A Calculus for Flow-Limited Authorization: Technical Report

Owen Arden
Department of Computer Science
Cornell University
owen@cs.cornell.edu

Andrew C. Myers
Department of Computer Science
Cornell University
andru@cs.cornell.edu

Corrected version: August 30^{th}, 2017[*]

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 [1, 2, 3], often implemented with cryptographic mechanisms [4, 5]; role-based access control (RBAC) [6]; and trust management [7, 8, 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

[*]This version has been updated to correct errors that were present in the CSF'16 submission. In particular, the semantic rules for `where` terms have been updated, and technical problems with theorem statements and proofs have been addressed. The authors would like to thank Ethan Cecchetti, who was instrumental in discovering and addressing these issues.

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., [1], [2], [7]) 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 [10], 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).[1] Abadi develops DCC as an authorization logic, but DCC is limited to static trust relationships defined externally to DCC programs by a lattice of principals. FLAC supports dynamic authorization by building on the Flow-Limited Authorization Model (FLAM) [12], which

[1]DCC was first presented in [11]. We use the abbreviation DCC to refer to the extension to polymorphic types in [2].

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 [13] 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 [12], 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.

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 [14] 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`, `receive`, 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. Next, q applies `receive` to the committed value. Finally, when p wishes to reveal the value, p applies the `open` operation to the received value, permitting q to learn it.

A commitment scheme must have several properties in order to be secure. First, q should not be able to receive a value that hasn’t been committed by p, since this could allow q to manipulate p to `open` a value it had not committed to. Second, q should not learn any secret of p that has not been opened by p. Third, p should not be able to `open` a different value than the one received by q.

One might wonder why a programmer would bother to create high-level *implementations* of operations like `commit`, `receive`, 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 [4], as well as in more recent ones like Macaroons [5].

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. [5], 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

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

$$[\text{BOT}]\ \mathcal{L} \vDash p \succcurlyeq \bot \qquad [\text{TOP}]\ \mathcal{L} \vDash \top \succcurlyeq p \qquad [\text{REFL}]\ \mathcal{L} \vDash p \succcurlyeq p \qquad [\text{PROJ}]\ \frac{\mathcal{L} \vDash p \succcurlyeq q}{\mathcal{L} \vDash p^{\pi} \succcurlyeq q^{\pi}} \qquad [\text{PROJR}]\ \mathcal{L} \vDash p \succcurlyeq p^{\pi}$$

$$[\text{CONJL}]\ \frac{\mathcal{L} \vDash p_k \succcurlyeq p \quad k \in \{1,2\}}{\mathcal{L} \vDash p_1 \wedge p_2 \succcurlyeq p} \qquad [\text{CONJR}]\ \frac{\mathcal{L} \vDash p \succcurlyeq p_1 \quad \mathcal{L} \vDash p \succcurlyeq p_2}{\mathcal{L} \vDash p \succcurlyeq p_1 \wedge p_2} \qquad [\text{DISJL}]\ \frac{\mathcal{L} \vDash p_1 \succcurlyeq p \quad \mathcal{L} \vDash p_2 \succcurlyeq p}{\mathcal{L} \vDash p_1 \vee p_2 \succcurlyeq p} \qquad [\text{DISJR}]\ \frac{\mathcal{L} \vDash p \succcurlyeq p_k \quad k \in \{1,2\}}{\mathcal{L} \vDash p \succcurlyeq p_1 \vee p_2}$$

$$[\text{TRANS}]\ \frac{\mathcal{L} \vDash p \succcurlyeq q \qquad \mathcal{L} \vDash q \succcurlyeq r}{\mathcal{L} \vDash p \succcurlyeq r}$$

Figure 1: Static principal lattice rules, adapted from FLAM [12]. The projection π may be either confidentiality ($\rightarrow$) or integrity ($\leftarrow$).

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., [5, 15, 16]) 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 [12].

Primitive principals such as `Alice`, `Bob`, etc., are represented as elements n of a (potentially infinite) set of names $\mathcal{N}$.[2] 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 $p \wedge q$, whereas the authority of either p or q is the disjunction $p \vee q$.

Unlike principals in other models, FLAM principals also represent information flow policies. The confidentiality of principal p is represented by the principal $p^{\rightarrow}$, called p's confidentiality projection. It denotes the authority necessary to *learn* anything p can learn. The integrity of principal p is represented by $p^{\leftarrow}$, called p's integrity projection. It denotes the authority to *influence* anything p can influence. All authority may be represented as some combination of confidentiality and integrity. For instance, principal p is equivalent to the conjunction $p^{\rightarrow} \wedge p^{\leftarrow}$, and in fact any FLAM principal can be written $p^{\rightarrow} \wedge q^{\leftarrow}$ for some p and q. The closure of the set of names $\mathcal{N}$ plus $\top$ and $\bot$ under the operators[3] $\wedge, \vee, \leftarrow, \rightarrow$ forms a lattice $\mathcal{L}$ ordered by an *acts-for relation* $\succcurlyeq$, defined by the inference rules in Figure 1. We write operators $\leftarrow, \rightarrow$ with higher precedence than $\wedge, \vee$; for instance, $p \wedge q^{\leftarrow}$ is equal to $p^{\rightarrow} \wedge (p \wedge q)^{\leftarrow}$. Projections distribute over $\wedge$ and $\vee$ so, for example, $(p \wedge q)^{\leftarrow} = (p^{\leftarrow} \wedge q^{\leftarrow})$. The confidentiality and integrity authority of principals are disjoint, so the confidentiality projection of an integrity projection is $\bot$ and vice-versa: $(p^{\leftarrow})^{\rightarrow} = \bot = (p^{\rightarrow})^{\leftarrow}$.

An advantage of this model is that secure information flow can be defined in terms of authority. An

[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.

information flow policy q is at least as *restrictive* as a policy p if q has at least the confidentiality authority $p^{\rightarrow}$ and p has at least the integrity authority $q^{\leftarrow}$. This relationship between the confidentiality and integrity of p and q reflects the usual duality seen in information flow control [17]. As in [12], we use the following shorthand for relating principals by policy restrictiveness:

$$
\begin{aligned}
p \sqsubseteq q &\triangleq (p^{\leftarrow} \wedge q^{\rightarrow}) \succcurlyeq (q^{\leftarrow} \wedge p^{\rightarrow}) \\
p \sqcup q &\triangleq (p \wedge q)^{\rightarrow} \wedge (p \vee q)^{\leftarrow} \\
p \sqcap q &\triangleq (p \vee q)^{\rightarrow} \wedge (p \wedge q)^{\leftarrow}
\end{aligned}
$$

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

Finally, 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 $q^{\rightarrow} \wedge r^{\leftarrow}$, the voice of that principal is

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

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

4 Flow-Limited Authorization Calculus

FLAC uses information flow to reason about the security implications of dynamically computed authority. Like previous information-flow type systems [18], FLAC incorporates types for reasoning about information flow, but FLAC's type system goes further by using Flow-Limited Authorization [12] 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 [12].

Figure 2 defines the FLAC syntax; evaluation contexts [19] are defined in Figure 3. The core operational semantics in Figure 4 is mostly standard except for `assume` terms, discussed below.

The core FLAC type system is presented in Figure 5. FLAC typing judgments have the form $\Pi; \Gamma; pc \vdash e : s$. The *delegation context*, Π, contains a set of labeled dynamic trust relationships $\langle p \succcurlyeq q \mid \ell \rangle$ where $p \succcurlyeq q$ (read as "p acts for q") is a delegation from q to p , and ℓ is the confidentiality and integrity of that information. The *typing context*, Γ, is a map from variables to types, and *pc* is the *program counter label*, a FLAM principal representing the confidentiality and integrity of control flow. The type system makes frequent use of judgments adapted from FLAM's inference rules [12]. These rules, adapted to FLAC, are presented in Figure 6.[5]

[4] In normal form, a principal is the conjunction of a confidentiality principal and an integrity principal. See [12] for details.

[5] In addition to the derivation label, the rules in [12] also include a *query label* that represents the confidentiality and integrity of a FLAM query context. The query label is unnecessary in FLAC, and hence omitted here, because we use FLAM judgments only in the type system—these "queries" only occur at compile time and do not create information flows.

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

$$
\begin{array}{rcl}
p, \ell, \mathit{pc} & ::= & n \;\big|\; \top \;\big|\; \bot \;\big|\; p^{\rightarrow} \;\big|\; p^{\leftarrow} \;\big|\; p \wedge p \;\big|\; p \vee p \\
s & ::= & (p \succcurlyeq p) \;\big|\; \texttt{unit} \;\big|\; (s + s) \;\big|\; (s \times s) \\
& & \;\big|\; s \xrightarrow{\mathit{pc}} s \;\big|\; \ell \;\texttt{says}\; s \;\big|\; X \;\big|\; \forall X.\, s \\
v & ::= & () \;\big|\; \langle v, v \rangle \;\big|\; \langle p \succcurlyeq p \rangle \;\big|\; (\eta_\ell\, v) \\
& & \;\big|\; \texttt{inj}_i\; v \;\big|\; \lambda(x\!:\!s)[\mathit{pc}].\, e \;\big|\; \Lambda X.\, e \\
& & \;\big|\; v \;\texttt{where}\; v \\
e & ::= & x \;\big|\; v \;\big|\; e\; e \;\big|\; \langle e, e \rangle \;\big|\; (\eta_\ell\, e) \\
& & \;\big|\; e s \;\big|\; \texttt{proj}_i\; e \;\big|\; \texttt{inj}_i\; e \\
& & \;\big|\; \texttt{case}\; v \;\texttt{of}\; \texttt{inj}_1(x).\, e \mid \texttt{inj}_2(x).\, e \\
& & \;\big|\; \texttt{bind}\; x = e \;\texttt{in}\; e \;\big|\; \texttt{assume}\; e \;\texttt{in}\; e \\
& & \;\big|\; e \;\texttt{where}\; v
\end{array}
$$

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

$$
\begin{array}{rcl}
E & ::= & [\cdot] \;\big|\; E\; e \;\big|\; v\; E \;\big|\; \langle E, e \rangle \;\big|\; \langle v, E \rangle \;\big|\; \texttt{proj}_i\; E \;\big|\; \texttt{inj}_i\; E \\
& & \;\big|\; (\eta_\ell\; E) \;\big|\; \texttt{bind}\; x = E \;\texttt{in}\; e \;\big|\; \texttt{bind}\; x = v \;\texttt{in}\; E \\
& & \;\big|\; E s \;\big|\; \texttt{assume}\; E \;\texttt{in}\; e \;\big|\; E \;\texttt{where}\; v \\
& & \;\big|\; \texttt{case}\; E \;\texttt{of}\; \texttt{inj}_1(x).\, e \mid \texttt{inj}_2(x).\, e
\end{array}
$$

Figure 3: FLAC evaluation contexts

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 [20, 21]). The purpose of FLAC's *pc* label is to control a different kind of side effect: changes to the delegation context, Π.[6] 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 $\varphi \xrightarrow{p^{\leftarrow}} \psi$ and φ, then (via APP) we may derive ψ in a context with integrity $p^{\leftarrow}$, but not in contexts that don't flow to $p^{\leftarrow}$. This feature offers needed control over how principals may apply existing facts to derive new facts.

Many FLAC terms are standard, such as pairs $\langle e_1, e_2 \rangle$, projections $\texttt{proj}_i\; e$, variants $\texttt{inj}_i\; e$, polymorphic type abstraction, $\Lambda X.\, e$, and case expressions. Function abstraction, $\lambda(x : s)[\mathit{pc}].\, e$, 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 $\Pi; \mathit{pc}; \mathit{pc} \Vdash \mathit{pc} \sqsubseteq \mathit{pc}'$ 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

[6] The same *pc* label could also be used to control implicit flows through assignments if FLAC were extended to support mutable references.

$\boxed{e \longrightarrow e'}$

[E-APP] $(\lambda(x\!:\!s)[pc].\, e)\; v \longrightarrow e[x \mapsto v]$ [E-TAPP] $(\Lambda X.\, e)\; s \longrightarrow e[X \mapsto s]$ [E-UNPAIR] $\texttt{proj}_i\; \langle v_1, v_2 \rangle \longrightarrow v_i$

[E-CASE] $(\texttt{case}\; (\texttt{inj}_1\; v)\; \texttt{of}\; \texttt{inj}_1(x).\, e_1 \mid \texttt{inj}_2(x).\, e_2) \longrightarrow e_i[x \mapsto v]$ [E-BINDM] $\texttt{bind}\; x = (\eta_\ell\; v)\; \texttt{in}\; e \longrightarrow e[x \mapsto v]$

[E-ASSUME] $\texttt{assume}\; \langle p \succcurlyeq q \rangle\; \texttt{in}\; e \longrightarrow e\; \texttt{where}\; \langle p \succcurlyeq q \rangle$ [E-EVAL] $\dfrac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$

Figure 4: FLAC operational semantics

CASE ensures that both expressions have the same type and thus the same protection level. The premise $\Pi; pc \vdash pc \leq s$ ensures that this type protects the current *pc* label.[7]

Like DCC, FLAC uses monadic operators to track dependencies. The monadic unit term $(\eta_\ell\; v)$ (UNITM) says that a value v of type s is *protected at level* ℓ. This protected value has the type $\ell\; \texttt{says}\; s$, meaning that it has the confidentiality and integrity of principal ℓ. 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 $\texttt{bind}\; x = (\eta_\ell\; v)\; \texttt{in}\; (\eta_{\ell'}\; x)$ is only well-typed if ℓ' protects values with confidentiality and integrity ℓ. Since `case` expressions may use the variable x for branching, BINDM raises the *pc* label to $pc \sqcup \ell$ to conservatively reflect the control-flow dependency.

Protection levels are defined by the set of inference rules in Figure 8, adapted from [22]. 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 does (P-FUN), and polymorphic types protect information at whatever level the abstracted type does (P-TFUN). If a type s already protects information at ℓ, then $\ell'\; \texttt{says}\; s$ still does (P-LBL1). Finally, if ℓ flows to ℓ', then $\ell'\; \texttt{says}\; s$ protects information at ℓ (P-LBL2).

Most of the novelty of FLAC lies in its delegation values and `assume` terms. These terms enable expressive reasoning about authority and information flow control. A delegation value serves as evidence of trust. For instance, the term $\langle p \succcurlyeq q \rangle$, read "$p$ acts for q", is evidence that q trusts p. Delegation values have *acts-for types*; $\langle p \succcurlyeq q \rangle$ has type $(p \succcurlyeq q)$. [8] The `assume` term enables programs to use evidence securely to create new flows between protection levels. In the typing context $\varnothing; x : p^{\leftarrow}\; \texttt{says}\; s; q^{\leftarrow}$ (i.e., $\Pi = \varnothing$, $\Gamma = x\!:\!p^{\leftarrow}\; \texttt{says}\; s$, and $pc = q^{\leftarrow}$), the following expression is not well typed:

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

since $p^{\leftarrow}$ does not flow to $q^{\leftarrow}$, as required by the premise $\Pi; pc \vdash \ell \leq s$ in rule BINDM. Specifically, we cannot derive $\Pi; pc \vdash p^{\leftarrow} \leq q^{\leftarrow}\; \texttt{says}\; s$ since P-LBL2 requires the FLAM judgment $\Pi; q^{\leftarrow}; q^{\leftarrow} \Vdash p^{\rightarrow} \sqsubseteq q^{\leftarrow}$ to hold.

However, the following expression is well typed:

$$\texttt{assume}\; \langle p^{\leftarrow} \succcurlyeq q^{\leftarrow} \rangle\; \texttt{in}\; \texttt{bind}\; x' = x\; \texttt{in}\; (\eta_{q^{\leftarrow}}\; x')$$

The difference is that the `assume` term adds a trust relationship, represented by an expression with an acts-for type, to the delegation context. In this case, the expression $\langle p^{\leftarrow} \succcurlyeq q^{\leftarrow} \rangle$ adds a trust relationship that

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

[8]This correspondence with delegation values makes acts-for types a kind of singleton type [23].

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

[VAR] $\quad \Pi; \Gamma, x : s, \Gamma'; pc \vdash x : s$ $\qquad$ [UNIT] $\quad \Pi; \Gamma; pc \vdash () : \texttt{unit}$ $\qquad$ [DEL] $\quad \Pi; \Gamma; pc \vdash \langle p \succcurlyeq q \rangle : (p \succcurlyeq q)$

$$\text{[LAM]} \quad \frac{\Pi; \Gamma, x{:}s_1; pc' \vdash e : s_2}{\Pi; \Gamma; pc \vdash \lambda(x{:}s_1)[pc'].\, e : (s_1 \xrightarrow{pc'} s_2)} \qquad \text{[APP]} \quad \frac{\begin{array}{c}\Pi; \Gamma; pc \vdash e : (s_1 \xrightarrow{pc'} s_2) \\ \Pi; \Gamma; pc \vdash e' : s_1 \qquad \Pi; pc; pc \Vdash pc \sqsubseteq pc'\end{array}}{\Pi; \Gamma; pc \vdash (e\ e') : s_2}$$

$$\text{[TLAM]} \quad \frac{\Pi; \Gamma, X; pc' \vdash e : s}{\Pi; \Gamma; pc \vdash \Lambda X.\, e : \forall X.\, s} \qquad \text{[TAPP]} \quad \frac{\Pi; \Gamma; pc \vdash e : \forall X.\, s}{\Pi; \Gamma; pc \vdash (e s') : s[X \mapsto s']} \qquad s' \text{ well-formed in } \Gamma$$

$$\text{[PAIR]} \quad \frac{\Pi; \Gamma; pc \vdash e_1 : s_1 \qquad \Pi; \Gamma; pc \vdash e_2 : s_2}{\Pi; \Gamma; pc \vdash \langle e_1, e_2 \rangle : (s_1 \times s_2)} \qquad \text{[UNPAIR]} \quad \frac{\Pi; \Gamma; pc \vdash e : (s_1 \times s_2)}{\Pi; \Gamma; pc \vdash (\texttt{proj}_i\ e) : s_i}$$

$$\text{[INJ]} \quad \frac{\Pi; \Gamma; pc \vdash e : s_i}{\Pi; \Gamma; pc \vdash (\texttt{inj}_i\ e) : (s_1 + s_2)} \qquad \text{[CASE]} \quad \frac{\begin{array}{c}\Pi; \Gamma; pc \vdash e : (s_1 + s_2) \qquad \Pi; pc \vdash pc \leq s \\ \Pi; \Gamma, x : s_1; pc \vdash e_1 : s \qquad \Pi; \Gamma, x : s_2; pc \vdash e_2 : s\end{array}}{\Pi; \Gamma; pc \vdash \texttt{case}\ e\ \texttt{of}\ \texttt{inj}_1(x).\ e_1 \mid \texttt{inj}_2(x).\ e_2 : s}$$

$$\text{[UNITM]} \quad \frac{\Pi; \Gamma; pc \vdash e : s}{\Pi; \Gamma; pc \vdash (\eta_\ell\ e) : \ell\ \texttt{says}\ s} \qquad \text{[BINDM]} \quad \frac{\begin{array}{c}\Pi; \Gamma; pc \vdash e : \ell\ \texttt{says}\ s' \qquad \Pi; \Gamma, x : s'; pc \sqcup \ell \vdash e' : s \\ \Pi; pc \vdash pc \sqcup \ell \leq s\end{array}}{\Pi; \Gamma; pc \vdash \texttt{bind}\ x = e\ \texttt{in}\ e' : s}$$

$$\text{[ASSUME]} \quad \frac{\begin{array}{c}\Pi; \Gamma; pc \vdash e : (p \succcurlyeq q) \\ \Pi; pc; pc \Vdash pc \succcurlyeq \nabla(q) \qquad \Pi; pc; pc \Vdash \nabla(p^{\rightarrow}) \succcurlyeq \nabla(q^{\rightarrow}) \\ \Pi; pc \vdash pc \leq s \qquad \Pi, \langle p \succcurlyeq q \mid pc \rangle; \Gamma; pc \vdash e' : s\end{array}}{\Pi; \Gamma; pc \vdash \texttt{assume}\ e\ \texttt{in}\ e' : s}$$

$$\text{[WHERE]} \quad \frac{\begin{array}{c}\Pi; \Gamma; pc \vdash v : (p \succcurlyeq q) \qquad \Pi; pc'; pc' \Vdash pc' \sqsubseteq pc \\ \Pi; pc'; pc' \Vdash pc' \succcurlyeq \nabla(q) \qquad \Pi; pc'; pc' \Vdash \nabla(p^{\rightarrow}) \succcurlyeq \nabla(q^{\rightarrow}) \\ \Pi; pc' \vdash pc' \leq s \qquad \Pi, \langle p \succcurlyeq q \mid pc' \rangle; \Gamma; pc' \vdash e : s\end{array}}{\Pi; \Gamma; pc \vdash (e\ \texttt{where}\ v) : s}$$

Figure 5: FLAC type system.

allows $p^{\leftarrow}$ to flow to $q^{\leftarrow}$. This is secure since $pc = q^{\leftarrow}$, meaning that only principals with integrity $q^{\leftarrow}$ have influenced the computation. With $\langle p^{\leftarrow} \succcurlyeq q^{\leftarrow} \mid q^{\leftarrow} \rangle$ in the delegation context, added via the ASSUME rule, the premises of BINDM are now satisfied, so the expression type-checks.

Creating a delegation value requires no special privilege because the type system ensures only high-integrity delegations are used as evidence 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 $\langle p^{\leftarrow} \succcurlyeq q^{\leftarrow} \rangle$ in the example above.[9] The second premise, $\Pi; pc; pc \Vdash pc \succcurlyeq \nabla(q)$, requires that the *pc* has enough integrity to be trusted by q, the principal whose security is affected. For instance, to make the assumption $p \succcurlyeq q$, the evidence represented by the term e must have at least the integrity of the voice of q, written $\nabla(q)$. Since the *pc* bounds the restrictiveness of the dependencies of e, this ensures that only information with integrity $\nabla(q)$ or higher may influence the evaluation of e. The third premise,

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

$$\boxed{\Pi; pc; \ell \Vdash p \succcurlyeq q}$$

$$[\text{R-STATIC}] \quad \frac{\mathcal{L} \vDash p \succcurlyeq q}{\Pi; pc; \ell \Vdash p \succcurlyeq q} \qquad [\text{R-ASSUME}] \quad \frac{\langle p \succcurlyeq q \mid \ell \rangle \in \Pi}{\Pi; pc; \ell \Vdash p \succcurlyeq q} \qquad [\text{R-CONJR}] \quad \frac{\begin{array}{c}\Pi; pc; \ell \Vdash p \succcurlyeq p_1 \\ \Pi; pc; \ell \Vdash p \succcurlyeq p_2\end{array}}{\Pi; pc; \ell \Vdash p \succcurlyeq p_1 \wedge p_2}$$

$$[\text{R-DISJL}] \quad \frac{\begin{array}{c}\Pi; pc; \ell \Vdash p_1 \succcurlyeq p \\ \Pi; pc; \ell \Vdash p_2 \succcurlyeq p\end{array}}{\Pi; pc; \ell \Vdash p_1 \vee p_2 \succcurlyeq p} \qquad [\text{R-TRANS}] \quad \frac{\begin{array}{c}\Pi; pc; \ell \Vdash p \succcurlyeq q \qquad \Pi; pc; \ell \Vdash q \succcurlyeq r \\ \Pi; pc; \ell \Vdash pc \succcurlyeq \nabla(r^{\rightarrow})\end{array}}{\Pi; pc; \ell \Vdash p \succcurlyeq r}$$

$$[\text{R-WEAKEN}] \quad \frac{\begin{array}{c}\Pi; pc'; \ell' \Vdash p \succcurlyeq q \qquad \Pi; pc \sqcup \ell'; \ell \Vdash \ell' \sqsubseteq \ell \\ \Pi; pc \sqcup \ell'; \ell \Vdash pc \sqsubseteq pc'\end{array}}{\Pi \cup \Pi'; pc; \ell \Vdash p \succcurlyeq q}$$

Figure 6: Inference rules for robust assumption, adapted from FLAM [12].

$$\boxed{e \longrightarrow e' \ \texttt{where}\ \langle p \succcurlyeq q \rangle}$$

$$[\text{W-APP}] \quad (v\ \texttt{where}\ \langle p \succcurlyeq q \rangle)\ v' \longrightarrow (v\ v')\ \texttt{where}\ \langle p \succcurlyeq q \rangle \qquad [\text{W-TAPP}] \quad (v\ \texttt{where}\ \langle p \succcurlyeq q \rangle)\ s \longrightarrow (v\ s)\ \texttt{where}\ \langle p \succcurlyeq q \rangle$$

$$[\text{W-UNPAIR}] \quad \texttt{proj}_i\ (\langle v_1, v_2 \rangle\ \texttt{where}\ \langle p \succcurlyeq q \rangle) \longrightarrow (\texttt{proj}_i\ \langle v_1, v_2 \rangle)\ \texttt{where}\ v$$

$$[\text{W-CASE}] \quad \begin{array}{c}(\texttt{case}\ (v\ \texttt{where}\ \langle p \succcurlyeq q \rangle)\ \texttt{of}\ \texttt{inj}_1(x).\ e_1 \mid \texttt{inj}_2(x).\ e_2) \longrightarrow \\ (\texttt{case}\ v\ \texttt{of}\ \texttt{inj}_1(x).\ e_1 \mid \texttt{inj}_2(x).\ e_2)\ \texttt{where}\ \langle p \succcurlyeq q \rangle\end{array}$$

$$[\text{W-UNITM}] \quad (\eta_\ell\ v\ \texttt{where}\ \langle p \succcurlyeq q \rangle) \longrightarrow (\eta_\ell\ v)\ \texttt{where}\ \langle p \succcurlyeq q \rangle \qquad [\text{W-BINDM}] \quad \begin{array}{c}\texttt{bind}\ x = (v\ \texttt{where}\ \langle p \succcurlyeq q \rangle)\ \texttt{in}\ e \longrightarrow \\ (\texttt{bind}\ x = v\ \texttt{in}\ e)\ \texttt{where}\ \langle p \succcurlyeq q \rangle\end{array}$$

$$[\text{W-ASSUME}] \quad \begin{array}{c}\texttt{assume}\ (v\ \texttt{where}\ \langle p \succcurlyeq q \rangle)\ \texttt{in}\ e \longrightarrow \\ (\texttt{assume}\ v\ \texttt{in}\ e)\ \texttt{where}\ \langle p \succcurlyeq q \rangle\end{array}$$

Figure 7: FLAC evaluation rules for `where` terms

$\Pi; pc; pc \Vdash \nabla(p^{\rightarrow}) \succcurlyeq \nabla(q^{\rightarrow})$, ensures that principal p has sufficient integrity to be trusted to enforce q's confidentiality, $q^{\rightarrow}$. This premise means that q permits data to be relabeled from $q^{\rightarrow}$ to $p^{\rightarrow}$.[10]

Assumption terms evaluate to `where` expressions (rule E-ASSUME). To simplify the formalization, these expressions are not part of the source language but are generated by the evaluation rules. The term e `where` v records that e is evaluated in a context that includes the delegation v. The rule WHERE gives a typing rule for `where` terms; though similar to ASSUME, it requires only that there exist a sufficiently trusted label pc' such that subexpression e type-checks. In the proofs in Section 7, we choose pc' using the typing judgment of the source-level `assume` that generates the `where` term.

Figure 7 presents evaluation rules for where terms. These terms are simply a bookkeeping mechanism: these evaluation rules simply record and maintain the authorization evidence used to justify new flows of information that occur during the evaluation of a FLAC program. 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

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

$\boxed{\Pi; pc \vdash \ell \leq s}$

$$[\text{P-UNIT}] \quad \Pi; pc \vdash \ell \leq \texttt{unit} \qquad [\text{P-PAIR}] \quad \frac{\Pi; pc \vdash \ell \leq s_1 \qquad \Pi; pc \vdash \ell \leq s_2}{\Pi; pc \vdash \ell \leq (s_1 \times s_2)} \qquad [\text{P-FUN}] \quad \frac{\Pi; pc \vdash \ell \leq s_2}{\Pi; pc \vdash \ell \leq s_1 \xrightarrow{pc'} s_2}$$

$$[\text{P-TFUN}] \quad \frac{\Pi; pc \vdash \ell \leq s}{\Pi; pc \vdash \ell \leq \forall X.\, s} \qquad [\text{P-LBL1}] \quad \frac{\Pi; pc \vdash \ell \leq s}{\Pi; pc \vdash \ell \leq \ell' \texttt{ says } s} \qquad [\text{P-LBL2}] \quad \frac{\Pi; pc; pc \Vdash \ell \sqsubseteq \ell'}{\Pi; pc \vdash \ell \leq \ell' \texttt{ says } s}$$

Figure 8: Type protection levels

$$
\begin{array}{l}
\texttt{commit} : \forall X.\, p^{\rightarrow} \texttt{ says } X \xrightarrow{p^{\leftarrow}} p \texttt{ says } X \\
\texttt{commit} = \\
\quad \Lambda X.\, \lambda(x : p^{\rightarrow} \texttt{ says } X)[p^{\leftarrow}]. \\
\qquad \texttt{assume } \langle \bot^{\leftarrow} \succcurlyeq p^{\leftarrow} \rangle \texttt{ in bind } x' = x \texttt{ in } (\eta_p\ x') \\
\\
\texttt{receive} : \forall X.\, p \texttt{ says } X \xrightarrow{q^{\leftarrow}} p \wedge q^{\leftarrow} \texttt{ says } X \\
\texttt{receive} = \\
\quad \Lambda X.\, \lambda(x : p \texttt{ says } X)[q^{\leftarrow}]. \\
\qquad \texttt{assume } \langle p^{\leftarrow} \succcurlyeq q^{\leftarrow} \rangle \texttt{ in bind } x' = x \texttt{ in } (\eta_{p \wedge q^{\leftarrow}}\ x') \\
\\
\texttt{open} : \forall X.\, p \wedge q^{\leftarrow} \texttt{ says } X \xrightarrow{\nabla(p^{\rightarrow})} p^{\leftarrow} \wedge q \texttt{ says } X \\
\texttt{open} = \\
\quad \Lambda X.\, \lambda(x : p \wedge q^{\leftarrow} \texttt{ says } X)[\nabla(p^{\rightarrow})]. \\
\qquad \texttt{assume } \langle \nabla(q^{\rightarrow}) \succcurlyeq \nabla(p^{\rightarrow}) \rangle \texttt{ in} \\
\qquad\quad \texttt{assume } \langle q^{\rightarrow} \succcurlyeq p^{\rightarrow} \rangle \texttt{ in bind } x' = x \texttt{ in } (\eta_{p^{\leftarrow} \wedge q}\ x')
\end{array}
$$

Figure 9: FLAC implementations of commitment scheme operations.

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.

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 9 contains the essential operations of a one-round commitment scheme—`commit`, `receive`, and `open`—implemented in FLAC. Typically, a principal p commits to a value and sends it to q, who receives it. Later, p opens the value, revealing it to q. The `commit` operation takes a value of any type (hence $\forall X$) with confidentiality $p^{\rightarrow}$ and produces a value with confidentiality and integrity p. In other words, p *endorses* [24] the value to have integrity $p^{\leftarrow}$.

Attackers should not be able to influence whether principal p commits to a particular value. The *pc* constraint on `commit` ensures that only principal p and principals trusted with at least p's integrity, $p^{\leftarrow}$, may apply `commit` to a value.[11] Furthermore, if the programmer omitted this constraint or instead chose $\bot^{\leftarrow}$, 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 $\nabla(p^{\leftarrow}) = p^{\leftarrow}$.

Next, principal q accepts a committed value from p using the `receive` operation. The `receive` operation endorses the value with q's integrity, resulting in a value at $p \wedge q^{\leftarrow}$, the confidentiality of p and the integrity of both p and q.

As with the `commit` operation, FLAC ensures that `receive` satisfies important information security properties. Other principals, including p, should not be able to influence which values q receives—otherwise an attacker could use `receive` to subvert q's integrity, using it to endorse arbitrary values. The *pc* constraint on `receive` ensures in this case that only q may apply `receive`. Furthermore, the type of x requires received values to have the integrity of p. Errors in either of these constraints would result in a typing error, either due to ASSUME as before, or due to BINDM, which requires that p must flow to $p \wedge q^{\leftarrow}$.

Additionally, `receive` accepts committed values with confidentiality at most $p^{\rightarrow}$. This constraint ensures that q does not `receive` values from p that might depend on q's secrets: unopened commitments, for example. In cryptographic protocols, this property is usually called *non-malleability* [26], and is important for scenarios in which security depends on the independence of values. Consider a sealed-bid auction where participants submit their bids via commitment protocols. Suppose that q `commit`s a bid b, protected by label q. Then p could theoretically influence a computation that computes a value $b+1$ with label $p \wedge q^{\rightarrow}$ since that label protects information at $q^{\rightarrow}$, but only has $p^{\leftarrow}$ integrity. If q received values from p that could depend on q's secrets, then p could outbid q by 1 without ever learning the value b.

Finally, `open` reveals a committed value to q by relabeling a value from $p \wedge q^{\leftarrow}$ to $p^{\leftarrow} \wedge q$, which is readable by principal q but retains the integrity of both p and q. Since `open` accepts a value protected by the integrity of both p and q and returns a value with the same integrity, the opened value must have been previously committed by p and received by q. Since the `open` operation reveals a value with confidentiality $p^{\rightarrow}$, it should only be invoked by principals that are trusted to speak for $p^{\rightarrow}$. Otherwise, q could open p's commitments. Hence, the *pc* label of `open` is $\nabla(p^{\rightarrow})$. For $p =$ `Alice`, say, the *pc* label would be `Alice`$^{\leftarrow}$. FLAC ensures these constraints are specified correctly; otherwise, `open`'s implementation could not produce a value with label $p^{\leftarrow} \wedge q$.

The implementation requires two `assume` terms. The outer term establishes that principals speaking for $q^{\rightarrow}$ also speak for $p^{\rightarrow}$ by creating an integrity relationship between their voices. With this relationship in place, the inner term may reveal the commitment to q.[12]

In DCC, functions are not annotated with *pc* labels and may be applied in any context. So a DCC

[11] 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 [25].

[12] i.e., it satisfies the ASSUME premise $\Pi; pc; pc \Vdash \nabla(p^{\rightarrow}) \succcurlyeq \nabla(q^{\rightarrow})$.

function analogous to open might have type

$$\texttt{dcc\_open} : \forall X.\, p \wedge q^{\leftarrow} \texttt{ says } X \rightarrow p^{\leftarrow} \wedge q \texttt{ says } X$$

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

To simplify the presentation of our commitment scheme operations, we make the assumption that q only receives one value. Therefore, p can only open one value, since only one value has been given the integrity of both p and q. A more general scheme can be achieved by pairing each committed value with a public identifier that is endorsed along with the value, but remains public. If q refuses to receive more that one commitment with the same identifier[13], p will be unable to open two commitments with the same value since it cannot create a pair that has the integrity of both p and q, even if p has multiple committed values (with different identifiers) to choose from. We present the simpler one-round commitment scheme above since it captures the essential information security properties of commitment while avoiding the tedious digression of defining encodings for numeric values and numeric comparisons.

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 code should be rejected because it would permit q to open p's commitments:

$$\Lambda X.\, \lambda(x \mathbin{:} p \wedge q^{\leftarrow} \texttt{ says } X)[q^{\leftarrow}].\, \texttt{assume } \langle q \succcurlyeq p \rangle \texttt{ in open } x$$

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 $pc_p = \nabla(p)$, $pc_q = \nabla(q)$, and

$$\Gamma_{cro} = \texttt{commit}, \texttt{receive}, \texttt{open}, x \mathbin{:} p^{\rightarrow} \texttt{ says } s, y \mathbin{:} p \wedge q^{\leftarrow} \texttt{ says } s$$

Intuitively, pc_p and pc_q are execution contexts under the control of p or q, respectively. Γ_{cro} is a typing context for programs using the commitment scheme.[14] The variable x represents an uncommitted value with p's confidentiality, whereas y is a committed value. Since we are interested in properties that hold for all principals p and q, we want the properties to hold in an empty delegation context: $\Pi = \varnothing$. Below, we omit the delegation context altogether for brevity.

Using results presented in Section 7, we can prove that:

- q **cannot receive a value that hasn't been committed.** For any e and s' such that $\Gamma_{cro}; pc_q \vdash e : p \wedge q^{\leftarrow} \texttt{ says } s'$, result of e is independent of x; specifically, for any v_1 and v_2, if $e[x \mapsto v_1] \longrightarrow^* v_1'$ and $e[x \mapsto v_2] \longrightarrow^* v_2'$, then $v_1' = v_2'$.
- q **cannot learn a value that hasn't been opened.** For any e, ℓ, and s' such that $\Gamma_{cro}; pc_q \vdash e : \ell \sqcap q^{\rightarrow} \texttt{ says } s'$, then the result of e is independent of x and y.
- p **cannot open a value that hasn't been received.** For any e such that $\Gamma_{cro}; pc_p \vdash e : p^{\leftarrow} \wedge q \texttt{ says } s'$, then the result of e is independent of x.

[13] For cryptographic commitment schemes, the commitment ciphertext itself could act as a public identifier, and q could rely on cryptographic assumptions that distinct values cannot (with high probability) have the same identifier instead of explicitly checking whether the identifier has been used before.

[14] For presentation purposes, we have omitted the types of commit, receive, and open in Γ_{cro}. Their types are as defined previously.

For the first two 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 $p \wedge q^{\leftarrow}$. Since such programs produce values with the integrity of p but are invoked by q, we want to ensure that no program exists that enables q to obtain a value with p's integrity that depends on x, which is a value without p's integrity. The second property concerns programs that produces values at $\ell \sqcap q^{\rightarrow}$ for any ℓ; these are values readable by q. Therefore, we want to ensure that no program exists that enables q to produce such a value that depends on x or y, which are not readable by q.

The final property considers programs that p might invoke to produce values at $p^{\leftarrow} \wedge q$, thus we consider FLAC programs that type-check in the $\Gamma_{cro}; pc_p$ context. Here, we want to ensure that no program invoked by p can produce a value at $p^{\leftarrow} \wedge q$ that depends on x, an unreceived value. Complete proofs of these properties are found in Appendix B.

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

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

which we abbreviate as $k^{\rightarrow} \overset{pc}{\Rightarrow} k^{\leftarrow}$. These terms act as bearer credentials for a principal k since they may be used as a proxy for k's confidentiality and integrity authority. Recall that $k^{\leftarrow} = k^{\leftarrow} \wedge \bot^{\rightarrow}$ and $k^{\rightarrow} = k^{\rightarrow} \wedge \bot^{\leftarrow}$. Then secrets protected by $k^{\rightarrow}$ can be declassified to $\bot^{\rightarrow}$, and untrusted data protected by $\bot^{\leftarrow}$ can be endorsed to $k^{\leftarrow}$. Thus this term wields the full authority of k, and if $pc = \bot^{\leftarrow}$, the credential may be used in any context—any "bearer" may use it. From such credentials, more restricted credentials can be derived. For example, the credential $k^{\rightarrow} \overset{pc}{\Rightarrow} \bot^{\rightarrow}$ 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 $k^{\rightarrow} \overset{pc}{\Rightarrow} k^{\leftarrow}$ 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 $k^{\leftarrow}$ integrity and no confidentiality. Suppose $\mathcal{L}$ is a security lattice used to type-check DCC programs with suitable encodings for k's confidentiality and integrity. If a DCC term has a type analogous to $k^{\rightarrow} \Rightarrow k^{\leftarrow}$, then $\mathcal{L}$ must have the property $k^{\rightarrow} \sqsubseteq \bot$ and $\bot \sqsubseteq k^{\leftarrow}$. This means that k has no confidentiality and no integrity. That FLAC terms may have this type for any principal k makes it straightforward to implement bearer credentials and demonstrates a useful application of FLAC's extra expressiveness.

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

$$\lambda(x : k^{\rightarrow} \overset{pc}{\Rightarrow} k^{\leftarrow})[pc].\, \Lambda X.\, \lambda(y : \phi)[pc].\, x X$$

which gives us a term with type

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

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

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

which abbreviates to

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

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 $\texttt{Alice}^{\rightarrow}$ $\texttt{says}$ $(k^{\rightarrow} \xRightarrow{\phi;pc} k^{\leftarrow})$. This confidentiality policy ensures the credential cannot accidentally be leaked to an attacker. A further step might be to constrain uses of this credential so that only Alice may invoke it to relabel information. If we require $pc = \texttt{Alice}^{\leftarrow}$, this credential may only be used in contexts trusted by Alice: $\texttt{Alice}^{\rightarrow}$ $\texttt{says}$ $(k^{\rightarrow} \xRightarrow{\phi;\texttt{Alice}^{\leftarrow}} k^{\leftarrow})$.

A subtle point about the way in which we construct caveats is that the caveats are polymorphic with respect to X, the same type variable the credential ranges over. This means that each caveat may constrain what types X may be instantiated with. For instance, suppose isEduc is a predicate for educational films; it holds (has a proof term with type isEduc X) for types like Bio and Doc, but not RomCom. Adding 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 isEduc RomCom could be applied, the bearer could only satisfy isEduc by instantiating X with Bio or Doc. Once X is instantiated with Bio or Doc, the credential cannot be used on a 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, $x_1 : (k_1 \texttt{ says ph})$ is a variable that stores a photo (type ph) protected by k_1. For each such variable x_1, Alice has a credential $k_1^{\rightarrow} \xRightarrow{\bot^{\leftarrow}} k_1^{\leftarrow}$, and can give access to users by providing this credential or deriving a more restricted one. To access x_1, Bob does not need the full authority of Alice or k_1—a more restricted credential suffices:

$$\begin{aligned}&\lambda(c : k_1 \xRightarrow{\texttt{Bob}^{\leftarrow}} \texttt{Bob}^{\rightarrow} \wedge k_1^{\leftarrow} \ \texttt{ph})[\texttt{Bob}^{\leftarrow}].\\ &\quad \texttt{bind}\ x_1' = c\ x_1\ \texttt{in}\ (\eta_{\texttt{Bob}^{\rightarrow} \wedge k_1^{\leftarrow}}\ x_1')\end{aligned}$$

Here, c is a credential $k_1 \xRightarrow{\texttt{Bob}^{\leftarrow}} \texttt{Bob}^{\rightarrow} \wedge k_1^{\leftarrow}$ whose polymorphic type has been instantiated with the photo type ph. This credential accepts a photo protected at k_1 and returns a photo protected at $\texttt{Bob}^{\rightarrow} \wedge k_1^{\leftarrow}$, which Bob is permitted to access.

The advantage of bearer credentials is that access to x_1 can be provided to principals other than k_1 in a decentralized way, without changing the policy on x_1. For instance, suppose Alice wants to issue a credential to Bob to access resources protected by k_1. Alice has a credential with type $k_1^{\rightarrow} \xRightarrow{\bot^{\leftarrow}} k_1^{\leftarrow}$, but she wants to ensure that only Bob (or principals Bob trusts) can use it. In other words, she wants to create a credential of type $k_1 \xRightarrow{\texttt{Bob}^{\leftarrow}} k_1^{\leftarrow}$, which needs Bob's integrity to use.

Alice can create such a credential using a wrapper that derives a more constrained credential from her original one.

$$\begin{aligned}&\lambda(c : k_1^{\rightarrow} \xRightarrow{\bot^{\leftarrow}} k_1^{\leftarrow})[\texttt{Alice}^{\leftarrow}].\\ &\quad \Lambda X.\ \lambda(y : k_1 \ \texttt{says}\ X)[\texttt{Bob}^{\leftarrow}].\\ &\qquad \texttt{bind}\ y' = y\ \texttt{in}\ (c\ X)\ (\eta_{k^{\rightarrow}}\ y')\end{aligned}$$

Then Bob can use this credential to access x_1 by deriving a credential of type $k_1 \xRightarrow{\texttt{Bob}^{\leftarrow}} \texttt{Bob}^{\rightarrow} \wedge k_1^{\leftarrow}\ \texttt{ph}$

using the function

$$\begin{array}{l}\lambda(c\!:\!k_1 \xRightarrow{\mathtt{Bob}^{\leftarrow}} k_1^{\leftarrow})[\mathtt{Bob}^{\leftarrow}].\\ \quad\lambda(y\!:\!k_1 \;\mathtt{says}\; \mathtt{ph})[\mathtt{Bob}^{\leftarrow}].\\ \quad\quad\mathtt{bind}\; y' = c\, \mathtt{ph}\, y \;\mathtt{in}\; (\eta_{\mathtt{Bob}^{\rightarrow}\wedge k_1^{\leftarrow}}\; y')\end{array}$$

which can be applied to obtain a value readable by Bob.

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

$$\begin{array}{l}\lambda(c\!:\!k_1 \xRightarrow{\mathtt{Bob}^{\leftarrow}} k_1^{\leftarrow})[\mathtt{Bob}^{\leftarrow}].\\ \quad\mathtt{assume}\; \langle \mathtt{Carol}^{\leftarrow} \succcurlyeq \mathtt{Bob}^{\leftarrow}\rangle \;\mathtt{in}\\ \quad\quad\lambda(\_\!:\!\mathtt{unit})[\mathtt{Carol}^{\leftarrow}].\\ \quad\quad\quad\mathtt{bind}\; x_1' = c\, \mathtt{ph}\, x_1 \;\mathtt{in}\; (\eta_{\mathtt{Carol}^{\rightarrow}\wedge k_1^{\leftarrow}}\; x_1')\end{array}$$

creates a wrapper around a specific photo x_1. Only principals trusted by Carol may invoke the wrapper, which produces a value of type $\mathtt{Carol}^{\rightarrow} \wedge k_1^{\leftarrow}$ $\mathtt{says}\;\mathtt{ph}$, permitting Carol to access the photo.

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

$$\Gamma_{bc} = x_i\!:\!k_i \;\mathtt{says}\; s_i, c_i\!:\!\mathtt{Alice}\;\mathtt{says}\;(k_i^{\leftarrow} \xRightarrow{\bot^{\leftarrow}} k_i^{\leftarrow})$$

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

- p **cannot access resources without a credential.** For any e, ℓ, and s' such that $\Gamma_{bc}; pc_p \vdash e : \ell \sqcap p^{\rightarrow}$ $\mathtt{says}\; s'$, the value of e is independent of x_i for all i.

- p **cannot use unrelated credentials to access resources.** For any e, ℓ, and s' such that

$$\Gamma_{bc}, c_p\!:\!(k_1^{\leftarrow} \xRightarrow{\bot^{\leftarrow}} k_1^{\leftarrow}); pc_p \vdash e : \ell \sqcap p^{\rightarrow} \;\mathtt{says}\; s'$$

  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

$$\begin{aligned}\Gamma_{bc}' = x_i\!:\!k_i \;\mathtt{says}\; s_i, c_i\!:\!\mathtt{Alice}\;\mathtt{says}\;(k_j^{\leftarrow} \xRightarrow{\bot^{\leftarrow}} k_j^{\leftarrow}),\\ c_F\!:\!\mathtt{Friends}\;\mathtt{says}\;(k_1^{\leftarrow} \xRightarrow{\bot^{\leftarrow}} k_1^{\leftarrow})\end{aligned}$$

  Then if $\Gamma_{bc}'; pc_A \vdash e : \ell \sqcap p^{\rightarrow}$ $\mathtt{says}\;(k_j^{\leftarrow} \xRightarrow{\bot^{\leftarrow}} k_j^{\leftarrow})$ for some e, ℓ, and s', 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 $\ell \sqcap p^{\rightarrow}$) that depends on any variable x_i. In the second, a credential c_p with type $k_1^{\leftarrow} \xRightarrow{\bot^{\leftarrow}} k_1^{\leftarrow}$ 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. [1] and Abadi [2]) is that implications that hold for top-level propositions also hold for propositions of any principal ℓ:

$$\vdash (s_1 \rightarrow s_2) \rightarrow (\ell \texttt{ says } s_1 \rightarrow \ell \texttt{ says } s_2)$$

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

$$\vdash (s_1 \xrightarrow{pc \sqcup \ell} s_2) \xrightarrow{pc} (\ell \texttt{ says } s_1 \xrightarrow{pc} \ell \texttt{ says } s_2)$$

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

$$\begin{aligned}&\lambda(f : (s_1 \xrightarrow{pc \sqcup \ell} s_2))[pc].\\ &\quad \lambda(x : \ell \texttt{ says } s_1)[pc].\,\texttt{bind } x' = x \texttt{ in } (\eta_\ell\ f\ x')\end{aligned}$$

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 $pc \sqcup \ell$. All theorems of DCC can be obtained by encoding them as FLAC implications with $pc = \top^{\rightarrow}$, the highest bound. Since any principal ℓ flows to $\top^{\rightarrow}$, such implications may be applied in any context.

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: $p^{\rightarrow} \overset{pc}{\Rightarrow} p^{\leftarrow}$.

Other properties of `says` common to DCC and other logics (cf. [27] for examples) are similarly refined by *pc* bounds. Two examples are: $\vdash s \xrightarrow{pc} \ell \texttt{ says } s$ which has proof term: $\lambda(x : s)[pc].\,(\eta_\ell\ s)$ and

$$\vdash \ell \texttt{ says } (s_1 \xrightarrow{pc \sqcup \ell} s_2) \xrightarrow{pc} (\ell \texttt{ says } s_1 \xrightarrow{pc} \ell \texttt{ says } s_2)$$

with proof term:

$$\begin{aligned}&\lambda(f : \ell \texttt{ says } (s_1 \xrightarrow{pc \sqcup \ell} s_2))[pc].\,\texttt{bind } x' = x \texttt{ in}\\ &\quad \lambda(y : \ell \texttt{ says } s_1)[pc].\,\texttt{bind } y' = y \texttt{ in } (\eta_\ell\ x'\ y')\end{aligned}$$

As in DCC, chains of `says` are commutative in FLAC:

$$\vdash \ell_1 \texttt{ says } \ell_2 \texttt{ says } s \xrightarrow{pc} \ell_2 \texttt{ says } \ell_1 \texttt{ says } s$$

with proof term

$$\begin{array}{l}\lambda(x\!:\!\ell_1\ \texttt{says}\ \ell_2\ \texttt{says}\ s)[pc].\\ \quad \texttt{bind}\ y = x\ \texttt{in}\ \texttt{bind}\ z = y\ \texttt{in}\ (\eta_{\ell_2}\ (\eta_{\ell_1}\ z))\end{array}$$

In some logics with different interpretations of says (e.g., CCD [28]) differently ordered chains are distinct, but here we find commutativity appealing since it matches the intuition from information flow control. When principal ℓ_1 says that ℓ_2 says s, we should protect s with a policy at least as restrictive as both ℓ_1 and ℓ_2, i.e., the principal $\ell_1 \sqcup \ell_2$. Since $\sqcup$ is commutative, who said what first is irrelevant.

6.2 Dynamic Hand-off

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

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

where $p \Rightarrow q$ is shorthand for

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

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

$$\forall X.\, (p\ \texttt{says}\ X \xrightarrow{pc} q\ \texttt{says}\ X)$$

which we wrote as $p \overset{pc}{\Rightarrow} q$ 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 ℓ and pc', let $pc = \ell^{\rightarrow} \wedge \nabla(p^{\rightarrow}) \wedge q^{\leftarrow}$*

$$\begin{array}{l}(\nabla(q^{\rightarrow}) \succcurlyeq \nabla(p^{\rightarrow})) \xrightarrow{pc} (p \sqsubseteq q) \xrightarrow{pc}\\ \quad \forall X.\, (p\ \textit{says}\ X \xrightarrow{pc'} q\ \textit{says}\ X)\end{array}$$

Proof term.

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

The principal $pc = \ell^{\rightarrow} \wedge \nabla(p^{\rightarrow}) \wedge q^{\leftarrow}$ restricts delegation (hand-off) to contexts with the integrity of $\nabla(p^{\rightarrow}) \wedge q^{\leftarrow}$. The two arguments are proofs of authority with acts-for types: a proof of $\nabla(q^{\rightarrow}) \succcurlyeq \nabla(p^{\rightarrow})$ and a proof of $p \sqsubseteq q$. The *pc* ensures that the proofs have sufficient integrity to be used in assume terms since it has the integrity of both $\nabla(p^{\rightarrow})$ and $q^{\leftarrow}$. Note that low-integrity or confidential delegation values must first be bound via `bind` before the above term may be applied. Thus the *pc* would reflect the protection level of both arguments. Principals $\ell^{\rightarrow}$ and *pc*′ are unconstrained.

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 authorization. FLAC ensures that the information security of these proofs is protected—like that of all other values—even as the policies of other information are being modified. Hence, authorization proofs cannot be used as a side channel to leak information.

7 Semantic security properties of FLAC

7.1 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 $a \wedge b$. The voice of $a \wedge b$, $\nabla(a \wedge b)$, is sufficient integrity to add a delegation $a \wedge b$ to a so that $a \succcurlyeq a \wedge b$. Alternatively, having only the integrity of $\nabla(b)$ is sufficient to add a delegation $a \succcurlyeq b$, which also results in $a \succcurlyeq a \wedge b$. To precisely characterize which trust relationships can not be added by program, we need to identify this minimal integrity $\nabla(b)$ given the pair of principals a and $a \wedge b$. The following definitions are in service of this goal.

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

Definition 1 (Principal Equivalence)**.** We say that two principals p and q are *equivalent* in $\Pi; pc$, denoted $\Pi; pc; pc \Vdash p \equiv q$, if

$$\mathcal{H}; c; pc; \ell \Vdash p \succcurlyeq q \text{ and } \mathcal{H}; c; pc; \ell \Vdash q \succcurlyeq p.$$

Next, we define the *factorization* of two principals in a given context. For two principals, p and q, their factorization involves representing q as the conjunction of two principals $q_s \wedge q_d$ such that $p \succcurlyeq q_s$ in the desired context. Note that p need not act for q_d.

Definition 2 (Factorization)**.** A $(\Pi; \mathit{pc})$-*factorization* of an ordered pair of principals (p, q) is a tuple (p, q_s, q_d) such that $\Pi; \mathit{pc}; \mathit{pc} \Vdash q \equiv q_s \wedge q_d$ and $\mathcal{H}; c; \mathit{pc}; \ell \Vdash p \succcurlyeq q_s$.

A factorization is *static* if $\Pi = \varnothing$ (and thus $\mathcal{L} \vDash p \succcurlyeq q_s$).

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 3 (Minimal Factorization)**.** A $(\Pi; \mathit{pc})$-factorization (p, q_s, q_d) of (p, q) is *minimal* if for any $(\Pi; \mathit{pc})$-factorization (p, q'_s, q'_d) of (p, q),

$$\mathcal{H}; c; \mathit{pc}; \ell \Vdash q_s \succcurlyeq q'_s \text{ and } \mathcal{H}; c; \mathit{pc}; \ell \Vdash q'_d \succcurlyeq q_d$$

The minimal factorization (p, q_s, q_d) of p and q for a given Π and pc identifies the authority necessary to cause p to act for q. Because q_s is the principal with the greatest authority such that $p \succcurlyeq q_s$ and $q \equiv q_s \wedge q_d$, then speaking for q_d is sufficient authority to cause p to act for q since adding the delegation $p \succcurlyeq q_d$ would imply that $p \succcurlyeq q$. This intuition also matches with the fact that $\mathcal{H}; c; \mathit{pc}; \ell \Vdash p \succcurlyeq q_d$ if and only if $q_d = \bot$, which is the case if and only if $\mathcal{H}; c; \mathit{pc}; \ell \Vdash p \succcurlyeq q$. Observe also that minimal $(\Pi; \mathit{pc})$-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 4 (Principal Subtraction)**.** Let (p, q_s, q_d) bet the minimal $(\Pi; \mathit{pc})$-factorization of (p, q). We define $q - p$ in $\Pi; \mathit{pc}$ to be q_d. That is, $\Pi; \mathit{pc}; \mathit{pc} \Vdash q - p \equiv q_d$. Note that $q - p$ is not defined outside of a judgement context.

Lemma 1 proves that minimal factorizations exist for all contexts and principals, so principal subtract is well defined.

Lemma 1 (Minimal Factorizations Exist)**.** *For any context* $(\Pi; \mathit{pc})$ *and principals* p, q*, there exists a minimal* $(\Pi; \mathit{pc})$*-factorization of* (p, q).

Proof. Given (p, q), we first let $q_s = p \vee q$. By definition, $\mathcal{H}; c; \mathit{pc}; \ell \Vdash p \succcurlyeq p \vee q$, and for all factorizations (p, q'_s, q'_d), $\mathcal{H}; c; \mathit{pc}; \ell \Vdash p \succcurlyeq q'_s$ and $\mathcal{H}; c; \mathit{pc}; \ell \Vdash q \succcurlyeq q'_s$, so $\mathcal{H}; c; \mathit{pc}; \ell \Vdash q_s \succcurlyeq q'_s$.

Now let $D = \{r \in \mathcal{L} \mid \Pi; \mathit{pc}; \mathit{pc} \Vdash q \equiv q_s \wedge r\}$. All principals in $\mathcal{L}$ can be represented as a finite set of meets and joins of names in $\mathcal{N}$ so q and q_s are finite. Π is also finite, adding only finitely-many dynamic equivalences, so D is finite up to equivalence. Moreover, $q \in D$ trivially, so D is non-empty and thus we can define $q_d = \bigvee D$.

Now let (p, q'_s, q'_d) be any factorization of (p, q). We must show that $\mathcal{H}; c; \mathit{pc}; \ell \Vdash q'_d \succcurlyeq q_d$. First we see that $\Pi; \mathit{pc}; \mathit{pc} \Vdash q \equiv q_s \wedge q'_d$. $\mathcal{H}; c; \mathit{pc}; \ell \Vdash q_s \succcurlyeq q'_s$ gives us $\mathcal{H}; c; \mathit{pc}; \ell \Vdash q_s \wedge q'_d \succcurlyeq q$ and the definitions of q_s and q'_d as parts of a factorization of (p, q) give us the other direction. Therefore, by construction, $q'_d \in D$, so by the definition of $\vee$ and q_d, $\mathcal{H}; c; \mathit{pc}; \ell \Vdash q'_d \succcurlyeq q_d$. Thus we see that (p, q_s, q_d) is a minimal $\Pi; \mathit{pc}$-factorization of (p, q). □

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

Lemma 2 (Delegation Invariance). *Let* $\Pi;\Gamma;pc \vdash e : s$ *such that* $e \longrightarrow e'$ `where` v. *Then there exist* $r, t \in \mathcal{L}$ *and* $\Pi' = \Pi, \langle r^\pi \succcurlyeq t^\pi \mid pc\rangle$ *such that* $\Pi;\Gamma;pc \vdash v : (r^\pi \succcurlyeq t^\pi)$ *and* $\Pi';\Gamma;pc \vdash e' : s$. *Moreover, for all principals* p *and* q *if* $\mathcal{H};c;pc;\ell \not\Vdash pc \succcurlyeq \nabla(q^\pi) - \nabla(p^\pi)$*, then*

$$\Pi';pc;pc \not\Vdash p^\pi \succcurlyeq q^\pi.$$

Proof. See Appendix A. □

First, Lemma 2 says that at each step of evaluation, there exists a Π' such that e' is well typed. More importantly, this Π' has a useful invariant. If *pc* does not speak for the authority required to cause q^π to delegate to p^π, then Π and Π' must agree on the trust relationship of p^π and q^π.

7.2 Noninterference

Lemma 2 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 [20]. This technique is relatively standard, so we omit it here. The complete proof of subject reduction is in our technical report [29]; proofs for other results are found in Appendix A.

In other noninterference results based on bracketed semantics, including [20], 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 $\Pi;\Gamma,x:s;pc \vdash e : s'$. Furthermore, let H be a principal such that $\Pi;pc \vdash H \leq s$ and $\Pi;pc \nvdash H \leq s'$. 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 [20], 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 $\Pi';pc \vdash H \leq s'$ to hold in a delegation context Π' of a subterm of e. The key to proving our result relies on using Lemma 2 to constrain the assumptions that can be added to Π'. Thus noninterference in FLAC is dependent on H and its relationship to *pc* and the type s'.

For more precision, Theorem 1 describes noninterference on confidentiality and integrity separately. It states that for some principal H^π that flows to s but not ℓ `says bool`, if *pc* cannot cause $H^\leftarrow$ to speak for $\nabla(H^\rightarrow)$ for confidentiality, or $\ell^\leftarrow$ for integrity, then an execution of e that differs only in the value of s-typed inputs, the computed values must be equal.[15]

Theorem 1 (Noninterference). *Let* $\Pi;\Gamma,x:s;pc \vdash e : \ell$ `says bool`. *If there exists some* H *and* π *such that*

1. $\Pi;pc \vdash H^\pi \leq s$
2. $\mathcal{H};c;pc;\ell \not\Vdash H^\pi \sqsubseteq \ell^\pi$
3. (a) *if* $\pi = \rightarrow$ *then* $\mathcal{H};c;pc;\ell \not\Vdash pc \succcurlyeq \nabla(H^\rightarrow) - H^\leftarrow$
   (b) *if* $\pi = \leftarrow$ *then* $\mathcal{H};c;pc;\ell \not\Vdash pc \succcurlyeq (\ell - H)^\leftarrow$

[15] It is standard for noninterference proofs in languages with higher-order functions to restrict their results to non-function types (cf. [20, 11, 30]). In this paper, we prove noninterference for boolean types, encoded as `bool` = (`unit` + `unit`). With an appropriate equivalence relation on terms, this noninterference result can be lifted to more general types.

then for all v_1, v_2 *with* $\Pi; \Gamma; pc \vdash v_i : s$*, if* $e[x \mapsto v_i] \longrightarrow^* v_i'$*, then* $v_1' = v_2'$.

Proof. By Lemma 2 and subject reduction on a bracketed semantics. See Appendix A for details. □

Condition 1 identifies s as a "high" type—at least as restricted as H. Condition 2 identifies ℓ `says bool` as a "low" type, to which information labeled H should not flow. Conditions 3a and 3b identify *pc* as having integrity compared to the difference between $H^{\leftarrow}$ and the voice $H^{\rightarrow}$ or $\ell^{\leftarrow}$. Given these conditions, if e evaluates to v_1' when $x = v_1$ and v_2' when $x = v_2$, then $v_1' = v_2'$.

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. The proofs of these properties are described in Appendix B.

7.3 Robust declassification

Using our noninterference result, we obtain a more general semantic security property for FLAC programs. That property, *robust declassification* [31], 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.

Programs and contexts that meet the requirements of Theorem 1 trivially satisfy robust declassification since no information is disclosed. In higher-integrity contexts where the pc speaks for $H^{\rightarrow}$ (and thus may influence its trust relationships), FLAC programs exhibit robust declassification.

Following Myers *et al.* [32], 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 with lambda-free types:

$$[\text{HOLE}] \qquad \frac{\Pi; pc \vdash H^{\leftarrow} \leq t \qquad \Pi; pc; pc \Vdash H^{\leftarrow} \succcurlyeq \nabla(pc)}{\Pi; \Gamma; pc \vdash [\bullet] : t}$$

We write $e[\vec{\bullet}]$ 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 $\bullet_i$. An attack $\vec{a}$ is a *fair attack* [31] on a well-typed program with holes $e[\vec{\bullet}]$ if the program $e[\vec{a}]$ is also well typed. Unfair attacks give the attacker enough power to break security directly, without exploiting existing declassifications. Fair attacks represent the power of the attacker over low-integrity portions of the program.

Theorem 2 (Robust declassification)**.** *Let* $e[\vec{\bullet}]$ *be a program such that*

1. $\Pi; \Gamma, x : s, \Gamma'; pc \vdash e[\vec{\bullet}] : \ell$ `says bool`
2. $\mathcal{H}; c; pc; \ell \not\Vdash pc \succcurlyeq (\ell - h)^{\leftarrow}$.

Then for all attacks $\vec{a_1}$ *and* $\vec{a_2}$ *and all inputs* v *such that* $\Pi; \Gamma, x : s, \Gamma'; pc \vdash e[\vec{a_i}] : \ell$ `says bool` *and* $\Pi; \Gamma; pc \vdash v : s$*, if* $e[\vec{a_i}][x \mapsto v] \longrightarrow^* v_i'$*, then* $v_1' = v_2'$.

Proof. By Lemma 2 and a generalization of Theorem 1 under attacks. See Appendix A for details. □

Our formulation of robust declassification is in some sense more general than previous definitions since it permits some endorsements, albeit restricted to untrusted principals that cannot influence the trust relationships of $\ell^{\leftarrow}$, the integrity of the result. Previous definitions of robust declassification [32, 31] forbid endorsement altogether; *qualified robustness* [32] permits endorsement but offers only possibilistic security.

8 Related Work

Many languages and systems for authorization or access control have combined aspects of information security and authorization (e.g., [33, 34, 35, 8, 36, 9]) in dynamic settings. However, almost all are susceptible to security vulnerabilities that arise from the interaction of information flow and authorization [12]: probing attacks, delegation loopholes, poaching attacks, and authorization side channels.

DCC [11, 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 [37] defines terms that enable or disable flows for a lexical scope, similar to `assume` terms, but does not restrict their usage. Rx [8] 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 [32]: 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 [38] discusses *probing attacks* that leak confidential information to an attacker. Garg and Pfenning [39] 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 [40], implicit flows due to side effects are controlled via *indirect reader* annotations on types. Zdancewic and Myers [41] and Flow Caml [20] 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 [22] 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* [42] 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 [21, 43] supports dynamically computed labels through a simple dependent type system. Jif also supports dynamically changing trust relationships through operations on principal objects [44]. 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* [45] 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 [46] 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 and Matt Stillerman for feedback on early drafts. Ethan Cecchetti discovered several technical errors in our original formalization, and was instrumental in addressing them for this corrected version. This work was supported by grant N00014-13-1-0089 from the Office of Naval Research, by MURI grant FA9550-12-1-0400, and by a grant from the National Science Foundation (CCF-0964409). This paper does not necessarily reflect the views of any of these sponsors.

References

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

[2] M. Abadi, "Access control in a core calculus of dependency," in *11th ACM SIGPLAN Int'l Conf. on Functional Programming*. New York, NY, USA: ACM, 2006, pp. 263–273.

[3] F. B. Schneider, K. Walsh, and E. G. Sirer, "Nexus Authorization Logic (NAL): Design rationale and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 14, no. 1, pp. 8:1–8:28, Jun. 2011.

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

[5] A. Birgisson, J. G. Politz, Úlfar Erlingsson, A. Taly, M. Vrable, and M. Lentczner, "Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud," in *Network and Distributed System Security Symp.*, 2014.

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

[7] N. Li, J. C. Mitchell, and W. H. Winsborough, "Design of a role-based trust-management framework," in *IEEE Symp. on Security and Privacy*, 2002, pp. 114–130.

[8] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic, "Managing policy updates in security-typed languages," in *19th IEEE Computer Security Foundations Workshop (CSFW)*, Jul. 2006, pp. 202–216.

[9] S. Bandhakavi, W. Winsborough, and M. Winslett, "A trust management approach for flexible policy management in security-typed languages," in *Computer Security Foundations Symposium, 2008*, 2008, pp. 33–47.

[10] B. Martin, M. Brown, A. Paller, D. Kirby, and S. Christey, "2011 cwe/sans top 25 most dangerous software errors," *Common Weakness Enumeration*, vol. 7515, 2011.

[11] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke, "A core calculus of dependency," in *26th ACM Symp. on Principles of Programming Languages (POPL)*, Jan. 1999, pp. 147–160.

[12] O. Arden, J. Liu, and A. C. Myers, "Flow-limited authorization," in *28th IEEE Symp. on Computer Security Foundations (CSF)*, Jul. 2015, pp. 569–583.

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

[14] M. Naor, "Bit commitment using pseudorandomness," *Journal of cryptology*, vol. 4, no. 2, pp. 151–158, 1991.

[15] J. Howell and D. Kotz, "A formal semantics for SPKI," in *ESORICS 2000*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, vol. 1895, pp. 140–158.

[16] M. Y. Becker, C. Fournet, and A. D. Gordon, "SecPAL: Design and semantics of a decentralized authorization language," *Journal of Computer Security*, vol. 18, no. 4, pp. 619–665, 2010.

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

[18] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan. 2003.

[19] A. K. Wright and M. Felleisen, "A syntactic approach to type soundness," *Information and Computation*, vol. 115, no. 1, pp. 38–94, 1994.

[20] F. Pottier and V. Simonet, "Information flow inference for ML," *ACM Trans. on Programming Languages and Systems*, vol. 25, no. 1, Jan. 2003.

[21] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *26th ACM Symp. on Principles of Programming Languages (POPL)*, Jan. 1999, pp. 228–241.

[22] S. Tse and S. Zdancewic, "Translating dependency into parametricity," in *9th ACM SIGPLAN Int'l Conf. on Functional Programming*, 2004, pp. 115–125.

[23] R. A. Eisenberg and S. Weirich, "Dependently typed programming with singletons," in *2012 Haskell Symposium*. ACM, Sep. 2012, pp. 117–130.

[24] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers, "Secure program partitioning," *ACM Trans. on Computer Systems*, vol. 20, no. 3, pp. 283–328, Aug. 2002.

[25] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers, "Sharing mobile code securely with information flow control," in *IEEE Symp. on Security and Privacy*, May 2012, pp. 191–205.

[26] D. Dolev, C. Dwork, and M. Naor, "Nonmalleable cryptography," in *SIAM Journal on Computing*, 2000, pp. 542–552.

[27] M. Abadi, "Logic in access control," in *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, ser. LICS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 228–233.

[28] ——, "Variations in access control logic," in *Deontic Logic in Computer Science*, ser. Lecture Notes in Computer Science, R. van der Meyden and L. van der Torre, Eds. Springer Berlin Heidelberg, 2008, vol. 5076, pp. 96–109.

[29] O. Arden and A. C. Myers, "A calculus for flow-limited authorization: Technical report," Cornell University Computing and Information Science, Tech. Rep. 1813–42406, Feb. 2015.

[30] L. Zheng and A. C. Myers, "Dynamic security labels and static information flow control," *International Journal of Information Security*, vol. 6, no. 2–3, Mar. 2007.

[31] S. Zdancewic and A. C. Myers, "Robust declassification," in *14th IEEE Computer Security Foundations Workshop (CSFW)*, Jun. 2001, pp. 15–23.

[32] A. C. Myers, A. Sabelfeld, and S. Zdancewic, "Enforcing robust declassification and qualified robustness," *Journal of Computer Security*, vol. 14, no. 2, pp. 157–196, 2006.

[33] W. H. Winsborough and N. Li, "Safety in automated trust negotiation," in *IEEE Symp. on Security and Privacy*, May 2004, pp. 147–160.

[34] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic, "Dynamic updating of information-flow policies," in *Foundations of Computer Security Workshop*, 2005.

[35] K. Minami and D. Kotz, "Secure context-sensitive authorization," *Journal of Pervasive and Mobile Computing*, vol. 1, no. 1, pp. 123–156, March 2005.

[36] ——, "Scalability in a secure distributed proof system," in *4th International Conference on Pervasive Computing*, ser. Lecture Notes in Computer Science, vol. 3968. Dublin, Ireland: Springer-Verlag, May 2006, pp. 220–237.

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

[38] M. Y. Becker, "Information flow in trust management systems," *Journal of Computer Security*, vol. 20, no. 6, pp. 677–708, 2012.

[39] D. Garg and F. Pfenning, "Non-interference in constructive authorization logic," in *19th IEEE Computer Security Foundations Workshop (CSFW)*, 2006.

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

[41] S. Zdancewic and A. C. Myers, "Secure information flow via linear continuations," *Higher-Order and Symbolic Computation*, vol. 15, no. 2–3, pp. 209–234, Sep. 2002.

[42] J.-B. Jeannin, G. de Caso, J. Chen, Y. Gurevich, P. Naldurg, and N. Swamy, "DKAL⋆: Constructing executable specifications of authorization protocols," in *Engineering Secure Software and Systems*. Springer, 2013, pp. 139–154.

[43] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom, "Jif 3.0: Java information flow," Jul. 2006, software release, http://www.cs.cornell.edu/jif.

[44] S. Chong, K. Vikram, and A. C. Myers, "SIF: Enforcing confidentiality and integrity in web applications," in *16th USENIX Security Symp.*, Aug. 2007.

[45] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang, "Secure distributed programming with value-dependent types," in *16th ACM SIGPLAN Int'l Conf. on Functional Programming*, ser. ICFP '11. New York, NY, USA: ACM, 2011, pp. 266–278.

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

[47] O. Arden, J. Liu, and A. C. Myers, "Flow-limited authorization: Technical report," Cornell University Computing and Information Science, Tech. Rep. 1813–40138, May 2015.

A Proofs of Noninterference and Robustness

Lemma 3 (Soundness). *If* $e \longrightarrow^* e'$ *then* $\lfloor e \rfloor_1 \longrightarrow \lfloor e' \rfloor_1$ *and* $\lfloor e \rfloor_2 \longrightarrow \lfloor e' \rfloor_2$.

Proof. By inspection of the rules in Figure 4 and Figure 10. □

Lemma 4 (Completeness). *If* $\lfloor e \rfloor_1 \longrightarrow^* v_1$ *and* $\lfloor e \rfloor_2 \longrightarrow^* v_2$*, then there exists some* v *such that* $e \longrightarrow^* v$.

Proof. Assume $\lfloor e \rfloor_1 \longrightarrow^* v_1$ and $\lfloor e \rfloor_2 \longrightarrow^* v_2$. The extended set of rules in Figure 10 always move brackets out of subterms, and therefore can only be applied a finite number of times. Therefore, by Lemma 3, if e diverges, either $\lfloor e \rfloor_1$ or $\lfloor e \rfloor_2$ diverge; this contradicts our assumption.

It remains to be shown that if the evaluation of e gets stuck, either $\lfloor e \rfloor_1$ or $\lfloor e \rfloor_2$ gets stuck. This is easily proven by induction on the structure of e. Therefore, since we assumed $\lfloor e \rfloor_i \longrightarrow^* v_i$, then e must terminate. Thus, there exists some v such that $e \longrightarrow^* v$. □

Lemma 5 (Substitution). *If* $\Pi; \Gamma, x : s'; pc \vdash e : s$ *and* $\Pi; \Gamma; pc \vdash v : s'$ *then* $\Pi; \Gamma; pc \vdash e[x \mapsto v] : s$.

Proof. By induction on the typing derivation for e. □

Lemma 6 (Type substitution). *If* $\Pi; \Gamma, X; pc \vdash e : s$ *then* $\Pi; \Gamma; pc \vdash e[X \mapsto s'] : s[X \mapsto s']$.

Proof. By induction on the derivation of $\Pi; \Gamma, X; pc \vdash e : s$. □

Lemma 7 (Projection). *If* $\Pi; \Gamma; pc \vdash e : s$ *then* $\Pi; \Gamma; pc \vdash \lfloor e \rfloor_i : s$

Proof. By induction on the derivation of $\Pi; \Gamma; pc \vdash e : s$. □

Lemma 8 (Values). *If* $\Pi; \Gamma; pc \vdash v : s$*, then* $\Pi; \Gamma; pc' \vdash v : s$ *for any* pc'.

Proof. By induction on the derivation of $\Pi; \Gamma; pc \vdash e : s$. □

Syntax extensions

$$\begin{array}{rcll} v & ::= & \dots & \mid (v \mid v) \\ e & ::= & \dots & \mid (e \mid e) \end{array}$$

Typing extensions

$$[\textsc{Bracket}] \quad \frac{\Pi; \Gamma; pc' \vdash e_1 : s \qquad \Pi; \Gamma; pc' \vdash e_2 : s \\ \Pi; pc; pc \Vdash (H \sqcup pc)^{\pi} \sqsubseteq pc'^{\pi} \qquad \Pi; pc \vdash H^{\pi} \leq s}{\Pi; \Gamma; pc \vdash (e_1 \mid e_2) : s}$$

Evaluation extensions

$$[\text{B-}\textsc{Step}] \quad \frac{e_i \longrightarrow e_i' \qquad e_j = e_j' \qquad \{i, j\} = \{1, 2\}}{(e_1 \mid e_2) \longrightarrow (e_1' \mid e_2')} \qquad [\text{B-}\textsc{App}] \quad (v_1 \mid v_2)\ v \longrightarrow (v_1\ \lfloor v \rfloor_1 \mid v_2\ \lfloor v \rfloor_2)$$

$$[\text{B-TApp}] \quad (v_1 \mid v_2)\ s \longrightarrow (v_1\ s \mid v_2\ s) \qquad [\text{B-}\textsc{Unpair}] \quad \mathtt{proj}_i\ (v_1 \mid v_2) \longrightarrow (\mathtt{proj}_i\ v_1 \mid \mathtt{proj}_i\ v_2)$$

$$[\text{B-}\textsc{Case}] \quad \begin{array}{l} \mathtt{case}\ (v_1 \mid v_2)\ \mathtt{of}\ \mathtt{inj}_1(x).\ e_1 \mid \mathtt{inj}_2(x).\ e_2 \longrightarrow \\ \big(\mathtt{case}\ v_1\ \mathtt{of}\ \mathtt{inj}_1(x).\ \lfloor e_1 \rfloor_1 \mid \mathtt{inj}_2(x).\ \lfloor e_2 \rfloor_1 \\ \quad \mid \mathtt{case}\ v_2\ \mathtt{of}\ \mathtt{inj}_1(x).\ \lfloor e_1 \rfloor_2 \mid \mathtt{inj}_2(x).\ \lfloor e_2 \rfloor_2\big) \end{array} \qquad [\text{B-}\textsc{UnitM}] \quad \eta_\ell\ (v_1 \mid v_2) \longrightarrow (\eta_\ell\ v_1 \mid \eta_\ell\ v_2)$$

$$[\text{B-}\textsc{BindM}] \quad \begin{array}{c} \mathtt{bind}\ x = (v_1 \mid v_2)\ \mathtt{in}\ e \longrightarrow \\ (\mathtt{bind}\ x = v_1\ \mathtt{in}\ \lfloor e \rfloor_1 \mid \mathtt{bind}\ x = v_2\ \mathtt{in}\ \lfloor e \rfloor_2) \end{array}$$

$$[\text{B-}\textsc{Assume}] \quad \begin{array}{c} \mathtt{assume}\ (v_1 \mid v_2)\ \mathtt{in}\ e \longrightarrow \\ (\mathtt{assume}\ v_1\ \mathtt{in}\ \lfloor e \rfloor_1 \mid \mathtt{assume}\ v_2\ \mathtt{in}\ \lfloor e \rfloor_2) \end{array}$$

Figure 10: Extensions for bracketed semantics

Lemma 9 (Robust transitivity)**.** *If* $\Pi; pc; \ell \Vdash p \succcurlyeq q$ *and* $\Pi; pc; \ell \Vdash q \succcurlyeq r$*, then* $\Pi; pc; \ell \Vdash p \succcurlyeq r$*.*

Proof. This is a consequence of the FLAM's Principal Factorization Lemma [12]. See [47] for Coq proof. □

Lemma 10 (Voices)**.** *If* $\Pi; pc; \ell \Vdash p \succcurlyeq q$ *then* $\Pi; pc; \ell \Vdash \nabla(p) \succcurlyeq \nabla(q)$*.*

Proof. By induction on the derivation of $\Pi; pc; \ell \Vdash p \succcurlyeq q$. $\mathcal{L} \vDash p \succcurlyeq q$ implies $\Pi; pc; \ell \Vdash \nabla(p) \succcurlyeq \nabla(q)$ (verified in [47]), and each $\langle p \succcurlyeq q \mid \ell \rangle \in \Pi$ has $\Pi; pc; \ell \Vdash \nabla(p^{\rightarrow}) \succcurlyeq \nabla(q^{\rightarrow})$, so $\langle p \succcurlyeq q \mid \ell \rangle \in \Pi$ implies $\Pi; pc; \ell \Vdash \nabla(p) \succcurlyeq \nabla(q)$. The remaining cases are trivial. □

Lemma 11 (*pc* reduction)**.** *If* $\Pi; \Gamma; pc' \vdash e : s$ *and* $\Pi; pc; pc \Vdash pc \sqsubseteq pc'$*, then* $\Pi; \Gamma; pc \vdash \lfloor e \rfloor_i : s$*.*

Proof. By induction on the derivation of $\Pi; \Gamma; pc' \vdash e : s$ and Lemma 9. Note that BRACKET does not preserve this property, hence the projection of e is necessary. □

Theorem 3 (Subject reduction)**.** *Suppose* $\Pi; \Gamma; pc \vdash e : s$ *and* $\lfloor e \rfloor_i \longrightarrow \lfloor e' \rfloor_i$*. If* $i \in \{1, 2\}$ *then assume* $\Pi; pc; pc \Vdash H \sqsubseteq pc$*. Then* $\Pi; \Gamma; pc \vdash e' : s$*.*

Proof.

Case (E-APP). e is $(\lambda(x\!:\!s')[pc'].\ e')\ v$, so by APP we have $\Pi; \Gamma; pc \vdash v : s'$ and $\Pi; pc; pc \Vdash pc \sqsubseteq pc'$ and by LAM we have $\Pi; \Gamma, x\!:\!s'; pc' \vdash e' : s$. Then by Lemma 8 we have $\Pi; \Gamma; pc' \vdash v : s'$, and by Lemma 5 we obtain $\Pi; \Gamma; pc' \vdash e'[x \mapsto v] : s$.

Case (E-TAPP). e is $(\Lambda X.\, e)\ s'$ and s is $s[X \mapsto s']$, so by TAPP we have $\Pi; \Gamma; pc \vdash e : \forall X.\, s$. Then by TLAM, we have $\Pi; \Gamma, X; pc \vdash e : s$ and by Lemma 6 we obtain $\Pi; \Gamma; pc \vdash e[X \mapsto s] : s[X \mapsto s']$.

Case (E-CASE). e is $(\texttt{case}\ (\texttt{inj}_1\ v)\ \texttt{of}\ \texttt{inj}_1(x).\ e_1 \mid \texttt{inj}_2(x).\ e_2)$. By INJ we have $\Pi; \Gamma; pc \vdash v : s_1$, and CASE gives us $\Pi; \Gamma; pc \vdash e_1 : s$. Therefore, by Lemma 5 we have $\Pi; \Gamma; pc \vdash e_1[x \mapsto v] : s$.

Case (E-BINDM). e is $\texttt{bind}\ x = (\eta_\ell\ v)\ \texttt{in}\ e'$ so by BINDM we have $\Pi; \Gamma; pc \vdash (\eta_\ell\ v) : \ell\ \texttt{says}\ s'$ and $\Pi; \Gamma; pc \sqcup \ell \vdash e' : s$. Rule UNITM and Lemma 8 give us $\Pi; \Gamma; pc \sqcup \ell \vdash v : s'$. Therefore, by Lemma 5 we have $\Pi; \Gamma; pc \sqcup \ell \vdash e'[x \mapsto v] : s$.

Case (E-ASSUME). e is $\texttt{assume}\ v\ \texttt{in}\ e''$ and e' is $e''\ \texttt{where}\ v$, Let $\Pi' = \Pi, \langle p \succcurlyeq q \mid pc \rangle$. By ASSUME we have $\Pi; \Gamma; pc \vdash v : (p \succcurlyeq q)$ and $\Pi'; \Gamma; pc \vdash e'' : s$. Therefore, by WHERE (choosing $pc' = pc$) we have $\Pi; \Gamma; pc \vdash (e''\ \texttt{where}\ v) : s$.

Case (E-EVAL). e is $E[e]$. By induction, $\Pi; \Gamma; pc \vdash e' : s'$. Therefore, $\Pi; \Gamma; pc \vdash E[e'] : s$.

Case (W-*). We prove the case for W-APP here. e is $(v''\ \texttt{where}\ v)\ v'$ and e' is $v''\ v'\ \texttt{where}\ v$. By APP and WHERE we have

$$\Pi, \langle p \succcurlyeq q \mid pc' \rangle; \Gamma; pc' \vdash v'' : s' \xrightarrow{pc''} s$$

$$\Pi; \Gamma; pc \vdash v' : s' \qquad \Pi; pc; pc \Vdash pc \sqsubseteq pc'' \qquad \Pi; \Gamma; pc \vdash v : (p \succcurlyeq q) \qquad \Pi; pc'; pc' \Vdash pc' \sqsubseteq pc$$

$$\Pi; pc'; pc' \Vdash pc' \succcurlyeq \nabla(q) \qquad \text{and } \Pi; pc'; pc' \Vdash \nabla(p^{\rightarrow}) \succcurlyeq \nabla(q^{\rightarrow})$$

From this, we obtain $\Pi, \langle p \succcurlyeq q \mid pc' \rangle; \Gamma; pc' \vdash v''v' : s$ via APP and Lemma 9. Then we get $\Pi; \Gamma; pc \vdash v''\ v'\ \texttt{where}\ v : s$ via WHERE. The remaining cases follow similarly to the case for W-APP, but using the relevant typing rule for the underlying term (e.g., UNPAIR, or CASE, *et cetera*) instead of APP.

Case (B-STEP). e is $(e_1 \mid e_2)$. Assume without loss of generality that $e_1 \longrightarrow e_1'$ and $e_2 = e_2'$. By BRACKET, $\Pi; \Gamma; pc \vdash e_1 : s$. By induction, $\Pi; \Gamma; pc \vdash e_1' : s$, thus BRACKET gives us $\Pi; \Gamma; pc \vdash (e_1' \mid e_2') : s$.

Case (B-APP). e is $(v_1 \mid v_2)\ v$. By APP we have $\Pi; \Gamma; pc \vdash (v_1 \mid v_2) : s' \xrightarrow{pc'} s$ and $\Pi; \Gamma; pc \vdash v : s'$, and by BRACKET, we have $\Pi; pc \vdash H \leq (s' \xrightarrow{pc'} s)$. By P-FUN, we have $\Pi; pc \vdash H \leq s$. By Lemma 7, we have $\Pi; \Gamma; pc \vdash v_i : (s' \xrightarrow{pc'} s)$. Therefore, by APP and BRACKET, we have $\Pi; \Gamma; pc \vdash (v_1\ \lfloor v \rfloor_1 \mid v_2\ \lfloor v \rfloor_2) : s$.

Case (B-TAPP). e is $(v_1 \mid v_2)\ s'$. By TAPP we have $\Pi; \Gamma; pc \vdash (v_1 \mid v_2) : \forall X.\, s$, and by BRACKET, we have $\Pi; pc \vdash H \leq (\forall X.\, s)$. By P-TFUN, we have $\Pi; pc \vdash H \leq s$. By Lemma 7, we have $\Pi; \Gamma; pc \vdash v_i : (\forall X.\, s)$. Therefore, by TAPP and BRACKET, we have $\Pi; \Gamma; pc \vdash (v_1\ s' \mid v_2\ s') : s$.

Case (B-UNPAIR). e is $\texttt{proj}_i\ (v_1 \mid v_2)$ and e' is $(\texttt{proj}_i\ v_1 \mid \texttt{proj}_i\ v_2)$. By UNPAIR, $\Pi; \Gamma; pc \vdash \texttt{proj}_i\ (v_1 \mid v_2) : s$, and by BRACKET, we have $\Pi; pc \vdash H \leq s$. Then by Lemma 7, we have $\Pi; \Gamma; pc \vdash \texttt{proj}_i\ v_i : s$, so UNPAIR and BRACKET give us $\Pi; \Gamma; pc \vdash (\texttt{proj}_i\ v_1 \mid \texttt{proj}_i\ v_2) : s$.

Case (B-CASE). e is $(\texttt{case}\ (v_1 \mid v_2)\ \texttt{of}\ \texttt{inj}_1(x).\ e_1 \mid \texttt{inj}_2(x).\ e_2)$ and e' is

$$(\texttt{case}\ v_1\ \texttt{of}\ \texttt{inj}_1(x).\ \lfloor e_1 \rfloor_1 \mid \texttt{inj}_2(x).\ \lfloor e_2 \rfloor_1 \square \mid \texttt{case}\ v_2\ \texttt{of}\ \texttt{inj}_1(x).\ \lfloor e_1 \rfloor_2 \mid \texttt{inj}_2(x).\ \lfloor e_2 \rfloor_2)$$

By BRACKET, for some pc' we have $\Pi; \Gamma; pc' \vdash v_i : s_i$ and $\Pi; pc; pc \Vdash (H \sqcup pc)^{\pi} \sqsubseteq pc'^{\pi}$. By CASE and Lemma 7, we have $\Pi; pc \vdash pc' \leq s$, therefore Lemma 9 gives us $\Pi; pc \vdash H \leq s$. We also have $\Pi; \Gamma; pc \vdash \lfloor e_1 \rfloor_i : s$ and $\Pi; \Gamma; pc \vdash \lfloor e_2 \rfloor_i : s$ for $i \in \{1, 2\}$. Therefore, by CASE we have $\Pi; \Gamma; pc \vdash \lfloor e' \rfloor_i : s$, and by BRACKET, we have $\Pi; \Gamma; pc \vdash e' : s$.

Case (B-UNITM). s is ℓ $\texttt{says}$ s, e is $\eta_\ell\ (v_1 \mid v_2)$, and e' is $(\eta_\ell\ v_1 \mid \eta_\ell\ v_2)$ By UNITM, $\Pi; \Gamma; pc \vdash (v_1 \mid v_2) : s$, and by BRACKET, we have $\Pi; pc \vdash H \leq s$. Then by P-LBL1, we have $\Pi; pc \vdash H \leq \ell\ \texttt{says}\ s$. Therefore UNITM and BRACKET give us $\Pi; \Gamma; pc \vdash (\eta_\ell\ v_1 \mid \eta_\ell\ v_2) : \ell\ \texttt{says}\ s$.

Case (B-BINDM). e is $\texttt{bind}\ x = (v_1 \mid v_2)\ \texttt{in}\ e''$, and e' is

$$(\texttt{bind}\ x = v_1\ \texttt{in}\ \lfloor e'' \rfloor_1 \mid \texttt{bind}\ x = v_2\ \texttt{in}\ \lfloor e'' \rfloor_2)$$

By BINDM and BRACKET, for some pc' we have $\Pi; \Gamma; pc' \vdash v_i : s'$ and $\Pi; pc; pc \Vdash (H \sqcup pc)^\pi \sqsubseteq pc'^\pi$. Also, by BINDM and Lemma 9, we have $\Pi; pc \vdash H \leq s$. Then, using Lemma 7, we have $\Pi; \Gamma; pc \vdash \texttt{bind}\ x = v_i\ \texttt{in}\ \lfloor e'' \rfloor_i : s$, so BRACKET gives us $\Pi; \Gamma; pc \vdash (\texttt{bind}\ x = v_1\ \texttt{in}\ \lfloor e'' \rfloor_1 \mid \texttt{bind}\ x = v_2\ \texttt{in}\ \lfloor e'' \rfloor_2) : s$.

Case (B-ASSUME). e is $\texttt{assume}\ (\langle p_1 \succcurlyeq q_1 \rangle \mid \langle p_2 \succcurlyeq q_2 \rangle)\ \texttt{in}\ e''$, and e' is $(\texttt{assume}\ v_1\ \texttt{in}\ \lfloor e'' \rfloor_1 \mid \texttt{assume}\ v_2\ \texttt{in}\ \lfloor e'' \rfloor_2)$. By ASSUME and BRACKET, for some pc' we have $\Pi; \Gamma; pc' \vdash v_i : (p \succcurlyeq q)$ and $\Pi; pc; pc \Vdash (H \sqcup pc)^\pi \sqsubseteq pc'^\pi$. By ASSUME and Lemma 7, we have $\Pi; pc \vdash pc' \leq s$, therefore Lemma 9 gives us $\Pi; pc \vdash H \leq s$. We also have $\Pi; \Gamma; pc \vdash v_i : (p \succcurlyeq q)$ and $\Pi, \langle p \succcurlyeq q \mid pc \rangle; \Gamma; pc \vdash \lfloor e'' \rfloor_i : s$ for $i \in \{1, 2\}$. Therefore, by ASSUME we have $\Pi; \Gamma; pc \vdash \texttt{assume}\ v_i\ \texttt{in}\ \lfloor e'' \rfloor_i : s$, and by BRACKET, we have $\Pi; \Gamma; pc \vdash e' : s$.

□

We extend the FLAM principal factorization lemma [12] to minimal factorizations.

Lemma 12 (Delegation Factorization)**.** *If $\mathcal{H}; c; pc; \ell \Vdash p \succcurlyeq q$ and (p, q_s, q_d) is the minimal static factorization of (p, q), then $\mathcal{H}; c; pc; \ell \Vdash p \succcurlyeq q_d$ and*

$$\mathcal{H}; c; pc; \ell \Vdash pc \succcurlyeq \nabla(q_d)$$

Proof. A Coq-verified proof in [47] showed that there is some static factorization (p, q'_s, q'_d) such that $\mathcal{H}; c; pc; \ell \Vdash p \succcurlyeq q'_d$ and $\mathcal{H}; c; pc; \ell \Vdash pc \succcurlyeq \nabla(q'_d)$. By the definition of minimal factorization, $\mathcal{L} \vDash q'_d \succcurlyeq q_d$, so $\mathcal{L} \vDash \nabla(q'_d) \succcurlyeq \nabla(q_d)$, and by transitivity on static acts-for relationships, $\mathcal{H}; c; pc; \ell \Vdash p \succcurlyeq q_d$ and $\mathcal{H}; c; pc; \ell \Vdash pc \succcurlyeq \nabla(q_d)$. □

Lemma 2 (Delegation Invariance)**.** *Let $\Pi; \Gamma; pc \vdash e : s$ such that $e \longrightarrow e'$ $\texttt{where}$ v. Then there exist $r, t \in \mathcal{L}$ and $\Pi' = \Pi, \langle r^\pi \succcurlyeq t^\pi \mid pc \rangle$ such that $\Pi; \Gamma; pc \vdash v : (r^\pi \succcurlyeq t^\pi)$ and $\Pi'; \Gamma; \beta; pc \vdash e' : s$. Moreover, for all principals p and q if $\mathcal{H}; c; pc; \ell \nVdash pc \succcurlyeq \nabla(q^\pi) - \nabla(p^\pi)$, then*

$$\Pi'; pc; pc \nVdash p^\pi \succcurlyeq q^\pi.$$

Proof. Let (p, q_s, q_d) be the minimal $(\Pi; pc)$-factorization of (p, q) (so $\Pi; pc; pc \Vdash q_d \equiv q - p$). First we note that $q_d \neq \bot$ and thus $\mathcal{H}; c; pc; \ell \nVdash p \succcurlyeq q_d$. Now assume for contradiction that $\Pi'; pc; pc \Vdash p \succcurlyeq q$.

We claim that $\mathcal{H}; c; pc; \ell \Vdash t^\pi \succcurlyeq q_d$. Assume for contradiction that this is false. By transitivity $\Pi'; pc; pc \Vdash p \succcurlyeq q_d$, and since $\mathcal{H}; c; pc; \ell \nVdash p \succcurlyeq q_d$, any derivation of this must use R-ASSUME with the delegation $\langle r^\pi \succcurlyeq t^\pi \mid pc \rangle$. This means that $\Pi; pc; pc \Vdash q_d \equiv t^\pi \wedge q'_d$ for some q'_d where $\mathcal{H}; c; pc; \ell \nVdash t^\pi \succcurlyeq q'_d$. Assume without loss of generality that $\mathcal{H}; c; pc; \ell \Vdash p \succcurlyeq q'_d$. Otherwise the same argument would give us that $\Pi; pc; pc \Vdash q'_d \equiv t^\pi \wedge q''_d$, so $q_d = t^\pi \wedge q''_d$ so we can let $q'_d = q''_d$. Since all representations are finite and with each iteration we remove at least one term from q'_d, we can only do this finitely many times until eventually $\mathcal{H}; c; pc; \ell \Vdash p \succcurlyeq q'_d$ (possibly because $q'_d = \bot$). However, this means that $(p, q_s \wedge q'_d, t^\pi)$ is a valid $(\Pi; pc)$-factorization of (p, q). Since $\mathcal{H}; c; pc; \ell \nVdash t^\pi \succcurlyeq q'_d$ by assumption, it is also the case that $\mathcal{H}; c; pc; \ell \nVdash t^\pi \succcurlyeq t^\pi \wedge q'_d$, which contradicts the assumption that (p, q_s, q_d) is minimal and thus $\mathcal{H}; c; pc; \ell \Vdash t^\pi \succcurlyeq q_d$.

If $\pi = \rightarrow$, then by WHERE with $pc' = pc$, $\mathcal{H}; c; pc; \ell \Vdash pc \succcurlyeq \nabla(t^\rightarrow)$ and by Lemma 10, $\mathcal{H}; c; pc; \ell \Vdash \nabla(t^\rightarrow) \succcurlyeq \nabla(q_d^\rightarrow)$. Thus, by transitivity, $\mathcal{H}; c; pc; \ell \Vdash pc \succcurlyeq \nabla(q_d^\rightarrow)$ which contradicts our assumption. Similarly, if $\pi = \leftarrow$, by WHERE with $pc' = pc$, $\mathcal{H}; c; pc; \ell \Vdash pc \succcurlyeq t^\leftarrow$, so by transitivity, $\mathcal{H}; c; pc; \ell \Vdash pc \succcurlyeq q_d^\leftarrow$ which again contradicts our assumption. ☐

Theorem 1 (Noninterference). *Let* $\Pi; \Gamma, x : s; pc \vdash e : \ell \texttt{ says bool}$*. If there exists some* H *and* π *such that*

1. $\Pi; pc \vdash H^\pi \leq s$
2. $\mathcal{H}; c; pc; \ell \nVdash H^\pi \sqsubseteq \ell^\pi$
3. (a) *if* $\pi = \rightarrow$ *then* $\mathcal{H}; c; pc; \ell \nVdash pc \succcurlyeq \nabla(H^\rightarrow) - H^\leftarrow$
   (b) *if* $\pi = \leftarrow$ *then* $\mathcal{H}; c; pc; \ell \nVdash pc \succcurlyeq (\ell - H)^\leftarrow$

then for all v_1, v_2 *with* $\Pi; \Gamma; pc \vdash v_i : s$*, if* $e[x \mapsto v_i] \longrightarrow^* v'_i$*, then* $v'_1 = v'_2$*.*

Proof. To prove this, we employ the bracketed semantics. By Lemma 8 $\Pi; \Gamma; \top^\rightarrow \vdash v_i : s$ and thus, for any H, $\Pi; \Gamma; pc \vdash (v_1 \mid v_2) : s$.

We now examine the term $e[x \mapsto (v_1 \mid v_2)]$. By Lemma 5, $\Pi; \Gamma; pc \vdash e[x \mapsto (v_1 \mid v_2)] : \ell \texttt{ says bool}$, and by assumption $e[x \mapsto v_i] \longrightarrow^* v_i$. Thus by Lemma 4 there is some v' such that $e[x \mapsto (v_1 \mid v_2)H^\pi] \longrightarrow^* v'$, and moreover, by Theorem 3 $\Pi; \Gamma; pc \vdash v' : \ell \texttt{ says bool}$.

We will show that $v'_1 = v'_2$ by showing that v' does not contain a bracket term. First we note that since $\mathcal{H}; c; pc; \ell \nVdash H^\pi \sqsubseteq \ell^\pi$, v' cannot itself be a bracketed term. Similarly, it cannot be the case that $v' = (\eta_\ell\ w)$ where w is bracketed. The only other option is that $v' = w \texttt{ where } u$ where w contains a bracket term. We will prove this is not the case by induction on the number of $\texttt{where}_\pi$ clauses.

For the base case with zero clauses, we have already shown this. Now we assume that $v' = w \texttt{ where } u$ and w is not itself a $\texttt{where}_\pi$ clause containing a bracketed term. There are two cases to consider, depending on π.

Case ($\pi = \rightarrow$). In this case condition 2 means $\mathcal{H}; c; pc; \ell \nVdash \ell^\rightarrow \succcurlyeq H^\rightarrow$. Since $\mathcal{H}; c; pc; \ell \nVdash pc \succcurlyeq \nabla(H^\rightarrow) - H^\leftarrow$, Lemma 2 gives us that there is some Π' such that $\Pi'; \Gamma; pc \vdash u : \ell \texttt{ says bool}$ and $\Pi'; pc; pc \nVdash \ell^\rightarrow \succcurlyeq H^\rightarrow$. Thus w cannot itself be a bracket term (or $(\eta_\ell\ w')$), and by the inductive hypothesis it is not a $\texttt{where}$ clause containing a bracket, thus proving that v' contains no bracketed terms.

Case ($\pi = \leftarrow$). In this case condition 2 means $\mathcal{H}; c; pc; \ell \nVdash H^\leftarrow \succcurlyeq \ell^\leftarrow$. Since $\mathcal{H}; c; pc; \ell \nVdash pc \succcurlyeq (\ell - H)^\leftarrow$, Lemma 2 and the same argument as the previous case show that w contains no bracketed terms.

Thus we have that v' contains no bracketed terms, so $v'_1 = \lfloor v' \rfloor_1 = \lfloor v' \rfloor_2 = v'_2$. ☐

To prove FLAC enforces robust declassification, we first prove a stronger property we call noninterference under attacks. This property demonstrates that attacks by low integrity principals cannot create interfering flows of information. This result uses the following property of principal subtraction.

Lemma 13. *For all principals* p*,* q*, and* r*,*

$$\mathcal{H}; c; pc; \ell \Vdash (p - q) \wedge (q - r) \succcurlyeq p - r.$$

Proof. Let $\overline{p}_d$ such that $\Pi; pc; pc \Vdash \overline{p}_d \equiv (p - q) \wedge (q - r)$. Now consider the following expression

$$\begin{aligned}
\Pi; pc; pc \Vdash [\overline{p}_d \vee p] \wedge (p \vee r) &\equiv p \vee (\overline{p}_d \wedge r) \\
&\equiv p \vee [(p - q) \wedge (r \wedge (q - r))] \\
&\succcurlyeq p \vee [(p - q) \wedge q] \\
&\succcurlyeq p \vee p \\
&\equiv p.
\end{aligned}$$

Thus we see that for any $(\Pi; pc)$-factorization $(r, p \vee r, p_d)$ of (r, p), $\mathcal{H}; c; pc; \ell \Vdash \overline{p_d} \vee p \succcurlyeq p_d$ which means $\mathcal{H}; c; pc; \ell \Vdash \overline{p_d} \succcurlyeq p_d$. In particular, since $(p - r)$ represents the minimal such factorization, $\mathcal{H}; c; pc; \ell \Vdash (p - q) \wedge (q - r) \succcurlyeq p - r$. ☐

Lemma 14 (Noninterference under attacks)**.** *Let $e[\vec{\bullet}]$ be a program such that*

1. $\Pi; \Gamma; pc \vdash e[\vec{\bullet}] : \ell$ `says bool`
2. *For all $\langle p \succcurlyeq q \mid pc' \rangle \in \Pi$, $\mathcal{H}; c; pc; \ell \Vdash pc' \sqsubseteq pc$*
3. *$\mathcal{H}; c; pc; \ell \nVdash H^\pi \sqsubseteq \ell^\pi$*
4. (a) *if $\pi = \rightarrow$ then $\mathcal{H}; c; pc; \ell \nVdash pc \succcurlyeq \nabla(H^\rightarrow) - H^\leftarrow$*
   (b) *if $\pi = \leftarrow$ then $\mathcal{H}; c; pc; \ell \nVdash pc \succcurlyeq (\ell - H)^\leftarrow$*

Then for all attacks $\vec{a_1}$ and $\vec{a_2}$ such that $\Pi; \Gamma; pc \vdash e[\vec{a_i}] : \ell$ `says bool`*, if $e[\vec{a_i}] \longrightarrow^* v_i$, then $v_1 = v_2$.*

Proof. Without loss of generality, assume $\vec{a_i}$ contains only one term, which we will refer to as a_i. Since $\vec{a_1}$ and $\vec{a_2}$ must have the same, finite number of terms n, if this were not the case we could construct a sequence of attacks $\vec{b_0}, \ldots, \vec{b_n}$ such that $\vec{b_i}$ includes the first i terms of $\vec{a_1}$ and the rest of the terms from $\vec{a_2}$. As each term must type check independently by HOLE, each of the $\vec{b_j}$s are valid attacks, each differ from the previous by at most one term, and $\vec{a_1} = \vec{b_n}$ and $\vec{a_2} = \vec{b_0}$. Thus it suffices to show that if at most one term differs, the attacks still produce the same result.

By HOLE, there is some Π', Γ', β', pc', and type τ such that

- $\Pi' \supseteq \Pi$,
- $\Gamma' \supseteq \Gamma$,
- $\Pi'; pc'; pc' \Vdash \beta' \sqsubseteq \beta$,
- $\Pi'; pc'; pc' \Vdash pc \sqsubseteq pc'$,
- $\Pi'; \Gamma'; \beta'; pc' \vdash [a_i]_{H^\pi} : \tau$,
- $\Pi'; pc' \vdash H^\leftarrow \leq \tau$.

Define $\mathrm{grd}(\tau)$ as the most restrictive principal protected by type τ. Specifically, the principal p such that $\varnothing; pc \vdash p \leq \tau$ and for all principals p' such that $\varnothing; pc \vdash p' \leq \tau$, we have $\mathcal{L} \vDash p' \sqsubseteq p$.

Let $\hat{H} = \mathrm{grd}(\tau)^\pi \wedge H^{\pi'}$ where $\pi' \neq \pi$ is the opposite projection. We note that $\Pi; \Gamma; pc \vdash e[a_i] : \ell$ `says bool` and $\mathcal{H}; c; pc; \ell \Vdash \hat{H}^\pi \sqsubseteq \mathrm{grd}(\tau)^\pi$ trivially, and we claim that Condition 4 holds with $\hat{H}$ in place of H. We prove this in two cases.

Case ($\pi = \rightarrow$). First we claim that $\mathcal{H}; c; pc; \ell \Vdash pc \succcurlyeq \nabla(\mathrm{grd}(\tau)^{\rightarrow}) - \nabla(H^{\rightarrow})$. By Condition 2 and R-WEAKEN, know that $\mathcal{H}; c; pc; \ell \Vdash pc \succcurlyeq \nabla(\mathrm{grd}(\tau)^{\rightarrow}) - \nabla(H^{\rightarrow})$ if and only if $\Pi; pc'; pc' \Vdash pc \succcurlyeq \nabla(\mathrm{grd}(\tau)^{\rightarrow}) - \nabla(H^{\rightarrow})$. Moreover, since $\Pi'; pc'; pc' \Vdash H^{\rightarrow} \succcurlyeq \mathrm{grd}(\tau)^{\rightarrow}$, finite iterated application of Lemma 2 gives us that $\Pi; pc'; pc' \Vdash pc \succcurlyeq \nabla(\mathrm{grd}(\tau)^{\rightarrow}) - \nabla(H^{\rightarrow})$. Therefore Condition 4, Lemma 13, and the definition of $\hat{H}$ give us

$$\mathcal{H}; c; pc; \ell \nVdash pc \succcurlyeq \nabla(\hat{H}^{\rightarrow}) - \hat{H}^{\leftarrow}.$$

Case ($\pi = \leftarrow$). By the same argument as above noting that HOLE gives us $\Pi'; pc'; pc' \Vdash H^{\leftarrow} \succcurlyeq \mathrm{grd}(\tau)^{\leftarrow}$, $\mathcal{H}; c; pc; \ell \Vdash pc \succcurlyeq (H - \mathrm{grd}(\tau))^{\leftarrow}$. Thus by Lemma 13, $\mathcal{H}; c; pc; \ell \nVdash pc \succcurlyeq \ell^{\leftarrow} - \mathrm{grd}(\tau)^{\leftarrow}$.

Therefore, if we let x be a fresh variable in $e[\vec{\bullet}]_{H^\pi}$, then we can let $e' = e[x]_{H^\pi}$, and $\Pi; \Gamma, x : \tau; \beta; pc \vdash e' : \ell$ `says bool`. If we let w_i such that $a_i \longrightarrow^* w_i$, then $e'[x \mapsto w_i] \longrightarrow^* v_i$ and by Theorem 1, $v_1 = v_2$. □

Theorem 2 (Robust declassification). *Let $e[\vec{\bullet}]$ be a program such that*

1. $\Pi; \Gamma, x : s, \Gamma'; pc \vdash e[\vec{\bullet}] : \ell$ *`says bool`*
2. $\mathcal{H}; c; pc; \ell \nVdash pc \succcurlyeq (\ell - h)^{\leftarrow}$.

Then for all attacks $\vec{a_1}$ and $\vec{a_2}$ and all inputs v such that $\Pi; \Gamma, x : s, \Gamma'; pc \vdash e[\vec{a_i}] : \ell$ `says bool` and $\Pi; \Gamma; pc \vdash v : s$, if $e[\vec{a_i}][x \mapsto v] \longrightarrow^* v_i'$, then $v_1' = v_2'$.

Proof. Let $e'[\vec{\bullet}] = e[\vec{\bullet}][x \mapsto v]$, and let $\vec{a_i'} = \vec{a_i}[x \mapsto v]$. By Lemma 5, $\Pi; \Gamma; pc \vdash e'[\vec{a_i'}] : \ell$ `says bool`. Thus by Lemma 14 $v_1' = v_2'$. □

B Commitment Scheme Verification

To prove the desired properties of commitment schemes for boolean values, let $s =$ `bool` and recall:
$\Gamma_{cro} =$ `commit`, `receive`, `open`, $x : p^{\rightarrow}$ `says` $s, y : p \wedge q^{\leftarrow}$ `says` s

- q **cannot `receive` a value that hasn't been committed.** Let $H = p^{\rightarrow} \wedge q^{\leftarrow}$. For any e and $\Gamma_{cro}; pc_q \vdash e : p \wedge q^{\leftarrow}$ `says bool`, observe that $\Pi; pc_q \vdash H \leq p^{\rightarrow}$ `says bool`, $\Pi; pc_q; pc_q \nVdash H^{\rightarrow} \sqsubseteq p^{\rightarrow}$, $\Pi; pc_q; pc_q \nVdash H^{\leftarrow} \sqsubseteq (p \wedge q)^{\leftarrow}$, and $\Pi; pc_q; pc_q \nVdash pc_q \succcurlyeq \nabla(H^{\rightarrow}) \wedge (p \wedge q)^{\leftarrow}$. Therefore, by Theorem 1, if $e[x \mapsto v_1] \longrightarrow^* v_1'$ and $e[x \mapsto v_2] \longrightarrow^* v_2'$, then $v_1' \simeq v_2'$.
- q **cannot learn a value that hasn't been opened.** Let $H = p^{\rightarrow} \wedge q^{\leftarrow}$. For any e, ℓ, and $\Gamma_{cro}; pc_q \vdash e : \ell \sqcap q^{\rightarrow}$ `says bool`, Observe that both $\Pi; pc_q \vdash H \leq p^{\rightarrow}$ `says bool` and $\Pi; pc_q \vdash H \leq p \wedge q^{\rightarrow}$ `says bool`. Therefore, Theorem 1 applies as above for both x and y. Thus if $e[x \mapsto v_1] \longrightarrow^* v_1'$ and $e[x \mapsto v_2] \longrightarrow^* v_2'$, then $v_1' \simeq v_2'$. and if $e[x \mapsto v_1] \longrightarrow^* v_1''$ and $e[x \mapsto v_2] \longrightarrow^* v_2''$, then $v_1'' \simeq v_2''$.
- p **cannot open a value that hasn't been received.** Let $H = p^{\rightarrow} \wedge p^{\leftarrow}$. For any e and $\Gamma_{cro}; pc_p \vdash e : p^{\leftarrow} \wedge q$ `says bool`, observe that $\Pi; pc_p \vdash H \leq p^{\rightarrow}$ `says bool`, $\Pi; pc_p; pc_p \nVdash H^{\rightarrow} \sqsubseteq q^{\rightarrow}$, $\Pi; pc_p; pc_p \nVdash H^{\leftarrow} \sqsubseteq (p \wedge q)^{\leftarrow}$, and $\Pi; pc_p; pc_p \nVdash pc_p \succcurlyeq \nabla(H^{\rightarrow}) \wedge (p \wedge q)^{\leftarrow}$. Therefore, by Theorem 1, if $e[x \mapsto v_1] \longrightarrow^* v_1'$ and $e[x \mapsto v_2] \longrightarrow^* v_2'$, then $v_1' \simeq v_2'$.