A Calculus for Flow-Limited Authorization

Technical Report April 22, 2021

Owen Arden[a], Anitha Gollamudi[b], Ethan Cecchetti[c], Stephen Chong[b], and Andrew C. Myers[c]

[a]UC Santa Cruz, Santa Cruz, CA, USA
  Email: owen@soe.ucsc.edu
[b]Harvard University, Cambridge, MA, USA
  Email: agollamudi@g.harvard.edu, chong@seas.harvard.edu
[c]Cornell University, Ithaca, NY, USA
  Email: ethan@cs.cornell.edu, andru@cs.cornell.edu

Abstract. Real-world applications routinely make authorization decisions based on dynamic computation. Reasoning about dynamically computed authority is challenging. Integrity of the system might be compromised if attackers can improperly influence the authorizing computation. Confidentiality can also be compromised by authorization, since authorization decisions are often based on sensitive data such as membership lists and passwords. Previous formal models for authorization do not fully address the security implications of permitting trust relationships to change, which limits their ability to reason about authority that derives from dynamic computation. Our goal is an approach to constructing dynamic authorization mechanisms that do not violate confidentiality or integrity.

The Flow-Limited Authorization Calculus (FLAC) is a simple, expressive model for reasoning about dynamic authorization as well as an information flow control language for securely implementing various authorization mechanisms. FLAC combines the insights of two previous models: it extends the Dependency Core Calculus with features made possible by the Flow-Limited Authorization Model. FLAC provides strong end-to-end information security guarantees even for programs that incorporate and implement rich dynamic authorization mechanisms. These guarantees include noninterference and robust declassification, which prevent attackers from influencing information disclosures in unauthorized ways. We prove these security properties formally for all FLAC programs and explore the expressiveness of FLAC with several examples.

1 Introduction

Authorization mechanisms are critical components in all distributed systems. The policies enforced by these mechanisms constrain what computation may be safely executed, and therefore an expressive policy language is important. Expressive mechanisms for authorization have been an active research area. A variety of approaches have been developed, including authorization logics [33, 2, 48], often implemented with cryptographic mechanisms [22, 15]; role-based access control (RBAC) [23]; and trust management [34, 50, 11].

However, the security guarantees of authorization mechanisms are usually analyzed using formal models that abstract away the computation and communication performed by the system. Developers must take great care to faithfully preserve the (often implicit) assumptions of the model, not only when implementing authorization mechanisms, but also when employing them. Simplifying abstractions can help extract formal security guarantees, but abstractions can also obscure the challenges of implementing and using an abstraction securely. This disconnect between abstraction and implementation can lead to vulnerabilities and covert channels that allow attackers to leak or corrupt information.

A common blind spot in many authorization models is confidentiality. Most models cannot express authorization policies that are confidential or are based on confidential data. Real systems, however, use confidential data for authorization all the time: users on social networks receive access to photos based on friend lists, frequent fliers receive tickets based on credit card purchase histories, and doctors exchange patient data while keeping doctor–patient relationships confidential. While many models can ensure, for instance, that only friends are permitted to access a photo,

few can say anything about the secondary goal of preserving the confidentiality of the friend list. Such authorization schemes may fundamentally require *some* information to be disclosed, but failing to detect these disclosures can lead to unintentional leaks.

Authorization without integrity is meaningless, so formal authorization models are typically better at enforcing integrity. However, many formal models make unreasonable or unintuitive assumptions about integrity. For instance, in many models (e.g., [33], [2], [34]) authorization policies either do not change or change only when modified by a trusted administrator. This is a reasonable assumption in centralized systems where such an administrator will always exist, but in decentralized systems, there may be no single entity that is trusted by all other entities.

Even in centralized systems, administrators must be careful when performing updates based on partially trusted information, since malicious users may try to confuse or mislead the administrator into carrying out an attack on their behalf. Unfortunately, existing authorization models offer little help to administrators that need to reason about how attackers may have influenced security-critical update operations.

Developers need a better programming model for implementing expressive dynamic authorization mechanisms. Errors that undermine the security of these mechanisms are common [35], so we want to be able to verify their security. We argue that information flow control (IFC) is a lightweight, useful tool for building secure authorization mechanisms since it offers compositional, end-to-end security guarantees. However, applying IFC to authorization mechanisms in a meaningful way requires building on a theory that integrates authority and information security. In this work, we show how to embed such a theory into a programming model, so that dynamic authorization mechanisms—as well as the programs that employ them—can be statically verified.

Approaching the verification of dynamic authorization mechanisms from this perspective is attractive for two reasons. First, it gives a model for building secure authorization mechanisms by construction rather than verifying them after the fact. This model offers programmers insight into the subtle interaction between information flow and authorization, and helps programmers address problems early, during the design process. Second, it addresses a core weakness lurking at the heart of existing language-based security schemes: that the underlying policies may change in a way that breaks security. By statically verifying the information security of dynamic authorization mechanisms, we expand the real-world scenarios in which language-based information flow control is useful and strengthen its security guarantees.

We demonstrate that such an embedding is possible by presenting a core language for authorization and information flow control, called the Flow-Limited Authorization Calculus (FLAC). FLAC is a functional language for designing and verifying decentralized authorization protocols. FLAC is inspired by the Polymorphic Dependency Core Calculus [2] (DCC).[1] Abadi develops DCC as an authorization logic, but DCC is limited to static trust relationships defined externally to DCC programs by a lattice of principals. FLAC supports dynamic authorization by building on the Flow-Limited Authorization Model (FLAM) [9], which unifies reasoning about authority, confidentiality, and integrity. Furthermore, FLAC is a language for information flow control. It uses FLAM's principal model and FLAM's logical reasoning rules to define an operational model and type system for authorization computations that preserve information security. George [25] also extends DCC's monadic approach to model information flow in authorization mechanisms, with more explicit modeling of a distributed execution environment.

The types in a FLAC program can be considered propositions [53] in an authorization logic, and the programs can be considered proofs that the proposition holds. Well-typed FLAC programs are both proofs of secure information flow and proofs of authorization, ensuring the confidentiality and integrity of not only data, but also authorization policies.

FLAC is useful from a logical perspective, but also serves as a core programming model for real language implementations. Since FLAC programs can dynamically authorize computation and flows of information, FLAC applies to more realistic settings than previous authorization logics. Thus FLAC offers more than a language and type system for proving propositions—FLAC programs do useful computation.

This paper makes the following contributions.

- We define FLAC, a language, type system, and semantics for dynamic authorization mechanisms with strong information security:

  - Programs in low-integrity contexts exhibit *noninterference*, ensuring attackers cannot leak or corrupt information, and cannot subvert authorization mechanisms.

  - Programs in higher-integrity contexts exhibit *robust declassification*, ensuring attackers cannot influence authorized disclosures of information.

---

[1]DCC was first presented in [4]. We use the abbreviation DCC to refer to the extension to polymorphic types in [2].

- We present two authorization mechanisms implemented in FLAC, commitment schemes and bearer credentials, and demonstrate that FLAC ensures the programs that use these mechanisms preserve the desired confidentiality and integrity properties.

We have organized our discussion of FLAC as follows. Section 2 introduces commitment schemes and bearer credentials, two examples of dynamic authorization mechanisms we use to explore the features of FLAC. Section 3 reviews the FLAM principal lattice [9], and Section 4 defines the FLAC language and type system. FLAC implementations of the dynamic authorization examples are presented in Section 7, and their properties are examined. Section 5 explores aspects of FLAC's proof theory, and Section 6 discusses semantic security guarantees of FLAC programs, including noninterference and robust declassification. We explore related work in Section 8 and conclude in Section 9.

Many of the contributions of this paper were previously published by Arden and Myers [7]. This article expands upon and strengthens the formal results, as well as corrects several technical errors in that work. This includes a more detailed treatment of how FLAC constrains delegations and a more general noninterference theorem on a new formal result regarding the compartmentalization of delegations. Most of the semantic security proofs in Section 6 are either new to this work or were redeveloped from scratch to account for changes made to the semantics and type system.

Furthermore, subsequent work based on FLAC by Cecchetti *et al.* [18] and Gollamudi *et al.* [26] exposed alternate design decisions that improve the connection between FLAC and cryptographic implementations of its protection abstractions. We feel some of these changes are objectively better than the original abstractions, and so incorporate them into the core FLAC formalism so that future work based on FLAC may benefit. Significant departures from the original formalization are footnoted, but minor changes and corrections are included without comment.

2 Dynamic authorization mechanisms

Dynamic authorization is challenging to implement correctly since authority, confidentiality, and integrity interact in subtle ways. FLAC helps programmers securely implement both authorization mechanisms and the code that uses them. FLAC types support the definition of compositional security abstractions, and vulnerabilities in the implementations of these abstractions are caught statically. Further, the guarantees offered by FLAC simplify reasoning about the security properties of these abstractions.

We illustrate the usefulness and expressive power of FLAC using two important security mechanisms: commitment schemes and bearer credentials. We show in Section 7 that these mechanisms can be implemented using FLAC, and that their security goals are easily verified in the context of FLAC.

2.1 Commitment schemes

A commitment scheme [43] allows one party to give another party a "commitment" to a secret value without revealing the value. The committing party may later reveal the secret in a way that convinces the receiver that the revealed value is the value originally committed.

Commitment schemes provide three essential operations: `commit`, `reveal`, and `open`.[2] Suppose p wants to commit to a value to principal q. First, p applies `commit` to the value and provides the result to q without revealing the committed value to q. When p wishes to reveal the value, it gives q a `reveal` operation q can use to open the previously sent commitment. Then q uses `reveal` to open the committed value, finally revealing it. In a cryptographic implementation, the `reveal` operation might correspond to the secret p used to encrypt the commitment, and the open operation might correspond to q using that secret to decrypt the commitment.

A commitment scheme must have several properties in order to be secure. First, q should not be able to open a value that p has not committed to, since this could allow q to manipulate p to open a value it had not committed to. Second, q should not be able to learn a secret of p that has not been committed to or revealed by p. Third, p should not be able to modify the committed value after it is received by q. Specifically, the `reveal` operation should only reveal the committed value and not modify it in any way.

One might wonder why a programmer would bother to create high-level *implementations* of operations like `commit`, `reveal`, and `open`. Why not simply treat these as primitive operations and give them type signatures so that programs using them can be type-checked with respect to those signatures? The answer is that an error in a type signature could lead to a serious vulnerability. Therefore, we want more assurance that the type signatures are correct. Modeling

---

[2]This commitment scheme example differs from the one presented in Arden and Myers [7], which is not compatible with changes to the type system presented here. Furthermore, we feel the API presented here is a better representation of the operations present in cryptographic commitment schemes.

such operations in FLAC is often easy and ensures that the type signature is consistent with a set of assumptions about existing trust relationships and the information flow context the operations are used within. These FLAC-based implementations serve as language-based specifications of the security properties required by implementations that use cryptography or trusted third parties.

2.2   Bearer credentials with caveats

A bearer credential is a capability that grants authority to any entity that possesses it. Many authorization mechanisms used in distributed systems employ bearer credentials in some form. Browser cookies that store session tokens are one example: after a website authenticates a user's identity, it gives the user a token to use in subsequent interactions. Since it is infeasible for attackers to guess the token, the website grants the authority of the user to any requests that include the token.

Bearer credentials create an information security conundrum for authorization mechanisms. Though they efficiently control access to restricted resources, they create vulnerabilities and introduce covert channels when used incorrectly. For example, suppose Alice shares a remotely hosted photo with her friends by giving them a credential to access the photo. Giving a friend such a credential doesn't disclose their friendship, but each friend that accesses the photo implicitly discloses the friendship to the hosting service. Such covert channels are pervasive, both in classic distributed authorization mechanisms like SPKI/SDSI [22], as well as in more recent ones like Macaroons [15].

Bearer credentials can also lead to vulnerabilities if they are leaked. If an attacker obtains a credential, it can exploit the authority of the credential. Thus, to limit the authority of a credential, approaches like SPKI/SDSI and Macaroons provide *constrained delegation* in which a newly issued credential attenuates the authority of an existing one by adding *caveats*. Caveats require additional properties to hold for the bearer to be granted authority. Session tokens, for example, might have a caveat that restricts the source IP address or encodes an expiration time. As pointed out by Birgisson et al. [15], caveats themselves can introduce covert channels if the properties reveal sensitive information.

FLAC is an effective framework for reasoning about bearer credentials with caveats since it captures the flow of credentials in programs as well as the sensitivity of the information the credentials and caveats derive from. We can reason about credentials and the programs that use them in FLAC with an approach similar to that used for commitment schemes. That we can do so in a straightforward way is somewhat remarkable: prior formalizations of credential mechanisms (e.g., [15, 30, 13]) usually do not consider confidentiality nor provide end-to-end guarantees about credential propagation.

3   The FLAM Principal Lattice

Like many models, FLAM uses *principals* to represent the authority of all entities relevant to a system. However, FLAM's principals and their algebraic properties are richer than in most models, so we briefly review the FLAM principal model and notation. Further details are found in the earlier paper [9].

Primitive principals such as Alice, Bob, etc., are represented as elements n of a (potentially infinite) set of names $\mathcal{N}$.[3] In addition to these names, FLAM uses $\top$ to represent a universally trusted primitive principal and $\bot$ to represent a universally untrusted primitive principal. The combined authority of two principals, p and q, is represented by the authority conjunction $p \wedge q$, whereas the authority of either p or q is the disjunction $p \vee q$.

Unlike principals in other models, FLAM principals also represent information flow policies. The confidentiality of principal p is represented by the principal $p^{\rightarrow}$, called p's confidentiality projection. It denotes the authority necessary to *learn* anything p can learn. The integrity of principal p is represented by $p^{\leftarrow}$, called p's integrity projection. It denotes the authority to *influence* anything p can influence.

These projections, conjunctions, and disjunctions allow us to construct the set of all principals from any set of names $\mathcal{N}$. The set $\mathcal{N} \cup \{\top, \bot\}$ under the syntax operators[4] $\wedge, \vee, \leftarrow, \rightarrow$ forms a set $\mathcal{L}$. The equivalence classes of this set

---

[3]Using $\mathcal{N}$ as the set of all names is convenient in our formal calculus, but a general-purpose language based on FLAC may wish to dynamically allocate names at runtime. Since knowing or using a principal's name holds no special privilege in FLAC, this presents no fundamental difficulties. To use dynamically allocated principals in type signatures, however, the language's type system should support types in which principal names may be existentially quantified.

[4]FLAM defines an additional set of operators called *ownership projections*, which we omit here to simplify our presentation.

$\boxed{\mathcal{L} \vDash p \geqslant q}$

[BOT] $\mathcal{L} \vDash p \geqslant \bot$ [TOP] $\mathcal{L} \vDash \top \geqslant p$ [REFL] $\mathcal{L} \vDash p \geqslant p$ [TRANS] $\dfrac{\mathcal{L} \vDash p \geqslant q \quad \mathcal{L} \vDash q \geqslant r}{\mathcal{L} \vDash p \geqslant r}$

[PROJ] $\dfrac{\mathcal{L} \vDash p \geqslant q}{\mathcal{L} \vDash p^\pi \geqslant q^\pi}$ [PROJR] $\mathcal{L} \vDash p \geqslant p^\pi$ [PROJIDEMP] $\mathcal{L} \vDash (p^\pi)^\pi \geqslant p^\pi$ [PROJBASIS] $\dfrac{\pi \neq \pi'}{\mathcal{L} \vDash \bot \geqslant (p^\pi)^{\pi'}}$

[PROJDISTCONJ] $\mathcal{L} \vDash p^\pi \wedge q^\pi \geqslant (p \wedge q)^\pi$ [PROJDISTDISJ] $\mathcal{L} \vDash (p \vee q)^\pi \geqslant p^\pi \vee q^\pi$

[CONJL] $\dfrac{\begin{array}{c}\mathcal{L} \vDash p_k \geqslant p \\ k \in \{1, 2\}\end{array}}{\mathcal{L} \vDash p_1 \wedge p_2 \geqslant p}$ [CONJR] $\dfrac{\begin{array}{c}\mathcal{L} \vDash p \geqslant p_1 \\ \mathcal{L} \vDash p \geqslant p_2\end{array}}{\mathcal{L} \vDash p \geqslant p_1 \wedge p_2}$ [CONJBASIS] $\mathcal{L} \vDash p^\rightarrow \wedge p^\leftarrow \geqslant p$

[CONJDISTDISJL] $\mathcal{L} \vDash (p \wedge q) \vee (p \wedge r) \geqslant p \wedge (q \vee r)$ [CONJDISTDISJR] $\mathcal{L} \vDash p \wedge (q \vee r) \geqslant (p \wedge q) \vee (p \wedge r)$

[DISJL] $\dfrac{\begin{array}{c}\mathcal{L} \vDash p_1 \geqslant p \\ \mathcal{L} \vDash p_2 \geqslant p\end{array}}{\mathcal{L} \vDash p_1 \vee p_2 \geqslant p}$ [DISJR] $\dfrac{\begin{array}{c}\mathcal{L} \vDash p \geqslant p_k \\ k \in \{1, 2\}\end{array}}{\mathcal{L} \vDash p \geqslant p_1 \vee p_2}$ [DISJBASIS] $\mathcal{L} \vDash \bot \geqslant p^\rightarrow \vee q^\leftarrow$

[DISJDISTCONJL] $\mathcal{L} \vDash (p \vee q) \wedge (p \vee r) \geqslant p \vee (q \wedge r)$ [DISJDISTCONJR] $\mathcal{L} \vDash p \vee (q \wedge r) \geqslant (p \vee q) \wedge (p \vee r)$

Figure 1: Static principal lattice rules. The projection π may be either confidentiality ($\rightarrow$) or integrity ($\leftarrow$). Adapted from (and equivalent to) the non-ownership fragment of FLAM's principal algebra [9].

under the *acts-for relation*[5] $\geqslant$ defined in Figure 1[6] form a complete distributive lattice. Using these rules we can derive the equivalences from FLAM's principal algebra. Two principals are considered equivalent if they act for each other:

$$p \equiv q \triangleq \mathcal{L} \vDash p \geqslant q \text{ and } \mathcal{L} \vDash q \geqslant p$$

We have proven in Coq [5] that this definition is equivalent to the algebraic definition (minus ownership) used in the FLAM Coq formalization [10]. We write operators $\leftarrow, \rightarrow$ with higher precedence than $\wedge, \vee$; for instance, $p \wedge q^\leftarrow$ is the same as writing $p \wedge (q^\leftarrow)$. Projections distribute over $\wedge$ and $\vee$ (rules PROJDISTCONJ and PROJDISTDISJ) so, for example, $(p \wedge q)^\leftarrow \equiv (p^\leftarrow \wedge q^\leftarrow)$.

All authority may be represented as some combination of confidentiality and integrity. Any principal p is equivalent to, via rules PROJR and CONJBASIS, the conjunction of its confidentiality and integrity authority: $p^\rightarrow \wedge p^\leftarrow$. In fact, any principal can be normalized [9] to $q^\rightarrow \wedge r^\leftarrow$ for some q and r. For example, Alice$^\rightarrow \wedge$ Bob is equivalent to (Alice $\wedge$ Bob)$^\rightarrow \wedge$ $Bob^\leftarrow$. The confidentiality and integrity authority of principals are disjoint (rule PROJBASIS), so the confidentiality projection of an integrity projection is $\bot$ and vice-versa: $(p^\leftarrow)^\rightarrow \equiv \bot \equiv (p^\rightarrow)^\leftarrow$.

An advantage of this model is that secure information flow can be defined in terms of authority. An information flow policy q is at least as *restrictive* as a policy p if q has at least the confidentiality authority $p^\rightarrow$ and p has at least the integrity authority $q^\leftarrow$. This relationship between the confidentiality and integrity of p and q reflects the usual duality seen in information flow control [14]. As in [9], we use the following shorthand for relating principals by policy

---

[5]FLAM's acts-for relation is inspired by the acts-for relation defined by the Decentralized Label Model [40] (DLM). In the DLM, the principal with the most authority is referred to as $\top$, and the upper bound of the authority of two principals is written $\wedge$, and the lower bound with $\vee$. Unfortunately, this departs from the usual notational conventions for lattices, where $\wedge$ is used for lower bounds (meets), and $\vee$ is used for upper bounds. We have stubbornly held on to the DLM-based acts-for notation, but this sometimes creates confusion for those more familiar with the standard lattice notation.

[6]The original FLAC formalization presented a set of static rules derived from FLAM's principal algebra, but still relied on FLAM's algebraic identities for completeness. Here we present a complete set of static rules and have proven their equivalence to FLAM's principal algebra (without ownership projections) in Coq [5].

restrictiveness:

$$p \sqsubseteq q \triangleq (p^{\leftarrow} \wedge q^{\rightarrow}) \geqslant (q^{\leftarrow} \wedge p^{\rightarrow})$$
$$p \sqcup q \triangleq (p \wedge q)^{\rightarrow} \wedge (p \vee q)^{\leftarrow}$$
$$p \sqcap q \triangleq (p \vee q)^{\rightarrow} \wedge (p \wedge q)^{\leftarrow}$$

Thus, $p \sqsubseteq q$ indicates the direction of secure information flow: from p to q. The information flow join $p \sqcup q$ is the least restrictive principal that both p and q flow to, and the information flow meet $p \sqcap q$ is the most restrictive principal that flows to both p and q.

An interesting feature of this definition is that the equivalence classes under $\geqslant$ and $\sqsubseteq$ are the same.

$$
\begin{aligned}
p \equiv q &= \mathcal{L} \models p \geqslant q \text{ and } \mathcal{L} \models q \geqslant p \\
&= \mathcal{L} \models p^{\rightarrow} \wedge p^{\leftarrow} \geqslant q^{\rightarrow} \wedge q^{\leftarrow} \text{ and } \mathcal{L} \models q^{\rightarrow} \wedge q^{\leftarrow} \geqslant p^{\rightarrow} \wedge p^{\leftarrow} \\
&= \mathcal{L} \models p^{\rightarrow} \geqslant q^{\rightarrow} \text{ and } \mathcal{L} \models q^{\rightarrow} \geqslant p^{\rightarrow} \\
&\quad \text{ and } \mathcal{L} \models p^{\leftarrow} \geqslant q^{\leftarrow} \text{ and } \mathcal{L} \models q^{\leftarrow} \geqslant p^{\leftarrow} \\
&= \mathcal{L} \models p^{\leftarrow} \wedge q^{\rightarrow} \geqslant q^{\leftarrow} \wedge p^{\rightarrow} \text{ and } \mathcal{L} \models q^{\leftarrow} \wedge p^{\rightarrow} \geqslant p^{\leftarrow} \wedge q^{\rightarrow} \\
&= \mathcal{L} \models p \sqsubseteq q \text{ and } \mathcal{L} \models q \sqsubseteq p
\end{aligned}
$$

These equivalence classes also form a (separate) complete distributive lattice ordered by $\sqsubseteq$ where $\sqcup$ is a join and $\sqcap$ is a meet. In this lattice, the *secret and untrusted* principal $\top^{\rightarrow} \wedge \bot^{\leftarrow}$ is the top element since it is the most restrictive information flow policy. Likewise, the *public and trusted* principal $\bot^{\rightarrow} \wedge \top^{\leftarrow}$ is the bottom element since it is the least restrictive policy. Because of this tight relationship between $\geqslant$ and $\sqsubseteq$ we can use the inference rules in Figure 1 to reason about the relationship between principals in both the authority lattice and the information flow lattice. We often write projected principals by themselves, but these principals are equivalent to the conjunction of themselves with the bottom element of the missing projection: e.g., $p^{\rightarrow} \equiv p^{\rightarrow} \wedge \bot^{\leftarrow}$.

In FLAM, the ability to "speak for" another principal is an integrity relationship between principals. This makes sense intuitively, because speaking for another principal influences that principal's trust relationships and information flow policies. FLAM defines the *voice* of a principal p, written $\nabla(p)$, as the integrity necessary to speak for that principal. Given a principal expressed in normal form[7] as $q^{\rightarrow} \wedge r^{\leftarrow}$, the voice of that principal is

$$\nabla(q^{\rightarrow} \wedge r^{\leftarrow}) \triangleq q^{\leftarrow} \wedge r^{\leftarrow}$$

For example, the voice of `Alice`, $\nabla(\texttt{Alice})$, is $\texttt{Alice}^{\leftarrow}$. The voice of `Alice`'s confidentiality $\nabla(\texttt{Alice}^{\rightarrow})$ is also $\texttt{Alice}^{\leftarrow}$.

All primitive principals speak for themselves: e.g., $\mathcal{L} \models \texttt{Alice} \geqslant \nabla(Alice)$, but principals with asymmetric confidentiality and integrity authority may not:

$$\mathcal{L} \nvDash \texttt{Alice}^{\rightarrow} \wedge \texttt{Bob}^{\leftarrow} \geqslant \nabla(\texttt{Alice}^{\rightarrow} \wedge \texttt{Bob}^{\leftarrow})$$
$$\mathcal{L} \nvDash \texttt{Alice}^{\rightarrow} \wedge \texttt{Bob}^{\leftarrow} \geqslant \texttt{Alice}^{\leftarrow} \wedge \texttt{Bob}^{\leftarrow}$$

4   Flow-Limited Authorization Calculus

FLAC uses information flow to reason about the security implications of dynamically computed authority. Like previous information-flow type systems [47], FLAC incorporates types for reasoning about information flow, but FLAC's type system goes further by using Flow-Limited Authorization [9] to ensure that principals cannot use FLAC programs to exceed their authority, or to leak or corrupt information. FLAC is based on DCC [2], but unlike DCC, FLAC supports reasoning about authority that derives from the evaluation of FLAC terms. In contrast, all authority in DCC derives from trust relationships defined by a fixed, external lattice of principals. Thus, using an approach based on DCC in systems where trust relationships change dynamically could introduce vulnerabilities like delegation loopholes, probing and poaching attacks, and authorization side channels [9].

Figure 2 defines the FLAC syntax. The core FLAC operational semantics and evaluation contexts [55] in Figure 3 are mostly standard except for E-ASSUME and E-UNITM, which we discuss below, along with additional rules that handle the propagation of the `where` terms introduced by E-ASSUME.

The core FLAC type system is presented in Figure 4. FLAC typing judgments have the form $\Pi; \Gamma; pc \vdash e : \tau$. The *delegation context*, Π, contains a set of dynamic trust relationships $\langle p \geqslant q \rangle$ where $p \geqslant q$ (read as "p acts for q") is a

---

[7]In normal form, a principal is the conjunction of a confidentiality principal and an integrity principal. See [9] for details.

$n \in \mathcal{N}$ (principal names)
$x \in \mathcal{V}$ (variable names)

$$
\begin{aligned}
p, \ell, pc \quad ::=& \quad n \mid \top \mid \bot \mid p^{\rightarrow} \mid p^{\leftarrow} \mid p \wedge p \mid p \vee p \\
\tau \quad ::=& \quad (p \succcurlyeq p) \mid \mathsf{unit} \mid \tau + \tau \mid \tau \times \tau \\
& \quad \mid \tau \xrightarrow{pc} \tau \mid \ell \; \mathsf{says} \; \tau \mid X \mid \forall X[pc].\, \tau \\
v \quad ::=& \quad () \mid \langle w, w \rangle \mid \langle p \succcurlyeq p \rangle \mid \overline{\eta}_\ell \, w \mid \mathsf{inj}_i \, w \\
& \quad \mid \lambda(x{:}\tau)[pc].\, e \;(\mathsf{closed}) \mid \Lambda X[pc].\, e \;(\mathsf{closed}) \\
w \quad ::=& \quad v \mid w \; \mathsf{where} \; v \\
e \quad ::=& \quad x \mid w \mid e \, e \mid \langle e, e \rangle \mid \eta_\ell \, e \\
& \quad \mid e \, \tau \mid \mathsf{proj}_i \, e \mid \mathsf{inj}_i \, e \\
& \quad \mid \lambda(x{:}\tau)[pc].\, e \mid \Lambda X[pc].\, e \\
& \quad \mid \mathsf{case} \; e \; \mathsf{of} \; \mathsf{inj}_1(x).\, e \mid \mathsf{inj}_2(x).\, e \\
& \quad \mid \mathsf{bind} \; x = e \; \mathsf{in} \; e \mid \mathsf{assume} \; e \; \mathsf{in} \; e \\
& \quad \mid e \; \mathsf{where} \; v
\end{aligned}
$$

Figure 2: FLAC syntax. Terms using where are syntactically prohibited in the source language and are produced only during evaluation.

$\boxed{e \longrightarrow e'}$

[E-APP] $\qquad (\lambda(x{:}\tau)[pc].\, e) \, w \longrightarrow e[x \mapsto w]$ $\qquad$ [E-TAPP] $\qquad (\Lambda X[pc].\, e) \, \tau \longrightarrow e[X \mapsto \tau]$

[E-UNPAIR] $\quad \mathsf{proj}_i \langle w_1, w_2 \rangle \longrightarrow w_i$ $\qquad$ [E-CASE] $\quad (\mathsf{case} \; (\mathsf{inj}_i \, w) \; \mathsf{of} \; \mathsf{inj}_1(x).\, e_1 \mid \mathsf{inj}_2(x).\, e_2) \longrightarrow e_i[x \mapsto w]$

[E-BINDM] $\quad \mathsf{bind} \; x = \overline{\eta}_\ell \, w \; \mathsf{in} \; e \longrightarrow e[x \mapsto w]$ $\qquad$ [E-ASSUME] $\quad \mathsf{assume} \; \langle p \succcurlyeq q \rangle \; \mathsf{in} \; e \longrightarrow e \; \mathsf{where} \; \langle p \succcurlyeq q \rangle$

[E-UNITM] $\qquad \eta_\ell \, w \longrightarrow \overline{\eta}_\ell \, w$ $\qquad$ [E-EVAL] $\qquad \dfrac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$

$$
\begin{aligned}
E \quad ::=& \quad [\cdot] \mid E \, e \mid w \, E \mid E \, \tau \mid \langle E, e \rangle \mid \langle w, E \rangle \mid \mathsf{proj}_i \, E \mid \mathsf{inj}_i \, E \mid \eta_\ell \, E \\
& \quad \mid \mathsf{bind} \; x = E \; \mathsf{in} \; e \mid \mathsf{assume} \; E \; \mathsf{in} \; e \mid \mathsf{case} \; E \; \mathsf{of} \; \mathsf{inj}_1(x).\, e \mid \mathsf{inj}_2(x).\, e \mid E \; \mathsf{where} \; v
\end{aligned}
$$

Figure 3: FLAC operational semantics

delegation from q to p. The *typing context*, Γ, is a associates variables to types, and *pc* is the *program counter label*, a FLAM principal representing the confidentiality and integrity of control flow. The type system makes frequent use of judgments of the form $\Pi \Vdash p \succcurlyeq q$ and $\Pi \Vdash p \sqsubseteq q$ for comparisons between principals. The derivation rules for these judgments are presented in Figure 5, and are adapted from FLAM's inference rules [9].[8]These derivation rules are presented in terms of the acts-for ordering, but recall that since $\sqsubseteq$ is defined in terms of $\succcurlyeq$, it makes sense to use either of these symbols to represent a derivation. The rules are mostly straightforward except for R-ASSUME, which allows delegations from the context Π to be used in derivations. The premise $\Pi \Vdash \nabla(p^{\rightarrow}) \succcurlyeq \nabla(q^{\rightarrow})$ enforces a well-formedness invariant on Π that ensures that delegations of confidentiality are consistent with the ability to speak for those principals. This premise is related to FLAM's LIFT rule. The ASSUME typing rule ensures that all delegations added to Π satisfy this invariant, and initial delegations in Π that fail to satisfy it cannot be used in derivations.

We also use judgments of the form $\Pi \vdash p \sqsubseteq \tau$ to denote that type τ is at least as restrictive as the principal p. The derivation rules for these judgments are presented in Figure 8 and discussed in more detail below.

---

[8]FLAM's rules [9] also include *query* and *result* labels as part of the judgment context that represent the confidentiality and integrity of a FLAM query context and result, respectively. The FLAM delegation context also includes labels for delegations, whereas FLAC's delegation context does not. These labels are unnecessary in FLAC because we use FLAM judgments only in the type system—these "queries" only occur at compile time and do not create information flows about which delegations are in effect. We formalize the connection between FLAC and FLAM in Appendix A.

$$\boxed{\Pi; \Gamma; pc \vdash e : \tau}$$

[VAR] $\Pi; \Gamma, x : \tau, \Gamma'; pc \vdash x : \tau \quad x \notin \text{dom } \Gamma'$ [UNIT] $\Pi; \Gamma; pc \vdash () : \text{unit}$ [DEL] $\Pi; \Gamma; pc \vdash \langle p \geqslant q \rangle : (p \geqslant q)$

[LAM] $$\frac{\Pi; \Gamma, x : \tau_1; pc' \vdash e : \tau_2}{\Pi; \Gamma; pc \vdash \lambda(x : \tau_1)[pc']. e : (\tau_1 \xrightarrow{pc'} \tau_2)}$$ [TLAM] $$\frac{\Pi; \Gamma, X; pc' \vdash e : \tau}{\Pi; \Gamma; pc \vdash \Lambda X[pc']. e : \forall X[pc']. \tau}$$

[APP] $$\frac{\Pi; \Gamma; pc \vdash e : (\tau_1 \xrightarrow{pc'} \tau_2) \quad \Pi; \Gamma; pc \vdash e' : \tau_1 \quad \Pi \Vdash pc \sqsubseteq pc'}{\Pi; \Gamma; pc \vdash (e\, e') : \tau_2}$$ [TAPP] $$\frac{\Pi; \Gamma; pc \vdash e : \forall X[pc']. \tau \quad \Pi \Vdash pc \sqsubseteq pc'}{\Pi; \Gamma; pc \vdash (e\, \tau') : \tau[X \mapsto \tau']}$$ τ' well-formed in Γ

[PAIR] $$\frac{\Pi; \Gamma; pc \vdash e_1 : \tau_1 \quad \Pi; \Gamma; pc \vdash e_2 : \tau_2}{\Pi; \Gamma; pc \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$ [UNPAIR] $$\frac{\Pi; \Gamma; pc \vdash e : \tau_1 \times \tau_2}{\Pi; \Gamma; pc \vdash (\text{proj}_i\, e) : \tau_i}$$

[INJ] $$\frac{\Pi; \Gamma; pc \vdash e : \tau_i \quad i \in \{1, 2\}}{\Pi; \Gamma; pc \vdash (\text{inj}_i\, e) : \tau_1 + \tau_2}$$ [CASE] $$\frac{\Pi; \Gamma; pc \vdash e : \tau_1 + \tau_2 \quad \Pi \vdash pc \sqsubseteq \tau \quad \Pi; \Gamma, x : \tau_1; pc \vdash e_1 : \tau \quad \Pi; \Gamma, x : \tau_2; pc \vdash e_2 : \tau}{\Pi; \Gamma; pc \vdash \text{case } e \text{ of } \text{inj}_1(x).\, e_1 \mid \text{inj}_2(x).\, e_2 : \tau}$$

[UNITM] $$\frac{\Pi; \Gamma; pc \vdash e : \tau \quad \Pi \Vdash pc \sqsubseteq \ell}{\Pi; \Gamma; pc \vdash \eta_\ell\, e : \ell \text{ says } \tau}$$ [SEALED] $$\frac{\Pi; \Gamma; pc \vdash v : \tau}{\Pi; \Gamma; pc \vdash \overline{\eta}_\ell\, v : \ell \text{ says } \tau}$$

[BINDM] $$\frac{\Pi; \Gamma; pc \vdash e : \ell \text{ says } \tau' \quad \Pi; \Gamma, x : \tau'; pc \sqcup \ell \vdash e' : \tau \quad \Pi \vdash pc \sqcup \ell \sqsubseteq \tau}{\Pi; \Gamma; pc \vdash \text{bind } x = e \text{ in } e' : \tau}$$

[ASSUME] $$\frac{\Pi; \Gamma; pc \vdash e : (p \geqslant q) \quad \Pi \Vdash pc \geqslant \nabla(q) \quad \Pi \Vdash \nabla(p^{\rightarrow}) \geqslant \nabla(q^{\rightarrow}) \quad \Pi, \langle p \geqslant q \rangle; \Gamma; pc \vdash e' : \tau}{\Pi; \Gamma; pc \vdash \text{assume } e \text{ in } e' : \tau}$$

[WHERE] $$\frac{\Pi; \Gamma; pc \vdash v : (p \geqslant q) \quad \Pi \Vdash \overline{pc} \geqslant \nabla(q) \quad \Pi \Vdash \nabla(p^{\rightarrow}) \geqslant \nabla(q^{\rightarrow}) \quad \Pi, \langle p \geqslant q \rangle; \Gamma; pc \vdash e : \tau}{\Pi; \Gamma; pc \vdash (e \text{ where } v) : \tau}$$

Figure 4: FLAC type system.

[R-STATIC] $$\frac{\mathcal{L} \models p \geqslant q}{\Pi \Vdash p \geqslant q}$$ [R-ASSUME] $$\frac{\langle p \geqslant q \rangle \in \Pi \quad \Pi \Vdash \nabla(p^{\rightarrow}) \geqslant \nabla(q^{\rightarrow})}{\Pi \Vdash p \geqslant q}$$

[R-CONJL] $$\frac{\Pi \Vdash p_k \geqslant q \quad k \in \{1, 2\}}{\Pi \Vdash p_1 \wedge p_2 \geqslant q_2}$$ [R-CONJR] $$\frac{\Pi \Vdash p \geqslant q_1 \quad \Pi \Vdash p \geqslant q_2}{\Pi \Vdash p \geqslant q_1 \wedge q_2}$$

[R-DISJL] $$\frac{\Pi \Vdash p_1 \geqslant q \quad \Pi \Vdash p_2 \geqslant q}{\Pi \Vdash p_1 \vee p_2 \geqslant q}$$ [R-DISJR] $$\frac{\Pi \Vdash p \geqslant q_k \quad k \in \{1, 2\}}{\Pi \Vdash p \geqslant q_1 \vee q_2}$$

[R-TRANS] $$\frac{\Pi \Vdash p \geqslant q \quad \Pi \Vdash q \geqslant r}{\Pi \Vdash p \geqslant r}$$

Figure 5: Inference rules for robust assumption, derived from FLAM [9].

The DEL rule types delegation expressions as singletons: for each delegation type there is a unique delegation expression. Rules LAM and TLAM require the body of the abstraction to be well typed at the annotated *pc*. Rules APP and TAPP ensure functions may only be applied in contexts which are no more restrictive than the annotation. Rule TAPP additionally requires that τ' be well formed with respect to Γ. Specifically, τ' may not have any free type variables not bound by Γ.

Since FLAC is a pure functional language, it might seem odd for FLAC to have a label for the program counter; such labels are usually used to control implicit flows through assignments (e.g., in [46, 39]). The purpose of FLAC's *pc* label is to control a different kind of side effect: changes to the delegation context, Π.[9]

In order to control what information can influence the creation of a new trust relationship in a delegation context, the type system tracks the confidentiality and security of control flow. Viewed as an authorization logic, FLAC's type system has the unique feature that it expresses deduction constrained by an information flow context.[10] For instance, if we have $\varphi \xrightarrow{p^{\leftarrow}} \psi$ and φ, then (via APP) we may derive ψ in a context with integrity $p^{\leftarrow}$, but not in contexts that don't flow to $p^{\leftarrow}$. This feature offers needed control over how principals may apply existing facts to derive new facts.

Many FLAC terms are standard, such as pairs $\langle e_1, e_2 \rangle$, projections $\mathtt{proj}_i\ e$, variants $\mathtt{inj}_i\ e$, and case expressions. Function abstraction, $\lambda(x : \tau)[pc].\ e$ and polymorphic type abstraction, $\Lambda X[pc].\ e$, include a *pc label* that constrains the information flow context in which the function may be applied. The rule APP ensures that function application respects these policies, requiring that the robust FLAM judgment $\Pi \Vdash pc \sqsubseteq pc'$ holds. This judgment ensures that the current program counter label, *pc*, flows to the function label, *pc'*.

Branching occurs in case expressions, which conditionally evaluate one of two expressions. The rule CASE ensures that both expressions have the same type and thus the same protection level. The premise $\Pi \vdash pc \sqsubseteq \tau$ ensures that this type protects the current *pc* label.

Like DCC, FLAC uses monadic operators to track dependencies. The monadic unit term $\eta_\ell\ w$ (UNITM) says that a value w of type τ is *protected at level* ℓ. This protected value has the type $\ell\ \mathtt{says}\ \tau$, meaning that it has the confidentiality and integrity of principal ℓ. Because w could implicitly reveal information about the dependencies of the computation that produced it, UNITM requires that $\Pi \Vdash pc \sqsubseteq \ell$.[11] When a monadic term $\eta_\ell\ w$ steps to $\overline{\eta}_\ell\ w$ we call it *sealed* since all free values have been substituted and the expression will not capture any additional information from its context. Sealed terms type under the rule SEALED which is more permissive since the *pc* premise is unnecessary.

Computation on protected values must occur in a protected context ("in the monad"), expressed using a monadic bind term. The typing rule BINDM ensures that the result of the computation protects the confidentiality and integrity of protected values. For instance, the expression $\mathtt{bind}\ x = \eta_\ell\ v\ \mathtt{in}\ \eta_{\ell'}\ x$ is only well-typed if ℓ' protects values with confidentiality and integrity ℓ. Since case expressions may use the variable x for branching, BINDM raises the *pc* label to $pc \sqcup \ell$ to conservatively reflect the control-flow dependency.

Protection levels are defined by the set of inference rules in Figure 8, adapted from [52]. Expressions with unit type (P-UNIT) do not propagate any information, so they protect information at any ℓ. Product types protect information at ℓ if both components do (P-PAIR). Function types protect information at ℓ if the return type and function label does (P-FUN), and polymorphic types protect information at whatever level the abstracted type and type function label does (P-TFUN). Finally, if ℓ flows to ℓ', then $\ell'\ \mathtt{says}\ \tau$ protects information at ℓ (P-LBL).[12] There are no protection rules for sum types or type variables since they do not protect information: inspecting the constructor of a sum type value reveals information, and type variables may be instantiated with types that offer different levels of protection or none at all. Because delegation expressions are singletons, a protection rules for $(p \succcurlyeq q)$ types similar to the protection rule for $\mathtt{unit}$ would in principle be admissible, but our examples and results did not require this permissiveness, and we have not explored its consequences.

Occasionally it is more convenient to write protection relations in terms of only confidentiality or integrity, so we also define a notation for authority projections on types in Figure 6.

---

[9]DFLATE [26], an extension of FLAC for modeling distributed applications with Trusted Execution Environments, uses the same *pc* label to control implicit flows due to communication side-effects. Extensions of FLAC to support mutable references or other effects could control implicit flows similarly.

[10]FLAFOL [29] further develops the idea of constraining logical deduction with information flow constraints in a first-order logic.

[11]Neither DCC [4] nor the original FLAC formalization [7] included this premise. DCC does not maintain a *pc* label at all. FLAC originally used a version of the DCC rule, but Cecchetti et al. [18] and Gollamudi et al. [26] added the *pc* restriction in support of a non-commutative $\mathtt{says}$. See Section 5.1 for additional details.

[12]DCC [4] and the original FLAC formalization included an additional protection rule that considered ℓ to be protected by $\ell'\ \mathtt{says}\ \tau$ if τ protects ℓ (even if ℓ' does not). This rule was removed by Cecchetti et al. and Gollamudi et al. [26] to make $\mathtt{says}$ non-commutative. Some variants of DCC [3] treat the $\mathtt{says}$ modality similarly.

$$(\tau \xrightarrow{pc} \tau)^\pi = (\tau \xrightarrow{pc^\pi} \tau^\pi)$$
$$(\ell \text{ says } \tau)^\pi = \ell^\pi \text{ says } \tau$$
$$(\tau \times \tau)^\pi = (\tau^\pi \times \tau^\pi)$$
$$(\forall X[pc].\,\tau)^\pi = (\forall X[pc^\pi].\,\tau^\pi)$$
$$\text{otherwise } \tau^\pi = \tau$$

Figure 6: Authority projections on types

Proposition 1.

$$\Pi \vdash \ell^\pi \sqsubseteq \tau^\pi \Leftrightarrow \begin{cases} \Pi \vdash \ell^\rightarrow \wedge \top^\leftarrow \sqsubseteq \tau & \text{for } \pi = \rightarrow \\ \Pi \vdash \ell^\leftarrow \sqsubseteq \tau & \text{for } \pi = \leftarrow \end{cases}$$

Proof. In the forward direction, by induction on the structure of τ. In the reverse direction, by induction on the derivation of $\Pi \vdash \ell^\rightarrow \wedge \top^\leftarrow \sqsubseteq \tau$ (for $\pi = \rightarrow$) and $\Pi \vdash \ell^\leftarrow \sqsubseteq \tau$ (for $\pi = \leftarrow$). □

Most of the novelty of FLAC lies in its delegation values and assume terms. These terms enable expressive reasoning about authority and information flow control. A delegation value serves as evidence of trust. For instance, the term $\langle p \geqslant q \rangle$, read "$p$ acts for q", is evidence that q trusts p. Delegation values have *acts-for types*; $\langle p \geqslant q \rangle$ has type $(p \geqslant q)$. [13] The assume term enables programs to use evidence securely to create new flows between protection levels. In the typing context $\varnothing; x : p^\leftarrow \text{ says } \tau; q^\leftarrow$ (i.e., $\Pi = \varnothing$, $\Gamma = x : p^\leftarrow \text{ says } \tau$, and $pc = q^\leftarrow$), the following expression is not well typed:

$$\text{bind } x' = x \text{ in } (\eta_{q^\leftarrow} x')$$

since $p^\leftarrow$ does not flow to $q^\leftarrow$, as required by the premise $\Pi \vdash \ell \sqsubseteq \tau$ in rule BINDM. Specifically, we cannot derive $\Pi \vdash p^\leftarrow \sqsubseteq q^\leftarrow \text{ says } \tau$ since P-LBL requires the FLAM judgment $\Pi \Vdash p^\leftarrow \sqsubseteq q^\leftarrow$ to hold.

However, the following expression is well typed:

$$\text{assume } \langle p^\leftarrow \geqslant q^\leftarrow \rangle \text{ in bind } x' = x \text{ in } (\eta_{q^\leftarrow} x')$$

The difference is that the assume term adds a trust relationship, represented by an expression with an acts-for type, to the delegation context. In this case, the expression $\langle p^\leftarrow \geqslant q^\leftarrow \rangle$ adds a trust relationship that allows $p^\leftarrow$ to flow to $q^\leftarrow$. This is secure since $pc = q^\leftarrow$, meaning that only principals with integrity $q^\leftarrow$ have influenced the computation. With $\langle p^\leftarrow \geqslant q^\leftarrow \rangle$ in the delegation context, added via the ASSUME rule, the premises of BINDM are now satisfied, so the expression type-checks.

Creating a delegation value requires no special privilege because the type system ensures only high-integrity delegations are used as evidence for enabling new flows. Using low-integrity evidence for authorization would be insecure since attackers could use delegation values to create new flows that reveal secrets or corrupt data. The premises of the ASSUME rule ensure the integrity of dynamic authorization computations that produce values like $\langle p^\leftarrow \geqslant q^\leftarrow \rangle$ in the example above.[14] The second premise, $\Pi \Vdash pc \geqslant \nabla(q)$, requires that the pc has enough integrity to be trusted by q, the principal whose security is affected. For instance, to make the assumption $p \geqslant q$, the evidence represented by the term e must have at least the integrity of the voice of q, written $\nabla(q)$. Since the pc bounds the restrictiveness of the dependencies of e, this ensures that only information with integrity $\nabla(q)$ or higher may influence the evaluation of e. The third premise, $\Pi \Vdash \nabla(p^\rightarrow) \geqslant \nabla(q^\rightarrow)$, ensures that principal p has sufficient integrity to be trusted to enforce q's confidentiality, $q^\rightarrow$. This premise means that q permits data to be relabeled from $q^\rightarrow$ to $p^\rightarrow$.[15]

The pc constraints on function application ensure that functions containing assume terms can only be applied in high-integrity contexts. For example, the following function declassifies one of Alice's secrets to Bob. The first assume establishes that Bob is trusted to speak for Alice, and the second delegates Alice's confidentiality authority to Bob.

---

[13]This correspondence with delegation values makes acts-for types a kind of singleton type [21].

[14]These premises are related to the robust FLAM rule LIFT.

[15]More precisely, it means that the voice of q's confidentiality, $\nabla(q^\rightarrow)$, permits data to be relabeled from $q^\rightarrow$ to $p^\rightarrow$. Recall that $\nabla(\text{Alice}^\rightarrow)$ is just Alice's integrity projection: $\text{Alice}^\leftarrow$.

$$\boxed{e \longrightarrow e'}$$

[W-APP] $(w \text{ where } v)\, e \longrightarrow (w\, e) \text{ where } v$ [W-TAPP] $(w \text{ where } v)\, \tau \longrightarrow (w\, \tau) \text{ where } v$

[W-UNPAIR] $\text{proj}_i\, (w \text{ where } v) \longrightarrow (\text{proj}_i\, w) \text{ where } v$

[W-CASE] $(\text{case } (w \text{ where } v) \text{ of } \text{inj}_1(x).\, e_1 \mid \text{inj}_2(x).\, e_2) \longrightarrow (\text{case } w \text{ of } \text{inj}_1(x).\, e_1 \mid \text{inj}_2(x).\, e_2) \text{ where } v$

[W-BINDM] $\text{bind } x = (w \text{ where } v) \text{ in } e \longrightarrow (\text{bind } x = w \text{ in } e) \text{ where } v$

[W-ASSUME] $\text{assume } (w \text{ where } v) \text{ in } e \longrightarrow (\text{assume } w \text{ in } e) \text{ where } v$

Figure 7: Propagation of where terms

The bind term then relabels Alice's secret to a label observable by Bob.

$$\text{declassify} :: (\text{Alice}^{\rightarrow} \text{ says } \tau) \xrightarrow{pc} (\text{Bob}^{\rightarrow} \text{ says } \tau)$$
$$\text{declassify} = \lambda(x{:}\text{Alice}^{\rightarrow} \text{ says } \tau)[pc].$$
$$\text{assume } \langle \text{Bob}^{\leftarrow} \geqslant \text{Alice}^{\leftarrow} \rangle \text{ in}$$
$$\text{assume } \langle \text{Bob}^{\rightarrow} \geqslant \text{Alice}^{\rightarrow} \rangle \text{ in}$$
$$\text{bind } x' = x \text{ in } (\eta_{\text{Bob}^{\rightarrow}}\, x')$$

If Bob could apply this function arbitrarily, then he could declassify all of Alice's secrets—not just the ones she intended to release. However, since the assume terms delegate Alice's confidentiality and integrity authority, this function is only well typed if pc speaks for Alice, or $\Pi \Vdash pc \geqslant \text{Alice}^{\leftarrow}$. Otherwise the assume terms are rejected by the type system. The constraints in the APP rule then ensure that this function can only be applied in a context that flows to pc.

Assumption terms evaluate to where expressions (rule E-ASSUME). These expressions are a purely formal book-keeping mechanism (i.e., they would be unnecessary in a FLAC-based implementation) to ensure that source-level terms that were well-typed because of an assume term remain well-typed during evaluation. This helps us distinguish insecure FLAC terms from terms whose policies have been legitimately downgraded. These where terms record and maintain the authorization evidence used to justify new flows of information during evaluation. They are not part of the source language and generated only by the evaluation rules. The term e where $\langle p \geqslant q \rangle$ records that e is typed in a context that includes the delegation $\langle p \geqslant q \rangle$.

The rule WHERE gives a typing rule for where terms; though similar to ASSUME, it requires only that $\nabla(q)$ delegate to the distinguished label $\overline{pc}$, which is a fixed parameter of the type system. The use of $\overline{pc}$ is purely technical: our proofs in Section 6 use $\overline{pc}$ to help reason about what new flows may have created by assume terms. The only requirement is that $\overline{pc}$ be as trusted as the pc label used to type-check the source program (or programs) that generated the where term. Since the pc increases monotonically when typing subexpressions, In our formal results, we choose $\overline{pc}$ to be $\top^{\leftarrow}$ since it is always valid. Selecting a more restrictive label could offer finer-grained reasoning about what downgrades may occur in non-source-level terms since it restricts which where-terms are well-typed.

Figure 7 presents evaluation rules for where terms. The rules are designed to treat where values like the value they enclose. For instance, applying a where term (rule W-APP) simply moves the value it is applied to inside the where term. If the where term was wrapping a lambda expression, then it may now be applied via APP. Otherwise, further reduction steps via W-APP may be necessary. We use the syntactic category w (see Figure 2) to refer to fully-evaluated where terms, or where *values*. In other words, a where value w is an expression consisting of a value v enclosed by one or more where clauses. A where value usually behaves like a value, but it is occasionally convenient to distinguish them.[16]

---

[16]The original FLAC formalization did not distinguish where values and values, and did not include the rules in Figure 7. Unfortunately, this resulted in stuck terms when where terms were not propagated appropriately. We have proven the above rules eliminate stuck terms for well-typed programs (Lemma 5).

$$\boxed{\Pi \vdash \ell \sqsubseteq \tau}$$

[P-UNIT] $\Pi \vdash \ell \sqsubseteq \mathsf{unit}$

[P-PAIR] $\dfrac{\Pi \vdash \ell \sqsubseteq \tau_1 \qquad \Pi \vdash \ell \sqsubseteq \tau_2}{\Pi \vdash \ell \sqsubseteq \tau_1 \times \tau_2}$

[P-FUN] $\dfrac{\Pi \vdash \ell \sqsubseteq \tau_2 \qquad \Pi \vdash \ell \sqsubseteq pc'}{\Pi \vdash \ell \sqsubseteq \tau_1 \xrightarrow{pc'} \tau_2}$

[P-TFUN] $\dfrac{\Pi \vdash \ell \sqsubseteq \tau \qquad \Pi \vdash \ell \sqsubseteq pc'}{\Pi \vdash \ell \sqsubseteq \forall X[pc'].\,\tau}$

[P-LBL] $\dfrac{\Pi \Vdash \ell \sqsubseteq \ell'}{\Pi \vdash \ell \sqsubseteq \ell' \text{ says } \tau}$

Figure 8: Type protection levels

5 FLAC Proof theory

5.1 Properties of `says`

FLAC's type system constrains how principals apply existing facts to derive new facts. For instance, a property of `says` in other authorization logics (e.g., Lampson et al. [33] and Abadi [2]) is that implications that hold for top-level propositions also hold for propositions of any principal ℓ:

$$\vdash (\tau_1 \to \tau_2) \to (\ell \text{ says } \tau_1 \to \ell \text{ says } \tau_2)$$

The *pc* annotations on FLAC function types refine this property. Each implication (in other words, each function) in FLAC is annotated with an upper bound on the information flow context it may be invoked within. To lift such an implication to operate on propositions protected at label ℓ, the label ℓ must flow to the *pc* of the implication. Thus, for all ℓ and τ_i,

$$\vdash (\tau_1 \xrightarrow{\ell} \tau_2) \xrightarrow{\ell} (\ell \text{ says } \tau_1 \xrightarrow{\ell} \ell \text{ says } \tau_2)$$

This judgment is a FLAC typing judgment in *logical form*, where terms have been omitted. We write such judgments with an empty typing context (as above) when the judgment is valid for any Π, Γ, and *pc*. A judgment in logical form is valid if a *proof term* exists for the specified type, proving the type is inhabited. The above type has proof term

$$\lambda(f\!:\!(\tau_1 \xrightarrow{\ell} \tau_2))[\ell].$$
$$\lambda(x\!:\!\ell \text{ says } \tau_1)[\ell].\,\mathsf{bind}\ x' = x\ \mathsf{in}\ \eta_\ell\ f\ x'$$

In order to apply f, we must first bind x, so according to rules BINDM and APP, the function f must have a label at least as restrictive as ℓ, and UNITM requires the label of the returned value must also be as restrictive as ℓ. We can actually prove a slightly more general version of the above theorem:

$$(\tau_1 \xrightarrow{pc \sqcup \ell} \tau_2) \xrightarrow{\ell} (\ell \text{ says } \tau_1 \xrightarrow{pc} pc \sqcup \ell \text{ says } \tau_2)$$

This version permits using the implications in more restrictive contexts, but doesn't map as well to a DCC theorem since the principal of the return type differs from the argument type.

These refinements of DCC's theorems are crucial for supporting applications like commitment schemes and bearer credentials. Our FLAC implementations, presented in detail in Sections 7.1 and 7.2, rely in part on restricting the *pc* to a specific principal's integrity. Without such refinements, principal q could open principal p's commitments using open, or create credentials with p's authority: $p^\to \xRightarrow{pc} p^\leftarrow$. With these refinements, we can express privileged implications (functions) that only trusted principals may apply.

Consider a DCC version of the `declassify` function type from Section 4:

$$\texttt{dcc\_declassify} :: (\text{Alice says } \tau) \to (\text{Bob says } \tau)$$

In DCC, functions are not annotated with *pc* labels and may be applied in any context. Therefore, *any* principal could use `dcc_declassify` to relabel Alice's information to Bob—including Bob.

Other properties of `says` common to DCC and other logics (cf. [1] for examples) are similarly refined by *pc* bounds. Two examples are: $\vdash \tau \xrightarrow{\ell} \ell \text{ says } \tau$ which has proof term: $\lambda(x\!:\!\tau)[\ell].\,\eta_\ell\ \tau$ and

$$\vdash \ell \text{ says } (\tau_1 \xrightarrow{\ell} \tau_2) \xrightarrow{\ell} (\ell \text{ says } \tau_1 \xrightarrow{\ell} \ell \text{ says } \tau_2)$$

with proof term:

$$\lambda(f : \ell \text{ says } (\tau_1 \xrightarrow{\ell} \tau_2))[\ell].\, \texttt{bind } f' = f \texttt{ in}$$
$$\lambda(y : \ell \text{ says } \tau_1)[\ell].\, \texttt{bind } y' = y \texttt{ in } \eta_\ell\, f'\, y'$$

Some theorems of DCC cannot be obtained in FLAC, due to the *pc* restriction on UNITM as well as the more restrictive protection relation. For example, chains of says are not commutative in FLAC. Given ℓ_1, ℓ_2, and *pc*,

$$\not\vdash \ell_1 \text{ says } \ell_2 \text{ says } \tau \xrightarrow{pc} \ell_2 \text{ says } \ell_1 \text{ says } \tau$$

unless $\Pi \Vdash \ell_1 \sqcup pc \sqsubseteq \ell_2$ and $\Pi \Vdash \ell_2 \sqcup pc \sqsubseteq \ell_1$, which implies ℓ_1, ℓ_2, and *pc* must be equivalent in Π. CCD [3], a logic related to DCC, is also non-commutative with respect to says, but does not have an associated term language.

Distinguishing the nesting order of says types is attractive for authorization settings since it encodes the provenance of statements. It also enables modeling of cryptographic mechanisms in FLAC (*cf.* [26]) where ℓ says τ is interpreted as a value of type τ protected by encryption key $\ell^\rightarrow$ and signing key $\ell^\leftarrow$. Preserving the order of nested types in this context is useful for modeling decryption and verification of protected values.

5.2 Dynamic Hand-off

Many authorization logics support delegation using a "hand-off" axiom. In DCC, this axiom is a provable theorem:

$$\vdash (q \text{ says } (p \Rightarrow q)) \rightarrow (p \Rightarrow q)$$

where $p \Rightarrow q$ is shorthand for

$$\forall X.\, (p \text{ says } X \rightarrow q \text{ says } X)$$

However, $p \Rightarrow q$ is only inhabited if $p \sqsubseteq q$ is derivable in the security lattice. Thus, DCC can reason about the consequences of an assumption that $p \sqsubseteq q$ holds (whether it is true for the lattice or not), but a DCC program cannot produce a term of type $p \Rightarrow q$ unless $p \sqsubseteq q$.

FLAC programs, on the other hand, can create new trust relationships from delegation expressions using assume terms. The type analogous to $p \Rightarrow q$ in FLAC is

$$\forall X[pc].\, (p \text{ says } X \xrightarrow{pc} q \text{ says } X)$$

which we write as $p \xRightarrow{pc} q$. FLAC programs construct terms of this type from proofs of authority, represented by terms with acts-for types. This feature enables a more general form of hand-off, which we state formally below.

Proposition 2 (Dynamic hand-off). *For all ℓ and pc', let $pc = \ell^\rightarrow \wedge \nabla(p^\rightarrow) \wedge q^\leftarrow \wedge \nabla(pc')$*

$$(\nabla(q^\rightarrow) \succcurlyeq \nabla(p^\rightarrow)) \xrightarrow{pc} (p \sqsubseteq q) \xrightarrow{pc} (pc' \sqsubseteq q) \xrightarrow{pc}$$
$$\forall X[pc'].\, (p \text{ says } X \xrightarrow{pc'} q \text{ says } X)$$

Proof term.

$$\lambda(pf_1 : (\nabla(q^\rightarrow) \succcurlyeq \nabla(p^\rightarrow)))[pc].\, \texttt{assume } pf_1 \texttt{ in}$$
$$\lambda(pf_2 : (p \sqsubseteq q))[pc].\, \texttt{assume } pf_2 \texttt{ in}$$
$$\lambda(pf_3 : (pc' \sqsubseteq q))[pc].\, \texttt{assume } pf_3 \texttt{ in}$$
$$\Lambda X[pc'].\, \lambda(x : p \text{ says } X)[pc'].\, \texttt{bind } x' = x \texttt{ in } \eta_q\, x'$$

The principal $pc = \ell^\rightarrow \wedge \nabla(p^\rightarrow) \wedge q^\leftarrow \wedge \nabla(pc')$ restricts delegation (hand-off) to contexts with sufficient integrity to authorize the delegations made by the assume terms. In other words, the context that creates these delegations must be authorized by the combined authority of $\nabla(p^\rightarrow)$, $q^\leftarrow$, and $\nabla(pc')$.

The three arguments are proofs of authority with acts-for types: a proof of $\nabla(q^\rightarrow) \succcurlyeq \nabla(p^\rightarrow)$, a proof of $p \sqsubseteq q$, and a proof of $pc' \sqsubseteq q$. The *pc* ensures that the proofs have sufficient integrity to be used in assume terms since it has the integrity of both $\nabla(p^\rightarrow)$ and $q^\leftarrow$. Note that low-integrity or confidential delegation values must first be bound via bind before the above term may be applied. Thus the *pc* would reflect the protection level of both arguments. Principals $\ell^\rightarrow$ and pc' are unconstrained, but the third proof argument ensures that flows from pc' to q are authorized, since a principal with access to q's secrets could infer something about the context (protected at pc') in which the hand-off function is applied. Other dynamic hand-off formulations are possible, Proposition 2 simply has the fewest

13

assumptions. Other formulations can eliminate the need for proofs pf_1, pf_2, and/or pf_3 if these relationships already exist in the context defining the hand-off.

Dynamic hand-off terms give FLAC programs a level of expressiveness and security not offered by other authorization logics. Observe that pc' may be chosen independently of the other principals. This means that although the pc prevents low-integrity principals from creating hand-off terms, a high-integrity principal may create a hand-off term and provide it to an arbitrary principal. Hand-off terms in FLAC, then, are similar to capabilities since even untrusted principals may use them to change the protection level of values. Unlike in most capability systems, however, the propagation of hand-off terms can be constrained using information flow policies.

Terms that have types of the form in Proposition 2 illustrate a subtlety of enforcing information flow in an authorization mechanism. Because these terms relabel information from one protection level to another protection level, the transformed information implicitly depends on the proofs of authorization. FLAC ensures that the information security of these proofs is protected—like that of all other values—even as the policies of other information are being modified. Hence, authorization proofs cannot be used as a side channel to leak information.

For example, if `Alice`'s trust relationship with `Bob` is secret, she might protect it at confidentiality $\texttt{Alice}^{\rightarrow}$. If `Alice` wants to delegate trust to `Bob` using an approach like Proposition 2, the delegation protected at $\texttt{Alice}^{\rightarrow}$ would first have to be bound:

$$\texttt{bind } d = (\eta_{\texttt{Alice}} \langle \texttt{Bob} \geqslant \texttt{Alice} \rangle) \texttt{ in } ...$$

This would imply (by the BINDM typing rule) that in order to create a term with type $\texttt{Bob} \xLongrightarrow{pc} \texttt{Alice}$ in the body of the bind, it must be the case that $\Pi \vdash \texttt{Alice} \sqsubseteq (\texttt{Alice} \xLongrightarrow{pc} \texttt{Bob})$, which in turn requires that $\Pi \Vdash \texttt{Alice} \sqsubseteq pc$ and $\Pi \Vdash \texttt{Alice} \sqsubseteq \texttt{Bob}$.

Of course, if $\Pi \Vdash \texttt{Alice} \sqsubseteq \texttt{Bob}$ already holds, then there is no need to delegate confidentiality authority to `Bob`. Therefore, a typical approach would first declassify the delegation to `Bob` (e.g., relabel it from $\texttt{Alice says } (\texttt{Bob} \geqslant \texttt{Alice})$ to $\texttt{Bob}^{\rightarrow} \wedge \texttt{Alice}{\leftarrow} \texttt{ says } (\texttt{Bob} \geqslant \texttt{Alice}))$, before handing off authority. This requirement ensures that the disclosure of the secret trust relationship is intentional, and is an example of a more general principle in FLAC: that restricted terms cannot be used to "downgrade themselves." We formalize this idea in Section 6.3 with the Delegation Compartmentalization lemma.

6 Semantic security properties of FLAC

In this section, we formalize our semantic security guarantees. Our results are based on a bracketed semantics in the style of Pottier and Simonet [46] extended to a trace-based semantic model. The observability of trace elements is defined by an erasure function and our noninterference and robust declassification theorems are stated in terms of indistinguishability on traces. Because FLAC supports downgrading via assume, the usual bracketed semantic approach to noninterference via type preservation is insufficient, so we develop an approach to characterize what downgrades are possible in well-typed FLAC programs. Finally, with the necessary infrastructure in place, we state and prove our noninterference and robust declassification theorems.

6.1 Trace indistinguishability

We express our semantic security results in terms of the *traces* of a program observable to an attacker. FLAC traces are simply the sequence of terms under the $\longrightarrow$ relation. That is, each evaluation step of the form $e \longrightarrow e'$ generates a new trace element e'. We write the trace generated by taking n steps from e to e' as $e \xrightarrow{t}{}^{*} e'$, where $t[0] = e$ and $t[n] = e'$.

FLAC traces are not fully observable to an attacker. Trace elements generated by protected information such as sealed values or protected contexts such as within a bind or lambda term are hidden from the attacker. This approach models scenarios where an attacker has a limited ability to observe program values, for example when sealed values are protected by a trustworthy a runtime or are signed and encrypted. For these scenarios, the UNITM typing rule and an observability function model how computation on sealed values, and thus the observability of intermediate values, is limited to principals with necessary permissions or cryptographic keys to access inputs and produce outputs.[17] An insecure program in this model allows an attacker to learn information by allowing protected information to flow to sealed values or contexts that are observable by the attacker.

---

[17]DFLATE [26] explores this connection with cryptographic enforcement more explicitly in a distributed extension of FLAC.

Syntax

$$e ::= \dots \mid (\!| \, e \, |\!)_\ell$$

Evaluation contexts

$$E ::= \dots \mid (\!| \, E \, |\!)_\ell$$

Evaluation rules

[E-APP*] $(\lambda(x\!:\!\tau)[pc].\, e)\, w \longrightarrow (\!| \, e[x \mapsto w] \, |\!)_{pc}$ [E-TAPP*] $(\Lambda X[pc].\, e)\, \tau \longrightarrow (\!| \, e[X \mapsto \tau] \, |\!)_{pc}$

[E-BINDM*] $\texttt{bind}\; x = \overline{\eta}_\ell\, w \;\texttt{in}\; e \longrightarrow (\!| \, e[x \mapsto w] \, |\!)_\ell$ [O-CTX] $(\!| \, w \, |\!)_\ell \longrightarrow w$

Typing rules

[CTX]
$$\frac{\Pi; \Gamma; pc' \vdash e : \tau \qquad \Pi \Vdash pc \sqsubseteq pc'}{\Pi; \Gamma; pc \vdash (\!| \, e \, |\!)_{pc'} : \tau}$$

Figure 9: Extensions to support protection contexts.

Before formally defining what portions of program trace are available to an attack, we must first add some additional bookkeeping to our semantic rules. Specifically, we need additional notation for evaluating some intermediate expressions. The notation $(\!| \, e \, |\!)_\ell$ denotes an intermediate expression e evaluated in a context protected at ℓ.

Figure 9 presents syntax and evaluation rules for introducing and eliminating protected contexts. Rules E-APP*, E-TAPP*, and E-BINDM* replace their counterparts in Figure 3 with rules that introduce a protected context based on the relevant label. Rule O-CTX eliminates the protected context when the intermediate expression is fully evaluated. The typing rule CTX ensures that expressions inside protected contexts are well-typed at the annotated label, and that the pc flows to the annotated label.

The portion of a FLAC program observable to an attacker is formally defined by an observation function, $\mathcal{O}(e, \Pi, p, \pi)$, defined in Figure 10. An expression e is observable by principal p in delegation context Π depending on the authority of p relative to the protected terms in e. The projection π specifies whether to consider the confidentiality of protected terms ($\pi = \rightarrow$) or the integrity ($\pi = \leftarrow$). For example, sealed values such as $\overline{\eta}_\ell\, w$ are observable by principal p if p acts for ℓ, otherwise they are erased. Protected contexts like $(\!| \, e \, |\!)_\ell$ are treated similarly. To simplify our proofs, we collapse terms whose subterms have been erased. For example $\langle \circ, \circ \rangle$ is collapsed to $\circ$.

Erasing delegations in the context of where terms essentially hides the delegations that justify a value's flow from the attacker. This is consistent with the idea that attackers may learn *implicit* information from flows that introduce new delegations. In other words, by observing differences in outputs the attacker may infer the existence of secret delegations, but these delegations are not *explicit* in the output. By contrast, these delegations are explicit in the output of DFLATE [26] programs, modeling values that carry certified justifications of why their flow was authorized by the program.

The observability of trace elements is defined in Figure 10 in terms of the observability function $\mathcal{O}$ for FLAC terms. We also lift the observability of trace elements to traces in a natural way. Note that duplicate entries (which may occur due to evaluation in protected contexts) are removed. Deduplication avoids unintentional sensitivity to the number of steps taken in unobservable contexts.

We now can define *trace indistinguishability*. Two traces t and t' are indistinguishable to a principal p^π in delegation context Π if the observable elements of t and t' are equal.

Definition 1 (Trace indistinguishability)**.**

$$t \approx^{\Pi}_{p^\pi} t' \quad \overset{\Delta}{\iff} \quad \mathcal{O}(t, \Pi, p, \pi) = \mathcal{O}(t', \Pi, p, \pi)$$

15

$$\mathcal{O}(x, \Pi, p, \pi) \quad = \quad x$$

$$\mathcal{O}(\langle a \succcurlyeq b \rangle, \Pi, p, \pi) \quad = \quad \langle a \succcurlyeq b \rangle$$

$$\mathcal{O}((), \Pi, p, \pi) \quad = \quad ()$$

$$\mathcal{O}(\eta_\ell\, e, \Pi, p, \pi) \quad = \quad \eta_\ell\, \mathcal{O}(e, \Pi, p, \pi)$$

$$\mathcal{O}(\bar{\eta}_\ell\, w, \Pi, p, \pi) \quad = \quad \begin{cases} \bar{\eta}_\ell\, \mathcal{O}(w, \Pi, p, \pi) & \text{if } \Pi \Vdash \ell^\pi \sqsubseteq p^\pi \\ \circ & \text{otherwise} \end{cases}$$

$$\mathcal{O}(\lambda(x{:}\tau)[pc].\, e, \Pi, p, \pi) \quad = \quad \begin{cases} \lambda(x{:}\tau)[pc].\, \mathcal{O}(e, \Pi, p, \pi) & \text{if } \Pi \Vdash pc^\pi \sqsubseteq p^\pi \\ \circ & \text{otherwise} \end{cases}$$

$$\mathcal{O}(\Lambda X[pc].\, e, \Pi, p, \pi) \quad = \quad \begin{cases} \Lambda X[pc].\, \mathcal{O}(e, \Pi, p, \pi) & \text{if } \Pi \Vdash pc^\pi \sqsubseteq p^\pi \\ \circ & \text{otherwise} \end{cases}$$

$$\mathcal{O}((\!|\, e\, |\!)_\ell, \Pi, p, \pi) \quad = \quad \begin{cases} (\!|\, \mathcal{O}(e, \Pi, p, \pi)\, |\!)_\ell & \text{if } \Pi \Vdash \ell^\pi \sqsubseteq p^\pi \\ \circ & \text{otherwise} \end{cases}$$

$$\mathcal{O}(e_1\, e_2, \Pi, p, \pi) \quad = \quad \begin{cases} \circ & \text{if } \mathcal{O}(e_i, \Pi, p, \pi) = \circ \\ \mathcal{O}(e_1, \Pi, p, \pi)\, \mathcal{O}(e_2, \Pi, p, \pi) & \text{otherwise} \end{cases}$$

$$\mathcal{O}(\langle e_1, e_2 \rangle, \Pi, p, \pi) \quad = \quad \begin{cases} \circ & \text{if } \mathcal{O}(e_i, \Pi, p, \pi) = \circ \\ \langle \mathcal{O}(e_1, \Pi, p, \pi), \mathcal{O}(e_2, \Pi, p, \pi) \rangle & \text{otherwise} \end{cases}$$

$$\mathcal{O}(\mathsf{proj}_i\, e, \Pi, p, \pi) \quad = \quad \mathsf{proj}_i\, \mathcal{O}(e, \Pi, p, \pi)$$

$$\mathcal{O}(\mathsf{inj}_i\, e, \Pi, p, \pi) \quad = \quad \begin{cases} \circ & \text{if } \mathcal{O}(e, \Pi, p, \pi) = \circ \\ \mathsf{inj}_i\, \mathcal{O}(e, \Pi, p, \pi) & \text{otherwise} \end{cases}$$

$$\mathcal{O}(\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inj}_1(x).\, e_1 \mid \mathsf{inj}_2(x).\, e_2, \Pi, p, \pi) \quad = \quad \begin{aligned}[t] &\mathsf{case}\ \mathcal{O}(e, \Pi, p, \pi)\ \mathsf{of} \\ &\quad \mathsf{inj}_1(x).\, \mathcal{O}(e_1, \Pi, p, \pi) \\ &\quad \mid \mathsf{inj}_2(x).\, \mathcal{O}(e_2, \Pi, p, \pi) \end{aligned}$$

$$\mathcal{O}(\mathsf{bind}\ x = e\ \mathsf{in}\ e', \Pi, p, \pi) \quad = \quad \mathsf{bind}\ x = \mathcal{O}(e, \Pi, p, \pi)\ \mathsf{in}\ \mathcal{O}(e', \Pi, p, \pi)$$

$$\mathcal{O}(\mathsf{assume}\ e\ \mathsf{in}\ e', \Pi, p, \pi) \quad = \quad \begin{cases} \circ & \text{if } \mathcal{O}(e, \Pi, p, \pi) = \circ \text{ and} \\ & \quad \mathcal{O}(e', \Pi, p, \pi) = \circ \\ \mathsf{assume}\ \mathcal{O}(e, \Pi, p, \pi)\ \mathsf{in}\ \mathcal{O}(e', \Pi, p, \pi) & \text{otherwise} \end{cases}$$

$$\mathcal{O}(e\ \mathsf{where}\ v, \Pi, p, \pi) \quad = \quad \mathcal{O}(e, \Pi, p, \pi)$$

(a) Observation function for intermediate FLAC terms.

$$\mathcal{O}([e], \Pi, p, \pi) \quad = \quad [\mathcal{O}(e, \Pi, p, \pi)]$$

$$\mathcal{O}([e; e'], \Pi, p, \pi) \quad = \quad \begin{cases} [\mathcal{O}(e, \Pi, p, \pi)] & \text{if } \mathcal{O}(e, \Pi, p, \pi) = \mathcal{O}(e', \Pi, p, \pi) \\ [\mathcal{O}(e, \Pi, p, \pi); \mathcal{O}(e', \Pi, p, \pi)] & \text{otherwise} \end{cases}$$

$$\mathcal{O}([e; e'] \cdot t, \Pi, p, \pi) \quad = \quad \mathcal{O}(\mathcal{O}([e; e'], \Pi, p, \pi) \cdot t, \Pi, p, \pi)$$

(b) Observation function for traces.

Figure 10: Observation function definitions

Syntax

$$w \quad ::= \quad \dots \mid (w \mid w)$$
$$e \quad ::= \quad \dots \mid (e \mid e)$$

Evaluation rules

[B-STEP] $\quad \dfrac{e_i \longrightarrow e_i' \qquad e_j' = e_j \qquad \{i,j\} = \{1,2\}}{(e_1 \mid e_2) \longrightarrow (e_1' \mid e_2')}$ [B-APP] $\quad (w_1 \mid w_2)\, w \longrightarrow (w_1 \lfloor w \rfloor_1 \mid w_2 \lfloor w \rfloor_2)$

[B-TAPP] $\quad (w \mid w')\, \tau \longrightarrow (w\, \tau \mid w'\, \tau)$ [B-UNPAIR] $\quad \mathsf{proj}_i\, (\langle w_{11}, w_{12} \rangle \mid \langle w_{21}, w_{22} \rangle) \longrightarrow (w_{1i} \mid w_{2i})$

[B-BINDM] $\qquad \mathsf{bind}\, x = (w \mid w')\, \mathsf{in}\, e \longrightarrow (\mathsf{bind}\, x = w\, \mathsf{in}\, \lfloor e \rfloor_1 \mid \mathsf{bind}\, x = w'\, \mathsf{in}\, \lfloor e \rfloor_2)$

[B-CASE] $\qquad \dfrac{\{i,j\} = \{1,2\}}{\begin{array}{c}\mathsf{case}\,(w \mid w')\,\mathsf{of}\,\mathsf{inj}_1(x).\, e_1 \mid \mathsf{inj}_2(x).\, e_2 \\ \longrightarrow (\mathsf{case}\, w\,\mathsf{of}\,\mathsf{inj}_1(x).\,\lfloor e_1 \rfloor_1 \mid \mathsf{inj}_2(x).\,\lfloor e_2 \rfloor_1 \mid \mathsf{case}\, w'\,\mathsf{of}\,\mathsf{inj}_1(x).\,\lfloor e_1 \rfloor_2 \mid \mathsf{inj}_2(x).\,\lfloor e_2 \rfloor_2)\end{array}}$

[B-ASSUME] $\qquad \mathsf{assume}\,(w \mid w')\,\mathsf{in}\, e \longrightarrow (\mathsf{assume}\, w\,\mathsf{in}\,\lfloor e \rfloor_1 \mid \mathsf{assume}\, w'\,\mathsf{in}\,\lfloor e \rfloor_2)$

Typing rules

[BRACKET] $\qquad \dfrac{\Pi \Vdash (H^\pi \sqcup pc^\pi) \sqsubseteq pc'^\pi \qquad \Pi;\Gamma;pc' \vdash e_1 : \tau \qquad \Pi;\Gamma;pc' \vdash e_2 : \tau \qquad \Pi \vdash H^\pi \sqsubseteq \tau^\pi}{\Pi;\Gamma;pc \vdash (e_1 \mid e_2) : \tau}$

[BRACKET-VALUES] $\qquad \dfrac{\Pi \vdash H^\pi \sqsubseteq \tau^\pi \qquad \Pi;\Gamma;pc \vdash w_1 : \tau \qquad \Pi;\Gamma;pc \vdash w_2 : \tau}{\Pi;\Gamma;pc \vdash (w_1 \mid w_2) : \tau}$

Observation function

$$\mathcal{O}((e_1 \mid e_2), \Pi, p, \pi) \quad = \quad \begin{cases} \circ & \mathcal{O}(e_i, \Pi, p, \pi) = \circ \\ (\mathcal{O}(e_1, \Pi, p, \pi) \mid \mathcal{O}(e_2, \Pi, p, \pi)) & \text{otherwise} \end{cases}$$

Figure 11: Extensions for bracketed semantics

6.2 Bracketed semantics

Our noninterference proof is based on the bracketed semantics approach used by Pottier and Simonet [46]. This approach extends FLAC with *bracketed expressions* which represent two executions of a program, and allows us to reason about noninterference in FLAC, a *2-safety hyperproperty* [20], as *type safety* in the extended language. Any two FLAC terms e_1 and e_2 in the unbracketed language can be combined into a term $(e_1 \mid e_2)$ in the bracketed language. For any term in the bracketed language, a projection function $\lfloor \cdot \rfloor_i$, for $i \in \{1, 2\}$, extracts the term from each execution. Specifically, $\lfloor (e_1 \mid e_2) \rfloor_i = e_i$ and projections are homomorphic on other expressions; for example $\lfloor \lambda(x : \tau)[pc].\, e \rfloor_i = \lambda(x : \tau)[pc].\,\lfloor e \rfloor_i$. Since each element of the trace defined by evaluation of $e \xrightarrow{t}{}^* e'$ is an intermediate FLAC term, we define projections on traces $\lfloor t \rfloor_i$ as the sequence of projected terms $\lfloor t[0] \rfloor_i, ..., \lfloor t[n] \rfloor_i$ where $\lfloor t[0] \rfloor_i = \lfloor e \rfloor_i$ and $\lfloor t[n] \rfloor_i = \lfloor e' \rfloor_i$.

Figure 11 presents the bracket extensions for FLAC. Where-values w and expressions e may be bracketed. B-STEP evaluates expressions inside of brackets. The remaining B-* evaluation rules propagate brackets out of subexpressions. Note that projections are applied as the scope of brackets expand so that brackets can never become nested.

The bracketed evaluation rules are designed to ensure bracketed terms do not get stuck unless the unbracketed terms do. We verify this with the following lemma. Informally, if the bracketed term is stuck then either left or right execution is also stuck.

Lemma 1 (Stuck expressions). *If e gets stuck then $\lfloor e \rfloor_i$ is stuck for some $i \in \{1, 2\}$.*

Proof. By induction on the structure of e. See Appendix A for complete proof. □

The bracketed type system is parameterized by a fixed principal H which specifies which policies are considered secret or untrusted, and an authority projection π, depending on whether the type system is verifying confidentiality ($\rightarrow$) or integrity ($\leftarrow$). The typing rules BRACKET and BRACKET-VALUES illustrate the primary purpose of the bracketed semantics: to link distinguishable evaluations of an expression to the expression's type. BRACKET requires that the bracketed expressions e_1 and e_2 are typable at a pc' that protects $H^\pi \sqcup pc^\pi$ and pc'^π and have a type τ^π protects H^π. BRACKET-VALUES relaxes the restriction on pc for where-values. Some of the bracketed evaluation rules are necessary for completeness with respect to the unbracketed semantics, but are not actually necessary for well typed programs. Specifically B-CASE and B-ASSUME step on expressions that are never well typed since types of the form $\tau + \tau'$ and $(p \succcurlyeq q)$ cannot protect any instantiation of H.

Following Pottier and Simonet [46], our first result on the bracketed semantics is that they are sound and complete with respect to the unbracketed semantics. By soundness, we mean that given a step in the bracketed execution, then at least one of the left or right projections take a step such that they are in relation with the bracketed execution. By completeness, we mean that given a left and right execution, we can construct a corresponding bracketed execution.

Lemma 2 (Soundness). *If $e \longrightarrow e'$ then $\lfloor e \rfloor_k \longrightarrow^* \lfloor e' \rfloor_k$ for $k \in \{1, 2\}$.*

Proof. By induction on the evaluation of e. Observe that all bracketed rules in Figure 11 except B-STEP only expand brackets, so $\lfloor e \rfloor_k = \lfloor e' \rfloor_k$ for $k \in \{1, 2\}$. For B-STEP, $\lfloor e \rfloor_i \longrightarrow \lfloor e' \rfloor_i$ and $\lfloor e \rfloor_j = \lfloor e' \rfloor_j$. □

Lemma 3 (Completeness). *If $\lfloor e \rfloor_1 \longrightarrow^* w_1$ and $\lfloor e \rfloor_2 \longrightarrow^* w_2$, then there exists some w such that $e \longrightarrow^* w$ and $\lfloor w \rfloor_i = w_i$ for $i \in \{1, 2\}$.*

Proof. Assume $\lfloor e \rfloor_1 \longrightarrow^* w_1$ and $\lfloor e \rfloor_2 \longrightarrow^* w_2$. The extended set of rules in Figure 11 always move brackets out of subterms, and therefore can only be applied a finite number of times. Therefore, by Lemma 2, if e diverges, either $\lfloor e \rfloor_1$ or $\lfloor e \rfloor_2$ diverge; this contradicts our assumption.

Furthermore, by Lemma 1, if the evaluation of e gets stuck, either $\lfloor e \rfloor_1$ or $\lfloor e \rfloor_2$ gets stuck. Therefore, since we assumed $\lfloor e \rfloor_i \longrightarrow^* w_i$, then e must terminate, so $e \longrightarrow^* w$. Finally, by induction on the number of evaluation steps and Lemma 2, $\lfloor w \rfloor_i = w_i$ for $i \in \{1, 2\}$. □

Using the soundness and completeness results, we can relate properties of programs executed under the bracketed semantics to executions under the non-bracketed semantics. In particular, since bracketed expressions represent distinguishable executions of a program, and may only have types that protect $H^\rightarrow$ (or $H^\leftarrow$), it is important to establish that the type of a FLAC expression is preserved by evaluation. Lemma 4 states this formally. This lemma helps us reason about whether an expression is bracketed based on its type.

Lemma 4 (Subject Reduction). *Let $\Pi; \Gamma; pc \vdash e : \tau$. If $e \longrightarrow e'$ then $\Pi; \Gamma; pc \vdash e' : \tau$.*

Proof. By induction on the evaluation of e. See Appendix A for proof and supporting lemmas. □

Lemma 5 (Progress). *If $\Pi; \varnothing; pc \vdash e : \tau$, then either $e \longrightarrow e'$ or e is a where value.*

Proof. By induction on the derivation of $\Pi; \varnothing; pc \vdash e : \tau$. See Appendix A for complete proof. □

6.3   Delegation Compartmentalization and Invariance

FLAC programs dynamically extend trust relationships, enabling new flows of information. Nevertheless, well-typed programs have end-to-end semantic properties that enforce strong information security. These properties derive primarily from FLAC's control of the delegation context. The ASSUME rule ensures that only high-integrity proofs of authorization can extend the delegation context, and furthermore that such extensions occur only in high-integrity, lexically-scoped contexts.

That low-integrity contexts cannot extend the delegation context turns out to be a crucial property. This property allows us to state a useful invariant about the evaluation of FLAC programs. Recall that assume terms evaluate to where terms in the FLAC semantics. Thus, FLAC programs typically compute values containing a hierarchy of nested where terms. The terms record the values whose types were used to extend the delegation context during type checking.

For a well-typed FLAC program, we can prove that certain trust relationships could not have been added by the program. Characterizing these relationships requires a concept of the minimal authority required to cause one principal

to act for another. Although similar, this idea is distinct from the voice of a principal. Consider the relationship between a and $a \wedge b$. The voice of $a \wedge b$, $\nabla(a \wedge b)$, is sufficient integrity to add a delegation $a \wedge b$ to a so that $a \succcurlyeq a \wedge b$. Alternatively, having only the integrity of $\nabla(b)$ is also sufficient to add a delegation $a \succcurlyeq b$, which also results in $a \succcurlyeq a \wedge b$.

In our theorems, we need to be able to characterize what flows might become enabled by a program. For instance, if we want to reason about whether $a \succcurlyeq a \wedge b$ in some scope (and delegation context) of a program, we need to identify the *minimal necessary integrity* $\nabla(b)$ that can close the gap in authority between the pair of principals a and $a \wedge b$. The following definitions are in service of this goal.

The first definition formalizes the idea that two principals are considered equivalent in a given context if they act for each other.

Definition 2 (Principal Equivalence). We say that two principals p and q are *equivalent* in Π, denoted $\Pi \Vdash p \equiv q$, if

$$\Pi \Vdash p \succcurlyeq q \text{ and } \Pi \Vdash q \succcurlyeq p.$$

Next, we define the *factorization* of two principals in a given context. For two principals, p and q, their factorization involves representing q as the conjunction of two principals $q_s \wedge q_d$ such that $p \succcurlyeq q_s$ in the desired context. Note that p need not act for q_d, and that factorizations always exists. For example, the factorization $q_d = q$ and $q_s = \bot$ is valid for any p, q, and Π.

Definition 3 (Factorization). A Π-*factorization* of an ordered pair of principals (p, q) is a tuple (p, q_s, q_d) such that $\Pi \Vdash q \equiv q_s \wedge q_d$ and $\Pi \Vdash p \succcurlyeq q_s$.

Factorization lets us split q's authority into a portion (q_s) that delegates to p, and a portion (q_d) that may or may not. A minimal factorization makes q_d as small as possible. Specifically, a minimal factorization of p and q is a q_s and q_d such that q_s has greater authority and q_d has less authority than any other factorization of p and q in the same context.

Definition 4 (Minimal Factorization). A Π-factorization (p, q_s, q_d) of (p, q) is *minimal* if for any Π-factorization (p, q'_s, q'_d) of (p, q),

$$\Pi \Vdash q_s \succcurlyeq q'_s \text{ and } \Pi \Vdash q'_d \succcurlyeq q_d$$

A minimal factorization (p, q_s, q_d) of p and q for a given Π and *pc* identifies the authority necessary to cause p to act for q. Because q_s is the principal with the greatest authority such that $p \succcurlyeq q_s$ and $q \equiv q_s \wedge q_d$, then speaking for q_d is sufficient authority to cause p to act for q since adding the delegation $p \succcurlyeq q_d$ would imply that $p \succcurlyeq q$. This intuition also matches with the fact that $\Pi \Vdash p \succcurlyeq q_d$ if and only if $q_d = \bot$, which is the case if and only if $\Pi \Vdash p \succcurlyeq q$.

Observe that minimal Π-factorizations are also trivially unique up to equivalence.

Proposition 3 (Subtraction equivalence). *Let (p, q_s, q_d) and (p, q'_s, q'_d) be minimal factorizations of p and q in Π. Then $\Pi \Vdash q_s \equiv q'_s$ and $\Pi \Vdash q_d \equiv q'_d$.*

Proof. By Definition 4, $\Pi \Vdash q_s \succcurlyeq q'_s$ and $\Pi \Vdash q'_s \succcurlyeq q_s$. Therefore, $\Pi \Vdash q_s \equiv q'_s$. Likewise, $\Pi \Vdash q_d \succcurlyeq q'_d$ and $\Pi \Vdash q'_d \succcurlyeq q_d$, so we have $\Pi \Vdash q_s \equiv q'_s$. □

Since the q_d component of minimal factorization can be thought of as the "gap" in authority between two principals, we use q_d to define the notion of principal *subtraction*.

Definition 5 (Principal Subtraction). Let (p, q_s, q_d) be a minimal Π-factorization of (p, q). We define $q - p$ in Π to be q_d. That is, $\Pi \Vdash q - p \equiv q_d$. Note that $q - p$ is not defined outside of a judgement context.

Since q_d is unique up to equivalence in Π, $q - p$ is also unique for a given Π.

To further illustrate principal subtraction as an authority gap, consider the following equivalence: If a principal acts for the authority gap between it and any other principal, then it also acts for that principal.

Lemma 6 (Authority Gap Identity). *For any p and q, $\Pi \Vdash p \succcurlyeq q - p \Leftrightarrow \Pi \Vdash p \succcurlyeq q$*

Proof. Let the minimal factorization of p and q in Π be (p, q_s, q_d) where $\Pi \Vdash q \equiv q_s \wedge q_d$ and $\Pi \Vdash p \succcurlyeq q_s$. In the forward direction, assume $\Pi \Vdash p \succcurlyeq q - p$. For contradiction, assume $\Pi \nVdash p \succcurlyeq q$. Then it must be the case that $\Pi \nVdash p \succcurlyeq q_d$. But by Definition 5 $q - p$ is defined as q_d, so this implies $\Pi \nVdash p \succcurlyeq q - p$, a contradiction. □

Lemma 7 proves that minimal factorizations exist for all contexts and principals, so principal subtraction is well defined.

Lemma 7 (Minimal Factorizations Exist)**.** *For any context Π and principals p, q, there exists a minimal Π-factorization of (p, q).*

Proof. Given (p, q), we first let $q_s = p \vee q$. By definition, $\Pi \Vdash p \geqslant p \vee q$, and for all factorizations (p, q'_s, q'_d), $\Pi \Vdash p \geqslant q'_s$ and $\Pi \Vdash q \geqslant q'_s$, so $\Pi \Vdash q_s \geqslant q'_s$.

Now let $D = \{ r \in \mathcal{L} \mid \Pi \Vdash q \equiv q_s \wedge r \}$. Using FLAM normal form [9], all principals in $\mathcal{L}$ can be represented as a finite set of meets and joins of elements in $\mathcal{N} \cup \{ \top, \bot \}$, so q and q_s are finite. Π is also finite, adding only finitely-many dynamic equivalences, so D is finite up to equivalence. Moreover, since $\Pi \Vdash q \equiv (p \vee q) \wedge q$ (by absorption) we have $q \in D$. Therefore D is always non-empty and we can define $q_d = \bigvee D$.

Now let (p, q'_s, q'_d) be any Π-factorization of (p, q). We must show that $\Pi \Vdash q'_d \geqslant q_d$.

First, see that $\Pi \Vdash q \equiv q_s \wedge q'_d$. for one direction, observe that $\Pi \Vdash q \geqslant q_s$ and $\Pi \Vdash q \geqslant q_d$ (by Definition 2). For the other direction, since $\Pi \Vdash q_s \geqslant q'_s$, we have $\Pi \Vdash q_s \wedge q'_d \geqslant q'_s \wedge q'_d$, so $\Pi \Vdash q_s \wedge q'_d \geqslant q$.

Therefore, by the definition of D, we know $q'_d \in D$, so by the definition of $\vee$ and q_d, $\Pi \Vdash q'_d \geqslant q_d$. Thus (p, q_s, q_d) is a minimal Π-factorization of (p, q). $\square$

We can now state precisely which trust relationships may change in a given information flow context.[18]

Lemma 8 (Delegation Invariance)**.** *Suppose $\Pi \Vdash pc \geqslant \nabla(t)$. For all principals p and q, if $\Pi, \langle r \geqslant t \rangle \Vdash p \geqslant q$, then either $\Pi \Vdash p \geqslant q$ or $\Pi \Vdash pc \geqslant \nabla(q - p)$.*

Proof. There are two cases: either $\Pi \Vdash p \geqslant q$ or $\Pi \nVdash p \geqslant q$. The case where $\Pi \Vdash p \geqslant q$ is trivial. We prove the other case: if $\Pi \nVdash p \geqslant q$ and $\Pi, \langle r \geqslant t \rangle \Vdash p \geqslant q$ then $\Pi \Vdash pc \geqslant \nabla(q - p)$.

Let $\Pi' = \Pi, \langle r \geqslant t \rangle$. Assume that

$$\Pi' \Vdash p \geqslant q \tag{1}$$
$$\Pi \nVdash p \geqslant q \tag{2}$$

and Let (p, q_s, q_d) be the minimal Π-factorization of (p, q). So, $\Pi \Vdash p \geqslant q_s$ Since $\Pi \nVdash p \geqslant q$, this implies that $\Pi \nVdash p \geqslant q_d$ but from (1), we have that

$$\Pi' \Vdash p \geqslant q_d \tag{3}$$

So any derivation of (3) must involve the derivation of $\Pi' \Vdash r \geqslant t$ via R-ASSUME, since R-ASSUME is the only rule that uses the contents of Π. Therefore, without loss of generality, we can assume that either (a) $\Pi \Vdash t \geqslant q_d$, or (b) for some q_1 and q_2 such that $\Pi \Vdash q_d \equiv q_1 \wedge q_2$, $\Pi \Vdash t \geqslant q_1$ and $\Pi \nVdash t \geqslant q_2$; otherwise $\Pi' \Vdash r \geqslant t$ is unnecessary to prove (3).

In fact, it must be the case that $\Pi \Vdash t \geqslant q_1 \wedge q_2$ (or equivalently $\Pi \Vdash t \geqslant q_d$). To see why, assume $\Pi \nVdash t \geqslant q_1 \wedge q_2$: specifically, $\Pi \Vdash t \geqslant q_1$ and $\Pi \nVdash t \geqslant q_2$. Since $\Pi' \Vdash p \geqslant q_s \wedge (q_1 \wedge q_2)$ but $\Pi \nVdash t \geqslant q_2$, it must be that $\Pi \nVdash p \geqslant q_s \wedge q_2$, otherwise $\Pi' \Vdash p \geqslant q_s \wedge (q_1 \wedge q_2)$ wouldn't hold. Therefore $(p, q_s \wedge q_2, q_1)$ is a Π-factorization of (p, q). Since (p, q_s, q_d) is a *minimal* Π-factorization, we have that $\Pi \Vdash q_2 \geqslant q_d$. From R-TRANS, we now have that $\Pi \Vdash t \geqslant q_d$, but since we assumed $\Pi \nVdash t \geqslant q_d$, we have a contradiction. Hence $\Pi \Vdash t \geqslant q_d$.

By the monotonicity of $\nabla(\cdot)$ with respect to $\geqslant$ (Proven in Coq for FLAM [9]), we have $\Pi \Vdash \nabla(t) \geqslant \nabla(q_d)$. As shown above, $\Pi \Vdash pc \geqslant \nabla(t)$, so by R-TRANS, $\Pi \Vdash pc \geqslant \nabla(q_d)$. However, recall that $\Pi \Vdash q - p \equiv q_d$ (Definition 5) and so $\Pi \Vdash pc \geqslant \nabla(q - p)$. Hence proved. $\square$

Corollary 1. *Suppose $\Pi, \langle r_0 \geqslant t_0 \rangle, ..., \langle r_n \geqslant t_n \rangle \Vdash pc \geqslant \nabla(t_i)$ for all $i \in [0, n]$. For all principals p and q, if $\Pi, \langle r_0 \geqslant t_0 \rangle, ..., \langle r_n \geqslant t_n \rangle \Vdash p \geqslant q$ then either $\Pi \Vdash p \geqslant q$ or $\Pi \Vdash pc \geqslant \nabla(q - p)$.*

Proof. For $i = 0$, we apply Lemma 8. For the inductive case, assume that for all p and q, $\Pi, \langle r_0 \geqslant t_0 \rangle, ..., \langle r_{n-1} \geqslant t_{n-1} \rangle \Vdash p \geqslant q$ implies either $\Pi \Vdash p \geqslant q$ or $\Pi \Vdash pc \geqslant \nabla(q - p)$. We want to prove that for all p and q, $\Pi, \langle r_0 \geqslant t_0 \rangle, ..., \langle r_n \geqslant t_n \rangle \Vdash p \geqslant q$ implies either $\Pi \Vdash p \geqslant q$ or $\Pi \Vdash pc \geqslant \nabla(q - p)$ also. By applying Lemma 8 to $\Pi, \langle r_0 \geqslant t_0 \rangle, ..., \langle r_n \geqslant t_n \rangle \Vdash p \geqslant q$, we obtain that either $\Pi, \langle r_0 \geqslant t_0 \rangle, ..., \langle r_{n-1} \geqslant t_{n-1} \rangle \Vdash p \geqslant q$ or $\Pi, \langle r_0 \geqslant t_0 \rangle, ..., \langle r_{n-1} \geqslant t_{n-1} \rangle \Vdash pc \geqslant \nabla(q - p)$. In the first case, applying the inductive hypothesis gives us that either $\Pi \Vdash p \geqslant q$ or $\Pi \Vdash pc \geqslant \nabla(q - p)$ holds. In the second case, applying the inductive hypothesis gives us that either $\Pi \Vdash pc \geqslant \nabla(q - p)$ or $\Pi \Vdash pc \geqslant \nabla(\nabla(q - p) - pc)$ holds. If $\Pi \Vdash pc \geqslant \nabla(q - p)$ holds, then we are

---

[18]The original delegation invariance lemma [7] was flawed due to a case where a minor delegation could have a cascading effect that enabled new delegations, breaking the desired invariant. This new (slightly more restrictive) formulation, stated in terms of principal subtraction, addresses more precisely the connection between the *pc* and the invariant trust relationships.

done. Therefore, assume $\Pi \Vdash pc \succcurlyeq \nabla(\nabla(q-p) - pc)$. Observe that $\nabla(q-p)$ is an integrity principal. Specifically, $\Pi \Vdash \nabla(q-p)^{\rightarrow} \equiv \bot$), so by Definition 3 $\Pi \Vdash (\nabla(q-p) - pc)^{\rightarrow} \equiv \bot$. The voice of an integrity principal is just the principal (Lemma 37), so $\Pi \Vdash \nabla(\nabla(q-p) - pc) \equiv (\nabla(q-p) - pc)$. Then by Lemma 6, we know that $\Pi \Vdash pc \succcurlyeq (\nabla(q-p) - pc)$ is equivalent to $\Pi \Vdash pc \succcurlyeq \nabla(q-p)$. $\square$

If e is a well-typed, closed, source-level FLAC program—in other words $\Pi; \varnothing; pc \vdash e : \tau$, for some Π, pc, and τ— then Lemma 8 is sufficient to characterize the delegations introduced in e. Since e is closed, any delegation $\langle r \succcurlyeq t \rangle$ is introduced by an assume term that types under a sub-derivation of $\Pi; \varnothing; pc \vdash e : \tau$. Since the pc only becomes more restrictive in subexpressions of e, we know that $\Pi \Vdash pc \succcurlyeq \nabla(t)$.

More generally, however, an open FLAC programs may receive as input (non-source-level) values which are typed in higher-integrity contexts, and thus may use delegations where $\Pi \nVdash pc \succcurlyeq \nabla(t)$. Such delegations are constrained only by $\overline{pc}$ (since $\Pi \Vdash \overline{pc} \succcurlyeq \nabla(t)$ for well-typed where-terms). However, as long as such delegations are "compartmentalized," they cannot be used to downgrade arbitrarily.

To characterize how non-source-level terms can affect downgrading, it will be convenient to distinguish source-level programs from their inputs. We generalize our substitution notation from a single substitution $[x \mapsto v]$ to a set S, where $S(x) = v$ encodes the substitution $[x \mapsto v]$ and the substitution of all free variables in e that are defined by S is written $e\, S$.

Definition 6. Given $\Pi; \Gamma; pc \vdash e : \tau$, a *well-typed substitution on e for Γ in Π* is a substitution S where for each free variable x_i of e with $\Gamma(x_i) = \tau_i'$, we have $\Pi; \Gamma; pc \vdash S(x_i) : \tau_i'$ and $e\, S$ is closed.

Now we can state our Delegation Compartmentalization lemma. As long as all terms in a well-typed substitution are either source-level terms or are at least as restrictive as H^{π}, then if a source-level program e evaluates to w' where $\langle r \succcurlyeq t \rangle$, either $\Pi \Vdash pc \succcurlyeq \nabla(t)$ or the result of the program is also as restrictive as H^{π}.

Lemma 9 (Delegation Compartmentalization). *Suppose $\Pi; \Gamma; pc \vdash e : \tau$. Let S be a well-typed substitution on e for Γ in Π where e is a source-level term and for all entries $[y \mapsto w_y] \in S$ such that $\Pi; \Gamma; pc \vdash w_y : \Gamma(y)$, and either*

> *1. w_y is a source-level term, or*
>
> *2. $\Pi \Vdash H^{\pi} \sqsubseteq \Gamma(y)^{\pi}$*

Then if $(e\, S) \longrightarrow^* w'$ where $\langle r \succcurlyeq t \rangle$, either

> *(a) $\Pi \Vdash pc \succcurlyeq \nabla(t)$, or*
>
> *(b) $\Pi \Vdash H^{\pi} \sqsubseteq \tau^{\pi}$*

Proof. From the subject reduction (Lemma 4), we have $\Pi; \Gamma; pc \vdash w'$ where $\langle r \succcurlyeq t \rangle : \tau$. From the typing rule WHERE, we have $\Pi; \Gamma; pc \vdash v : (r \succcurlyeq t)$ for some r and t, and $\Pi'; \Gamma; pc \vdash w' : \tau$ for $\Pi' = \Pi, \langle r \succcurlyeq t \rangle$. There are two cases.

Case 1: Suppose $(e\, S) \longrightarrow^* w'$ where $\langle r \succcurlyeq t \rangle$ such that $\langle r \succcurlyeq t \rangle$ belongs to S. We know that all values substituted from S are well-typed w.r.t Π and pc. Without loss of generality, let $\langle r \succcurlyeq t \rangle$ be reduced from w_y. We have two cases:

> **Case w_y is a source-level term:** We are given $\Pi; \Gamma; pc \vdash w_y : \Gamma(y)$. Since w_y does not have any where terms, it must be the case that either w_y itself is $\langle r \succcurlyeq t \rangle$ expression or $\langle r \succcurlyeq t \rangle$ appears in some assume term in w_y. The latter is only possible if w_y is a lambda or type abstraction, because no other values can embed assume terms. From the monotonicity of pc (lemma 17), we have that any delegation $\langle r \succcurlyeq t \rangle$ propagated from w_y should satisfy $\Pi \Vdash pc \succcurlyeq \nabla(t)$. Hence proved (a)
>
> **Case $\Pi \Vdash H^{\pi} \sqsubseteq \Gamma(y)^{\pi}$:** Since (a) does not hold, we have that $\langle r \succcurlyeq t \rangle$ appears in S. Without loss of generality, assume that $\langle r \succcurlyeq t \rangle$ appears in w_y and that w_y is where term. Invoking Lemmas 33 and 34 that state how delegations propagate from where terms , we have the required proof.

Case 2: Suppose $(e\, S) \longrightarrow^* w'$ where $\langle r \succcurlyeq t \rangle$ such that $\langle r \succcurlyeq t \rangle$ did not propagate from S. Then, it must be the case that $(e\, S) \longrightarrow^* E[\text{assume } \langle r \succcurlyeq t \rangle \text{ in } e'] \longrightarrow^* w'$ where $\langle r \succcurlyeq t \rangle$. From subject reduction (Lemma 4), we have that for some Π', Γ', pc' and τ', $\Pi'; \Gamma'; pc' \vdash \text{assume } \langle r \succcurlyeq t \rangle \text{ in } e' : \tau'$ and so $\Pi' \Vdash pc' \succcurlyeq \nabla(t)$ (from the typing rule ASSUME). Note that $E[\text{assume } \langle r \succcurlyeq t \rangle \text{ in } e']$ gives us a valid $T[\text{assume } \langle r \succcurlyeq t \rangle \text{ in } e']$ such that $T = E$ (Lemma 14). Invoking the monotonicity of pc (Lemma 17) on $\Pi; \Gamma; pc \vdash T[\text{assume } \langle r \succcurlyeq t \rangle \text{ in } e'] : \tau$, we have that $\Pi \Vdash pc \sqsubseteq pc'$. Since Π' is an extension of Π

(Lemma 20), we have $\Pi' \Vdash pc \sqsubseteq pc'$. Applying transitivity on $\Pi' \Vdash pc \sqsubseteq pc'$ and $\Pi' \Vdash pc' \succcurlyeq \nabla(t)$, we have $\Pi' \Vdash pc \succcurlyeq \nabla(t)$. Depending on the set difference $\Pi' - \Pi$, we have two more cases:

Case 2.1: Case where some of the delegations in $\Pi' - \Pi$ have propagated from S. Then, going by the argument similar to the previous case, we have that either $\Pi \Vdash pc \succcurlyeq \nabla(t)$ or $\Pi \Vdash H^\pi \sqsubseteq \tau^\pi$. Hence proved.

Case 2.2: Case where none of the delegations in $\Pi' - \Pi$ have propagated from S. Then, by ASSUME, monotonicity of the pc (Lemma 17), and R-TRANS, for each delegation $\langle a \succcurlyeq b \rangle$ in $\Pi' - \Pi$, we have that $\Pi' \Vdash pc \succcurlyeq \nabla(b)$. Applying Corollary 1 to $\Pi' \Vdash pc \succcurlyeq \nabla(t)$ gives us either $\Pi \Vdash pc \succcurlyeq \nabla(t)$ or $\Pi \Vdash pc \succcurlyeq \nabla(\nabla(t) - pc)$, which are equivalent by Lemma 6.

$\square$

6.4 Noninterference

Lemmas 8 and 9 are critical for our proof of *noninterference*, a result that states that public and trusted output of a program cannot depend on restricted (secret or untrustworthy) information. Our proof of noninterference for FLAC programs relies on a proof of subject reduction under a bracketed semantics, based on the proof technique of Pottier and Simonet [46]. This technique is mostly standard, so we omit it here. The complete proof of subject reduction and other results are found in Appendix A.

In other noninterference results based on bracketed semantics, including [46], noninterference follows almost directly from the proof of subject reduction. This is because the subject reduction proof shows that evaluating a term cannot change its type. In FLAC, however, subject reduction alone is insufficient; evaluation may enable flows from secret or untrusted inputs to public and trusted types.

To see how, suppose e is a well-typed program according to $\Pi; \Gamma, x : \tau; pc \vdash e : \tau'$. Furthermore, let H be a principal such that $\Pi \vdash H \sqsubseteq \tau$ and $\Pi \nvdash H \sqsubseteq \tau'$. In other words, x is a "high" variable (more restrictive; secret and untrusted), and e evaluates to a "low" result (less restrictive; public and trusted). In [46], executions that differ only in secret or untrusted inputs must evaluate to the same value, since otherwise the value would not be well typed. In FLAC, however, if the pc has sufficient integrity, then an assume term could cause $\Pi'; pc \vdash H \sqsubseteq \tau'$ to hold in a delegation context Π' of a subterm of e. Additionally, even if the pc is low integrity, an input to the program may capture a delegation that the source program could not assume directly. For example, a dynamic hand-off term like those discussed in Section 5.2, with type

$$\forall X[pc].\,(p \text{ says } X \xrightarrow{pc} q \text{ says } X)$$

could enable the same flows as a delegation $\langle p \succcurlyeq q \rangle$, but without the condition that $\Pi \Vdash pc \succcurlyeq \nabla(q)$.

The key to proving our result relies on using Lemma 8 to constrain the delegations that can be added to Π' by the source-level terms, Lemma 9 to specify how non-source-level terms must be compartmentalized. Thus noninterference in FLAC is dependent on H and its relationship to pc and the type τ'. For confidentiality, most of this reasoning occurs in the proof of Lemma 10, which does most of the heavy lifting for the noninterference proof. Specifically, Lemma 10 states that for a well-typed program e, if the pc is insufficiently trusted to create new flows from $H^\rightarrow$ to $\ell^\rightarrow$, then if the portion of $\lfloor e \rfloor_1$ observable to $\ell^\rightarrow$ under Π is equal to the observable portion of $\lfloor e \rfloor_2$, then if $e \longrightarrow^* e'$, the observable portions of e' are still equal.

Lemma 10 (Confidentiality Erasure Conservation). *Suppose $\Pi; \Gamma; pc \vdash e : \tau$ and let S be a well-typed substitution of e for Γ in Π. Then for some H and ℓ such that $\Pi \nVdash \ell^\rightarrow \succcurlyeq H^\rightarrow$ and $\Pi \nVdash pc \succcurlyeq \nabla(H^\rightarrow - \ell^\rightarrow)$, if $\mathcal{O}(\lfloor e\,S \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor e\,S \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$ and e is a source-level term, and for all entries $[y \mapsto w_y] \in S$ with $\Pi; \Gamma; pc \vdash w_y : \Gamma(y)$, either w_y is a source-level term or $\Pi \Vdash H^\rightarrow \wedge \top^\leftarrow \sqsubseteq \Gamma(y)$ then $(e\,S) \longrightarrow e'$ implies $\mathcal{O}(\lfloor e' \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor e' \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$.*

Proof Sketch. By induction on the evaluation of e. The most interesting case is the step O-CTX where the term escapes it protection context. Here we invoke the delegation invariance (Lemma 8) to argue that the delegations in a well-typed program cannot close the authority gap between $H^\rightarrow$ and $\ell^\rightarrow$, and thus do not change the observability of a term.

Detailed proof is presented in Appendix A.4.

$\square$

Lemma 10 holds for mutiple steps as well and is presented in the Corollary 2.

Corollary 2 (Erasure Conservation for Multiple Steps). *Suppose $\Pi; \Gamma; pc \vdash e : \tau$ and let S be a well-typed substitution of e for Γ in Π. Then for some H and ℓ such that $\Pi \nVdash \ell^\rightarrow \succcurlyeq H^\rightarrow$ and $\Pi \nVdash pc \succcurlyeq \nabla(H^\rightarrow - \ell^\rightarrow)$, if $\mathcal{O}(\lfloor e\,S \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow$*

) $= \mathcal{O}(\lfloor e\ S \rfloor_2, \Pi, \ell^{\rightarrow}, \rightarrow)$ *and* e *is a source-level term, and for all entries* $[y \mapsto w_y] \in S$ *with* $\Pi; \Gamma; pc \vdash w_y :$ $\Gamma(y)$, *either* w_y *is a source-level term or* $\Pi \Vdash H^{\rightarrow} \sqsubseteq \Gamma(y)^{\rightarrow}$ *then* $(e\ S) \longrightarrow^{*} e'$ *implies* $\mathcal{O}(\lfloor e' \rfloor_1, \Pi, \ell^{\rightarrow}, \rightarrow) =$ $\mathcal{O}(\lfloor e' \rfloor_2, \Pi, \ell^{\rightarrow}, \rightarrow)$.

Proof. From from the transitive closure of Lemma 10. $\hfill\square$

Theorem 1 concerns programs with an input of type τ' and an output of type τ. The input x is secret, thus τ' must protect $H^{\rightarrow}$ (Condition 1). The outputs, however, are public and therefore information derived from the inputs should not flow to the outputs (Condition 2). Therefore, if pc is insufficiently trusted to create new flows from $H^{\rightarrow}$ to $\ell^{\rightarrow}$ (Condition 3) then executions of e that differ only in the value of τ'-typed inputs values must produce traces that are indistinguishable.

Theorem 1 (Confidentiality Noninterference). *Let* $\Pi; \Gamma; x : \tau'; pc \vdash e : \tau$ *for some* H *and* ℓ *such that*

    1. $\Pi \Vdash H^{\rightarrow} \sqsubseteq \tau'^{\rightarrow}$

    2. $\Pi \nVdash H^{\rightarrow} \sqsubseteq \ell^{\rightarrow}$

    3. $\Pi \nVdash pc \geqslant \nabla(H^{\rightarrow} - \ell^{\rightarrow})$

If e *is a source-level term, and* S *is a well-typed substitution for* Γ *in* Π *where for all entries* $[y \mapsto w_y] \in S$ *with* $\Pi; \Gamma; pc \vdash w_y : \Gamma(y)$, *either* w_y *is a source-level term or* $\Pi \Vdash H^{\rightarrow} \wedge \top^{\leftarrow} \sqsubseteq \Gamma(y)$. *Then for all* $w_z, z \in \{1, 2\}$ *such that* $\Pi; \Gamma; pc \vdash w_z : \tau'$, *if* $(e\ S)[x \mapsto w_z] \xrightarrow{t_z}^{*} w'_z$, *then* $t_1 \approx^{\Pi}_{\ell^{\rightarrow}} t_2$.

Proof. From the soundness and completeness properties of the bracketed language (Lemmas 2 and 3), we can construct a bracketed execution $(e\ S)[x \mapsto (v_1 \mid v_2)] \xrightarrow{t}^{*} v'$ such that $\lfloor v' \rfloor_z = v'_z$ and $\lfloor t \rfloor_z = t_z$ for $z = \{1, 2\}$. We will occasionally write v or v' as shorthand for $(v_1 \mid v_2)$ or $(v'_1 \mid v'_2)$.

Since $\Pi; \Gamma; pc \vdash v_z : \tau'$ for $z \in \{1, 2\}$, then we have $\Pi; \Gamma; pc \vdash (v_1 \mid v_2) : \tau'$ via BRACKET-VALUES. Therefore, by Lemma 23 (Variable Substitution) of $\Pi; \Gamma; x : \tau'; pc \vdash e : \ell$ says τ, we have $\Pi; \Gamma; pc \vdash e[x \mapsto (v_1 \mid v_2)] : \ell$ says τ. Then by induction of the number of evaluation steps in $(e\ S)[x \mapsto v] \xrightarrow{t}^{*} e'$ and subject reduction (Lemma 4), we have $\Pi; \Gamma; pc \vdash e' : \ell$ says τ.

We now want to prove that $t_1 \approx^{\Pi}_{\ell^{\rightarrow}} t_2$. First, consider $(e\ S)[x \mapsto (v_1 \mid v_2)]$. To prove that $\mathcal{O}((e\ S)[x \mapsto v_1], \Pi, \ell^{\rightarrow}, \rightarrow$ $) = \mathcal{O}((e\ S)[x \mapsto v_2], \Pi, \ell^{\rightarrow}, \rightarrow)$, it suffices to show that $\mathcal{O}(v_1, \Pi, \ell^{\rightarrow}, \rightarrow) = \mathcal{O}(v_2, \Pi, \ell^{\rightarrow}, \rightarrow)$. Note that conditions 1 and 2 satisfy all the necessary conditions for invoking Lemma 31. By Lemma 31 (erasure of projected protected values is equal), we have that $\mathcal{O}(v_1, \Pi, \ell^{\rightarrow}, \rightarrow) = \mathcal{O}(v_2, \Pi, \ell^{\rightarrow}, \rightarrow)$. Thus $\mathcal{O}(\lfloor e[x \mapsto v] \rfloor_1, \Pi, \ell^{\rightarrow}, \rightarrow) = \mathcal{O}(\lfloor e[x \mapsto v] \rfloor_2, \Pi, \ell^{\rightarrow}, \rightarrow)$. Invoking erasure conservation (Corollary 2), we have $\mathcal{O}(\lfloor e' \rfloor_1, \Pi, \ell^{\rightarrow}, \rightarrow) = \mathcal{O}(\lfloor e' \rfloor_2, \Pi, \ell^{\rightarrow}, \rightarrow)$. Thus $t_1 \approx^{\Pi}_{\ell^{\rightarrow}} t_2$. $\hfill\square$

Specializing our noninterference results on confidentiality provides more precision, but integrity versions of Lemma 10 and Theorem 1 hold by similar arguments. We present their statements here, but not the corresponding proofs.

Lemma 11 (Integrity Erasure Conservation). *Suppose* $\Pi; \Gamma; pc \vdash e : \tau$ *and let* S *be a well-typed substitution of e for* Γ *in* Π. *Then for some* H *and* ℓ *such that* $\Pi \nVdash H^{\leftarrow} \geqslant \ell^{\leftarrow}$ *and* $\Pi \nVdash pc \geqslant \ell^{\leftarrow} - H^{\leftarrow}$, *if* $\mathcal{O}(\lfloor e\ S \rfloor_1, \Pi, \ell^{\leftarrow}, \leftarrow) =$ $\mathcal{O}(\lfloor e\ S \rfloor_2, \Pi, \ell^{\leftarrow}, \leftarrow)$ *and* e *is a source-level term, and for all entries* $[y \mapsto w_y] \in S$ *with* $\Pi; \Gamma; pc \vdash w_y : \Gamma(y)$, *either* w_y *is a source-level term or* $\Pi \Vdash H^{\leftarrow} \sqsubseteq \Gamma(y)^{\leftarrow}$ *then* $(e\ S) \longrightarrow e'$ *implies* $\mathcal{O}(\lfloor e' \rfloor_1, \Pi, \ell^{\leftarrow}, \leftarrow) = \mathcal{O}(\lfloor e' \rfloor_2, \Pi, \ell^{\leftarrow}, \leftarrow)$.

Theorem 2 (Integrity Noninterference). *Let* $\Pi; \Gamma; x : \tau'; pc \vdash e : \tau$ *for some* H *and* ℓ *such that*

    1. $\Pi \Vdash H^{\leftarrow} \sqsubseteq \tau'$

    2. $\Pi \nVdash H^{\leftarrow} \sqsubseteq \ell^{\leftarrow}$

    3. $\Pi \nVdash pc \geqslant \ell^{\leftarrow} - H^{\leftarrow}$

If e *is a source-level term, and* S *is a well-typed substitution for* Γ *in* Π *where for all entries* $[y \mapsto w_y] \in S$ *with* $\Pi; \Gamma; pc \vdash w_y : \Gamma(y)$, *either* w_y *is a source-level term or* $\Pi \Vdash H^{\leftarrow} \sqsubseteq \Gamma(y)^{\leftarrow}$. *Then for all* $w_z, z \in \{1, 2\}$ *such that* $\Pi; \Gamma; pc \vdash w_z : \tau'$, *if* $(e\ S)[x \mapsto w_z] \xrightarrow{t_z}^{*} w'_z$, *then* $t_1 \approx^{\Pi}_{\ell^{\leftarrow}} t_2$.

Noninterference is a key tool for obtaining many of the security properties we seek. For instance, noninterference is essential for verifying the properties of commitment schemes and bearer credentials discussed in Section 2.

Unlike some definitions of noninterference, our definition does not prohibit all downgrading. Instead, Conditions 1 and 2 define relationships between principals H and ℓ, and Condition 3 ensures (via Lemma 8) those relationships remain unchanged by delegations made within the program, and thus by Lemma 10, the trace of a program observable to an attack remains unchanged during evaluation.

6.5   Robust declassification

Robust declassification [56] requires disclosures of secret information to be independent of low-integrity information. Robust declassification permits some confidential information to be disclosed to an attacker, but attackers can influence neither the decision to disclose information nor the choice of what information is disclosed. Therefore, robust declassification is a more appropriate security condition than noninterference when programs are intended to disclose information.

Following Myers et al. [41], we extend our set of terms with a "hole" term $[\bullet^\tau]$ representing portions of a program that are under the control of an attacker. Attackers may insert any term of type τ to complete the program. We extend the type system with the following rule for holes, parameterized on the same H used by the bracketed typing rules BRACKET and BRACKET-VALUES:

$$[\textsc{Hole}] \qquad \frac{\Pi \Vdash H^\leftarrow \geqslant pc^\leftarrow \qquad \Pi \Vdash pc^\rightarrow \sqsubseteq \Delta(H^\leftarrow)}{\Pi; \Gamma; pc \vdash \bullet^\tau : \tau}$$

Where $\Delta(H^\leftarrow)$ is the *view* of principal $H^\leftarrow$. The view of a principal is a dual notion to voice, introduced by Cecchetti *et al.* [18] to represent the confidentiality observable to a principal. Given a principal in normal form $q^\rightarrow \wedge r^\leftarrow$, the view of that principal is

$$\Delta(q^\rightarrow \wedge r^\leftarrow) \triangleq q^\rightarrow \wedge r^\rightarrow$$

In other words $\Delta(H^\leftarrow)$ represents the confidentiality of information observable to the attacker $H^\leftarrow$. Therefore, the HOLE premises $\Pi \Vdash H^\leftarrow \geqslant pc^\leftarrow$ and $\Pi \Vdash pc^\rightarrow \sqsubseteq \Delta(H^\leftarrow)$ require that holes be inserted only in contexts that are controlled by and observable to the attacker.

We write $e[\vec{\bullet^\tau}]$ to denote a program e with holes. Let an *attack* be a vector $\vec{a}$ of terms and $e[\vec{a}]$ be the program where a_i is substituted for $\bullet_i^{\tau_i}$.

Definition 7. Π-fair attacks   An attack $\vec{a}$ is a Π-*fair attack* on a well-typed program with holes $\Pi; \Gamma; pc \vdash e[\vec{\bullet^{\vec{\tau}*}}] :$ τ if the program $e[\vec{a}]$ is also well typed (thus $\Pi; \Gamma; pc \vdash e[\vec{a}] : \tau$), and furthermore, for each $a_i \in \vec{a}$, we have $\Pi; \Gamma_i^*; H^\leftarrow \wedge \Delta(H^\leftarrow) \vdash a_i : \tau_i^*$ where each entry in Γ_i^* has the form $y : \ell$ says τ'' and $\Pi \Vdash \ell^\rightarrow \sqsubseteq \Delta(H^\leftarrow)$.

By specifying the relationships between confidential information labeled $H^\rightarrow$ and the integrity of the attacker $H^\leftarrow$, we can precisely express the authority an attacker is able to wield in its Π-fair attacks against protected information. Proposition 4 states that as long as the attacker cannot observe secret information labeled $H^\rightarrow$, or $\Pi \nVdash H^\rightarrow \sqsubseteq \Delta(H^\leftarrow)$, then Π-fair attacks cannot reference secret variables directly.

Proposition 4 (No free secrets)**.** *For H and Π such that $\Pi \nVdash H^\rightarrow \sqsubseteq \Delta(H^\leftarrow)$, suppose $\vec{a}$ is a Π-fair attack on a program $e[\vec{\bullet^{\vec{\tau}*}}]$ where $\Pi; \Gamma, x : \tau'; pc \vdash e[\vec{\bullet^{\vec{\tau}*}}] : \tau$. Then $\Pi \Vdash H^\rightarrow \sqsubseteq \tau'$ implies x is not a free variable of $a_i \in \vec{a}$ for all i.*

Proof. Since $\vec{a}$ is a Π-fair attack, we have $\Pi; \Gamma_i'; H^\leftarrow \wedge \Delta(H^\leftarrow) \vdash a_i : \tau_i^*$ for each a_i in $\vec{a}$ and τ_i^* in $\vec{\tau}*$. By assumption, $\Pi \Vdash H^\rightarrow \sqsubseteq \tau'$ and $\Pi \nVdash H^\rightarrow \sqsubseteq \Delta(H^\leftarrow)$, so by the definition of Π-fair attacks, x cannot be in Γ_i'. Therefore, x is not free in a_i. $\qquad\Box$

In other words, Π-fair attacks must be well-typed on variables observable to the attacker in delegation context Π. Unfair attacks give the attacker enough power to break security directly by creating new declassifications without exploiting existing ones. Restricting attacks to noninterfering Π-fair attacks also rule out nonsensical scenarios such as when the "attacker" has the authority to read the confidential information. Fair attacks under the conditions of Proposition 4 represent a reasonable upper-bound on the power of the attacker over low-integrity portions of the program. Intuitively, low-integrity portions of the program are treated as though they are executed on a host controlled by the attacker.

Our robust declassification theorem is stated in terms of an attacker's delegation context Π_H which is used to specify the noninterfering Π_H-fair attacks on a program. The program itself is typed under a distinct delegation context Π that may be extended by assume terms to permit new flows of information, including flows observable to the attacker.

Theorem 3 (Robust declassification). *Suppose* $\Pi; \Gamma, x:\tau', \Gamma'; pc \vdash e[\vec{\bullet}^{\vec{\tau}*}] : \tau$. *For* Π_H *and* H *such that*

1. $\Pi_H \vdash H^{\rightarrow} \sqsubseteq \tau'$

2. $\Pi_H \nvdash H^{\rightarrow} \sqsubseteq \Delta(H^{\leftarrow})$

3. $\Pi_H \nvdash H^{\leftarrow} \succcurlyeq \nabla(H^{\rightarrow})$,

Then for all Π_H*-fair attacks* $\vec{a_1}$ *and* $\vec{a_2}$ *such that* $\Pi; \Gamma, x:\tau', \Gamma'; pc \vdash e[\vec{a_i}] : \tau$ *and* $\Pi; \Gamma; pc \vdash v_i : \tau'$, *if* $e[a_j][x \mapsto v_i] \longrightarrow e'_{ij}$ *for* $i, j \in \{1, 2\}$, *then for the traces* $t_{ij} = (e[a_j][x \mapsto v_i]) \cdot e'_{ij})$, *we have*

$$t_{11} \approx^{\Pi}_{\Delta(H^{\leftarrow})} t_{21} \iff t_{12} \approx^{\Pi}_{\Delta(H^{\leftarrow})} t_{22}$$

Proof Sketch. By induction on the evaluation of e. For detailed proof, refer to Section B. $\square$

Our formulation of robust declassification is in some sense more general than previous definitions since it permits some endorsements. Previous definitions of robust declassification [41, 56] forbid endorsement altogether. *Qualified robustness* [41] treats endorsed values as having an arbitrary value, so executing a program with endorsements generates a *set* of traces. Two executions are considered indistinguishable under qualified robustness if, for each trace generated by one execution, there exists a low-equivalent trace that is generated by the other execution. Our definition of robust declassification does not apply to programs that endorse and then declassify values that may have been influenced by the attacker. Qualified robustness does apply to such programs.

For some programs, FLAC offers a stronger guarantee than the possibilistic one offered by qualified robustness. Consider $H = \text{bob}^{\leftarrow} \wedge \text{alice}^{\rightarrow}$ and two secret inputs $\overline{\eta}_{\text{alice}^{\rightarrow} \wedge \text{carol}^{\leftarrow}} v_i$ for $i \in 1, 2$. Program $p[\bullet^{\tau'}]$ declassifies alice's secret to bob, then passes the result to a function that endorses it from carol to bob. The result is passed to a function controlled by the attacker, $\text{bob}^{\leftarrow}$.

$$
\begin{aligned}
\text{declass} =& \lambda(y:\text{alice}^{\rightarrow} \wedge \text{carol}^{\leftarrow} \text{ says } \tau)[\text{alice}^{\leftarrow}]. \\
& \quad \text{assume } \langle \text{bob} \succcurlyeq \text{alice} \rangle \text{ in } (\text{bind } y' = y \text{ in } \overline{\eta}_{\text{bob}^{\rightarrow} \wedge \text{carol}^{\leftarrow}} y') \\
\text{endorse} =& \lambda(y:\text{bob}^{\rightarrow} \wedge \text{carol}^{\leftarrow} \text{ says } \tau)[\text{carol}^{\leftarrow}]. \\
& \quad \text{assume } \langle \text{bob} \succcurlyeq \text{carol} \rangle \text{ in } (\text{bind } y' = y \text{ in } \overline{\eta}_{\text{bob}^{\rightarrow} \wedge \text{bob}^{\leftarrow}} y') \\
p[\bullet^{\tau'}] =& (\lambda(y:\text{bob says } \tau)[\text{bob}^{\leftarrow}]. [\bullet^{\tau'}]) \text{ (endorse (declass } x))
\end{aligned}
$$

p is well typed in the following context:

$$\langle \text{alice}^{\leftarrow} \succcurlyeq \text{bob}^{\leftarrow} \rangle, \langle \text{carol}^{\leftarrow} \succcurlyeq \text{bob}^{\leftarrow} \rangle; x : \text{alice}^{\rightarrow} \wedge \text{carol}^{\leftarrow} \text{ says } \tau; (\text{alice} \wedge \text{carol})^{\leftarrow} \vdash p[\bullet^{\tau'}] : \text{bob says } \tau$$

The program p and choice of H fullfil the conditions of Theorem 3 with the secret inputs substituted for the variable x and $\Pi_H = \Pi = \{\langle \text{alice}^{\leftarrow} \succcurlyeq \text{bob}^{\leftarrow} \rangle, \langle \text{carol}^{\leftarrow} \succcurlyeq \text{bob}^{\leftarrow} \rangle\}$. Thus no fair attacks substituted into the hole can violate robust declassification, for all traces generated by different choices of attacks and inputs. Under the qualified robustness semantics, the value returned by endorse would be treated as an arbitrary value and would guarantee only that, out of all of the traces generated by each choice of return value, there *exists* some trace that satisfies robustness.

Theorem 3 also permits some declassifications that prior definitions of robust declassification reject. For example, our definition admits declassifications of x even if $\Pi \Vdash H^{\leftarrow} \sqsubseteq \tau'$. In other words, even though low-integrity attacks cannot influence declassification, it is possible to declassify a secret input that has low-integrity. Therefore, an attacker that is permitted to influence the secret input could affect how much information is revealed. This is an example of a *malleable information flow* [18], which neither FLAC nor prior definitions (in the presence of endorsement) prevent in general. Cecchetti *et al.* [18] present a language based on FLAC that replaces assume with restricted declassification and endorsement terms to enforce nonmalleable information flow.

7   Examples revisited

We now implement our examples from Section 2 in FLAC and discuss their formal properties. Using FLAC ensures that authority and information flow assumptions are explicit, and that programs using these abstractions are secure with respect to those assumptions. In this section, we discuss how FLAC types help enforce specific end-to-end security properties for commitment schemes and bearer credentials.

$\texttt{commit} : \forall N[p^\leftarrow]. \, \forall X[p^\leftarrow]. \, N \xrightarrow{p^\leftarrow} p^\rightarrow \text{ says } X \xrightarrow{p^\leftarrow} p \text{ says } (N, X)$

$\texttt{commit} = \Lambda N[p^\leftarrow]. \, \Lambda X[p^\leftarrow]. \, \lambda(n : N)[p^\leftarrow]. \, \lambda(x : p^\rightarrow \text{ says } X)[p^\leftarrow].$
$\qquad\qquad \text{assume } \langle \perp^\leftarrow \geqslant p^\leftarrow \rangle \text{ in bind } x' = x \text{ in } \eta_p \, (n, x')$

$\texttt{reveal} : \forall N[\nabla(p^\rightarrow)]. \, \forall X[q^\leftarrow]. \, p \text{ says } (N, X) \xrightarrow{q^\leftarrow} q^\rightarrow \wedge p^\leftarrow \text{ says } (N, X)$

$\texttt{reveal} = \Lambda N[\nabla(p^\rightarrow) \wedge p^\leftarrow]. \, \text{assume } \langle \nabla(q^\rightarrow) \geqslant \nabla(p^\rightarrow) \rangle \text{ in assume } \langle q^\leftarrow \geqslant p^\leftarrow \rangle \text{ in}$
$\qquad\qquad \text{assume } \langle q^\rightarrow \geqslant p^\rightarrow \rangle \text{ in } \Lambda X[q^\leftarrow]. \, \lambda(x : p \text{ says } (N, X))[q^\leftarrow]. \, \text{bind } x' = x \text{ in } \eta_{q^\rightarrow \wedge p^\leftarrow} \, x'$

$\texttt{open} : \forall N[q^\leftarrow]. \, \forall X[q^\leftarrow]. \, (\forall Y[q^\leftarrow]. \, p \text{ says } (N, Y) \xrightarrow{q^\leftarrow} q^\rightarrow \wedge p^\leftarrow \text{ says } (N, Y)) \xrightarrow{q^\leftarrow}$
$\qquad\qquad p \text{ says } (N, X) \xrightarrow{q^\leftarrow} q^\leftarrow \text{ says } (q^\rightarrow \wedge p^\leftarrow \text{ says } (N, X))$

$\texttt{open} = \Lambda N[q^\leftarrow]. \, \Lambda X[q^\leftarrow]. \, \lambda(f : (\forall Y[q^\leftarrow]. \, p \text{ says } (N, Y) \xrightarrow{q^\leftarrow} q^\rightarrow \wedge p^\leftarrow \text{ says } (N, Y)))[q^\leftarrow].$
$\qquad\qquad \lambda(x : p \text{ says } (N, X))[q^\leftarrow]. \, \eta_{q^\leftarrow} \, (f \, X \, x)$

Figure 12: FLAC implementations of commitment scheme operations.

7.1 Commitment Schemes

Figure 12 contains the essential operations of a commitment scheme—commit, reveal, and open—implemented in FLAC. Principal p commits to pairs of the form (n, v) where n is a term that encodes a type-level integer N. Any reasonable integer encoding is permissible provided that each integer N is a singleton type. For example, we could use pairs to define the zero type as unit and the successor type as unit $\times \, \tau$ where τ is any valid integer type. Thus an integer N would be represented by N nested pairs. To prevent q from influencing which committed values are revealed, p commits to a single value for each integer type.

The commit operation takes a value of any type (hence $\forall X$) with confidentiality $p^\rightarrow$ and produces a value with confidentiality and integrity p. In other words, p *endorses* [58] the value to have integrity $p^\leftarrow$.

Attackers should not be able to influence whether principal p commits to a particular value. The *pc* constraint on commit ensures that only principal p and principals trusted with at least p's integrity, $p^\leftarrow$, may apply commit to a value. Furthermore, if the programmer omitted this constraint or instead chose $\perp^\leftarrow$, then commit would be rejected by the type system. Specifically, the assume term would not type-check via rule ASSUME since the *pc* does not act for $\nabla(p^\leftarrow)$, which is equal to $p^\leftarrow$.

When p is ready to reveal a previously committed value, it instantiates the function reveal with the integer type N of the committed value and sends the result to q. The body of reveal creates delegations $\langle \nabla(q^\rightarrow) \geqslant \nabla(p^\rightarrow) \rangle$ and $\langle q^\rightarrow \geqslant p^\rightarrow \rangle$, permitting the inner function to relabel its argument from p to $q^\rightarrow \wedge p^\leftarrow$. The outer assume term establishes that principals speaking for $q^\rightarrow$ also speak for $p^\rightarrow$ by creating an integrity relationship between their voices. With this relationship in place, the inner assume term may delegate p's confidentiality to q.[19]

Only principals trusted by $\nabla(p^\rightarrow)$ and $p^\leftarrow$ may instantiate reveal with the integer N of the commitment, therefore q cannot invoke reveal until it is authorized by p. If, for example, the type function had *pc* annotation q instead of $\nabla(p^\rightarrow)$, it would be rejected by the type system since the assume term would not check under the ASSUME rule. Note however, that once the delegation is established, the inner function may be invoked by q since it has the *pc* annotation $q^\leftarrow$. Without the delegation, however, the bind term would fail to type-check under BINDM since p would not flow to $q^\rightarrow \wedge p^\leftarrow$.

Given a function of the type instantiated by reveal and a committed value of type p says X, the open function may be invoked by q to relabel the committed value to a type observable by q. Because open may only be invoked by principals trusted by $q^\leftarrow$, p cannot influence which value of type p says (N, X) open is applied to. If q only accepts a one value for each integer type N, then the argument must be the one originally committed to by p. Furthermore, since the reveal function provided to p is parametric with respect to X, it cannot return a value other than (a relabeled

---

[19]Specifically, it satisfies the ASSUME premise $\Pi \Vdash \nabla(p^\rightarrow) \geqslant \nabla(q^\rightarrow)$.

version of) its argument. The return type $q^{\leftarrow}$ says $q^{\rightarrow} \wedge p^{\leftarrow}$ says X protects this relationship between the committed value and the opened one: it indicates that q trusts that $q^{\rightarrow} \wedge p^{\leftarrow}$ says X is the one originally committed to by p. [20]

The real power of FLAC is that the security guarantees of well-typed FLAC functions like those above are compositional. The FLAC type system ensures the security of both the functions themselves and the programs that use them. For instance, the following code should be rejected because it would permit q to reveal p's commitments.

$$\Lambda N[q^{\leftarrow}].\,\Lambda X[q^{\leftarrow}].\,\lambda(x{:}p \wedge q^{\leftarrow} \text{ says } X)[q^{\leftarrow}].\,\mathsf{assume}\ \langle q \geqslant p \rangle \text{ in } \mathsf{reveal}\ N\ X\ x$$

The pc constraints on this function all have the integrity of q, but the body of the function uses assume to create a new delegation from p to q. If this assume was permitted by the type system, then q could use the function to reveal p's committed values. Since ASSUME requires the pc to speak for p, this function is rejected.

FLAC's guarantees make it possible to state general security properties of all programs that use the above commitment scheme, even if those programs are malicious. For example, suppose we have a typing context that includes the commit, reveal, and open functions from Figure 12.

$$\Gamma_{cro} = \mathsf{commit}{:}\forall N[p^{\leftarrow}].\,\forall X[p^{\leftarrow}].\,N \xrightarrow{p^{\leftarrow}} p^{\rightarrow} \text{ says } X \xrightarrow{p^{\leftarrow}} p \text{ says } (N, X),$$

$$\mathsf{reveal}{:}\forall N[\nabla(p^{\rightarrow})].\,\forall X[q^{\leftarrow}].\,p \text{ says } (N, X) \xrightarrow{q^{\leftarrow}} q^{\rightarrow} \wedge p^{\leftarrow} \text{ says } (N, X)$$

$$\mathsf{open}{:}\forall N[q^{\leftarrow}].\,\forall X[q^{\leftarrow}].\,(\forall Y[q^{\leftarrow}].\,p \text{ says } (N, Y) \xrightarrow{q^{\leftarrow}} q^{\rightarrow} \wedge p^{\leftarrow} \text{ says } (N, Y)) \xrightarrow{q^{\leftarrow}}$$

We can consider programs under the control of principal q by considering source-level FLAC terms that type under Γ_{cro} at $pc = q^{\leftarrow}$. Note that this category includes potentially malicious programs that attempt to abuse the commitment scheme operations or otherwise attempt to access committed values. Since we are interested in properties that hold for all principals p and q, we want the properties to hold in an empty delegation context: $\Pi = \varnothing$. Below, we omit the delegation context altogether for brevity.

FLAC's noninterference guarantee helps rule out information that an attacker can influence or learn. We instantiate the environment Γ_{cro} with a well-typed substitution S_{cro} that replaces commit, reveal, and open with the terms defined in Figure 12. We can now show that:

- q **cannot learn a value that hasn't been revealed.** For simplicity, we instantiate the type variables N and X with τ_N and τ_X. We can represent values of some type τ_X observable to q with the type $q^{\rightarrow}$ says τ_X. For any e such that

  $$\Gamma_{cro}, x{:}p \text{ says } (\tau_N, \tau_X); q^{\leftarrow} \vdash e : q^{\rightarrow} \text{ says } \tau$$

  and any committed values v_1 and v_2 where $q^{\leftarrow} \vdash v_i : p \text{ says } (\tau_N, \tau_X)$, if $e\,S_{cro}[x \mapsto v_1] \longrightarrow^* v_1'$ and $e\,S_{cro}[x \mapsto v_2] \longrightarrow^* v_2'$, then by Theorem 1 with $H = p^{\rightarrow} \wedge q^{\leftarrow}$ and $\ell = q^{\rightarrow} \wedge p^{\leftarrow}$ $t_1 \approx_{q^{\rightarrow}}^{\varnothing} t_2$. Note that the same approach can also be used to prove q cannot learn p's uncommitted secrets.

- p **cannot cause** q **to open modified value.** We represent a value opened by q with the type $q^{\leftarrow}$ says $q^{\rightarrow} \wedge p^{\leftarrow}$ says (τ_N, τ_X), which represents an open value trusted by q. Only q should have the authority to accept and open a commitment, so we want to prove that p cannot produce a value of this type, even when the open operation is in scope. For any e such that

  $$\Gamma_{cro}, x{:}p \text{ says } (\tau_N, \tau_X); p^{\leftarrow} \vdash e : q^{\leftarrow} \text{ says } (q^{\rightarrow} \wedge p^{\leftarrow} \text{ says } (\tau_N, \tau_X))$$

  and any committed values v_1 and v_2 where $p^{\leftarrow} \vdash v_i : p \text{ says } (\tau_N, \tau_X)$, if $e\,S_{cro}[x \mapsto v_1] \longrightarrow^* v_1'$ and $e\,S_{cro}[x \mapsto v_2] \longrightarrow^* v_2'$, then by Theorem 2 with $H = q^{\rightarrow} \wedge p^{\leftarrow}$ and $\ell = p^{\rightarrow} \wedge q^{\leftarrow}$ $t_1 \approx_{q^{\leftarrow}}^{\varnothing} t_2$.

For these properties we consider programs using our commitment scheme that q might invoke, hence we consider FLAC programs that type-check in the $\Gamma_{cro}; pc_q$ context. In the first property, we are concerned with programs that produce values protected by policy p. Since such programs produce values with the integrity of p but are invoked by q, we want to ensure that no program exists that enables q to obtain a value with p's integrity that depends on x, which is a value without p's integrity. The second property concerns programs that produces values at $\ell \sqcap q^{\rightarrow}$ for any ℓ; these are values readable by q. Therefore, we want to ensure that no program exists that enables q to produce such a value that depends on x or y, which are not readable by q.

Each of these properties hold by a straightforward application of our noninterference theorems (Theorems 1 and 2). This result is strengthened by our robust declassification theorem (Theorem 3), which ensures that attacks by q on p's programs cannot subvert the intended declassifications.

---

[20]The original commitment scheme presented in Arden and Myers [7] contained a receive and open terms that are rejected by our type system because of the updated UNITM rule. The new premise $\Pi \Vdash pc \sqsubseteq \ell$ would require these terms to be executed at a pc trusted by both p and q. Since such a pc does not adequately model cryptographic commitment schemes where no trusted third party is required, we modified the scheme to better fit the new type system requirements.

7.2 Bearer Credentials

We can also use FLAC to implement bearer credentials, our second example of a dynamic authorization mechanism. We represent a bearer credential with authority k in FLAC as a term with the type

$$\forall X[pc].\, k^{\rightarrow} \text{ says } X \xrightarrow{pc} k^{\leftarrow} \text{ says } X$$

which we abbreviate as $k^{\rightarrow} \xRightarrow{pc} k^{\leftarrow}$. These terms act as bearer credentials for a principal k since they may be used as a proxy for k's confidentiality and integrity authority. Recall that $k^{\leftarrow} = k^{\leftarrow} \wedge \perp^{\rightarrow}$ and $k^{\rightarrow} = k^{\rightarrow} \wedge \perp^{\leftarrow}$. Then secrets protected by $k^{\rightarrow}$ can be declassified to $\perp^{\rightarrow}$, and untrusted data protected by $\perp^{\leftarrow}$ can be endorsed to $k^{\leftarrow}$. Thus this term wields the full authority of k, and if $pc = \perp^{\leftarrow}$, the credential may be used in any context—any "bearer" may use it. From such credentials, more restricted credentials can be derived. For example, the credential $k^{\rightarrow} \xRightarrow{pc} \perp^{\rightarrow}$ grants the bearer authority to declassify k-confidential values, but no authority to endorse values.

It is interesting to note that DCC terms analogous to FLAC terms with type $k^{\rightarrow} \xRightarrow{pc} k^{\leftarrow}$ would only be well-typed in DCC if k is equivalent to $\perp$. This is because the function takes an argument with $k^{\rightarrow}$ confidentiality and no integrity, and produces a value with $k^{\leftarrow}$ integrity and no confidentiality. Suppose $\mathcal{L}$ is a security lattice used to type-check DCC programs with suitable encodings for k's confidentiality and integrity. If a DCC term has a type analogous to $k^{\rightarrow} \Rightarrow k^{\leftarrow}$, then $\mathcal{L}$ must have the property $k^{\rightarrow} \sqsubseteq \perp$ and $\perp \sqsubseteq k^{\leftarrow}$. This means that k has no confidentiality and no integrity. That FLAC terms may have this type for any principal k makes it straightforward to implement bearer credentials and demonstrates a useful application of FLAC's extra expressiveness.

The pc of a credential $k^{\rightarrow} \xRightarrow{pc} k^{\leftarrow}$ acts as a sort of caveat: it restricts the information flow context in which the credential may be used. We can add more general caveats to credentials by wrapping them in lambda terms. To add a caveat ϕ to a credential with type $k^{\rightarrow} \xRightarrow{pc} k^{\leftarrow}$, we use a wrapper:

$$\lambda(x \colon k^{\rightarrow} \xRightarrow{pc} k^{\leftarrow})[pc].\, \Lambda X[pc].\, \lambda(y \colon \phi)[pc].\, xX$$

which gives us a term with type

$$\forall X[pc].\, \phi \xrightarrow{pc} k^{\rightarrow} \text{ says } X \xrightarrow{pc} k^{\leftarrow} \text{ says } X$$

This requires a term with type ϕ (in which X may occur) to be applied before the authority of k can be used. Similar wrappers allow us to chain multiple caveats; i.e., for caveats $\phi_1 \ldots \phi_n$, we obtain the type

$$\forall X[pc].\, \phi_1 \xrightarrow{pc} \ldots \xrightarrow{pc} \phi_n \xrightarrow{pc} k^{\rightarrow} \text{ says } X \xrightarrow{pc} k^{\leftarrow} \text{ says } X$$

which abbreviates to

$$k^{\rightarrow} \xRightarrow{\phi_1 \times \cdots \times \phi_n ; pc} k^{\leftarrow}$$

We will also use the syntax $(k^{\rightarrow} \xRightarrow{\phi_1 \times \cdots \times \phi_n ; pc} k^{\leftarrow})\, \tau$ (suggesting a type-level application) as an abbreviation for

$$\phi_1[X \mapsto \tau] \xrightarrow{pc} \ldots \xrightarrow{pc} \phi_n[X \mapsto \tau] \xrightarrow{pc} k^{\rightarrow} \text{ says } \tau \xrightarrow{pc} k^{\leftarrow} \text{ says } \tau$$

Like any other FLAC terms, credentials may be protected by information flow policies. So a credential that should only be accessible to Alice might be protected by the type $\text{Alice}^{\rightarrow} \text{ says } (k^{\rightarrow} \xRightarrow{\phi; pc} k^{\leftarrow})$. This confidentiality policy ensures the credential cannot accidentally be leaked to an attacker. A further step might be to constrain uses of this credential so that only Alice may invoke it to relabel information. If we require $pc = \text{Alice}^{\leftarrow}$, this credential may only be used in contexts trusted by Alice: $\text{Alice}^{\rightarrow} \text{ says } (k^{\rightarrow} \xRightarrow{\phi; \text{Alice}^{\leftarrow}} k^{\leftarrow})$.

A subtle point about the way in which we construct caveats is that the caveats are polymorphic with respect to X, the same type variable the credential ranges over. This means that each caveat may constrain what types X may be instantiated with. For instance, suppose we want to encode a relation isEduc for specifying movie topics that are educational. One possible encoding is a polymorphic function of type

$$\text{isEduc} : \forall X[p].\, p \text{ says } (X \xrightarrow{p} X, U)$$

where p is the principal with movie classification authority, and U is a unique type-level integer (such as those described in Section 7.1) associated with the isEduc relation (and no other). Values of type $p \text{ says } (\tau \xrightarrow{p} \tau, U)$ can be used as evidence that type τ belongs to isEduc, the relation associated with U.[21]

---

[21] Using an instantiated identity function type $\tau \xrightarrow{p} \tau$ instead of just τ avoids having to produce a value of type τ just to define the relation. The pc label chosen for the identity function is irrelevant.

Only code that is at least as trusted as pc may apply this function, therefore only authorized code may add types to the isEduc relation. We might, for instance, apply isEduc to types like Biography and Documentary, but not RomanticComedy. Adding pc says $(X \xrightarrow{pc} X, U)$ as a caveat to a credential would mean that the bearer of the credential could use the credential plus evidence of membership in isEduc to access biographies and documentaries. Since no term of type pc says $(\text{RomanticComedy} \xrightarrow{pc} \text{RomanticComedy}, U)$ exists (nor could be created by the bearer), the bearer can only satisfy the caveat by instantiating X with Biography or Documentary. Once X is instantiated with Biography or Documentary, the credential cannot be used on a RomanticComedy value. Thus we have two mechanisms for constraining the use of credentials: information flow policies to constrain propagation, and caveats to establish prerequisites and constrain the types of data covered by the credential.

As another example of using such credentials, suppose Alice hosts a file sharing service. For a simpler presentation, we use free variables to refer to these files; for instance, $x_1 : (k_1 \text{ says photo})$ is a variable that stores a photo (type photo) protected by k_1. For each such variable x_1, Alice has a credential $k_1^{\rightarrow} \overset{\perp^{\leftarrow}}{\Longrightarrow} k_1^{\leftarrow}$, and can give access to users by providing this credential or deriving a more restricted one. To access x_1, Bob does not need the full authority of Alice or k_1—a more restricted credential suffices. Alice can provide Bob with a credential c of type $(k_1 \overset{\text{Bob}^{\leftarrow}}{\Longrightarrow} k_1^{\leftarrow})$ photo. By applying this credential to x_1, Bob is able to access the result of type $k_1^{\leftarrow}$ says photo since its confidentiality is now public.

This example demonstrates an advantage of bearer credentials: access to x_1 can be provided to principals other than k_1 in a decentralized way, without changing the policy on x_1. Suppose Alice has a credential with type $k_1^{\rightarrow} \overset{\perp^{\leftarrow}}{\Longrightarrow} k_1^{\leftarrow}$ and wants to issue the above credential to Bob. Alice can create such a credential using a wrapper that derives the more constrained credential from her original one.

$$\lambda(c : k_1^{\rightarrow} \overset{\perp^{\leftarrow}}{\Longrightarrow} k_1^{\leftarrow})[\text{Alice}^{\leftarrow}].$$
$$\lambda(y : k_1 \text{ says photo})[\text{Bob}^{\leftarrow}].$$
$$\text{bind } y' = y \text{ in } (c \text{ photo}) (\eta_{k_1^{\rightarrow}} y')$$

This function has type $(k_1^{\rightarrow} \overset{\perp^{\leftarrow}}{\Longrightarrow} k_1^{\leftarrow}) \xrightarrow{\text{Alice}^{\leftarrow}} (k_1 \overset{\text{Bob}^{\leftarrow}}{\Longrightarrow} k_1^{\leftarrow})$ photo: given her root credential $k_1^{\rightarrow} \overset{\perp^{\leftarrow}}{\Longrightarrow} k_1^{\leftarrow}$, Alice (or someone she trusts) can create a restricted credential that allows Bob (or someone he trusts) to access values of type photo protected under k_1.

Bob can also use this credential to share photos with friends. For instance, the function

$$\lambda(c : (k_1 \overset{\text{Bob}^{\leftarrow}}{\Longrightarrow} k_1^{\leftarrow}) \text{ photo})[\text{Bob}^{\leftarrow}].$$
$$\text{assume } \langle \text{Carol}^{\leftarrow} \geqslant \text{Bob}^{\leftarrow} \rangle \text{ in}$$
$$\lambda(y : k_1 \text{ says photo})[\text{Carol}^{\leftarrow}].$$
$$\text{bind } y' = y \text{ in } (c \text{ photo}) (\eta_{k_1^{\rightarrow}} y')$$

creates a wrapper around a Bob's credential that is invokable by Carol. Using the assume term, Bob delegates authority to Carol so that his credential may be invoked at pc Carol$^{\leftarrow}$.

The properties of FLAC let us prove many general properties about such bearer-credential programs. We first define the context we are interested in. We can model the premise that only Alice has access to credentials by protecting the secrets and the credentials in the typing context:

$$\Gamma_{bc} = x_i : k_i \text{ says } \tau_i, c_i : k_i \text{ says } (k_i^{\rightarrow} \overset{\perp^{\leftarrow}}{\Longrightarrow} k_i^{\leftarrow})$$

and delegating authority from principal k_i to Alice in the delegation context:

$$\Pi_{bc} = \langle \text{Alice} \geqslant k_i \rangle$$

where k_i is a primitive principal protecting the ith resource of type τ_i, and c_i is a credential for the ith resource and protected by principal k_i, which delegates authority to Alice.

- p **cannot access resources without a credential.** A concise way of representing values that are observable to p (with any integrity) is the principal $p^{\rightarrow}$, which is the most restrictive policy observable to p. Since any less restrictive policy observable to p flows to this policy, any output observable to p is protected at this type.

29

Suppose e is a program that computes such an output. For any e such that $\Gamma_{bc}; p^{\leftarrow} \vdash e : p^{\rightarrow}$ says τ', for some τ', we can show that the evaluation of e is independent of the secret variables x_i. To see how, fix some index n and define

$$S_{bc} = [x_j \mapsto \overline{\eta}_{k_j} \, v_j][c_i \mapsto (\overline{\eta}_{k_i} \, ...)]$$

for all i and all j where $j \neq n$ and $p^{\leftarrow} \vdash v_j : k_j^{\rightarrow}$ says τ_j. Additionally, suppose we have a two values v_n^1 and v_n^2 such that $p^{\leftarrow} \vdash v_n^i : k_n$ says τ_n for $i \in \{1, 2\}$. Then if $S_{bc} \, e[x_n \mapsto v_n^1] \xrightarrow{t_1}^* w_n^1$ and $S_{bc} \, e[x_n \mapsto v_n^2] \xrightarrow{t_2}^* w_n^2$, by Theorem 1 with $H = k_n^{\rightarrow} \wedge p^{\leftarrow}$ (and $\ell^{\rightarrow} = p^{\rightarrow}$), we have $t_1 \approx_{p^{\rightarrow}}^{\Pi_{bc}} t_2$. Since we left n abstract, we can repeat this argument for all x_i.

- **Alice cannot accidentally disclose secrets by issuing credentials.** Suppose e is a program that outputs a credential observable to p that declassifies secrets protected at $k_n^{\rightarrow}$. For any e such that $\Gamma_{bc}, y : $ Alice says $\tau; k_n^{\leftarrow} \vdash e : p^{\rightarrow}$ says $k_n^{\rightarrow} \xRightarrow{\perp^{\leftarrow}} p_n^{\rightarrow}$, for some τ, we can show that the evaluation of e does not depend on Alice's secret variable y. Define S_{bc} as above and consider values v^1 and v^2 such that $p^{\leftarrow} \vdash v^i : $ Alice says τ' for $i \in \{1, 2\}$. Then if $S_{bc} \, e[y \mapsto v^1] \xrightarrow{t_1}^* w^1$ and $S_{bc} \, e[y \mapsto v^2] \xrightarrow{t_2}^* w^2$, by Theorem 1 with $H = $ Alice$^{\rightarrow} \wedge p^{\leftarrow}$ (and $\ell^{\rightarrow} = p^{\rightarrow}$), we have $t_1 \approx_{p^{\rightarrow}}^{\Pi_{bc}} t_2$.

These properties demonstrate the some of the power of FLAC's type system. The first ensures that credentials really are necessary for p to access protected resources, even indirectly. If p has no credentials, and the type system ensures that p cannot invoke a program that produces a value p can read (represented by $\ell \sqcap p^{\rightarrow}$) that depends on the value of any variable x_i. The second property eliminates covert channels like the one discussed in Section 2.2. It implies that credentials issued by Alice do not leak information. By implementing and using bearer credentials in FLAC, we can demonstrate these properties with relatively little effort by appealing to Theorem 1.

8   Related Work

Arden, Liu, and Myers [9] introduced the Flow-Limited Authorization Model for reasoning about trust relationships that may be secret or untrustworthy. FLAC's type systems and semantic results rely on a fragment of this logic that reasons from the perspective of the compiler. In this fragment, all FLAM queries occur at compile time, and the compiler does not depend on any (secret or public) dynamic data to answer queries. Furthermore, the compiler does not communicate with other principals during derivation. Our static lattice rules (Figure 1) provide an alternate axiomatization of the FLAM principal algebra, but we have proven them equivalent in Coq for the ownership-free fragment of FLAM.

The original FLAC formalization [7] contained several errors both in the language definition and the proofs. Some of the design changes we made for the current formalization were to correct errors. For example, we added a (necessary) *pc* annotation to type functions similar to that for lambda values, and added missing where-propagation rules, along with a progress result, Lemma 5. Other changes were inspired by subsequent work like DFLATE [26] and NMIFC [18]. For example, protection contexts (Figure 9) were inspired by the TEE abstraction from DFLATE to properly handle the erasure of intermediate expressions in the trace semantics, and both NMIFC and DFLATE first adopted the more restrictive UNITM and protection rules that make the says modality non-commutative.

The original noninterference and robust declassification results were on output terms only, similar to the Pottier and Simonet [46] noninterference result. Our formalization leverages a trace semantics and erasure function, similar to the approach in DFLATE. The new noninterference and robustness theorems are thus stronger than the original statements since attackers not only see the output of a program but also its (observable) intermediate values. We conjecture that most of the design changes would be necessary to repair the proofs of the output-only versions of our theorems (with suitable updates reflecting changes to the Delegation Invariance statement), but we have not attempted to distinguish which (if any) changes were necessary only for the stronger results.

NMIFC and DFLATE each handle downgrading differently than FLAC. NMIFC does not permit arbitrary delegation via an assume term, but instead uses declassify and endorse for downgrading confidentiality and integrity explicitly. DFLATE treats delegation contexts and where terms subtly differently than FLAC. First, DFLATE functions are explicitly annotated with the delegations they capture from the delegation context, and may only be applied in contexts where those delegations are valid. In contrast, FLAC functions implicitly capture delegations and two functions of the same type may capture different delegations. Second, rather than being purely formal bookkeeping mechanisms, where terms serve as certificates of delegated authority at runtime and are propagated between hosts. One implication is that where delegations are directly observable, whereas FLAC's erasure function omits them.

Flamio [45] is a course-grained, dynamic IFC language similar to LIO [49], but using the FLAM distributed authorization logic for dynamic label comparisons. FLAC's type system also uses the FLAM logic, but only a static, local fragment, which significantly simplifies the information flows that occur due to authorization checks.

Many languages and systems for authorization or access control have combined aspects of information security and authorization (e.g., [54, 28, 37, 50, 38, 11]) in dynamic settings. However, almost all are susceptible to security vulnerabilities that arise from the interaction of information flow and authorization [9].

DCC [4, 2] has been used to model both authorization and information flow, but not simultaneously. DCC programs are type-checked with respect to a static security lattice, whereas FLAC programs can introduce new trust relationships during evaluation, enabling more general applications.

Boudol [16] defines an imperative language with terms that enable or disable flows for a lexical scope—similar to assume terms—but does not restrict their usage. Rx [50] and RTI [11] use labeled roles to represent information flow policies. The integrity of a role restricts who may change policies. However, information flow in these languages is not robust [41]: attackers may indirectly affect how flows change when authorized principals modify policies.

Some prior approaches have sought to reason about the information security of authorization mechanisms. Becker [12] discusses *probing attacks* that leak confidential information to an attacker. Garg and Pfenning [24] present a logic that ensures assertions made by untrusted principals cannot influence the truth of statements made by other principals.

Previous work has studied information flow control with higher-order functions and side effects. In the SLam calculus [27], implicit flows due to side effects are controlled via *indirect reader* annotations on types. Zdancewic and Myers [57] and Flow Caml [46] control implicit flows via *pc* annotations on function types. FLAC also controls side effects via a *pc* annotation, but here the side effects are changes in trust relationships that define which flows are permitted. Tse and Zdancewic [52] also extend DCC with a program-counter label but for a different purpose: their *pc* tracks information about the protection context, permitting more terms to be typed.

DKAL⋆ [31] is an executable specification language for authorization protocols, simplifying analysis of protocol implementations. FLAC may be used as a specification language, but FLAC offers stronger guarantees regarding the information security of specified protocols. Errors in DKAL⋆ specifications could lead to vulnerabilities. For instance, DKAL⋆ provides no intrinsic guarantees about confidentiality, which could lead to authorization side channels or probing attacks.

The Jif programming language [39, 42] supports dynamically computed labels through a simple dependent type system. Jif also supports dynamically changing trust relationships through operations on principal objects [19]. Because the signatures of principal operations (e.g., to add a new trust relationship) are missing the constraints imposed by FLAC, authorization can be used as a covert channel.

Aura [32] embeds a DCC-based proof language and type system in a dependently-typed general-purpose functional language. As in DCC, Aura programs may derive new authorization proofs using existing proof terms and a monadic bind operator. However, since Aura only tracks dependencies between proofs, it is ill-suited for reasoning about the end-to-end information-flow properties of authorization mechanisms. In general, dependently-typed languages (e.g., [51, 36, 44, 17]) are also expressive enough to encode relations and constraints like those used in FLAC's type system, but all FLAC programs have the semantic security guarantees presented in Section 6 by construction.

9   Discussion and Future Directions

Existing security models do not account fully for the interactions between authorization and information flow. The result is that both the implementations and the uses of authorization mechanisms can lead to insecure information flows that violate confidentiality or integrity. The security of information flow mechanisms can also be compromised by dynamic changes in trust. This paper has proposed FLAC, a core programming language that coherently integrates these two security paradigms, controlling the interactions between dynamic authorization and secure information flow. FLAC offers strong guarantees and can serve as the foundation for building software that implements and uses authorization securely. Further, FLAC can be used to reason compositionally about secure authorization and secure information flow, guiding the design and implementation of future security mechanisms.

We have already mentioned subsequent work like DFLATE [26], which extends FLAC concepts to distributed TEE applications, and NMIFC [18], which enforces nonmalleable information flow control, a new downgrading semantic condition that generalizes robust declassification to include integrity downgrading. To simplify its formalization, NMIFC removed the assume term from FLAC. Another promising direction is extending FLAC to incorporate abstractions for new mechanisms such as secure quorum replication, secure multi-party computation, and other cryptographic mechanisms as security abstractions in the language. Abstractions for secure mobile code sharing, such as that sup-

ported by Mobile Fabric [8], would also be interesting since it would require considering the provider of code in addition to its information flow properties.

Features such as quantification over principals and dynamic principal values, as found in Jif [39, 59, 42] would increase the usefulness of specifying security policies with FLAC types. Initial explorations of these features have been implemented in Flame [6], an embedded Haskell DSL based on FLAC.

Acknowledgments

References

[1] M. Abadi. Logic in access control. In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, LICS '03, pages 228–233, Washington, DC, USA, 2003. IEEE Computer Society. URL http://dx.doi.org/10.1109/LICS.2003.1210062.

[2] M. Abadi. Access control in a core calculus of dependency. In *11th ACM SIGPLAN Int'l Conf. on Functional Programming*, pages 263–273, New York, NY, USA, 2006. ACM. doi: 10.1145/1159803.1159839. URL http://doi.acm.org/10.1145/1159803.1159839.

[3] M. Abadi. Variations in access control logic. In R. van der Meyden and L. van der Torre, editors, *Deontic Logic in Computer Science*, volume 5076 of *Lecture Notes in Computer Science*, pages 96–109. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-70524-6. doi: 10.1007/978-3-540-70525-3_9. URL http://dx.doi.org/10.1007/978-3-540-70525-3_9.

[4] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 147–160, Jan. 1999. URL http://dl.acm.org/citation.cfm?id=292555.

[5] O. Arden. FLAM Coq proof (Static Actsfor Branch). https://bitbucket.org/owenarden/flam-proof/src/static_actsfor/, 2020.

[6] O. Arden and P. Buiras. Flame. Software release, https://users.soe.ucsc.edu/~owen/projects/flame/.

[7] O. Arden and A. C. Myers. A calculus for flow-limited authorization. In *29th IEEE Computer Security Foundations Symp. (CSF)*, pages 135–147, June 2016. URL http://www.cs.cornell.edu/andru/papers/flac.

[8] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers. Sharing mobile code securely with information flow control. In *IEEE Symp. on Security and Privacy*, pages 191–205, May 2012. URL http://www.cs.cornell.edu/andru/papers/mobile.html.

[9] O. Arden, J. Liu, and A. C. Myers. Flow-limited authorization. In *28th IEEE Computer Security Foundations Symp. (CSF)*, pages 569–583, July 2015. URL http://www.cs.cornell.edu/andru/papers/flam.

[10] O. Arden, J. Liu, and A. C. Myers. Flow-limited authorization: Technical report. Technical Report 1813–40138, Cornell University Computing and Information Science, May 2015. URL http://hdl.handle.net/1813/40138.

[11] S. Bandhakavi, W. Winsborough, and M. Winslett. A trust management approach for flexible policy management in security-typed languages. In *Computer Security Foundations Symposium, 2008*, pages 33–47, 2008. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4556677.

[12] M. Y. Becker. Information flow in trust management systems. *Journal of Computer Security*, 20(6):677–708, 2012.

[13] M. Y. Becker, C. Fournet, and A. D. Gordon. SecPAL: Design and semantics of a decentralized authorization language. *Journal of Computer Security*, 18(4):619–665, 2010. URL http://research.microsoft.com/apps/pubs/default.aspx?id=70334.

[14] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, Apr. 1977. (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.).

[15] A. Birgisson, J. G. Politz, Ú. Erlingsson, A. Taly, M. Vrable, and M. Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *Network and Distributed System Security Symp.*, 2014. URL http://theory.stanford.edu/~ataly/Papers/macaroons.pdf.

[16] G. Boudol. Secure information flow as a safety property. In *Formal Aspects in Security and Trust (FAST)*, pages 20–34. Springer, 2008.

[17] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming*, 23(5):552, 2013.

[18] E. Cecchetti, A. C. Myers, and O. Arden. Nonmalleable information flow control. In *24th ACM Conf. on Computer and Communications Security (CCS)*, pages 1875–1891, Oct. 2017.

[19] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *16th USENIX Security Symp.*, Aug. 2007. URL http://www.cs.cornell.edu/andru/papers/sif.pdf.

[20] M. R. Clarkson and F. B. Schneider. Hyperproperties. In *IEEE Computer Security Foundations Symp. (CSF)*, pages 51–65, June 2008.

[21] R. A. Eisenberg and S. Weirich. Dependently typed programming with singletons. In *2012 Haskell Symposium*, pages 117–130. ACM, Sept. 2012. doi: 10.1145/2364506.2364522. URL http://doi.acm.org/10.1145/2364506.2364522.

[22] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. Internet RFC-2693, Sept. 1999.

[23] D. Ferraiolo and R. Kuhn. Role-based access controls. In *15th National Computer Security Conference*, 1992.

[24] D. Garg and F. Pfenning. Non-interference in constructive authorization logic. In *19th IEEE Computer Security Foundations Workshop (CSFW)*, New Jersey, USA, 2006. IEEE. URL http://dl.acm.org/citation.cfm?id=1155684.

[25] M. D. George. *The Decentralized Security Principle for Structuring Distributed Systems*. PhD thesis, Cornell University Department of Computer Science. In preparation.

[26] A. Gollamudi, S. Chong, and O. Arden. Information flow control for distributed trusted execution environments. In *32nd IEEE Computer Security Foundations Symp. (CSF)*, June 2019.

[27] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 365–377, San Diego, California, Jan. 1998.

[28] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic. Dynamic updating of information-flow policies. In *Foundations of Computer Security Workshop*, 2005. URL http://www.cs.umd.edu/~mwh/papers/secupdate.pdf.

[29] A. K. Hirsch, P. H. A. d. Amorim, E. Cecchetti, R. Tate, and O. Arden. First-order logic for flow-limited authorization. In *33rd IEEE Computer Security Foundations Symp. (CSF)*, pages 123–138, 2020. doi: 10.1109/CSF49147.2020.00017.

[30] J. Howell and D. Kotz. A formal semantics for SPKI. In *ESORICS 2000*, volume 1895 of *Lecture Notes in Computer Science*, pages 140–158. Springer Berlin Heidelberg, 2000. URL http://dx.doi.org/10.1007/10722599_9.

[31] J.-B. Jeannin, G. de Caso, J. Chen, Y. Gurevich, P. Naldurg, and N. Swamy. DKAL⋆: Constructing executable specifications of authorization protocols. In *Engineering Secure Software and Systems*, pages 139–154. Springer, 2013. URL http://link.springer.com/chapter/10.1007/978-3-642-36563-8_10.

[32] L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. Aura: A programming language for authorization and audit. In *13th ACM SIGPLAN Int'l Conf. on Functional Programming*, Sept. 2008.

[33] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. In *13th ACM Symp. on Operating System Principles (SOSP)*, pages 165–182, Oct. 1991. *Operating System Review*, 253(5).

[34] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust-management framework. In *IEEE Symp. on Security and Privacy*, pages 114–130, 2002. URL http://dl.acm.org/citation.cfm?id=829514.830539.

[35] B. Martin, M. Brown, A. Paller, D. Kirby, and S. Christey. 2011 cwe/sans top 25 most dangerous software errors. *Common Weakness Enumeration*, 7515, 2011. URL `http://cwe.mitre.org/top25/`.

[36] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL `http://coq.inria.fr`. Version 8.0.

[37] K. Minami and D. Kotz. Secure context-sensitive authorization. *Journal of Pervasive and Mobile Computing*, 1(1):123–156, March 2005. doi: 10.1016/j.pmcj.2005.01.004. URL `http://www.cs.dartmouth.edu/~dfk/papers/minami-jcsa.pdf`.

[38] K. Minami and D. Kotz. Scalability in a secure distributed proof system. In *4th International Conference on Pervasive Computing*, volume 3968 of *Lecture Notes in Computer Science*, pages 220–237, Dublin, Ireland, May 2006. Springer-Verlag. doi: 10.1007/11748625_14. URL `http://www.cs.dartmouth.edu/~dfk/papers/minami-scalability.pdf`.

[39] A. C. Myers. JFlow: Practical mostly-static information flow control. In *26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, Jan. 1999. URL `http://www.cs.cornell.edu/andru/papers/popl99/popl99.pdf`.

[40] A. C. Myers and B. Liskov. Complete, safe decentralized information flow. Submitted for Publication, 1998.

[41] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, 2006. URL `http://www.cs.cornell.edu/andru/papers/robdecl-jcs`.

[42] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif 3.0: Java information flow. Software release, `http://www.cs.cornell.edu/jif`, July 2006. URL `http://www.cs.cornell.edu/jif`.

[43] M. Naor. Bit commitment using pseudorandomness. *Journal of cryptology*, 4(2):151–158, 1991.

[44] U. Norell. Dependently typed programming in agda. In *International school on advanced functional programming*, pages 230–266. Springer, 2008.

[45] M. V. Pedersen and S. Chong. Programming with flow-limited authorization: Coarser is better. In *4th IEEE European Symposium on Security and Privacy*, Piscataway, NJ, USA, June 2019. IEEE Press.

[46] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Trans. on Programming Languages and Systems*, 25(1), Jan. 2003.

[47] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003. URL `http://www.cs.cornell.edu/andru/papers/jsac/sm-jsac03.pdf`.

[48] F. B. Schneider, K. Walsh, and E. G. Sirer. Nexus Authorization Logic (NAL): Design rationale and applications. *ACM Trans. Inf. Syst. Secur.*, 14(1):8:1–8:28, June 2011. doi: 10.1145/1952982.1952990. URL `http://doi.acm.org/10.1145/1952982.1952990`.

[49] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in haskell. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, page 95–106, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450308601. doi: 10.1145/2034675.2034688. URL `https://doi.org/10.1145/2034675.2034688`.

[50] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing policy updates in security-typed languages. In *19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 202–216, July 2006. URL `http://www.cs.umd.edu/projects/PL/rx/rx.pdf`.

[51] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *16th ACM SIGPLAN Int'l Conf. on Functional Programming*, ICFP '11, pages 266–278, New York, NY, USA, 2011. ACM. URL `http://doi.acm.org/10.1145/2034773.2034811`.

[52] S. Tse and S. Zdancewic. Translating dependency into parametricity. In *9th ACM SIGPLAN Int'l Conf. on Functional Programming*, pages 115–125, 2004. doi: 10.1145/1016850.1016868. URL `http://doi.acm.org/10.1145/1016850.1016868`.

[53] P. Wadler. Propositions as types. *Communications of the ACM*, 2015.

[54] W. H. Winsborough and N. Li. Safety in automated trust negotiation. In *IEEE Symp. on Security and Privacy*, pages 147–160, May 2004. doi: 10.1109/SECPRI.2004.1301321. URL `http://dl.acm.org/citation.cfm?id=1178623`.

[55] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. ISSN 0890-5401. doi: 10.1006/inco.1994.1093. URL `http://dx.doi.org/10.1006/inco.1994.1093`.

[56] S. Zdancewic and A. C. Myers. Robust declassification. In *14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 15–23, June 2001. doi: 10.1109/CSFW.2001.930133. URL http://www.cs.cornell.edu/andru/papers/csfw01.pdf.

[57] S. Zdancewic and A. C. Myers. Secure information flow via linear continuations. *Higher-Order and Symbolic Computation*, 15(2–3):209–234, Sept. 2002. ISSN 1388-3690. doi: 10.1023/A:1020843229247. URL http://dx.doi.org/10.1023/A%3A1020843229247.

[58] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *ACM Trans. on Computer Systems*, 20(3):283–328, Aug. 2002. URL http://www.cs.cornell.edu/andru/papers/sosp01/spp-tr.pdf.

[59] L. Zheng and A. C. Myers. Dynamic security labels and noninterference. In *2nd Workshop on Formal Aspects in Security and Trust (FAST), IFIP TC1 WG1.7*. Springer, Aug. 2004. URL http://www.cs.cornell.edu/andru/papers/dynlabel.pdf.

A Proofs for Noninterference and Robust Declassification

A.1 FLAM and FLAC

We formalize the relation between FLAM and FLAC.

Lemma 12 (FLAC implies FLAM). *Let $\mathcal{H}(c) = \{\langle p \succcurlyeq q \mid \perp^{\rightarrow} \wedge \top^{\leftarrow}\rangle \mid \langle p \succcurlyeq q\rangle \in \Pi\}$. If $\Pi \Vdash p \succcurlyeq q$, then $\mathcal{H}; c; \perp^{\rightarrow} \wedge \top^{\leftarrow}; \perp^{\rightarrow} \wedge \top^{\leftarrow} \Vdash p \succcurlyeq q$.*

Proof. Proof is by induction on the derivation of the robust assumption $\Pi \Vdash p \succcurlyeq q$. Interesting case is R-ASSUME.

Case R-ASSUME: From the premises, we have that

$$\langle p \succcurlyeq q\rangle \in \Pi \tag{4}$$
$$\Pi; pc; \ell \Vdash \nabla(p^{\rightarrow}) \succcurlyeq \nabla(q^{\rightarrow}) \tag{5}$$

From (4) and DEL, we have that

$$\mathcal{H}; c; \perp^{\rightarrow} \wedge \top^{\leftarrow}; \perp^{\rightarrow} \wedge \top^{\leftarrow} \vdash p \succcurlyeq q$$

From [Weaken], we get that $\mathcal{H}; c; \wedge \nabla(q)) \vdash p \succcurlyeq q$. From the (5) and R-LIFT we thus have $\mathcal{H}; c; \perp^{\rightarrow} \wedge \top^{\leftarrow}; \perp^{\rightarrow} \wedge \top^{\leftarrow} \Vdash p \succcurlyeq q$.

Case R-Static: Since $\mathcal{L} \models p \succcurlyeq q$, we have from FLAM R-STATIC that $\mathcal{H}; c; \perp^{\rightarrow} \wedge \top^{\leftarrow}; \perp^{\rightarrow} \wedge \top^{\leftarrow} \Vdash p \succcurlyeq q$.

Case R-ConjR: Trivial.

Case R-DisjL: Trivial.

Case R-Trans: Trivial.

Case R-Weaken: Trivial.

$\square$

Lemma 13 (FLAM implies FLAC). *For a trust configuration such that, for all $n \neq c$, $\mathcal{H}(n) = \varnothing$, and for all $\langle p, q, \ell\rangle \in \mathcal{H}(c)$, $\ell = \perp^{\rightarrow} \wedge \top^{\leftarrow}$ and $\mathcal{H}; c; \perp^{\rightarrow} \wedge \top^{\leftarrow}; \perp^{\rightarrow} \wedge \top^{\leftarrow} \Vdash \nabla(p \rightarrow) \succcurlyeq \nabla(q \rightarrow)$, if $\mathcal{H}; c; \perp^{\rightarrow} \wedge \top^{\leftarrow}; \perp^{\rightarrow} \wedge \top^{\leftarrow} \Vdash p \succcurlyeq q$, then $\Pi \Vdash p \succcurlyeq q$.*

Proof Sketch. By induction on the FLAM derivation. Without loss of generality, we assume that the derivation of $\mathcal{H}; c; \perp^{\rightarrow} \wedge \top^{\leftarrow}; \top^{\rightarrow} \wedge \perp^{\leftarrow} \Vdash p \succcurlyeq q$ contains no applications of R-WEAKEN or R-FWD. Since all delegations are local to c, public, and trusted, they are unnecessary.

Next, observe that because delegations are public and trusted, any non-robust FLAM derivation may be lifted to a robust one by adding an application of FLAM's R-LIFT rule wherever a DEL rule occurs, and applying relevant robust rules in place of the non-robust rules. Rules corresponding to each non-robust rule are either part of the core robust rules or have been proven admissible [10]. The rule for R-TRANS is an exception, since it requires an additional premise to be satisfied, but since the query label is always $\perp^{\rightarrow} \wedge \top^{\leftarrow}$ this is trivially satisfied. $\square$

$$
\begin{aligned}
T \quad ::= \quad & [\cdot] \mid T\,e \mid e\,T \mid T\,\tau \mid \langle T, e\rangle \mid \langle e, T\rangle \mid \mathsf{proj}_i\,T \mid \mathsf{inj}_i\,T \mid \eta_\ell\,T \\
& \mid \quad \mathsf{bind}\ x = T\ \mathsf{in}\ e \mid \mathsf{bind}\ x = e\ \mathsf{in}\ T \mid \mathsf{assume}\ T\ \mathsf{in}\ e \mid \mathsf{assume}\ e\ \mathsf{in}\ T \\
& \mid \quad \mathsf{case}\ T\ \mathsf{of}\ \mathsf{inj}_1(x).\,e \mid \mathsf{inj}_2(x).\,e \mid \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inj}_1(x).\,T \mid \mathsf{inj}_2(x).\,e \\
& \mid \quad \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inj}_1(x).\,e \mid \mathsf{inj}_2(x).\,T \mid T\ \mathsf{where}\ v \mid e\ \mathsf{where}\ T \\
U \quad ::= \quad & \langle U, w\rangle \mid \langle w, U\rangle \mid \bar{\eta}_\ell\,U \mid \mathsf{inj}_i\,U \\
& \mid \quad \lambda(x{:}\tau)[pc].\,U \mid \Lambda X[pc].\,U
\end{aligned}
$$

Figure 13: Subterm language for expressions and values

A.2 Proofs for Type Preservation (Lemma 4)

Before proving type preservation, we define the grammar for sub terms that is used in proving the monotonicity of the program counter, and prove few supporting lemmas.

Every evaluation context can be represented as a subterm context but not the converse. The following lemma is useful to convert evaluation contexts into subterm contexts.

Lemma 14 (Evaluation Context implies Subterm Context). *For all E and e, if $\Pi; \Gamma; pc \vdash E[e] : \tau$ then exists T such that $T = E$ and $\Pi; \Gamma; pc \vdash E[e] : \tau$.*

Proof. Induction on the structure of E. ☐

Lemma 15 (Robust Assumption). *If $\Pi \Vdash pc \geqslant \nabla(b)$, then $\Pi \cdot \langle a \geqslant b\rangle \Vdash pc \geqslant \nabla(b)$ for any $a, b \in \mathcal{L}$.*

Proof. By inspection of the rules in Figure 5. ☐

Lemma 16 (Robust Protection). *If $\Pi \vdash pc \sqsubseteq \tau$, then $\Pi \cdot \langle a \geqslant b\rangle \vdash pc \sqsubseteq \tau$ for any $a, b \in \mathcal{L}$.*

Proof. By Lemma 15 and inspection of the rules in Figure 8. ☐

Monotnicity of *pc* is standard in many information flow control type systems. FLAC, too, has one.

Lemma 17 (Monotonicity of *pc*). *Let $\Pi; \Gamma; pc \vdash e : \tau$. If $e = T[e']$ such that for some $\Pi', \Gamma', pc', \Pi'; \Gamma'; pc' \vdash e' : \tau'$, then $\Pi \Vdash pc \sqsubseteq pc'$.*

Proof. Induction on the structure of the T. The interesting case is when $T = \mathsf{bind}\ x = e_1\ \mathsf{in}\ T'$. Consider the typing of $T[e']$; that is $\Pi; \Gamma; pc \vdash T[e'] : \tau$. From the typing rule BINDM, we have that $\Pi'; \Gamma, x{:}\tau'; pc \sqcup \ell \vdash e' : \tau$ (assuming $\Pi; \Gamma; pc \vdash e_1 : \ell\ \mathsf{says}\ \tau'$ for some ℓ). Here $pc' = pc \sqcup \ell$ and so $\Pi \Vdash pc \sqsubseteq pc'$. ☐

We need few helper lemmas to state the properties of delegation contexts.

Lemma 18. *If $\Pi \Vdash q \geqslant \nabla(t)$ and $\Pi \Vdash p \sqsubseteq q$ then $\Pi \Vdash p \geqslant \nabla(t)$.*

Proof. Follows from robust acts-for inference rules. ☐

Lemma 19 (Π Extension). *If $\Pi; \Gamma; pc \vdash e : \tau$ then $\Pi \cdot \langle p \geqslant q\rangle; \Gamma; pc \vdash e : \tau$ for any $p, q \in \mathcal{L}$.*

Proof. By Lemma 15, Lemma 16 and inspection of the typing rules. ☐

Lemma 20 (Monotonicity of Delegation Context). *If $\Pi; \Gamma; pc \vdash e : \tau$ then for all T, e' such that $T[e'] = e$, $\Pi'; \Gamma'; pc' \vdash e' : \tau'$ such that $\Pi \subseteq \Pi'$.*

Proof Sketch. The only sub terms that change the delegation context are assume and where. That is, $T = [\cdot]\ \mathsf{where}\ v$ or $T = \mathsf{assume}\ e\ \mathsf{in}\ [\cdot]$. However, they add delegations. Thus, $\Pi \subseteq \Pi'$. ☐

The following lemma is required to prove the type preservation. It says that an expression e well-typed at pc is still well-typed at a reduced pc'.

Lemma 21 (PC Reduction). *Let $\Pi; \Gamma; pc \vdash e : \tau$. For all pc, pc', such that $\Pi \Vdash pc' \sqsubseteq pc$ then $\Pi; \Gamma; pc' \vdash e : \tau$ holds.*

Proof. Proof is by induction on the derivation of the typing judgment.

Case VAR: Straightforward from the corresponding typing judgment.

Case UNIT: Straightforward from the corresponding typing judgment.

Case DEL: Straightforward from the corresponding typing judgment.

Case LAM: Straightforward from the corresponding typing judgment.

Case APP: Given, $\Pi; \Gamma; pc \vdash e\ e' : \tau$. From APP, we have

$$\Pi; \Gamma; pc \vdash e : \tau_1 \xrightarrow{pc''} \tau_2 \tag{6}$$

$$\Pi; \Gamma; pc \vdash e' : \tau_1 \tag{7}$$

$$\Pi \Vdash pc \sqsubseteq pc'' \tag{8}$$

Applying induction to the premises we have $\Pi; \Gamma; pc' \vdash e : \tau_1 \xrightarrow{pc''} \tau_2$ and $\Pi; \Gamma; pc' \vdash e' : \tau_1$. From R-TRANS, we have $\Pi \Vdash pc' \sqsubseteq pc''$. Hence we have all the premises.

Case TLAM: Straightforward from the corresponding typing judgment.

Case TAPP: Similar to App case.

Case PAIR: Straightforward from the corresponding typing judgment.

Case UNPAIR :   Straightforward from the corresponding typing judgment.

Case INJ: Straightforward from the corresponding typing judgment.

Case CASE: Straightforward from the corresponding typing judgment.

Case UNITM: Given $\Pi; \Gamma; pc \vdash \eta_\ell\ e : \tau$, by UNITM we have $\Pi \Vdash pc \sqsubseteq \ell$ and $\Pi; \Gamma; pc \vdash e : \tau$. By the induction hypothesis, we have $\Pi; \Gamma; pc' \vdash e : \tau$, and since $\Pi \Vdash pc' \sqsubseteq pc$, then by R-TRANS, we have $\Pi \Vdash pc' \sqsubseteq \ell$. Therefore by UNITM we have $\Pi; \Gamma; pc' \vdash \eta_\ell\ e : \tau$.

Case SEALED: Straightforward from the corresponding typing judgment.

Case BINDM: Given $\Pi; \Gamma; pc \vdash \mathsf{bind}\ x = e\ \mathsf{in}\ e' : \tau$, by BINDM we have

$$\Pi; \Gamma; pc \vdash e : \ell\ \mathsf{says}\ \tau' \tag{9}$$

$$\Pi; \Gamma, x : \tau'; pc \sqcup \ell \vdash e' : \tau \tag{10}$$

$$\Pi \vdash pc \sqcup \ell \sqsubseteq \tau \tag{11}$$

Since $\Pi \Vdash pc' \sqsubseteq pc$. By the monotonicity of join with respect to $\sqsubseteq$, we also have $\Pi \Vdash pc' \sqcup \ell \sqsubseteq pc \sqcup \ell$. Therefore, by the induction hypothesis applied to 9 and 10, we have

$$\Pi; \Gamma; pc' \vdash e : \ell\ \mathsf{says}\ \tau' \tag{12}$$

$$\Pi; \Gamma, x : \tau'; pc' \sqcup \ell \vdash e' : \tau \tag{13}$$

and by R-TRANS we get $\Pi \vdash pc \sqcup \ell \sqsubseteq \tau$. Then via BINDM we get

$$\Pi; \Gamma; pc' \vdash \mathsf{bind}\ x = e\ \mathsf{in}\ e' : \tau$$

Case ASSUME: Given, $\Pi; \Gamma; pc \vdash \mathsf{assume}\ e\ \mathsf{in}\ e' : \tau$, by ASSUME we have

$$\Pi; \Gamma; pc \vdash e : (p \succcurlyeq q) \tag{14}$$

$$\Pi \Vdash pc \succcurlyeq \nabla(q) \tag{15}$$

$$\Pi \Vdash \nabla(p^{\rightarrow}) \succcurlyeq \nabla(q^{\rightarrow}) \tag{16}$$

$$\Pi, \langle p \succcurlyeq q \rangle; \Gamma; pc \vdash e' : \tau \tag{17}$$

Applying induction hypothesis to (14) and (17) we have $\Pi; \Gamma; pc' \vdash e : (p \succcurlyeq q)$ and $\Pi, \langle p \succcurlyeq q \rangle; \Gamma; pc' \vdash e' : \tau$. Since $\Pi \Vdash pc' \sqsubseteq pc$, we have $\Pi \Vdash pc' \succcurlyeq \nabla(q)$. Combining, we have all the premises for ASSUME and thus $\Pi; \Gamma; pc' \vdash \mathsf{assume}\ e\ \mathsf{in}\ e' : \tau$

Case WHERE: Given, $\Pi; \Gamma; pc \vdash v$ where $e : \tau$, by WHERE we have

$$\Pi; \Gamma; pc \vdash v : (p \succcurlyeq q) \tag{18}$$
$$\Pi \Vdash \overline{pc} \succcurlyeq \nabla(q) \tag{19}$$
$$\Pi \Vdash \nabla(p^{\rightarrow}) \succcurlyeq \nabla(q^{\rightarrow}) \tag{20}$$
$$\Pi; \Gamma; pc \vdash e : \tau \tag{21}$$

Applying induction hypothesis to (18) and (21), we have $\Pi; \Gamma; pc' \vdash v : (p \succcurlyeq q)$ and $\Pi, \langle p \succcurlyeq q \rangle; \Gamma; pc' \vdash e : \tau$. Then by WHERE, we have $\Pi; \Gamma; pc' \vdash v$ where $e : \tau$.

Case BRACKET: The premise $\Pi \Vdash H^{\pi} \sqcup pc^{\pi} \sqsubseteq pc''^{\pi}$ implies $\Pi \Vdash H^{\pi} \sqcup pc'^{\pi} \sqsubseteq pc''^{\pi}$. The result follows from BRACKET.

Case BRACKET-VALUES: Applying induction to the premises gives the required conclusion.

$\square$

Lemma 22 (Values PC). *If $\Pi; \Gamma; pc \vdash w : \tau$, then $\Pi; \Gamma; pc' \vdash w : \tau$ for any pc'.*

Proof. By induction on the typing derivation of w. Observe that only APP, CASE, UNITM, BINDM, and ASSUME contain premises that constrain typing based on the judgment pc, and these rules do not apply to w terms. $\square$

We now prove a bunch of substitution lemmas. These are necessary whenever a program variable or a type variable is substituted.

Lemma 23 (Variable Substitution). *If $\Pi; \Gamma, x : \tau'; pc \vdash e : \tau$ and $\Pi; \Gamma; pc \vdash w : \tau'$, then $\Pi; \Gamma; pc \vdash e[x \mapsto w] : \tau$.*

Proof. Proof is by induction on the typing derivation of e. Observe that by Lemma 22 and Lemma 19, we have $\Pi'; \Gamma; pc' \vdash w : \tau'$ for any pc' and Π' such that $\Pi \subseteq \Pi'$. Therefore, each inductive case follows by straightforward application of the induction hypothesis. $\square$

Lemma 24 (Variable Substitution Under Contexts). *If $\Pi; \Gamma, x : \tau'; pc \vdash \langle\!| \, pc' \, |\!\rangle_e : \tau$ and $\Pi; \Gamma; pc \vdash w : \tau'$, then $\Pi; \Gamma; pc \vdash \langle\!| \, pc' \, |\!\rangle_{e[x \mapsto w]} : \tau$.*

Proof. Follows from Lemma 23. $\square$

Lemma 25 (Type Substitution). *Let τ' be well-formed in Γ, X, Γ'. If $\Pi; \Gamma, X, \Gamma'; pc \vdash e : \tau$ then $\Pi; \Gamma, \Gamma'[X \mapsto \tau']; pc \vdash e[X \mapsto \tau'] : \tau[X \mapsto \tau']$.*

Proof. Proof is by the induction on the typing derivation of $\Pi; \Gamma, X, \Gamma'; pc \vdash e : \tau$. $\square$

Lemma 26 (Projection Preserves Types). *If $\Pi; \Gamma; pc \vdash e : \tau$, then $\Pi; \Gamma; pc \vdash \lfloor e \rfloor_i : \tau$ for $i = \{1, 2\}$.*

Proof. Proof is by induction on the typing derivation of $\Pi; \Gamma; pc \vdash e : \tau$. The interesting case is $e = (e_1 \mid e_2)$. By BRACKET, we have $\Pi; \Gamma; pc' \vdash e_i : \tau$ for some pc' such that $\Pi \Vdash (H^{\pi} \sqcup pc^{\pi}) \sqsubseteq pc'^{\pi}$. Therefore, by Lemma 21, we have $\Pi; \Gamma; pc \vdash e_i : \tau$. $\square$

We expand the proof for the helper lemma necessary to prove the adequacy of bracketed terms.

Lemma 1 (Stuck expressions). *If e gets stuck then $\lfloor e \rfloor_i$ is stuck for some $i \in \{1, 2\}$.*

Proof. We prove by induction on the structure of e.

Case w: No reduction rules apply to terms in the syntactic category w (including $(w \mid w')$). Hence $\lfloor x \rfloor_i$ is stuck as well.

Case x: No reduction rules apply to a variable. Hence $\lfloor x \rfloor_i$ is stuck as well.

Case $(e_1 \mid e_2)$: By B-STEP, e is only gets stuck is if both e_1 and e_2 get stuck.

Case $e\,e'$**:** Since $e\,e'$ is stuck, then B-APP, W-APP, E-APP are not applicable. It follows that either (1) e is not of the form $(w \mid w')$, w where v, or $\lambda(x{:}\tau)[pc']. e$ or (2) e has the form $\lambda(x{:}\tau)[pc']. e$, but e' is stuck. For the first case, $\lfloor e \rfloor_i$ is also not of the form $(w \mid w')$, w where v, or $\lambda(x{:}\tau)[pc']. e$, so $\lfloor e\,e' \rfloor_i$ is also stuck. For the second case, applying the induction hypothesis gives us that $\lfloor e' \rfloor_i$ is stuck for some $i \in \{1,2\}$, so $\lfloor \lambda(x{:}\tau)[pc']. e\,e' \rfloor_i$ is stuck for the same i.

Case $e\,\tau$**:** Since $e\,e'$ is stuck, then B-TAPP, W-TAPP, E-TAPP are not applicable. It follows that e is not of the form $(w \mid w')$, w where v, or $\Lambda X[pc']. e$ Therefore, $\lfloor e \rfloor_i$ is also not of the form $(w \mid w')$, w where v, or $\lambda(x{:}\tau)[pc']. e$, so $\lfloor e\,e' \rfloor_i$ is also stuck.

Case $\eta_\ell\,e$**:** Since $\eta_\ell\,e$ is stuck, then E-UNITM is not applicable, so e does not have the form w. Therefore, E-UNITM is also not applicable to $\lfloor \eta_\ell\,e \rfloor_i$. Therefore, e must be stuck. Applying induction hypothesis, it follows that $\lfloor e \rfloor_i$ is also stuck and so $\lfloor \eta_\ell\,e \rfloor_i$ is also stuck.

Case $\mathbf{proj}_j\,e$**:** Similar to the above case.

Case $\mathbf{inj}_j\,e$**:** Similar to the above case.

Case $\langle e, e \rangle$**:** Similar to the above case.

Case $\mathbf{case}\,e\,\mathbf{of}\,\mathbf{inj}_1(x).\,e_1 \mid \mathbf{inj}_2(x).\,e_2$**:** Since B-CASE, W-CASE, and E-CASE are not applicable, it follows that e is not of the form $(w \mid w')$, w where v, or $\mathbf{inj}_j\,v$. It follows that $\lfloor \mathbf{case}\,e\,\mathbf{of}\,\mathbf{inj}_1(x).\,e_1 \mid \mathbf{inj}_2(x).\,e_2 \rfloor_i$ is also stuck.

Case $\mathbf{bind}\,x = v\,\mathbf{in}\,e'$**:** Similar to the above case.

Case $\mathbf{assume}\,e\,\mathbf{in}\,e'$**:** Similar to the above case.

Case $e\,\mathbf{where}\,v$**:** Similar to the above case.

$\square$

We are now ready to prove subject reduction.

Lemma 4 (Subject Reduction). *Let* $\Pi; \Gamma; pc \vdash e : \tau$. *If* $e \longrightarrow e'$ *then* $\Pi; \Gamma; pc \vdash e' : \tau$.

Proof. **Case E-APP:** Given $e = (\lambda(x{:}\tau)[pc'].\,e)\,w$ and $\Pi; \Gamma; pc \vdash (\lambda(x{:}\tau)[pc'].\,e)\,v : \tau_2$. From APP we have,

$$\Pi; \Gamma, x : \tau'; pc' \vdash e : \tau \tag{22}$$

$$\Pi; \Gamma; pc \vdash w : \tau' \tag{23}$$

$$\Pi \Vdash pc' \sqsubseteq pc \tag{24}$$

Therefore, via PC reduction (Lemma 21) and variable substitution (Lemma 23), we have that $\Pi; \Gamma; pc' \vdash e[x \mapsto w] : \tau$.

Case E-TAPP: Similar to above case, but using Lemma 25.

Case E-CASE1 : Given $e = \mathbf{case}\,(\mathbf{inj}_1\,w)\,\mathbf{of}\,\mathbf{inj}_1(x).\,e_1 \mid \mathbf{inj}_2(x).\,e_2$ and $e' = e_1[x \mapsto v]$. Also, $\Pi; \Gamma; pc \vdash \mathbf{case}\,(\mathbf{inj}_1\,w)\,\mathbf{of}\,\mathbf{inj}_1(x).\,e_1 \mid \mathbf{inj}_2(x).\,e_2 : \tau$. From the premises we have, $\Pi; \Gamma; pc \vdash \mathbf{inj}_1\,v : \tau' + \tau''$ and $\Pi; \Gamma, x : \tau'; pc \vdash e_1 : \tau$. From INJ, we have $\Pi; \Gamma; pc \vdash v : \tau'$. Invoking variable substitution lemma (Lemma 23), we have $\Pi; \Gamma; pc \vdash e_1[x \mapsto w] : \tau$.

Case E-CASE2 : Similar to above.

Case E-UNITM: Given $e = \eta_\ell\,w$ and $e' = \overline{\eta}_\ell\,w$. Also, $\Pi; \Gamma; pc \vdash \eta_\ell\,w : \ell\,\mathsf{says}\,\tau$. From the premises it follows that $\Pi; \Gamma; pc \vdash \overline{\eta}_\ell\,w : \ell\,\mathsf{says}\,\tau$.

Case E-BINDM: Given $e = \mathbf{bind}\,x = w\,\mathbf{in}\,e'$ and $e' = e'[x \mapsto w]$ Also, $\Pi; \Gamma; pc \vdash \mathbf{bind}\,x = w\,\mathbf{in}\,e' : \tau$. From the premises, we have the following:

$$\Pi; \Gamma; pc \vdash w : \tau' \tag{25}$$

$$\Pi; \Gamma, x : \tau'; pc \sqcup \ell \vdash e' : \tau \tag{26}$$

$$\Pi \vdash pc \sqcup \ell \sqsubseteq \tau \tag{27}$$

$$\Pi \Vdash p \succcurlyeq pc \tag{28}$$

We have to prove that $\Pi; \Gamma; pc \vdash e'[x \mapsto w] : \tau$. Since we have that $\Pi \Vdash p \succcurlyeq pc$, applying PC reduction (Lemma 21) to the premise (26), we have $\Pi; \Gamma, x : \tau'; pc \vdash e' : \tau$.

Invoking variable substitution lemma (Lemma 23), we thus have $\Pi; \Gamma; pc \vdash e'[x \mapsto w] : \tau$.

Case E-ASSUME: Given $e = \text{assume } \langle p \succcurlyeq q \rangle \text{ in } e'$ and $e' = e' \text{ where } \langle p \succcurlyeq q \rangle$. Also, $\Pi; \Gamma; pc \vdash \text{assume } \langle p \succcurlyeq q \rangle \text{ in } e' : \tau$. From ASSUME, we have

$$\Pi; \Gamma; pc \vdash \langle p \succcurlyeq q \rangle : (p \succcurlyeq q) \tag{29}$$

$$\Pi, \langle p \succcurlyeq q \rangle; \Gamma; pc \vdash e' : \tau \tag{30}$$

$$\Pi \Vdash pc \succcurlyeq \nabla(q) \tag{31}$$

$$\Pi \Vdash \nabla(pl^{\rightarrow}) \succcurlyeq \nabla(q^{\rightarrow}) \tag{32}$$

We need to prove:

$$\Pi; \Gamma; pc \vdash e' \text{ where } \langle p \succcurlyeq q \rangle : \tau$$

Comparing with the given premises, we already have the required premises.

$$\Pi; \Gamma; pc \vdash \langle p \succcurlyeq q \rangle : (p \succcurlyeq q) \tag{33}$$

$$\Pi, \langle p \succcurlyeq q \rangle; \Gamma; pc \vdash e : \tau \tag{34}$$

$$\Pi \Vdash \overline{pc} \succcurlyeq \nabla(q) \tag{35}$$

$$\Pi \Vdash \nabla(p^{\rightarrow}) \succcurlyeq \nabla(q^{\rightarrow}) \tag{36}$$

Hence proved.

Case E-EVAL: For $e = E[e]$ and $e' = E[e']$ where $\Pi; \Gamma; pc \vdash E[e] : \tau$, we have $\Pi'; \Gamma'; pc' \vdash e : \tau'$ for some pc', τ', Π', and Γ' such that $\Pi' \supseteq \Pi$, $\Gamma' \supseteq \Gamma$. By the induction hypothesis we have $\Pi'; \Gamma'; pc' \vdash e' : \tau'$. Observe that with the exception of SEALED and WHERE, the premises of all typing rules use terms from the syntactic category e. Therefore if a derivation for $\Pi; \Gamma; pc \vdash E[e] : \tau$ exists, it must be the case that derivation for $\Pi; \Gamma; pc \vdash E[e'] : \tau$ exists where the derivation of $\Pi'; \Gamma'; pc' \vdash e : \tau'$ is replaced with $\Pi'; \Gamma'; pc' \vdash e' : \tau'$. Rules SEALED and WHERE have premises that use terms from the syntactic category v, but since these are fully evaluated, e cannot be equal to a v term since no e' exists such that $v \longrightarrow^* e'$.

Case W-APP: Given $e = (w \text{ where } \langle p \succcurlyeq q \rangle) \, e$ and $e' = (w \, e) \text{ where } \langle p \succcurlyeq q \rangle$. We have to prove that

$$\Pi; \Gamma; pc \vdash (w \, e) \text{ where } \langle p \succcurlyeq q \rangle : \tau$$

From APP we have:

$$\Pi; \Gamma; pc \vdash w \text{ where } \langle p \succcurlyeq q \rangle : \tau_1 \xrightarrow{pc'} \tau \tag{37}$$

$$\Pi; \Gamma; pc \vdash e : \tau_1 \tag{38}$$

$$\Pi \Vdash pc \sqsubseteq pc' \tag{39}$$

$$\tag{40}$$

Rule WHERE gives us the following:

$$\Pi; \Gamma; pc \vdash \langle p \succcurlyeq q \rangle : (p \succcurlyeq q) \tag{41}$$

$$\Pi \Vdash \overline{pc} \succcurlyeq \nabla(q) \tag{42}$$

$$\Pi \Vdash \nabla(p^{\rightarrow}) \succcurlyeq \nabla(q^{\rightarrow}) \tag{43}$$

$$\Pi, \langle p \succcurlyeq q \rangle; \Gamma; pc \vdash w : \tau_1 \xrightarrow{pc'} \tau \tag{44}$$

We now want to show that e' is well typed via WHERE. The key premise is to show that the subexpression $(w \, e)$ is well-typed via APP. That is,

$$\Pi, \langle p \succcurlyeq q \rangle; \Gamma; pc \vdash w \, e : \tau \tag{45}$$

Applying Lemma 19 (extending delegation contexts for well-typed terms) to (38) and Lemma 15 (extending delegation contexts for assumptions) to (39), we have:

$$\Pi, \langle p \succcurlyeq q \rangle; \Gamma; pc \vdash e : \tau_1 \tag{46}$$

$$\Pi, \langle p \succcurlyeq q \rangle \Vdash pc \sqsubseteq pc' \tag{47}$$

Combining with (44), we have (45) which when combined with remaining premises ((41), (42) and (43)) give us $\Pi; \Gamma; pc \vdash (w \, e) \text{ where } \langle p \succcurlyeq q \rangle : \tau$.

Case W-TAPP: Given $e = (w \text{ where } \langle p \geqslant q \rangle) \, \tau$ and

$e' = (w \, \tau')$ where $\langle p \geqslant q \rangle$. We have to prove that

$$\Pi; \Gamma; pc \vdash (v \, \tau') \text{ where } \langle p \geqslant q \rangle : \tau[X \mapsto \tau']$$

From TAPP we have:

$$\Pi; \Gamma; pc \vdash w \text{ where } \langle p \geqslant q \rangle : \forall X[pc']. \tau \tag{48}$$

$$\Pi \Vdash pc \sqsubseteq pc' \tag{49}$$

Rule WHERE gives us the following:

$$\Pi; \Gamma; pc \vdash \langle p \geqslant q \rangle : (p \geqslant q) \tag{50}$$

$$\Pi \Vdash \overline{pc} \geqslant \nabla(q) \tag{51}$$

$$\Pi \Vdash \nabla(p^{\rightarrow}) \geqslant \nabla(q^{\rightarrow}) \tag{52}$$

$$\Pi, \langle p \geqslant q \rangle; \Gamma; pc \vdash v : \forall X[pc']. \tau \tag{53}$$

We now want to show that e' is well typed via WHERE. The key premise is to show that the subexpression $(v \, \tau')$ is well-typed via TAPP. That is,

$$\Pi, \langle p \geqslant q \rangle; \Gamma; pc \vdash w \, \tau' : \tau[X \mapsto \tau'] \tag{54}$$

Applying Lemma 15 (extending delegation context for well-typed terms) to (49), we get:

$$\Pi, \langle p \geqslant q \rangle \Vdash pc \sqsubseteq pc' \tag{55}$$

Combining with (53), we have (54) which when combined with remaining premises ((50), (51) and (52)) give $\Pi; \Gamma; pc \vdash (w \, \tau') \text{ where } \langle p \geqslant q \rangle : \tau[X \mapsto \tau']$.

Case W-UNPAIR: Given $e = \text{proj}_i \, (\langle w_1, w_2 \rangle \text{ where } \langle p \geqslant q \rangle)$ and $e' = (\text{proj}_i \, \langle w_1, w_2 \rangle) \text{ where } \langle p \geqslant q \rangle$. We have to prove that

$$\Pi; \Gamma; pc \vdash (\text{proj}_i \, \langle w_1, w_2 \rangle) \text{ where } \langle p \geqslant q \rangle : \tau_i$$

From UNPAIR , we have:

$$\Pi; \Gamma; pc \vdash \langle w_1, w_2 \rangle \text{ where } \langle p \geqslant q \rangle : \tau_1 \times \tau_2 \tag{56}$$

From (56) and WHERE, we have:

$$\Pi; \Gamma; pc \vdash \langle p \geqslant q \rangle : (p \geqslant q) \tag{57}$$

$$\Pi \Vdash \overline{pc} \geqslant \nabla(q) \tag{58}$$

$$\Pi \Vdash \nabla(p^{\rightarrow}) \geqslant \nabla(q^{\rightarrow}) \tag{59}$$

$$\Pi, \langle p \geqslant q \rangle; \Gamma; pc \vdash \langle w_1, w_2 \rangle : \tau_1 \times \tau_2 \tag{60}$$

From (60) and UNPAIR , we have: $\Pi, \langle p \geqslant q \rangle; \Gamma; pc \vdash \text{proj}_i \, \langle w_1, w_2 \rangle : \tau_i$. Combining with remaining premises ((57) to (60)) we have $\Pi; \Gamma; pc \vdash \text{proj}_i \, \langle w_1, w_2 \rangle \text{ where } \langle p \geqslant q \rangle : \tau_i$.

Case W-CASE: Given

$$e = \text{case } (w \text{ where } \langle p \geqslant q \rangle) \text{ of } \text{inj}_1(x). \, e_1 \mid \text{inj}_2(x). \, e_2$$

and

$$e' = (\text{case } w \text{ of } \text{inj}_1(x). \, e_1 \mid \text{inj}_2(x). \, e_2) \text{ where } \langle p \geqslant q \rangle$$

We have to prove that

$$\Pi; \Gamma; pc \vdash (\text{case } w \text{ of } \text{inj}_1(x). \, e_1 \mid \text{inj}_2(x). \, e_2) \text{ where } \langle p \geqslant q \rangle : \tau$$

From CASE we have:

$$\Pi; \Gamma; pc \vdash w \text{ where } \langle p \geqslant q \rangle : \tau_1 + \tau_2 \tag{61}$$

$$\Pi \vdash pc \sqcup \ell \sqsubseteq \tau \tag{62}$$

$$\Pi \Vdash pc \sqsubseteq \ell \tag{63}$$

$$\Pi; \Gamma, \, x : \tau_1; pc \sqcup \ell \vdash e_1 : \tau \tag{64}$$

$$\Pi; \Gamma, \, x : \tau_2; pc \sqcup \ell \vdash e_2 : \tau \tag{65}$$

From (61) and rule WHERE, we get the following:

$$\Pi; \Gamma; pc \vdash \langle p \geqslant q \rangle : (p \geqslant q) \tag{66}$$

$$\Pi \Vdash \overline{pc} \geqslant \nabla(q) \tag{67}$$

$$\Pi \Vdash \nabla(p^{\rightarrow}) \geqslant \nabla(q^{\rightarrow}) \tag{68}$$

$$\Pi; \Gamma; pc \vdash w : \tau_1 + \tau_2 \tag{69}$$

The key premise to prove is that:

$$\Pi, \langle p \geqslant q \rangle; \Gamma; pc \vdash (\texttt{case } w \texttt{ of inj}_1(x).\ e_1 \mid \texttt{inj}_2(x).\ e_2) : \tau$$

which follows from (69) and extending delegations in equations (62) to (65) (Lemma 19). Combining with remaining premises, we have $\Pi; \Gamma; pc \vdash (\texttt{case } w \texttt{ of inj}_1(x).\ e_1 \mid \texttt{inj}_2(x).\ e_2) \texttt{ where } \langle p \geqslant q \rangle : \tau$.

Case W-BINDM: Given $e = \texttt{bind } x = (w \texttt{ where } \langle p \geqslant q \rangle) \texttt{ in } e$ and $e' = (\texttt{bind } x = w \texttt{ in } e) \texttt{ where } \langle p \geqslant q \rangle$. We have to prove that

$$\Pi; \Gamma; pc \vdash (\texttt{bind } x = w \texttt{ in } e) \texttt{ where } \langle p \geqslant q \rangle : \tau$$

From BINDM we have:

$$\Pi; \Gamma; pc \vdash (w \texttt{ where } \langle p \geqslant q \rangle) : \ell \texttt{ says } \tau' \tag{70}$$

$$\Pi; \Gamma, x : \tau'; pc \sqcup \ell \vdash e : \tau \tag{71}$$

$$\Pi \vdash pc \sqcup \ell \sqsubseteq \tau \tag{72}$$

From (70) and rule WHERE, we get the following:

$$\Pi; \Gamma; pc \vdash \langle p \geqslant q \rangle : (p \geqslant q) \tag{73}$$

$$\Pi \Vdash \overline{pc} \geqslant \nabla(q) \tag{74}$$

$$\Pi \Vdash \nabla(p^{\rightarrow}) \geqslant \nabla(q^{\rightarrow}) \tag{75}$$

$$\Pi, \langle p \geqslant q \rangle; \Gamma; pc \vdash w : \ell \texttt{ says } \tau' \tag{76}$$

We now want to show that e' is well typed via WHERE. That is, we need the following premises,

$$\Pi, \langle p \geqslant q \rangle; \Gamma; pc \vdash \texttt{bind } x = w \texttt{ in } e : \tau \tag{77}$$

$$\Pi; \Gamma; pc \vdash \langle p \geqslant q \rangle : (p \geqslant q) \tag{78}$$

$$\Pi \Vdash \overline{pc} \geqslant \nabla(q) \tag{79}$$

$$\Pi \Vdash \nabla(p^{\rightarrow}) \geqslant \nabla(q^{\rightarrow}) \tag{80}$$

Extending the delegation context (Lemma 19) in the premises (71), (72) and from (76) we have (77).

We already have (78) from (73); (79) from (74); (80) from (75). Combining, we have $\Pi; \Gamma; pc \vdash (\texttt{bind } x = w \texttt{ in } e) \texttt{ where } \langle p \geqslant q \rangle : \tau$.

Case W-ASSUME: Given $e = \texttt{assume } w \texttt{ where } \langle a \geqslant b \rangle \texttt{ in } e$ and $e' = \texttt{assume } w \texttt{ in } e \texttt{ where } \langle a \geqslant b \rangle$. From ASSUME, we have

$$\Pi; \Gamma; pc \vdash v \texttt{ where } \langle a \geqslant b \rangle : (r \geqslant s) \tag{81}$$

$$\Pi, \langle r \geqslant s \rangle; \Gamma; pc \vdash e : \tau \tag{82}$$

$$\Pi \Vdash pc \geqslant \nabla(s) \tag{83}$$

$$\Pi \Vdash \nabla(r^{\rightarrow}) \geqslant \nabla(s^{\rightarrow}) \tag{84}$$

Expanding the first premise using WHERE, we have

$$\Pi; \Gamma; pc \vdash \langle a \geqslant b \rangle : (a \geqslant b) \tag{85}$$

$$\Pi, \langle a \geqslant b \rangle; \Gamma; pc \vdash v : (r \geqslant s) \tag{86}$$

$$\Pi \Vdash \overline{pc} \geqslant \nabla(b) \tag{87}$$

$$\Pi \Vdash \nabla(a^{\rightarrow}) \geqslant \nabla(b^{\rightarrow}) \tag{88}$$

We want to show

$$\Pi; \Gamma; pc \vdash \langle a \geqslant b \rangle : (a \geqslant b) \tag{89}$$

$$\Pi, \langle a \geqslant b \rangle; \Gamma; pc \vdash \texttt{assume } v \texttt{ in } e : \tau \tag{90}$$

$$\Pi \Vdash \overline{pc} \geqslant \nabla(b) \tag{91}$$

$$\Pi \Vdash \nabla(a^{\rightarrow}) \geqslant \nabla(b^{\rightarrow}) \tag{92}$$

Extending delegation context (Lemma 19) in the premises (82), (83), (88) and combining with topmost premise we have the (90). Remaining premises follow from (85), (87) and (88).

Case B-STEP: Given $e = (e_1 \mid e_2)$ and $e' = (e'_1 \mid e'_2)$. Also $\Pi; \Gamma; pc \vdash (e_1 \mid e_2) : \tau$. We have to prove

$$\Pi; \Gamma; pc \vdash (e'_1 \mid e'_2) : \tau$$

Without loss of generality, let $i = 1$. Thus from the premises of B-STEP, we have $e_1 \longrightarrow e'_1$ and $e'_2 = e_2$. Since sealed values can not take a step, inverting the well-typedness of bracket is only possible through BRACKET and not through BRACKET-VALUES. From BRACKET, we have

$$\Pi; \Gamma; pc' \vdash e_1 : \tau \tag{93}$$

$$\Pi; \Gamma; pc' \vdash e_2 : \tau \tag{94}$$

$$\Pi \Vdash (H^\pi \sqcup pc^\pi) \sqsubseteq pc'^\pi \tag{95}$$

$$\Pi \vdash H^\pi \sqsubseteq \tau^\pi \tag{96}$$

Since (93) holds, applying induction to the premise $e_1 \longrightarrow e'_1$, we have that $\Pi; \Gamma; pc' \vdash e'_1 : \tau$. Combining with remaining premises ((94) to (96)) we thus have that $\Pi; \Gamma; pc \vdash (e'_1 \mid e'_2) : \tau$.

Case B-APP: Given $e = (w_1 \mid w_2) \, w'$ and $e' = (w_1 \lfloor w' \rfloor_1 \mid w_2 \lfloor w \rfloor_2)$. Also given that $\Pi; \Gamma; pc \vdash (w_1 \mid w_2) \, w' : \tau_2$ is well-typed, from APP, we have the following:

$$\Pi; \Gamma; pc \vdash (w_1 \mid w_2) : \tau_1 \xrightarrow{pc''} \tau_2 \tag{97}$$

$$\Pi; \Gamma; pc \vdash w' : \tau_1 \tag{98}$$

$$\Pi \Vdash pc \sqsubseteq pc'' \tag{99}$$

Thus from BRACKET-VALUES, we have $\Pi \Vdash H^\pi \sqsubseteq (\tau_1 \xrightarrow{pc''} \tau_2)^\pi$. That is, from the definition of type projection (Figure 6), we have $\Pi \Vdash H^\pi \sqsubseteq \tau_1 \xrightarrow{pc''^\pi} \tau_2^\pi$. From P-FUN, we thus have

$$\Pi \Vdash H^\pi \sqsubseteq \tau_2^\pi \tag{100}$$

$$\Pi \Vdash H^\pi \sqsubseteq pc''^\pi \tag{101}$$

We need to prove

$$\Pi; \Gamma; pc \vdash (w_1 \lfloor w' \rfloor_1 \mid w_2 \lfloor w' \rfloor_2) : \tau_2$$

That is we need the following premises of BRACKET.

$$\Pi; \Gamma; pc' \vdash w_1 \lfloor w' \rfloor_1 : \tau_2 \tag{102}$$

$$\Pi; \Gamma; pc' \vdash w_2 \lfloor w' \rfloor_2 : \tau_2 \tag{103}$$

$$\Pi \Vdash H^\pi \sqcup pc^\pi \sqsubseteq pc'^\pi \tag{104}$$

$$\Pi \Vdash H^\pi \sqsubseteq \tau_2^\pi \tag{105}$$

Let $pc' = pc''$. We have (104) from (99) and (101). We already have (105) from (100). To prove (102), we need the following premises:

$$\Pi; \Gamma; pc' \vdash w_1 : \tau_1 \xrightarrow{pc''} \tau_2 \tag{106}$$

$$\Pi; \Gamma; pc' \vdash \lfloor w' \rfloor_1 : \tau_2 \tag{107}$$

$$\Pi \Vdash pc' \sqsubseteq pc'' \tag{108}$$

The last premise (108) holds trivially (from reflexivity). Applying Lemma 22 (values can be typed under any pc) to (97) we have (106). Applying Lemma 22 (values can be typed under any pc) and Lemma (26) (projection preserves typing) to (98) we have (107). Thus from APP, we have (102). Similarly, (103) holds. Hence proved.

Case B-TAPP: Similar to above (B-APP) case.

Case B-UNPAIR: Given $e = \text{proj}_i \, (\langle w_{11}, w_{12} \rangle \mid \langle w_{21}, w_{22} \rangle)$ and $e' = (w_{1i} \mid w_{2i})$. Also $\Pi; \Gamma; pc \vdash \text{proj}_i \, (\langle w_{11}, w_{12} \rangle \mid \langle w_{21}, w_{22} \rangle) : \tau_i$ We have to prove

$$\Pi; \Gamma; pc \vdash (w_{1i} \mid w_{2i}) : \tau_i$$

From UNPAIR , we have:

$$\Pi; \Gamma; pc \vdash (\langle w_{11}, w_{12} \rangle \mid \langle w_{21}, w_{22} \rangle) : \tau_1 \times \tau_2 \tag{109}$$

Since they are already values, they can be inverted using BRACKET-VALUES. This approach is more conservative.

$$\Pi; \Gamma; pc \vdash \langle w_{11}, w_{12} \rangle : \tau_1 \times \tau_2 \tag{110}$$

$$\Pi; \Gamma; pc \vdash \langle w_{21}, w_{22} \rangle : \tau_1 \times \tau_2 \tag{111}$$

$$\Pi \vdash H^\pi \sqsubseteq (\tau_1 \times \tau_2)^\pi \tag{112}$$

From (110), (111) and UNPAIR , we have $\Pi; \Gamma; pc \vdash w_{1i} : \tau_i$ and $\Pi; \Gamma; pc \vdash w_{2i} : \tau_i$ for $i = \{1, 2\}$. From (112), type projection (Figure 6) and P-PAIR, we have $\Pi \vdash H^\pi \sqsubseteq \tau_i^\pi$. Combining with other premises, $\Pi; \Gamma; pc \vdash (w_{1i} \mid w_{2i}) : \tau_i$ follows from BRACKET-VALUES.

Case B-BINDM: Given $e = \mathtt{bind}\, x = (\overline{\eta}_\ell\, w_1 \mid \overline{\eta}_\ell\, w_2)\, \mathtt{in}\, e$. We have that:

$$e' = (\mathtt{bind}\, x = \overline{\eta}_\ell\, w_1\, \mathtt{in}\, \lfloor e \rfloor_1 \mid \mathtt{bind}\, x = \overline{\eta}_\ell\, w_2\, \mathtt{in}\, \lfloor e \rfloor_2)$$

Also $\Pi; \Gamma; pc \vdash \mathtt{bind}\, x = (\overline{\eta}_\ell\, w_1 \mid \overline{\eta}_\ell\, w_2)\, \mathtt{in}\, e : \tau$. From BINDM, we have

$$\Pi; \Gamma; pc \vdash (\overline{\eta}_\ell\, w_1 \mid \overline{\eta}_\ell\, w_2) : \ell\, \mathsf{says}\, \tau' \tag{113}$$

$$\Pi; \Gamma, x : \tau'; pc \sqcup \ell \vdash e : \tau \tag{114}$$

$$\Pi \vdash pc \sqcup \ell \sqsubseteq \tau \tag{115}$$

From (113) and BRACKET-VALUES, we have

$$\Pi; \Gamma; pc \vdash \overline{\eta}_\ell\, w_1 : \ell\, \mathsf{says}\, \tau' \tag{116}$$

$$\Pi; \Gamma; pc \vdash \overline{\eta}_\ell\, w_2 : \ell\, \mathsf{says}\, \tau' \tag{117}$$

$$\Pi \Vdash H^\pi \sqsubseteq \ell^\pi \tag{118}$$

We have to prove that

$$\Pi; \Gamma; pc \vdash (\mathtt{bind}\, x = \overline{\eta}_\ell\, w_1\, \mathtt{in}\, \lfloor e \rfloor_1 \mid \mathtt{bind}\, x = \overline{\eta}_\ell\, w_2\, \mathtt{in}\, \lfloor e \rfloor_2) : \tau$$

For some $\widehat{pc}$ we need the following premises to satisfy BRACKET:

$$\Pi; \Gamma; \widehat{pc} \vdash \mathtt{bind}\, x = \overline{\eta}_\ell\, w_1\, \mathtt{in}\, \lfloor e \rfloor_1 : \tau \tag{119}$$

$$\Pi; \Gamma; \widehat{pc} \vdash \mathtt{bind}\, x = \overline{\eta}_\ell\, w_2\, \mathtt{in}\, \lfloor e \rfloor_2 : \tau \tag{120}$$

$$\Pi \Vdash (H^\pi \sqcup pc^\pi) \sqsubseteq \widehat{pc}^\pi \tag{121}$$

$$\Pi \vdash H^\pi \sqsubseteq \tau^\pi \tag{122}$$

A natural choice for $\widehat{pc}$ is $pc \sqcup \ell$. From Lemma 22 (values can be typed under any pc), we have

$$\Pi; \Gamma; \widehat{pc} \vdash \overline{\eta}_\ell\, w_i : \tau'$$

Applying Lemma 26 (bracket projection preserves typing) to (114), we have

$$\Pi; \Gamma, x : \tau'; \widehat{pc} \vdash \lfloor e \rfloor_i : \tau$$

From BINDM, we therefore have (119) and (120). Applying R-TRANS to (118) and (115), we have (122). Thus we have all required premises.

Case B-CASE: Does not occur. Not well-typed.

Case B-ASSUME: Does not occur. Not well-typed.

$\square$

A.3 Proof for Progress (Lemma 5)

Lemma 5 (Progress). *If* $\Pi; \varnothing; pc \vdash e : \tau$, *then either* $e \longrightarrow e'$ *or* e *is a where value.*

Proof. Proof is by induction on the derivation of the typing judgment.

Case Var: Does not occur as e is closed, and Γ is empty.

Case Unit: Already a value.

44

Case Del: Already a value.

Case Lam: Already a value.

Case TLam: Already a value.

Case App: Given $\Gamma; pc \vdash e\, e' : \tau$. From APP, we have the following

$$\Pi; \Gamma; pc \vdash e : (\tau_1 \xrightarrow{pc'} \tau_2) \tag{123}$$

$$\Pi; \Gamma; pc \vdash e' : \tau_1 \tag{124}$$

$$\Pi \Vdash pc \sqsubseteq pc' \tag{125}$$

Applying IH to (123) and (124), we have that e takes a step or is some value such that the type is $\tau_1 \xrightarrow{pc'} \tau_2$. Similarly, e' either takes a step or is some value w. When both e and e' are values, we have the following cases: $e = \lambda(x{:}\tau_1)[pc']. \, e'''$ or $e = \lambda(x{:}\tau_1)[pc']. \, e'''$ where v.

> **Case 1:** . Let $e = \lambda(x{:}\tau_1)[pc']. \, e'''$. Then by E-APP, we have that $\lambda(x{:}\tau_1)[pc']. \, e''' \; w \longrightarrow e'''[x \mapsto w]$.
>
> **Case 2:** Let $e = \lambda(x{:}\tau_1)[pc']. \, e'''$ where v. Then by W-APP, we have that $\lambda(x{:}\tau_1)[pc']. \, e'''$ where $v \; w \longrightarrow \lambda(x{:}\tau_1)[pc']. \, e''' \; w$ where v.

Case TApp: Similar to APP case except that the argument is a type and thus does not take a step by itself.

Case Pair: Applying IH to the premises of PAIR, we have that either e_i takes a step or is already a value for $i \in \{1, 2\}$.

Case UnPair Given $\Gamma; pc \vdash \text{proj}_i \, e : \tau$. Applying I.H. to the premises of UNPAIR , we have that either e takes a step or is already a value. If e is a value then from the well-typedness of e, we have that $e = \langle w_1, w_2 \rangle$ or $e = \langle w_1, w_2 \rangle$ where v. In the former case, $\text{proj}_i \, e$ takes a step according to E-UNPAIR . In the latter case, it takes a step according to W-UNPAIR.

Case Inj: Given $\Gamma; pc \vdash \text{inj}_i \, e : \tau$. Applying I.H. to the premises of INJI , we have that either e takes a step or is already a value. If e is a value we have that $\text{inj}_i \, e$ is also a value.

Case Case Given $\Gamma; pc \vdash \text{case } e \text{ of } \text{inj}_1(x). \, e_1 \mid \text{inj}_2(x). \, e_2 : \tau$. Applying I.H. we have that e either takes a step or is already some value. By well-typedness, we have that either $e = \text{inj}_i \, w$ or $e = \text{inj}_i \, w$ where v. In the former case, it takes a step following E-CASE. In the latter case, it takes a step according to W-CASE.

Case UnitM: Applying I.H. to the premises of UNITM we have that e either takes a step or is already a some value w. In the latter case, it takes a step as per E-UNITM.

Case Sealed: Already a value.

Case BindM: Applying I.H. to the premises of BINDM, e takes a step or is already a value. In the latter case, from the well-typedness of e we have that either $e = \overline{\eta}_\ell \, w$ or $e = \overline{\eta}_\ell \, w$ where v. In the former case, it takes a step according to E-BINDM. In the latter case, it takes a step as per W-BINDM.

Case Assume: Applying I.H. to the premises of ASSUME, we have that either e takes a step or is some value. In the latter case, from the well-typedness, we have that either $e = \langle p \succcurlyeq q \rangle$ or $e = \langle p \succcurlyeq q \rangle$ where v. In the former case, it takes a step according to E-ASSUME. In the latter case, it takes a step according to W-ASSUME.

Case Where: Applying I.H. to the premises of WHERE, we have that either e takes a step or is some value. In the latter case, the entire term is a value.

$\square$

A.4   Proofs for Erasure Conservation (Lemma 10)

To prove erasure conservation, we first prove that substituting program or type variable and evaluation context substitution preserves erasure.

Lemma 27 (Substitution preserves erasure). *Let* $\Pi; \Gamma, x : \tau'; pc \vdash e : \tau$ *and* $\Pi; \Gamma, pc \vdash w : \tau'$. *For all* pc', ℓ, *if* $\mathcal{O}(\lfloor e \rfloor_1, \Pi, \ell, \pi) = \mathcal{O}(\lfloor e \rfloor_2, \Pi, \ell, \pi)$ *and* $\mathcal{O}(\lfloor w \rfloor_1, \Pi, \ell, \pi) = \mathcal{O}(\lfloor w \rfloor_2, \Pi, \ell, \pi)$ *then* $\mathcal{O}(\lfloor e[x \mapsto w] \rfloor_1, \Pi, \ell, \pi) = \mathcal{O}(\lfloor e[x \mapsto w] \rfloor_2, \Pi, \ell, \pi)$.

Proof. Proof is by induction on the structure of the expression e.

Case Var: If $e = x$ then $\lfloor e[x \mapsto w] \rfloor_i = \lfloor w \rfloor_i$. We are already given that $\mathcal{O}(\lfloor w \rfloor_1, \Pi, \ell, \pi) = \mathcal{O}(\lfloor w \rfloor_2, \Pi, \ell, \pi)$. Unfolding the definition of $\mathcal{O}$ function, we thus have that $\mathcal{O}(\lfloor e[x \mapsto w] \rfloor_1, \Pi, \ell, \pi) = \mathcal{O}(\lfloor e[x \mapsto w] \rfloor_2, \Pi, \ell, \pi)$
If $e \neq x$ then $\lfloor (\!| e[x \mapsto w] |\!)_{pc'} \rfloor_i = \lfloor (\!| w |\!)_{pc'} \rfloor_i$. We are already given that $\mathcal{O}(\lfloor e \rfloor_1, \Pi, \ell, \pi) = \mathcal{O}(\lfloor e \rfloor_2, \Pi, \ell, \pi)$.

Case Unit: Substitution does not affect e.

Case $\langle p \succcurlyeq q \rangle$: Substitution does not affect e.

Case $\eta_{\ell'} \, e'$: We are given that $\mathcal{O}(\lfloor \eta_{\ell'} \, e' \rfloor_1, \Pi, \ell, \pi) = \mathcal{O}(\lfloor \eta_{\ell'} \, e' \rfloor_2, \Pi, \ell, \pi)$. We have to prove

$$\mathcal{O}(\lfloor \eta_{\ell'} \, e'[x \mapsto w] \rfloor_1, \Pi, \ell, \pi) = \mathcal{O}(\lfloor \eta_{\ell'} \, e'[x \mapsto w] \rfloor_2, \Pi, \ell, \pi)$$

This implies, from the projection definition, we have to prove

$$\mathcal{O}(\eta_{\ell'} \, \lfloor e'[x \mapsto w] \rfloor_1, \Pi, \ell, \pi) = \mathcal{O}(\eta_{\ell'} \, \lfloor e'[x \mapsto w] \rfloor_2, \Pi, \ell, \pi)$$

If ℓ' is higher than ℓ, it is trivial as both sides are holes. If not, we have to prove that

$$\overline{\eta}_{\ell'} \, \mathcal{O}(\lfloor e'[x \mapsto w] \rfloor_1, \Pi, \ell, \pi) = \overline{\eta}_{\ell'} \, \mathcal{O}(\lfloor e'[x \mapsto w] \rfloor_2, \Pi, \ell, \pi)$$

Our argument proceeds as follows. Applying the projection to the given terms, we have

$$\mathcal{O}(\lfloor \eta_{\ell'} \, e' \rfloor_i, \Pi, \ell, \pi) = \mathcal{O}(\eta_{\ell'} \, \lfloor e' \rfloor_i, \Pi, \ell, \pi)$$

Unfolding the definition of $\mathcal{O}$ function, the interesting case is when ℓ' is not higher (in lattice) than ℓ. That is,

$$\mathcal{O}(\eta_{\ell'} \, \lfloor e' \rfloor_i, \Pi, \ell, \pi) = \overline{\eta}_{\ell'} \, \mathcal{O}(\lfloor e' \rfloor_i, \Pi, \ell, \pi)$$

Applying I.H to e', we have that,

$$\mathcal{O}(\lfloor e'[x \mapsto w] \rfloor_1, \Pi, \ell, \pi) = \mathcal{O}(\lfloor e'[x \mapsto w] \rfloor_2, \Pi, \ell, \pi)$$

Thus,

$$\overline{\eta}_{\ell'} \, \mathcal{O}(\lfloor e'[x \mapsto w] \rfloor_1, \Pi, \ell, \pi) = \overline{\eta}_{\ell'} \, \mathcal{O}(\lfloor e'[x \mapsto w] \rfloor_2, \Pi, \ell, \pi)$$

Hence proved.

Case $\overline{\eta}_{\ell'} \, w'$: Similar to above case.

Other: The argument for other cases is a straight forward application of inductive hypothesis.

$\square$

Lemma 28 (Type substitution preserves erasure). *Let $\Pi; \Gamma, X; pc \vdash e' : \tau'$. For all pc', ℓ, if $\mathcal{O}(\lfloor e' \rfloor_1, \Pi, \ell, \pi) = \mathcal{O}(\lfloor e' \rfloor_2, \Pi, \ell, \pi)$ then $\mathcal{O}(\lfloor e'[X \mapsto \tau] \rfloor_1, \Pi, \ell, \pi) = \mathcal{O}(\lfloor e'[X \mapsto \tau] \rfloor_2, \Pi, \ell, \pi)$.*

Proof. Proof by inducting on the structure of expression. Moreover, $\mathcal{O}$ function does not erase types. $\square$

Lemma 29 (Evaluation Context substitution preserves erasure). *Let $\Pi; \Gamma; pc \vdash E[e] : \tau$. Then, $\mathcal{O}(\lfloor E[e] \rfloor_1, \Pi, \ell, \pi) = \mathcal{O}(\lfloor E[e] \rfloor_2, \Pi, \ell, \pi) \iff \mathcal{O}(\lfloor e \rfloor_1, \Pi, \ell, \pi) = \mathcal{O}(\lfloor e \rfloor_2, \Pi, \ell, \pi)$*

Proof. Consider the forward direction. We have to prove

$$\mathcal{O}(\lfloor E[e] \rfloor_1, \Pi, \ell, \pi) = \mathcal{O}(\lfloor E[e] \rfloor_2, \Pi, \ell, \pi) \implies \mathcal{O}(\lfloor e \rfloor_1, \Pi, \ell, \pi) = \mathcal{O}(\lfloor e \rfloor_2, \Pi, \ell, \pi)$$

Proof is by induction on the structure of the evaluation context.

Case •: Given $E = \bullet$ and so $E[e] = e$. Thus

$$\mathcal{O}(\lfloor E[e] \rfloor_1, \Pi, \ell, \pi) = \mathcal{O}(\lfloor E[e] \rfloor_2, \Pi, \ell, \pi) \implies \mathcal{O}(\lfloor e \rfloor_1, \Pi, \ell, \pi) = \mathcal{O}(\lfloor e \rfloor_2, \Pi, \ell, \pi)$$

Case $E\ e'$: Given

$$\mathcal{O}(\lfloor E[e]\ e'\rfloor_1, \Pi, \ell, \pi) = \mathcal{O}(\lfloor E[e]\ e'\rfloor_2, \Pi, \ell, \pi)$$

This implies,

$$\mathcal{O}(\lfloor E[e]\rfloor_1, \Pi, \ell, \pi) = \mathcal{O}(\lfloor E[e]\rfloor_2, \Pi, \ell, \pi) \tag{126}$$

$$\mathcal{O}(\lfloor e'\rfloor_1, \Pi, \ell, \pi) = \mathcal{O}(\lfloor e'\rfloor_2, \Pi, \ell, \pi) \tag{127}$$

From APP, we have $\Pi; \Gamma; pc \vdash E[e] : \tau_1 \xrightarrow{pc'} \tau_2$ and $\Pi; \Gamma; pc \vdash e' : \tau_1$. Using (126) applying I.H to the former, we have

$$\mathcal{O}(\lfloor e\rfloor_1, \Pi, \ell, \pi) = \mathcal{O}(\lfloor e\rfloor_2, \Pi, \ell, \pi) \tag{128}$$

Hence proved.

Other: Argument similar to previous cases follows.

$\square$

The root of a where term is the term obtained after peeling off all the outer delegations.

Definition 8 (Root term)**.**

$$\mathcal{R}(e) = \begin{cases} \mathcal{R}(e') & \text{if } e = e' \text{ where } v \\ e & \text{o.w} \end{cases}$$

We have the property that erasing a where term is equal to erasing its root.

Lemma 30 ($\mathcal{O}$ of where term)**.** *For all e, Π and p, $\mathcal{O}(e, \Pi, p, \pi) = \mathcal{O}(\mathcal{R}(e), \Pi, p, \pi)$*

Proof Sketch. Immediate from the definitions of $\mathcal{O}$ (Figure 10) and $\mathcal{R}(e)$ (Definition 8). $\square$

The following lemma is a sanity check on the correctness of the erasure function (Figure 10). Intuitively, two well-typed values should be observationally equivalent with respect to an observer that cannot observe secretsor untrusted values. In the below lemma, p^π is the observer and secret or untrusted data is labelled at H^π. This is crucial to proving the erasure conservation lemma.

Lemma 31 (Correctness of $\mathcal{O}$)**.** *For all Π', Γ, pc and $i \in \{1, 2\}$, if $\Pi'; \Gamma; pc \vdash w_i : \tau$ such that $\Pi \vdash H^\pi \sqsubseteq \tau^\pi$ and $\Pi \not\vdash H^\pi \sqsubseteq p^\pi$, then $\mathcal{O}(w_1, \Pi, p, \pi) = \mathcal{O}(w_2, \Pi, p, \pi)$.*

Proof. Proof is by induction on the structure of the value $(w_1 \mid w_2)$. We only show valid in which w_1 and w_2 has same type and structure.[22] Note that in all the below cases, the term $(w_1 \mid w_2)$ is well-typed by the typing rule BRACKET-VALUES.

Unit: Given $(w_1 \mid w_2) = (() \mid ())$. From the definition of $\mathcal{O}$ (Figure 10), we have $\mathcal{O}((), \Pi, p, \pi) = ()$, and hence $\mathcal{O}(\lfloor w\rfloor_1, \Pi, p, \pi) = \mathcal{O}(\lfloor w\rfloor_2, \Pi, p, \pi)$.

$\langle p \geqslant q \rangle$**:** Similar to above

UnitM: Given $(w_1 \mid w_2) = (\overline{\eta}_\ell\ w_1' \mid \overline{\eta}_\ell\ w_2')$ and $\Pi'; \Gamma; pc \vdash \overline{\eta}_\ell\ w_i' : \tau$ such that $\tau = \ell$ says τ'. From the definition of type projection in Figure 6, we have that $\Pi \vdash H^\pi \sqsubseteq \tau^\pi$ is $\Pi \vdash H^\pi \sqsubseteq \ell^\pi$ says τ'. From P-LBL, we thus have $\Pi \Vdash H^\pi \sqsubseteq \ell^\pi$. This combined with the given premise $\Pi \not\vdash H^\pi \sqsubseteq p^\pi$ gives $\Pi \not\vdash \ell^\pi \sqsubseteq p^\pi$. And so, $\mathcal{O}(w_i, \Pi, p, \pi) = \circ$.

Lam: Given $(w_1 \mid w_2) = (\lambda(x:\tau_1)[pc'].\ e_1 \mid \lambda(x:\tau_1)[pc'].\ e_2)$ such that $\tau = \tau_1 \xrightarrow{pc'} \tau_2$. Since we have $\Pi \vdash H^\pi \sqsubseteq (\tau_1 \xrightarrow{pc'} \tau_2)^\pi$, from the definition of type projection in Figure 6, it follows that $\Pi \vdash H^\pi \sqsubseteq \tau_1 \xrightarrow{pc'^\pi} \tau_2^\pi$. From P-FUN we have that $\Pi \Vdash H^\pi \sqsubseteq pc'^\pi$. This combined with the given premise $\Pi \not\vdash H^\pi \sqsubseteq p^\pi$ gives $\Pi \not\vdash pc'^\pi \sqsubseteq p^\pi$. From the definition of $\mathcal{O}$ (Definition 10), we have $\mathcal{O}(\lambda(x : \tau_1)[pc'].\ e_i, \Pi, p, \pi) = \circ$ if $\Pi \not\vdash pc'^\pi \sqsubseteq p^\pi$, and thus $\mathcal{O}(w_1, \Pi, p, \pi) = \mathcal{O}(w_2, \Pi, p, \pi)$.

---

[22]Ideally, the requirement that w_1 and w_2 have the same structure does not follow from the fact that w_1 and w_2 have same type, and thus should be stated in the lemma. For simplicity, we are not stating it. However, the callers of this lemma do maintain that property.

TLam: Similar to the above case.

Inj$i$: This is an invalid case since we have $(w_1 \mid w_2) = (\text{inj}_i\ w_1 \mid \text{inj}_i\ w_2)$, however, w cannot be well-typed.

Pair: Given $(w_1 \mid w_2) = (\langle w_{11}, w_{12} \rangle \mid \langle w_{21}, w_{22} \rangle)$. We have that $\Pi'; \Gamma; pc \vdash \langle w_{i1}, w_{i2} \rangle : \tau_1 \times \tau_2$. We have to prove

$$\mathcal{O}(\langle w_{11}, w_{12} \rangle, \Pi, p, \pi) = \mathcal{O}(\langle w_{21}, w_{22} \rangle, \Pi, p, \pi)$$

It suffices to prove the following.

$$\mathcal{O}(w_{11}, \Pi, p, \pi) = \mathcal{O}(w_{21}, \Pi, p, \pi)$$
$$\mathcal{O}(w_{12}, \Pi, p, \pi) = \mathcal{O}(w_{22}, \Pi, p, \pi)$$

Consider the terms from the pair projection w_{11} and w_{21}, and w_{12} and w_{22}. Since these are well-typed from the typing rule PAIR, applying I.H. using the corresponding typing judgments of the terms yields the required proof.

Where: We have $(w_1 \mid w_2) = (w_1'\ \text{where}\ v_1 \mid w_2'\ \text{where}\ v_2)$. From the definition of $\mathcal{O}$ (Definition 10), we have that $\mathcal{O}(w_i\ \text{where}\ v_i, \Pi, p, \pi) = \mathcal{O}(w_i, \Pi, p, \pi)$. We have that the type of w_i (i.e., τ) is protected.

Let the root value of w_i be v_i'. Then from the definition of $\mathcal{R}(w_i)$ and Lemma 30, we have that

$$\mathcal{O}(w_i, \Pi, p, \pi) = \mathcal{O}(v_i', \Pi, p, \pi)$$

Also, the type of v_i' is protected w.r.t Π (given from BRACKET-VALUES). Note that applying $\mathcal{O}$ on v_i' yields a hold. The argument uses induction on the structure of values.

We now proceed with the case analysis on the structure of the values. That is, v_i' is either $()$, $\langle r \geqslant t \rangle$, $\lambda(x{:}\tau_1)[pc']. e'$, $\Lambda X[pc']. e'$, or $\langle w_{i1}', w_{i2}' \rangle$. Applying $\mathcal{O}$ function to the all the values except the pair, yields a hole.

For the pair, we invoke the inductive argument similar to the one proved above. We only sketch the high-level proof: the root values for w_{i1}' and w_{i2}' are protected and are thus erased yielding a hole.

Since $\mathcal{O}(w_i\ \text{where}\ v_i, \Pi, p, \pi) = \mathcal{O}(w_i, \Pi, p, \pi) = \mathcal{O}(v_i', \Pi, p, \pi) = \circ$, we have the required proof that $\mathcal{O}(w_1\ \text{where}\ v_1, \Pi, p, \pi) = \mathcal{O}(w_i\ \text{where}\ v_2, \Pi, p, \pi)$.

Hence proved. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ □

The following three lemmas are useful for establishing the properties of where propagation.

Lemma 32 (Correctness of Where propagation). *Let $\Pi; \Gamma, x : \tau'; pc \vdash e : \tau$ and $\Pi; \Gamma; pc \vdash w : \tau'$ such that $\Pi \vdash H^{\rightarrow} \wedge \top^{\leftarrow} \sqsubseteq \tau'$ and $\langle r \geqslant t \rangle \notin e$. If $e[x \mapsto w] \longrightarrow w'\ \text{where}\ \langle r \geqslant t \rangle$, then $\Pi \vdash H^{\rightarrow} \wedge \top^{\leftarrow} \sqsubseteq \tau$.*

Proof. Proof is by induction on the structure of e.

Var: Not a valid expression, since x is substituted with a value, it cannot take a step. However, note that $\Pi; \Gamma, x : \tau'; pc \vdash x : \tau$ implies $\tau = \tau'$ and thus we have $\Pi \vdash H^{\rightarrow} \sqsubseteq \tau^{\rightarrow}$.

App: Given $e[x \mapsto w\ \text{where}\ \langle r \geqslant t \rangle] \longrightarrow w'\ \text{where}\ \langle r \geqslant t \rangle$ such that $e = e_1\ e_2$. From the relevant rules E-EVAL, E-APP and W-APP, only W-APP is possible. It implies that $e_1 = x = w''\ \text{where}\ \langle r \geqslant t \rangle$ such that $\tau' = \tau_1 \xrightarrow{pc'} \tau_2$ and $\tau = \tau_2$. Since $\Pi \vdash H^{\rightarrow} \sqsubseteq \tau'$, from P-FUN , we have $\Pi \vdash H^{\rightarrow} \sqsubseteq \tau_2$. Hence proved.

Proj: Given $e[x \mapsto w\ \text{where}\ \langle r \geqslant t \rangle] \longrightarrow w'\ \text{where}\ \langle r \geqslant t \rangle$ such that $e = \text{proj}_i\ e'$. From the relevant rules E-EVAL, E-PROJ and W-PROJ , only W-PROJ is relevant. Thus $e' = x$ and $e = \text{proj}_i\ x$ and $e'[x \mapsto w'\ \text{where}\ \langle r \geqslant t \rangle] = \text{proj}_i\ w'\ \text{where}\ \langle r \geqslant t \rangle$ such that $\tau' = \tau_1 \times \tau_2$ and $\tau = \tau_i$ for $i \in \{1, 2\}$. Since $\Pi \vdash H^{\rightarrow} \sqsubseteq \tau'$, from rule P-PROD , we have that Since $\Pi \vdash H^{\rightarrow} \sqsubseteq \tau_i$. Hence proved.

Case: Given $e[x \mapsto w\ \text{where}\ \langle r \geqslant t \rangle] \longrightarrow w'\ \text{where}\ \langle r \geqslant t \rangle$ such that such that

$$e = \text{case}\ e'\ \text{of}\ \text{inj}_1(y).\ e_1 \mid \text{inj}_2(y).\ e_2$$

From the relevant rules E-EVAL, E-CASE and W-CASE, only W-CASE is relevant. Thus

$$e = (\text{case}\ x\ \text{of}\ \text{inj}_1(y).\ e_1 \mid \text{inj}_2(y).\ e_2)$$

and
$$e[x \mapsto w' \text{ where } \langle r \geqslant t \rangle] = (\text{case } (w'' \text{ where } \langle r \geqslant t \rangle) \text{ of } \text{inj}_1(y).\ e_1 \mid \text{inj}_2(y).\ e_2)$$
Not a valid case since x has to be of sum type, and sum type cannot be protected.

BindM: Given $e[x \mapsto w \text{ where } \langle r \geqslant t \rangle] \longrightarrow w' \text{ where } \langle r \geqslant t \rangle$ such that such that $e = (\text{bind } y = e_1 \text{ in } e_2)$. From the relevant rules E-EVAL, E-BINDM and W-BINDM, only W-BINDM is relevant. Thus $e = (\text{bind } y = x \text{ in } e_2)$ and $e[x \mapsto w \text{ where } \langle r \geqslant t \rangle] = (\text{bind } y = (w \text{ where } \langle r \geqslant t \rangle) \text{ in } e_2)$. Without loss of generality, assume that $\tau' = \ell \text{ says } \tau''$. Then, from BINDM, we have that $\Pi \vdash pc \sqcup \ell \sqsubseteq \tau$. Since $\Pi \vdash H^\rightarrow \wedge \top^\leftarrow \sqsubseteq \tau'$ it implies that $\Pi \vdash H^\rightarrow \wedge \top^\leftarrow \sqsubseteq \tau$. Hence proved.

Assume: Given $e[x \mapsto w \text{ where } \langle r \geqslant t \rangle] \longrightarrow w' \text{ where } \langle r \geqslant t \rangle$ such that such that $e = (\text{assume } e_1 \text{ in } e_2)$. From the relevant rules E-EVAL, E-ASSUME and W-ASSUME, only W-ASSUME is relevant. Thus $e = (\text{assume } x \text{ in } e_1)$ and $e[x \mapsto w' \text{ where } \langle r \geqslant t \rangle] = (\text{assume } (w'' \text{ where } \langle r \geqslant t \rangle) \text{ in } e_1)$. From ASSUME we have $\tau = \tau'$. Hence proved.

Other: Not valid cases.

$\square$

Lemma 33 (Protected Where terms)**.** *Let* $\Pi; \Gamma, x : \tau'; pc \vdash e_0 : \tau$ *and* $\Pi; \Gamma; pc \vdash w \text{ where } v : \tau'$ *such that* $\Pi \vdash H^\rightarrow \wedge \top^\leftarrow \sqsubseteq \tau'$, $v = \langle r \geqslant t \rangle$, $v \notin e_0$ *and* $\Pi \nvdash pc \geqslant \nabla(t)$. *If* $e_0[x \mapsto w \text{ where } v] \longrightarrow^* w' \text{ where } v$, *then* $\Pi \vdash H^\pi \sqsubseteq \tau^\pi$.

Proof. Without loss of generality, let

1. $e_0[x \mapsto w \text{ where } v] \longrightarrow^* E[e]$ such that $e = T[w \text{ where } v]$

2. $E[e] \longrightarrow E[e' \text{ where } v]$

3. $E[e' \text{ where } v] \longrightarrow^* E'[w' \text{ where } v]$

4. $E'[w' \text{ where } v] \longrightarrow w' \text{ where } v$

Consider item 1. Since e_0 and $w \text{ where } v$ are well typed, by variable substitution (Lemma 23) so is $e_0[x \mapsto w \text{ where } v]$. Then, by subject reduction (Lemma 4), we know that $E[e]$ is well-typed. Therefore, e is also well-typed (for some context and type), thus $\Pi'; \Gamma'; pc' \vdash e : \tau''$.

Consider item 2. We are given $\Pi \vdash H^\pi \sqsubseteq \tau'^\pi$, and so $\Pi' \vdash H^\rightarrow \wedge \top^\leftarrow \sqsubseteq \tau''$ Also, applying subject reduction(Lemma 4), we have $\Pi'; \Gamma'; pc' \vdash e' \text{ where } v : \tau''$. Applying Lemma 32, we have that $\Pi'; \Gamma'; pc' \vdash e' \text{ where } v : \tau''$ such that $\Pi' \vdash H^\rightarrow \wedge \top^\leftarrow \sqsubseteq \tau''$. (Note that $\Pi \subseteq \Pi'$.)

Consider item 3. From subject reduction (Lemma 4), we have that $\Pi''; \Gamma''; pc'' \vdash w' \text{ where } v : \tau'''$.

Consider item 4. From subject reduction (Lemma 4), we have that $\Pi; \Gamma; pc \vdash w' \text{ where } v : \tau$. Thus $\Pi'' = \Pi, \Gamma'' = \Gamma, pc'' = pc'$. Also, $E' = [\cdot]$ or $E' = (\!| [\cdot] |\!)_{pc''}$. This is because, there are no more W-* steps.

Consider item 3 again. We have that $\Pi; \Gamma; pc \vdash w' \text{ where } v : \tau'$. Repeatedly applying Lemma 32, we have that $\Pi \vdash H^\rightarrow \wedge \top^\leftarrow \sqsubseteq \tau'$. Hence proved. $\square$

However, what happens if the type of $w \text{ where } v$ (from S) is not protected? The following lemma states that even in such cases, propagation of v ensures that the type of wrapped term is protected. The key insight is that $w \text{ where } v$ must be a subterm of a protected expression, and operating on protected expressions occurs only in protected contexts.

Lemma 34 (Protected Where terms-2)**.** *Let* $\Pi; \Gamma, x : \tau'; pc \vdash e_0 : \tau$ *and* $\Pi; \Gamma; pc \vdash U[w \text{ where } v] : \tau'$ *such that the following hold.*

1. τ' *is protected. That is,* $\Pi \vdash H^\rightarrow \wedge \top^\leftarrow \sqsubseteq \tau'$

2. *the type of* $w \text{ where } v$ *is not protected. That is,* $\Pi'; \Gamma'; pc' \vdash w \text{ where } v : \tau''$ *such that* $\Pi' \vdash H^\rightarrow \wedge \top^\leftarrow \sqsubseteq \tau''$, *for some* Π', Γ' *and* pc'.

3. $v = \langle r \geqslant t \rangle$ *for some* r *and* t, *and* $v \notin e_0$

    4. $\Pi \not\vdash pc \geqslant \nabla(t)$

If $e_0[x \mapsto U[w \text{ where } v]] \longrightarrow^* w' \text{ where } v$, *then* $\Pi \vdash H^\rightarrow \wedge \top^\leftarrow \sqsubseteq \tau$.

Proof. Since the type of w where v is unprotected but is a subterm of a term with protected type, the only possibility is $U[w \text{ where } v] = \eta_\ell U'[w \text{ where } v]$ such that ℓ protects $H^\rightarrow \wedge \top^\leftarrow$ or $U[w \text{ where } v] = \lambda(y{:}\tau_1)[\ell].\,U'[w \text{ where } v]$ such that $\tau_1 \xrightarrow{\ell} \tau_2$ protects $H^\rightarrow \wedge \top^\leftarrow$. Then, we have the following two possibilities.

1. (a) $e_0[x \mapsto U[w \text{ where } v]] \longrightarrow^* E[\text{bind } y = \eta_\ell U'[w \text{ where } v] \text{ in } e]$
    (b) $E[\text{bind } y = \eta_\ell U'[w \text{ where } v] \text{ in } e'] \longrightarrow E[(\!| e'[y \mapsto U'[w \text{ where } v]] \,|\!)_\ell]$
    (c) $E[(\!| e'[y \mapsto U'[w \text{ where } v]] \,|\!)_\ell] \longrightarrow^* E[(\!| w'' \text{ where } v \,|\!)_\ell]$
    (d) $E[(\!| w'' \text{ where } v \,|\!)_\ell] \longrightarrow E[w'' \text{ where } v]$
    (e) $E[w'' \text{ where } v] \longrightarrow^* E'[w' \text{ where } v]$
    (f) $E'[w' \text{ where } v] \longrightarrow w' \text{ where } v$

    Consider item 1b. From the typing rule BINDM, we have that the type of $e'[y \mapsto U'[w \text{ where } v]]$ protects ℓ, and thus protects $H^\rightarrow \wedge \top^\leftarrow$.

    Consider 1c. Applying subject reduction (Lemma 4), we thus have that the type of w'' where v is protected.

    Consider steps from 1d to 1f. Invoking Lemma 33, we have that the type of w' where v is protected.

2. (a) $e_0[x \mapsto U[w \text{ where } v]] \longrightarrow^* E[\lambda(y{:}\tau_1)[\ell].\,U'[w \text{ where } v] \; w_2]$
    (b) $E[\lambda(y{:}\tau_1)[\ell].\,U'[w \text{ where } v] \; w_2] \longrightarrow E[(\!| U'[w \text{ where } v][y \mapsto w_2] \,|\!)_\ell]$
    (c) $E[(\!| U'[w \text{ where } v][y \mapsto w_2] \,|\!)_\ell] \longrightarrow^* E[(\!| w'' \text{ where } v \,|\!)_\ell]$
    (d) $E[(\!| w'' \text{ where } v \,|\!)_\ell] \longrightarrow E[w'' \text{ where } v]$
    (e) $E[w'' \text{ where } v] \longrightarrow^* E'[w' \text{ where } v]$
    (f) $E'[w' \text{ where } v] \longrightarrow w' \text{ where } v$

    Consider item 2a. From the typing rule APP, we have that the type of $U'[w \text{ where } v]$ (in which y is free) is τ_2. From variable substitution (Lemma 23), the type of $U'[w \text{ where } v][y \mapsto w_2]$ is τ_2.

    Consider item 2b. We are given that $\tau_1 \xrightarrow{\ell} \tau_2$ protects $H^\rightarrow \wedge \top^\leftarrow$. From P-FUN, this implies τ_2 protects $H^\rightarrow \wedge \top^\leftarrow$. That is, the type of $U'[w \text{ where } v][y \mapsto w_2]$ protects $H^\rightarrow \wedge \top^\leftarrow$.

    Consider 2c. Applying subject reduction (Lemma 4) to the evaluation steps inside the evaluation context E, we thus have that the type of w'' where v protects $H^\rightarrow \wedge \top^\leftarrow$.

    Consider steps from 2d to 2f. Invoking Lemma 33, we have that the type of w' where v protects $H^\rightarrow \wedge \top^\leftarrow$.

Hence proved.   □

Lemma 35 ($\mathcal{O}$ is preserved under monotonic Π). *If* $\mathcal{O}(w_1, \Pi, v, p, \pi) = \mathcal{O}(w_2, \Pi, v, p, \pi)$ *then* $\mathcal{O}(w_1, \Pi, p, \pi) = \mathcal{O}(w_2, \Pi, p, \rightarrow)$.

Proof Sketch. Induction on the cases of $\mathcal{O}$. Attacker p cannot enable more information flows under Π than under Π, v. Thus, from the definition of $\mathcal{O}$, in each case terms erased to holes under Π, v still get erased to holes under Π.   □

We now present the proof of erasure conservation for confidentiality (Lemma 10): evaluation step preserves erasure with respect to the confidentiality projections of an observer. An analogous lemma holds for integrity.

Lemma 10 (Confidentiality Erasure Conservation). *Suppose* $\Pi; \Gamma; pc \vdash e : \tau$ *and let* S *be a well-typed substitution of* e *for* Γ *in* Π. *Then for some* H *and* ℓ *such that* $\Pi \not\Vdash \ell^\rightarrow \geqslant H^\rightarrow$ *and* $\Pi \not\Vdash pc \geqslant \nabla(H^\rightarrow - \ell^\rightarrow)$, *if* $\mathcal{O}(\lfloor e\,S \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor e\,S \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$ *and* e *is a source-level term, and for all entries* $[y \mapsto w_y] \in S$ *with* $\Pi; \Gamma; pc \vdash w_y : \Gamma(y)$, *either* w_y *is a source-level term or* $\Pi \Vdash H^\rightarrow \wedge \top^\leftarrow \sqsubseteq \Gamma(y)$ *then* $(e\,S) \longrightarrow e'$ *implies* $\mathcal{O}(\lfloor e' \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor e' \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$.

Proof. By induction on the evaluation of e, using the definition of $\mathcal{O}$ (Figure 10) and delegation invariance (Lemma 8).

Case E-APP*: We have $\mathcal{O}(\lfloor \lambda(x{:}\tau)[pc'].\, e'\; w \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor \lambda(x{:}\tau)[pc'].\, e'\; w \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$. We have to show that $\mathcal{O}(\lfloor (\!| \, e'[x \mapsto w] \, |\!)_{pc'} \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor (\!| \, e'[x \mapsto w] \, |\!)_{pc'} \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$. Depending on the observability of pc', we have two subcases.

> **Case $\Pi \Vdash \ell^\rightarrow \geqslant pc'^\rightarrow$:** We have $\mathcal{O}(\lfloor e' \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor e' \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$ and $\mathcal{O}(\lfloor w \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor w \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$ Since substitution under context preserves observation function (Lemma 27), we thus have $\mathcal{O}(\lfloor (\!| \, e'[x \mapsto w] \, |\!)_{pc'} \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor (\!| \, e'[x \mapsto w] \, |\!)_{pc'} \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$.
>
> **Case $\Pi \nVdash \ell^\rightarrow \geqslant pc'^\rightarrow$:** We have to show that if $\mathcal{O}(\lfloor \lambda(x:\tau)[pc'].\, e' \rfloor_i, \Pi, \ell^\rightarrow, \rightarrow) = \circ$ then $\mathcal{O}(\lfloor (\!| \, e'[x \mapsto w] \, |\!)_{pc'} \rfloor_i, \Pi, \ell^\rightarrow, \rightarrow) = \circ$. This holds by the definition of $\mathcal{O}$ since $\mathcal{O}(\lfloor \lambda(x:\tau)[pc'].\, e' \rfloor_i, \Pi, \ell^\rightarrow, \rightarrow) = \circ$ implies that $\Pi \nVdash \ell^\rightarrow \geqslant pc'^\rightarrow$.

Case E-TAPP*: Similar to the E-APP* case but using Lemma 28.

Case E-BINDM*: Given $\mathsf{bind}\; x = \overline{\eta}_{\ell'}\; w\; \mathsf{in}\; e' \longrightarrow (\!| \, e'[x \mapsto w] \, |\!)_{\ell'}$. Also given,

$$\mathcal{O}(\lfloor \mathsf{bind}\; x = \overline{\eta}_{\ell'}\; w\; \mathsf{in}\; e' \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor \mathsf{bind}\; x = \overline{\eta}_{\ell'}\; w\; \mathsf{in}\; e' \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$$

> **Case $\Pi \Vdash \ell^\rightarrow \geqslant \ell'^\rightarrow$:** We have $\mathcal{O}(\lfloor e' \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor e' \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$ and $\mathcal{O}(\lfloor w \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor w \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$. Since substitution under context preserves observation function (Lemma 27), we thus have $\mathcal{O}(\lfloor (\!| \, e'[x \mapsto w] \, |\!)_{\ell'} \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor (\!| \, e'[x \mapsto w] \, |\!)_{\ell'} \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$.
>
> **Case $\Pi \nVdash \ell^\rightarrow \geqslant \ell'^\rightarrow$:** It remains to show that if $\mathcal{O}(\lfloor \overline{\eta}_{\ell'}\; w \rfloor_i, \Pi, \ell^\rightarrow, \rightarrow) = \circ$ then $\mathcal{O}(\lfloor (\!| \, e'[x \mapsto w] \, |\!)_{\ell'} \rfloor_i, \Pi, \ell^\rightarrow, \rightarrow) = \circ$. This holds by the definition of $\mathcal{O}$ since $\mathcal{O}(\lfloor \overline{\eta}_{\ell'}\; w \rfloor_i, \Pi, \ell^\rightarrow, \rightarrow) = \circ$ implies that $\Pi \nVdash \ell^\rightarrow \geqslant \ell'^\rightarrow$.

Case O-CTX: Given $(\!| \, w \, |\!)_{\ell'} \longrightarrow w$. Also given $\mathcal{O}(\lfloor (\!| \, w \, |\!)_{\ell'} \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor (\!| \, w \, |\!)_{\ell'} \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$. We have to prove that

$$\mathcal{O}(\lfloor w \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor w \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$$

We have two cases: either $\mathcal{O}(\lfloor (\!| \, w \, |\!)_{\ell'} \rfloor_i, \Pi, \ell^\rightarrow, \rightarrow) \neq \circ$ or $\mathcal{O}(\lfloor (\!| \, w \, |\!)_{\ell'} \rfloor_i, \Pi, \ell^\rightarrow, \rightarrow) = \circ$. If $\mathcal{O}(\lfloor (\!| \, w \, |\!)_{\ell'} \rfloor_i, \Pi, \ell^\rightarrow, \rightarrow) \neq \circ$, then the proof is straightforward (since the $\mathcal{O}$ function is invoked on same term w). If $\mathcal{O}(\lfloor (\!| \, w \, |\!)_{\ell'} \rfloor_i, \Pi, \ell^\rightarrow, \rightarrow) = \circ$, we have to show that

$$\mathcal{O}(\lfloor w \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor w \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$$

By the typing rule CTX, we know $\Pi; \Gamma; pc \vdash w : \tau$. If $\lfloor (\!| \, w \, |\!)_{\ell'} \rfloor_1 \neq \lfloor (\!| \, w \, |\!)_{\ell'} \rfloor_2$, then w contains a bracket subexpression. That is either $w = (w_1\; \mathsf{where}\; v_1 \mid w_2\; \mathsf{where}\; v_2)$ or $w = (w_1 \mid w_2)\; \mathsf{where}\; v$. Note that v cannot be bracketed as per B-ASSUME. Consider the former case where $w = (w_1\; \mathsf{where}\; v_1 \mid w_2\; \mathsf{where}\; v_2)$. From BRACKET-VALUES

$$\Pi \vdash H^\rightarrow \sqsubseteq \tau^\rightarrow \tag{129}$$

$$\Pi; \Gamma; pc \vdash w_i\; \mathsf{where}\; v_i : \tau \tag{130}$$

$$\Pi; \Gamma; pc \vdash v_i : \langle r \geqslant t \rangle \tag{131}$$

From (129), we have that τ is protected. Invoking Lemma 31 (erasure on protected expressions) on w, we thus have $\mathcal{O}(\lfloor w \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor w \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$.

In the latter case, we have that $w = (w_1 \mid w_2)\; \mathsf{where}\; v$. From the well-typedness of WHERE, we have

$$\Pi, v; \Gamma; pc \vdash (w_1 \mid w_2) : \tau \tag{132}$$

From BRACKET-VALUES

$$\Pi, v \vdash H^\rightarrow \sqsubseteq \tau^\rightarrow \tag{133}$$

$$\Pi, v; \Gamma; pc \vdash w_i : \tau \tag{134}$$

From the soundness of the bracketed semantics (Lemma 2), we have that $(\!| \, w \, |\!)_{\ell'} \longrightarrow w$ implies $\lfloor (\!| \, w \, |\!)_{\ell'} \rfloor_i \longrightarrow \lfloor w \rfloor_i$ such that $\lfloor w \rfloor_i = w_i\; \mathsf{where}\; v$. Invoking delegation compartmentalization (Lemma 9) on the step $\lfloor (\!| \, w \, |\!)_{\ell'} \rfloor_i \longrightarrow w_i\; \mathsf{where}\; v$, we have two cases:

Subcase $\Pi \Vdash pc \geqslant \nabla(t)$: Invoking delegation invariance (Lemma 8), we have that $\Pi \nVdash \ell^\rightarrow \geqslant H^\rightarrow$ and $\Pi \nVdash pc \geqslant \nabla(\ell^\rightarrow - H^\rightarrow)$ implies the following. [23]

$$\Pi, v \nVdash \ell^\rightarrow \geqslant H^\rightarrow$$

---

[23]Contrapositive of Lemma 8.

Invoking Lemma 31 (Correctness of $\mathcal{O}$) on (134) and the above judgment, we have that $\mathcal{O}(w_1, \{\Pi, v\}, \ell^\rightarrow, \rightarrow) = \mathcal{O}(w_2, \{\Pi, v\}, \ell^\rightarrow, \rightarrow)$. Invoking Lemma 35 that preserves $\mathcal{O}$ under a restricted Π, we have the required $\mathcal{O}(w_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(w_2, \Pi, \ell^\rightarrow, \rightarrow)$. Hence proved.

Subcase $\Pi \vdash H^\rightarrow \sqsubseteq \tau^\rightarrow$**:** We have that $\Pi \vdash H^\rightarrow \sqsubseteq \tau^\rightarrow$. Using $\Pi \vdash H^\rightarrow \sqsubseteq \tau^\rightarrow$ and $\Pi \not\vdash H^\rightarrow \sqsubseteq \ell^\rightarrow$, invoke Lemma 31 (correctness of $\mathcal{O}$) on (134) (with $\Pi' = \Pi, v$), yields the required $\mathcal{O}(w_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(w_2, \Pi, \ell^\rightarrow, \rightarrow)$. Hence proved.

Case ASSUME: Given assume $\langle p \geqslant q \rangle$ in $e \longrightarrow e$ where $\langle p \geqslant q \rangle$. Also, given that $\mathcal{O}(\lfloor \text{assume } \langle p \geqslant q \rangle \text{ in } e \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor \text{assume } \langle p \geqslant q \rangle \text{ in } e \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$. We have to prove that

$$\mathcal{O}(\lfloor e \text{ where } \langle p \geqslant q \rangle \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor e \text{ where } \langle p \geqslant q \rangle \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$$

From the given conditions and definition of $\mathcal{O}$, we already have $\mathcal{O}(\lfloor e \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor e \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$. Hence $\mathcal{O}(\lfloor e \text{ where } \langle p \geqslant q \rangle \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor e \text{ where } \langle p \geqslant q \rangle \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$.

Case W-APP: Given $(w \text{ where } v)e \longrightarrow w\,e$ where v. Also given that $\mathcal{O}(\lfloor (w \text{ where } v)e \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor (w \text{ where } v)e \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$. We have to prove that

$$\mathcal{O}(\lfloor w\,e \text{ where } v \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor w\,e \text{ where } v \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$$

Note that by virtue of B-ASSUME, v cannot be a bracket value. Thus $\lfloor v \rfloor_1 = \lfloor v \rfloor_2$. From the given conditions, we have

$$\mathcal{O}(\lfloor w \text{ where } v \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor w \text{ where } v \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow) \tag{135}$$

$$\mathcal{O}(\lfloor e \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor e \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow) \tag{136}$$

Expanding the $\mathcal{O}$ function in (135), we have

$$\mathcal{O}(\lfloor w \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) \text{ where } \lfloor v \rfloor_1 = \mathcal{O}(\lfloor w \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow) \text{ where } \lfloor v \rfloor_2 \tag{137}$$

Combining (136) and (137) we have

$$\mathcal{O}(\lfloor w\,e \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) \text{ where } \lfloor v \rfloor_1 = \mathcal{O}(\lfloor w\,e \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow) \text{ where } \lfloor v \rfloor_2$$

From the $\mathcal{O}$ function this is equal to

$$\mathcal{O}(\lfloor w\,e \text{ where } v \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor w\,e \text{ where } v \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow)$$

Hence proved.

Case W-TAPP: Similar to W-APP case.

Case W-UNPAIR: Similar to W-APP case.

Case W-CASE: Similar to W-APP case.

Case W-BINDM: Similar to W-APP case.

Case W-ASSUME: Similar to W-APP case.

Case B-STEP: Straightforward application of induction hypothesis.

Case B-*: Observe that, with the exception of B-STEP, all B-* rules step from $e \longrightarrow^* e'$ such that $\lfloor e \rfloor_i = \lfloor e' \rfloor_i$. Therefore the lemma trivially holds.

E-Unpair: Trivial.

E-Case: Trivial.

E-UnitM: Trivial.

E-Eval: Given the following:

$$E[e] \longrightarrow E[e'] \tag{138}$$

$$\Pi; \Gamma; pc \vdash E[e] : \tau \tag{139}$$

$$\mathcal{O}(\lfloor E[e] \rfloor_1, \Pi, \ell^\rightarrow, \rightarrow) = \mathcal{O}(\lfloor E[e] \rfloor_2, \Pi, \ell^\rightarrow, \rightarrow) \tag{140}$$

We need to show

$$\mathcal{O}(\lfloor E[e'] \rfloor_1, \Pi, \ell^{\rightarrow}, \rightarrow) = \mathcal{O}(\lfloor E[e'] \rfloor_2, \Pi, \ell^{\rightarrow}, \rightarrow) \tag{141}$$

Applying Lemma 29 to (140), we have

$$\mathcal{O}(\lfloor e \rfloor_1, \Pi, \ell^{\rightarrow}, \pi) = \mathcal{O}(\lfloor e \rfloor_2, \Pi, \ell^{\rightarrow}, \pi) \tag{142}$$

Since we have (140), we can apply induction hypothesis to the premise of E-EVAL yielding

$$\mathcal{O}(\lfloor e' \rfloor_1, \Pi, \ell^{\rightarrow}, \pi) = \mathcal{O}(\lfloor e' \rfloor_2, \Pi, \ell^{\rightarrow}, \pi) \tag{143}$$

Applying Lemma (29) to (143), we thus have (141)

$\square$

B   Proofs for Robust Declassification (Lemma 3)

Lemma 36 (Substitution preserves $\mathcal{O}$). *Let* $\Pi; \Gamma, x : \tau', \Gamma'; pc \vdash e : \tau$ *and* $\Pi; \Gamma; pc \vdash v_i : \tau'$ *for* $i \in \{1, 2\}$. *If* $e[x \mapsto v_1] \approx_{p\pi}^{\Pi} e[x \mapsto v_2]$ *then either* e *does not have free occurrence of* x *or* $v_1 \approx_{p\pi}^{\Pi} v_2$.

Proof. Induction on the structure of e. $\quad\square$

Lemma 37 (Voice of Integrity Principal). $\Pi \Vdash \nabla(\ell^{\leftarrow}) \equiv \ell^{\leftarrow}$

Proof. Proven in Coq [5]. $\quad\square$

Lemma 38 (Duality of Voice and View). *For all* ℓ, $\Delta(\nabla(\ell^{\rightarrow})) \equiv \ell^{\rightarrow}$.

Proof. We have the following:

$$\nabla(\ell^{\rightarrow}) = \ell^{\leftarrow} \tag{144}$$
$$\Delta(\nabla(\ell^{\rightarrow})) = \Delta(\ell^{\leftarrow}) = \ell^{\rightarrow} \tag{145}$$

From (145), we thus have $\Delta(\nabla(\ell^{\rightarrow})) \equiv \ell^{\rightarrow}$. $\quad\square$

Theorem 3 (Robust declassification). *Suppose* $\Pi; \Gamma, x : \tau', \Gamma'; pc \vdash e[\vec{\bullet}^{\tau^*}] : \tau$. *For* Π_H *and* H *such that*

1. $\Pi_H \vdash H^{\rightarrow} \sqsubseteq \tau'$

2. $\Pi_H \nvdash H^{\rightarrow} \sqsubseteq \Delta(H^{\leftarrow})$

3. $\Pi_H \nvdash H^{\leftarrow} \geqslant \nabla(H^{\rightarrow})$,

Then for all Π_H-*fair attacks* $\vec{a_1}$ *and* $\vec{a_2}$ *such that* $\Pi; \Gamma, x : \tau', \Gamma'; pc \vdash e[\vec{a_i}] : \tau$ *and* $\Pi; \Gamma; pc \vdash v_i : \tau'$, *if* $e[a_j][x \mapsto v_i] \longrightarrow e'_{ij}$ *for* $i, j \in \{1, 2\}$, *then for the traces* $t_{ij} = (e[a_j][x \mapsto v_i]) \cdot e'_{ij}$), *we have*

$$t_{11} \approx_{\Delta(H^{\leftarrow})}^{\Pi} t_{21} \iff t_{12} \approx_{\Delta(H^{\leftarrow})}^{\Pi} t_{22}$$

Proof. By induction on the evaluation of e. A note on the notation: whenever the context is clear, we elide the type annotation on the hole.

Case E-APP* Since $\vec{a_1}$ and $\vec{a_2}$ are Π_H-fair attacks, we have that the attacks are well-typed. That is, for each $a_{rj} \in \vec{a_j}$,

$$\Pi_H; \Gamma_{rj}; pc \vdash a_{rj} : \tau_{rj} \text{ for } j \in \{1, 2\} \tag{146}$$

Without loss of generality, assume that $e[\vec{a}_j] = (e'[a_{kj}]e''[a_{lj}])$. Given $e[\vec{a}_j][x \mapsto v_i] \longrightarrow e'_{ij}$. That is, $(e'[a_{kj}]e''[a_{lj}])[x \mapsto v_i] \longrightarrow e'_{ij}$. Inverting the last condition, we have the following.

$$\begin{array}{lll} e'[\vec{a}_{kj}] & \triangleq (\lambda(y{:}\tau'')[pc'].\, e_{\lambda j}) & \text{for some } y, \tau'', pc' \text{ and } e_{\lambda j} \text{ corresponding to attack } j \in \{1, 2\} \\ e''[\vec{a}_{lj}] & \triangleq v_{aj} & \text{for some value } v_{aj} \text{ corresponding to attack } j \in \{1, 2\} \end{array} \tag{147}$$

At a high-level, (147) says that the attack is either an argument to the function or the function itself or both. We are also given that the terms are well-typed. From T-APP , we have

$$\Pi; \Gamma, x{:}\tau', \Gamma'; pc \vdash e'[\vec{a}_{k1}] \, e''[\vec{a}_{l1}] : \tau \tag{148}$$

$$\Pi; \Gamma, x{:}\tau', \Gamma'; pc \vdash e'[\vec{a}_{k2}] \, e''[\vec{a}_{l2}] : \tau \tag{149}$$

Since we are given that $\Pi; \Gamma, x{:}\tau', \Gamma'; pc \vdash e[\vec{\bullet}^{\vec{\tau}*}] : \tau$, we have

$$\Pi; \Gamma, x{:}\tau', \Gamma'; pc \vdash e'[\vec{\bullet}] : \tau \tag{150}$$

$$\Pi; \Gamma, x{:}\tau', \Gamma'; pc \vdash e''[\vec{\bullet}] : \tau'' \tag{151}$$

Note that we have elided the type annotations on holes to enhance readability. Substituting holes in (150) and (151) with $\vec{a}_1$, we have

$$\Pi; \Gamma, x{:}\tau', \Gamma'; pc \vdash (\lambda(y{:}\tau'')[pc'].\, e_{\lambda 1}) \, v_{a1} : \tau \tag{152}$$

$$\Pi; \Gamma, x{:}\tau', \Gamma'; pc \vdash (\lambda(y{:}\tau'')[pc'].\, e_{\lambda 2}) \, v_{a2} : \tau \tag{153}$$

From E-APP*, we have the following step.

$$\big((\lambda(y{:}\tau'')[pc'].\, e_{\lambda j}) \, v_{aj}\big)[x \mapsto v_i] \longrightarrow (\!| \, e_{\lambda j}[y \mapsto v_{aj}] \, |\!)_{pc'}[x \mapsto v_i] \tag{154}$$

Using the above relations, we are now ready to prove the `if` direction. Assume that the left hand holds.

$$e[\vec{a}_1][x \mapsto v_1] \cdot e'_{11} \approx^{\Pi}_{\Delta(H^{\leftarrow})} e[\vec{a}_1][x \mapsto v_2] \cdot e'_{21}$$

Substituting e'_{11} and e'_{21} with the result of step in (154) we have,

$$\big((\lambda(y{:}\tau'')[pc'].\, e_{\lambda 1}) \, v_{a1}\big)[x \mapsto v_1] \cdot (\!| \, e_{\lambda 1}[y \mapsto v_{a1}] \, |\!)_{pc'}[x \mapsto v_1]$$
$$\approx^{\Pi}_{\Delta(H^{\leftarrow})}$$
$$\big((\lambda(y{:}\tau'')[pc'].\, e_{\lambda 1}) \, v_{a1}\big)[x \mapsto v_2] \cdot (\!| \, e_{\lambda 1}[y \mapsto v_{a1}] \, |\!)_{pc'}[x \mapsto v_2] \tag{155}$$

Unfolding the definition of trace equivalence, we have

$$\big((\lambda(y{:}\tau'')[pc'].\, e_{\lambda 1}) \, v_{a1}\big)[x \mapsto v_1] \approx^{\Pi}_{\Delta(H^{\leftarrow})} \big((\lambda(y{:}\tau'')[pc'].\, e_{\lambda 1}) \, v_{a1}\big)[x \mapsto v_2] \tag{156}$$

$$(\!| \, e_{\lambda 1}[y \mapsto v_{a1}] \, |\!)_{pc'}[x \mapsto v_1] \approx^{\Pi}_{\Delta(H^{\leftarrow})} (\!| \, e_{\lambda 1}[y \mapsto v_{a1}] \, |\!)_{pc'}[x \mapsto v_2] \tag{157}$$

Without loss of generality, let $\vec{a}_2 \triangleq \vec{a}_{k2} \cdot \vec{a}_{l2}$. Following (147), we have the following when the application is filled with second attack.

$$e'[\vec{a}_{k2}] \triangleq \lambda(y{:}\tau'')[pc'].\, e_{\lambda 2} \tag{158}$$

$$e''[\vec{a}_{l2}] \triangleq v_{a2} \tag{159}$$

We have to prove the following:

$$\big((\lambda(y{:}\tau'')[pc'].\, e_{\lambda 2}) \, v_{a2}\big)[x \mapsto v_1] \approx^{\Pi}_{\Delta(H^{\leftarrow})} \big((\lambda(y{:}\tau'')[pc'].\, e_{\lambda 2}) \, v_{a2}\big)[x \mapsto v_2] \tag{160}$$

From Proposition 4, we have that $\vec{a}_{k1}$ and $\vec{a}_{l1}$ do not have any free occurrences of the variable x. Thus x only occurs in $e'[\bullet] \, e''[\bullet]$. Let x be some arbitrary occurrence in $e'[\vec{a}_{k1}] \, e''[\vec{a}_{l1}]$ such that $\Pi; \Gamma', x{:}\tau', \Gamma''; pc \vdash e'[\vec{a}_{k1}] \, e''[\vec{a}_{l1}] : \tau$. Since x is substituted with v_1 and v_2 and substitution preserves observation equivalence (Lemma 36), from (156) we have

$$\mathcal{O}(v_1, \Pi, \Delta(H^{\leftarrow}), \pi) = \mathcal{O}(v_2, \Pi, \Delta(H^{\leftarrow}), \pi) \tag{161}$$

The crucial insight here is that though x may occur in a sub term that is well-typed under different delegation context Π', however, the $\mathcal{O}$ function still uses the original Π under which the program is well-typed. Hence (161) uses the original delegation context Π, rather than Π'.

Again, from Proposition 4, we have that $\vec{a}_2$ cannot contain x. That is, $\vec{a}_{k2}$ and $\vec{a}_{l2}$ do not have free occurrences of x. Thus x only occurs in $e'[\bullet] \, e''[\bullet]$. Similar to the argument under $\vec{a}_1$, let x be some arbitrary occurrence in $e'[\vec{a}_{k2}] \, e''[\vec{a}_{l2}]$ such that $\Pi; \Gamma'_2, x : \tau', \Gamma''_2; pc'' \vdash x : \tau'$. Again, x is substituted with v_1 and v_2, to prove (164), it suffices to prove

$$\mathcal{O}(v_1, \Pi, \Delta(H^{\leftarrow}), \pi) = \mathcal{O}(v_2, \Pi, \Delta(H^{\leftarrow}), \pi) \tag{162}$$

This follows trivially from (161). We still have to prove that the step preserves erasure of terms:

$$(e_{\lambda 2}[y \mapsto v_{a2}])_{pc'}[x \mapsto v_1] \approx^{\Pi}_{\Delta(H^{\leftarrow})} (e_{\lambda 2}[y \mapsto v_{a2}])_{pc'}[x \mapsto v_2]$$

That is,

$$\mathcal{O}((e_{\lambda 2}[y \mapsto v_{a2}])_{pc'}[x \mapsto v_1], \Pi, \Delta(H^{\leftarrow}), \rightarrow) = \mathcal{O}((e_{\lambda 2}[y \mapsto v_{a2}])_{pc'}[x \mapsto v_2], \Pi, \Delta(H^{\leftarrow}), \rightarrow) \tag{163}$$

Applying type preservation (Lemma 4) to (152) and (153), we have that

$$\Pi; \Gamma, x : \tau', \Gamma'; pc \vdash (e_{\lambda 1}[y \mapsto v_{a1}])_{pc'} : \tau \tag{164}$$

$$\Pi; \Gamma, x : \tau', \Gamma'; pc \vdash (e_{\lambda 2}[y \mapsto v_{a2}])_{pc'} : \tau \tag{165}$$

From variable substitution under a context (Lemma 24), (164) and (165) lead to

$$\Pi; \Gamma, \Gamma'; pc \vdash (e_{\lambda 2}[y \mapsto v_{a2}][x \mapsto v_1])_{pc'} : \tau \tag{166}$$

$$\Pi; \Gamma, \Gamma'; pc \vdash (e_{\lambda 2}[y \mapsto v_{a2}][x \mapsto v_2])_{pc'} : \tau \tag{167}$$

It suffices to prove the following

$$\mathcal{O}(v_1, \Pi, \Delta(H^{\leftarrow}), \pi) = \mathcal{O}(v_2, \Pi, \Delta(H^{\leftarrow}), \pi) \tag{168}$$

This is already proven in (161). Hence proved.

The other if direction follows using analogous argument.

Case E-TApp*: Similar to E-App*.

Case O-Ctx: Given $(w[\vec{a}_1])_{\ell}[x \mapsto v_1] \longrightarrow w_1[\vec{a}_1]$ and $(w[\vec{a}_1])_{\ell}[x \mapsto v_2] \longrightarrow w_2[\vec{a}_1]$. We have to prove that

$$(w[\vec{a}_1])_{\ell}[x \mapsto v_1] \cdot w_1[\vec{a}_1] \approx^{\Pi}_{\Delta(H^{\leftarrow})} (w[\vec{a}_1])_{\ell}[x \mapsto v_2] \cdot w_2[\vec{a}_1]$$

$$\Longleftrightarrow$$

$$(w[\vec{a}_2])_{\ell}[x \mapsto v_1] \cdot w_1[\vec{a}_2] \approx^{\Pi}_{\Delta(H^{\leftarrow})} (w[\vec{a}_2])_{\ell}[x \mapsto v_2] \cdot w_2[\vec{a}_2]$$

We will prove one direction of the implication. The other direction is analogous. We are given that $\Pi; \Gamma, x : \tau', \Gamma'; pc \vdash e[\bullet^{\vec{\tau}*}] : \tau$, we

$$\Pi; \Gamma, x : \tau', \Gamma'; pc \vdash (w[\vec{\bullet}])_{\ell} : \tau \tag{169}$$

Substituting the hole in (169) with $\vec{a}_1$, we have

$$\Pi; \Gamma, x : \tau', \Gamma'; pc \vdash (w[\vec{a}_1])_{\ell} : \tau \tag{170}$$

Assume

$$(w[\vec{a}_1])_{\ell}[x \mapsto v_1] \cdot w_1[\vec{a}_1] \approx^{\Pi}_{\Delta(H^{\leftarrow})} (w[\vec{a}_1])_{\ell}[x \mapsto v_2] \cdot w_2[\vec{a}_1]$$

From the erasure definition, this implies

$$(w[\vec{a}_1])_{\ell}[x \mapsto v_1] \approx^{\Pi}_{\Delta(H^{\leftarrow})} (w[\vec{a}_1])_{\ell}[x \mapsto v_2] \tag{171}$$

$$w_1[\vec{a}_1] \approx^{\Pi}_{\Delta(H^{\leftarrow})} w_2[\vec{a}_1] \tag{172}$$

From Proposition 4, we have that $\vec{a}_2$ cannot contain x. Thus x only occurs in $(w[\vec{\bullet}])_{\ell}$. Let x be some arbitrary occurrence in $(w[\vec{\bullet}])_{\ell}$ such that $\Pi; \Gamma', x : \tau', \Gamma''; pc' \vdash (w[\vec{a}_2])_{\ell} : \tau'$. Since x is substituted with v_1 and v_2 and substitution preserves equivalence (Lemma 36), from (171) we have

$$\mathcal{O}(v_1, \Pi, \Delta(H^{\leftarrow}), \pi) = \mathcal{O}(v_2, \Pi, \Delta(H^{\leftarrow}), \pi) \tag{173}$$

Again, from Proposition 4, we have that $\vec{a}_2$ cannot contain x. Thus x only occurs in $(w[\vec{\bullet}])_{\ell}$.

We need to prove,

$$(w[\vec{a}_2])_{\ell}[x \mapsto v_1] \cdot w_1[\vec{a}_2] \approx^{\Pi}_{\Delta(H^{\leftarrow})} (w[\vec{a}_2])_{\ell}[x \mapsto v_2] \cdot w_2[\vec{a}_2]$$

That is, we need to prove the following.

$$(\! | \, w[\vec{a}_2] \, | \!)_\ell [x \mapsto v_1] \approx^\Pi_{\Delta(H^\leftarrow)} (\! | \, w[\vec{a}_2] \, | \!)_\ell [x \mapsto v_2] \tag{174}$$

$$w_1[\vec{a}_2] \approx^\Pi_{\Delta(H^\leftarrow)} w_2[\vec{a}_2] \tag{175}$$

Similar to the argument under $\vec{a}_1$, let x be some arbitrary occurrence in $w[\vec{a}_2]$. Again, x is substituted with v_1 and v_2, to prove (174), it suffices to prove

$$\mathcal{O}(v_1, \Pi, \Delta(H^\leftarrow), \pi) = \mathcal{O}(v_2, \Pi, \Delta(H^\leftarrow), \pi) \tag{176}$$

And, judgment (173) already gives this.

We will now focus on proving (175). Note that $w_1[\bullet] = w[\bullet][x \mapsto v_1]$ and $w_2[\bullet] = w[\bullet][x \mapsto v_2]$. From (174) and the definition of erasure function for contexts, we have that

$$w[\vec{a}_2][x \mapsto v_1] \approx^\Pi_{\Delta(H^\leftarrow)} w[\vec{a}_2][x \mapsto v_2]$$

Thus (175) follows trivially from (174). Hence proved.

Case E-CASE: Given $(\text{case } \text{inj}_k \, w \text{ of } \text{inj}_1(y). \, e_1 \mid \text{inj}_2(y). \, e_2)[\vec{a}_j][x \mapsto v_i] \longrightarrow e_k[\vec{a}_j][y \mapsto w]$ for $i, j, k \in \{1, 2\}$. We have to prove that

$$(\text{case } \text{inj}_k \, w \text{ of } \text{inj}_1(y). \, e_1 \mid \text{inj}_2(y). \, e_2)[\vec{a}_1][x \mapsto v_1] \cdot e_k[\vec{a}_j][x \mapsto v_1][y \mapsto w]$$
$$\approx^\Pi_{\Delta(H^\leftarrow)}$$
$$(\text{case } \text{inj}_k \, w \text{ of } \text{inj}_1(y). \, e_1 \mid \text{inj}_2(y). \, e_2)[\vec{a}_1][x \mapsto v_2] \cdot e_k[\vec{a}_j][x \mapsto v_2][y \mapsto w]$$
$$\Longleftrightarrow$$
$$(\text{case } \text{inj}_k \, w \text{ of } \text{inj}_1(y). \, e_1 \mid \text{inj}_2(y). \, e_2)[\vec{a}_2][x \mapsto v_1] \cdot e_k[\vec{a}_1][x \mapsto v_1][y \mapsto w]$$
$$\approx^\Pi_{\Delta(H^\leftarrow)}$$
$$(\text{case } \text{inj}_k \, w \text{ of } \text{inj}_1(y). \, e_1 \mid \text{inj}_2(y). \, e_2)[\vec{a}_2][x \mapsto v_2] \cdot e_k[\vec{a}_2][x \mapsto v_2][y \mapsto w]$$

We will prove only one direction. Assume

$$(\text{case } \text{inj}_k \, w \text{ of } \text{inj}_1(y). \, e_1 \mid \text{inj}_2(y). \, e_2)[\vec{a}_1][x \mapsto v_1] \cdot e_k[\vec{a}_1][x \mapsto v_1][y \mapsto w]$$
$$\approx^\Pi_{\Delta(H^\leftarrow)}$$
$$(\text{case } \text{inj}_k \, w \text{ of } \text{inj}_1(y). \, e_1 \mid \text{inj}_2(y). \, e_2)[\vec{a}_2][x \mapsto v_2] \cdot e_k[\vec{a}_2][x \mapsto v_2][y \mapsto w]$$

That implies

$$(\text{case } \text{inj}_k \, w \text{ of } \text{inj}_1(y). \, e_1 \mid \text{inj}_2(y). \, e_2)[\vec{a}_1][x \mapsto v_1]$$
$$\approx^\Pi_{\Delta(H^\leftarrow)}$$
$$(\text{case } \text{inj}_k \, w \text{ of } \text{inj}_1(y). \, e_1 \mid \text{inj}_2(y). \, e_2)[\vec{a}_1][x \mapsto v_2]$$

and

$$e_k[\vec{a}_2][x \mapsto v_1][y \mapsto w] \approx^\Pi_{\Delta(H^\leftarrow)} e_k[\vec{a}_2][x \mapsto v_2][y \mapsto w]$$

We need to prove

$$(\text{case } \text{inj}_k \, w \text{ of } \text{inj}_1(y). \, e_1 \mid \text{inj}_2(y). \, e_2)[\vec{a}_2][x \mapsto v_1]$$
$$\approx^\Pi_{\Delta(H^\leftarrow)}$$
$$(\text{case } \text{inj}_k \, w \text{ of } \text{inj}_1(y). \, e_1 \mid \text{inj}_2(y). \, e_2)[\vec{a}_2][x \mapsto v_2] \tag{177}$$

and

$$e_k[\vec{a}_2][x \mapsto v_1][y \mapsto w] \approx^\Pi_{\Delta(H^\leftarrow)} e_k[\vec{a}_2][x \mapsto v_2][y \mapsto w] \tag{178}$$

Let x be some arbitrary occurrence in $\text{case } \text{inj}_k \, w \text{ of } \text{inj}_1(y). \, e_1 \mid \text{inj}_2(y). \, e_2[\vec{a}_1]$ such that $\Pi; \Gamma', x : \tau', \Gamma''; pc' \vdash \text{case } \text{inj}_k \, w \text{ of } \text{inj}_1(y). \, e_1 \mid \text{inj}_2(y). \, e_2[\vec{a}_1] : \tau'$. Since x is substituted with v_1 and v_2 and substitution preserves observational equivalence (Lemma 36), we have

$$\mathcal{O}(v_1, \Pi, \Delta(H^\leftarrow), \pi) = \mathcal{O}(v_2, \Pi, \Delta(H^\leftarrow), \pi) \tag{179}$$

Again, from Proposition 4, we have that $\vec{a}_2$ cannot contain x. Thus x only occurs in

case $\mathrm{inj}_k\ w$ of $\mathrm{inj}_1(y).\ e_1\ |\ \mathrm{inj}_2(y).\ e_2)[\bullet]$. Similar to the argument under $\vec{a}_1$, let x be some arbitrary occurrence in case $\mathrm{inj}_k\ w$ of $\mathrm{inj}_1(y).\ e_1\ |\ \mathrm{inj}_2(y).\ e_2[\vec{a}_2]$ such that $\Pi; \Gamma_2', x : \tau', \Gamma_2''; pc \vdash$ case $\mathrm{inj}_k\ w$ of $\mathrm{inj}_1(y).\ e_1\ |\ \mathrm{inj}_2(y).\ e_2[\vec{a}_2] : \tau'$. Again, x is substituted with v_1 and v_2. Following an argument similar to previous cases, it suffice to prove

$$\mathcal{O}(v_1, \Pi, \Delta(H^\leftarrow), \pi) = \mathcal{O}(v_2, \Pi, \Delta(H^\leftarrow), \pi) \tag{180}$$

And, judgment (179) already gives us the required proof.

We now focus on proving (178). From (177), we have

$$\mathrm{inj}_k\ w[\vec{a}_2][x \mapsto v_1] \approx^{\Pi}_{\Delta(H^\leftarrow)} \mathrm{inj}_k\ w[\vec{a}_2][x \mapsto v_2] \tag{181}$$

$$e_1[\vec{a}_2][x \mapsto v_1] \approx^{\Pi}_{\Delta(H^\leftarrow)} e_1[\vec{a}_2][x \mapsto v_2] \tag{182}$$

$$e_2[\vec{a}_2][x \mapsto v_1] \approx^{\Pi}_{\Delta(H^\leftarrow)} e_2[\vec{a}_2][x \mapsto v_2] \tag{183}$$

We are given that

$$\Pi; \Gamma, x : \tau', \Gamma'; pc \vdash (\text{case } \mathrm{inj}_k\ w \text{ of } \mathrm{inj}_1(y).\ e_1\ |\ \mathrm{inj}_2(y).\ e_2)[\vec{a}_1] : \tau$$

Inverting the rule CASE, we have

$$\Pi; \Gamma, x : \tau', \Gamma', y : \tau_1 + \tau_2; pc \vdash (e_k)[\vec{a}_1] : \tau$$

Since variable substitution is preserved under contexts (Lemma 24), from (181), (182), we have (178). Hence proved.

Case E-BINDM*: Given $(\text{bind } y = \overline{\eta}_\ell\ w \text{ in } e_2)[\vec{a}_j][x \mapsto v_i] \longrightarrow e_2[\vec{a}_j][y \mapsto w]$ for $i, j \in \{1, 2\}$. We have to prove that

$$(\text{bind } y = \overline{\eta}_\ell\ w \text{ in } e_2)[\vec{a}_1][x \mapsto v_1] \cdot e_2[\vec{a}_j][x \mapsto v_1][y \mapsto w]$$
$$\approx^{\Pi}_{\Delta(H^\leftarrow)}$$
$$(\text{bind } y = \overline{\eta}_\ell\ w \text{ in } e_2)[\vec{a}_1][x \mapsto v_2] \cdot e_2[\vec{a}_j][x \mapsto v_2][y \mapsto w]$$
$$\Longleftrightarrow$$
$$(\text{bind } y = \overline{\eta}_\ell\ w \text{ in } e_2)[\vec{a}_2][x \mapsto v_1] \cdot e_2[\vec{a}_j][x \mapsto v_1][y \mapsto w]$$
$$\approx^{\Pi}_{\Delta(H^\leftarrow)}$$
$$(\text{bind } y = \overline{\eta}_\ell\ w \text{ in } e_2)[\vec{a}_2][x \mapsto v_2] \cdot e_2[\vec{a}_j][x \mapsto v_2][y \mapsto w]$$

The argument is similar to the above case.

Case E-ASSUME: Given $(\text{assume } \langle p \succcurlyeq q \rangle \text{ in } e_2)[\vec{a}_j][x \mapsto v_i] \longrightarrow (e_2 \text{ where } \langle p \succcurlyeq q \rangle)[\vec{a}_j][x \mapsto v_i]$ for $i, j \in \{1, 2\}$. We have to prove that

$$(\text{assume } \langle p \succcurlyeq q \rangle \text{ in } e_2)[\vec{a}_1][x \mapsto v_1] \cdot (e_2 \text{ where } \langle p \succcurlyeq q \rangle)[\vec{a}_j][x \mapsto v_1]$$
$$\approx^{\Pi}_{\Delta(H^\leftarrow)}$$
$$(\text{assume } \langle p \succcurlyeq q \rangle \text{ in } e_2)[\vec{a}_1][x \mapsto v_2] \cdot (e_2 \text{ where } \langle p \succcurlyeq q \rangle)[\vec{a}_j][x \mapsto v_2]$$
$$\Longleftrightarrow$$
$$(\text{assume } \langle p \succcurlyeq q \rangle \text{ in } e_2)[\vec{a}_2][x \mapsto v_1] \cdot (e_2 \text{ where } \langle p \succcurlyeq q \rangle)[\vec{a}_j][x \mapsto v_1]$$
$$\approx^{\Pi}_{\Delta(H^\leftarrow)}$$
$$(\text{assume } \langle p \succcurlyeq q \rangle \text{ in } e_2)[\vec{a}_2][x \mapsto v_2] \cdot (e_2 \text{ where } \langle p \succcurlyeq q \rangle)[\vec{a}_j][x \mapsto v_2]$$

The argument is similar to the previous cases. To prove in one direction, we assume the following.

$$(\text{assume } \langle p \succcurlyeq q \rangle \text{ in } e_2)[\vec{a}_1][x \mapsto v_1] \cdot (e_2 \text{ where } \langle p \succcurlyeq q \rangle)[\vec{a}_1][x \mapsto v_1]$$
$$\approx^{\Pi}_{\Delta(H^\leftarrow)}$$
$$(\text{assume } \langle p \succcurlyeq q \rangle \text{ in } e_2)[\vec{a}_1][x \mapsto v_2] \cdot (e_2 \text{ where } \langle p \succcurlyeq q \rangle)[\vec{a}_1][x \mapsto v_2]$$

This gives us

$$(\text{assume } \langle p \geqslant q \rangle \text{ in } e_2)[\vec{a}_1][x \mapsto v_1] \approx^{\Pi}_{\Delta(H^{\leftarrow})} (\text{assume } \langle p \geqslant q \rangle \text{ in } e_2)[\vec{a}_1][x \mapsto v_2] \qquad (184)$$

$$(e_2 \text{ where } \langle p \geqslant q \rangle)[\vec{a}_1][x \mapsto v_1] \approx^{\Pi}_{\Delta(H^{\leftarrow})} (e_2 \text{ where } \langle p \geqslant q \rangle)[\vec{a}_1][x \mapsto v_2] \qquad (185)$$

Consider (184). Let x be some arbitrary occurrence in $(\text{assume } \langle p \geqslant q \rangle \text{ in } e_2)[\vec{a}_1]$ such that $\Pi; \Gamma', x : \tau', \Gamma''; pc' \vdash (\text{assume } \langle p \geqslant q \rangle \text{ in } e_2)[\vec{a}_1] : \tau'$. Since x is substituted with v_1 and v_2 and substitution preserves observational equivalence (Lemma 36), we have

$$\mathcal{O}(v_1, \Pi, \Delta(H^{\leftarrow}), \pi) = \mathcal{O}(v_2, \Pi, \Delta(H^{\leftarrow}), \pi) \qquad (186)$$

Again, from Proposition 4, we have that $\vec{a}_2$ cannot contain x. Thus x only occurs in $(\text{assume } \langle p \geqslant q \rangle \text{ in } e_2)[\vec{\bullet}]$. Similar to the argument under $\vec{a}_1$, let x be some arbitrary occurrence in $(\text{assume } \langle p \geqslant q \rangle \text{ in } e_2)[\vec{a}_2]$ such that $\Pi; \Gamma'_2, x : \tau', \Gamma''_2; pc'' \vdash (\text{assume } \langle p \geqslant q \rangle \text{ in } e_2)[\vec{a}_2] : \tau'$. Again, x is substituted with v_1 and v_2. Following an argument similar to one in E-APP* case, it suffices prove the following

$$\mathcal{O}(v_1, \Pi, \Delta(H^{\leftarrow}), \pi) = \mathcal{O}(v_2, \Pi, \Delta(H^{\leftarrow}), \pi) \qquad (187)$$

Judgment (186) already gives us the above judgment. Hence

$$(\text{assume } \langle p \geqslant q \rangle \text{ in } e_2)[\vec{a}_2][x \mapsto v_1] \approx^{\Pi}_{\Delta(H^{\leftarrow})} (\text{assume } \langle p \geqslant q \rangle \text{ in } e_2)[\vec{a}_2][x \mapsto v_2]$$

A similar argument applied to (185) gives us

$$(e_2 \text{ where } \langle p \geqslant q \rangle)[\vec{a}_2][x \mapsto v_1] \approx^{\Pi}_{\Delta(H^{\leftarrow})} (e_2 \text{ where } \langle p \geqslant q \rangle)[\vec{a}_2][x \mapsto v_2]$$

Hence proved.

Case W-APP: Similar to the assume case.

Case W-TAPP: Similar to the assume case.

Case W-UNPAIR: Trivial.

Case W-CASE: Similar to the assume case.

Case W-BINDM: Similar to the assume case.

Case W-ASSUME: Similar to the assume case.

$\square$