

# Lintent: towards security type-checking of Android applications

Michele Bugliesi, Stefano Calzavara, and Alvise Spanò

Università Ca' Foscari Venezia

**Abstract.** The widespread adoption of Android devices has attracted the attention of a growing computer security audience. Fundamental weaknesses and subtle design flaws of the Android architecture have been identified, studied and fixed, mostly through techniques from data-flow analysis, runtime protection mechanisms, or changes to the operating system. This paper complements this research by developing a framework for the analysis of Android applications based on typing techniques. We introduce a formal calculus for reasoning on the Android inter-component communication API and a type-and-effect system to statically prevent privilege escalation attacks on well-typed components. Drawing on our abstract framework, we develop a prototype implementation of **Lintent**, a security type-checker for Android applications integrated with the Android Development Tools suite. We finally discuss preliminary experiences with our tool, which highlight real attacks on existing applications.

## 1 Introduction

Mobile phones have quickly evolved from simple devices intended for phone calls and text messaging, to powerful handheld PDAs, hosting sophisticated applications that manage personal data and interact on-line to share information and access (security-sensitive) services. This evolution has attracted the interest of a growing community of researchers on mobile phone security, and on Android security in particular.

Fundamental weaknesses and subtle design flaws of the Android architecture have been identified, studied and fixed. Originated with the seminal work in [9], a series of papers have developed techniques to ensure various system-level information-flow properties, by means of data-flow analysis [13], runtime detection mechanisms [7] and changes to the operating system [12]. Other papers have applied similar techniques to the study of the intent-based communication model of Android and its interaction with the underlying permission system [5,2]. Somewhat surprisingly, typing techniques have instead received very limited attention, with few notable exceptions to date ([3], and more recently [1]). As a result, the potential extent and scope of type-based analysis has been so far left largely unexplored. In the present paper we make a step towards filling this gap.

*Contributions.* Our analysis of the Android platform is targeted at the static detection of privilege escalation attacks, a vulnerability which exposes the framework to the risk of unauthorized permission usage by malicious applications. To

carry out our study, we introduce  $\pi$ -Perms, a simple formal calculus for reasoning about inter-component interaction in Android (Section 3). Albeit small and abstract,  $\pi$ -Perms captures the most relevant aspects of the Android message passing architecture and its relationships with the underlying permission system. Our formalization pays off, as it allows us to unveil subtle attack surfaces to the current Android implementation that had not been evaluated before.

We tackle the problem of programmatically preventing privilege escalation attacks inside  $\pi$ -Perms, by spelling out a formal definition of safety (Section 4) and proposing a sound security type system which statically enforces such notion, despite the best efforts of an opponent (Section 5). Providing the desired protection turns out to be challenging, since the inadvertent disclosure of sensitive data may enable some typically overlooked privilege escalation scenarios.

Based on our formal framework, we then develop a prototype implementation of **Lintent**, a type-based analyzer integrated with the Android Development Tools suite (Section 6). **Lintent** integrates our typing technique for privilege escalation detection within a full-fledged static analysis framework aimed at supporting a robust and more reliable development process. **Lintent** is the first type-based analyzer for Android applications and its implementation highlights a number of engineering challenges which should likely be tackled by any other type-based verification tool for Android. We discuss preliminary experiences with our tool, which highlight real attacks on existing applications (Section 7).

Enhancing the Android development process is increasingly being recognized as an urgent need [4,10,8,16,6]: **Lintent** represents a first step in that direction<sup>1</sup>.

## 2 Android Overview

*Intents.* Once installed on a device, Android applications run isolated from each other in their own security sandbox. Data and functionality sharing among different applications is implemented through a message-passing paradigm built on top of *intents*, i.e., asynchronous messages providing an abstract description of an operation to be performed. Intents may be either *explicit* or *implicit*: the former specify their intended receiver by name and are always securely delivered to it; the latter, instead, do not mention any specific receiver and just require delivery to any application that supports a given operation (an *action*).

*Components.* Intents are delivered to application *components*, the essential building blocks of Android applications. There are four different types of components. An *activity* represents a screen with a user interface: activities are started with an intent and possibly return a result upon termination. A *service* runs in the background to perform long-running computations: services can either be started with an intent, or expose a remote method invocation interface to a client by returning it a *binder* object. A *broadcast receiver* waits for intents sent to multiple applications. A *content provider* manages a shared set of persistent application

---

<sup>1</sup> Companion tech report and **Lintent** at <https://github.com/alvisespano/lintent>

data. Content providers are not accessed through intents, but through a CRUD (Create-Read-Update-Delete) interface reminiscent of SQL.

*Protection mechanisms.* The Android security model implements isolation and privilege separation on top of a simple permission system. Android permissions are identified by strings and can be defined by either the operating system or the applications. Permissions are assigned at installation time and are shared by all the components of the same application; if any of the requested permissions is not granted by the user, the application is not installed. The Android communication API offers various protection mechanisms to the different component types. In particular, all components may declare permissions which must be owned by other components requesting access to them; on the other hand, only broadcast requests may specify a permission which a receiver must hold to get the message. A limited form of permission delegation is implemented in Android by special objects known as *pending intents*: we will return on this point later on.

### 3 $\pi$ -Perms: a calculus for Android applications

We describe  $\pi$ -Perms, a simple formal calculus which captures the essence of inter-component communication in Android. We detail the connections between  $\pi$ -Perms and the Android platform in Section 3.2.

#### 3.1 Syntax and semantics

We presuppose disjoint collections of names  $m, n$  and variables  $x, y, z$ , and use the meta-variables  $u, v$  to range over *values*, i.e., both names and variables. We denote permissions with typewriter capital letters, as in PERMS, and assume they form a complete lattice with partial order  $\sqsubseteq$ , top and bottom elements  $\top$  and  $\perp$  respectively, and join and meet operators  $\sqcup$  and  $\sqcap$  respectively.

An *expression* represents a sequential program, which runs with a given set of assigned permissions and may return a value. As part of its computation, an expression may perform function calls from a pool of *function definitions*. The syntax of expressions is defined in Table 1.

$E ::=$	<i>expressions</i>	$D ::=$	<i>definitions</i>
$D \setminus E$	evaluation	$u(x \triangleleft \text{CALL}).E$	function def.
$\bar{u}\langle v \triangleright \text{RECV} \rangle$	invocation	$D \wedge D$	conjunction
let $x = E$ in $E'$	let expr.		
$(\nu n) E$	restriction		
$[\text{PERMS}] E$	perm. assign.		
$v$	value		

**Table 1.** Syntax of  $\pi$ -Perms expressions

The expression  $D \setminus E$  runs  $E$  in the pool of function definitions  $D$ . An invocation  $\bar{u}\langle v \triangleright \text{RECV} \rangle$  tries to call function  $u$ , supplying  $v$  as an argument; the

invocation succeeds only if the callee has at least permissions **RECV**. A let expression  $\text{let } x = E \text{ in } E'$  evaluates  $E$  to a name  $n$  and then behaves as  $E'$  with  $x$  substituted by  $n$ . A restriction  $(\nu n) E$  creates a fresh name  $n$  and then behaves as  $E$ . The expression  $[\text{PERMS}] E$  represents  $E$  running with permissions **PERMS**. A definition  $u(x \triangleleft \text{CALL}).E$  introduces a function  $u$ ; only callers with at least permissions **CALL** can invoke this function, supplying an argument for  $x$ . Multiple function definitions can be combined into a pool with the  $\wedge$  operator. Function definitions, “let” and  $\nu$  are binding operators for variables and names, respectively: the notions of free names  $fn$  and free variables  $fv$  arise as expected.

The formal semantics of  $\pi$ -Perms is given by the small-step reduction relation  $E \rightarrow E'$  defined in Table 2.

(R-CALL)		
$\frac{\text{CALL} \sqsubseteq \text{PERMS} \quad \text{RECV} \sqsubseteq \text{PERMS}'}{n(x \triangleleft \text{CALL}).[\text{PERMS}'] E \setminus [\text{PERMS}] \bar{n}\langle m \triangleright \text{RECV} \rangle \rightarrow [\text{PERMS}'] E\{m/x\}}$		
(R-RETURN)	(R-CONTEXT)	(R-STRUCT)
$\frac{}{\text{let } x = [\text{PERMS}] n \text{ in } E \rightarrow E\{n/x\}}$	$\frac{E \rightarrow E'}{\mathcal{C}[E] \rightarrow \mathcal{C}[E']}$	$\frac{E \Rightarrow E_1 \rightarrow E_2 \Rightarrow E'}{E \rightarrow E'}$

*Reduction contexts:*  $\mathcal{C}[\cdot] ::= \cdot \mid \text{let } x = \mathcal{C}[\cdot] \text{ in } E \mid (\nu n) \mathcal{C}[\cdot] \mid D \setminus \mathcal{C}[\cdot]$

**Table 2.** Reduction semantics for  $\pi$ -Perms

Rule (R-CALL) implements the security “cross-check” between caller and callee, which we discussed earlier: if either the caller is not assigned permissions **CALL**, or the callee is not granted permissions **RECV**, then the invocation fails. Whenever the invocation is successful, the expression runs with the permissions of the callee. The other rules are essentially standard, we just note that (R-STRUCT) closes reduction under *heating*, an asymmetric variant of the standard structural congruence relation. The heating relation  $E \Rightarrow E'$  allows to syntactically rearrange  $E$  into  $E'$ , for instance by exchanging the order of the function definitions and by extruding the scope of bound names (see Appendix A.1).

### 3.2 $\pi$ -Perms vs Android

*Intents.*  $\pi$ -Perms can encode both implicit and explicit intents. Communication in  $\pi$ -Perms is non-deterministic, in that a function invocation  $\bar{n}\langle m \triangleright \text{RECV} \rangle$  can trigger any function definition  $n(x \triangleleft \text{CALL}).E$  in the same scope, provided that the permission checks are satisfied. Technically, this non-determinism is achieved through the heating relation, which allows to liberally rearrange the pool of function definitions. Hence, communication in  $\pi$ -Perms naturally accounts for implicit intents, which represent the most interesting aspect of Android communication. Explicit intents can be recovered by univocally assigning each function definition with a distinct, unique permission: explicit communication is then encoded by requiring the callee to possess (at least) such permission.

*Components.* All of Android’s intent-based component types are represented in  $\pi$ -Perms by means of function definitions. Activities in Android may be started by invoking the methods `startActivity` or `startActivityForResult`; in our calculus we treat the two cases uniformly, by having functions always return a result. Services may either be started by `startService` or become the end-point of a long-running connection with a client through an invocation to `bindService`. The former behaviour is modelled directly in  $\pi$ -Perms by a function call, while the latter is subtler and its encoding leads to some interesting findings (see below). Broadcast communication can be captured by a sequence of function invocations: this simple treatment suffices for our present security analysis.

*Protection mechanisms.*  $\pi$ -Perms is defined around a generic complete lattice of permissions. In Android this lattice is built over permission sets, with set inclusion as the underlying partial order. The Android communication API only allows broadcast transmissions to be protected by permissions, namely requiring receivers to be granted specific permissions to get the intent. Function invocation in  $\pi$ -Perms accounts for the more general behaviour available to broadcast transmissions, since unprotected communication can be encoded simply by specifying  $\perp$  as the permission required to the callee, as in  $\bar{n}\langle m \triangleright \perp \rangle$ .

*Binders.* In Android a component can invoke the method `bindService` to establish a connection with a service and retrieve an `IBinder` object, which transparently dispatches method calls from the client to the service. This behavior is captured in  $\pi$ -Perms by relying on its provision for dynamic component creation. To illustrate, let  $D$  contain the following service definition:

$$D \triangleq s(x \triangleleft C).[P] (\nu b) (b(y \triangleleft \perp).[P] \bar{a}\langle y \triangleright \perp \rangle \setminus b) \quad (1)$$

and consider the  $\pi$ -Perms encoding of a component binding to service  $s$ :

$$a(x \triangleleft P).[P] E \wedge D \setminus [C] \text{ let } z = \bar{s}\langle n \triangleright \perp \rangle \text{ in } \bar{z}\langle n \triangleright \perp \rangle$$

Service  $s$  runs with permissions  $P$  and requires permissions  $C$  to establish a connection. When a connection is successfully established, the service returns a fresh binder  $b$ , encoded as a function granted the same permissions  $P$  as  $s$ ; later, the client can perform an invocation to  $b$  (bound to  $z$ ) to get access to the function  $a$ . The example unveils a potentially dangerous behaviour of the current Android implementation of `IBinder`’s: notice in particular that the function  $b$  may be invoked with no constraint, even though binding to  $s$  was protected by permissions  $C$ . We find this implementation potentially dangerous, since it is exposed to privilege escalation when binders are improperly disclosed.

*Pending intents.*  $\pi$ -Perms can naturally encode the simple form of permission delegation enabled by pending intents: “by giving a `PendingIntent` to another application, you are granting it the right to perform the operation you have specified as if the other application was yourself (with the same permissions and identity)” [15]. This informal description perfectly fits the previous encoding of

binders in  $\pi$ -Perms, in that any component exposed to the binder  $b$  is allowed to invoke the corresponding function and let it run with permissions  $P$ . Hence, pending intents can be modelled in the very same way as binders, and are exposed to the same weaknesses whenever they are inadvertently disclosed.

## 4 Privilege escalation (formally)

Davi *et al.* first pointed out a conceptual weakness in the Android permission system, showing that it is vulnerable to privilege escalation attacks [5]. To illustrate, consider three applications  $A$ ,  $B$  and  $C$ . Application  $A$  is granted no permission; application  $B$ , instead, is granted permission  $P$ , which is needed to access  $C$ . Apparently, data and requests from  $A$  should not be able to reach  $C$ ; on the other hand, if  $B$  can be freely accessed from  $A$ , then it may possibly act as a proxy between  $A$  and  $C$ .

We formalize a notion of safety against privilege escalation based on the IPC Inspection mechanism proposed by Felt *et al.* to dynamically prevent privilege escalation attacks on Android [11]. The idea behind IPC Inspection is simple: when an application receives a message from another application, a centralized reference monitor lowers the privileges of the recipient to the intersection of the privileges of the two interacting applications. A patched Android system implementing IPC Inspection is therefore protected against privilege escalation attacks “by design”: we then take such a system as a reference specification and state a simulation-based notion of safety on top of it. As we discuss at the end of this section, the resulting definition provides an effective proof technique for the characterization of privilege escalation safety based on non-interference in [12].

To formalize the semantics of the IPC inspection mechanism, we first annotate each function definition of a given expression with a distinct label  $\ell$  drawn from a denumerable set  $\mathcal{L}$ , disjoint from the set of values. The annotations make it possible to univocally identify the function triggered in response to each call, and hence trace the call chain. The IPC inspection semantics is then rendered formally by the labelled reduction relation  $E \xrightarrow{\alpha}_i E'$  in Table 3, where  $\alpha$  ranges uniformly over the set of annotation labels and the distinguished symbol  $\cdot \notin \mathcal{L}$ .

(R-CALL-IPC)		
$\frac{\text{RECV} \sqsubseteq \text{PERMS}' \quad \text{CALL} \sqsubseteq \text{PERMS}}{n^\ell(x \triangleleft \text{CALL}).[\text{PERMS}'] E \setminus [\text{PERMS}] \bar{n}\langle m \triangleright \text{RECV} \rangle \xrightarrow{\ell}_i [\text{PERMS} \sqcap \text{PERMS}'] E\{m/x\}}$		
(R-RETURN-IPC)	(R-CONTEXT-IPC)	(R-STRUCT-IPC)
$\text{let } x = [\text{PERMS}] n \text{ in } E \dot{\rightarrow}_i E\{n/x\}$	$\frac{E \xrightarrow{\alpha}_i E'}{\mathcal{C}[E] \xrightarrow{\alpha}_i \mathcal{C}[E']}$	$\frac{E \Rightarrow E_1 \xrightarrow{\alpha}_i E_2 \Rightarrow E'}{E \xrightarrow{\alpha}_i E'}$

**Table 3.** Reduction semantics for  $\pi$ -Perms under IPC Inspection

Note that, while the labelled transitions help tracking the dynamics of the call chains, the labels themselves do not have any import at runtime: in fact,

function invocations do not mention labels at all and the semantics is still non-deterministic. We similarly label the original semantics in Table 2.

Let now  $E_1 \asymp E_2$  note two expressions that are syntactically equal but for their granted permissions (see Appendix A.2 for a formal definition).

**Definition 1 (IPC-Simulation).** *A binary relation  $\mathcal{R}$  contained in  $\asymp$  is an IPC-simulation if and only if whenever  $E_1 \mathcal{R} E_2$  and  $E_1 \xrightarrow{\alpha} E'_1$  there exists  $E'_2$  such that  $E_2 \xrightarrow{\alpha}_i E'_2$  with  $E'_1 \mathcal{R} E'_2$ . We say that  $E_1$  is IPC-simulated by  $E_2$  (written  $E_1 \preceq_{IPC} E_2$ ) iff there exists an IPC-simulation  $\mathcal{R}$  such that  $E_1 \mathcal{R} E_2$ .*

The requirement  $E_1 \asymp E_2$  guarantees that the labels that annotate the function definitions occurring in the two expressions are consistent (i.e., the same function bears the same label in  $E_1$  and  $E_2$ ) while disregarding any difference in the assigned permissions introduced upon reduction (cf. (R-CALL) against (R-CALL-IPC)). Given the previous definition, our notion of safety is immediate: an expression  $E$  is safe if and only if all its possible executions are oblivious of IPC Inspection being enabled or not.

**Definition 2 (Safety).** *An expression  $E$  is safe against privilege escalation attacks if and only if  $E \preceq_{IPC} E$ .*

Though our definition is inspired by IPC Inspection, it reveals an important aspect which was never discussed before. Namely, we notice that improper disclosure of some specific data, such as binders or pending intents, may lead to the development of applications which are unsafe according to Definition 2. This is precisely the case of example (1) where  $b$  exercises permissions  $\mathsf{P}$ , but can be disclosed to any component which is granted permissions  $\mathsf{C}$ . A sample Android application suffering of a similar flaw is given in Appendix A.3.

Our notion of safety is already a strong property, but we target a more ambitious goal: we desire protection despite the best efforts of an active opponent. In our model an opponent is a malicious, but unprivileged, Android application installed on the same device. Notice that the term “unprivileged” is loosely used here: we are not assuming that the opponent is granted no permission at all, but rather that it is not assigned any sensitive permission beforehand (in that case, it would have no reason in escalating privileges). In a typical security analysis, one can single out all the permissions under the control of the opponent (e.g.,  $\mathsf{INTERNET}$ ) and identify the set of these permissions with  $\perp$ .

**Definition 3 (Opponent).** *A definition  $O$  is an opponent if and only if each permission assignment in  $O$  is  $\perp$ .*

**Definition 4 (Robust Safety).** *An expression  $E$  is robustly safe against privilege escalation attacks if and only if  $O \setminus E$  is safe for all opponents  $O$ .*

*Privilege escalation and non-interference.* As we anticipated, a recent paper by Fragkaki *et al.* [12] proposes a definition of safety against privilege escalation inspired by the classic notion of non-interference for information flow control. Their

definition essentially demands that any call chain ending in a “high” (permission-protected) component exists in a system only if it exists in a variant of same system, where the “low” (unprivileged) components have been pruned away. We can rephrase their notion in our setting and prove that our definition implies, and hence may be employed as a proof technique for, theirs.

Let  $|E|_\ell$  denote the expression obtained from  $E$  by erasing all the function definitions labelled with  $\ell' \neq \ell$  and which are granted permissions  $P \sqsubset \text{CALL}$ , where  $\text{CALL}$  are the permissions required to invoke the function identified by  $\ell$ .

**Definition 5 (NI-Safety).** *An expression  $E$  is NI-safe if and only if, for every  $\ell$  occurring in  $E$  and for every reduction sequence  $E \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_n} E_n \xrightarrow{\ell} E_{n+1}$ , there exist  $E'_1, \dots, E'_{n+1}$  such that  $|E|_\ell \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_n} E'_n \xrightarrow{\ell} E'_{n+1}$ .*

**Proposition 1 (Safety vs NI-safety).** *Safety implies NI-safety.*

The proof is in Appendix A.4. We can thus confirm that the IPC Inspection mechanism enforces a reasonable semantic security property and justify further our choice of taking it as the building block for our safety notion. With respect to NI-safety, our notion has the important advantage of enabling a powerful form of coinductive reasoning, which is central to proving our main result (Theorem 2 below). A still open question is if the two notions of safety are actually equivalent. We notice that for non-deterministic transition systems (bi)simulation-based equivalences are typically finer than trace equivalences, but at the time of writing we were not able to identify a counterexample in our setting.

## 5 Preventing privilege escalation by types and effects

*Types and typing environments.* A type  $\tau$  may be either  $\text{Un}$  or a function type  $\text{Fun}(\text{CALL}, \tau \rightarrow \tau')^{\text{SECR}}$ . Type  $\text{Un}$  is the base type, which is used both as a building block for function types and to encompass all the data which are under the control of the opponent. Types of the form  $\text{Fun}(\text{CALL}, \tau \rightarrow \tau')^{\text{SECR}}$  are inhabited by functions which input arguments of type  $\tau$  and return results of type  $\tau'$ . Functions with this type can be invoked only by callers which are granted at least permissions  $\text{CALL}$ , and should only be disclosed to components running with at least permissions  $\text{SECR}$ . We define the *secrecy level* of a type  $\tau$ , written  $\mathcal{S}(\tau)$ , as expected, by having  $\mathcal{S}(\text{Un}) = \perp$  and  $\mathcal{S}(\text{Fun}(\text{CALL}, \tau \rightarrow \tau')^{\text{SECR}}) = \text{SECR}$ . A typing *environment*  $\Gamma$  is a finite map from values to types. The *domain* of  $\Gamma$ , written  $\text{dom}(\Gamma)$ , is the set of the values on which  $\Gamma$  is defined.

*Typing values.* The typing rules for values are simple and given in Table 4.

$\frac{\Gamma(v) = \tau}{\Gamma \vdash v : \tau}$	$\frac{\Gamma \vdash v : \tau \quad \mathcal{S}(\tau) = \perp}{\Gamma \vdash v : \text{Un}}$
---	--

**Table 4.** Typing rules for values



(T-PROJ) is standard, while (T-PUB) makes it possible to treat all public data as “untyped”, since they may possibly be disclosed to the opponent.

*Typing expressions.* The typing rules for expressions are in Table 5. The main judgement  $\Gamma \vdash_P E : \tau \blacktriangleright Q$  is read as: expression  $E$ , running with permissions  $P$ , has type  $\tau$  in  $\Gamma$  and exercises at most permissions  $Q$  throughout its execution. We also define an auxiliary judgement  $\Gamma \vdash D$  to be read as: definition  $D$  is well-formed in  $\Gamma$ . The two judgement forms are mutually dependent.

(T-DEF)		
$\frac{\Gamma \vdash u : \text{Fun}(\text{CALL}, \tau \rightarrow \tau')^{\text{SECR}} \quad \Gamma, x : \tau \vdash_\top E : \tau' \blacktriangleright Q \quad Q \sqsubseteq \text{CALL} \sqcup \text{SECR} \quad \text{CALL} \sqcup \text{SECR} = \perp \Rightarrow \Gamma, x : \text{Un} \vdash_\top E : \text{Un} \blacktriangleright \perp \quad x \notin \text{dom}(\Gamma)}{\Gamma \vdash u(x \triangleleft \text{CALL}).E}$		
		(T-CONJ)
		$\frac{\Gamma \vdash D_1 \quad \Gamma \vdash D_2}{\Gamma \vdash D_1 \wedge D_2}$
(T-EVAL)	(T-CALL)	
$\frac{\Gamma \vdash D \quad \Gamma \vdash_P E : \tau \blacktriangleright Q}{\Gamma \vdash_P D \setminus E : \tau \blacktriangleright Q}$	$\frac{\Gamma \vdash u : \text{Fun}(\text{CALL}, \tau \rightarrow \tau')^{\text{SECR}} \quad \Gamma \vdash v : \tau \quad \perp \sqsubseteq \text{RECV} \sqcup \text{SECR} \quad \text{CALL} \sqcup \text{SECR} \sqsubseteq P}{\Gamma \vdash_P \bar{u}\langle v \triangleright \text{RECV} \rangle : \tau' \blacktriangleright \text{CALL} \sqcup \text{SECR}}$	
(T-VAL)	(T-FAIL)	(T-PERMS)
$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash_P v : \tau \blacktriangleright \perp}$	$\frac{\Gamma \vdash u : \text{Fun}(\text{CALL}, \tau \rightarrow \tau')^{\text{SECR}} \quad \Gamma \vdash v : \tau'' \quad \text{RECV} \sqcup \text{SECR} = \perp \Rightarrow \mathcal{S}(\tau'') = \perp \quad \text{CALL} \not\sqsubseteq P}{\Gamma \vdash_P \bar{u}\langle v \triangleright \text{RECV} \rangle : \text{Un} \blacktriangleright P}$	$\frac{\Gamma \vdash_Q E : \tau \blacktriangleright R \quad Q \sqsubseteq P}{\Gamma \vdash_P [Q] E : \tau \blacktriangleright R}$
(T-LET)	(T-RESTR)	
$\frac{\Gamma \vdash_P E : \tau \blacktriangleright Q \quad \Gamma, x : \tau \vdash_P E' : \tau' \blacktriangleright R \quad x \notin \text{dom}(\Gamma)}{\Gamma \vdash_P \text{let } x = E \text{ in } E' : \tau' \blacktriangleright Q \sqcup R}$	$\frac{\Gamma, n : \tau \vdash_P E : \tau' \blacktriangleright Q \quad n \notin \text{dom}(\Gamma)}{\Gamma \vdash_P (\nu n) E : \tau' \blacktriangleright Q}$	
(T-DEF-UN)	(T-CALL-UN)	
$\frac{\Gamma \vdash u : \text{Un} \quad \Gamma, x : \text{Un} \vdash_\perp E : \text{Un} \blacktriangleright \perp \quad x \notin \text{dom}(\Gamma)}{\Gamma \vdash u(x \triangleleft \text{CALL}).E}$	$\frac{\Gamma \vdash u : \text{Un} \quad \Gamma \vdash v : \text{Un}}{\Gamma \vdash_\perp \bar{u}\langle v \triangleright \text{RECV} \rangle : \text{Un} \blacktriangleright \perp}$	

**Table 5.** Typing rules for definitions and expressions

We first note that our effect system discriminates between *granted* and *exercised* permissions. For instance, the expression  $a(x \triangleleft \perp).[P] \bar{b}\langle n \triangleright \perp \rangle \setminus E$  could either be well-typed or not, even though the function  $a$  is publicly known, but is granted permissions  $P \sqsubseteq \perp$ . The crux here is if the permissions  $P$  must be actually exercised or not to perform the invocation to  $b$ .

Apparently, we could enforce protection against privilege escalation by simply checking for each function definition that the privileges exercised by the function body are at most equal to the privileges required to invoke the function.

However, since binders and pending intents allow indiscriminate access to potentially privileged components, our type system must also assign an appropriate secrecy level to these sensitive data and prevent their inadvertent disclosure. It turns out that in rule (T-DEF) we must actually check that the permissions  $\mathbf{Q}$  exercised by the function body must be at most equal to the join between the permissions  $\mathbf{CALL}$ , needed to pass the security runtime checks upon invocation, and the permissions  $\mathbf{SECR}$ , needed to learn the name of the function.

Interestingly, the opponent can play an active role in trying to get binders and pending intents under its control. In particular, by using rules (T-DEF-UN) and (T-CALL-UN), it can define arbitrary new functions and invoke existing ones, completely disregarding the restrictions enforced by typing. Protecting well-typed components requires then some special care: for instance, in rule (T-DEF) we must type-check public functions under the additional assumption that their input parameter is provided by the opponent with type  $\mathbf{Un}$ ; of course, in this case no privilege must be exercised. Similarly, in rule (T-CALL) we cannot trust the return type of a function when the invocation can be dispatched to the opponent: this justifies the third premise of the rule.

Rule (T-FAIL) allows to provide an argument of arbitrary type to any function which will never be invoked at runtime, since the caller is granted permissions  $\mathbf{P}$ , but the function requires permissions  $\mathbf{CALL} \not\sqsubseteq \mathbf{P}$  to be invoked. Again, the information  $\mathbf{CALL}$  in the function type can be trusted only when the function is not defined by the opponent, hence some additional care is needed to prevent secrecy violations in that case (see the third premise of the rule).

We conclude the description of the type system with an important remark on expressiveness. Some of the constraints imposed by our typing rules are rather restrictive for practical use, but are central to enforcing the conditions of Definition 2 and its robust variant. Our implementation, however, features a number of escape hatches based on Java annotations to keep programming practical, much in the spirit of the declassification/endorsement constructs customary to the literature on information-flow control. We discuss this point further in Section 6.

*Formal results.* The safety result below follows by a “simulation-aware” variant of a standard Subject Reduction theorem for our type system, which captures the step-by-step relationships between the standard semantics and our reference semantics. The proof relies on a co-inductive argument enabled by the Subject Reduction theorem: full details can be found in the companion technical report.

**Theorem 1 (Type Safety).** *If  $\Gamma \vdash_{\top} E : \tau \blacktriangleright \mathbf{P}$ , then  $E \preceq_{IPC} E$ .*

The next result states that our type system does not constrain the opponent. Its proof follows by a simple structural induction.

**Lemma 1 (Opponent Typability).** *Let  $O$  be an opponent and let  $\Gamma \vdash u : \mathbf{Un}$  for all  $u \in \text{fnfv}(O)$ , then  $\Gamma \vdash O$ .*

By combining the two previous results, we can prove our main theorem.

**Theorem 2 (Robust Safety).** *Let  $\mathcal{S}(\tau) = \perp$  for every  $u$  such that  $\Gamma(u) = \tau$ . If  $\Gamma \vdash_{\top} E : \tau \blacktriangleright \mathbf{P}$ , then  $E$  is robustly safe against privilege escalation attacks.*

## 6 Implementation

We have implemented the type system as a tool (**Lintent**) designed as a plug-in for Android **Lint**, the widely popular utility distributed with Android’s ADT.

**Lintent** performs a number of static checks over permissions usage, analyzing the application source code and the manifest permission declarations, and eventually warning the developer in case of potential attack surfaces for privilege escalation scenarios. As a byproduct of its analysis, **Lintent** is able to detect over-privileged or under-privileged applications, and suggest fixes. Additionally, **Lintent** infers and records the types of data injected into and extracted from intents, while tracking the flow of inter-component message passing. This is needed to prevent privilege escalation attacks exploiting improper disclosure of binders or pending intents, and at the same time proves very effective in detecting common programming errors related to misuse of intents [16].

**Lintent** analyzes Java source code, mainly because it is being developed as part of a larger project aimed at developing type-based verification tools for Android developers. In principle, the same analysis could be performed on the Java bytecode, though reasoning about types at the bytecode level is arguably more demanding than at source level [14]. Below, we give a brief overview of the main features of the tool and of the the main challenges we had to face during its development.

*Type reconstruction.* The hardest challenge for the implementation is related to the widespread use of “untyped” coding patterns supported by the current Android API. Consider, for instance, a simple scenario of intent usage with multiple data types:

```
class SenderActivity extends Activity {
    static class MySer implements Serializable { ... }

    void mySenderMethod() {
        Intent i = new Intent(this, ReceiverActivity.class);
        i.putExtra("k1", 3);
        i.putExtra("k2", "some_string");
        i.putExtra("k3", new MySer());
        startActivity(i,0);
    }
}
```

On the recipient side, intent “extras” are retrieved by freely accessing the intent as if it was a dictionary, so the receiver may actually retrieve data of unexpected type and fail at runtime, or disregard altogether some keys provided by the sender [16].

```
class ReceiverActivity extends Activity {
    static class WS implements Serializable { ... }
```

```

void onCreate(Bundle savedInstanceState) {
    Intent i = getIntent();
    String k1 = i.getStringExtra("k1"); // run-time type error!
    WS o = (WS)i.getSerializableExtra("k3"); // dynamic cast fails!
    // data associated to k2 is never extracted!
}
}

```

The example highlights a total lack of static control over standard intents manipulation operations: with these premises, no type-based analysis can be soundly performed. For this reason, intents are treated in **Lintent** as record types of the form  $\{k_1 : T_1, \dots, k_n : T_n\}$ , where each  $k_i$  is a string constant and each  $T_i$  is a Java type. This enforces a much stronger discipline on data passing between components, which is consistent with our type system, where a function type  $\text{Fun}(\text{CALL}, \tau \rightarrow \tau')^{\text{SECR}}$  constrains the caller in providing an argument of type  $\tau$  and the callee in returning a result of type  $\tau'$ . A similar discipline is crucial in Android applications to protect the secrecy of binders and pending intents.

Notice that, since the `putExtra` method is overloaded to different types, the type of the second argument of each call must be reconstructed in order to keep track of the actual type of the value bound to each key. As a valuable byproduct of this analysis, **Lintent** is able to warn the user in case of ill-typed or dangerous manipulations of intents.

*Partial evaluation.* As noted above, each piece of data put into an intent must be bound to a key, hence an intent object can be seen as a dictionary of the form  $\{k_1 \mapsto v_1, \dots, k_n \mapsto v_n\}$ . Unfortunately, the dictionary keys are run-time (String) objects and therefore plain expressions in Java. Whether they happen to be string literals or the result of complex method calls computing a String object is irrelevant: in any case they belong to the run-time world. The very same problem arises for result codes and Intent constructor invocations: both the sender component and the recipient class object supplied as arguments could be results of computations, and the same holds true for action strings in case of implicit intent construction. Partial evaluation is required for reconstructing the intent record type labels described above.

*API signatures and permissions.* Implementing the rules of the type system for  $\pi$ -Perms requires a preliminary analysis to detect the corresponding patterns in the Android source code. The analysis is far from trivial given the complexity of the Android communication API, which offers several different patterns to implement inter-component communication. Moreover, many Android API calls require non-empty permission sets and must be detected and tracked by our tool: **Lintent** retrieves a set of mappings between API method signatures and permissions from a set of external files<sup>2</sup>, which are thus updatable with no need to rebuild the tool. Finally, **Lintent** must perform type resolution for third-party libraries: access to `jar` files must be granted to the tool to let it inspect the contents of imported packages and classes through the `javap` disassembler.

<sup>2</sup> Currently such permission map files are those distributed with Stowaway [10].

*Java annotations support.* We rely on Java annotations to provide some escape hatches from the tight discipline imposed by `Lintent`. Several privileged components intentionally expose functionalities, thus we define annotations of the form `@priv{endorse="P"}` to mark methods such as `onCreate()` with a set of permissions `P` that the type checked will disregard. More precisely, if the method exercises the permissions set `Q`, the associated component is deemed well-typed as long as it is protected with at least permissions `Q \setminus P`. A similar treatment is implemented for pending intents based on the annotation `@priv{declassify="P"}`, to lower the secrecy level of such objects computed by `Lintent`.

## 7 `Lintent`: typing experiments and findings

At the time of writing `Lintent` is able to type-check activities, started services and broadcast receivers. The current prototype should be considered in alpha stage, as we are currently performing tests, fixing bugs and adding support for some missing Java language features, which unfortunately are important to type-check real Android applications. Still, we were able to analyze some existing open-source applications from the Google Play store and identify privilege escalation attacks on them. In our case studies we performed a code refactoring to avoid the usage of some Java features which are still unsupported by `Lintent`, like reflective calls and nested classes. However, our findings are confirmed by running the original applications on a Nexus device and implementing an attack.

The first case study we consider is `APN-Switch`, a widget that allows to enable and disable the device data connection with a click. Of course, these network operations are sensitive, hence the application requires the permission `CHANGE_NETWORK_STATE` to be installed. Unfortunately, `APN-Switch` is exposed to privilege escalation attacks: an unprivileged malicious application can forge an intent on the action string `ch.blinkenlights.android.apnswitch.CLICK` and simulate a click of the user on the widget, thus enabling (or disabling) the device data connection as if it were granted the `CHANGE_NETWORK_STATE` permission.

Our second case study is `Wifi Fixer`, a small application aimed at fixing several problems with the Android wifi. Also `Wifi Fixer` suffers of privilege escalation attacks, since it requires the permission `CHANGE_WIFI_STATE` to toggle on and off the wifi connection, but any unprivileged application can send an intent on the action string `org.wahtod.wififixer.ACTION_WIFI_OFF` to disconnect the wifi. Interestingly, the widget handling the wifi connection is declared as an internal component, hence it cannot receive intents from third-party applications; however, a public broadcast receiver in the application can act as a proxy to the widget, thus allowing to escalate privileges.

Both `APN-Switch` and `Wifi Fixer` are released on the official Google Play store, hence available to a wide audience. We argue that a tool as `Lintent` can prove helpful in assisting well-meaning developers in identifying potential attack surfaces for privilege escalation, and many other common programming mistakes, way before their applications reach the market. We are also testing `Lintent` on some simpler examples exhibiting potentially harmful behaviours, like the

public disclosure of pending intents enabling access to privileged components. For instance, the application in Appendix A.3 is correctly identified as dangerous.

## 8 Related work

The literature on Android application security is substantial, as reported in a recent survey by Enck [6].

*Android permissions.* Davi *et al.* [5] were the first to point out the weaknesses of the Android permission system with respect to privilege escalation attacks. Later, Felt *et al.* proposed IPC Inspection as a runtime protection mechanism [11]. Though effective, IPC Inspection may induce substantial performance overhead, as it requires to keep track of different application instances. In a recent paper, Bugiel *et al.* describe a sophisticated runtime framework for enforcing protection against privilege escalation attacks [2]. Notably, their solution comprises countermeasures against colluding applications, an aspect which is neglected by both IPC Inspection and **Lintent**. Providing such guarantees, however, requires a centralized solution built over the operating system. Our approach is complementary: runtime protection is useful against malicious applications which reach the Android market, while static analysis techniques can prove helpful for well-meaning developers. Finally, Felt *et al.* proposed Stowaway, a tool for detecting overprivilege in Android applications [10].

*Android communication.* Chin *et al.* [4] were the first to study the threats related to the Android message-passing system. They provide also a tool, ComDroid, which is able to detect potential vulnerabilities in the usage of intents. ComDroid does not provide any formal guarantee about the effectiveness of the proposed secure communication guidelines; in our work, instead, we reason about intents usage in a formal calculus and we are able to confirm many previous observations as sound programming practices. The robustness of inter-component communication in Android has been studied also by Maji *et al.* through fuzzy testing techniques, exposing some interesting findings [16].

*Formal models.*  $\pi$ -Perms is partly inspired by a core formal language proposed by Chaudhuri [3]. With respect to Chaudhuri’s model,  $\pi$ -Perms provides a more thorough treatment of the Android system, including implicit communication, runtime registration of new components, service binding and pending intents. In later work, Fuchs *et al.* build on the calculus proposed by Chaudhuri to implement SCanDroid, a provably sound static checker of information-flow properties of Android applications [13]. At the time of writing, however, SCanDroid is not publicly available. Another work by Fragkaki *et al.* discusses a number of enhancements over the Android permission system and validates their effectiveness in an abstract model [12] (cf. Section 4). The focus of the work remains on runtime protection mechanisms, however, as opposed to static analysis. Finally, Armando *et al.* proposed a formal model of the Android operating system and

a verification technique based on history expressions [1]. However, any specific security analysis is left for future work and no implementation is provided.

## 9 Conclusions

We have proposed a sound type-based analysis technique targeted at the static detection of privilege escalation attacks on Android, and developed **Lintent**, a prototype security type-checker which implements our analysis. Our tool addresses a number of engineering challenges which are central to the practical development of any sound type-checker for Android applications. We showed the effectiveness of our tool by unveiling real attacks on existing applications.

As part of our future work, we want to focus on the study of robust declassification and endorsement programming patterns in our formal framework, to assess the impact on security of the Java annotations discussed in Section 6. On the practical side, we want to further develop **Lintent** and add support for many features of the Android platform which are still missing. We also plan to integrate **Lintent** with a frontend to a decompiler as **ded** [8] to support the analysis of third-party applications.

## References

1. Armando, A., Costa, G., Merlo, A.: Formal modeling and verification of the Android security framework. In: TGC (2012)
2. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R., Shastri, B.: Towards taming privilege-escalation attacks on Android. In: NDSS (2012)
3. Chaudhuri, A.: Language-based security on Android. In: PLAS. pp. 1–7 (2009)
4. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: MobiSys. pp. 239–252 (2011)
5. Davi, L., Dmitrienko, A., Sadeghi, A.R., Winandy, M.: Privilege escalation attacks on Android. In: ISC. pp. 346–360 (2010)
6. Enck, W.: Defending users against smartphone apps: Techniques and future directions. In: ICISS. pp. 49–70 (2011)
7. Enck, W., Gilbert, P., gon Chun, B., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.: Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: OSDI. pp. 393–407 (2010)
8. Enck, W., Ocateau, D., McDaniel, P., Chaudhuri, S.: A study of Android application security. In: USENIX Security Symposium (2011)
9. Enck, W., Ongtang, M., McDaniel, P.D.: Understanding Android security. IEEE Security & Privacy 7(1), 50–57 (2009)
10. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: CCS. pp. 627–638 (2011), <http://www.android-permissions.org/>
11. Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission re-delegation: Attacks and defenses. In: USENIX Security Symposium (2011)
12. Fragkaki, E., Bauer, L., Jia, L., Swasey, D.: Modeling and enhancing Android’s permission system. In: ESORICS. pp. 1–18 (2012)
13. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: Scandroid: Automated security certification of Android applications (2009), Technical report, University of Maryland.

14. Gagnon, E., Hendren, L.J., Marceau, G.: Efficient inference of static types for Java bytecode. In: SAS. pp. 199–219 (2000)
15. Google Inc: Reference documentation for `android.app.PendingIntent`. <http://developer.android.com/reference/android/app/PendingIntent.html>
16. Maji, A.K., Arshad, F.A., Bagchi, S., Rellermeyer, J.S.: An empirical study of the robustness of inter-component communication in Android. In: DSN (2012)



## A Additional material

### A.1 The heating relation

We define the heating relation  $\Rightarrow$  as the smallest preorder on expressions closed under the rules in Table 6. We write  $E \equiv E'$  if and only if  $E \Rightarrow E'$  and  $E' \Rightarrow E$ .

$\frac{(H\text{-CONTEXT}) \quad E \Rightarrow E'}{\mathcal{C}[E] \Rightarrow \mathcal{C}[E']}$	$\frac{(H\text{-EXTR-1}) \quad n \notin fn(E')}{\text{let } x = (\nu n) E \text{ in } E' \Rightarrow (\nu n) (\text{let } x = E \text{ in } E')}$
$\frac{(H\text{-EXTR-2}) \quad n \notin fn(D)}{D \setminus (\nu n) E \Rightarrow (\nu n) (D \setminus E)}$	$\frac{(H\text{-FLIP-1})}{[\text{PERMS}] (\nu n) E \Rightarrow (\nu n) [\text{PERMS}] E}$
$\frac{(H\text{-FLIP-2})}{[\text{PERMS}] (D \setminus E) \Rightarrow D \setminus [\text{PERMS}] E}$	$\frac{(H\text{-COMM})}{(D_1 \wedge D_2) \setminus E \equiv (D_2 \wedge D_1) \setminus E}$
$\frac{(H\text{-ASSOC})}{(D_1 \wedge D_2) \wedge D_3 \setminus E \equiv D_1 \wedge (D_2 \wedge D_3) \setminus E}$	$\frac{(H\text{-CONJ})}{D_1 \setminus (D_2 \setminus E) \equiv (D_1 \wedge D_2) \setminus E}$
$\frac{(H\text{-MOVE})}{D \setminus (\text{let } x = E \text{ in } E') \equiv \text{let } x = (D \setminus E) \text{ in } E'}$	
$\frac{(H\text{-DISTR})}{[\text{PERMS}] \text{let } x = E \text{ in } E' \Rightarrow \text{let } x = [\text{PERMS}] E \text{ in } [\text{PERMS}] E'}$	

**Table 6.** Heating relation for  $\pi$ -Perms

We briefly discuss some aspects of the heating relation: rules (H-EXTR-1) and (H-EXTR-2) formalize scope extrusion, much in the same spirit as in the pi-calculus. Rules (H-FLIP-1) and (H-FLIP-2) perform some house-keeping needed to export new names and functions dynamically created by a running expression. Rules (H-COMM) and (H-ASSOC) are used in combination with (H-CONJ) to liberally rearrange a pool of function definitions, by ignoring their order. Rule (H-MOVE) is needed both to perform function calls inside the reduction context of a let expression (when read from left to right) and to export new function definitions (when read from right to left). Rule (H-DISTR) is borrowed from [3] and it distributes permission assignments over a let expression.

## A.2 Equivalence up to granted permissions

We define the equivalence up to granted permissions relation  $\asymp$  as the smallest equivalence relation on expressions closed under the rules in Table 7.

$$\begin{array}{c}
\text{[P]} E \asymp \text{[Q]} E \qquad \frac{E \asymp E'}{(\nu n) E \asymp (\nu n) E'} \qquad \frac{E \asymp E'}{D \setminus E \asymp D \setminus E'} \\
\\
\frac{E_1 \asymp E'_1 \quad E_2 \asymp E'_2}{\text{let } x = E_1 \text{ in } E_2 \asymp \text{let } x = E'_1 \text{ in } E'_2}
\end{array}$$

**Table 7.** Equivalence up to granted permissions

## A.3 Sample misuse of Pending Intents

Consider an activity `FirstApp`, which is granted the `BLUETOOTH` permission. This permission is needed to access the system component `BluetoothAdapter`, which manages the bluetooth interface of an Android device, and allows to turn it on and off. The activity `FirstApp` does not directly exercise the permission `BLUETOOTH`, but it delegates it to another application. Hence, `FirstApp` invokes the factory method `getActivity` to create a pending intent `pInt`, which grants access to the system component `BluetoothAdapter`. The pending intent is then wrapped into a standard intent `sndApp`, which is sent to a second application.

```

public class FirstApp extends Activity {

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // create a pending intent to access the BluetoothAdapter
        Intent myInt = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
        PendingIntent pInt = PendingIntent.getActivity(this, 0, myInt, 0);

        // package the pending intent into a standard intent
        Intent sndApp = new Intent("ACT_STRING");
        sndApp.putExtra("pending", pInt);

        // send the pending intent
        startActivity(sndApp);
    }
}

```

A second activity `SecondApp` can then retrieve the intent through an invocation to `getIntent`, and extract the inner pending intent. Then, the activity can invoke the method `send` on the pending intent to get access to the `BluetoothAdapter`, even if it is not granted the required `BLUETOOTH` permission.

```

public class SecondApp extends Activity {

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // retrieve the pending intent
        Intent i = getIntent();
        PendingIntent pi = (PendingIntent) i.getParcelableExtra("pending");

        // launch the BluetoothAdapter
        if (pi != null)
            try {
                pi.send();
            }
            catch (CanceledException e) {
                // do something here
            }
    }
}

```

The problem with the previous code is that `FirstApp` packages the pending intent `pInt` into an *implicit* intent, thus any component registered over the action string `ACT_STRING` can intercept it and abuse the `BLUETOOTH` permission.

#### A.4 Proof of Proposition 1

*Statement.* Safety implies NI-safety.

*Proof.* Let  $E \preceq E$  and assume  $E \xrightarrow{\alpha_1} E_1 \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_n} E_n \xrightarrow{\ell} E_{n+1}$ . Since  $E \preceq E$ , we know that  $E \xrightarrow{\alpha_1}_i E'_1 \xrightarrow{\alpha_2}_i \dots \xrightarrow{\alpha_n}_i E'_n \xrightarrow{\ell}_i E'_{n+1}$  for some  $E'_1, \dots, E'_{n+1}$  such that  $E_1 \asymp E'_1, \dots, E_{n+1} \asymp E'_{n+1}$ . By definition of the semantics  $\xrightarrow{\alpha}_i$ , we know that all the functions invoked in the call chain identified by  $\alpha_1, \dots, \alpha_n$  must be granted at least the permissions `CALL` needed to invoke  $\ell$ . Hence, such function definitions are present also in  $|E|_\ell$  and we can mimic the very same trace there.

## B Soundness proofs

We detail a full proof of soundness for the type-and-effect system of Section 5. In the next results we unfold the (R-CONTEXT)/(H-CONTEXT) rule from Table 2/ 6 into a number of different reduction/heating rules, one for each possible context.

### B.1 Basic results

**Notation 1.** *We adopt the following notational conventions:*

- (i) *We often write  $\Gamma \vdash E : \tau \blacktriangleright P$  when  $\Gamma \vdash_Q E : \tau \blacktriangleright P$  for some  $Q$ .*
- (ii) *We write  $\Gamma \vdash_Q^\xi E : \tau \blacktriangleright P$  if  $\xi$  is a type derivation ending with  $\Gamma \vdash_Q E : \tau \blacktriangleright P$ .*
- (iii) *We write  $\Gamma \vdash_Q \mathcal{J}$  to stand for any of the following judgements:*
  - $\Gamma \vdash u : \tau$  for some  $u$  and  $\tau$
  - $\Gamma \vdash D$  for some  $D$
  - $\Gamma \vdash_Q E : \tau \blacktriangleright P$  for some  $E, \tau$  and  $P$ .

**Proposition 2 (Uniqueness of Function Types).** *If  $\Gamma \vdash u : \text{Fun}(\text{CALL}, \tau_1 \rightarrow \tau_2)^{\text{SECR}}$  and  $\Gamma \vdash u : \text{Fun}(\text{CALL}', \tau'_1 \rightarrow \tau'_2)^{\text{SECR}'}$ , then  $\text{CALL} = \text{CALL}'$ ,  $\tau_1 = \tau'_1$ ,  $\tau_2 = \tau'_2$  and  $\text{SECR} = \text{SECR}'$ .*

*Proof.* Immediate by inspection of the type rules, since the only rule which can derive function types is (T-PROJ) and  $\Gamma$  is a map from values to types.

**Proposition 3 (Soundness of Secrecy Levels).** *If  $\Gamma \vdash u : \tau$  and  $\Gamma \vdash u : \tau'$ , then  $\mathcal{S}(\tau) = \mathcal{S}(\tau')$ .*

*Proof.* By induction on the sum of the depth of the derivations of  $\Gamma \vdash u : \tau$  and  $\Gamma \vdash u : \tau'$ . The only interesting case is when  $\Gamma \vdash u : \tau$  was derived by (T-PROJ) and  $\Gamma \vdash u : \tau'$  was derived by (T-PUB), or vice-versa. Without loss of generality, consider the first possibility: in this case we know that  $\Gamma \vdash u : \tau$  by the premise  $\Gamma(u) = \tau$  and  $\Gamma \vdash u : \tau'$  with  $\tau' = \text{Un}$  by the premise  $\Gamma \vdash u : \tau''$  for some  $\tau''$  such that  $\mathcal{S}(\tau'') = \perp$ . By inductive hypothesis we then have  $\mathcal{S}(\tau) = \mathcal{S}(\tau'') = \perp$ . We conclude by noting that  $\mathcal{S}(\tau') = \mathcal{S}(\text{Un}) = \perp = \mathcal{S}(\tau)$ .

**Lemma 2 (Weakening).** *If  $\Gamma \vdash_Q \mathcal{J}$  and  $u \notin \text{dom}(\Gamma)$ , then  $\Gamma, u : \tau \vdash_Q \mathcal{J}$ . (Moreover, the effects computed throughout the entire type derivation do not change.)*

*Proof.* By a standard induction on the derivation of  $\Gamma \vdash_Q \mathcal{J}$ .

**Lemma 3 (Substitution).** *Let  $\Gamma, x : \tau \vdash_Q \mathcal{J}$  with  $x \notin \text{dom}(\Gamma)$ . If  $\Gamma \vdash n : \tau$ , then  $\Gamma \vdash_Q \mathcal{J}\{n/x\}$ . (Moreover, the effects computed throughout the entire type derivation do not change.)*

*Proof.* By a standard induction on the derivation of  $\Gamma, x : \tau \vdash_Q \mathcal{J}$ .

**Lemma 4 (Heating Preserves Typing).** *If  $\Gamma \vdash_Q E : \tau \blacktriangleright P$  and  $E \Rightarrow E'$ , then  $\Gamma \vdash_Q E' : \tau \blacktriangleright P$ .*

*Proof.* By induction on the derivation of  $E \Rightarrow E'$ . The reflexivity case is trivial and the transitivity case immediately follows by inductive hypothesis, so we focus on the remaining rules:

*Case (H-EVAL):* let  $D \setminus E \Rightarrow D \setminus E'$  by the premise  $E \Rightarrow E'$ . Since  $\Gamma \vdash_Q D \setminus E : \tau \blacktriangleright P$ , we have  $\Gamma \vdash D$  and  $\Gamma \vdash_Q E : \tau \blacktriangleright P$  by (T-EVAL). By inductive hypothesis  $\Gamma \vdash_Q E' : \tau \blacktriangleright P$ , hence  $\Gamma \vdash_Q D \setminus E' : \tau \blacktriangleright P$  by (T-EVAL);

*Case (H-LET):* assume  $\text{let } x = E \text{ in } E'' \Rightarrow \text{let } x = E' \text{ in } E''$  by the premise  $E \Rightarrow E'$ . Since  $\Gamma \vdash_R \text{let } x = E \text{ in } E'' : \tau \blacktriangleright \text{PERMS}$ , we have  $\Gamma \vdash_R E : \tau' \blacktriangleright P$  and  $\Gamma, x : \tau' \vdash_R E'' : \tau \blacktriangleright Q$  with  $P \sqcup Q = \text{PERMS}$  by (T-LET). By inductive hypothesis  $\Gamma \vdash_R E' : \tau' \blacktriangleright P$ , hence  $\Gamma \vdash_R \text{let } x = E' \text{ in } E'' : \tau \blacktriangleright \text{PERMS}$  by (T-LET);

*Case (H-RESTR):* let  $(\nu n) E \Rightarrow (\nu n) E'$  by the premise  $E \Rightarrow E'$ . Since  $\Gamma \vdash_Q (\nu n) E : \tau' \blacktriangleright \text{PERMS}$ , we have  $\Gamma, n : \tau \vdash_Q E : \tau' \blacktriangleright \text{PERMS}$  by (T-RESTR). By inductive hypothesis  $\Gamma, n : \tau \vdash_Q E' : \tau' \blacktriangleright \text{PERMS}$ , hence  $\Gamma \vdash_Q (\nu n) E' : \tau' \blacktriangleright \text{PERMS}$  by (T-RESTR).

*Case (H-EXTR-1):* assume  $\text{let } x = (\nu n) E_1 \text{ in } E_2 \Rightarrow (\nu n) (\text{let } x = E_1 \text{ in } E_2)$  with  $n \notin \text{fn}(E_2)$ . Since  $\Gamma \vdash_R \text{let } x = (\nu n) E_1 \text{ in } E_2 : \tau_2 \blacktriangleright \text{PERMS}$ , we have  $\Gamma \vdash_R (\nu n) E_1 : \tau_1 \blacktriangleright P$  and  $\Gamma, x : \tau_1 \vdash_R E_2 : \tau_2 \blacktriangleright Q$  with  $P \sqcup Q = \text{PERMS}$  and  $x \notin \text{dom}(\Gamma)$  by (T-LET). The former judgement can be derived only by (T-RESTR), hence we have  $\Gamma, n : \tau \vdash_R E_1 : \tau_1 \blacktriangleright P$  with  $n \notin \text{dom}(\Gamma)$ . Now we apply Lemma 2 (Weakening) to derive  $\Gamma, n : \tau, x : \tau_1 \vdash_R E_2 : \tau_2 \blacktriangleright Q$  from  $\Gamma, x : \tau_1 \vdash_R E_2 : \tau_2 \blacktriangleright Q$ , hence we have  $\Gamma, n : \tau \vdash_R \text{let } x = E_1 \text{ in } E_2 : \tau_2 \blacktriangleright \text{PERMS}$  by (T-LET) and we conclude  $\Gamma \vdash_R (\nu n) (\text{let } x = E_1 \text{ in } E_2) : \tau_2 \blacktriangleright \text{PERMS}$  by (T-RESTR);

*Case (H-EXTR-2):* let  $D \setminus (\nu n) E \Rightarrow (\nu n) (D \setminus E)$  with  $n \notin \text{fn}(D)$ . Given that  $\Gamma \vdash_Q D \setminus (\nu n) E : \tau' \blacktriangleright \text{PERMS}$ , we have  $\Gamma \vdash D$  and  $\Gamma \vdash_Q (\nu n) E : \tau' \blacktriangleright \text{PERMS}$  by (T-EVAL). The latter judgement can be derived only by (T-RESTR), hence we have  $\Gamma, n : \tau \vdash_Q E : \tau' \blacktriangleright \text{PERMS}$  with  $n \notin \text{dom}(\Gamma)$ . Now we apply Lemma 2 (Weakening) to derive  $\Gamma, n : \tau \vdash D$  from  $\Gamma \vdash D$ , hence we have  $\Gamma, n : \tau \vdash_Q D \setminus E : \tau' \blacktriangleright \text{PERMS}$  by (T-EVAL) and we conclude  $\Gamma \vdash_Q (\nu n) (D \setminus E) : \tau' \blacktriangleright \text{PERMS}$  by (T-RESTR);

*Case (H-FLIP-1):* let  $[\text{PERMS}] (\nu n) E \Rightarrow (\nu n) [\text{PERMS}] E$ . Since  $\Gamma \vdash_Q [\text{PERMS}] (\nu n) E : \tau' \blacktriangleright \text{PERMS}'$ , we have  $\Gamma \vdash_{\text{PERMS}} (\nu n : \tau) E : \tau' \blacktriangleright \text{PERMS}'$  and  $\text{PERMS} \sqsubseteq Q$  by (T-PERMS). The latter judgement can be derived only by (T-RESTR), hence we have  $\Gamma, n : \tau \vdash_{\text{PERMS}} E : \tau' \blacktriangleright \text{PERMS}'$  with  $n \notin \text{dom}(\Gamma)$ . We then get  $\Gamma, n : \tau \vdash_Q [\text{PERMS}] E : \tau' \blacktriangleright \text{PERMS}'$  by (T-PERMS) and we conclude  $\Gamma \vdash_Q (\nu n) [\text{PERMS}] E : \tau' \blacktriangleright \text{PERMS}'$  by (T-RESTR);

*Case (H-FLIP-2):* let  $[\text{PERMS}] (D \setminus E) \Rightarrow D \setminus [\text{PERMS}] E$ . Since  $\Gamma \vdash_Q [\text{PERMS}] (D \setminus E) : \tau \blacktriangleright \text{PERMS}'$ , we have  $\Gamma \vdash_{\text{PERMS}} D \setminus E : \tau \blacktriangleright \text{PERMS}'$  and  $\text{PERMS} \sqsubseteq Q$  by (T-PERMS). The latter judgement can be derived only by (T-EVAL), hence we have  $\Gamma \vdash D$  and  $\Gamma \vdash_{\text{PERMS}} E : \tau \blacktriangleright \text{PERMS}'$ . We then get  $\Gamma \vdash_Q [\text{PERMS}] E : \tau \blacktriangleright \text{PERMS}'$  by (T-PERMS) and we conclude  $\Gamma \vdash_Q D \setminus [\text{PERMS}] E : \tau \blacktriangleright \text{PERMS}'$  by (T-EVAL);

*Case (H-COMM):* let  $(D_1 \wedge D_2) \setminus E \Rightarrow (D_2 \wedge D_1) \setminus E$ . Since  $\Gamma \vdash_Q (D_1 \wedge D_2) \setminus E : \tau \blacktriangleright \text{PERMS}$ , we have  $\Gamma \vdash D_1 \wedge D_2$  and  $\Gamma \vdash_Q E : \tau \blacktriangleright \text{PERMS}$  by (T-EVAL). The former judgement can be derived only by (T-CONJ), hence we have  $\Gamma \vdash D_1$

and  $\Gamma \vdash D_2$ . We then get  $\Gamma \vdash D_2 \wedge D_1$  by (T-CONJ) and we conclude  $\Gamma \vdash_Q (D_2 \wedge D_1) \setminus E : \tau \blacktriangleright \text{PERMS}$  by (T-EVAL). The other direction is analogous.

*Case (H-ASSOC):* let  $(D_1 \wedge D_2) \wedge D_3 \setminus E \Rightarrow D_1 \wedge (D_2 \wedge D_3) \setminus E$ . Since  $\Gamma \vdash_Q (D_1 \wedge D_2) \wedge D_3 \setminus E : \tau \blacktriangleright \text{PERMS}$ , we have  $\Gamma \vdash (D_1 \wedge D_2) \wedge D_3$  and  $\Gamma \vdash_Q E : \tau \blacktriangleright \text{PERMS}$  by (T-EVAL). The former judgement can be derived only by (T-CONJ), hence we have  $\Gamma \vdash D_1 \wedge D_2$  and  $\Gamma \vdash D_3$ . Again the former judgement can be derived only by (T-CONJ), hence we have  $\Gamma \vdash D_1$  and  $\Gamma \vdash D_2$ . We then get  $\Gamma \vdash D_2 \wedge D_3$  by (T-CONJ) and  $\Gamma \vdash D_1 \wedge (D_2 \wedge D_3)$  again by (T-CONJ), so we conclude  $\Gamma \vdash_Q D_1 \wedge (D_2 \wedge D_3) \setminus E : \tau \blacktriangleright \text{PERMS}$  by (T-EVAL). The other direction is analogous.

*Case (H-MOVE):* assume  $D \setminus (\text{let } x = E \text{ in } E') \Rightarrow \text{let } x = (D \setminus E) \text{ in } E'$ . Since  $\Gamma \vdash_R D \setminus (\text{let } x = E \text{ in } E') : \tau \blacktriangleright \text{PERMS}$ , we have  $\Gamma \vdash D$  and  $\Gamma \vdash_R \text{let } x = E \text{ in } E' : \tau \blacktriangleright \text{PERMS}$  by (T-EVAL). The latter judgement can be derived only by (T-LET), hence we have  $\Gamma \vdash_R E : \tau' \blacktriangleright P$  and  $\Gamma, x : \tau' \vdash_R E' : \tau \blacktriangleright Q$  with  $P \sqcup Q = \text{PERMS}$  and  $x \notin \text{dom}(\Gamma)$ . We then get  $\Gamma \vdash_R D \setminus E : \tau' \blacktriangleright P$  by (T-EVAL) and we conclude  $\Gamma \vdash_R \text{let } x = (D \setminus E) \text{ in } E' : \tau \blacktriangleright \text{PERMS}$  by (T-LET).

Assume now  $\text{let } x = (D \setminus E) \text{ in } E' \Rightarrow D \setminus (\text{let } x = E \text{ in } E')$ . Since  $\Gamma \vdash_R \text{let } x = (D \setminus E) \text{ in } E' : \tau \blacktriangleright \text{PERMS}$ , we have  $\Gamma \vdash_R D \setminus E : \tau' \blacktriangleright P$  and  $\Gamma, x : \tau' \vdash_R E' : \tau \blacktriangleright Q$  with  $P \sqcup Q = \text{PERMS}$  and  $x \notin \text{dom}(\Gamma)$  by (T-LET). The former judgement can be derived only by (T-EVAL), hence we have  $\Gamma \vdash D$  and  $\Gamma \vdash_R E : \tau' \blacktriangleright P$ . We then get  $\Gamma \vdash_R \text{let } x = E \text{ in } E' : \tau \blacktriangleright \text{PERMS}$  by (T-LET) and we conclude  $\Gamma \vdash_R D \setminus (\text{let } x = E \text{ in } E') : \tau \blacktriangleright \text{PERMS}$  by (T-EVAL).

*Case (H-CONJ):* let  $D_1 \setminus (D_2 \setminus E) \Rightarrow (D_1 \wedge D_2) \setminus E$ . Since  $\Gamma \vdash_Q D_1 \setminus (D_2 \setminus E) : \tau \blacktriangleright \text{PERMS}$ , we have  $\Gamma \vdash D_1$  and  $\Gamma \vdash_Q D_2 \setminus E : \tau \blacktriangleright \text{PERMS}$  by (T-EVAL). The latter judgement can be derived only by (T-EVAL), hence we have  $\Gamma \vdash D_2$  and  $\Gamma \vdash_Q E : \tau \blacktriangleright \text{PERMS}$ . We then get  $\Gamma \vdash D_1 \wedge D_2$  by (T-CONJ) and we conclude  $\Gamma \vdash_Q (D_1 \wedge D_2) \setminus E : \tau \blacktriangleright \text{PERMS}$  by (T-EVAL).

Assume now  $(D_1 \wedge D_2) \setminus E \Rightarrow D_1 \setminus (D_2 \setminus E)$ . Since  $\Gamma \vdash_Q (D_1 \wedge D_2) \setminus E : \tau \blacktriangleright \text{PERMS}$ , we have  $\Gamma \vdash D_1 \wedge D_2$  and  $\Gamma \vdash_Q E : \tau \blacktriangleright \text{PERMS}$  by (T-EVAL). The former judgement can be derived only by (T-CONJ), hence we have  $\Gamma \vdash D_1$  and  $\Gamma \vdash D_2$ . We then get  $\Gamma \vdash_Q D_2 \setminus E : \tau \blacktriangleright \text{PERMS}$  by (T-EVAL) and we conclude  $\Gamma \vdash_Q D_1 \setminus (D_2 \setminus E) : \tau \blacktriangleright \text{PERMS}$  again by (T-EVAL).

*Case (H-DISTR):* assume  $[\text{PERMS}] \text{let } x = E_1 \text{ in } E_2 \Rightarrow \text{let } x = [\text{PERMS}] E_1 \text{ in } [\text{PERMS}] E_2$ .

Since  $\Gamma \vdash_R [\text{PERMS}] \text{let } x = E_1 \text{ in } E_2 : \tau_2 \blacktriangleright \text{PERMS}'$ , we have  $\Gamma \vdash_{\text{PERMS}} \text{let } x = E_1 \text{ in } E_2 : \tau_2 \blacktriangleright \text{PERMS}'$  and  $\text{PERMS} \sqsubseteq R$  by (T-PERMS). The latter judgement can be derived only by (T-LET), hence we have  $\Gamma \vdash_{\text{PERMS}} E_1 : \tau_1 \blacktriangleright P$  and  $\Gamma, x : \tau_1 \vdash_{\text{PERMS}} E_2 : \tau_2 \blacktriangleright Q$  with  $P \sqcup Q = \text{PERMS}'$ . We then have  $\Gamma \vdash_R [\text{PERMS}] E_1 : \tau_1 \blacktriangleright P$  and  $\Gamma, x : \tau_1 \vdash_R [\text{PERMS}] E_2 : \tau_2 \blacktriangleright Q$  by (T-PERMS), hence we conclude  $\Gamma \vdash_R \text{let } x = [\text{PERMS}] E_1 \text{ in } [\text{PERMS}] E_2 : \tau_2 \blacktriangleright \text{PERMS}'$  by (T-LET).

## B.2 Proof of subject reduction

**Definition 6 (Permission Lowering).** Let  $\Gamma \vdash_Q^\xi E : \tau \blacktriangleright P$ . We define the permission lowering of the expression  $E$  with respect to the type derivation  $\xi$ , written  $\xi \cdot E$ , by induction on the structure of  $E$ :

- $E = [\text{PERMS}] E' \Rightarrow \xi \cdot E \triangleq [P \sqcap \text{PERMS}] E'$ ;

- $E = (\nu n) E' \Rightarrow \xi \cdot E \triangleq (\nu n) (\xi' \cdot E')$ ;
- $E = D \setminus E' \Rightarrow \xi \cdot E \triangleq D \setminus (\xi' \cdot E')$ ;
- $E = (\text{let } x = E_1 \text{ in } E_2) \Rightarrow \xi \cdot E \triangleq \text{let } x = (\xi_1 \cdot E_1) \text{ in } (\xi_2 \cdot E_2)$ ,

where  $\xi'$ ,  $\xi_1$  and  $\xi_2$  denote the sub-derivations of  $\xi$  assigning types to the sub-expressions  $E'$ ,  $E_1$  and  $E_2$ , respectively. In all the other cases, we let  $\xi \cdot E \triangleq E$ .

**Lemma 5 (Deterministic Lowering).** *If  $\Gamma \vdash_{\mathbf{Q}}^{\xi} E : \tau \blacktriangleright \mathbf{P}$  and  $\Gamma \vdash_{\mathbf{Q}'}^{\xi'} E : \tau' \blacktriangleright \mathbf{P}'$ , then  $\xi \cdot E = \xi' \cdot E$ .*

*Proof.* We first prove the following statement:

$$\text{If } \Gamma \vdash_{\mathbf{Q}}^{\xi} E : \tau \blacktriangleright \mathbf{P} \text{ and } \Gamma \vdash_{\mathbf{Q}'}^{\xi'} E : \tau' \blacktriangleright \mathbf{P}', \text{ then } \mathbf{P} = \mathbf{P}'.$$

The proof is by induction on the structure of  $E$ . The only interesting case is when  $E$  is an invocation, i.e., when  $E = \bar{u}\langle v \triangleright \text{RECV} \rangle$ . Assume then that  $\Gamma \vdash_{\mathbf{Q}}^{\xi} \bar{u}\langle v \triangleright \text{RECV} \rangle : \tau \blacktriangleright \mathbf{P}$  and  $\Gamma \vdash_{\mathbf{Q}'}^{\xi'} \bar{u}\langle v \triangleright \text{RECV} \rangle : \tau' \blacktriangleright \mathbf{P}'$ , we perform a case analysis on the last typing rule applied in  $\xi$  and  $\xi'$ :

- Case (T-CALL)/(T-CALL):* let  $\Gamma \vdash u : \text{Fun}(\text{CALL}, \tau_1 \rightarrow \tau_2)^{\text{SECR}}$  among the premises of  $\xi$  and  $\Gamma \vdash u : \text{Fun}(\text{CALL}', \tau'_1 \rightarrow \tau'_2)^{\text{SECR}'}$  among the premises of  $\xi'$ , then we have  $\mathbf{P} = \text{CALL} \sqcup \text{SECR}$  and  $\mathbf{P}' = \text{CALL}' \sqcup \text{SECR}'$ . The conclusion follows by Proposition 2 (Uniqueness of Function Types);
- Case (T-CALL-UN)/(T-CALL-UN):* the case is immediate, since  $\mathbf{P} = \mathbf{P}' = \perp$ ;
- Case (T-FAIL)/(T-FAIL):* the case is immediate, since  $\mathbf{P} = \mathbf{P}' = \mathbf{Q}$ ;
- Case (T-CALL)/(T-FAIL):* let  $\Gamma \vdash u : \text{Fun}(\text{CALL}, \tau_1 \rightarrow \tau_2)^{\text{SECR}}$  among the premises of  $\xi$  and let  $\Gamma \vdash u : \text{Fun}(\text{CALL}', \tau'_1 \rightarrow \tau'_2)^{\text{SECR}'}$  among the premises of  $\xi'$ . Since we have  $\text{CALL} \sqcup \text{SECR} \sqsubseteq \mathbf{Q}$  in  $\xi$ , we know that  $\text{CALL} \sqsubseteq \mathbf{Q}$  by transitivity; however, we also have  $\text{CALL}' \not\sqsubseteq \mathbf{Q}$  in  $\xi'$ , so we get a contradiction by Proposition 2 (Uniqueness of Function Types);
- Case (T-CALL)/(T-CALL-UN):* let  $\Gamma \vdash u : \text{Fun}(\text{CALL}, \tau_1 \rightarrow \tau_2)^{\text{SECR}}$  among the premises of  $\xi$ , we have  $\mathbf{P} = \text{CALL} \sqcup \text{SECR} \sqsubseteq \mathbf{Q}$ . But note that  $\mathbf{Q} = \perp$ , otherwise we could not apply rule (T-CALL-UN), hence  $\mathbf{P} = \perp$  by anti-symmetry. Since  $\mathbf{P}' = \perp$ , we conclude;
- Case (T-FAIL)/(T-CALL-UN):* in this case we have  $\mathbf{P} = \mathbf{Q}$ . But note that  $\mathbf{Q} = \perp$ , otherwise we could not apply rule (T-CALL-UN). Since  $\mathbf{P}' = \perp$ , we conclude.

The symmetric cases are analogous.

The main statement is proved again by induction on the structure of  $E$ . The only interesting case is when  $E$  is a permission assignment, i.e., when  $E = [\text{PERMS}] E'$ . Assume then  $\Gamma \vdash_{\mathbf{Q}}^{\xi} [\text{PERMS}] E' : \tau \blacktriangleright \mathbf{P}$  and  $\Gamma \vdash_{\mathbf{Q}'}^{\xi'} [\text{PERMS}] E' : \tau' \blacktriangleright \mathbf{P}'$ , in this case both  $\xi$  and  $\xi'$  are concluded by an application of rule (T-PERMS), hence we know that  $\Gamma \vdash_{\text{PERMS}} E' : \tau \blacktriangleright \mathbf{P}$  among the premises of  $\xi$  and  $\Gamma \vdash_{\text{PERMS}} E' : \tau' \blacktriangleright \mathbf{P}'$  among the premises of  $\xi'$ . By the previous result we have  $\mathbf{P} = \mathbf{P}'$ , thus  $\xi \cdot E = [\mathbf{P} \sqcap \text{PERMS}] E' = [\mathbf{P}' \sqcap \text{PERMS}] E' = \xi' \cdot E$ .

**Notation 2.** *By Lemma 5 (Deterministic Lowering), for any well-typed expression  $E$  we can write  $\Gamma \cdot E$  to stand for  $\xi \cdot E$  for an arbitrarily chosen type derivation  $\xi$  such that  $\Gamma \vdash_{\mathbf{Q}}^{\xi} E : \tau \blacktriangleright \mathbf{P}$  for some  $\mathbf{P}$ ,  $\mathbf{Q}$  and  $\tau$ .*

**Definition 7 (Expression Ordering).** We overload the symbol  $\sqsubseteq$  to denote the smallest pre-order on expressions closed under the following inference rules:

$$\frac{\text{PERMS}_1 \sqsubseteq \text{PERMS}_2}{[\text{PERMS}_1] E \sqsubseteq [\text{PERMS}_2] E} \quad \frac{E \sqsubseteq E'}{(\nu n) E \sqsubseteq (\nu n) E'} \quad \frac{E \sqsubseteq E'}{D \setminus E \sqsubseteq D \setminus E'}$$

$$\frac{E_1 \sqsubseteq E'_1 \quad E_2 \sqsubseteq E'_2}{\text{let } x = E_1 \text{ in } E_2 \sqsubseteq \text{let } x = E'_1 \text{ in } E'_2}$$

**Proposition 4 (Soundness of Lowering).** For any  $E$  such that  $\Gamma \vdash E : \tau \blacktriangleright \mathbf{P}$ , we have  $E \sqsubseteq \Gamma \cdot E$ .

*Proof.* By induction on the structure of  $E$ .

**Lemma 6 (Lowering Respects Heating).** Let  $\Gamma \vdash E : \tau \blacktriangleright \mathbf{P}$ . If  $E \Rightarrow E'$ , then  $\Gamma \cdot E'$  is defined and  $\Gamma \cdot E \Rightarrow E''$  for some  $E'' \sqsubseteq \Gamma \cdot E'$ .

*Proof.* First of all, we note that  $\Gamma \vdash E' : \tau \blacktriangleright \mathbf{P}$  by Lemma 4 (Heating Preserves Typing), hence  $\Gamma \cdot E'$  is defined. We then proceed by induction on the derivation of  $E \Rightarrow E'$ :

*Case (H-EVAL):* let  $D \setminus E \Rightarrow D \setminus E'$  by the premise  $E \Rightarrow E'$ . Since  $\Gamma \vdash D \setminus E : \tau \blacktriangleright \mathbf{P}$ , we have  $\Gamma \vdash D$  and  $\Gamma \vdash E : \tau \blacktriangleright \mathbf{P}$  by (T-EVAL). By inductive hypothesis  $\Gamma \cdot E \Rightarrow E''$  for some  $E'' \sqsubseteq \Gamma \cdot E'$ , hence we have:

$$\begin{aligned} \Gamma \cdot (D \setminus E) &\triangleq D \setminus (\Gamma \cdot E) \\ &\Rightarrow D \setminus E'' \\ &\sqsubseteq D \setminus (\Gamma \cdot E') \\ &\triangleq \Gamma \cdot (D \setminus E'). \end{aligned}$$

*Case (H-LET):* assume  $\text{let } x = E \text{ in } E'' \Rightarrow \text{let } x = E' \text{ in } E''$  by the premise  $E \Rightarrow E'$ . Since  $\Gamma \vdash \text{let } x = E \text{ in } E'' : \tau \blacktriangleright \text{PERMS}$ , we have  $\Gamma \vdash E : \tau' \blacktriangleright \mathbf{P}$  and  $\Gamma, x : \tau' \vdash E'' : \tau \blacktriangleright \mathbf{Q}$  with  $\mathbf{P} \sqcup \mathbf{Q} = \text{PERMS}$  by (T-LET). By inductive hypothesis  $\Gamma \cdot E \Rightarrow \hat{E}$  for some  $\hat{E} \sqsubseteq \Gamma \cdot E'$ , hence we have:

$$\begin{aligned} \Gamma \cdot (\text{let } x = E \text{ in } E'') &\triangleq \text{let } x = (\Gamma \cdot E) \text{ in } (\Gamma, x : \tau) \cdot E'' \\ &\Rightarrow \text{let } x = \hat{E} \text{ in } (\Gamma, x : \tau) \cdot E'' \\ &\sqsubseteq \text{let } x = (\Gamma \cdot E') \text{ in } (\Gamma, x : \tau) \cdot E'' \\ &\triangleq \Gamma \cdot (\text{let } x = E' \text{ in } E''). \end{aligned}$$

*Case (H-RESTR):* let  $(\nu n) E \Rightarrow (\nu n) E'$  by the premise  $E \Rightarrow E'$ . Since  $\Gamma \vdash (\nu n) E : \tau' \blacktriangleright \text{PERMS}$ , we have  $\Gamma, n : \tau \vdash E : \tau' \blacktriangleright \text{PERMS}$  by (T-RESTR). By inductive hypothesis  $(\Gamma, n : \tau) \cdot E \Rightarrow E''$  for some  $E'' \sqsubseteq (\Gamma, n : \tau) \cdot E'$ , hence we have:

$$\begin{aligned} \Gamma \cdot ((\nu n) E) &\triangleq (\nu n) ((\Gamma, n : \tau) \cdot E) \\ &\Rightarrow (\nu n) E'' \\ &\sqsubseteq (\nu n) ((\Gamma, n : \tau) \cdot E') \\ &\triangleq \Gamma \cdot ((\nu n) E'). \end{aligned}$$



*Case (H-EXTR-1):* assume  $\text{let } x = (\nu n) E_1 \text{ in } E_2 \Rightarrow (\nu n) (\text{let } x = E_1 \text{ in } E_2)$  with  $n \notin \text{fn}(E_2)$ . Since  $\Gamma \vdash \text{let } x = (\nu n) E_1 \text{ in } E_2 : \tau_2 \blacktriangleright \mathbf{R}$ , we have  $\Gamma \vdash (\nu n) E_1 : \tau_1 \blacktriangleright \mathbf{P}$  and  $\Gamma, x : \tau_1 \vdash E_2 : \tau_2 \blacktriangleright \mathbf{Q}$  with  $\mathbf{P} \sqcup \mathbf{Q} = \mathbf{R}$  and  $x \notin \text{dom}(\Gamma)$  by (T-LET). The former judgement can be derived only by (T-RESTR), hence we have  $\Gamma, n : \tau \vdash E_1 : \tau_1 \blacktriangleright \mathbf{P}$  with  $n \notin \text{dom}(\Gamma)$ . Now we apply Lemma 2 (Weakening) to derive  $\Gamma, n : \tau, x : \tau_1 \vdash E_2 : \tau_2 \blacktriangleright \mathbf{Q}$  from  $\Gamma, x : \tau_1 \vdash E_2 : \tau_2 \blacktriangleright \mathbf{Q}$ . Notice that the lemma also implies that  $(\Gamma, x : \tau_1) \cdot E_2 = (\Gamma, n : \tau, x : \tau_1) \cdot E_2$ , hence we have:

$$\begin{aligned} \Gamma \cdot (\text{let } x = (\nu n) E_1 \text{ in } E_2) &\triangleq \text{let } x = (\Gamma \cdot (\nu n) E_1) \text{ in } (\Gamma, x : \tau_1) \cdot E_2 \\ &\triangleq \text{let } x = (\nu n) ((\Gamma, n : \tau) \cdot E_1) \text{ in } (\Gamma, x : \tau_1) \cdot E_2 \\ &\Rightarrow (\nu n) (\text{let } x = ((\Gamma, n : \tau) \cdot E_1) \text{ in } (\Gamma, x : \tau_1) \cdot E_2) \\ &= (\nu n) (\text{let } x = ((\Gamma, n : \tau) \cdot E_1) \text{ in } (\Gamma, n : \tau, x : \tau_1) \cdot E_2) \\ &\triangleq \Gamma \cdot ((\nu n) (\text{let } x = E_1 \text{ in } E_2)). \end{aligned}$$

*Case (H-EXTR-2):* let  $D \setminus (\nu n) E \Rightarrow (\nu n) (D \setminus E)$  with  $n \notin \text{fn}(D)$ . Given that  $\Gamma \vdash D \setminus (\nu n) E : \tau' \blacktriangleright \text{PERMS}$ , we have  $\Gamma \vdash D$  and  $\Gamma \vdash (\nu n) E : \tau' \blacktriangleright \text{PERMS}$  by (T-EVAL). The latter judgement can be derived only by (T-RESTR), hence we have  $\Gamma, n : \tau \vdash E : \tau' \blacktriangleright \text{PERMS}$  with  $n \notin \text{dom}(\Gamma)$ . Hence, we have:

$$\begin{aligned} \Gamma \cdot (D \setminus (\nu n) E) &\triangleq D \setminus (\Gamma \cdot (\nu n) E) \\ &\triangleq D \setminus (\nu n) ((\Gamma, n : \tau) \cdot E) \\ &\Rightarrow (\nu n) (D \setminus ((\Gamma, n : \tau) \cdot E)) \\ &\triangleq \Gamma \cdot ((\nu n) (D \setminus E)) \end{aligned}$$

*Case (H-FLIP-1):* let  $[\text{PERMS}] (\nu n) E \Rightarrow (\nu n) [\text{PERMS}] E$ . Since  $\Gamma \vdash_{\mathbf{R}} [\text{PERMS}] (\nu n) E : \tau' \blacktriangleright \mathbf{Q}$ , we have  $\Gamma \vdash_{\text{PERMS}} (\nu n : \tau) E : \tau' \blacktriangleright \mathbf{Q}$  and  $\text{PERMS} \sqsubseteq \mathbf{R}$  by (T-PERMS). The latter judgement can be derived only by (T-RESTR), hence we have  $\Gamma, n : \tau \vdash_{\text{PERMS}} E : \tau' \blacktriangleright \mathbf{Q}$  with  $n \notin \text{dom}(\Gamma)$ . We then get  $\Gamma, n : \tau \vdash_{\mathbf{R}} [\text{PERMS}] E : \tau' \blacktriangleright \mathbf{Q}$  by (T-PERMS). Hence, we have:

$$\begin{aligned} \Gamma \cdot ([\text{PERMS}] (\nu n) E) &\triangleq [\text{PERMS} \sqcap \mathbf{Q}] (\nu n) E \\ &\Rightarrow (\nu n) [\text{PERMS} \sqcap \mathbf{Q}] E \\ &\triangleq (\nu n) ((\Gamma, n : \tau) \cdot [\text{PERMS}] E) \\ &\triangleq \Gamma \cdot ((\nu n) [\text{PERMS}] E). \end{aligned}$$

*Case (H-FLIP-2):* let  $[\text{PERMS}] (D \setminus E) \Rightarrow D \setminus [\text{PERMS}] E$ . Since  $\Gamma \vdash_{\mathbf{R}} [\text{PERMS}] (D \setminus E) : \tau \blacktriangleright \mathbf{Q}$ , we have  $\Gamma \vdash_{\text{PERMS}} D \setminus E : \tau \blacktriangleright \mathbf{Q}$  and  $\text{PERMS} \sqsubseteq \mathbf{R}$  by (T-PERMS). The latter judgement can be derived only by (T-EVAL), hence we have  $\Gamma \vdash D$  and  $\Gamma \vdash_{\text{PERMS}} E : \tau \blacktriangleright \mathbf{Q}$ . We then get  $\Gamma \vdash_{\mathbf{R}} [\text{PERMS}] E : \tau \blacktriangleright \mathbf{Q}$  by (T-PERMS). Hence,

we have:

$$\begin{aligned}
\Gamma \cdot ([\text{PERMS}] (D \setminus E)) &\triangleq [\mathbf{Q} \sqcap \text{PERMS}] (D \setminus E) \\
&\Rightarrow D \setminus [\mathbf{Q} \sqcap \text{PERMS}] E \\
&\triangleq D \setminus (\Gamma \cdot ([\text{PERMS}] E)) \\
&\triangleq \Gamma \cdot (D \setminus [\text{PERMS}] E).
\end{aligned}$$

*Case (H-COMM):* let  $(D_1 \wedge D_2) \setminus E \Rightarrow (D_2 \wedge D_1) \setminus E$ . Since  $\Gamma \vdash (D_1 \wedge D_2) \setminus E : \tau \blacktriangleright \text{PERMS}$ , we have  $\Gamma \vdash D_1 \wedge D_2$  and  $\Gamma \vdash E : \tau \blacktriangleright \text{PERMS}$  by (T-EVAL). Hence, we have:

$$\begin{aligned}
\Gamma \cdot ((D_1 \wedge D_2) \setminus E) &\triangleq (D_1 \wedge D_2) \setminus (\Gamma \cdot E) \\
&\Rightarrow (D_2 \wedge D_1) \setminus (\Gamma \cdot E) \\
&\triangleq \Gamma \cdot ((D_2 \wedge D_1) \setminus E).
\end{aligned}$$

The other direction is analogous.

*Case (H-ASSOC):* let  $(D_1 \wedge D_2) \wedge D_3 \setminus E \Rightarrow D_1 \wedge (D_2 \wedge D_3) \setminus E$ . Since  $\Gamma \vdash (D_1 \wedge D_2) \wedge D_3 \setminus E : \tau \blacktriangleright \text{PERMS}$ , we have  $\Gamma \vdash (D_1 \wedge D_2) \wedge D_3$  and  $\Gamma \vdash E : \tau \blacktriangleright \text{PERMS}$  by (T-EVAL). Hence, we have:

$$\begin{aligned}
\Gamma \cdot ((D_1 \wedge D_2) \wedge D_3 \setminus E) &\triangleq (D_1 \wedge D_2) \wedge D_3 \setminus (\Gamma \cdot E) \\
&\Rightarrow D_1 \wedge (D_2 \wedge D_3) \setminus (\Gamma \cdot E) \\
&\triangleq \Gamma \cdot (D_1 \wedge (D_2 \wedge D_3) \setminus E).
\end{aligned}$$

The other direction is analogous.

*Case (H-CONJ):* let  $D_1 \setminus (D_2 \setminus E) \Rightarrow (D_1 \wedge D_2) \setminus E$ . Since  $\Gamma \vdash D_1 \setminus (D_2 \setminus E) : \tau \blacktriangleright \text{PERMS}$ , we have  $\Gamma \vdash D_1$  and  $\Gamma \vdash D_2 \setminus E : \tau \blacktriangleright \text{PERMS}$  by (T-EVAL). The latter judgement can be derived only by (T-EVAL), hence we have  $\Gamma \vdash D_2$  and  $\Gamma \vdash E : \tau \blacktriangleright \text{PERMS}$ . Hence, we have:

$$\begin{aligned}
\Gamma \cdot (D_1 \setminus (D_2 \setminus E)) &\triangleq D_1 \setminus D_2 \setminus (\Gamma \cdot E) \\
&\Rightarrow D_1 \wedge D_2 \setminus (\Gamma \cdot E) \\
&\triangleq \Gamma \cdot ((D_1 \wedge D_2) \setminus E).
\end{aligned}$$

The other direction is similar.

*Case (H-MOVE):* assume  $D \setminus (\text{let } x = E \text{ in } E') \Rightarrow \text{let } x = (D \setminus E) \text{ in } E'$ . Since  $\Gamma \vdash D \setminus (\text{let } x = E \text{ in } E') : \tau \blacktriangleright \mathbf{R}$ , we have  $\Gamma \vdash D$  and  $\Gamma \vdash \text{let } x = E \text{ in } E' : \tau \blacktriangleright \mathbf{R}$  by (T-EVAL). The latter judgement can be derived only by (T-LET), hence we have  $\Gamma \vdash E : \tau' \blacktriangleright \mathbf{P}$  and  $\Gamma, x : \tau' \vdash E' : \tau \blacktriangleright \mathbf{Q}$  with  $\mathbf{P} \sqcup \mathbf{Q} = \mathbf{R}$  and  $x \notin \text{dom}(\Gamma)$ . Hence, we have:

$$\begin{aligned}
\Gamma \cdot (D \setminus (\text{let } x = E \text{ in } E')) &\triangleq D \setminus (\Gamma \cdot (\text{let } x = E \text{ in } E')) \\
&\triangleq D \setminus \text{let } x = (\Gamma \cdot E) \text{ in } (\Gamma, x : \tau') \cdot E' \\
&\Rightarrow \text{let } x = D \setminus (\Gamma \cdot E) \text{ in } (\Gamma, x : \tau') \cdot E' \\
&\triangleq \text{let } x = \Gamma \cdot (D \setminus E) \text{ in } (\Gamma, x : \tau') \cdot E' \\
&\triangleq \Gamma \cdot (\text{let } x = (D \setminus E) \text{ in } E').
\end{aligned}$$

The other direction is similar.

*Case (H-DISTR):* assume  $[\text{PERMS}] \text{let } x = E_1 \text{ in } E_2 \Rightarrow \text{let } x = [\text{PERMS}] E_1 \text{ in } [\text{PERMS}] E_2$  with  $\Gamma \vdash [\text{PERMS}] \text{let } x = E_1 \text{ in } E_2 : \tau_2 \blacktriangleright \text{PERMS}'$ . The judgement must have been derived by (T-PERMS), hence we know  $\Gamma \vdash_{\text{PERMS}} \text{let } x = E_1 \text{ in } E_2 : \tau_2 \blacktriangleright \text{PERMS}'$ . This can be derived only by (T-LET), so we have  $\Gamma \vdash_{\text{PERMS}} E_1 : \tau_1 \blacktriangleright P$  and  $\Gamma, x : \tau_1 \vdash_{\text{PERMS}} E_2 : \tau_2 \blacktriangleright Q$  with  $P \sqcup Q = \text{PERMS}'$ . We can then apply (T-PERMS) to derive  $\Gamma \vdash_{\text{PERMS}} [\text{PERMS}] E_1 : \tau_1 \blacktriangleright P$  and  $\Gamma, x : \tau_1 \vdash_{\text{PERMS}} [\text{PERMS}] E_2 : \tau_2 \blacktriangleright Q$ . Now we notice that we have:

$$\begin{aligned} \Gamma \cdot ([\text{PERMS}] \text{let } x = E_1 \text{ in } E_2) &\triangleq [(\text{P} \sqcup \text{Q}) \sqcap \text{PERMS}] \text{let } x = E_1 \text{ in } E_2 \\ &\Rightarrow \text{let } x = [(\text{P} \sqcup \text{Q}) \sqcap \text{PERMS}] E_1 \text{ in } [(\text{P} \sqcup \text{Q}) \sqcap \text{PERMS}] E_2 \\ &\sqsupseteq \text{let } x = [\text{P} \sqcap \text{PERMS}] E_1 \text{ in } [\text{Q} \sqcap \text{PERMS}] E_2 \\ &\triangleq \text{let } x = (\Gamma \cdot [\text{PERMS}] E_1) \text{ in } ((\Gamma, x : \tau_1) \cdot [\text{PERMS}] E_2) \\ &\triangleq \Gamma \cdot (\text{let } x = [\text{PERMS}] E_1 \text{ in } [\text{PERMS}] E_2). \end{aligned}$$

**Lemma 7 (Monotonicity of Heating).** *If  $E_1 \Rightarrow E_2$  and  $E_1 \sqsubseteq E'_1$ , then  $E'_1 \Rightarrow E'_2$  for some  $E'_2 \sqsupseteq E_2$ .*

*Proof.* By a straightforward induction on the derivation of  $E_1 \Rightarrow E_2$ .

**Lemma 8 (Monotonicity of Reduction).** *If  $E_1 \xrightarrow{\alpha}_i E_2$  and  $E_1 \sqsubseteq E'_1$ , then  $E'_1 \xrightarrow{\alpha}_i E'_2$  for some  $E'_2 \sqsupseteq E_2$ .*

*Proof.* By a straightforward induction on the derivation of  $E_1 \xrightarrow{\alpha}_i E_2$ .

**Proposition 5 (Monotonicity of Typing).** *If  $\Gamma \vdash_{\text{Q}} [\text{PERMS}] E : \tau \blacktriangleright P$  and  $\text{Q} \sqsubseteq \text{R}$ , then  $\Gamma \vdash_{\text{R}} [\text{PERMS}] E : \tau \blacktriangleright P$ .*

*Proof.* Since  $\Gamma \vdash_{\text{Q}} [\text{PERMS}] E : \tau \blacktriangleright P$  can be derived only by (T-PERMS), we know that  $\Gamma \vdash_{\text{PERMS}} E : \tau \blacktriangleright P$  and  $\text{PERMS} \sqsubseteq \text{Q}$ . Hence,  $\text{PERMS} \sqsubseteq \text{R}$  by transitivity and we get  $\Gamma \vdash_{\text{R}} [\text{PERMS}] E : \tau \blacktriangleright P$  again by (T-PERMS).

**Theorem 3 (Simulation-Aware Subject Reduction).** *If  $\Gamma \vdash_{\top} E : \tau \blacktriangleright \text{PERMS}$  and  $E \xrightarrow{\alpha} E'$ , then  $\Gamma \vdash_{\top} E' : \tau \blacktriangleright \text{PERMS}'$  for some  $\text{PERMS}' \sqsubseteq \text{PERMS}$ . Moreover, there exists  $E''$  such that  $\Gamma \cdot E \xrightarrow{\alpha}_i E''$  and  $E'' \sqsupseteq \Gamma \cdot E'$ .*

*Proof.* By induction on the derivation of  $E \xrightarrow{\alpha} E'$ :

*Case (R-CALL):* assume  $n^\ell(x \triangleleft \text{CALL}).[\text{PERMS}'] E \setminus [\text{PERMS}] \bar{n} \langle m \triangleright \text{RECV} \rangle \xrightarrow{\ell} [\text{PERMS}'] E \{m/x\}$  with:

- (1)  $\text{CALL} \sqsubseteq \text{PERMS}$
- (2)  $\text{RECV} \sqsubseteq \text{PERMS}'$ .

By hypothesis we know that:

$$\Gamma \vdash_{\top} n(x \triangleleft \text{CALL}).[\text{PERMS}'] E \setminus [\text{PERMS}] \bar{n} \langle m \triangleright \text{RECV} \rangle : \tau \blacktriangleright P,$$

which must follow by an instance of (T-EVAL). Hence, we know that  $\Gamma \vdash n^\ell(x \triangleleft \text{CALL}).[\text{PERMS}'] E$  and  $\Gamma \vdash_{\top} [\text{PERMS}] \bar{n} \langle m \triangleright \text{RECV} \rangle : \tau \blacktriangleright P$  must hold. We perform a case analysis on how these latter two judgements are derived.

If  $\Gamma \vdash n^\ell(x \triangleleft \text{CALL}).[\text{PERMS}'] E$  was derived by (T-DEF), we know that:

$$(3) \Gamma \vdash n : \text{Fun}(\text{CALL}, \tau_n \rightarrow \tau'_n)^{\text{SECR}}$$

$$(4) \Gamma, x : \tau_n \vdash_{\top} [\text{PERMS}'] E : \tau'_n \blacktriangleright Q' \text{ with } x \notin \text{dom}(\Gamma)$$

$$(5) Q' \sqsubseteq \text{CALL} \sqcup \text{SECR}$$

$$(6) \text{CALL} \sqcup \text{SECR} = \perp \Rightarrow \Gamma, x : \text{Un} \vdash_{\top} [\text{PERMS}'] E : \text{Un} \blacktriangleright \perp \text{ with } x \notin \text{dom}(\Gamma).$$

We distinguish three cases, according to the rule used to derive  $\Gamma \vdash_{\top} [\text{PERMS}] \bar{n}\langle m \triangleright \text{RECV} \rangle : \tau \blacktriangleright P$ . If the latter judgement was derived by (T-CALL) after an application of (T-PERMS), then we know that:

$$(7) \Gamma \vdash n : \text{Fun}(\text{CALL}', \hat{\tau}_n \rightarrow \hat{\tau}'_n)^{\text{SECR}'}$$

$$(8) \Gamma \vdash m : \hat{\tau}_n$$

$$(9) \text{CALL}' \sqcup \text{SECR}' \sqsubseteq \text{PERMS}$$

$$(10) P = \text{CALL}' \sqcup \text{SECR}'.$$

By Proposition 2 (Uniqueness of Function Types), we know that  $\hat{\tau}_n = \tau_n$ ,  $\hat{\tau}'_n = \tau'_n = \tau$ ,  $\text{CALL} = \text{CALL}'$  and  $\text{SECR} = \text{SECR}'$ . By (4) and (8), using Lemma 3 (Substitution), we then get  $\Gamma \vdash_{\top} [\text{PERMS}'] E\{m/x\} : \tau \blacktriangleright Q'$ . Notice also that  $Q' \sqsubseteq P$  by (5) and (10). Now we note that:

$$\begin{aligned} & \Gamma \cdot (n^\ell(x \triangleleft \text{CALL}).[\text{PERMS}'] E \setminus [\text{PERMS}] \bar{n}\langle m \triangleright \text{RECV} \rangle) \\ & \triangleq n^\ell(x \triangleleft \text{CALL}).[\text{PERMS}'] E \setminus [(\text{CALL} \sqcup \text{SECR}) \sqcap \text{PERMS}] \bar{n}\langle m \triangleright \text{RECV} \rangle \\ & = n^\ell(x \triangleleft \text{CALL}).[\text{PERMS}'] E \setminus [\text{CALL} \sqcup \text{SECR}] \bar{n}\langle m \triangleright \text{RECV} \rangle & \text{by (9)} \\ & \xrightarrow{\ell}_i [(\text{CALL} \sqcup \text{SECR}) \sqcap \text{PERMS}'] E\{m/x\} \\ & \sqsupseteq [Q' \sqcap \text{PERMS}'] E\{m/x\} & \text{by } Q' \sqsubseteq P \\ & \triangleq \Gamma \cdot ([\text{PERMS}'] E\{m/x\}), \end{aligned}$$

where the reduction step can be performed, since  $\text{CALL} \sqsubseteq \text{CALL} \sqcup \text{SECR}$ .

Assume then that  $\Gamma \vdash_{\top} [\text{PERMS}] \bar{n}\langle m \triangleright \text{RECV} \rangle : \tau \blacktriangleright P$  was derived by (T-CALL-UN) after an application of (T-PERMS), then we know that:

$$(11) \Gamma \vdash n : \text{Un}$$

$$(12) \Gamma \vdash m : \text{Un}$$

$$(13) \tau = \text{Un}$$

$$(14) P = \perp$$

$$(15) \text{PERMS} = \perp.$$

Since (3) and (11) hold, by Proposition 3 (Soundness of Secrecy Levels) we know that  $\text{SECR} = \perp$ . Since hypothesis (1) states  $\text{CALL} \sqsubseteq \text{PERMS}$  and (15) holds true, we know that  $\text{CALL} = \perp$  by anti-symmetry, so we have  $\text{CALL} \sqcup \text{SECR} = \perp$ . By (6) we can then get  $\Gamma, x : \text{Un} \vdash_{\top} [\text{PERMS}'] E : \text{Un} \blacktriangleright \perp$ , hence, by (12) and Lemma 3 (Substitution), we get  $\Gamma \vdash_{\top} [\text{PERMS}'] E\{m/x\} : \tau \blacktriangleright \perp$ . Now we note that:

$$\begin{aligned} & \Gamma \cdot (n^\ell(x \triangleleft \text{CALL}).[\text{PERMS}'] E \setminus [\text{PERMS}] \bar{n}\langle m \triangleright \text{RECV} \rangle) \\ & = \Gamma \cdot (n^\ell(x \triangleleft \text{CALL}).[\text{PERMS}'] E \setminus [\perp] \bar{n}\langle m \triangleright \text{RECV} \rangle) & \text{by (15)} \\ & \triangleq n^\ell(x \triangleleft \text{CALL}).[\text{PERMS}'] E \setminus [\perp] \bar{n}\langle m \triangleright \text{RECV} \rangle \\ & \xrightarrow{\ell}_i [\perp \sqcap \text{PERMS}'] E\{m/x\} \\ & = [\perp] E\{m/x\} \\ & \triangleq \Gamma \cdot ([\text{PERMS}'] E\{m/x\}), \end{aligned}$$

where the reduction step can be performed, since we showed that  $\text{CALL} = \perp$ .  
 Finally, assume that  $\Gamma \vdash_{\top} [\text{PERMS}] \bar{n} \langle m \triangleright \text{RECV} \rangle : \tau \blacktriangleright \text{P}$  was derived by (T-FAIL)  
 after an application of (T-PERMS), then we know that:

$$(16) \quad \Gamma \vdash n : \text{Fun}(\text{CALL}', \hat{\tau}_n \rightarrow \hat{\tau}'_n)^{\text{SECR}'}$$

$$(17) \quad \text{CALL}' \not\sqsubseteq \text{PERMS}.$$

Since (3) and (16) hold, by Proposition 2 (Uniqueness of Function Types) we know that  $\text{CALL} \not\sqsubseteq \text{PERMS}$ , but this is in contradiction with  $\text{CALL} \sqsubseteq \text{PERMS}$  from hypothesis (1), hence the case is trivial.

Let us now consider the case when  $\Gamma \vdash n^\ell(x \triangleleft \text{CALL}).[\text{PERMS}'] E$  was derived by (T-DEF-UN). In this case we know that:

$$(18) \quad \Gamma \vdash n : \text{Un}$$

$$(19) \quad \Gamma, x : \text{Un} \vdash_{\perp} [\text{PERMS}'] E : \text{Un} \blacktriangleright \perp \text{ with } x \notin \text{dom}(\Gamma).$$

Note that (19) can be derived only after an application of (T-PERMS), which implies  $\text{PERMS}' \sqsubseteq \perp$ . By anti-symmetry, we then get:

$$(20) \quad \text{PERMS}' = \perp.$$

We distinguish three cases, according to the rule used to derive  $\Gamma \vdash_{\top} [\text{PERMS}] \bar{n} \langle m \triangleright \text{RECV} \rangle : \tau \blacktriangleright \text{P}$ . If the latter judgement was derived by (T-CALL-UN) after an application of (T-PERMS), then we know that:

$$(21) \quad \Gamma \vdash m : \text{Un}$$

$$(22) \quad \tau = \text{Un}$$

$$(23) \quad \text{P} = \perp$$

$$(24) \quad \text{PERMS} = \perp.$$

By (19) and (21), using Lemma 3 (Substitution), we get  $\Gamma \vdash_{\perp} [\text{PERMS}'] E\{m/x\} : \tau \blacktriangleright \perp$ , hence we get  $\Gamma \vdash_{\top} [\text{PERMS}'] E\{m/x\} : \tau \blacktriangleright \perp$  by Proposition 5 (Monotonicity of Typing). Now we note that:

$$\begin{aligned} & \Gamma \cdot (n^\ell(x \triangleleft \text{CALL}).[\text{PERMS}'] E \setminus [\text{PERMS}] \bar{n} \langle m \triangleright \text{RECV} \rangle) \\ &= \Gamma \cdot (n^\ell(x \triangleleft \text{CALL}).[\text{PERMS}'] E \setminus [\perp] \bar{n} \langle m \triangleright \text{RECV} \rangle) \quad \text{by (24)} \\ &\triangleq n^\ell(x \triangleleft \text{CALL}).[\text{PERMS}'] E \setminus [\perp] \bar{n} \langle m \triangleright \text{RECV} \rangle \\ &\xrightarrow{\ell}_i [\perp \sqcap \text{PERMS}'] E\{m/x\} \\ &= [\perp] E\{m/x\} \\ &\triangleq \Gamma \cdot ([\text{PERMS}'] E\{m/x\}), \end{aligned}$$

where the reduction step can be performed. In fact,  $\text{CALL} \sqsubseteq \text{PERMS}$  by hypothesis (1) and  $\text{PERMS} = \perp$  by (24), i.e.,  $\text{CALL} = \perp$  by anti-symmetry.

Assume then that  $\Gamma \vdash_{\top} [\text{PERMS}] \bar{n} \langle m \triangleright \text{RECV} \rangle : \tau \blacktriangleright \text{P}$  was derived by (T-CALL) after an application of (T-PERMS), then we know that:

$$(25) \quad \Gamma \vdash n : \text{Fun}(\text{CALL}', \tau_n \rightarrow \tau'_n)^{\text{SECR}}$$

$$(26) \quad \perp \sqsubseteq \text{RECV} \sqcup \text{SECR}.$$

Since (18) and (25) hold, by Proposition 3 (Soundness of Secrecy Levels) we know that  $\text{SECR} = \perp$ . Since  $\text{RECV} \sqsubseteq \text{PERMS}'$  by hypothesis (2) and (20) holds, we know that  $\text{RECV} = \perp$  by anti-symmetry, thus  $\text{RECV} \sqcup \text{SECR} = \perp$  and we get a contradiction by (26), i.e., the rule could not be applied and the case is trivial. Finally, assume that  $\Gamma \vdash_{\top} [\text{PERMS}] \bar{n} \langle m \triangleright \text{RECV} \rangle : \tau \blacktriangleright \text{P}$  was derived by (T-FAIL) after an application of (T-PERMS), then we know that:

- (27)  $\Gamma \vdash n : \text{Fun}(\text{CALL}', \tau_n \rightarrow \tau_n')^{\text{SECR}}$
- (28)  $\Gamma \vdash m : \tau_n''$
- (29)  $\text{RECV} \sqcup \text{SECR} = \perp \Rightarrow \mathcal{S}(\tau_n'') = \perp$
- (30)  $\tau = \text{Un}$
- (31)  $P = \text{PERMS}$ .

Since (18) and (27) hold, by Proposition 3 (Soundness of Secrecy Levels) we know that  $\text{SECR} = \perp$ . Since  $\text{RECV} \sqsubseteq \text{PERMS}'$  by hypothesis (2) and (20) holds, we know that  $\text{RECV} = \perp$  by anti-symmetry, thus  $\text{RECV} \sqcup \text{SECR} = \perp$  and we get  $\mathcal{S}(\tau_n'') = \perp$  by (29). This implies, using (28) and (T-PUB), that  $\Gamma \vdash m : \text{Un}$ , hence by (19) and Lemma 3 (Substitution) we get  $\Gamma \vdash_{\perp} [\text{PERMS}'] E\{m/x\} : \tau \blacktriangleright \perp$ , hence we get  $\Gamma \vdash_{\top} [\text{PERMS}'] E\{m/x\} : \tau \blacktriangleright \perp$  by Proposition 5 (Monotonicity of Typing). Now we note that:

$$\begin{aligned}
& \Gamma \cdot (n^{\ell}(x \triangleleft \text{CALL}).[\text{PERMS}'] E \setminus [\text{PERMS}] \bar{n}\langle m \triangleright \text{RECV} \rangle) \\
&= \Gamma \cdot (n^{\ell}(x \triangleleft \text{CALL}).[\perp] E \setminus [\text{PERMS}] \bar{n}\langle m \triangleright \text{RECV} \rangle) && \text{by (20)} \\
&\triangleq n^{\ell}(x \triangleleft \text{CALL}).[\perp] E \setminus [\text{PERMS}] \bar{n}\langle m \triangleright \text{RECV} \rangle && \text{by (31)} \\
&\xrightarrow{\ell}_i [\text{PERMS} \sqcap \perp] E\{m/x\} \\
&= [\perp] E\{m/x\} \\
&\triangleq \Gamma \cdot ([\text{PERMS}'] E\{m/x\}),
\end{aligned}$$

where the reduction step can be performed, since  $\text{CALL} \sqsubseteq \text{PERMS}$  by (1).

*Case (R-LET):* assume  $\text{let } x = E_1 \text{ in } E_2 \xrightarrow{\alpha} \text{let } x = E'_1 \text{ in } E_2$  by the premise  $E_1 \xrightarrow{\alpha} E'_1$ . By hypothesis we know that  $\Gamma \vdash_{\top} \text{let } x = E_1 \text{ in } E_2 : \tau' \blacktriangleright P \sqcup Q$ , which must follow by an instance of (T-LET). Hence, we have  $\Gamma \vdash_{\top} E_1 : \tau \blacktriangleright P$  and  $\Gamma, x : \tau \vdash_{\top} E_2 : \tau' \blacktriangleright Q$ . By inductive hypothesis we have  $\Gamma \vdash_{\top} E'_1 : \tau \blacktriangleright P'$  with  $P' \sqsubseteq P$ , hence  $\Gamma \vdash_{\top} \text{let } x = E'_1 \text{ in } E_2 : \tau \blacktriangleright P' \sqcup Q$  by (T-LET).

Again by inductive hypothesis, we also know that  $\Gamma \cdot E_1 \xrightarrow{\alpha}_i E'_1$  with  $E'_1 \sqsubseteq \Gamma \cdot E'_1$ , hence we have:

$$\begin{aligned}
\Gamma \cdot (\text{let } x = E_1 \text{ in } E_2) &\triangleq \text{let } x = (\Gamma \cdot E_1) \text{ in } ((\Gamma, x : \tau) \cdot E_2) \\
&\xrightarrow{\alpha}_i \text{let } x = E'_1 \text{ in } ((\Gamma, x : \tau) \cdot E_2) \\
&\sqsupseteq \text{let } x = (\Gamma \cdot E'_1) \text{ in } ((\Gamma, x : \tau) \cdot E_2) \\
&\triangleq \Gamma \cdot (\text{let } x = E'_1 \text{ in } E_2).
\end{aligned}$$

*Case (R-RETURN):* assume  $\text{let } x = [\text{PERMS}] n \text{ in } E \xrightarrow{\alpha} E\{n/x\}$ . By hypothesis we know that  $\Gamma \vdash_{\top} \text{let } x = [\text{PERMS}] n \text{ in } E : \tau' \blacktriangleright P \sqcup Q$ , which must follow by an instance of (T-LET). Hence, we have  $\Gamma \vdash_{\top} [\text{PERMS}] n : \tau \blacktriangleright P$  and  $\Gamma, x : \tau \vdash_{\top} E : \tau' \blacktriangleright Q$  with  $x \notin \text{dom}(\Gamma)$ . The former judgement must have been derived by an application of (T-VAL) after an instance of (T-PERMS), thus we know that  $\Gamma \vdash n : \tau$  and by Lemma 3 (Substitution) we have  $\Gamma \vdash_{\top} E\{n/x\} : \tau' \blacktriangleright Q$ .

Now we note that:

$$\begin{aligned}
\Gamma \cdot (\text{let } x = [\text{PERMS}] n \text{ in } E) &\triangleq \text{let } x = [\text{P} \sqcap \text{PERMS}] n \text{ in } ((\Gamma, x : \tau) \cdot E) \\
&\xrightarrow{i} ((\Gamma, x : \tau) \cdot E) \{n/x\} \\
&= \Gamma \cdot (E \{n/x\}).
\end{aligned}$$

The last step uses Lemma 3 (Substitution) and some simple syntactic observations to conclude.

*Case (R-RESTR):* assume  $(\nu n) E \xrightarrow{\alpha} (\nu n) E'$  by the premise  $E \xrightarrow{\alpha} E'$ . By hypothesis we know that  $\Gamma \vdash_{\top} (\nu n) E : \tau' \blacktriangleright \text{PERMS}$ , which must follow by an instance of (T-RESTR), hence we have  $\Gamma, n : \tau \vdash_{\top} E : \tau' \blacktriangleright \text{PERMS}$  with  $n \notin \text{dom}(\Gamma)$ . By inductive hypothesis we have  $\Gamma, n : \tau \vdash_{\top} E' : \tau' \blacktriangleright \text{PERMS}'$  for some  $\text{PERMS}' \sqsubseteq \text{PERMS}$ , hence  $\Gamma \vdash_{\top} (\nu n) E' : \tau' \blacktriangleright \text{PERMS}'$  by (T-RESTR).

Again by inductive hypothesis, we also know that  $(\Gamma, n : \tau) \cdot E \xrightarrow{\alpha}_i E''$  with  $E'' \sqsupseteq (\Gamma, n : \tau) \cdot E'$ , hence we have:

$$\begin{aligned}
\Gamma \cdot (\nu n) E &\triangleq (\nu n) ((\Gamma, n : \tau) \cdot E) \\
&\xrightarrow{\alpha}_i (\nu n) E'' \\
&\sqsupseteq (\nu n) ((\Gamma, n : \tau) \cdot E') \\
&\triangleq \Gamma \cdot (\nu n) E'.
\end{aligned}$$

*Case (R-EVAL):* assume  $D \setminus E \xrightarrow{\alpha} D \setminus E'$  by the premise  $E \xrightarrow{\alpha} E'$ . By hypothesis we know that  $\Gamma \vdash_{\top} D \setminus E : \tau \blacktriangleright \text{PERMS}$ , which must follow by an instance of (T-EVAL), hence we have  $\Gamma \vdash_{\top} D$  and  $\Gamma \vdash_{\top} E : \tau \blacktriangleright \text{PERMS}$ . By inductive hypothesis we have  $\Gamma \vdash_{\top} E' : \tau' \blacktriangleright \text{PERMS}'$  for some  $\text{PERMS}' \sqsubseteq \text{PERMS}$ , hence  $\Gamma \vdash_{\top} D \setminus E : \tau \blacktriangleright \text{PERMS}'$  by (T-STORE).

Again by inductive hypothesis, we also know that  $\Gamma \cdot E \xrightarrow{\alpha}_i E''$  with  $E'' \sqsupseteq \Gamma \cdot E'$ , hence we have:

$$\begin{aligned}
\Gamma \cdot (D \setminus E) &\triangleq D \setminus (\Gamma \cdot E) \\
&\xrightarrow{\alpha}_i D \setminus E'' \\
&\sqsupseteq D \setminus (\Gamma \cdot E') \\
&\triangleq \Gamma \cdot (D \setminus E').
\end{aligned}$$

*Case (R-STRUCT):* assume  $E \xrightarrow{\alpha} E'$  by the premises  $E \Rightarrow E_1$ ,  $E_1 \xrightarrow{\alpha} E_2$  and  $E_2 \Rightarrow E'$ . By hypothesis we know that  $\Gamma \vdash_{\top} E : \tau \blacktriangleright \text{PERMS}$ , hence  $\Gamma \vdash_{\top} E_1 : \tau \blacktriangleright \text{PERMS}$  by Lemma 4 (Heating Preserves Typing). By inductive hypothesis we then have  $\Gamma \vdash_{\top} E_2 : \tau \blacktriangleright \text{PERMS}'$  for some  $\text{PERMS}' \sqsubseteq \text{PERMS}$ , hence we have  $\Gamma \vdash_{\top} E' : \tau \blacktriangleright \text{PERMS}'$  again by Lemma 4 (Heating Preserves Typing).

Now we show the second part of the statement. Using Lemma 6 (Lowering Respects Heating), by  $E \Rightarrow E_1$  we have  $\Gamma \cdot E \Rightarrow E'' \sqsupseteq \Gamma \cdot E_1$  for some  $E''$ . By inductive hypothesis, we know that  $\Gamma \cdot E_1 \xrightarrow{\alpha}_i E'_1 \sqsupseteq \Gamma \cdot E_2$  for some  $E'_1$ , hence

by Lemma 8 (Monotonicity of Reduction) we get  $E'' \xrightarrow{\alpha}_i E_1'' \sqsupseteq E_1' \sqsupseteq \Gamma \cdot E_2$  for some  $E_1''$ . Using again Lemma 6 (Lowering Respects Heating), by  $E_2 \Rightarrow E'$  we have  $\Gamma \cdot E_2 \Rightarrow E_2'' \sqsupseteq \Gamma \cdot E'$  for some  $E_2''$ . By Lemma 7 (Monotonicity of Heating) we then have  $E_1'' \Rightarrow E_2' \sqsupseteq E_2''$  for some  $E_2'$  and we conclude  $\Gamma \cdot E \xrightarrow{\alpha}_i E_2' \sqsupseteq \Gamma \cdot E'$  by an application of (R-STRUCT).

### B.3 Proof of (robust) safety

**Proposition 6 (Equivalence up to Permissions).** *The following statements hold:*

- (i) *for every pair of expressions  $E_1, E_2$  such that  $E_1 \sqsubseteq E_2$ , we have  $E_1 \asymp E_2$ ;*
- (ii) *for every expression  $E$  such that  $\Gamma \cdot E$  is defined, we have  $\Gamma \cdot E \asymp E$ .*

*Proof.* Point (i) follows by induction on the derivation of  $E_1 \sqsubseteq E_2$ , while point (ii) follows by Proposition 4 (Soundness of Lowering) and point (i).

**Restatement of Theorem 1.** *If  $\Gamma \vdash_{\top} E : \tau \blacktriangleright P$ , then  $E \preceq_{IPC} E$ .*

*Proof.* Let  $\mathcal{R} = \{(E_1, E_2) \mid \Gamma \vdash_{\top} E_1 : \tau \blacktriangleright P_1 \wedge E_2 \sqsupseteq \Gamma \cdot E_1\}$ . We show that  $\mathcal{R}$  is a simulation. Notice first that for every  $E_1, E_2$  such that  $(E_1, E_2) \in \mathcal{R}$  we have  $E_1 \asymp E_2$  by Proposition 6 (Equivalence up to Permissions) and the transitivity of the  $\asymp$  relation, hence we just need to show that the transitions match as prescribed.

Let  $(E_1, E_2) \in \mathcal{R}$ , then we know that  $\Gamma \vdash_{\top} E_1 : \tau \blacktriangleright P_1$  and  $E_2 \sqsupseteq \Gamma \cdot E_1$ . Assume  $E_1 \xrightarrow{\alpha} E_1'$ , then by Theorem 3 (Simulation-Aware Subject Reduction) we have  $\Gamma \vdash_{\top} E_1' : \tau \blacktriangleright P_1'$  with  $P_1' \sqsubseteq P_1$  and  $\Gamma \cdot E_1 \xrightarrow{\alpha}_i E_1''$  for some  $E_1'' \sqsupseteq \Gamma \cdot E_1'$ . By Lemma 8 (Monotonicity of Reduction) we then have  $E_2 \xrightarrow{\alpha}_i E_2'$  for some  $E_2' \sqsupseteq E_1'' \sqsupseteq \Gamma \cdot E_1'$ , hence  $(E_1', E_2') \in \mathcal{R}$  and we conclude that  $\mathcal{R}$  is a simulation.

Finally, we note that by Proposition 4 (Soundness of Lowering) we have  $E \sqsupseteq \Gamma \cdot E$ , hence  $(E, E) \in \mathcal{R}$  and we conclude  $E \preceq_{IPC} E$  as desired.

**Restatement of Lemma 1.** *Let  $O$  be an opponent and let  $\Gamma \vdash u : \text{Un}$  for all  $u \in \text{fnfv}(O)$ , then  $\Gamma \vdash O$ .*

*Proof.* Let  $E$  be any expression such that each permission assignment occurring within  $E$  is  $\perp$ . Since the structure of definitions and expressions is given by mutually inductive productions, we simultaneously prove the following statements:

- (i)  $\forall u \in \text{fnfv}(E) : \Gamma \vdash u : \text{Un} \Rightarrow \Gamma \vdash_{\perp} E : \text{Un} \blacktriangleright \perp$
- (ii)  $\forall u \in \text{fnfv}(O) : \Gamma \vdash u : \text{Un} \Rightarrow \Gamma \vdash O$ .

The proof of point (i) is by induction on the structure of  $E$ , while the proof of point (ii) is by induction on the structure of  $O$ .

**Restatement of Theorem 2.** *Let  $\mathcal{S}(\tau) = \perp$  for every  $u$  such that  $\Gamma(u) = \tau$ . If  $\Gamma \vdash_{\top} E : \tau \blacktriangleright P$ , then  $E$  is robustly safe against privilege escalation attacks.*



*Proof.* Let  $O$  be an arbitrary opponent. Let  $\Gamma^*$  be the typing environment defined as follows:

$$\Gamma^*(u) = \begin{cases} \Gamma(u) & \text{if } u \in \text{dom}(\Gamma) \\ \text{Un} & \text{if } u \notin \text{dom}(\Gamma) \wedge u \in \text{fnfv}(O) \end{cases}$$

We let  $\Gamma^*(u)$  be undefined for any  $u$  such that  $u \notin \text{dom}(\Gamma) \cup \text{fnfv}(O)$ .

Now we note that  $\forall u \in \text{dom}(\Gamma^*) : \Gamma^* \vdash u : \text{Un}$ , hence  $\Gamma^* \vdash O$  by Lemma 1 (Opponent Typability). By Lemma 2 (Weakening) we also have  $\Gamma^* \vdash_{\top} E : \tau \blacktriangleright \mathbf{P}$ , thus  $\Gamma^* \vdash_{\top} O \setminus E : \tau \blacktriangleright \mathbf{P}$  by rule (T-EVAL). Hence, the conclusion follows by Theorem 1 (Type Safety).