A Calculus for Flow-Limited Authorization

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
E-mail: owen@soe.ucsc.edu

\(^{b}\) Harvard University, Cambridge, MA, USA
E-mail: agollamudi@g.harvard.edu, chong@seas.harvard.edu

\(^{c}\) Cornell University, Ithaca, NY, USA
E-mail: {ethan,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 a way to construct dynamic authorization mechanisms that do not violate confidentiality or integrity.

We introduce the Flow-Limited Authorization Calculus (FLAC), which is both a simple, expressive model for reasoning about dynamic authorization and also 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 \([27,2,38]\), often implemented with cryptographic mechanisms \([19,13]\); role-based access control (RBAC) \([20]\); and trust management \([28,39,9]\).

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 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., [27], [2], [28]) 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 use the administrator to carry out an attack on their behalf. Unfortunately, existing 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 [29], so we want to be able to verify their security. We argue that information flow control is a lightweight, useful tool for building secure authorization mechanisms. Using information flow control is attractive since it offers compositional, end-to-end security guarantees. However, applying information flow control to these 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 sometimes 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).\footnote{DCC was first presented in [4]. We use the abbreviation DCC to refer to the extension to polymorphic types in [2].} 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) [7], 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.

The types in a FLAC program can be considered propositions [42] in an authorization logic, and the programs can be considered proofs that the proposition holds. Well-typed FLAC programs are not only proofs of authorization, but also proofs of secure information flow, ensuring the confidentiality and integrity of authorization policies and of the data those policies depend upon.

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 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.

– 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 [7], and Section 4 defines the FLAC language and type system. FLAC implementations of the dynamic authorization examples are presented in Section 5, and their properties are examined. Section 6 explores aspects of FLAC’s proof theory, and Section 7 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 [5]. This article expands upon and strengthens the formal results, as well as corrects several technical errors in that work. Most of the semantic security proofs in Section 7 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 Ceccetti \textit{et al.} [15] and Gollamudi \textit{et al.} \cite{Gollamudi2020} 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 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 5 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 [35] 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. Suppose $p$ wants to commit to a value to principal $q$. First, $p$ applies commit to the value and provides the result to $q$ which does not reveal the value to $q$. When $p$ wishes to reveal the value, it supplies $q$ with a reveal operation for the commitment it sent previously. Then $q$ uses reveal to open the received value, revealing it to $q$.

A commitment scheme must have several properties in order to be secure. First, $q$ should not be able to open a value that hasn’t been committed to by $p$, 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 $q$.

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. Implementing 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 models of the security properties achieved 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 [19], as well as in more recent ones like Macaroons [13].

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. [13], 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., [13,24,11]) 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 [7].

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

Unlike principals in other models, FLAM principals also represent information flow policies. The confidentiality of principal $p$ is represented by the principal $p^\land$, called $p$’s confidentiality projection. It denotes the authority necessary to learn anything $p$ can learn. The integrity of principal $p$ is represented by $p^\lor$, called $p$’s integrity projection. It denotes the authority to influence anything $p$ can influence. All authority may be represented as some combination of confidentiality and integrity. For instance, principal $p$ is equivalent to the conjunction $p^\land \land p^\lor$, and in fact any FLAM principal can be written $p^\land \land q^\land$ for some $p$ and $q$. The closure of the set of names $\mathcal{N}$ plus $\top$ and $\bot$ under the operators $\land$, $\lor$, $\leftarrow$, $\rightarrow$ forms a lattice $\mathcal{L}$ ordered by an acts-for relation $\gg$, defined by the inference rules in Figure 1. We write operators $\leftarrow$, $\rightarrow$ with higher precedence than $\land$, $\lor$; for instance, $p \land q^\rightarrow$ is equal to $p^\rightarrow \land (p \land q)^\rightarrow$. Projections distribute over $\land$ and $\lor$, so for example, $(p \land q)^\rightarrow = (p^\rightarrow \land q^\rightarrow)$. The confidentiality and integrity authority of principals are disjoint, so the confidentiality projection of an integrity projection is $\bot$ and vice-versa: $(p^\rightarrow)^\land = \bot = (p^\land)^\rightarrow$.

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 au-

---

2 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.

3 FLAM defines an additional set of operators called ownership projections, which we omit here to simplify our presentation.
\[
\mathcal{L} \models p \geq q
\]

\[
\begin{align*}
\text{[BOT]} & \quad \mathcal{L} \models p \geq \perp \\
\text{[TOP]} & \quad \mathcal{L} \models \top \geq p \\
\text{[REFL]} & \quad \mathcal{L} \models p \geq p \\
\text{[PROJ]} & \quad \mathcal{L} \models p \geq q \quad \mathcal{L} \models p^\pi \geq q^\pi \\
\text{[PROIR]} & \quad \mathcal{L} \models p \geq p^\pi
\end{align*}
\]

\[
\begin{align*}
\text{[CONJ]} & \quad \mathcal{L} \models p_k \geq p \quad k \in \{1, 2\} \\
\text{[DISJ]} & \quad \mathcal{L} \models p \geq p_1 \wedge p_2 \geq p \\
\text{[CONJR]} & \quad \mathcal{L} \models p \geq p_1 \quad \mathcal{L} \models p \geq p_2 \\
\text{[DISJR]} & \quad \mathcal{L} \models p \geq p_1 \vee p_2 \geq p \\
\text{[TRANS]} & \quad \mathcal{L} \models p \geq q \quad \mathcal{L} \models q \geq r \\
\end{align*}
\]

Fig. 1. Static principal lattice rules, adapted from FLAM [7]. The projection \( \pi \) may be either confidentiality (\( \to \)) or integrity (\( \leftarrow \)).

authority \( p^- \) and \( p \) has at least the integrity authority \( q^- \). This relationship between the confidentiality and integrity of \( p \) and \( q \) reflects the usual duality seen in information flow control [12]. As in [7], we use the following shorthand for relating principals by policy restrictiveness:

\[
\begin{align*}
p \trianglelefteq q & \triangleq (p^- \wedge q^-) \geq (q^- \wedge p^-) \\
p \sqcup q & \triangleq (p \wedge q)^- \wedge (p \vee q)^- \\
p \sqcap q & \triangleq (p \vee q)^- \wedge (p \wedge q)^-
\end{align*}
\]

Thus, \( p \trianglelefteq q \) 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 \( q \).

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\(^4\) as \( q^- \wedge r^- \), the voice of that principal is

\[
\nabla(q^- \wedge r^-) \triangleq q^- \wedge r^-
\]

For example, the voice of Alice, \( \nabla(Alice) \), is \( Alice^- \). The voice of Alice’s confidentiality \( \nabla(Alice^-) \) is also \( Alice^- \).

\[4\text{In normal form, a principal is the conjunction of a confidentiality principal and an integrity principal. See [7] for details.}\]


FLAC uses information flow to reason about the security implications of dynamically computed authority. Like previous information-flow type systems [37], FLAC incorporates types for reasoning about
Fig. 2. FLAC syntax. Terms using where are syntactically prohibited in the source language and are produced only during evaluation.

\[
\begin{align*}
\tau ::= (p \triangleright p) \mid \text{unit} \mid \tau + \tau \mid \tau \times \tau \\
\ell \text{ says } X \mid \forall X[pc]. \tau \\
v ::= () \mid \langle w, w \rangle \mid \langle p \triangleright p \rangle \mid \eta_\ell w \\
inj_1 w \mid \lambda(x: \tau)[pc]. e \mid \Lambda X[pc]. e \\
w ::= v \mid where v \\
e ::= x \mid w \mid e \mid \langle e, e \rangle \mid \eta_\ell e \\
\text{ case } e \text{ of } \text{inj}_1(x). e \mid \text{inj}_2(x). e \\
bind x = e \in e \mid assume e \in e \\
e \mid where v
\end{align*}
\]

Fig. 3. FLAC operational semantics

\[
E ::= [] \mid E e \mid w E \mid E \tau \mid \langle E, e \rangle \mid \langle w, E \rangle \mid \text{proj}_1 E \mid \text{inj}_1 E \mid \eta_\ell E \\
\mid bind x = E \in e \mid assume E \in e \mid \text{case } E \text{ of } \text{inj}_1(x). e \mid \text{inj}_2(x). e \mid E \text{ where } v
\]

information flow, but FLAC’s type system goes further by using Flow-Limited Authorization [7] 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 deriving 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 [7].
\[
\begin{array}{ll}
\Pi; \Gamma; pc \vdash e : \tau \\
[\text{VAR}] \quad \Pi; \Gamma, x : \tau, \Pi'; pc \vdash x : \tau \\
\quad \Pi; \Gamma; pc \vdash () : \text{unit} \\
\quad \Pi; \Gamma; pc \vdash \langle p \geq q \rangle : (p \geq q)
\end{array}
\]

\[
\begin{array}{ll}
\Pi; \Gamma; pc \vdash \lambda (x : \tau_1)[pc'] . e : (\tau_1 \mapsto \tau_2) \\
\Pi; \Gamma; pc \vdash X : \tau \\
\Pi; \Gamma; pc \vdash \Lambda X [pc'] . e : \forall X [pc'] . \tau \\
\Pi; \Gamma; pc \vdash (e \tau') : \tau [X \mapsto \tau'] \\
\tau' \text{ well-formed in } \Gamma
\end{array}
\]

\[
\begin{array}{ll}
\Pi; \Gamma; pc \vdash e : \tau_1 \\
\Pi; \Gamma; pc \vdash e_1 : \tau_2 \\
\Pi; \Gamma; pc \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2
\end{array}
\]

\[
\begin{array}{ll}
\Pi; \Gamma; pc \vdash e : \tau \\
\Pi \vdash pc \subseteq \ell \\
\Pi; \Gamma; pc \vdash \eta_\ell e : \ell \text{ says } \tau
\end{array}
\]

\[
\begin{array}{ll}
\Pi; \Gamma; pc \vdash e : \ell \text{ says } \tau' \\
\Pi; \Gamma; x : \tau' ; pc \sqcup \ell \vdash e' : \tau
\end{array}
\]

\[
\begin{array}{ll}
\Pi; \Gamma; pc \vdash \text{bind } x = e \text{ in } e' : \tau
\end{array}
\]

\[
\begin{array}{ll}
\Pi; \Gamma; pc \vdash e : (p \geq q) \\
\Pi \vdash pc \geq \nabla (q) \\
\Pi \vdash \nabla (p^-) \geq \nabla (q^-)
\end{array}
\]

\[
\begin{array}{ll}
\Pi; \Gamma; pc \vdash \text{assume } e \text{ in } e' : \tau
\end{array}
\]

\[
\begin{array}{ll}
\Pi; \Gamma; pc \vdash v : (p \geq q) \\
\Pi \vdash pc \geq \nabla (q) \\
\Pi \vdash \nabla (p^-) \geq \nabla (q^-)
\end{array}
\]

\[
\begin{array}{ll}
\Pi; \Gamma; pc \vdash (e \text{ where } v) : \tau
\end{array}
\]

Fig. 4. FLAC type system.

Figure 2 defines the FLAC syntax. The core FLAC operational semantics and evaluation contexts [44] 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, \(\Pi\), contains a set of labeled dynamic trust relationships \(\langle p \geq q \rangle\) where \(p \geq q\) (read as “\(p\) acts for \(q\)”) is a delegation from \(q\) to \(p\). The typing context, \(\Gamma\), is a map from variables to types, and \(pc\) is the program counter label, a FLAC principal representing the confidentiality and integrity of control flow. The type system makes frequent use of judgments adapted from FLAM’s inference rules [7]. These rules, adapted to FLAC, are presented in Figure 5.\(^5\)

---

\(^5\)FLAM’s rules [7] 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. These labels are unnecessary in FLAC because we use FLAM
FLAC to support mutable references could control implicit flows similarly.

formalize the connection between FLAC and FLAM in Appendix A.

FLAC originally used a version of the DCC rule, but Cecchetti et al. [15] and Gollamudi et al. [27] added the pc restriction in support of a non-commutative says. See Section 6.1 for additional details.

judgments only in the type system—these “queries” only occur at compile time and do not create information flows. We formalize the connection between FLAC and FLAM in Appendix A.

6DFLATE [27], an extension of FLAC, uses the same pc label to control implicit flows due to communication. Extensions of FLAC to support mutable references could control implicit flows similarly.

7This premise simplifies our proofs, but does not appear to be strictly necessary; BINDM ensures the same property.

8Neither DCC [4] nor the original FLAC formalization [5] 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. [15] and Gollamudi et al. [27] added the pc restriction in support of a non-commutative says. See Section 6.1 for additional details.

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 [36,32]). The purpose of FLAC’s pc label is to control a different kind of side effect: changes to the delegation context, π. In order to control what information can influence whether a new trust relationship is added to the 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. For instance, if we have ϕ ↛ ψ and ϕ, then (via APP) we may derive ψ in a context with integrity p−, but not in contexts that don’t flow to p−. This feature offers needed control over how principals may apply existing facts to derive new facts.

Many FLAC terms are standard, such as pairs ⟨e1, e2⟩, projections proj e, variants inj e, polymorphic type abstraction, Λ X [pc]. e, and case expressions. Function abstraction, λ (x : τ) [pc]. e, includes 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 π ⊢ pc ⊆ 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 π ⊢ pc ⊆ τ ensures that this type protects the current pc label.

Like DCC, FLAC uses monadic operators to track dependencies. The monadic unit term ηℓ v (UNITM) says that a value v of type τ is protected at level ℓ. This protected value has the type ℓ says τ, meaning that it has the confidentiality and integrity of principal ℓ. Because v could implicitly reveal information about the dependencies of the computation that produced it, UNITM requires that π ⊢ pc ⊆ ℓ.

[Fig. 5. Inference rules for robust assumption, adapted from FLAM [7].]
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 `bind x = ηℓ v in ηℓ 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 ∥ ℓ to conservatively reflect the control-flow dependency.

Protection levels are defined by the set of inference rules in Figure 7, adapted from [41]. 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 ℓ' says τ protects information at ℓ (P-LBL).9

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 `assume x: p−⇒ q−` says τ; q− (i.e., Π = ∅, Γ = x: p− says τ, and pc = q−), the following expression is not well typed:

\[
\text{bind } x' = x \text{ in } ηq− x'
\]

since p− does not flow to q−, as required by the premise Π ⊢ ℓ ⊆ τ in rule `BINDM`. Specifically, we cannot derive Π ⊢ p− ⊆ q− says τ since P-LBL requires the FLAM judgment Π ⊢ p− ⊆ q− to hold.

However, the following expression is well typed:

\[
\text{assume } \langle p− ⇒ q− \rangle \text{ in bind } x' = x \text{ in } ηq− x'
\]

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 `⟨ p− ⇒ q− ⟩` adds a trust relationship that allows p− to flow to q−. This is secure since pc = q−, meaning that only principals with integrity q− have influenced the computation. With `⟨ p− ⇒ q− ⟩` 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 that enable 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 `⟨ p− ⇒ q− ⟩` in the example above.11 The second premise, Π ⊢ pc ≫ ∇q(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 ⊃ q, the evidence represented by the term e must have at least the integrity of the voice of q, written ∇q(q). Since the pc bounds the restrictiveness of the

---

9 DCC [4] and the original FLAC formalization included an additional protection rule that considered ℓ to be protected by ℓ' says τ if τ protects ℓ (even if ℓ' does not). This rule was removed by Ceccetti et al. and Gollamudi et al. [?] to make says non-commutative.

10 This correspondence with delegation values makes acts-for types a kind of singleton type [18].

11 These premises are related to the robust FLAM rule LIFT.
dependencies of ε, this ensures that only information with integrity ρ(q) or higher may influence the evaluation of ε. The third premise, \( \Pi \vdash \rho(p^-) \gg \rho(q^-) \), ensures that principal p has sufficient integrity to be trusted to enforce q’s confidentiality, q^-_. This premise means that q permits data to be relabeled from q^- to p^-_.

Assumption terms evaluate to where expressions (rule E-ASSUME). These expressions are a purely formal bookkeeping mechanism (i.e., they would be unnecessary in a FLAC-based implementation) for tracking implicit flows due to dynamic delegations. They 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 \( \langle p \gg q \rangle \) records that ε is evaluated in a context that includes the delegation \( \langle p \gg q \rangle \).

The rule WHERE gives a typing rule for where terms; though similar to ASSUME, it requires only that ρ(q) delegate to the distinguished label π. The use of π is purely technical: our proofs in Section 7 use π to help reason about what new flows may have created by assume terms. The only requirement is that π be as trusted as the pc label used to type-check the program (or programs) that generated the where term. Since the pc increases monotonically when typing subexpressions, the typical choice for π is for a source program e such that \( \Pi; \Gamma; pc \vdash e : \tau \) would be pc.

Figure 6 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.

\[ ε \rightarrow ε’ \]

[W-APP] \( (w \text{ where } v) \ e \rightarrow (w \ e) \text{ where } v \)

[W-UNPAIR] \( \text{proj}_w (w \text{ where } v) \rightarrow (\text{proj}_v w) \text{ where } v \)

[W-CASE] \( (\text{case } w \text{ where } v \rangle \text{ of inj}_1(x) \text{ | inj}_2(x) \text{ | } e_1 \text{ | inj}_2(x) \text{ | } e_2) \rightarrow (\text{case } w \text{ of } \text{inj}_1(x) \text{ | } e_1 \text{ | inj}_2(x) \text{ | } e_2) \text{ where } v \)

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

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

Fig. 6. Propagation of where terms

---

12 More precisely, it means that the voice of q's confidentiality, \( \rho(q^-) \), permits data to be relabeled from q^- to p^-_. Recall that \( \rho(Alice^-) \) is just Alice’s integrity projection: Alice^-_.

13 Choosing π to be \( T^- \) is always valid, but selecting a more restrictive label permits finer-grained reasoning about what downgrades may occur in ε.

14 The original FLAC formalization did not distinguish where values and values, and did not include the rules in Figure 6. Unfortunately, this resulted in stuck terms when where terms were not propagated appropriately.
Fig. 7. Type protection levels

\[
\begin{align*}
\Pi \vdash \ell \in \tau & \quad \Pi \vdash \ell \in \text{unit} & \quad [\text{P-UNIT}] \\
\Pi \vdash \ell \in \tau_1 & \quad \Pi \vdash \ell \in \tau_2 & \quad [\text{P-PAIR}] \\
\Pi \vdash \ell \in \tau_1 \times \tau_2 & \quad [\text{P-FUN}] \\
\Pi \vdash \ell \in \nu \tau_1 & \quad \Pi \vdash \ell \in \nu \tau_2 & \quad [\text{P-LB}] \\
\Pi \vdash \ell \in \ell' & \quad \Pi \vdash \ell \in \ell' \text{ says } \tau & \quad [\text{P-T}]
\end{align*}
\]

Fig. 8. FLAC implementations of commitment scheme operations.

5. Examples revisited

We can now implement our examples from Section 2 in FLAC. 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 at a high level how FLAC types help enforce specific end-to-end security properties for commitment schemes and bearer credentials. Section 7 formalizes the semantic security properties of all well-typed FLAC programs.

5.1. Commitment Schemes

Figure 8 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 $N$.\(^{15}\) 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 $p^-\gamma$ and produces a value with confidentiality and integrity $p$. In other words, $p$ endorses the value to have integrity $p^-\gamma$.

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, $p^-\gamma$, may apply commit to a value.\(^{16}\) Furthermore, if the programmer omitted this constraint or instead chose $\bot^-\gamma$, say, 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 $\nabla (p^-\gamma)$, which is equal to $p^-\gamma$.

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 $\langle \nabla (q^-\gamma) \geq \nabla (p^-\gamma) \rangle$ and $\langle q^-\gamma \geq q^-\gamma \rangle$, permitting the inner function to relabel its argument from $p$ to $q^-\gamma \land p^-\gamma$. The outer assume term establishes that principals speaking for $q^-\gamma$ also speak for $p^-\gamma$ by creating an integrity relationship between their voices. With this relationship in place, the inner assume term may delegate $p$'s confidentiality to $q$.\(^{17}\)

Only principals trusted by $\nabla (p^-\gamma)$ and $p^-\gamma$ 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 $\nabla (p^-\gamma)$, 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 $q^-\gamma$. Without the delegation, however, the bind term would fail to type-check under BINDM since $p$ would not flow to $q^-\gamma \land p^-\gamma$.

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 $q^-\gamma$, $p$ 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 version of) its argument. The return type $q^-\gamma$ says $q^-\gamma \land p^-\gamma$ says $X$ protects this relationship between the committed value and the opened one: it indicates that $q$ trusts that $q^-\gamma \land p^-\gamma$ says $X$ is the one originally committed to by $p$.

In DCC, functions are not annotated with $pc$ labels and may be applied in any context. So a DCC function analogous to reveal might have type

$$\text{dcc\_reveal} : \forall N. \forall X. p \text{ says } (N, X) \rightarrow q^-\gamma \land p^-\gamma \text{ says } (N, X)$$

However, dcc\_reveal would not be appropriate for a commitment scheme since any principal could use it to relabel information from $p$-confidential ($p^-\gamma$) to $q$-confidential ($q^-\gamma$).

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

---

\(^{15}\) Any reasonable integer encoding is permissible provided that each integer $N$ is a singleton type.

\(^{16}\) We make the reasonable assumption that an untrusted programmer cannot modify high-integrity code, thus the influence of attackers is captured by the $pc$ and the protection levels of values. Enforcing this assumption is beyond the scope of FLAC, but has been explored in [6].

\(^{17}\) Specifically, it satisfies the ASSUME premise $\Pi \vdash \nabla (p^-\gamma) \geq \nabla (q^-\gamma)$. 
reveal p’s commitments.

\[ \Lambda N[q^-]. \Lambda X[q^-]. \lambda(x:p \land q^- \text{ says } X)(q^-). \text{assume } \langle q \geq p \rangle \text{ in reveal } N X x \]

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 \textsc{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 \( pc \) is rejected.

The function to reveal \( pc \) is independent of \( q \), and

\[ \Gamma_{cro} = \text{commit, reveal, open, } (\text{reveal } N'), x:p^- \text{ says } \tau, y:p \text{ says } (N, \tau), z:p \text{ says } (N', \tau) \]

Intuitively, \( pc_p \) and \( pc_q \) are execution contexts under the control of \( p \) or \( q \), respectively. \( \Gamma_{cro} \) is a typing context for programs using the commitment scheme. For presentation purposes, we have omitted the types of \text{commit}, \text{reveal}, and \text{open} in \( \Gamma_{cro} \). Their types are as defined previously. The variable \( x \) represents an uncommitted value with \( p \)'s confidentiality, whereas \( y \) and \( z \) are a committed value. The type \( (\text{reveal } N') \) is shorthand for \( \text{reveal} \) instantiated with integer type \( N' \) and represents that \( z \) has been revealed to \( q \) by \( p \).

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: \( \Pi = \emptyset \). Below, we omit the delegation context altogether for brevity.

FLAC’s noninterference guarantee helps rule out information that an attacker can influence or learn. Using results presented in Section 7, we can prove that:

- \( q \) cannot open a value that hasn’t been committed. For any \( e \) and \( \tau' \) such that \( \Gamma_{cro}; pc_q \vdash e : q^- \land p^- \text{ says } \tau' \), the result of evaluating \( e \) is independent of \( x \); specifically, for any \( v_1 \) and \( v_2 \), if \( e[x \mapsto v_1] \rightarrow^* v'_1 \) and \( e[x \mapsto v_2] \rightarrow^* v'_2 \), then \( v'_1 = v'_2 \).

- \( q \) cannot learn a value that hasn’t been revealed. For any \( e \), \( \ell \), and \( \tau' \) such that \( \Gamma_{cro}; pc_q \vdash e : \ell \cap q^- \text{ says } \tau' \), then the result of \( e \) is independent of \( x \) and \( y \).

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 \( \ell \cap q^- \) for any \( \ell \); 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 (Theorem 2). This result is strengthened by our robust declassification theorem (Theorem 1), which ensures that attacks by \( q \) on \( p \)'s programs cannot subvert the intended declassifications.
5.2. 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

$$\forall X[pc]. k^\rightarrow \text{say} X \xrightarrow{pc} k^\rightarrow \text{say} X$$

which we abbreviate as $k^\rightarrow \xrightarrow{pc} k^\rightarrow$. 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 $k^\rightarrow = k^\rightarrow \land \bot^\rightarrow$ and $k^\rightarrow = k^\rightarrow \land \bot^\rightarrow$. Then secrets protected by $k^\rightarrow$ can be declassified to $\bot^\rightarrow$, and untrusted data protected by $\bot^\rightarrow$ can be endorsed to $k^\rightarrow$. Thus this term wields the full authority of $k$, and if $pc = \bot^\rightarrow$, 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 $k^\rightarrow \xrightarrow{pc} \bot^\rightarrow$ grants the bearer authority to declassify $k$-confidential values, but no authority to endorse values.

We postpone an in-depth discussion of terms with types of the form $k^\rightarrow \xrightarrow{pc} k^\rightarrow$ until Section 6.2, but it is interesting to note that an analogous term in DCC is only well-typed if $k$ is equivalent to $\bot$. This is because the function takes an argument with $k^\rightarrow$ confidentiality and no integrity, and produces a value with $k^\rightarrow$ integrity and no confidentiality. Suppose $L$ 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 $k^\rightarrow \xrightarrow{pc} k^\rightarrow$, then $L$ must have the property $k^\rightarrow \subseteq \bot$ and $\bot \subseteq k^\rightarrow$. 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 $k^\rightarrow \xrightarrow{pc} k^\rightarrow$ 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 $\phi$ to a credential with type $k^\rightarrow \xrightarrow{pc} k^\rightarrow$, we use a wrapper:

$$\lambda(x:k^\rightarrow \xrightarrow{pc} k^\rightarrow)[pc]. \lambda(y:\phi)[pc]. xX$$

which gives us a term with type

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

This requires a term with type $\phi$ (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

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

which abbreviates to

$$k^\rightarrow \xrightarrow{\phi_1 \cdot \cdot \cdot \phi_n [pc]} k^\rightarrow$$
We will also use the syntax $(k^\mapsto k'^-)$ as an abbreviation for $\phi_1[X \mapsto \tau] \xrightarrow{pc} \ldots \xrightarrow{pc} \phi_n[X \mapsto \tau] \xrightarrow{pc} k'^- \text{ says } \tau \xrightarrow{pc} k'' \text{ says } \tau$.

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 $\text{Alice}^- \text{ says } k'^-$. 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 $pc = \text{Alice}^-$, this credential may only be used in contexts trusted by Alice: $\text{Alice}^- \text{ says } (k^\mapsto \phi_{\text{Alice}^-} k'^-)$.  

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 $\text{isEduc}$ is a predicate for educational films; it holds (has a proof term with type $\text{isEduc} X$) for types like $\text{Bio}$ and $\text{Doc}$, but not $\text{RomCom}$. Adding $\text{isEduc} X$ as a caveat to a credential would mean that the bearer of the credential could use it to access biographies and documentaries, but could not use it to access romantic comedies. Since no term of type $\text{isEduc} \text{ RomCom}$ could be applied, the bearer could only satisfy $\text{isEduc}$ by instantiating $X$ with $\text{Bio}$ or $\text{Doc}$. Once $X$ is instantiated with $\text{Bio}$ or $\text{Doc}$, the credential cannot be used on a $\text{RomCom}$ 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 a more in-depth 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, $x_1 : (k_1 \text{ says } \text{ph})$ is a variable that stores a photo (type $\text{ph}$) protected by $k_1$. For each such variable $x_1$, Alice has a credential $k_1^\mapsto k_1^-$, 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 $(k_1 \mapsto (k_1 \text{ Bob}^+ \mapsto \text{ph}^-))$. By applying this credential to $x_1$, Bob is able to access the result of $k_1^-$ 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 $k_1^- \xrightarrow{pc} k_1^-$ 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.

$$\lambda(c:k_1^- \mapsto k_1^-)[\text{Alice}^-].$$

$$\lambda(y:k_1 \text{ says } \text{ph})[\text{Bob}^-].$$

$$\text{bind } y' = y \text{ in } (c \text{ ph}) (\eta_{k \mapsto y'})$$
This function has type \((k^+_1 \xrightarrow{v} k^+_1) \xrightarrow{v} (k^+_1 \xrightarrow{v} k^+_1)\) ph: given her root credential \(k^+_1\), Alice (or someone she trusts) can create a restricted credential that allows Bob (or someone he trusts) to access values of type ph protected under \(k^+_1\).

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

\[
\lambda(x: (k^+_1 \xrightarrow{v} k^+_1) \text{ph})[\text{Bob}^+].
\]

assumes \(\langle \text{Carol}^+ \geq \text{Bob}^+ \rangle\) in

\[
\lambda((y: k^+_1 \text{ph})[\text{Carol}^+]).
\]

binds \(y' = y\) in \((c \text{ ph}) (\eta_{k \rightarrow y'})\)

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^+.

The properties of FLAC let us prove many general properties about such bearer-credential programs; here, we examine three properties. For \(i \in \{1..n\}\), let

\[
\Gamma_{bc} = \forall i: k_i \text{ says } \tau_i, c_i: \text{Alice says } (k^+_i \xrightarrow{v} k^+_i)
\]

where \(k_i\) is a primitive principal protecting the \(i\)th resource of type \(\tau_i\), and \(c_i\) is a credential for the \(i\)th resource and protected by Alice. Assume \(k_i \notin \{\text{Alice, Friends, } p\}\) for all \(i\) where \(p\) represents a (potentially malicious) user of Alice’s service, and Friends is a principal for Alice’s service, (e.g., Friends = (Bob \lor Carol)). Also, define \(pc_p = p^+\) and \(pc_A = Alice^+\).

- **p cannot access resources without a credential.** For any \(e, \ell,\) and \(\tau'\) such that \(\Gamma_{bc};pc_p \vdash e: \ell \cap p^+\) says \(\tau'\), the value of \(e\) is independent of \(x_i\) for all \(i\).

- **p cannot use unrelated credentials to access resources.** For any \(e, \ell,\) and \(\tau'\) such that

\[
\Gamma_{bc};c_p:(k^+_1 \xrightarrow{v} k^+_1);pc_p \vdash e: \ell \cap p^+\text{ says } \tau'
\]

the value \(e\) computes is independent of \(x_i\) for \(i \neq 1\).

- **Alice cannot disclose secrets by issuing credentials.** For all \(i\) and \(j \neq 1\), define

\[
\Gamma'_{bc} = \forall i: k_j \text{ says } \tau_i, c_i: \text{Alice says } (k^+_j \xrightarrow{v} k^+_j),
\]

\[
c_F: \text{Friends says } (k^+_1 \xrightarrow{v} k^+_1)
\]

Then if \(\Gamma'_{bc};pc_A \vdash e: \ell \cap p^+\) says \((k^+_j \xrightarrow{v} k^+_j)\) for some \(e, \ell,\) and \(\tau'\), the value of \(e\) is independent of \(x_1\).

These properties demonstrate the power of FLAC’s type system. The first two ensure that credentials really are necessary for \(p\) to access protected resources, even indirectly. In the first, \(p\) has no credentials, and the type system ensures that \(p\) cannot invoke a program that produces a value \(p\) can read (represented
by $\ell \rightarrow p^{-}$) that depends on any variable $x_i$. In the second, a credential $c_p$ with type $k_1 \rightarrow k_1$ is accessible to $p$, but $p$ cannot use it to access other variables. The third property eliminates covert channels like the one discussed in Section 2.2. It implies that credentials issued by Alice do not leak information, in this case about Alice’s friends. By implementing bearer credentials in FLAC, we can demonstrate these three properties with relatively little effort.

6. FLAC Proof theory

6.1. 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. [27] and Abadi [2]) is that implications that hold for top-level propositions also hold for propositions of any principal $\ell$:

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

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 $\ell$, the label $\ell$ must flow to the $pc$ of the implication. Thus, for all $\ell$ and $\tau_1$,

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

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 $\Pi$, $\Gamma$, 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

$$\lambda(f:(\tau_1 \rightarrow \tau_2))[\ell].$$
$$\lambda(x:\ell \; \text{says} \; \tau_1)[\ell].\; \text{bind} \; x' = x \; \in \; \eta_{\ell} \; f \; x'$$

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 $\ell$, and UNITM requires the label of the returned value must also be as restrictive as $\ell$. We can actually prove a slightly more general version of the above theorem:

$$\frac{(\tau_1 \rightarrow \tau_2) \rightarrow (\ell \; \text{says} \; \tau_1 \rightarrow \ell \; \text{says} \; \tau_2)}{(\tau_1 \rightarrow \tau_2) \rightarrow (\ell \; \text{says} \; \tau_1 \rightarrow \ell \; \text{says} \; \tau_2)}$$

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. Recall from Sections 5.1 and 5.2 that the security of these mechanisms relied 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 $p$ authority: $p \rightarrow \frac{pc}{\implies} p^{-}$. 
Other properties of says common to DCC and other logics (cf. [1] for examples) are similarly refined by $pc$ bounds. Two examples are: $\vdash \tau \xrightarrow{\ell} \ell$ 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:

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

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

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

Preserving 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. [?]) where $\ell$ says $\tau$ is interpreted as a value of type $\tau$ protected by encryption key $\ell^{-}$ and signing key $\ell^{+}$. Preserving the order of nested types in this context is critical for modeling decryption and verification of protected values.

6.2. Dynamic Hand-off

Many authorization logics support delegation using a “hand-off” axiom. In DCC, this axiom is actually 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 $p \sqsubseteq q$ in the security lattice. Thus, DCC can reason about the consequences of $p \sqsubseteq q$ (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 wrote as $p \xrightarrow{pc} q$ in Section 5.2. 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 1** (Dynamic hand-off). For all $\ell$ and $pc'$, let $pc = \ell^{-}\land \nabla(p^{-}) \land q^{-} \land \nabla(pc')$

$$(\nabla(q^{-}) \geq \nabla(p^{-})) \xrightarrow{pc} (p \subseteq q) \xrightarrow{pc} (pc' \subseteq q) \xrightarrow{pc},$$

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

*Proof term.*

$$\lambda(pf_1:\nabla(q^{-}) \geq \nabla(p^{-}))[pc]. \text{assume } pf_1 \text{ in }$$

$$\lambda(pf_2:(p \subseteq q))[pc]. \text{assume } pf_2 \text{ in }$$

$$\lambda(pf_3:(pc' \subseteq q))[pc]. \text{assume } pf_3 \text{ in }$$

$$\Delta X[pc']. \lambda(x:p \text{ says } X)[pc']. \text{bind } x' = x \text{ in } \eta_q x'$$

The principal $pc = \ell^{-}\land \nabla(p^{-}) \land q^{-} \land \nabla(pc')$ 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 $\nabla(p^{-})$, $q^{-}$, and $\nabla(pc')$.

The three arguments are proofs of authority with acts-for types: a proof of $\nabla(q^{-}) \geq \nabla(p^{-})$, a proof of $p \subseteq q$, and a proof of $pc' \subseteq q$. The $pc$ ensures that the proofs have sufficient integrity to be used in assume terms since it has the integrity of both $\nabla(p^{-})$ and $q^{-}$. 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 $\ell^{-}$ 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 1 simply has the fewest 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 1 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 authority. 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.
Syntax

\[ e ::= \ldots \mid \langle e \rangle_\ell \]

Evaluation contexts

\[ E ::= \ldots \mid \langle E \rangle_\ell \]

Evaluation rules

\[ \text{[E-APP\textsuperscript{*}]} \ (\lambda(x:\tau)[pc]. e) \ w \rightarrow \langle e[x \mapsto w] \rangle_{pc} \quad \text{[E-TAPP\textsuperscript{*}]} \ (\Lambda X[pc]. e) \ \tau \rightarrow \langle e[X \mapsto \tau] \rangle_{pc} \]

\[ \text{[E-BINDM\textsuperscript{*}]} \ \text{bind} \ x = \eta_\ell w \ \text{in} e \rightarrow \langle e[x \mapsto w] \rangle_\ell \quad \text{[O-CTX]} \ \langle w \rangle_\ell \rightarrow w \]

Typing rules

\[ \frac{\Pi; \Gamma\vdash pc' \vdash e : \tau \quad \Pi \vdash pc \subseteq pc'}{\Pi; \Gamma\vdash \langle e \rangle_{pc'} : \tau} \]  

Fig. 9. Extensions to support protection contexts.

7. Semantic security properties of FLAC

7.1. Trace indistinguishability

We express our semantic results in terms of the traces of a program observable to an attacker. FLAC traces are simply the sequence of terms under the \( \rightarrow \) relation. That is, each evaluation step of the form \( e \rightarrow e' \) generates a new trace element \( e' \). We write the trace generated by taking \( n \) steps from \( e \) to \( e' \) as \( e \rightarrow^* e' \), 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 the scenario where sealed values are implemented by encrypting and signing data, and computation may only be done on these values by hosts with access to the necessary cryptographic keys.\textsuperscript{18} 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.

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 \( \langle e \rangle_\ell \) denotes an intermediate expression \( e \) evaluated in a context protected at \( \ell \). We extend our grammar in Figure 2 with this notation as below:

Figure 9 presents rules for introducing and eliminating protected contexts. Rules E-APP\textsuperscript{*}, E-TAPP\textsuperscript{*}, and E-BINDM\textsuperscript{*} replace their counterparts in Figure 3 with rules that introduce a protected context based

\textsuperscript{18}DFLATE \cite{DFLATE} makes this connection with cryptographic enforcement more explicit for a distributed extension of FLAC.
O(x, Π, p, π) = x
O((a ⩾ b), Π, p, π) = O((a, Π, p, π)) = O(⟨⟩, Π, p, π) = O(ηe, Π, p, π) = O(ηe w, Π, p, π) = \{ \begin{align*} &\text{otherwise} \\
&\text{}\Pi \vdash p \geq \ell π \end{align*} \}
O(λ(x: τ)[pc]. e, Π, p, π) = \{ \begin{align*} &\Pi \vdash p \geq pc\pi \\
&\text{otherwise} \end{align*} \}
O(ΛX[pc]. e, Π, p, π) = \{ \begin{align*} &\Pi \vdash p \geq pc\pi \\
&\text{otherwise} \end{align*} \}
O(⟨ w ⟩, Π, p, π) = \{ \begin{align*} &\Pi \vdash p \geq \ell π \\
&\text{otherwise} \end{align*} \}
O(e e′, Π, p, π) = O(e, Π, p, π) O(e′, Π, p, π)
O(⟨e_1, e_2⟩, Π, p, π) = \{ \begin{align*} &\Pi \vdash e_1, e_2 \leftarrow \Pi \vdash \ell π \\
&\text{otherwise} \end{align*} \}
O(projj e, Π, p, π) = projj O(e, Π, p, π)
O(injj e, Π, p, π) = \{ \begin{align*} &\text{otherwise} \\
&\Pi \vdash e \rightarrow \Pi \vdash \ell π \end{align*} \}
O(case e of injj_1(x). e_1 | injj_2(x). e_2, Π, p, π) = \begin{align*} &\text{case } O(e, Π, p, π) \text{ of} \\
&\text{injj}_1(x). O(e_1, Π, p, π) \\
&\text{injj}_2(x). O(e_2, Π, p, π) \end{align*}
O(bind x = e in e′, Π, p, π) = bind x = O(e, Π, p, π) in O(e′, Π, p, π)
O(assume e in e′, Π, p, π) = \{ \begin{align*} &\Pi \vdash e \leftarrow \Pi \vdash \ell π \\
&\Pi \vdash e′ \leftarrow \Pi \vdash \ell π \\
&\text{otherwise} \end{align*} \}
O(e where ν, Π, p, π) = \{ \begin{align*} &\Pi \vdash e \leftarrow \Pi \vdash \ell π \\
&\Pi \vdash \nu \rightarrow \Pi \vdash \ell π \\
&\text{otherwise} \end{align*} \}

(a) Observation function for intermediate FLAC terms.

O([e], Π, p, π) = [O(e, Π, p, π)]
O([e; e′], Π, p, π) = \{ \begin{align*} &\Pi \vdash e, e′ \rightarrow \Pi \vdash \ell π \end{align*} \}
O(⟨e; e′⟩ · t, Π, p, π) = O(O([e; e′], Π, p, π) · t, Π, p, π)

(b) Observation function for traces.

Fig. 10. Observation function definitions
on the relevant label. Rule O-CTX eliminates the protected context when the intermediate expression is fully evaluated.

The FLAC programs observable to an attacker are defined by an observation function $O(e, \Pi, p, \pi)$, defined in Figure 10a. An expression $e$ is observable by $p$ in delegation context $\Pi$ depending on the authority of $p$ relative to the protected terms in $e$. The projection $\pi$ specifies whether to consider the confidentiality of protected terms ($\pi = \rightarrow$) or the integrity ($\pi = \leftarrow$). For example, sealed values such as $\overline{\ell}_{\pi} w$ are observable by principal $p$ if $p$ acts for $\ell$, otherwise they are erased. Protected contexts like $\langle e \rangle_\ell$ are treated similarly. To simplify our proofs, we collapse terms whose subterms have been erased. For example $\langle \circ, \circ \rangle$ is collapsed to $\circ$. For values with singleton types like $()$ and $\langle a \geq b \rangle$ we erase the value since no information is learned from the value itself.

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 different delegations, but they are not explicit in the output. In DFLATE [3], these delegations are explicit in the output, modeling values that carry certified justifications of why their flow was authorized by the program.

The observability of trace elements is defined in Figure 10b in terms of the observability function for terms. Note that duplicate entries (which may occur due to evaluation in protected contexts) are removed.

We now can define trace indistinguishability. Two traces $t$ and $t'$ are indistinguishable to a principal $p^{\pi}$ in delegation context $\Pi$ if the observable elements of $t$ and $t'$ are equal.

**Definition 1** (Trace indistinguishability).

$$ t \approx_{\Pi} t' \Delta O(t, \Pi, p, \pi) = O(t', \Pi, p, \pi) $$

### 7.2. Robust declassification

**Robust declassification** [45] 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. [33], we extend our set of terms with a “hole” term $\bullet$ representing portions of a program that are under the control of an attacker. We extend the type system with the following rule for holes:

\[
\Pi \vdash H^- \supseteq pc^- \quad \Pi \vdash pc^- \subseteq \Delta(H^-) \\
\Pi, \Gamma; pc \vdash \bullet : \tau
\]

Where $\Delta(H^-)$ is the view of principal $H^-$. The view of a principal is a dual notion to voice, introduced by Cecchetti et al. [15] to represent the confidentiality observable to a principal. Given a principal in normal form $q^- \land r^-$, the view of that principal is

$$ \Delta(q^- \land r^-) \equiv q^- \land r^- $$
In other words $\Delta(H^-)$ represents the confidentiality of information observable to the attacker $H^-$. Therefore, the HOLE premises $\Pi \vdash H^- \Rightarrow \text{pc}^-$ and $\Pi \vdash \text{pc}^- \subseteq \Delta(H^-)$ require that holes be inserted only in contexts that are controlled by and observable to the attacker.

We write $e[\bullet]$ 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 $\bullet_i$.

**Definition 2** (\Pi-fair attacks). An attack $\vec{a}$ is a \Pi-fair attack on a well-typed program with holes $\Pi; \Gamma; \text{pc} \vdash e[\vec{a}] : \tau$ if the program $e[\vec{a}]$ is also well typed (thus $\Pi; \Gamma; \text{pc} \vdash e[\vec{a}] : \tau$), and furthermore, for each $a_i \in \vec{a}$, we have $\Pi; \Gamma'; \text{pc} \vdash [a_i] : \tau'$ where each entry in $\Gamma'$ has the form $y: \ell$ says $\tau''$ and $\forall y: \ell$ says $\tau'' \in \Gamma'$. $\Pi \vdash \ell \rightarrow \subseteq \Delta(H^-)$.

By specifying the relationships between confidential information labeled $H^-$ and the integrity of the attacker $H^-$, we can precisely express the authority an attacker is able to wield in its \Pi-fair attacks against protected information. Proposition 2 states the conditions under which a \Pi-fair attack is noninterfering. Condition 1 specifies that $\tau'$ contains confidential information (labeled $H^-$), Condition 2 specifies that this information is not visible to the attacker (a principal with integrity $H^-$) since it does not flow to the attacker’s view. Finally, Condition 3 specifies that the attacker $H^-$ has insufficient integrity to declassify information at label $H^-$. 

**Proposition 2** (Noninterfering fair attacks). For $H$ such that

1. $\Pi \vdash H^- \subseteq \tau'$
2. $\Pi \nvDash H^- \subseteq \Delta(H^-)$
3. $\Pi \nvDash H^- \Rightarrow \nabla(H^-)$,

let $\Pi; \Gamma; x : \tau', \Gamma' ; \text{pc} \vdash \bullet : \tau$. Then for all \Pi-fair attacks $a$ where $\Pi' ; \Gamma, x : \tau', \Gamma' ; \text{pc} \vdash a : \tau$ for $\Pi' \supseteq \Pi$, and $a[x \mapsto v_i] \rightarrow a'_i$, $a'_1 = a_2$.

**Proof.** By Definition 2, there exists some $\Gamma''$ such that $\Pi; \Gamma''; \text{pc} \vdash a_i : \tau'$ where $\forall y : \tau'' \in \Gamma'$. $\Pi \vdash \tau'' \subseteq \Delta(H^-)$. Therefore since $\Pi \vdash H^- \subseteq \tau'$, we have $x \notin \Gamma''$ and thus $a[x \mapsto v_i] = a$. Consequently, $a'_1 = a_2$. 

In other words, \Pi-fair attacks must be well-typed on variables observable to the attacker in delegation context $\Pi$. Under the three conditions in Proposition 2, these attacks are noninterfering. Unfair or interfering attacks give the attacker enough power to break security directly by creating new declassifications without exploiting existing ones. Restricting attacks to noninterfering $\Pi$-fair attacks also rule out nonsensical scenarios such as when the “attacker” has the authority to read the confidential information. Noninterfering fair attacks represent an upper-bound on the power of the attacker over low-integrity portions of the program.

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

**Theorem 1** (Robust declassification). Suppose $\Pi; \Gamma, x : \tau', \Gamma' ; \text{pc} \vdash e[\bullet] : \tau$. For $\Pi_H$ and $H$ such that

1. $\Pi_H \vdash H^- \subseteq \tau'$
2. $\Pi_H \nvDash H^- \subseteq \Delta(H^-)$
3. $\Pi_H \nvDash H^- \Rightarrow \nabla(H^-)$,
Then for all $\Pi_{H}$-fair attacks $\alpha_{1}$ and $\alpha_{2}$ such that $\Pi; \Gamma, x:\tau', \Gamma': pc \vdash e[\alpha_{1}] : \tau$, if $e[\alpha_{j}][x \mapsto v_{i}] \rightarrow e'_{ij}$ for $i, j \in \{1, 2\}$, then the for the traces $t_{ij} = (e[\alpha_{j}][x \mapsto v_{i}]) : e'_{ij}$, we have

$$t_{11} \equiv^{H}_{\Delta(H^{-})} t_{21} \iff t_{12} \equiv^{H}_{\Delta(H^{-})} t_{22}$$

Proof. By induction on the evaluation of $e$.

**Case E-APP** ($e = (\lambda(y : \tau'')[pc'], e'')[a_{i}]$): By Proposition 2 $x$ is not free in $a_{i}$, therefore

$$e[x \mapsto v_{i}] = (\lambda(y : \tau'')[pc'], e''[x \mapsto v_{i}])[a_{i}]$$

Suppose $e''[x \mapsto v_{1}] \equiv^{H}_{\Delta(H^{-})} e''[x \mapsto v_{2}]$ then

$$(\lambda(y : \tau'')[pc'], e''[x \mapsto v_{1}])[a_{j}] \equiv^{H}_{\Delta(H^{-})} (\lambda(y : \tau'')[pc'], e''[x \mapsto v_{2}])[a_{j}]$$

Since $v_{1}$ and $v_{2}$ are closed, we also have

$$e''[x \mapsto v_{1}][y \mapsto a_{j}] \equiv^{H}_{\Delta(H^{-})} e''[x \mapsto v_{2}][y \mapsto a_{j}]$$

By a symmetric argument, we also have that $e''[x \mapsto v_{1}] \not\equiv^{H}_{\Delta(H^{-})} e''[x \mapsto v_{2}]$ implies

$$e''[x \mapsto v_{1}][y \mapsto a_{j}] \not\equiv^{H}_{\Delta(H^{-})} e''[x \mapsto v_{2}][y \mapsto a_{j}]$$

Then it remains to show that if $\mathcal{O}(\lambda(y : \tau'')[pc'], e'', \Pi, \Delta(H^{-}), \rightarrow) = \circ$ then

$$\mathcal{O}(\langle e''[x \mapsto v_{1}][y \mapsto a_{1}] \rangle_{pc'}, \Pi, \Delta(H^{-}), \rightarrow) = \mathcal{O}(\langle e''[x \mapsto v_{2}][y \mapsto a_{1}] \rangle_{pc'}, \Pi, \Delta(H^{-}), \rightarrow)$$

$$\iff$$

$$\mathcal{O}(\langle e''[x \mapsto v_{1}][y \mapsto a_{2}] \rangle_{pc'}, \Pi, \Delta(H^{-}), \rightarrow) = \mathcal{O}(\langle e''[x \mapsto v_{2}][y \mapsto a_{2}] \rangle_{pc'}, \Pi, \Delta(H^{-}), \rightarrow)$$

This holds by the definition of $\mathcal{O}$ since $\mathcal{O}(\lambda(y : \tau'')[pc'], e'', \Pi, \Delta(H^{-}), \rightarrow) = \circ$ implies that $\Pi \not\models \Delta(H^{-}) \models pc'$.

**Case E-APP** ($e = a_{j} x$): If $\mathcal{O}(v_{1}, \Pi, \Delta(H^{-}), \rightarrow) \not\equiv \mathcal{O}(v_{2}, \Pi, \Delta(H^{-}), \rightarrow)$ then by definition $t[0]_{1j} \equiv^{H}_{\Delta(H^{-})} t[0]_{2j}$. Thus $t_{1j} \equiv^{H}_{\Delta(H^{-})} t_{2j}$. Therefore assume $\mathcal{O}(v_{1}, \Pi, \Delta(H^{-}), \rightarrow) = \mathcal{O}(v_{2}, \Pi, \Delta(H^{-}), \rightarrow)$.

Since $a_{j}$ has the form $\lambda(y : \tau)[pc']$, $e''_{j}$, we need to show that

$$\mathcal{O}(\langle e''_{1}[y \mapsto v_{1}] \rangle_{pc'}, \Pi, \Delta(H^{-}), \rightarrow) = \mathcal{O}(\langle e''_{1}[y \mapsto v_{2}] \rangle_{pc'}, \Pi, \Delta(H^{-}), \rightarrow)$$

$$\iff$$

$$\mathcal{O}(\langle e''_{2}[y \mapsto v_{1}] \rangle_{pc'}, \Pi, \Delta(H^{-}), \rightarrow) = \mathcal{O}(\langle e''_{2}[y \mapsto v_{2}] \rangle_{pc'}, \Pi, \Delta(H^{-}), \rightarrow)$$

Without loss of generality, assume $\Pi \models \Delta(H^{-}) \models pc'$. Then since $\mathcal{O}(v_{1}, \Pi, \Delta(H^{-}), \rightarrow) = \mathcal{O}(v_{2}, \Pi, \Delta(H^{-}), \rightarrow)$, by the definition of $\mathcal{O}$, we also have

$$\mathcal{O}(\langle e''_{1}[y \mapsto v_{1}] \rangle_{pc'}, \Pi, \Delta(H^{-}), \rightarrow) = \mathcal{O}(\langle e''_{1}[y \mapsto v_{2}] \rangle_{pc'}, \Pi, \Delta(H^{-}), \rightarrow)$$
Case E-TAPP*: Similar to E-APP*.
Case O-CTX: \(e = \{a_j\}_E\) By Proposition 2.
Case O-CTX: \(e = \{x\}_E\) Trivial.
Case E-CASE \(e = \text{case } x \text{ of inj}_1(y). a_j \mid \text{inj}_2(y). a_j\): Not well typed since \(x\) cannot be a sum type.
Case E-CASE \(e = \text{case } a_j \text{ of inj}_1(y). e_1 \mid \text{inj}_2(y). e_2\): Observe

\[
e[x \mapsto v_i] = \text{case } a_j \text{ of inj}_1(y). e_1[x \mapsto v_i] \mid \text{inj}_2(y). e_2[x \mapsto v_i]
\]

Suppose

\[
\begin{aligned}
\text{case } a_j \text{ of inj}_1(y). e_1[x \mapsto v_1] & \mid \text{inj}_2(y). e_2[x \mapsto v_1] \\
\approx^P_{\Delta(H^{-})}
\end{aligned}
\]

\[
\begin{aligned}
\text{case } a_j \text{ of inj}_1(y). e_1[x \mapsto v_2] & \mid \text{inj}_2(y). e_2[x \mapsto v_2] \\
\end{aligned}
\]

then for \(k \in \{1, 2\}\) we have

\[
e_k[x \mapsto v_1] \approx^P_{\Delta(H^{-})} e_k[x \mapsto v_2]
\]

therefore since \(v_1\) and \(v_2\) are closed, we also have

\[
e_k[x \mapsto v_1][y \mapsto a_j] \approx^P_{\Delta(H^{-})} e_k[x \mapsto v_2][y \mapsto a_j]
\]

Now suppose

\[
\begin{aligned}
\text{case } a_j \text{ of inj}_1(y). e_1[x \mapsto v_1] & \mid \text{inj}_2(y). e_2[x \mapsto v_1] \\
\not\approx^P_{\Delta(H^{-})}
\end{aligned}
\]

\[
\begin{aligned}
\text{case } a_i \text{ of inj}_1(y). e_1[x \mapsto v_2] & \mid \text{inj}_2(y). e_2[x \mapsto v_2] \\
\end{aligned}
\]

then by definition \(t[0]_{1j} \not\approx^P_{\Delta(H^{-})} t[0]_{2j}\). Thus \(t_{1j} \not\approx^P_{\Delta(H^{-})} t_{2j}\).

Case E-BINDM* \((e = \text{bind } y = [a_j] \text{ in } e)\) Follows by an argument similar to that for E-App.

Case E-BINDM* \((e = \text{bind } y = x \text{ in } e[a_j])\) By BINDM (wlog), \(\Pi; \Gamma, x : H^{-} \text{ says } \tau', \Gamma', y : \tau'' ; p = pc \land H^{-} \vdash [a_j] : \tau \) and by HOLE, \(\Pi \vdash pc \land H^{-} \models \Delta(H^{-})\). Therefore the value of \(x\) is distinguishable by \(\Delta(H^{-})\) and \(t[0]_{1j} \not\approx^P_{\Delta(H^{-})} t[0]_{2j}\).

Case E-ASSUME \((e = \text{assume } (p \not\geq q) \text{ in } e[a_j])\): By Proposition 2, \(x\) is not free in \(a_j\), therefore \(e[a_j][x \mapsto v_1] = e[x \mapsto v_1][a_j]\). If

\[
O(e[x \mapsto v_1][a_j], \Pi, \Delta(H^{-}), \rightarrow) \neq O(e[x \mapsto v_2][a_j], \Pi, \Delta(H^{-}), \rightarrow)
\]

then by definition \(t_{1j} \not\approx^P_{\Delta(H^{-})} t_{2j}\). If instead we have

\[
O(e[x \mapsto v_1][a_j], \Pi, \Delta(H^{-}), \rightarrow) = O(e[x \mapsto v_2][a_j], \Pi, \Delta(H^{-}), \rightarrow)
\]
then we need to prove that
\[ O(e'_1, \Pi, \Delta(H^-), \rightarrow) = O(e'_2, \Pi, \Delta(H^-), \rightarrow) \]
where \( e' \) is of the form \( e'_{ij} = e[x \mapsto v_j][a_j] \) where \( \langle p \geq q \rangle \). Since \( x \) is not free in \( a_j \), if
\[ O(e[x \mapsto v_1][a_j], \Pi \cdot \langle p \geq q \rangle, \Delta(H^-), \rightarrow) = O(e[x \mapsto v_2][a_j], \Pi \cdot \langle p \geq q \rangle, \Delta(H^-), \rightarrow) \]
for \( j = 1 \), then it holds for \( j = 2 \) as well. Similarly, if
\[ O(e[x \mapsto v_1][a_j], \Pi \cdot \langle p \geq q \rangle, \Delta(H^-), \rightarrow) \neq O(e[x \mapsto v_2][a_j], \Pi \cdot \langle p \geq q \rangle, \Delta(H^-), \rightarrow) \]
for \( j = 1 \), then it holds for \( j = 2 \).

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.

Our formulation of robust declassification is in some sense more general than previous definitions since it permits some endorsements. Previous definitions of robust declassification [33, 45] forbid endorsement altogether; qualified robustness [33] permits endorsement but offers only possibilistic security.

Theorem 1 permits some declassifications that prior definitions reject. For example, our definition admits declassifications of \( x \) even if \( \Pi \vdash H^- \subseteq \tau' \). 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 [15], which neither FLAC nor prior definitions (in the presence of endorsement) prevent in general. Cecchetti et al. [15] present a language based on FLAC that replaces assume with restricted declassification and endorsement terms to enforce nonmalleable information flow.

7.3. Bracketed semantics

Theorem 1 demonstrates that the indistinguishability of FLAC program traces is invariant under fair attacks. However, this result is only meaningful if FLAC programs that process protected information actually generate indistinguishable traces when we expect them to. These expectations are formalized in our noninterference theorem, but to understand and prove that theorem, we need additional formal tools.

Our noninterference proof is based on the bracketed semantics approach used by Pottier and Simonet [36]. 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 [17], as type safety in the extended language.

Figure 11 presents the bracket extensions for FLAC. Where-values \( w \) and expressions \( e \) may be bracketed. New B-* evaluation rules propagate brackets out of subexpressions and ensure bracketed terms do not get stuck unless the unbracketed terms do. B-STEP evaluates expressions inside of brackets.
Syntax

\[ w ::= \ldots \mid (w \mid w) \]
\[ e ::= \ldots \mid (e \mid e) \]

Evaluation rules

\[ \text{[B-STEP]} \quad e_i \rightarrow e'_i \quad e_j = e_j \quad \{i,j\} = \{1,2\} \quad (e_1 | e_2) \rightarrow (e'_1 | e'_2) \]
\[ \text{[B-APP]} \quad (w_1 \mid w_2) \rightarrow (w_1 | w_1 | w_2 \mid w_2) \]
\[ \text{[B-TAPP]} \quad (w \mid w') \tau \rightarrow (w \tau \mid w' \tau) \]
\[ \text{[B-UNPAIR]} \quad \text{proj}_i (w \mid w') \rightarrow (\text{proj}_i w \mid \text{proj}_i w') \]
\[ \text{[B-BINDM]} \quad \text{bind } x = (w \mid w') \quad (\text{bind } x = w \text{ in } e_1 \mid \text{bind } x = w' \text{ in } e_2) \]
\[ \text{[B-CASE]} \quad \{i,j\} = \{1,2\} \quad \text{case } (w \mid w') \text{ of inj}_1(x). e_1 \mid \text{inj}_2(x). e_2 \rightarrow (\text{case } w \text{ of inj}_1(x). [e_1]_1 \mid \text{inj}_2(x). [e_2]_1 \mid \text{case } w' \text{ of inj}_1(x). [e_1]_2 \mid \text{inj}_2(x). [e_2]_2) \]
\[ \text{[B-ASSUME]} \quad \text{assume } (w \mid w') \quad (\text{assume } w \text{ in } e_1 \mid \text{assume } w' \text{ in } e_2) \]

Typing rules

\[ \text{[BRACKET]} \quad e_1 = v_1 \iff e_2 \neq v_2 \quad \Pi \vdash (H^\pi \sqcup pc) \subseteq pc' \quad \Pi; \Gamma; pc' \vdash e_1 : \tau \quad \Pi; \Gamma; pc' \vdash e_2 : \tau \quad \Pi \vdash H^\pi \subseteq \tau \quad \Pi; \Gamma; pc \vdash (e_1 \mid e_2) : \tau \]
\[ \text{[BRACKET-VALUES]} \quad \Pi \vdash H^\pi \subseteq \tau \quad \Pi; \Gamma; pc \vdash w_1 : \tau \quad \Pi; \Gamma; pc \vdash w_2 : \tau \quad \Pi; \Gamma; pc \vdash (w_1 \mid w_2) : \tau \]

Observation function

\[ O((e_1 \mid e_2), \Pi, p, \pi) = \begin{cases} \circ & O(e_1, \Pi, p, \pi) = \circ \\ (O(e_1, \Pi, p, \pi) \mid O(e_2, \Pi, p, \pi)) & \text{otherwise} \end{cases} \]

Fig. 11. Extensions for bracketed semantics

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 \(H^\pi \sqcup pc\) and \(pc'\) and have a type \(\tau\) protects \(H^\pi\). BRACKET-VALUES relaxes the restriction on \(pc\) for where-values.

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 execu-
tion. By completeness, we mean that given a left and right execution, we can construct a corresponding bracketed execution. We now prove the soundness of the bracketed semantics.

**Lemma 1 (Soundness).** If $e \rightarrow e'$ then $[e]_k \rightarrow^* [e']_k$ for $k \in \{1, 2\}$.

**Proof.** By induction on the evaluation of $e$. Observe that all bracketed rules in Figure ?? except B-STEP only expand brackets, so $[e]_k = [e']_k$ for $k \in \{1, 2\}$. For B-STEP, $[e]_j \rightarrow [e']_j$ and $[e]_j = [e']_j$. □

Before proving completeness, we prove if the bracketed term is stuck then either left or right execution are stuck.

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

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

**Lemma 3 (Completeness).** If $[e]_1 \rightarrow^* v_1$ and $[e]_2 \rightarrow^* v_2$, then there exists some $v$ such that $e \rightarrow^* v$.

**Proof.** Assume $[e]_1 \rightarrow^* v_1$ and $[e]_2 \rightarrow^* v_2$. The extended set of rules in Figure ?? always move brackets out of subterms, and therefore can only be applied a finite number of times. Therefore, by Lemma 1, if $e$ diverges, either $[e]_1$ or $[e]_2$ diverge; this contradicts our assumption.

Furthermore, by Lemma 2, if the evaluation of $e$ gets stuck, either $[e]_1$ or $[e]_2$ gets stuck. Therefore, since we assumed $[e]_1 \rightarrow^* v_1$, then $e$ must terminate. Thus, there exists some $v$ such that $e \rightarrow^* v$. □

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 $H^{-}$), 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 $e \rightarrow e'$ then $\Pi; \Gamma; pc \vdash e : \tau$.

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

### 7.4. Delegation 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 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 \( a \land b \). The voice of \( a \land b \), \( \nabla(a \land b) \), is sufficient integrity to add a delegation \( a \land b \) to \( a \) so that \( a \geq a \land b \). Alternatively, having only the integrity of \( \nabla(b) \) is sufficient to add a delegation \( a \geq b \), which also results in \( a \geq a \land b \). To precisely characterize which trust relationships cannot be added by a program, we need to identify this minimal integrity \( \nabla(b) \) given the pair of principals \( a \) and \( a \land b \). 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 3 (Principal Equivalence).** We say that two principals \( p \) and \( q \) are *equivalent* in \( \Pi \), denoted \( \Pi \vdash p \equiv q \), if

\[
\Pi \vdash p \geq q \text{ and } \Pi \vdash q \geq p.
\]

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 \( q_s \land q_d \) such that \( p \geq q_s \) in the desired context. Note that \( p \) need not act for \( q_d \).

**Definition 4 (Factorization).** A \( \Pi \)-factorization of an ordered pair of principals \((p, q)\) is a tuple \((p, q_s, q_d)\) such that \( \Pi \vdash q \equiv q_s \land q_d \) and \( \Pi \vdash p \geq q_s \).

A factorization is *static* if \( \Pi = \emptyset \) (and thus \( \mathcal{L} \vdash p \geq q_s \)).

Finally, the 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 5 (Minimal Factorization).** A \( \Pi \)-factorization \((p, q_s, q_d)\) of \((p, q)\) is *minimal* if for any \( \Pi \)-factorization \((p, q'_s, q'_d)\) of \((p, q)\),

\[
\Pi \vdash q_s \geq q'_s \text{ and } \Pi \vdash q'_d \geq q_d.
\]

The minimal factorization \((p, q_s, q_d)\) of \( p \) and \( q \) for a given \( \Pi \) and \( p c \) identifies the authority necessary to cause \( p \) to act for \( q \). Because \( q_s \) is the principal with the greatest authority such that \( p \geq q_s \) and \( q \equiv q_s \land q_d \), then speaking for \( q_d \) is sufficient authority to cause \( p \) to act for \( q \) since adding the delegation \( p \geq q_d \) would imply that \( p \geq q \). This intuition also matches with the fact that \( \Pi \vdash p \geq q_d \) if and only if \( q_d = \perp \), which is the case if and only if \( \Pi \vdash p \geq q \). Observe also that minimal \( \Pi \)-factorizations are also trivially unique up to equivalence.

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 6 (Principal Subtraction).** Let \((p, q_s, q_d)\) be the minimal \( \Pi \)-factorization of \((p, q)\). We define \( q - p \) in \( \Pi \) 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.

Lemma 5 proves that minimal factorizations exist for all contexts and principals, so principal subtract is well defined.

**Lemma 5 (Minimal Factorizations Exist).** *For any context \( \Pi \) and principals \( p, q \), there exists a minimal \( \Pi \)-factorization of \((p, q)\).*
Proof. Given \((p, q)\), we first let \(q_s = p \lor q\). By definition, \(\Pi \vdash p \Rightarrow p \lor q\), and for all factorizations \((p, q_s, q_{d'})\), \(\Pi \vdash p > q_s\) and \(\Pi \vdash q > q_s\), so \(\Pi \vdash q_s > q_s\).

Now let \(D = \{r \in \mathcal{L} \mid \Pi \vdash q \equiv q_s \land r\}\). All principals in \(\mathcal{L}\) can be represented as a finite set of meets and joins of names in \(\mathcal{N}\), so \(q_s\) and \(q_{d'}\) are finite. \(\Pi\) is also finite, adding only finitely-many dynamic equivalences, so \(D\) is finite up to equivalence. Moreover, since \(\Pi \vdash q \equiv (p \lor q) \land q\) (by absorption) we have \(q \in D\). Therefore \(D\) is always non-empty and we can define \(q_{d} = \sqrt{D}\).

Now let \((p, q_s, q_{d'})\) be any \(\Pi\)-factorization of \((p, q)\). We must show that \(\Pi \vdash q_{d}' > q_{d}\).

First, see that \(\Pi \vdash q = q_s \land q_{d'}\), for one direction, observe that \(\Pi \vdash q > q_s\) and \(\Pi \vdash q > q_{d}\) (by Definition 3). For the other direction, since \(\Pi \vdash q_s > q'_s\), we have \(\Pi \vdash q_s \land q'_{d} > q'_s \land q_{d}\), so \(\Pi \vdash q_s \land q'_{d} > q\).

Therefore, by the definition of \(D\), we know \(q_{d}' \in D\), so by the definition of \(\lor\) and \(q_{d}\), \(\Pi \vdash q_{d}' > q_{d}\). Thus \((p, q_s, q_{d})\) is a minimal \(\Pi\)-factorization of \((p, q)\). \(\square\)

We can now state precisely which trust relationships may change in a given information flow context.\(^{19}\)

Lemma 6 (Delegation Invariance). Let \(\Pi; \Gamma; pc \vdash e : \tau\) such that \(e \rightarrow^{*} e'\) where \(v\). Then

1. There exist \(r, t \in \mathcal{L}\) and \(\Pi' = \Pi; \langle r \Rightarrow t \rangle\) such that \(\Pi; \Gamma; pc \vdash v : (r \Rightarrow t)\), \(\Pi \vdash pc \Rightarrow \nabla(t)\) and \(\Pi'; \Gamma; pc \vdash e' : \tau\).

2. For all principals \(p\) and \(q\), if \(\Pi \not\vdash p > q\) and \(\Pi \not\vdash pc > \nabla(q - p)\), then \(\Pi' \not\vdash p > q\).

Proof. We prove the first part using subject reduction (Lemma 4) to get \(\Pi; \Gamma; pc \vdash e'\) where \(v : \tau\). Then, from WHERE we have \(\Pi'; \Gamma; pc \vdash e' : \tau\).

We prove the second part by contradiction. Assume that

\[
\Pi' \vdash p > q
\]

(1)

Let \((p, q_s, q_{d'})\) be the minimal \(\Pi\)-factorization of \((p, q)\). So, \(\Pi \vdash p > q_s\). Since \(\Pi \not\vdash p > q\), this implies that \(\Pi \not\vdash p > q_{d}\), but from (1), we have that

\[
\Pi' \vdash p > q_{d}\]

(2)

So any derivation of (2) must involve the derivation of \(\Pi' \vdash r \Rightarrow t\) via R-ASSUME, since R-ASSUME is the only rule that uses the contents of \(\Pi\). Therefore, without loss of generality, we can assume that either (a) \(\Pi \vdash t \geq q_{d}\), or (b) for some \(q_1\) and \(q_2\) such that \(\Pi \vdash q_{d} = q_1 \land q_2\), \(\Pi \vdash t \geq q_1\) and \(\Pi \not\vdash t \geq q_2\); otherwise \(\Pi' \vdash r \Rightarrow t\) is unnecessary to prove (2).

In fact, it must be the case that \(\Pi \vdash t \geq q_{d}\) (or equivalently \(\Pi \vdash t \geq q_1 \land q_2\)). To see why, assume \(\Pi \not\vdash t \geq q_1 \land q_2\): specifically, \(\Pi \vdash t \geq q_1\) and \(\Pi \not\vdash t \geq q_2\). Since \(\Pi' \vdash p \geq q_s \land (q_1 \land q_2)\) but \(\Pi \not\vdash t \geq q_2\), it must be that \(\Pi \not\vdash p \geq q_s \land q_2\), otherwise \(\Pi' \vdash p \geq q_s \land (q_1 \land q_2)\) wouldn't hold.

Therefore \((p, q_s \land q_1, q_2)\) is a \(\Pi\)-factorization of \((p, q)\). Since \((p, q_s, q_{d'})\) is a minimal \(\Pi\)-factorization, we have that \(\Pi \vdash q_2 \geq q_{d}\). From R-TRANS, we now have that \(\Pi \vdash t \geq q_{d}\), but since we assumed \(\Pi \not\vdash t \geq q_{d}\), we have a contradiction. Hence \(\Pi \vdash t \geq q_{d}\).

\(^{19}\)The original delegation invariance lemma [5] 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.
By the monotonicity of $\nabla(\cdot)$ with respect to $\triangleright$ (Proven in Coq for FLAM [7]), we have $\Pi \vdash \nabla(t) \triangleright \nabla(qd)$. As shown above, $\Pi \vdash pc \triangleright \nabla(t)$, so by R-TRANS, $\Pi \vdash pc \triangleright \nabla(qd)$. However, recall that $\Pi \not\vdash q - p \equiv qd$ (Definition 6) contradicting our assumption that $\Pi \not\vdash pc \triangleright \nabla(q - p)$.

Therefore, we obtain that $\Pi \not\vdash p \triangleright q$.

\section{Noninterference}

Lemma 6 is 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 [36]. 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 [36], 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 $\Pi; \Gamma, x : s; pc \vdash e : \tau'$. Furthermore, let $H$ be a principal such that $\Pi \vdash H \subseteq \tau$ and $\Pi \not\vdash H \subseteq \tau'$. 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 [36], 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 $\Pi'; pc \vdash H \subseteq \tau'$ to hold in a delegation context $\Pi'$ of a subterm of $e$.

The key to proving our result relies on using Lemma 6 to constrain the assumptions that can be added to $\Pi'$. Thus noninterference in FLAC is dependent on $H$ and its relationship to $pc$ and the type $\tau'$. For confidentiality, most of this reasoning occurs in the proof of Lemma 7, which does most of the heavy lifting for the noninterference proof. Specifically, Lemma 7 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 $[e]$ observable to $\ell \rightarrow$ under $\Pi$ is equal to the observable portion of $[e]_2$, then if $e \rightarrow^* e'$, the observable portions of $e'$ are still equal. A corresponding lemma for integrity holds by a similar argument.

**Lemma 7** (Erasure conservation). Suppose $\Pi; \Gamma; pc \vdash e : \tau$, then for some $H$ and $\ell$ such that $\Pi \not\vdash pc \triangleright \nabla(H \rightarrow - \ell \rightarrow)$, if $O([e]_1, \Pi, \ell \rightarrow, \rightarrow) = O([e]_2, \Pi, \ell \rightarrow, \rightarrow)$, then $e \rightarrow^* e'$ implies $O([e']_1, \Pi, \ell \rightarrow, \rightarrow) = O([e']_2, \Pi, \ell \rightarrow, \rightarrow)$.

**Proof.** By induction on the evaluation of $e$, using the definition of $O$ and Lemma 6.

**Case E-APP*:** We have $O(\lambda(x : \tau)[pc'], e', w)|_{pc'}_1, \Pi, \ell \rightarrow, \rightarrow) = O(\lambda(x : \tau)[pc'], e', w)|_{pc'}_2, \Pi, \ell \rightarrow, \rightarrow)$.

We have to show that $O(\lambda(x : \tau)[pc'], e', w)|_{pc'}_1, \Pi, \ell \rightarrow, \rightarrow) = O(\lambda(x : \tau)[pc'], e', w)|_{pc'}_2, \Pi, \ell \rightarrow, \rightarrow)$.

Depending on the observability of $pc'$, we have two subcases.

**Case PC $\not\vdash \ell \rightarrow \triangleright pc'$:** Here $O([w]|_{pc'}_1, \Pi, \ell \rightarrow, \rightarrow) = O([w]|_{pc'}_2, \Pi, \ell \rightarrow, \rightarrow)$ and $O([e']_1, \Pi, \ell \rightarrow, \rightarrow) = O([e']_2, \Pi, \ell \rightarrow, \rightarrow)$. And so, $O(\lambda(x : \tau)[pc'], e', w)|_{pc'}_1, \Pi, \ell \rightarrow, \rightarrow) = O(\lambda(x : \tau)[pc'], e', w)|_{pc'}_2, \Pi, \ell \rightarrow, \rightarrow)$.

**Case PC $\vdash \ell \rightarrow \triangleright pc'$:** It remains to show that if $O(\lambda(x : \tau)[pc'], e', w)|_{pc'}_1, \Pi, \ell \rightarrow, \rightarrow) = \circ$ then $O(\lambda(x : \tau)[pc'], e', w)|_{pc'}_2, \Pi, \ell \rightarrow, \rightarrow) = \circ$. This holds by the definition of $O$ since $O(\lambda(x : \tau)[pc'], e', \ell \rightarrow, \rightarrow) = \circ$ implies that $\Pi \not\vdash \ell \triangleright pc'$. **Case E-TAPP*: Similar to the E-APP* case.
Case **E-BINDM**: We have

\[ \mathcal{O}(\text{bind } x = e_1 w \text{ in } e', 1, \Pi, \ell, \rightarrow) = \mathcal{O}(\text{bind } x = e_2 w \text{ in } e', 2, \Pi, \ell, \rightarrow) \]

**Case II** \( \not\vdash \ell \rightarrow \ell' \): Here \( \mathcal{O}(\langle e' \rangle, 1, \Pi, \ell, \rightarrow) = \mathcal{O}(\langle e' \rangle, 2, \Pi, \ell, \rightarrow) \) and \( \mathcal{O}(\langle w \rangle, 1, \Pi, \ell, \rightarrow) = \mathcal{O}(\langle w \rangle, 2, \Pi, \ell, \rightarrow) \).

**Case II** \( \vdash \ell \rightarrow \ell' \): It remains to show that if \( \mathcal{O}(\langle \eta e, w \rangle, 1, \Pi, \ell, \rightarrow) = \circ \) then \( \mathcal{O}(\langle e [x \mapsto w] \rangle, 1, \Pi, \ell, \rightarrow) = \circ \). This holds by the definition of \( \mathcal{O} \) since \( \mathcal{O}(\langle \eta e, w \rangle, 1, \Pi, \ell, \rightarrow) = \circ \) implies that \( \Pi \not\vdash \ell \rightarrow \ell' \).

**Case O-CTX**: Given \( \mathcal{O}(\langle w \rangle, e_1, 1, \Pi, \ell, \rightarrow) = \mathcal{O}(\langle w \rangle, e_2, 2, \Pi, \ell, \rightarrow) \). We have to prove that

\[ \mathcal{O}(\langle w \rangle, 1, \Pi, \ell, \rightarrow) = \mathcal{O}(\langle w \rangle, 2, \Pi, \ell, \rightarrow) \]

We have two cases: either \( \mathcal{O}(\langle w \rangle, e_1, 1, \Pi, \ell, \rightarrow) \neq \circ \) or \( \mathcal{O}(\langle w \rangle, e_2, 1, \Pi, \ell, \rightarrow) = \circ \). If \( \mathcal{O}(\langle w \rangle, e_1, 1, \Pi, \ell, \rightarrow) = \circ \), then by induction on the structure of \( w \), we will show that \( \mathcal{O}(\langle w \rangle, 1, \Pi, \ell, \rightarrow) = \circ \). By **CTX**, we know \( \Pi; \Gamma; \rho \vdash w : \tau \). If \( \langle w \rangle, e_1 \neq \langle w \rangle, e_2 \), then \( w \) contains a bracket subexpression \( (w_1 | w_2) \). The “root” values \( v_i \) of \( w_i \) have one of the following forms:

**Subcase** \( v = \lambda (x : \tau)[p] \): By Lemma 4, **WHERE**, and **LAM**, we have \( \Pi; \Gamma; \rho \vdash \lambda (x : \tau)[p] \). \( e' : \tau \vdash \tau'' \) (thus \( \tau = \tau' \vdash \tau'' \)), where \( \Pi' \supseteq \Pi \). By **BRACKET-VALUE** and **P-LBLN**, \( \Pi' \vdash H \rightarrow p \vdash \Pi' \vdash H \rightarrow \tau'' \). By assumption, we have \( \Pi \not\vdash \rho \vdash \tau \). Therefore we know that \( \Pi \not\vdash \ell \rightarrow \tau \) since otherwise (by Definition 6) \( \Pi \vdash \Pi \vdash \Pi' \vdash H \rightarrow \ell \rightarrow \tau'' \) \( \Pi' \vdash \Pi' \vdash \Pi' \vdash H \rightarrow \ell \rightarrow \tau'' \). By the definition of \( \mathcal{O} \), we have \( \mathcal{O}(\lambda (x : \tau)[p], e', \Pi', \ell, \rightarrow) = \circ \) and so \( \mathcal{O}(w, \Pi, \ell, \rightarrow) = \circ \) (See the case where for the **O** function). Hence proved.

**Subcase** \( v = \lambda X[p'] \): Similar to above case.

**Subcase** \( v = \eta e \): By Lemma 4, **WHERE**, and **SEALED**, we have \( \Pi; \Gamma; \rho \vdash \eta e \). \( e' : \tau \vdash \tau'' \) (thus \( \tau = \tau' \vdash \tau'' \)), where \( \Pi' \supseteq \Pi \). By **BRACKET-VALUE** and **P-LBLN**, \( \Pi' \vdash H \rightarrow \Pi' \vdash H \rightarrow \ell \). By assumption, we have \( \Pi \not\vdash \rho \vdash \tau \). Therefore we know that \( \Pi \not\vdash \ell \rightarrow \tau \) since otherwise (by Definition 6) \( \Pi \vdash \Pi \vdash \Pi' \vdash H \rightarrow \ell \rightarrow \tau'' \). Therefore, \( \mathcal{O}(\eta e, \Pi, \ell, \rightarrow) = \circ \) and \( \mathcal{O}(w, \Pi, \ell, \rightarrow) = \circ \).

**Subcase** \( v = \langle w_1, w_2 \rangle \): Straightforward application of induction hypothesis.

**Case ASSUME**: Given that \( \mathcal{O}(\langle \text{assume } \langle p > q \rangle, e_1 \rangle, 1, \Pi, \ell, \rightarrow) = \mathcal{O}(\langle \text{assume } \langle p > q \rangle, e_2 \rangle, 2, \Pi, \ell, \rightarrow) \). We have to prove that

\[ \mathcal{O}(\langle e \text{ where } \langle p > q \rangle \rangle, 1, \Pi, \ell, \rightarrow) = \mathcal{O}(\langle e \text{ where } \langle p > q \rangle \rangle, 2, \Pi, \ell, \rightarrow) \]

There are two cases: either \( \mathcal{O}(\langle \text{assume } \langle p > q \rangle, e_1 \rangle, 1, \Pi, \ell, \rightarrow) \neq \circ \) or \( \mathcal{O}(\langle \text{assume } \langle p > q \rangle, e_1 \rangle, 1, \Pi, \ell, \rightarrow) \circ \) is. Consider the former case. From the given conditions, we have \( \mathcal{O}(\langle e_1 \rangle, 1, \Pi, \ell, \rightarrow) = \mathcal{O}(\langle e_2 \rangle, 2, \Pi, \ell, \rightarrow) \). Therefore, \( \mathcal{O}(\langle e_1 \rangle, 1, \Pi, \langle p > q \rangle, \ell, \rightarrow) = \mathcal{O}(\langle e_2 \rangle, 2, \Pi, \langle p > q \rangle, \ell, \rightarrow) \). Hence proved.

Consider the latter case. That is, \( \mathcal{O}(\langle \text{assume } \langle p > q \rangle, e_1 \rangle, 1, \Pi, \ell, \rightarrow) \neq \circ \). By the definition of \( \mathcal{O} \) and \( \langle \cdot \rangle \), we know that \( \mathcal{O}(\langle e \rangle, 1, \Pi, \ell, \rightarrow) = \circ \). We will show that \( \mathcal{O}(\langle e \text{ where } \langle p > q \rangle \rangle, 1, \Pi, \ell, \rightarrow) \neq \circ \).
For more precision, we specialize our noninterference theorem on confidentiality, but a noninterference theorem for integrity holds by a similar argument. Theorem 2 states that for some principal H⁻ that flows to \( \tau' \) (Condition 1) but not \( \ell \) says \( \tau \) (Condition 2), if \( pc \) is insufficiently trusted to create new flows from \( H^- \) to \( \ell^- \) (Condition 3), then an execution of \( e \) that differs only in the value of \( \tau' \)-typed inputs, the computed values must be equal.

**Theorem 2 (Confidentiality Noninterference).** Let \( \Pi; \Gamma; x : \tau'; pc \vdash e : \ell \) says \( \tau \) for some \( H^- \) such that

1. \( \Pi \vdash H^- \land \top^+ \subseteq \tau' \)
2. \( \Pi \not\vdash H^- \subseteq \ell^- \)
3. \( \Pi \not\vdash pc \not\vdash \nabla(H^- - \ell^-) \)

Then for all \( v_z, z \in \{1, 2\} \) such that \( \Pi; \Gamma; v_z : \tau' \), if \( e[x \mapsto v_z] \xrightarrow{\ell^-} v'_z \) such that \( [v'_z]_z = v'_z \) and \( [t]_z = t_z \) for \( z = \{1, 2\} \), we will occasionally write \( v \) or \( v' \) as shorthand for \( (v_1 | v_2) \) or \( (v'_1 | v'_2) \).

Since \( \Pi; \Gamma; pc \vdash v_z : \tau' \) for \( z \in \{1, 2\} \), then we have \( \Pi; \Gamma; pc \vdash (v_1 | v_2) : \tau' \) via \texttt{Bracket-Values}. Therefore, by Lemma 15 (Variable Substitution) of \( \Pi; \Gamma; x : \tau'; pc \vdash e : \ell \) says \( \tau \), we have \( \Pi; \Gamma; pc \vdash \)
$e[x \mapsto (v_1 | v_2)] : \ell$ says $\tau$. Then by repeatedly applying Lemma 4 for each step $e[x \mapsto v] \xrightarrow{\ell} e'$, we have $\Pi; \Gamma; e : \ell \vdash e'$.

We now want to prove that $t_1 \approx^{\Pi} t_2$. First, consider $e[x \mapsto (v_1 | v_2)]$. To prove that $O(e[x \mapsto v_1], \Pi, \ell^{-}, \rightarrow) = O(e[x \mapsto v_2], \Pi, \ell^{-}, \rightarrow)$, it suffices to show that $O(v_1, \Pi, \ell^{-}, \rightarrow) = O(v_2, \Pi, \ell^{-}, \rightarrow)$. We do this by induction on the structure of $v_2$.

**Case** $v_2 = ()$: Trivial.

**Case** $v_2 = \text{inj}_i e'$: Not well typed since $\Pi \not\vdash H^{-} \subseteq \tau'$ cannot hold for types of the form $\tau_1 + \tau_2$.

**Case** $v_2 = \text{proj}_i w$: Since $\Pi \vdash H^{-} \subseteq \tau'$, by UNITM and P-LBL we know $\Pi \vdash H^{-} \subseteq \ell'$. Therefore we have $\Pi \not\vdash \ell^{-} \supseteq \ell'^{-}$, since otherwise we would have $\Pi \vdash H^{-} \subseteq \ell^{-}$, contradicting our assumption that $\Pi \not\vdash H^{-} \land \top^{-} \subseteq \ell$. By the definition of $O$, $\Pi \not\vdash \ell^{-} \supseteq \ell'^{-}$ implies that $O(\text{proj}_i w, \Pi, \ell'^{-}, \rightarrow) = o$ for $z \in \{1, 2\}$.

**Case** $v_2 = \lambda(x : \tau'') [pc']. e'$: Since $\Pi \vdash H^{-} \subseteq \tau'$, by LAM and P-FUN we know $\Pi \vdash H^{-} \subseteq pc'$. Therefore we have $\Pi \not\vdash \ell'^{-} \supseteq pc'^{-}$, since otherwise we would have $\Pi \vdash H^{-} \subseteq \ell'^{-}$, contradicting our assumption that $\Pi \not\vdash H^{-} \land \top^{-} \subseteq \ell$. By the definition of $O$, $\Pi \not\vdash \ell'^{-} \supseteq pc'^{-}$ implies that $O(\lambda(x : \tau'') [pc']. e', \Pi, \ell'^{-}, \rightarrow) = o$ for $z \in \{1, 2\}$.

**Case** $v_2 = \Delta X [pc'] e'$: Similar to above case.

**Case** $v_2 = \langle v'_1, v'_2 \rangle$: Straightforward application of the induction hypothesis.

Thus $O([e[x \mapsto v]], \Pi, \ell'^{-}, \rightarrow) = O([e[x \mapsto v]], \Pi, \ell'^{-}, \rightarrow)$. Now, by Lemma 7, for any $e'$ such that $e \xrightarrow{\ell} e'$, we have $O([e'], \Pi, \ell'^{-}, \rightarrow) = O([e'], \Pi, \ell'^{-}, \rightarrow)$. Therefore $t_1 \approx^{\Pi} t_2$. □

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 discussed in Section 5.1.

Unlike some definitions of noninterference, our definition does not prohibit all downgrading. Instead, Conditions 1 and 2 define relationships between principals $H$ and $\ell$, and Condition 3 ensures (via Lemma 6) those relationships remain unchanged by delegations made within the program, and thus by Lemma 7, the trace of a program observable to an attack remains unchanged during evaluation.

### 8. Related Work

Many languages and systems for authorization or access control have combined aspects of information security and authorization (e.g., [43,23,30,39,31,9]) in dynamic settings. However, almost all are susceptible to security vulnerabilities that arise from the interaction of information flow and authorization [7]: probing attacks, delegation loopholes, poaching attacks, and authorization side channels.

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 [14] defines terms that enable or disable flows for a lexical scope, similar to assume terms, but does not restrict their usage. Rx [39] and RTI [9] 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 [33]: 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 [10] discusses probing attacks that leak confidential information to an attacker. Garg and
Pfenning [21] 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 [22], implicit flows due to side effects are controlled via indirect reader annotations on types. Zdancewic and Myers [46] and Flow Caml [36] 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 [41] 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* [25] 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 [32,34] supports dynamically computed labels through a simple dependent type system. Jif also supports dynamically changing trust relationships through operations on principal objects [16]. 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. FLAC shows how to close these channels in languages like Jif.

Dependently-typed languages are often expressive enough to encode authorization policies, information flow policies, or both. The F* [40] type system is capable of enforcing information flow and authorization policies. Typing rules like those in FLAC could probably be encoded within its type system, but so could incorrect, insecure rules. Thus, FLAC contributes a model for encodings that enforce strong information security. Aura [26] 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.

9. Conclusion

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.

Acknowledgments

We thank Mike George, Elaine Shi, and Fred Schneider for helpful discussions, our anonymous reviewers for their comments and suggestions, and Jed Liu, Matt Stillerman, and Priyanka Mondal for...
feedback on early drafts. This work was supported by grant N00014-13-1-0089 from the Office of Naval Research, by MURI grant FA9550-12-1-0400, by grant FA9550-16-1-0351 from the Air Force Office of Scientific Research, by an NDSEG fellowship, by the Intelligence Advanced Research Projects Activity (2019-19-020700006), and by grants from the National Science Foundation (CCF-0964409, CNS-1565387, CNS-1704845, CCF-1750060). This paper does not necessarily reflect the views of any of these sponsors.

References


A. New proof

A.1. Proofs for Noninterference

Before proving noninterference, we prove few supporting lemmas required for proving type preservation.

Lemma 8 (FLAC implies FLAM). Let $H(c) = \{ (p \gg q \mid \downarrow \wedge \top^\rightarrow) \mid (p \gg q) \in \Pi \}$. If $\Pi \vdash p \gg q$, then $H; c; \downarrow \wedge \top^\rightarrow; \downarrow \wedge \top^\rightarrow \vdash p \gg q$.

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

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

$$\langle p \gg q \rangle \in \Pi$$

$$\Pi; pc; \ell \vdash \nabla(p^{-}) \gg \nabla(q^{-})$$

From (3) and DEL, we have that

$$H; c; \downarrow \wedge \top^\rightarrow; \downarrow \wedge \top^\rightarrow \vdash p \gg q$$

From [ Weaken], we get that $H; c; \top \gg (q^{-}) \vdash p \gg q$. From the (4) and R-LIFT we thus have $H; c; \downarrow \wedge \top^\rightarrow; \downarrow \wedge \top^\rightarrow \vdash p \gg q$.

Case R-Static: Since $L \vdash p \gg q$, we have from FLAM R-STATIC that $H; c; \downarrow \wedge \top^\rightarrow; \downarrow \wedge \top^\rightarrow \vdash p \gg q$.

Case R-ConjR: Trivial.

Case R-DisjL: Trivial.

Case R-Trans: Trivial.

Case R-Weaken: Trivial.

Lemma 9 (FLAM implies FLAC). For a trust configuration such that, for all $n \neq c$, $H(n) = \emptyset$, and for all $\langle p, q, \ell \rangle \in H(c)$, $\ell = \downarrow \wedge \top^\rightarrow$ and $H; c; \downarrow \wedge \top^\rightarrow; \downarrow \wedge \top^\rightarrow \vdash \nabla(p \rightarrow) \gg \nabla(q \rightarrow)$, if $H; c; \downarrow \wedge \top^\rightarrow; \downarrow \wedge \top^\rightarrow \vdash p \gg q$, then $\Pi \vdash p \gg q$.

Proof. (Sketch) By induction on the FLAM derivation. Without loss of generality, we assume that the derivation of $H; c; \downarrow \wedge \top^\rightarrow; \top \wedge \top^\rightarrow \vdash p \gg q$ 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 [8]. The rule for R-TRANS is an exception, since it requires an additional premise to be satisfied, but since the query label is always $\bot \rightarrow \land \top \rightarrow$ this is trivially satisfied.

**Lemma 10** (Robust Assumption). If $\Pi \vdash pc \geq \nabla(b)$, then $\Pi \cdot \langle a \geq b \rangle \vdash pc \geq \nabla(b)$ for any $a, b \in \mathcal{L}$.

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

**Lemma 11** (Robust Protection). If $\Pi \vdash pc \subseteq \tau$, then $\Pi \cdot \langle a \geq b \rangle \vdash pc \subseteq \tau$ for any $a, b \in \mathcal{L}$.

*Proof.* By Lemma 10 and inspection of the rules in Figure 7.

**Lemma 12** (II Extension). If $\Pi; \Gamma; pc \vdash e : \tau$ then $\Pi; \Gamma; pc \vdash e : \tau$ for any $p, q \in \mathcal{L}$.

*Proof.* By Lemma 10, Lemma 11 and inspection of the typing rules.

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

**Lemma 13** (PC Reduction). Let $\Pi; \Gamma; pc \vdash e : \tau$. For all $pc, pc'$, such that $\Pi \vdash pc' \subseteq pc$ 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, $\Pi; \Gamma; pc \vdash e \cdot e' : \tau$. From APP, we have

\[
\Pi; \Gamma; pc \vdash e : \tau_1 \xrightarrow{pc''} \tau_2 \\
\Pi; \Gamma; pc \vdash e' : \tau_1 \\
\Pi \vdash pc \subseteq pc''
\]

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

**Case SEALED:** Straightforward from the corresponding typing judgment.
Case **BINDM**: Given \( \Pi; \Gamma; \text{bind } x = e \in e' : \tau \), by **BINDM** we have

\[
\begin{align*}
\Pi; \Gamma; \text{bind } x &= e : \ell \quad \text{says } \tau' \\
\Pi; \Gamma; x : \tau'; \text{bind } x &= e' : \tau \\
\Pi; \text{bind } x &= \ell \subseteq \tau
\end{align*}
\]

(8) (9) (10)

Since \( \Pi \vdash pc' \subseteq pc \). By the monotonicity of join with respect to \( \subseteq \), we also have \( \Pi \vdash pc' \subseteq pc \subseteq \ell \). Therefore, by the induction hypothesis applied to 8 and 9, we have

\[
\begin{align*}
\Pi; \Gamma; pc' &\quad \text{bind } x = e \in e' : \tau' \\
\Pi; \Gamma; x : \tau'; pc' &\subseteq \ell \quad \text{bind } e' : \tau
\end{align*}
\]

(11) (12)

and by **R-TRANS** we get \( \Pi \vdash pc \subseteq \ell \subseteq \tau \). Then via **BINDM** we get

\[\Pi; \Gamma; pc' \quad \text{bind } x = e \in e' : \tau\]

Case **ASSUME**: Given, \( \Pi; \Gamma; pc \vdash \text{assume } e \in e' : \tau \), by **ASSUME** we have

\[
\begin{align*}
\Pi; \Gamma; pc &\vdash e : (p \geq q) \\
\Pi &\vdash pc \geq \nabla(q) \\
\Pi &\vdash \nabla(p^-) \geq \nabla(q^-) \\
\Pi, (p \geq q); \Gamma; pc &\vdash e' : \tau
\end{align*}
\]

(13) (14) (15) (16)

Applying induction hypothesis to (13) and (16) we have \( \Pi; \Gamma; pc' \vdash e : (p \geq q) \) and \( \Pi, (p \geq q); \Gamma; pc' \vdash e' : \tau \). Since \( \Pi \vdash pc' \subseteq pc \), we have \( \Pi \vdash pc' \geq \nabla(q) \). Combining, we have all the premises for **ASSUME** and thus \( \Pi; \Gamma; pc' \vdash \text{assume } e \in e' : \tau \)

Case **WHERE**: Given, \( \Pi; \Gamma; pc \vdash v \text{ where } e : \tau \), by **WHERE** we have

\[
\begin{align*}
\Pi; \Gamma; pc &\vdash v : (p \geq q) \\
\Pi &\vdash pc \geq \nabla(q) \\
\Pi &\vdash \nabla(p^-) \geq \nabla(q^-) \\
\Pi; \Gamma; pc &\vdash e : \tau
\end{align*}
\]

(17) (18) (19) (20)

Applying induction hypothesis to (17) and (20), we have \( \Pi; \Gamma; pc' \vdash v : (p \geq q) \) and \( \Pi, (p \geq q); \Gamma; pc' \vdash e : \tau \). Then by **WHERE**, we have \( \Pi; \Gamma; pc' \vdash v \text{ where } e : \tau \).

Case **BRACKET**: The premise \( \Pi \vdash H^n \sqcup pc \subseteq pc'' \) implies \( \Pi \vdash H^n \sqcup pc' \subseteq pc'' \).

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

\[\square\]

**Lemma 14 (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.

**Lemma 15** (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 14, we have \( \Pi; \Gamma ; pc \vdash e \) for any \( pc \). Therefore, each inductive case follows by straightforward application of the induction hypothesis.

**Lemma 16** (Type Substitution). Let \( \tau' \) be well-formed in \( \Gamma, X, \Gamma' \). If \( \Pi; \Gamma, X, \Gamma' ; pc \vdash e : \tau \) then \( \Pi; \Gamma, \Gamma'[X \mapsto \tau'] ; pc \vdash e[X \mapsto \tau'] : \tau' \).

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

**Lemma 17** (Projection Preserves Types). If \( \Pi; \Gamma ; pc \vdash e : \tau \), then \( \Pi; \Gamma ; pc \vdash [e]_{i} : \tau \) for \( i = 1, 2 \).

Proof. Proof is by induction on the typing derivation of \( \Pi; \Gamma ; pc \vdash e : \tau \). The interesting case is \( e = (e_{1} , e_{2}) \). By BRACKET, we have \( \Pi; \Gamma ; pc \vdash e_{i} : \tau \) for some \( pc \) such that \( \Pi \vdash (H^{\tau} \sqcap pc) \subseteq pc \). Therefore, by Lemma 13, we have \( \Pi; \Gamma ; pc \vdash e_{i} : \tau \).

**Lemma 2** (Stuck expressions). If \( e \) gets stuck then \( [e]_{i} \) 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 \( [x]_{i} \) is stuck as well.

**Case** \( x \): No reduction rules apply to a variable. Hence \( [x]_{i} \) is stuck as well.

**Case** \( (e_{1} , e_{2}) \): 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, \( [e]_{i} \) is also not of the form \( (w \mid w') \), \( w \) where \( v \), or \( \lambda(x : \tau)[pc'] \), \( e \), so \( [e e']_{i} \) is also stuck. For the second case, applying the induction hypothesis gives us that \( [e]_{i} \), is stuck for some \( i \in \{1, 2\} \), so \( \lambda(x : \tau)[pc'] \), \( e e' \), 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 \( \Delta X[pc'] \), \( e \) Therefore, \( [e]_{i} \) is also not of the form \( (w \mid w') \), \( w \) where \( v \), or \( \lambda(x : \tau)[pc'] \), \( e \), so \( [e e']_{i} \) is also stuck.

**Case** \( \eta \ e \): Since \( \eta \ e \) is stuck, then E-UNITM is not applicable, so \( e \) does not have the form \( w \). Therefore, E-UNITM is also not applicable to \( \eta \ e \). Therefore, \( e \) must be stuck. Applying induction hypothesis, it follows that \( [e]_{i} \) is also stuck and so \( \eta \ e \), is also stuck.

**Case** \( \text{proj}_{j} \ e \): Similar to the above case.

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

**Case** \( \text{case} \ e \) of \( \text{inj}_{1} (x) \). \( e_{1} \mid \text{inj}_{2} (x) \). \( e_{2} \): 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 \( \text{inj}_{j} \ v \). It follows that \( \text{case} \ e \) of \( \text{inj}_{1} (x) \). \( e_{1} \mid \text{inj}_{2} (x) \). \( e_{2} \), is also stuck.

**Case** \( \text{bind} \ x = v \ in \ e' \): Similar to the above case.
Case assume $e$ in $e'$: Similar to the above case.

Case $e$ where $v$: Similar to the above case.

Using the above lemmas, we are now ready to prove type preservation.

**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,

\[
\Pi; \Gamma, x : \tau'; pc' \vdash e : \tau \tag{21}
\]
\[
\Pi; \Gamma; pc \vdash w : \tau' \tag{22}
\]
\[
\Pi \vdash pc' \subseteq pc \tag{23}
\]

Therefore, via PC reduction (Lemma 13) and Variable Substitution (Lemma 15), we have that $\Pi; \Gamma; pc' \vdash [x \mapsto w] : \tau$.

Case **E-TAPP:** Similar to above case, but using Lemma 16.

Case **E-CASE1:** Given $e = \text{case } (\text{inj}_1 w)$ of $\text{inj}_1(x)$. $e_1 \mid \text{inj}_2(x)$. $e_2$ and $e' = e_1[x \mapsto v]$. Also, $\Pi; \Gamma; pc \vdash \text{case } (\text{inj}_1 w)$ of $\text{inj}_1(x)$. $e_1 \mid \text{inj}_2(x)$. $e_2 : \tau$. From the premises we have, $\Pi; \Gamma; pc \vdash \text{inj}_1 v : \tau'$ and $\Pi; \Gamma, x : \tau'; pc \vdash e_1 : \tau$. From IJ, we have $\Pi; \Gamma; pc \vdash v : \tau'$.

Invoking variable substitution lemma (Lemma 15), 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 $e' = \ell \eta w$. Also, $\Pi; \Gamma; pc \vdash \eta \ell \ w : \ell$ says $\tau$. From the premises it follows that $\Pi; \Gamma; pc \vdash \eta \ell \ w : \ell$ says $\tau$.

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

\[
\Pi; \Gamma; pc \vdash w : \tau' \tag{24}
\]
\[
\Pi; \Gamma, x : \tau'; pc \sqcup \ell \vdash e' : \tau \tag{25}
\]
\[
\Pi \vdash pc \sqcup \ell \subseteq \tau \tag{26}
\]
\[
\Pi \vdash p \geq pc \tag{27}
\]

We have to prove that $\Pi; \Gamma; pc \vdash e'[x \mapsto w] : \tau$. Since we have that $\Pi \vdash p \geq pc$, applying PC reduction (Lemma 13) to the premise (25), we have $\Pi; \Gamma, x : \tau'; pc \vdash e' : \tau$.

Invoking variable substitution lemma (Lemma 15), we thus have $\Pi; \Gamma; pc \vdash e'[x \mapsto w] : \tau$.

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

\[
\Pi; \Gamma; pc \vdash \langle p \geq q \rangle : \langle p \geq q \rangle \tag{28}
\]
\[
\Pi, \langle p \geq q \rangle; \Gamma; pc \vdash e' : \tau \tag{29}
\]
\[
\Pi \vdash pc \geq \nabla(q) \tag{30}
\]
\[
\Pi \vdash \nabla(pl^\tau) \geq \nabla(q^\tau) \tag{31}
\]
We need to prove:

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

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

(32) $$\Pi; \Gamma; pc \vdash \langle p \geq q \rangle : (p \geq q)$$
(33) $$\Pi, \langle p \geq q \rangle; \Gamma; pc \vdash e : \tau$$
(34) $$\Pi \vdash pc \geq \nabla (q)$$
(35) $$\Pi \vdash \nabla (p^-) \geq \nabla (q^-)$$

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 $$pc'$$, $$\tau'$$, $$\Pi'$$, and $$\Gamma'$$ 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 SEAL 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 SEAL 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 $$v \rightarrow^* e'$$.

**Case W-App:** Given $$e = \langle w \rangle$$ where $$\langle p \geq q \rangle$$ and $$e' = \langle w e \rangle$$ where $$\langle p \geq q \rangle$$. We have to prove that

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

From APP we have:

(36) $$\Pi; \Gamma; pc \vdash w \text{ where } \langle p \geq q \rangle : \tau_1 \overset{pc'}\rightarrow \tau$$
(37) $$\Pi; \Gamma; pc \vdash e : \tau_1$$
(38) $$\Pi \vdash pc \subseteq pc'$$
(39) $$\Pi, \langle p \geq q \rangle; \Gamma; pc \vdash w : \tau_1 \overset{pc'}\rightarrow \tau$$

Rule WHERE gives us the following:

(40) $$\Pi; \Gamma; pc \vdash \langle p \geq q \rangle : (p \geq q)$$
(41) $$\Pi \vdash pc \geq \nabla (q)$$
(42) $$\Pi \vdash \nabla (p^-) \geq \nabla (q^-)$$
(43) $$\Pi, \langle p \geq q \rangle; \Gamma; pc \vdash w : \tau_1 \overset{pc'}\rightarrow \tau$$

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

$$\Pi, \langle p \geq q \rangle; \Gamma; pc \vdash w e : \tau$$
Applying Lemma 12 (extending delegation contexts for well-typed terms) to (37) and Lemma 10 (extending delegation contexts for assumptions) to (38), we have:

\[
\Pi, \langle p \gg q \rangle; \Gamma; pc \vdash e : \tau_1
\]
\[
\Pi, \langle p \gg q \rangle \vdash pc \subseteq pc'
\]

Combining with (43), we have (44) which when combined with remaining premises ((40), (41) and (42)) give us \(\Pi; \Gamma; pc \vdash (w e)\) where \(\langle p \gg q \rangle : \tau\).

**Case W-TAPP:** Given \(e = (w\ \langle p \gg q \rangle)\ \tau\) and \(\epsilon' = (w\ \tau')\) where \(\langle p \gg q \rangle\). We have to prove that

\[
\Pi; \Gamma; pc \vdash (v\ \tau')\ \tau[X \rightarrow \tau']
\]

From TAPP we have:

\[
\Pi; \Gamma; pc \vdash w\ \langle p \gg q \rangle : \forall X[pc'].\ \tau
\]

\[
\Pi \vdash pc \subseteq pc'
\]

Rule WHERE gives us the following:

\[
\Pi; \Gamma; pc \vdash \langle p \gg q \rangle : (p \gg q)
\]

\[
\Pi \vdash pc \gg \Gamma
\]

\[
\Pi \vdash \nabla(p^-) \gg \nabla(q^-)
\]

\[
\Pi, \langle p \gg q \rangle; \Gamma; pc \vdash v : \forall X[pc'].\ \tau
\]

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

\[
\Pi, \langle p \gg q \rangle; \Gamma; pc \vdash w\ \tau' : \tau[X \rightarrow \tau']
\]

Applying Lemma 10 (extending delegation context for well-typed terms) to (48), we get:

\[
\Pi, \langle p \gg q \rangle \vdash pc \subseteq pc'
\]

Combining with (52), we have (53) which when combined with remaining premises ((49), (50) and (51)) give us \(\Pi; \Gamma; pc \vdash (w\ \tau')\) where \(\langle p \gg q \rangle : \tau[X \rightarrow \tau']\).

**Case W-UNPAIR:** Given \(e = \text{proj}_{ij}(\langle w_1, w_2 \rangle)\) where \(\langle p \gg q \rangle\) and \(\epsilon' = (\text{proj}_{ij}(w_1, w_2))\) where \(\langle p \gg q \rangle\). We have to prove that

\[
\Pi; \Gamma; pc \vdash (\text{proj}_{ij}(w_1, w_2))\ \tau_i
\]

From UNPAIR, we have:

\[
\Pi; \Gamma; pc \vdash \langle w_1, w_2 \rangle\ \tau_1 \times \tau_2
\]
From (55) and WHERE, we have:

$$\Pi; \Gamma; pc \vdash \langle p \geq q \rangle: (p \geq q) \tag{56}$$

$$\Pi \vdash pc \geq \nabla(q) \tag{57}$$

$$\Pi \vdash \nabla(p^\ast) \geq \nabla(q^\ast) \tag{58}$$

$$\Pi, \langle p \geq q \rangle; \Gamma; pc \vdash \langle w_1, w_2 \rangle: \tau_1 \times \tau_2 \tag{59}$$

From (59) and UNPAIR, we have: $$\Pi, \langle p \geq q \rangle; \Gamma; pc \vdash \text{proj}_i \langle w_1, w_2 \rangle : \tau_i.$$ Combining with remaining premises ((56) to (58)) we have $$\Pi, \langle p \geq q \rangle; \Gamma; pc \vdash \text{proj}_i \langle w_1, w_2 \rangle : \tau_i.$$

**Case W-CASE:** Given $$e = \text{case} (w \text{ where } \langle p \geq q \rangle) \text{ of inj}_1(x). e_1 \mid \text{inj}_2(x). e_2$$ and $$e' = (\text{case} w \text{ of inj}_1(x). e_1 \mid \text{inj}_2(x). e_2)$$ we have to prove that

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

From CASE we have:

$$\Pi; \Gamma; pc \vdash w \text{ where } \langle p \geq q \rangle : \tau_1 + \tau_2 \tag{60}$$

$$\Pi \vdash pc \sqcup \ell \subseteq \tau \tag{61}$$

$$\Pi \vdash pc \subseteq \ell \tag{62}$$

$$\Pi, \Gamma, x : \tau_1; pc \sqcup \ell \vdash e_1 : \tau \tag{63}$$

$$\Pi, \Gamma, x : \tau_2; pc \sqcup \ell \vdash e_2 : \tau \tag{64}$$

From (60) and rule WHERE, we get the following:

$$\Pi; \Gamma; pc \vdash \langle p \geq q \rangle: (p \geq q) \tag{65}$$

$$\Pi \vdash pc \geq \nabla(q) \tag{66}$$

$$\Pi \vdash \nabla(p^\ast) \geq \nabla(q^\ast) \tag{67}$$

$$\Pi; \Gamma; pc \vdash w : \tau_1 + \tau_2 \tag{68}$$

The key premise to prove is that:

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

which follows from (68) and extending delegations in equations (61) to (64) (Lemma 12). Combining with remaining premises, we have

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

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

$$\Pi, \langle p \geq q \rangle; \Gamma; pc \vdash (\text{bind } x = w \text{ in } e) : \tau$$
From $\text{BINDM}$ we have:

\[
\Pi; \Gamma; pc \vdash \langle w \text{ where } \langle p \gg q \rangle \rangle : \ell \text{ says } \tau'
\]

\[
\Pi; \Gamma, x : \tau'; pc \sqcup \ell \vdash e : \tau
\]

\[
\Pi \vdash pc \sqcup \ell \sqsubseteq \tau
\]

(69) \hspace{1cm} (70) \hspace{1cm} (71)

From (69) and rule $\text{WHERE}$, we get the following:

\[
\Pi; \Gamma; pc \vdash \langle p \gg q \rangle : (p \gg q)
\]

\[
\Pi \vdash \overline{pc} \gg \nabla(q)
\]

\[
\Pi \vdash \nabla(p^{-}) \gg \nabla(q^{-})
\]

\[
\Pi, \langle p \gg q \rangle; \Gamma; pc \vdash w : \ell \text{ says } \tau'
\]

(72) \hspace{1cm} (73) \hspace{1cm} (74) \hspace{1cm} (75)

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

\[
\Pi, \langle p \gg q \rangle; \Gamma; pc \vdash \text{bind } x = w \text{ in } e : \tau
\]

\[
\Pi; \Gamma; pc \vdash \langle p \gg q \rangle : (p \gg q)
\]

\[
\Pi \vdash \overline{pc} \gg \nabla(q)
\]

\[
\Pi \vdash \nabla(p^{-}) \gg \nabla(q^{-})
\]

(76) \hspace{1cm} (77) \hspace{1cm} (78) \hspace{1cm} (79)

Extending the delegation context (Lemma 12) in the premises (70), (71) and from (75) we have (76).

We already have (77) from (72); (78) from (73); (79) from (74). Combining, we have $\Pi; \Gamma; pc \vdash (\text{bind } x = w \text{ in } e) \text{ where } \langle p \gg q \rangle : \tau$.

**Case $\text{W-ASSUME}$**: Given $e = \text{assume } w \text{ where } \langle a \gg b \rangle \text{ in } e$ and $e' = \text{assume } w \text{ in } e \text{ where } \langle a \gg b \rangle$.

From $\text{ASSUME}$, we have

\[
\Pi; \Gamma; pc \vdash v \text{ where } \langle a \gg b \rangle : (r \gg s)
\]

\[
\Pi, \langle r \gg s \rangle; \Gamma; pc \vdash e : \tau
\]

\[
\Pi \vdash pc \gg \nabla(s)
\]

\[
\Pi \vdash \nabla(r^{-}) \gg \nabla(s^{-})
\]

(80) \hspace{1cm} (81) \hspace{1cm} (82) \hspace{1cm} (83)

Expanding the first premise using $\text{WHERE}$, we have

\[
\Pi; \Gamma; pc \vdash \langle a \gg b \rangle : (a \gg b)
\]

\[
\Pi, \langle a \gg b \rangle; \Gamma; pc \vdash v : (r \gg s)
\]

\[
\Pi \vdash \overline{pc} \gg \nabla(b)
\]

\[
\Pi \vdash \nabla(a^{-}) \gg \nabla(b^{-})
\]

(84) \hspace{1cm} (85) \hspace{1cm} (86) \hspace{1cm} (87)
We want to show
\[ \Pi; \Gamma; pc \vdash (a \equiv b) : (a \equiv b) \]  
\[ (88) \]
\[ \Pi, (a \equiv b); \Gamma; pc \vdash \text{assume } v \text{ in } e : \tau \]
\[ (89) \]
\[ \Pi \vdash pc \equiv \nabla(b) \]  
\[ (90) \]
\[ \Pi \vdash \nabla(a^-) \equiv \nabla(b^-) \]  
\[ (91) \]

Extending delegation context (Lemma 12) in the premises (81), (82), (87) and combining with topmost premise we have the (89). Remaining premises follow from (84), (86) and (87).

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

Without loss of generality, let \( i = 1 \). Thus from the premises of B-STEP, we have \( e_1 \rightarrow e'_1 \) 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
\[ \Pi; \Gamma; \Theta; p; pc' \vdash e_1 : \tau \]  
\[ (92) \]
\[ \Pi; \Gamma; \Theta; p; pc' \vdash e_2 : \tau \]  
\[ (93) \]
\[ \Pi \vdash (H^\pi \sqcup pc) \subseteq pc' \]
\[ (94) \]
\[ \Pi \vdash H^\pi \subseteq \tau \]  
\[ (95) \]

Since (92) holds, applying induction to the premise \( e_1 \rightarrow e'_1 \), we have that \( \Pi; \Gamma; \Theta; p; pc' \vdash e'_1 : \tau \). Combining with remaining premises ((93) to (95)) we thus have that \( \Pi; \Gamma; \Theta; p; pc \vdash (e'_1 \mid e'_2) : \tau \).

**Case B-APP:** Given \( e = (w_1 \mid w_2) \rightarrow w' \) and \( e' = (w_1' [w']_1 \mid w_2 [w']_2) \). Also given that \( \Pi; \Gamma; pc \vdash (w_1 \mid w_2) \rightarrow w' : \tau_2 \) is well-typed, from APP, we have the following:
\[ \Pi; \Gamma; pc \vdash (w_1 \mid w_2) : \tau_1 \xrightarrow{pc''} \tau_2 \]  
\[ (96) \]
\[ \Pi; \Gamma; pc \vdash w' : \tau_1 \]  
\[ (97) \]
\[ \Pi \vdash pc \subseteq pc'' \]  
\[ (98) \]

Thus from BRACKET-VALUES, we have \( \Pi \vdash H^\pi \subseteq \tau_1 \xrightarrow{pc''} \tau_2 \). From P-FUN, we thus have
\[ \Pi \vdash H^\pi \subseteq \tau_2 \]  
\[ (99) \]
\[ \Pi \vdash H^\pi \subseteq pc'' \]  
\[ (100) \]

We need to prove
\[ \Pi; \Gamma; pc \vdash (w_1' [w']_1 \mid w_2 [w']_2) : \tau_2 \]
That is we need the following premises of BRACKET.

\[ \Pi; \Gamma; pc' \vdash w_1 [w'_1]_1 : \tau_2 \]  
\[ \Pi; \Gamma; pc' \vdash w_2 [w'_2]_2 : \tau_2 \]  
\[ \Pi \vdash H^\pi \sqsubseteq pc \subseteq pc' \]  
\[ \Pi \vdash H^\pi \subseteq \tau_2 \]

Let \( pc' = pc'' \). We have (103) from (98) and (100). We already have (104) from (99). To prove (101), we need the following premises:

\[ \Pi; \Gamma; pc' \vdash w_1 : \tau_1 \vdash w'' \]  
\[ \Pi; \Gamma; pc' \vdash [w'_1]_2 : \tau_2 \]  
\[ \Pi \vdash pc' \subseteq pc'' \]

The last premise (107) holds trivially (from reflexivity). Applying Lemma 14 (values can be typed under any \( pc \)) to (96) we have (105). Applying Lemma 14 (values can be typed under any \( pc \)) and Lemma (17) (projection preserves typing) to (97) we have (106). Thus from APP, we have (101). Similarly, (102) holds. Hence proved.

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

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

\[ \Pi; \Gamma; pc \vdash (w_{1i} | w_{2i}) : \tau_i \]

From UNPAIR, we have:

\[ \Pi; \Gamma; pc \vdash (\langle w_{11}, w_{12} \rangle | \langle w_{21}, w_{22} \rangle) : \tau_1 \times \tau_2 \]

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

\[ \Pi; \Gamma; pc \vdash \langle w_{11}, w_{12} \rangle : \tau_1 \times \tau_2 \]  
\[ \Pi; \Gamma; pc \vdash \langle w_{21}, w_{22} \rangle : \tau_1 \times \tau_2 \]  
\[ \Pi \vdash H^\pi \subseteq \tau_1 \times \tau_2 \]

From (109), (110) 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 (111) and P-PAIR, we have \( \Pi \vdash H^\pi \subseteq \tau_i \). Combining with other premises, from BRACKET-VALUES, we have \( \Pi; \Gamma; pc \vdash (w_{1i} | w_{2i}) : \tau_i \).
Case B-BINDM: Given \( e = \text{bind } x = (\bar{\eta}_\ell w_1 | \bar{\eta}_\ell w_2) \) in \( e \). Here \( e' = (\text{bind } x = \bar{\eta}_\ell w_1 \text{ in } | e \rangle_1 | \text{bind } x = \bar{\eta}_\ell w_2 \text{ in } | e \rangle_2) \). Also \( \Pi; \Gamma; pc \vdash \text{bind } x = (\bar{\eta}_\ell w_1 | \bar{\eta}_\ell w_2) \) in \( e : \tau \). From BINDM, we have

1. \( \Pi; \Gamma; pc \vdash (\bar{\eta}_\ell w_1 | \bar{\eta}_\ell w_2) : \ell \text{ says } \tau' \) (112)
2. \( \Pi; \Gamma, x : \tau'; pc \cup \ell \vdash e : \tau \) (113)
3. \( \Pi \vdash pc \cup \ell \subseteq \tau \) (114)

From (112) and BRACKET-VALUES, we have

1. \( \Pi; \Gamma; pc \vdash \bar{\eta}_\ell w_1 : \ell \text{ says } \tau' \) (115)
2. \( \Pi; \Gamma; pc \vdash \bar{\eta}_\ell w_2 : \ell \text{ says } \tau' \) (116)
3. \( \Pi \vdash H^\pi \subseteq \ell \) (117)

We have to prove that

\( \Pi; \Gamma; pc \vdash (\text{bind } x = \bar{\eta}_\ell w_1 \text{ in } | e \rangle_1 | \text{bind } x = \bar{\eta}_\ell w_2 \text{ in } | e \rangle_2) : \tau \)

For some \( \hat{pc} \) we need the following premises to satisfy BRACKET:

1. \( \Pi; \Gamma; \hat{pc} \vdash x = \bar{\eta}_\ell w_1 \text{ in } | e \rangle_1 : \tau \) (118)
2. \( \Pi; \Gamma; \hat{pc} \vdash x = \bar{\eta}_\ell w_2 \text{ in } | e \rangle_2 : \tau \) (119)
3. \( \Pi \vdash (H^\pi \cup pc) \subseteq \hat{pc} \) (120)
4. \( \Pi \vdash H^\pi \subseteq \tau \) (121)

A natural choice for \( \hat{pc} \) is \( pc \cup \ell \). From Lemma 14 (values can be typed under any \( pc \)), we have

\( \Pi; \Gamma; \hat{pc} \vdash \bar{\eta}_\ell w_1 : \tau' \)

Applying Lemma 17 (bracket projection preserves typing) to (113), we have

\( \Pi; \Gamma, x : \tau'; \hat{pc} \vdash | e \rangle_1 : \tau \)

From BINDM, we therefore have (118) and (119). Applying R-TRANS to (117) and (114), we have (121). Thus we have all required premises.

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