially `maintrigger` = 0, `trigger0` = 0, `trigger1` = 0, `result` = 0, `mask` is a power of 2 and `PIN` is an arbitrary natural number less than twice of `mask`, then the value of `result` will be equal with `PIN` after the termination of the program. Although processes `t0` and `t1` are not permitted to access the privacy `PIN`, they leak it indirectly.

Our requirement of termination will rule out this program, because for process `t0`, if the value of the shared variable `trigger0` is changed to 0, then it will never terminate. To reject this program, Smith and Volpano required that loop conditions in the program should not contain privacy[10]. This requirement rejects some obviously secure programs, such as the following program.

```
{s}
{ }
x := get(s);
while (x < 10) do {
    x := x+1;
}
```

This program satisfies our termination requirement, because at each program point, the change of the shared variable `s` will not lead this program to infinite loop.

Definition 4 (Program Equivalence). $\mathcal{P} \approx_L^{\Theta,\Gamma_s,\Psi,\Delta,i} \mathcal{P}'$ *iff for all* o_1*,* Σ*,* Σ'*,* δ*,* δ'*,* σ *and* σ'*, if*

1) $\Sigma = (\delta, \sigma)$, $\Sigma' = (\delta', \sigma')$, $\mathsf{INV}(\Sigma, \Theta)$, $\mathsf{INV}(\Sigma', \Theta)$ *and* $\Sigma \approx_L^{\Theta,\Gamma_s,\Psi,\Delta,i} \Sigma'$;

2) $\mathsf{terminate}(\mathcal{P}, \delta, \Theta)$ *and* $\mathsf{terminate}(\mathcal{P}', \delta', \Theta)$;

3) $\Theta \vdash (\mathcal{P}, \Sigma) \longrightarrow^* (\mathit{skip} \parallel \cdots \parallel \mathit{skip}, (_, (_, _, _, o_1)))$;

then there exists o_2*, such that* $\Theta \vdash (\mathcal{P}', \Sigma') \longrightarrow^* (\mathit{skip} \parallel \cdots \parallel \mathit{skip}, (_, (_, _, _, o_2)))$ *and* $o_1 \approx^i o_2$.

Informally, $\mathcal{P} \approx_L^{\Theta,\Gamma_s,\Psi,\Delta,i} \mathcal{P}'$ says that the outputs influenced by the process i of the program $\mathcal{P}$ and of the program $\mathcal{P}'$ are indistinguishable, i.e., ($o_1 \approx^i o_2$), if the initial states of these two programs agree on all variables whose labels are less than L. Note here we assume the execution of the two program configurations must terminate, which also implies that the dynamic checking does not abort the execution. The initial state should satisfy the invariant property. Since the execution of the program is non-deterministic due to the interleaving process interaction, the equivalence of two programs cannot be defined as the equivalence of the final states in the arbitrary executions. We say that a program $\mathcal{P}$ is equivalent with another program $\mathcal{P}'$, if for any execution of $\mathcal{P}$ from the initial state Σ, there is an execution path of $\mathcal{P}'$ from the equivalent state Σ', such that they generate the equivalent outputs. Note that a state $\Sigma = (_, (_, _, _, o))$ means that there exists δ, $\mathbb{s}$, Q and $\mathbb{Q}$ such that $\Sigma = (\delta, (\mathbb{s}, Q, \mathbb{Q}, o))$. Based on the definition of the program equivalence, the leak-freedom property can be defined as below.

Definition 5 (Leak-Freedom). *A program* $\mathcal{P}$ *is leak-freedom with respect to the policy set* Θ*, the type environment of shared variables* Γ_s*, pc environments* Ψ *and type environments* Δ*, i.e.,* $LF(\mathcal{P}, \Theta, \Gamma_s, \Psi, \Delta)$*, iff for all* i *and* gsp*, if* $\Theta(i) = (_, gsp)$*, then* $\mathcal{P} \approx_{gsp}^{\Theta,\Gamma_s,\Psi,\Delta,i} \mathcal{P}$.

A program $\mathcal{P}$ satisfies leak-freedom, if it is equivalent with itself. That is, for each process i, it can

generate the same output influenced by the process i, regardless of the value of the high-level parts of the initial state whose labels are not lower than the process i's send permission.

```
t0. {mask, trigger0, maintrigger, result}
    {result}
    maskcopy := get(mask);
    while (maskcopy != 0) do {
        trigger0copy := get(trigger0);
        while (trigger0copy = 0) do {}
        resultcopy := get(result);
        put(result, resultcopy | maskcopy);
        put(trigger0, 0);
        maintriggercopy := get(maintrigger);
        maintriggercopy := maintriggercopy + 1;
        put(maintrigger, maintriggercopy);
        if (maintriggercopy = 1)
        then put(trigger1, 1);
        else skip;
    }
    out(result);
||
t1. {mask, trigger1, maintrigger, result}
    {result}
    maskcopy := get(mask);
    while (maskcopy != 0) do {
        trigger1copy := get(trigger1);
        while (trigger1copy = 0) do {}
        resultcopy := get(result);
        put(result, resultcopy & ~maskcopy);
        put(trigger1, 0);
        maintriggercopy := get(maintrigger);
        maintriggercopy := maintriggercopy + 1;
        put(maintrigger, maintriggercopy);
        if (maintriggercopy = 1)
        then put(trigger0, 1);
        else skip;
    }
    out(result);
||
t2. {PIN, mask, trigger0, trigger1, maintrigger}
    {}
    maskcopy := get(mask);
    while (maskcopy != 0) do {
        put(maintrigger, 0);
        if ((PIN & maskcopy) = 0)
        then put(trigger0, 1);
        else put(trigger1, 1);
        maintriggercopy := get(maintrigger);
        while (maintriggercopy != 2) do {}
        maskcopy := maskcopy/2;
        put(mask, maskcopy);
    }
    put(trigger0,1);
    put(trigger1,1);
```

Fig.15. Example of privacy leak caused by process cooperation.

8 Soundness

We give the soundness theorem in Theorem 1, which ensures that our hybrid approach of combining the type system and the dynamic checking is sufficient to prevent collusive information leak. It says that for any program p, if it can be translated to the instrumented version $\mathcal{P}$, then $\mathcal{P}$ satisfies the leak-freedom property. Here "$\mathrm{TP}(p, \Psi, \Gamma_s, \Delta) \twoheadrightarrow (\mathcal{P}, \Theta)$" (defined in Fig.10) performs the static checking and translates the program p into the annotated program $\mathcal{P}$.

Theorem 1 (Soundness). *For any* p*,* Ψ*,* Γ_s*,* Δ*,* Θ *and* $\mathcal{P}$*, if* $TP(p, \Psi, \Gamma_s, \Delta) \twoheadrightarrow (\mathcal{P}, \Theta)$*, then* $LF(\mathcal{P}, \Theta, \Gamma_s, \Psi, \Delta)$*.*

To prove the leak-freedom property which is defined as the equivalence of two concurrent programs, by following the proof techniques proposed in [18, 19], we introduce a compositional process-local simulation relation (see Definition 7) to accomplish the proof. Firstly, by applying Lemma 1, the target goal of the leak-freedom property can be converted into proving that the program simulation relation holds between two identical annotated programs. Secondly, by applying Lemma 2 which shows the parallel compositionality of the process-local simulation, we are able to decompose the program simulation relation into the process-local simulation relation of each process. Finally, by applying Lemma 3, we are required to show that the annotated process is obtained by doing the command translation, which can be simply derived from the program transformation rule TProg in Fig.10. We mechanize all the definitions and proofs in Coq 8.7.0 which contains around 15 500 lines of Coq scripts[4]. Coq offers a rich capability to express the theorem and the corresponding proof based on a strict mathematical theory, and has been widely used to illustrate the correctness of the proof. With the help of Coq, we provide the machine checkable proof of the soundness theorem. The correctness of the proof can be checked by the Coq proof checker whose size is small and its correctness can be easily verified.

Definition 6 (Program Simulation). Ψ*,* Δ*,* Δ'*,* Θ*,* Γ_s*,* $i \models \mathcal{P} \preceq_L \mathcal{P}'$ *if and only if for all* δ *and* δ'*, if* $\delta \approx_L^{\Psi,\Delta} \delta'$*,* $\mathsf{terminate}(\mathcal{P}, \delta, \Theta)$ *and* $\mathsf{terminate}(\mathcal{P}', \delta', \Theta)$*, then* $\Psi, \Delta', \Theta, \Gamma_s, i \models (\mathcal{P}, \delta) \preceq_L (\mathcal{P}', \delta')$*. Whenever* $\Psi, \Delta', \Theta, \Gamma_s, i \models (\mathcal{P}, \delta) \preceq_L (\mathcal{P}', \delta')$*, the followings hold:*

1) *for all* Σ*,* Σ'*,* Σ_1*,* σ*,* σ'*,* $\mathcal{P}_1$*,* δ_1 *and* σ_1*, if*

[4] https://formal-android-security.github.io, May 2019.

a) $\Sigma = (\delta, \sigma)$, $\Sigma' = (\delta', \sigma')$, $\Sigma_1 = (\delta_1, \sigma_1)$, $\mathsf{INV}(\Sigma, \Theta)$, $\mathsf{INV}(\Sigma', \Theta)$ *and* $\sigma \approx_L^{\Theta,\Gamma_s,i} \sigma'$;

b) $\Theta \vdash (\mathcal{P}, \Sigma) \longrightarrow (\mathcal{P}_1, \Sigma_1)$;

then there exist $\mathcal{P}_1'$, Σ_1', δ_1' *and* σ_1', *such that*:

a) $\Sigma_1' = (\delta_1', \sigma_1')$, $\mathsf{INV}(\Sigma_1, \Theta)$, $\mathsf{INV}(\Sigma_1', \Theta)$ *and* $\sigma_1 \approx_L^{\Theta,\Gamma_s,i} \sigma_1'$;

b) $\Theta \vdash (\mathcal{P}', \Sigma') \longrightarrow^* (\mathcal{P}_1', \Sigma_1')$;

c) $\Psi, \Delta', \Theta, \Gamma_s, i \models (\mathcal{P}_1, \delta_1) \preceq_L (\mathcal{P}_1', \delta_1')$;

2) *if* $\mathcal{P} = (skip \parallel \cdots \parallel skip)$, *for all* Σ, Σ', σ *and* σ', *if* $\Sigma = (\delta, \sigma)$ *and* $\Sigma' = (\delta', \sigma')$, $\mathsf{INV}(\Sigma', \Theta)$ *and* $\sigma \approx_L^{\Theta,\Gamma_s,i} \sigma'$; *then there exists* Σ_1' *such that* $\Theta \vdash (\mathcal{P}', \Sigma') \longrightarrow^* (skip \parallel \cdots \parallel skip, \Sigma_1')$, $\mathsf{INV}(\Sigma_1', \Theta)$ *and* $\Sigma \approx_L^{\Theta,\Gamma_s,\Psi,\Delta,i} \Sigma_1'$.

The above definition says that an annotated program $\mathcal{P}$ is simulated by another annotated program $\mathcal{P}'$, if $(\mathcal{P}, \delta)$ can be simulated by $(\mathcal{P}', \delta')$, i.e., $\Psi, \Delta', \Theta, \Gamma_s, i \models (\mathcal{P}, \delta) \preceq_L (\mathcal{P}', \delta')$, assuming the equivalence of δ and δ', and the termination of $\mathcal{P}$ and $\mathcal{P}'$ running under the private state sets δ and δ' respectively. The program simulation relation $\Psi, \Delta', \Theta, \Gamma_s, i \models (\mathcal{P}, \delta) \preceq_L (\mathcal{P}', \delta')$ relates two pairs of a program and a private state set, instead of two execution configurations. The shared states should always be L-equal throughout establishing the simulation relation, and the private states are only required to be L-equal when the program terminates.

Definition 7 (Process-Local Simulation). Ψ, Γ, Γ', Θ, $\Gamma_s, i \models \mathcal{C} \preceq_L^t \mathcal{C}'$ *if and only if for all* ρ *and* ρ', *if* $\rho \approx_L^{\Psi,\Gamma} \rho'$, *for all* σ_0 *such that* $\mathsf{processterm}(t, \mathcal{C}, \rho, \sigma_0, \Theta)$ *and for all* σ_0' *such that* $\mathsf{processterm}(t, \mathcal{C}', \rho', \sigma_0', \Theta)$, *then* $\Psi, \Gamma', \Theta, \Gamma_s, i \models (\mathcal{C}, \rho) \preceq_L^t (\mathcal{C}', \rho')$. *Whenever* $\Psi, \Gamma', \Theta, \Gamma_s, i \models (\mathcal{C}, \rho) \preceq_L^t (\mathcal{C}', \rho')$, *then the followings are true*:

1) *for all* σ, σ', $\mathcal{C}_1$, ρ_1 *and* σ_1, *if*

a) $\mathsf{INVP}(\rho, t, \Theta)$, $\mathsf{INVS}(\sigma, \Theta)$, $\mathsf{INVP}(\rho', t, \Theta)$, $\mathsf{INVS}(\sigma', \Theta)$ *and* $\sigma \approx_L^{\Theta,\Gamma_s,i} \sigma'$;

b) $\Theta \vdash (\mathcal{C}, (\rho, \sigma)) \hookrightarrow_t (\mathcal{C}_1, (\rho_1, \sigma_1))$;

then there exist $\mathcal{C}_1'$, ρ_1' *and* σ_1' *such that*

a) $\Theta \vdash (\mathcal{C}', (\rho', \sigma')) \hookrightarrow_t^* (\mathcal{C}_1', (\rho_1', \sigma_1'))$;

b) $\Psi, \Gamma', \Theta, \Gamma_s, i \models (\mathcal{C}_1, \rho_1) \preceq_L^t (\mathcal{C}_1', \rho_1')$;

c) $\mathsf{INVP}(\rho_1, t, \Theta)$, $\mathsf{INVS}(\sigma_1, \Theta)$, $\mathsf{INVP}(\rho_1', t, \Theta)$, $\mathsf{INVS}(\sigma_1', \Theta)$ *and* $\sigma_1 \approx_L^{\Theta,\Gamma_s,i} \sigma_1'$;

2) *if* $\mathcal{C} = skip$, *for all* σ *and* σ', *if* $\mathsf{INVP}(\rho', t, \Theta)$, $\mathsf{INVS}(\sigma', \Theta)$ *and* $\sigma \approx_L^{\Theta,\Gamma_s,i} \sigma'$, *then there exist* ρ_1' *and* σ_1' *such that*

a) $\Theta \vdash (\mathcal{C}', (\rho', \sigma')) \hookrightarrow_t^* (skip, (\rho_1', \sigma_1'))$;

b) $\mathsf{INVP}(\rho_1', t, \Theta)$, $\mathsf{INVS}(\sigma_1', \Theta)$, $\sigma \approx_L^{\Theta,\Gamma_s,i} \sigma_1'$ *and* $\rho \approx_L^{\Psi,\Gamma'} \rho_1'$.

The process-local simulation "$\Psi, \Gamma, \Gamma', \Theta, \Gamma_s, i \models \mathcal{C} \preceq_L^t \mathcal{C}'$" says that the annotated command $\mathcal{C}$ is simulated by $\mathcal{C}'$. The simulation relation requires that the execution of $\mathcal{C}$ is related to the execution of $\mathcal{C}'$ running under the equivalent private states. Starting from the L-equal shared states, each step of $\mathcal{C}$ corresponds to zero or multiple steps of $\mathcal{C}'$, and the resulting shared states are still L-equal. If $\mathcal{C}$ terminates, then $\mathcal{C}'$ terminates as well, and the final states should be L-equal. Here the invariant of L-equality over shared states can be considered as the assumption from environments, which allows us to prove the parallel compositionality given in Lemma 2.

Lemma 1 (Program Simulation Implies Program Equivalence). *For all* Ψ, Δ, Δ', Θ, Γ_s, i, $\mathcal{P}$ *and* $\mathcal{P}'$, *if* $\Psi, \Delta, \Delta', \Theta, \Gamma_s, i \models \mathcal{P} \preceq_{\Theta(i).gsp} \mathcal{P}'$, *then* $\mathcal{P} \approx_{\Theta(i).gsp}^{\Theta,\Gamma_s,\Psi,\Delta,i} \mathcal{P}'$.

Proof. To prove $\mathcal{P} \approx_{\Theta(i).gsp}^{\Theta,\Gamma_s,\Psi,\Delta,i} \mathcal{P}'$, we need to prove that for all k, o_1, Σ, Σ', δ, δ', σ and σ', if

1) $\Psi, \Delta, \Delta', \Theta, \Gamma_s, i \models \mathcal{P} \preceq_{\Theta(i).gsp} \mathcal{P}'$;

2) $\Sigma = (\delta, \sigma)$ and $\Sigma' = (\delta', \sigma')$;

3) $\mathsf{terminate}(\mathcal{P}, \delta, \Theta)$ and $\mathsf{terminate}(\mathcal{P}', \delta', \Theta)$;

4) $\mathsf{INV}(\Sigma, \Theta)$, $\mathsf{INV}(\Sigma', \Theta)$ and $\Sigma \approx_L^{\Theta,\Gamma_s,\Psi,\Delta,i} \Sigma'$;

5) $\Theta \vdash (\mathcal{P}, \Sigma) \longrightarrow^k (skip \parallel \cdots \parallel skip, (_, (_, _, _, o_1)))$;

then there exists o_2, such that $\Theta \vdash (\mathcal{P}', \Sigma') \longrightarrow^* (skip \parallel \cdots \parallel skip, (_, (_, _, _, o_2)))$ and $o_1 \approx^i o_2$. We first do induction over k, and the base case is trivial. For the induction case, we unfold the program simulation relation and apply it together with the induction hypothesis to finish the proof. □

Lemma 2 (Parallel Compositionality). *For all* Ψ, Δ, Δ', Θ, Γ_s, L, i, $\mathcal{P} = (\mathcal{C}_1 \parallel \cdots \parallel \mathcal{C}_n)$ *and* $\mathcal{P}' = (\mathcal{C}_1' \parallel \cdots \parallel \mathcal{C}_n')$, *if for all* $0 \leqslant t \leqslant n$ *such that* $\Psi(t), \Delta(t), \Delta'(t), \Theta, \Gamma_s, i \models \mathcal{C}_t \preceq_L^t \mathcal{C}_t'$, *then* $\Psi, \Delta, \Delta', \Theta, \Gamma_s, i \models \mathcal{P} \preceq_L \mathcal{P}'$.

Proof. The above lemma can be proved by doing co-induction. □

Lemma 3 (Command Translation Implies Process-Local Simulation). *For all* Ψ, Γ, Γ', Θ, Γ_s, t, c *and* $\mathcal{C}$, *if* $TC(c, \Psi, \Theta(t), \Gamma_s, \Gamma) \rightsquigarrow (\mathcal{C}, \Gamma')$, *then for all* i, *we have* $\Psi, \Gamma, \Gamma', \Theta, \Gamma_s, i \models \mathcal{C} \preceq_{\Theta(i).gsp}^t \mathcal{C}$.

Proof. We can prove the above lemma by doing induction over the inference rules of the command translation. □

9 Implementation

The checking rules of our approach can be easily extended to analyze Android applications. For Android applications, the operations to send a message (e.g., Intent) to another component or receive a message from another component can be viewed as the *send*/*recv*

primitives, and the rules for this pair of commands can be applied to prevent the information leak caused by the inter-component communication. The data that can be accessed by various components, such as the files, content providers and databases, can be viewed as the shared variables. To prevent the privacy leakage caused by the information transmission through the intermediate data, according to our rules, the intermediate data can be only allowed to store some specific privacy, or even no privacy. The operations to read/write the intermediate data are viewed as the *get*/*put* primitives. The functions that send the data outside the device (such as writing to the network or Bluetooth) can be viewed as the *out* primitives. And only the data satisfying the send permission can be sent out. The rules for the *if* and *while* commands can be used to track the implicit information flow caused by the control flow.

Our approach has been implemented in Android and the concrete implementation AndroidLeaker can be found in [20]. The static checking is implemented in the soot framework for Android application packages. The dynamic checking is achieved through code instrumentation instead of changing the implementation of system calls. Some annotated instructions are inserted before each message-sending operation to check the validity of the communication. The implementation requires programmers to provide *gap* and *gsp* together with the APK package as inputs.

Other than the permissions, the implementation does not require any changes or annotations of the code. By default each send and receive are treated as the more conservative *hsend* and *hrecv* primitives. Programmers can force the use of *lsend* and *lrecv* by adding annotations, but it is not obligatory.

AndroidLeaker[20] is tested on the DroidBench[1] test cases, gets 82% precision for information leak caused by individual applications, and can analyze the information leak caused by inter-component communication more precisely. Unlike purely dynamic mechanism (e.g., TaintDroid[2]) that adds overhead for each command, AndroidLeaker only adds 5.6 ms for each inter-component communication system call. By contrast, TaintDroid introduces 14% CPU overhead on average for each application. AndroidLeaker will be more efficient than the purely dynamic checking, if the communication happens rarely.

10 Related Work

There have been many static analysis based tools proposed[1,3,7–9,21–25] to address the security problems of Android systems. CHEX[24] tries to detect the so-called hijacking vulnerable applications, which may expose interfaces to perform privileged actions so that other unauthorized applications can exploit them to access the confidential data. Our method can also detect this kind of vulnerable applications because an application is forbidden to access or send confidential data on behalf of another unauthorized application. On the other hand, unlike our work, CHEX[24] is not a general purpose tool to prevent different kinds of information leak.

Leakminer[26] and FlowDroid[1] are both based on the soot framework and implemented for the Android system, but they do not consider the cooperation among the applications and focus on the analysis of one application.

Epicc[9] detects Android ICC (inter-component communication) vulnerabilities, including sending an Intent that may be intercepted by a malicious component, or exposing components to be launched by a malicious Intent. This is similar to our concern of the communication among the applications. Since Epicc performs static analysis and adopts a conservative strategy, while we do dynamic checking when the communication occurs, we can potentially achieve less false positives.

AppIntent[7] analyzes the sensitive data transmission in Android by using the configuration file (i.e., AndroidManifest.xml) to infer the applications that this application may communicate with, but the result is just a very coarse estimation, which may lead to many false positives.

ComDroid[27] detects the Intents possibly intercepted by malicious applications and the exposed components that may be utilized by the malicious applications to acquire confidential data or perform unauthorized actions. It does not concern whether the existence of those intents or components indeed leaks private information or not, therefore is overly conservative and may generate many false positives.

There are also some dynamic-checking based approaches[2,5,28,29]. TaintDroid[2] and TaintART[5] track the flow of privacy sensitive data between applications. They automatically taint the data from privacy-sensitive sources and track the labels when the sensitive data propagates. However, they do not prevent the leaks via implicit flows. Aquifer[28] prevents an application from exporting the data it gets from another application and prevents the confidential data from being exported without users' consciousness. It does not prevent an application from exporting the

data by calling an authorized application. Felt *et al.*[29] discussed the possible ways to perform permission re-delegation attacks and proposed the IPC inspection to detect permission re-delegation. The intuition is that an application reduces its privilege after receiving messages from a less privileged application. Its aim is to prevent the confidential data from being accessed by a non-privileged application, rather than to prevent the confidential data from being shared once it has been accessed. AppAudit[6] performs dynamic analysis through the simulation of the execution of the bytecode, with the help of static analysis to narrow down the analysis scope. The communications among applications are not considered.

On the other hand, unlike the aforementioned tools that are all implemented for Android, we only discuss the key ideas based on a simplified core language. we formally define and prove the soundness of our approach. As far as we know, only Chaudhuri[30] studied the Android security formally previously. He designed a formal language to describe Android applications and proposed a type system to enforce secure flows among content providers. But he did not model the inter-process communication explicitly; thus the collusive leak cannot be prevented.

There have been many formal studies about the security for concurrent programs[10−15,31−34]. To our knowledge, the first semantically sound security enforcement for concurrent programs is proposed by Smith and Volpanno[11]. They developed a security type system which is able to guarantee non-interference for concurrent programs. In their type systems, loop conditions should not contain any privacy, and loops should not appear in the body of the conditionals whose branch conditions contain privacy. Subsequently, they extended the non-interference to the probabilistic non-interference to rule out probabilistic timing channels in the programs[10], and described how they can be eliminated without making the type system more restricted. But their solution depends on a non-standard *protect* primitive which is difficult to implement[12,31,34]. Sabelfeld and Sands eliminated the need of *protect* primitive by padding the branches of the conditionals[12]. They based the security condition on bisimulation, and got a scheduler-independent and compositional non-interference property. But their type systems still require that loop conditions should not contain any privacy.

The requirement that some loop or branch conditions should not contain any privacy[11,12] is introduced to prevent the information leak caused by the influence on termination among processes. As shown by the program in Fig.15, another process may make the loop of current process whose loop condition contains privacy terminate only when some privacy (e.g., s) equals a given value. Then the value of s is inferred after the termination of current loop. Rejection of loops whose loop conditions contain privacy prevents this kind of leak, but it also limits the expressiveness of programs (e.g., cannot search a given privacy among a group of privacy). Our type system allows loop and branch conditions to contain privacy.

The scheduler-independence of the security condition has been widely investigated[12−14,31]. Zdancewic and Myers[13] extended non-interference with observational determinism. To achieve the observational determinism, they required that the concurrent program should be race-free. Their approach to preventing races is to enforce that only one of the processes can write the shared memory. This requirement is too strong for Android applications, because every application with permissions may write the privacy, which is shared among applications.

Russo and Sabelfeld introduced the interaction between processes and the scheduler to prevent privacy leak caused by interleaving of concurrent programs[31]. They used a novel pair of primitives, *hide* and *unhide*, to allow processes to affect the scheduling of the scheduler. It is difficult to adapt this method in Android, because to allow the interaction between processes and the scheduler, the Android system should be deeply customized.

Mantel and Sudbrock partitioned commands into high commands that definitely do not modify the public variables and into low commands that potentially modify values of the public variables[14]. Although loop and branch conditions in high commands are allowed to contain privacy, those in low commands are not. Each Android application may modify the public outputs, thereby they will be classified as low commands, which removes the benefits of this method.

None of the existing scheduler-independent solutions[12−14,31] are proper for Android applications. We do not aim to achieve the scheduler-independent security. Instead we build our security condition on the standard interleaving semantics, assuming that nothing about the scheduler is known.

Another desirable property of security analysis for concurrent programs is the compositionality property[12,14,15,31−33]. The solutions based on the

bisimulation[12,14,15] forbid all (or some) loop and branch conditions to contain privacy. Additionally, although Mantel *et al.* permitted the branch conditions to be high, their type rules for conditionals depend on command equivalence based on the semantics of the command, which leads the type system to be not wholly syntax-directed and not easily implemented[15]. Russo and Sabelfeld based their approaches on a non-standard *hide* and *unhide* primitives[31,32], which is difficult to be implemented in Android applications. Askarov *et al.* tracked the information flow dynamically and performed the static analysis during the execution[33]. To implement this hybrid mechanism, Android needs to be greatly modified. Our type system is compositional (shown by Theorem 1 in Section 8), and large numbers of Android applications can be analyzed separately, without knowing the applications which may run together with them.

Costanzo *et al.*[35] used a flexible observation function to specify the security policy and prove that the non-interference can be preserved by security-preserving simulation, and they used the methodology to prove the memory separation among the individual processes in a practical kernel, i.e., mCertiKos-secure, but the IPC (inter-process communication) is disabled in their kernel.

11 Conclusions

In this paper, we proposed a new mechanism to prevent the information leak in Android. It combines static information flow control with runtime checking to effectively prevent the privacy leak caused by the cooperation of the applications. This hybrid approach may potentially have less false positive than the pure static method, and may benefit from the static analysis to get lower runtime overhead than pure dynamic checking. The soundness of this approach is formally specified and proved in Coq.

References

[1] Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, le Traon Y, Octeau D, McDaniel P. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proc. the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2014, pp.259-269.

[2] Enck W, Gilbert P, Chun B G, Cox L P, Jung J, McDaniel P, Sheth A N. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. the 9th USENIX Conference on Operating Systems Design and Implementation*, October 2010, pp.393-407.

[3] Gibler C, Crussell J, Erickson J, Chen H. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Proc. the 5th International Conference on Trust and Trustworthy Computing*, June 2012, pp.291-307.

[4] Sakamoto S, Okuda K, Nakatsuka R, Yamauchi T. DroidTrack: Tracking information diffusion and preventing information leakage on Android. In *Proc. the 2013 Multimedia and Ubiquitous Engineering*, May 2013, pp.243-251.

[5] Sun M, Wei T, Lui J C S. TaintART: A practical multilevel information-flow tracking system for Android runtime. In *Proc. the 2016 ACM SIGSAC Conference on Computer and Communications Security*, October 2016, pp.331-342.

[6] Xia M, Gong L, Lyu Y, Qi Z, Liu X. Effective real-time Android application auditing. In *Proc. the 2015 IEEE Symposium on Security and Privacy*, May 2015, pp.899-914.

[7] Yang Z, Yang M, Zhang Y, Gu G, Ning P, Wang X S. AppIntent: Analyzing sensitive data transmission in Android for privacy leakage detection. In *Proc. the 2013 ACM SIGSAC Conference on Computer and Communications Security*, November 2013, pp.1043-1054.

[8] Zhao Z, Colon O F C. "TrustDroid™": Preventing the use of SmartPhones for information leaking in corporate networks through the used of static analysis taint tracking. In *Proc. the 7th International Conference on Malicious and Unwanted Software*, October 2012, pp.135-143.

[9] Octeau D, McDaniel P, Jha S, Bartel A, Bodden E, Klein J, le Traon Y. Effective inter-component communication mapping in Android with *Epicc*: An essential step towards holistic security analysis. In *Proc. the 22nd USENIX Conference on Security*, August 2013, pp.543-558.

[10] Volpano D M, Smith G. Probabilistic noninterference in a concurrent language. In *Proc. the 11th IEEE Computer Security Foundations Workshop*, June 1998, pp.34-43.

[11] Smith G, Volpano D M. Secure information flow in a multi-threaded imperative language. In *Proc. the 25th ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1998, pp.355-364.

[12] Sabelfeld A, Sands D. Probabilistic noninterference for multi-threaded programs. In *Proc. the 13th IEEE Computer Security Foundations Workshop*, July 2000, pp.200-214.

[13] Zdancewic S, Myers A C. Observational determinism for concurrent program security. In *Proc. the 16th IEEE Computer Security Foundations Workshop*, June 2003, pp.29-43.

[14] Mantel H, Sudbrock H. Flexible scheduler-independent security. In *Proc. the 15th European Symposium on Research in Computer Security*, September 2010, pp.116-133.

[15] Mantel H, Sands D, Sudbrock H. Assumptions and guarantees for compositional noninterference. In *Proc. the 24th IEEE Computer Security Foundations Symposium*, June 2011, pp.218-232.

[16] Goguen J A, Meseguer J. Security policies and security models. In *Proc. the 1982 IEEE Symposium on Security and Privacy*, April 1982, pp.11-20.

[17] Goguen J A, Meseguer J. Unwinding and inference control. In *Proc. the 1984 IEEE Symposium on Security and Privacy*, April 1984, pp.75-87.

[18] Liang H, Feng X, Fu M. A rely-guarantee-based simulation for verifying concurrent program transformations. In *Proc. the 39th ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2012, pp.455-468.

[19] Liang H, Feng X. Modular verification of linearizability with non-fixed linearization points. In *Proc. the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2013, pp.459-470.

[20] Zhang Z, Feng X. AndroidLeaker: A hybrid checker for collusive leak in Android applications. In *Proc. the 3rd International Symposium on Dependable Software Engineering Theories, Tools, and Applications*, October 2017, pp.164-180.

[21] Xiao X, Tillmann N, Fähndrich M, de Halleux J, Moskal M. User-aware privacy control via extended static-information-flow analysis. In *Proc. the 2012 IEEE/ACM International Conference on Automated Software Engineering*, September 2012, pp.80-89.

[22] Mann C, Starostin A. A framework for static detection of privacy leaks in Android applications. In *Proc. the 27th Annual ACM Symposium on Applied Computing*, March 2012, pp.1457-1462.

[23] Kim J, Yoon Y, Yi K, Shin J. ScanDal: Static analyzer for detecting privacy leaks in Android applications. In *Proc. the 2012 Mobile Security Technologies*, May 2012.

[24] Lu L, Li Z, Wu Z, Lee W, Jiang G. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *Proc. the 2012 ACM Conference on Computer and Communications Security*, October 2012, pp.229-240.

[25] Xu R, Saïdi H, Anderson R. Aurasium: Practical policy enforcement for Android applications. In *Proc. the 21st USENIX Conference on Security Symposium*, August 2012, pp.539-552.

[26] Yang Z, Yang M. LeakMiner: Detect information leakage on Android with static taint analysis. In *Proc. the 3rd World Congress on Software Engineering*, November 2012, pp.101-104.

[27] Chin E, Felt A P, Greenwood K, Wagner D. Analyzing inter-application communication in Android. In *Proc. the 9th International Conference on Mobile Systems, Applications, and Services*, June 2011, pp. 239-252.

[28] Nadkarni A, Enck W. Preventing accidental data disclosure in modern operating systems. In *Proc. the 2013 ACM Conference on Computer and Communications Security*, November 2013, pp.1029-1042.

[29] Felt A P, Wang H J, Moshchuk A, Hanna S, Chin E. Permission re-delegation: Attacks and defenses. In *Proc. the 20th USENIX Conference on Security*, August 2011, Article No. 22.

[30] Chaudhuri A. Language-based security on Android. In *Proc. the 2009 Workshop on Programming Languages and Analysis for Security*, June 2009, pp.1-7.

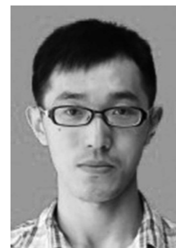
[31] Russo A, Sabelfeld A. Securing interaction between threads and the scheduler. In *Proc. the 19th IEEE Computer Security Foundations Workshop*, July 2006, pp.177-189.

[32] Russo A, Sabelfeld A. Securing interaction between threads and the scheduler in the presence of synchronization. *The Journal of Logic and Algebraic Programming*, 2009, 78(7): 593-618.

[33] Askarov A, Chong S, Mantel H. Hybrid monitors for concurrent noninterference. In *Proc. the 28th IEEE Computer Security Foundations Symposium*, July 2015, pp.137-151.

[34] Russo A, Sabelfeld A. Security for multithreaded programs under cooperative scheduling. In *Proc. the 6th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics*, June 2006, pp.474-480.

[35] Costanzo D, Shao Z, Gu R. End-to-end verification of information-flow security for C and assembly programs. In *Proc. the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2016, pp.648-664.



Zi-Peng Zhang is a senior engineer of Huawei Technologies Co., Ltd., Shanghai. He received his Bachelor's and Ph.D. degrees in computer science from University of Science and Technology of China, Hefei, in 2010 and 2018 respectively. His research interests include program analysis, formal verification, and information flow control.

Ming Fu is a technical expert of Huawei Technologies Co., Ltd., Shanghai. He received his Bachelor's and Ph.D. degrees in computer science from University of Science and Technology of China, Hefei, in 2004 and 2010 respectively. His research interests include OS kernel verification, real-time embedded operating systems, concurrency verification, formal methods, program logic, and interactive theorem proving.

Xin-Yu Feng is a professor in the State Key Laboratory for Novel Software Technology and the Department of Computer Science and Technology, Nanjing University, Nanjing. He received his Bachelor's degree and Master's degree in computer science from Nanjing University, Nanjing, in 1999 and 2002 respectively. He then received his Ph.D. degree in computer science from Yale University, New Haven, in 2007. His research interests are in the area of formal methods and programming languages. In particular, he is interested in developing theories, programming languages and tools to build formally certified system software, with rigorous guarantees of safety and correctness.