A Calculus for Flow-Limited Authorization

Corrected version: September 16th, 2017

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

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.

I. 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 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). 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 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 II introduces commitment schemes and bearer credentials, two examples of dynamic authorization mechanisms we use to explore the features of FLAC. Section III reviews the FLAM principal lattice [12], and Section IV defines the FLAC language and type system. FLAC implementations of the dynamic authorization examples are presented in Section V, and their properties are examined. Section VI explores aspects of FLAC’s proof theory, and Section VII discusses semantic security guarantees of FLAC programs, including noninterference and robust declassification. We explore related work in Section VIII and conclude in Section IX.

II. 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 V that these mechanisms can be implemented using FLAC, and that their security goals are easily verified in the context of FLAC.

A. 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.

B. 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 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.

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

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

An advantage of this model is that secure information flow can be defined in terms of authority. An information flow policy \( q \) is at least as restrictive as a policy \( p \) if \( q \) has at least the confidentiality authority \( p \rightarrow \) and \( p \) has at least the integrity

\footnotetext{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.}

\footnotetext{3}{FLAM defines an additional set of operators called ownership projections, which we omit here to simplify our presentation.}
authority $q^-$. 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:

\[
p \sqsubseteq q \triangleq (p^- \land q^-) \triangleright (q^- \land p^-)
\]

\[
p \sqcap q \triangleq (p \land q^-) \triangleright (p \lor q \land p^-)
\]

\[
p \sqcup q \triangleq (p \lor q^-) \triangleright (p \lor q \land q^-)
\]

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 $\triangledown(p)$, as the integrity necessary to speak for that principal. Given a principal expressed in normal form$^4$ as $q^- \land r^-$, the voice of that principal is

\[
\triangledown(q^- \land r^-) \triangleq q^- \land r^-
\]

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

IV. 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, FLAM 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, $\Pi$, contains a set of labeled dynamic trust relationships $\langle p \triangleright q \mid \ell \rangle$ where $p \triangleright q$ (read as “$p$ acts for $q$”) is a delegation from $q$ to $p$, and $\ell$ is the confidentiality and integrity of that information. The typing context, $\Gamma$, is a map from variables to types, and $pc$ is the program counter label, a FLAC principal representing the confidentiality and integrity of control flow. The type system makes frequent use of judgments adapted from FLAM’s inference rules [12].

$^4$In normal form, a principal is the conjunction of a confidentiality principal and an integrity principal. See [12] for details.
Fig. 4: FLAC operational semantics

These rules, adapted to FLAC, are presented in Figure 6.5

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, $\Pi$. 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 example, if we have $\varphi \xrightarrow{\psi} \varphi'$ then (via APP) we may derive $\psi$ in a context with integrity $p^\rightarrow$, but not in contexts that don’t flow to $p^\rightarrow$. This feature offers needed control over how principals may apply existing facts to derive new facts.

Many FLAC terms are standard, such as pairs $(e_1, e_2)$, projections $\text{proj}_1 e$, variants $\text{inj}_j e$, polymorphic type abstraction $\Lambda X. e$, and case expressions. Function abstraction, $\lambda x: s)[pc]. e$, 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 FLAC judgment $\Pi; pc; pc \vdash pc \subseteq pc'$ holds. This judgment ensures that the current program counter label, pc, flows to the function label, pc'.

Branching occurs in case expressions, which conditionally evaluate one of two expressions. The rule CASE ensures that both expressions have the same type and thus the same protection level. The premise $\Pi; pc \vdash pc \leq s$ ensures that this type protects the current pc label.7

Fig. 5: FLAC type system.

Like DCC, FLAC uses monadic operators to track dependencies. The monadic unit term $(\eta \ell \nu) (\text{UNITM})$ says that a value $v$ of type $s$ is protected at level $\ell$. This protected value has the type $\ell$ says $s$, meaning that it has the confidentiality and integrity of principal $\ell$. 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 $\text{bind } x = (\eta \ell \nu) \in (\eta \ell' \nu) x$ is only well-typed if $\ell'$ protects
values with confidentiality and integrity $\ell$. 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 $\ell$. Product types protect information at $\ell$ if both components do (P-PAIR). Function types protect information at $\ell$ 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 $\ell$, then $\ell'$ says $s$ still does (P-LBL1). Finally, if $\ell$ flows to $\ell'$, then $\ell'$ says $s$ protects information at $\ell$ (P-LBL1.2).

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 $(p \gg q)$, read “$p$ acts for $q$”, is evidence that $q$ trusts $p$. Delegation values have acts-for types; $(p \gg q)$ has type $(p \gg q)$. The assume term enables programs to use evidence securely to create new flows between protection levels. In the typing context $\Theta; x:p \rightarrow s; q \leftarrow x; p \rightarrow s$, and $pc = q \rightarrow s$, the following expression is not well typed:

$$\text{bind } x' = x \text{ in } (\eta_p \rightarrow x')$$

since $p \rightarrow s$ does not flow to $q \rightarrow s$, as required by the premise $\Pi; pc \vdash x \in q \rightarrow s$ in rule BindM. Specifically, we cannot derive $\Pi; pc \vdash p \rightarrow s \rightarrow q \rightarrow s$ since P-LBL2 requires the FLAM judgment $\Pi; q \rightarrow s; q \rightarrow s \vdash p \rightarrow s \rightarrow q \rightarrow s$ to hold.

However, the following expression is well typed:

$$\text{assume } (p \rightarrow s \gg q \rightarrow s) \text{ in bind } x' = x \text{ in } (\eta_p \rightarrow x')$$

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

Creating a delegation value requires no special privilege because the type system ensures only high-integrity delegations are used as evidence that enable new flows. Using low-integrity evidence for authorization would be insecure since attackers could use delegation values to create new flows that reveal secrets or corrupt data. The premises of the Assume rule ensure the integrity of dynamic authorization computations that produce values like $(p \rightarrow s \gg q \rightarrow s)$ in the example above.9 The second premise, $\Pi; pc \vdash q \rightarrow s$, requires that the principal $p$ has sufficient integrity to be trusted by $q$, the principal whose security is affected. For instance, to make the assumption $p \gg q$, the evidence represented by the term $e$ must have at least the integrity of the voice of $q$, written $\nabla(q)$. Since the $pc$ bounds the restrictiveness of the dependencies of $e$, this ensures that only information with integrity $\nabla(q)$ or higher may influence the evaluation of $e$. The third premise, $\Pi; pc \vdash \nabla(q) \gg \nabla(q')$, ensures that principal $p$ has sufficient integrity to be trusted to enforce $q$’s confidentiality, $q'$. This premise means that $q$ permits data to be relabeled from $q \rightarrow s$ to $p \rightarrow s$.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

9These premises are related to the robust FLAM rule Lift.

10More precisely, it means that the voice of $q$’s confidentiality, $\nabla(q')$, permits data to be relabeled from $q \rightarrow s$ to $p \rightarrow s$. Recall that $\nabla(Alice^{\rightarrow s})$ is just Alice’s integrity projection: Alice$^{\rightarrow s}$.
ASSUME, it requires only that there exist a sufficiently trusted label \( p^c \) such that subexpression \( e \) type-checks. In the proofs in Section VII, we choose \( p^c \) using the typing judgment of the source-level ASSUME that generates the \texttt{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 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.

V. EXAMPLES REVISITED

We can now implement our examples from Section II 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 VII formalizes the semantic security properties of all well-typed FLAC programs.

A. 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 \( p^\rightarrow \).

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 \( p^\rightarrow \) can commit.

\[
\begin{align*}
\text{commit:} & \quad \forall X. p^\rightarrow \text{ says } X \quad \text{\( p^\rightarrow \) says } X \\
\text{commit =} & \quad \\Lambda X. \lambda (x:p^\rightarrow \text{ says } X)[p^\rightarrow] \\
& \quad \quad \text{assume } \langle \perp^\rightarrow \triangleright p^\rightarrow \rangle \text{ in bind } x' = x \in (\eta_p x') \\
\text{receive:} & \quad \forall X. p \text{ says } X \quad q^\rightarrow \text{ says } X \\
\text{receive =} & \quad \\Lambda X. \lambda (x:p \text{ says } X)[q^\rightarrow] \\
& \quad \quad \text{assume } \langle p^\rightarrow \triangleright q^\rightarrow \rangle \text{ in bind } x' = x \in (\eta_p \land q^\rightarrow x') \\
\text{open:} & \quad \forall X. p \land q^\rightarrow \text{ says } X \quad \forall (p^\rightarrow) \rightarrow p^\rightarrow \land q \text{ says } X \\
\text{open =} & \quad \\Lambda X. \lambda (x:p^\rightarrow \land q^\rightarrow \text{ says } X)[\forall (p^\rightarrow)] \\
& \quad \quad \text{assume } \langle \forall (q^\rightarrow) \triangleright \forall (p^\rightarrow) \rangle \text{ in } \\
& \quad \quad \quad \text{assume } \langle q^\rightarrow \triangleright p^\rightarrow \rangle \text{ in bind } x' = x \in (\eta_p \land q x') \\
\end{align*}
\]

Fig. 9: FLAC implementations of commitment scheme operations.
information at \( q^- \), but only has \( p^- \) 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 of \( b \).

Finally, \texttt{open} reveals a committed value to \( q \) by relabeling a value from \( p \land q^- \) to \( p^- \land q \), which is readable by principal \( q \) but retains the integrity of both \( p \) and \( q \). Since \texttt{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 \texttt{open} operation reveals a value with confidentiality \( p^- \), it should only be invoked by principals that are trusted to speak for \( p^- \). Otherwise, \( q \) could open \( p \)'s commitments. Hence, the \texttt{pc} label of \texttt{open} is \( \nabla(p^-) \). For \( p = \text{Alice} \), say, the \texttt{pc} label would be \( \text{Alice}^- \). FLAC ensures these constraints are specified correctly; otherwise, \texttt{open}’s implementation could not produce a value with label \( p^- \land q \).

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

In DCC, functions are not annotated with \texttt{pc} labels and may be applied in any context. So a DCC function analogous to \texttt{open} might have type

\[
\texttt{dcc\_open} : \forall X. p \land q^- \text{ says } X \rightarrow p^- \land q \text{ says } X
\]

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

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\(^{12}\), \( p \) 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:

\[
\lambda(x:p \land q^- \text{ says } X) [q^-], \text{assume } \langle q \gg p \rangle \text{ in open } x
\]

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 \( \text{pc}_p = \nabla(p), \text{pc}_q = \nabla(q) \), and

\[
\Gamma_{cro} = \text{commit, receive, open, } x:p^- \text{ says } s, y:p \land q^- \text{ says } s
\]

Intuitively, \( \text{pc}_p \) and \( \text{pc}_q \) are execution contexts under the control of \( p \) or \( q \), respectively. \( \Gamma_{cro} \) is a typing context for programs using the commitment scheme.\(^{14}\) 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: \( \Pi = \emptyset \).

Below, we omit the delegation context altogether for brevity. Using results presented in Section VII, we can prove that:

- \( q \) cannot receive a value that hasn’t been committed. For any \( e \) and \( s' \) such that \( \Gamma_{cro}; \text{pc}_q \vdash e : p \land q^- \text{ says } s' \), 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 \( e \), \( \ell \), and \( s' \) such that \( \Gamma_{cro}; \text{pc}_q \vdash e : \ell \land q^- \text{ says } s' \), 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 \( \Gamma_{cro}; \text{pc}_p \vdash e : p^- \land q \text{ says } s' \), then the result of \( e \) is independent of \( x \).

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 \( \Gamma_{cro}; \text{pc}_q \) context. In the first property, we are concerned with programs that produce values protected by policy \( p \land q^- \). 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 \( \ell \land q^- \) for any \( \ell \); these are values readable by \( q \). Therefore, we want to ensure that no program exists that enables \( q \) to produce such a value that depends on \( x \) or \( y \), which are not readable by \( q \).

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

B. Bearer Credentials

We can also use FLAC to implement bearer credentials, our second example of a dynamic authorization mechanism. We

\(^{12}\) i.e., it satisfies the \texttt{ Assume} premise \( \Pi; \text{pc}; p ; \nabla(p^-) \gg \nabla(q^-) \).

\(^{13}\)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 \texttt{commit, receive, and open} in \( \Gamma_{cro} \). Their types are as defined previously.
represent a bearer credential with authority $k$ in FLAC as a term with the type

$$\forall X. k^- \rightarrow X \searrow k^- \rightarrow X$$

which we abbreviate as $k^- \searrow k^-$. 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 $k^- = k^- \land \perp^-$ and $k^- = k^- \land \perp^-$. Then secrets protected by $k^-$ can be declassified to $\perp^-$, and untrusted data protected by $\perp^-$ can be endorsed to $k^-$. Thus this term wields the full authority of $k$, and if $pc = \perp^-$, 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 $k^- \searrow k^- \rightarrow \perp^-$ 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 $k^- \searrow k^- \rightarrow k^-$ until Section VI-B, but it is interesting to note that an analogous term in DCC is only well-typed if $k$ is equivalent to $\perp$. This is because the function takes an argument with $k^- \rightarrow$ confidentiality and no integrity, and produces a value with $k^-$ 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 $k^- \rightarrow k^-$, then $\mathcal{L}$ must have the property $k^- \subseteq \perp$ and $\perp \subseteq k^-$. 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 $k^- \searrow k^- \rightarrow k^-$ acts as a sort of caveat: it restricts the information flow context in which the credential may be used. We can add more general caveats to credentials by wrapping them in lambda terms. To add a caveat $\phi$ to a credential with type $k^- \searrow k^-$, we use a wrapper:

$$\lambda(x: k^- \searrow k^-)[pc]. \Lambda X. \lambda(y: \phi)[pc]. xX$$

which gives us a term with type

$$\forall X. \phi \searrow k^- \rightarrow X \searrow k^- \rightarrow X$$

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

$$\forall X. \phi_1 \searrow \ldots \searrow \phi_n \searrow k^- \rightarrow X \searrow k^- \rightarrow X$$

which abbreviates to

$$k^- \searrow \phi_1 \times \ldots \times \phi_n \searrow k^-$$

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

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

A subtle point about the way in which we construct caveats is that the caveats are polymorphic with respect to $X$, the same type variable the credential ranges over. This means that each caveat may constrain what types $X$ may be instantiated with. For instance, suppose $isEdu\mathcal{C}$ is a predicate for educational films; it holds (has a proof term with type $isEdu\mathcal{C} X$) for types like $Bio$ and $Doc$, but not $RomCom$. Adding $isEdu\mathcal{C} X$ 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 $isEdu\mathcal{C} RomCom$ could be applied, the bearer could only satisfy $isEdu\mathcal{C}$ 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, $x_1 : (k_1 \rightarrow \phi X)$ is a variable that stores a photo (type $\phi$) protected by $k_1$. For each such variable $x_1$, Alice has a credential $k_1 \rightarrow \searrow k_1^\rightarrow \searrow \perp^-$, 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:

$$\lambda(c: k_1 \searrow k_1^\rightarrow \phi X)[Bob^\rightarrow].$$

$$\text{bind } x_1' = c \times x_1 \text{ in } (\eta_{Bob^\rightarrow \searrow k_1^\rightarrow X} x_1')$$

Here, $c$ is a credential $k_1 \searrow k_1^\rightarrow \phi X$ whose polymorphic type has been instantiated with the type $\phi$ above. This credential accepts a photo protected at $k_1$ and returns a photo protected at $Bob^\rightarrow \searrow k_1^\rightarrow \phi X$, 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 $k_1 \searrow k_1^\rightarrow \searrow \perp^-$, 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 $k_1 \searrow \phi X$ whose polymorphic type has been instantiated with the photo type $\phi$, which Bob is permitted to access.

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

$$\lambda(c: k_1 \searrow k_1^\rightarrow \searrow \phi X)[Alice^\rightarrow].$$

$$\text{bind } y' = y \text{ in } (\eta_{Alice^\rightarrow \searrow k_1^\rightarrow X} y')$$

Then Bob can use this credential to access $x_1$ by deriving a
credential of type \( k_i \xrightarrow{\text{Bob}^{-1}} \text{Bob}^{-1} \land k_i^{-1} \text{ph} \) using the function

\[
\lambda(c:k_i \xrightarrow{\text{Bob}^{-1}} k_i^{-1})[\text{Bob}^{-1}].
\]

\[
\lambda(y:k_i \text{ says ph})[\text{Bob}^{-1}].
\]

\[
\text{bind } y' = c \text{ ph } y \text{ in } (\eta_{\text{Bob}^{-1} \land k_i^{-1}} y')
\]

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

\[
\lambda(c:k_i \xrightarrow{\text{Bob}^{-1}} k_i^{-1})[\text{Bob}^{-1}].
\]

\[
\text{assume } (\text{Carol}^{-1} \vdash \text{Bob}^{-1}) \text{ in } \lambda(\text{unit})[\text{Carol}^{-1}].
\]

\[
\text{bind } x_i' = c \text{ ph } x_i \text{ in } (\eta_{\text{Carol}^{-1} \land k_i^{-1}} x_i')
\]

creates a wrapper around a specific photo \( x_i \). Only principals trusted by Carol may invoke the wrapper, which produces a value of type \( \text{Carol}^{-1} \land k_i^{-1} \) says 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

\[\Gamma_{bc} = x_i:k_i \text{ says } s_i, c_i: \text{Alice says } (k_i^{-1} \xrightarrow{\lambda} k_i^{-1})\]

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 \( k_i \not\in \{\text{Alice.Friends}, p\} \) for all \( i \) where \( p \) represents a (potentially malicious) user of Alice’s service, and Friends is a principal for Alice’s friends, (e.g., Friends = (Bob ∨ Carol)). Also, define \( \text{pc}_p = p^{-1} \) and \( \text{pc}_A = \text{Alice}^{-1} \).

- **p cannot access resources without a credential.** For any \( e, \ell, \) and \( s' \) such that \( \Gamma_{bc}, c_p: (k_i^{-1} \xrightarrow{\lambda} k_i^{-1}); \text{pc}_p \vdash e: \ell \land p\rightarrow \text{says } s' \), the value of \( e \) is independent of \( x_i \) for all \( i \).
- **p cannot use unrelated credentials to access resources.** For any \( e, \ell, \) and \( s' \) such that

\[\Gamma_{bc}, c_p: (k_i^{-1} \xrightarrow{\lambda} k_i^{-1}); \text{pc}_p \vdash e: \ell \land p\rightarrow \text{says } s' \]

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

\[\Gamma_{bc}' = x_i:k_i \text{ says } s_i, c_i: \text{Alice says } (k_i^{-1} \xrightarrow{\lambda} k_i^{-1}),
\]

\[c_p: \text{Friends says } (k_i^{-1} \xrightarrow{\lambda} k_i^{-1})\]

Then if \( \Gamma_{bc}', c_p: A \vdash e: \ell \land p\rightarrow \text{says } s' \) for some \( e, \ell, \) and \( s' \), the value of \( e \) is independent of \( x_i \).

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 \( \ell \land p\rightarrow \) that depends on any variable \( x_i \). In the second, a credential \( c_p \) with type \( k_i^{-1} \xrightarrow{\lambda} k_i^{-1} \) is accessible to \( p \), but \( p \) cannot use it to access other variables. The third property eliminates covert channels like the one discussed in Section II-B. 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.

VI. FLAC PROOF THEORY

A. 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 \( \ell \):

\[\vdash (s_1 \rightarrow s_2) \rightarrow (\ell \text{ says } s_1 \rightarrow \ell \text{ says } s_2)\]

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

\[\vdash (s_1 \xrightarrow{\text{pc}_i} s_2) \xrightarrow{\ell} (\ell \text{ says } s_1 \xrightarrow{\text{pc}_i} \ell \text{ says } s_2)\]

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 \( \Pi, \Gamma, \) and \( \text{pc} \). A judgment in logical form is valid if a proof term exists for the specified type, proving the type is inhabited. The above type has proof term

\[\lambda(f: (s_1 \xrightarrow{\text{pc}_i} s_2))[\text{pc}]. \lambda(x: \ell \text{ says } s_1)[\text{pc}]. \text{bind } x' = x \text{ in } (\eta_x f x')\]

In order to apply \( f \), we must first bind \( x \), so according to rules BindM and App, the function \( f \) must have a label at least as restrictive as \( \text{pc} \lor \ell \). All theorems of DCC can be obtained by encoding them as FLAC implications with \( \text{pc} = \top \), the highest bound. Since any principal \( \ell \) flows to \( \top \), 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 V-A and V-B that the security of these mechanisms relied in part on restricting the \( \text{pc} \) 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: \( p^{-1} \xrightarrow{\text{pc}_p} p^{-1}. \)

Other properties of says common to DCC and other logics (cf. [27] for examples) are similarly refined by \( \text{pc} \) bounds. Two examples are:

\[\vdash s \xrightarrow{\text{pc}_i} \ell \text{ says } s \text{ which has proof term: } \lambda(x: s)[\text{pc}_i]. (\eta_x s) \text{ and } \lambda(\ell \text{ says } s_1)[\text{pc}_i]. \text{bind } x' = x \text{ in } \lambda(y: \ell \text{ says } s_1)[\text{pc}]. \text{bind } y' = y \text{ in } (\eta_x x' y')\]
As in DCC, chains of says are commutative in FLAC:
\[ \vdash \ell_1 \text{ says } \ell_2 \text{ says } s \xrightarrow{\text{pc}} \ell_2 \text{ says } \ell_1 \text{ says } s \]
with proof term
\[ \lambda(x: \ell_1 \text{ says } \ell_2 \text{ says } s)[\text{pc}]. \]
\[ \text{bind } y = x \text{ in } \text{bind } z = y \in (\eta_{\ell_2} (\eta_{\ell_1} z)) \]

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

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

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.

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.

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.

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.

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.
principals $a$ and $a \land b$. The following definitions are in service of this goal.

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

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

$$\Pi; pc; pc \vdash p \succ q \text{ and } \Pi; pc; pc \vdash q \succ p.$$  

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

**Definition 2 (Factorization).** A $(\Pi; pc)$-factorization of an ordered pair of principals $(p, q)$ is a tuple $(p, q_s, q_d)$ such that $\Pi; pc; pc \vdash q \equiv q_s \land q_d$ and $\Pi; pc; pc \vdash p \succ q_s$.

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

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 $(\Pi; pc)$-factorization $(p, q_s, q_d)$ of $(p, q)$ is **minimal** if for any $(\Pi; pc)$-factorization $(p, q'_s, q'_d)$ of $(p, q)$,

$$\Pi; pc; pc \vdash q'_s \succ q_s \text{ and } \Pi; pc; pc \vdash q'_d \succ q_d.$$  

The minimal factorization $(p, q_s, q_d)$ of $p$ and $q$ for a given $\Pi$ 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 $p \succ q_s$ and $q \equiv q_s \land q_d$, then speaking for $q_d$ is sufficient authority to cause $p$ to act for $q$ since adding the delegation $p \succ q_d$ would imply that $p \succ q$. This intuition also matches with the fact that $\Pi; pc; pc \vdash p \succ q_d$ if and only if $q_d = \bot$, which is the case if and only if $\Pi; pc; pc \vdash p \succ q$. Observe also that minimal $(\Pi; pc)$-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 $(p, q_s, q_d)$ be the minimal $(\Pi; pc)$-factorization of $(p, q)$. We define $q - p$ in $\Pi; pc$ to be $q_d$. That is, $\Pi; pc; pc \vdash q - p \equiv q_d$. 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 $(\Pi; pc)$ and principals $p, q$, there exists a minimal $(\Pi; pc)$-factorization of $(p, q)$.

**Proof.** Given $(p, q)$, we first let $q_s = p \lor q$. By definition, $\Pi; pc; pc \vdash p \equiv p \lor q$, and for all factorizations $(p, q'_s, q'_d)$, $\Pi; pc; pc \vdash p \succ q'_s$ and $\Pi; pc; pc \vdash q \succ q'_s$, so $\Pi; pc; pc \vdash q'_s \succ q'_s$.

Now let $D = \{ r \in \mathcal{L} \mid \Pi; pc; pc \vdash q \equiv q_r \land r \}$. All principals in $\mathcal{L}$ can be represented as a finite set of meets and joins of names in $N$ so $q$ and $q_s$ are finite. $\Pi$ 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 $\Pi; pc; pc \vdash q'_d \succ q_d$. First we see that $\Pi; pc; pc \vdash q \equiv q_s \land q_d$. $\Pi; pc; pc \vdash q'_s$ gives us $\Pi; pc; pc \vdash q \equiv q_s \land q_d$ 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 $\lor$ and $q_d$, $\Pi; pc; pc \vdash q'_d \succ q_d$. Thus we see that $(p, q_s, q_d)$ is a minimal $\Pi; pc$-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 $e \rightarrow e' \text{ where } v$. Then there exist $r, t \in \mathcal{L}$ and $\Pi' = (\Pi; (\nu^{q_s} \rightarrow t^{r} \mid pc))$ such that $\Pi; \Gamma; pc \vdash v : (\nu^{q_s} \rightarrow t^{r})$ and $\Pi'; \Gamma; pc \vdash e' : s$. Moreover, for all principals $p$ and $q$ if $\Pi; pc; pc \not\models \pi \models q \models q^\pi$, then

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

**Proof.** See Appendix A.

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

**B. Noninterference.**

Noninterference 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 $\Pi; pc \vdash H \subseteq s$ and $\Pi; pc \not\models H \subseteq s'$. 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 $\Pi' : \text{pc} \vdash H \leq s'$ to hold in a delegation context $\Pi'$ 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 $\Pi'$. Thus noninterference in FLAC is dependent on $H$ and its relationship to $\text{pc}$ and the type $s'$.

For more precision, Theorem 1 describes noninterference on confidentiality and integrity separately. It states that for some principal $H^+$ that flows to $s$ but not $\ell$ says bool, if pc cannot cause $H^+$ to speak for $\nabla(H^-)$ for confidentiality, or $\ell^+$ 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 $\Pi; \Gamma, x : s; \text{pc} \vdash e : \ell$ says bool. If there exists some $H$ and $\pi$ such that

1. $\Pi; \text{pc} \vdash H^+ \leq s$
2. $\Pi; \text{pc} ; \pi \not\vdash H^+ \subseteq \ell^+$
3) a) if $\pi = \rightarrow$ then $\Pi; \text{pc} ; \pi \vdash \nabla(H^-) - H^+$ b) if $\pi = \leftarrow$ then $\Pi; \text{pc} ; \pi \vdash (\ell - H)^+$

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

**Proof.** By Lemma 2 and subject reduction on a bracketed semantics. See Appendix A for details. \(\square\)

Condition 1 identifies $s$ as a “high” type—at least as restricted as $H$. Condition 2 identifies $\ell$ 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 $H^-$ and the voice $H^+$ or $\ell^+$. 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 V-A. The proofs of these properties are described in Appendix B.

**C. 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 $\text{pc}$ speaks for $H^+$ (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:

\[
\text{(Hole)} \quad \Pi; \text{pc} \vdash H^+ \leq \ell; \quad \Pi; \text{pc} ; \text{pc} \vdash H^+ \gg \nabla(p)c \quad \Pi; \text{pc} \vdash [\bullet] \vdash \ell,
\]

We write $e[\bullet]$ to denote a program $e$ with holes. Let an attack be a vector $\vec{a}$ of terms and $e[\vec{a}]$ be the program where $a_i$ is substituted for $\bullet$. An attack $\vec{a}$ is a fair attack [31] on a well-typed program with holes $e[\bullet]$ 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 $e[\bullet]$ be a program such that

1) $\Pi; \Gamma, x : s, \Gamma'; \text{pc} \vdash e[\bullet] : \ell$ says bool
2) $\Pi; \text{pc} ; \pi \not\vdash (\ell - h)^+$

Then for all attacks $\vec{a}_1$ and $\vec{a}_2$ and all inputs $v$ such that $\Pi; \Gamma, x : s, \Gamma'; \text{pc} \vdash e[\vec{a}_1] : \ell$ says bool and $\Pi; \Gamma, x : s, \Gamma'; \text{pc} \vdash v : s$, if $e[\vec{a}_1][x \mapsto v] \rightarrow^* v_1'$, then $v_1' = v_2'$.

**Proof.** By Lemma 2 and a generalization of Theorem 1 under attacks. See Appendix A for details. \(\square\)

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 $\ell^+$, the integrity of the result. Previous definitions of robust declassification [32], [31] forbid endorsement altogether; qualified robustness [32] permits endorsement but offers only probabilistic security.

**VIII. 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 Pfennig [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’s 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.

IX. 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


Syntax extensions

\[ v ::= \ldots \mid \{ v \} \]

Typing extensions

\[ \Pi; \Gamma; \sigma \vdash e_1 : s \quad \Pi; \Gamma; \sigma \vdash e_2 : s \]

[BRACKET]

\[ \Pi; \sigma; e \vdash (H \cup \sigma)^n \subseteq \sigma^n \quad \Pi; \sigma; \vdash H \subseteq s \]

Evaluation extensions

\[ e_i \rightarrow e'_i \quad e_j = e'_j \]

\[ \{ i, j \} = \{ 1, 2 \} \]

[STEP]

\[ (v_1 \mid v_2) \rightarrow (v_1 [v'_1] \mid v_2 [v'_2]) \]

[B-APP]

\[ (v_1 \mid v_2) v \rightarrow (v_1 [v'_1] \mid v_2 [v'_2]) \]

[B-TAPP]

\[ (v_1 \mid v_2) s \rightarrow (v_1 s \mid v_2 s) \]

[B-UNPAIR]

\[ \text{proj}_1(v_1 \mid v_2) \rightarrow (\text{proj}_1 v_1 \mid \text{proj}_2 v_2) \]

[B-CASE]

\[ \begin{cases} v_1 \text{ of inj}_1(x) \mid [v'_1]_1 \mid [v'_2]_2 \\ v_2 \text{ of inj}_2(x) \mid [v'_1]_2 \mid [v'_2]_2 \end{cases} \]

[B-UNITM]

\[ \eta \text{ of inj}_1(v_1 \mid v_2) \rightarrow (\eta v_1 \mid \eta v_2) \]

[B-BINDM]

\[ \text{bind } x = (v_1 \mid v_2) \text{ in } e \rightarrow \text{bind } x = v_1 \text{ in } e]_1 \text{ with } \text{bind } x = v_2 \text{ in } e]_2 \]

[B-ASSUME]

\[ \text{assume } v_1 \text{ in } e]_1 \text{ with } \text{assume } v_2 \text{ in } e]_2 \]

Fig. 10: Extensions for bracketed semantics

APPENDIX A
PROOFS OF NONINTERFERENCE AND ROBUSTNESS

Lemma 3 (Soundness). If \( e \rightarrow^* e' \) then \( [e]_1 \rightarrow [e']_1 \) and \( [e]_2 \rightarrow [e']_2 \).
Proof. By inspection of the rules in Figure 4 and Figure 10.

Lemma 4 (Completeness). If \( [e]_1 \rightarrow^* v_1 \) and \( [e]_2 \rightarrow^* v_2 \), then there exists some \( v \) such that \( e \rightarrow^* v \).

Proof. Assume \( [e]_1 \rightarrow^* v_1 \) and \( [e]_2 \rightarrow^* v_2 \). The extended set of rules in Figure 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 \( [e]_1 \) or \( [e]_2 \) diverge; this contradicts our assumption.

It remains to be shown that if the evaluation of \( e \) gets stuck, either \( [e]_1 \) or \( [e]_2 \) gets stuck. This is easily proven by induction on the structure of \( e \). Therefore, since we assumed \( [e]_1 \rightarrow^* v_1 \), then \( e \) must terminate. Thus, there exists some \( v \) such that \( e \rightarrow^* v \).

Lemma 5 (Substitution). If \( \Pi; \Gamma; : x \rightarrow s ; \mathsf{pc} \vdash e \rightarrow s \) and \( \Pi; \Gamma; ; \mathsf{pc} \vdash v \rightarrow s' \) then \( \Pi; \Gamma; ; \mathsf{pc} \vdash e[x \rightarrow v] \rightarrow s' \).

Proof. By induction on the typing derivation for \( e \).

Lemma 6 (Values). If \( \Pi; \Gamma; ; \mathsf{pc} \vdash v \rightarrow s \), then \( \Pi; \Gamma; ; \mathsf{pc'} \vdash v \rightarrow s \) for any \( \mathsf{pc'} \).

Proof. By induction on the derivation of \( \Pi; \Gamma; ; \mathsf{pc} \vdash e \rightarrow s \).

Lemma 7 (Robust transitivity). If \( \Pi; \mathsf{pc}; \ell; \vdash p \rightarrow q \) and \( \Pi; \mathsf{pc}; \ell; \vdash q \rightarrow r \), then \( \Pi; \mathsf{pc}; \ell; \vdash p \rightarrow r \).

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

Lemma 8 (Voices). If \( \Pi; \mathsf{pc}; \ell; \vdash p \rightarrow q \rightarrow q \) then \( \Pi; \mathsf{pc}; \ell; \vdash \nabla(p) \rightarrow \nabla(q) \rightarrow q \).

Proof. By induction on the derivation of \( \Pi; \mathsf{pc}; \ell; \vdash e \rightarrow s \) and \( \Pi; \mathsf{pc}; \mathsf{pc} \vdash \mathsf{pc} \subseteq \mathsf{pc'} \), then \( \Pi; \Gamma; ; \mathsf{pc} \vdash [e]_i \rightarrow s \).

Proof. By induction on the derivation of \( \Pi; \Gamma; ; \mathsf{pc} \vdash e \rightarrow s \) and Lemma 7. Note that BRACKET does not preserve this property, hence the projection of \( e \) is necessary.

Theorem 3 (Subject reduction). Suppose \( \Pi; \mathsf{pc} \vdash e \rightarrow s \) and \( [e]_i \rightarrow [e']_i \). If \( i \in \{1, 2\} \) then assume \( \Pi; \mathsf{pc} \vdash \Pi \subseteq \mathsf{pc} \).

Then \( \Pi; \Gamma; ; \mathsf{pc} \vdash e' \rightarrow s \).

Proof. By induction on the derivation of \( \Pi; \Gamma; ; \mathsf{pc} \vdash e \rightarrow s \). See the technical report [29] for details.

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

Lemma 10 (Delegation Factorization). If \( \Pi; \mathsf{pc}; \ell; \vdash p \rightarrow q \rightarrow q \) and \( (p, q, q_d) \) is the minimal static factorization of \( (p, q) \), then \( \Pi; \mathsf{pc}; \ell; \vdash p \rightarrow q_d \) and \( \Pi; \mathsf{pc}; \ell; \vdash \nabla(q_d) \).

Proof. A Coq-verified proof in [47] showed that there is some static factorization \( (p, q', q_d') \) such that \( \Pi; \mathsf{pc}; \ell; \vdash p \rightarrow q_d \) and \( \Pi; \mathsf{pc}; \ell; \vdash \nabla(q_d) \). By the definition of minimal factorization, \( \mathcal{L} \vdash q_d \rightarrow q_d \), so \( \mathcal{L} \vdash \nabla(q_d') \rightarrow \nabla(q_d) \), and by transitivity on static acts-for relationships, \( \Pi; \mathsf{pc}; \ell; \vdash p \rightarrow q_d \) and \( \Pi; \mathsf{pc}; \ell; \vdash \nabla(q_d) \).

Lemma 2 (Delegation Invariance). Let \( \Pi; \Gamma; ; \mathsf{pc} \vdash e \rightarrow e' \) where \( v \). Then there exist \( r, t \in \mathcal{L} \) and \( \Pi' = (\mathcal{V} \neq \mathcal{T} \mid \mathsf{pc}) \) such that \( \Pi; \mathsf{pc} ; \ell; \vdash v \rightarrow (r \rightarrow t) \) and \( \Pi'; \beta; \mathsf{pc} ; \ell; \vdash e' \rightarrow s \). Moreover, for all principals \( p \) and \( q \) if \( \Pi; \mathsf{pc} ; \ell; \vdash p \rightarrow q \) then \( \Pi'; \mathsf{pc} ; \ell; \vdash p \rightarrow q \).

Proof. Let \( (p, q, q_d) \) be the minimal \( (\mathsf{pc}, \ell)-\)factorization of \( (p, q) \) (so \( \Pi; \mathsf{pc} ; \ell; \vdash q_d \rightarrow q - p \)). First we note that \( q_d \neq q \) and thus \( \Pi; \mathsf{pc} ; \ell; \vdash p \rightarrow q \). Now assume for contradiction that \( \Pi'; \mathsf{pc} ; \ell; \vdash p \rightarrow q \).

We claim that \( \Pi; \mathsf{pc} ; \ell; \vdash t \rightarrow q \). Assume for contradiction that this is false. By transitivity \( \Pi'; \mathsf{pc} ; \ell; \vdash p \rightarrow q \), and since \( \Pi; \mathsf{pc} ; \ell; \vdash p \rightarrow q \), any derivation of this must use R-ASSUME with the delegation \( (r \rightarrow t) \rightarrow q \). This means that \( \Pi; \mathsf{pc} ; \ell; \vdash q_d \rightarrow t \rightarrow q_d' \) for some \( q_d' \). Assume without loss of generality that \( \Pi; \mathsf{pc} ; \ell; \vdash q_d \rightarrow t \rightarrow q_d' \). Otherwise the same argument would give us that \( \Pi; \mathsf{pc} ; \ell; \vdash q_d \rightarrow t \rightarrow q_d' \). 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 \( \Pi; \mathsf{pc} ; \ell; \vdash p \rightarrow q_d' \) (possibly because \( q_d' = \bot \)). However, this means that \( (p, q_d' \rightarrow q_d') \) is a valid \( (\mathsf{pc}, \ell)-\)factorization of \( (p, q) \). Since \( \Pi; \mathsf{pc} ; \ell; \vdash t \rightarrow q_d' \) by assumption, it is also the case that \( \Pi; \mathsf{pc} ; \ell; \vdash t \rightarrow q_d' \), which contradicts the assumption that \( (p, q, q_d) \) is minimal and thus \( \Pi; \mathsf{pc} ; \ell; \vdash t \rightarrow q_d \).

If \( p = \rightarrow \), then by WHERE with \( p; \ell; \vdash p \rightarrow \nabla(t) \rightarrow q \) and by Lemma 8, \( \Pi; \mathsf{pc} ; \ell; \vdash \nabla(q) \rightarrow \nabla(q_d') \). Thus, by transitivity, \( \Pi; \mathsf{pc} ; \ell; \vdash p \rightarrow \nabla(q_d') \), which contradicts our assumption. Similarly, if \( p = \leftarrow \), by WHERE with \( p; \ell; \vdash p \rightarrow \nabla(q_d') \), \( \Pi; \mathsf{pc} ; \ell; \vdash p \rightarrow \nabla(q_d') \), so by transitivity, \( \Pi; \mathsf{pc} ; \ell; \vdash p \rightarrow \nabla(q_d') \), which again contradicts our assumption.

Theorem 1 (Noninterference). Let \( \Pi; \Gamma; , x \rightarrow s ; \mathsf{pc} \vdash e \rightarrow \ell \mathsf{pc} \) sbool. If there exists some \( H \) and \( \pi \) such that

1) \( \Pi; \mathsf{pc} \vdash H \pi \leq s \)
2) \( \Pi; \mathsf{pc} \vdash H \pi \leq \ell \pi \)
3) a) if \( \pi = \rightarrow \) then \( \Pi; \mathsf{pc} \vdash \nabla(H \pi) \rightarrow H \rightarrow \pi \)
   b) if \( \pi = \leftarrow \) then \( \Pi; \mathsf{pc} \vdash \nabla(H \pi) \rightarrow (H \pi) \rightarrow \pi \)
then for all \( v_1, v_2 \) with \( \Pi; \mathsf{pc} \vdash v_1 \rightarrow s \), if \( e[x \rightarrow v_1] \rightarrow^* v'_1 \), then \( v'_1 \rightarrow v_2 \).

Proof. To prove this, we employ the bracketed semantics. By Lemma 6, \( \Pi; \Gamma; \vdash v_1 \rightarrow s \) and thus, for any \( H \), \( \Pi; \mathsf{pc} \vdash (v_1) \rightarrow (v_2) \).

We now examine the term \( e[x \rightarrow v_1] \rightarrow^* v_1 \). Thus by Lemma 4 there is some \( v' \)
such that \( e[x \mapsto (v_1 | v_2)H^?] \rightarrow^* v' \), and moreover, by Theorem 3 \( \Pi; \Gamma; pc \vdash v : \ell \) says \text{bool}.

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

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

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

Case \( (\pi = \leftarrow) \). In this case condition 2 means \( \Pi; pc; pc \not\vdash H^\ast \sqsubseteq \ell^\ast \). Since \( \Pi; pc; pc \not\vdash pc \not\vdash (\ell - H)^\ast \), 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 \( v'_1 = [v'_1]_1 = [v'_1]_2 = v'_2 \).

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 11.** For all principals \( p, q, \) and \( r \),
\[ \Pi; pc; pc \vdash (p - q) \land (q - r) \not\vdash p - r. \]

**Proof.** Let \( \overline{p_d} \) such that \( \Pi; pc; pc \vdash \overline{p_d} \equiv (p - q) \land (q - r) \). Now consider the following expression:
\[ \Pi; pc; pc \vdash [\overline{p_d} \lor p] \land (p \lor r) \equiv p \lor (\overline{p_d} \land r) \equiv p \lor ((p - q) \land (r \land (q - r))) \equiv p \lor p \equiv p. \]
Thus we see that for any \( (\Pi; pc)\)-factorization \((r, p \lor r, p_d)\) of \((r, p)\), \( \Pi; pc; pc \vdash \overline{p_d} \lor p \equiv p_d \) which means \( \Pi; pc; pc \vdash \overline{p_d} \lor p_d \). In particular, since \( (p - r) \) represents the minimal such factorization, \( \Pi; pc; pc \vdash (p - q) \lor (q - r) \not\vdash p - r. \)

**Lemma 12** (Noninterference under attacks). Let \( e[\bullet] \) be a program such that
1) \( \Pi; \Gamma; pc \vdash e[\bullet] : \ell \) says \text{bool}.
2) For all \( (p \not\vdash q; pc') \in \Pi; pc; pc' \equiv pc \).
3) \( \Pi; pc; pc \not\vdash H^\ast \sqsubseteq \ell^\ast \).
4a) if \( \pi = \rightarrow \) then \( \Pi; pc; pc \not\vdash \nabla(H \rightarrow) - H \).

Then for all attacks \( a_1^\ast \) and \( a_2^\ast \) such that \( \Pi; \Gamma; pc \vdash e[a_1^\ast] : \ell \) says \text{bool}, if \( e[a_1^\ast] \rightarrow^* v_1 \), then \( v_1 = v_2 \).

**Proof.** Using HOLE and Lemmas 2 and 11. See the technical report [29] for details.

**Theorem 2** (Robust declassification). Let \( e[\bullet] \) be a program such that
1) \( \Pi; \Gamma; x:s, \Gamma'; pc \vdash e[\bullet] : \ell \) says \text{bool}.
2) \( \Pi; pc; pc \not\vdash (\ell - h)^\ast \).

Then for all attacks \( a_1^\ast \) and \( a_2^\ast \) and all inputs \( v \) such that \( \Pi; \Gamma; x:s, \Gamma'; pc \vdash e[a_1^\ast] : \ell \) says \text{bool} and \( \Pi; pc \vdash v : s \), if \( e[a_1^\ast]|x \mapsto v \rightarrow^* v'_1 \), then \( v'_1 = v'_2 \).

**Proof.** Let \( e'[\bullet] = e[\bullet]|x \mapsto v \), and let \( a'_1 = a_1^\ast|x \mapsto v \). By Lemma 5, \( \Pi; \Gamma; pc \vdash e[a'_1^\ast] : \ell \) says \text{bool}. Thus by Lemma 12 \( v'_1 = v'_2 \).

**APPENDIX B**

**COMMUNITY SCHEME VERIFICATION**

To prove the desired properties of commitment schemes for boolean values, let \( s = \text{bool} \) and recall:
\( \Gamma_{cro} = \text{commit, receive, open, } x:p^\rightarrow \text{ says } s, y:p \land q^- \text{ says } s \)

- \( q \) cannot receive a value that hasn’t been committed. Let \( H = p^- \land q^- \). For any \( e \) and \( \Gamma_{cro}; pc_q \vdash e : p \land q^- \) says \text{bool}, observe that \( \Pi; pc_q \vdash H \not\vdash p^- \) says \text{bool}, \( \Pi; pc_q; pc \not\vdash H^- \sqsubseteq p^- \), \( \Pi; pc_q; pc \not\vdash \nabla(H^-) \sqsubseteq (p \land q^-)^\ast \), and \( \Pi; pc_q; pc \not\vdash pc_q \not\vdash \nabla(H^-) \sqsubseteq (p \land q^-)^\ast \). Therefore, by Theorem 1, if \( e[x \mapsto v_1] \rightarrow^* v'_1 \) and \( e[x \mapsto v_2] \rightarrow^* v'_2 \), then \( v'_1 \approx v'_2 \).

- \( q \) cannot learn a value that hasn’t been opened. Let \( H = p^- \land q^- \). For any \( e, \ell \), and \( \Gamma_{cro}; pc_q \vdash e : \ell \land q^- \) says \text{bool}. Observe that both \( \Pi; pc_q \vdash H \not\vdash p^- \) says \text{bool} and \( \Pi; pc_q \vdash H \not\vdash p \land q^- \) says \text{bool}. Therefore, Theorem 1 applies as above for both \( x \) and \( y \).

Thus if \( e[x \mapsto v_1] \rightarrow^* v'_1 \) and \( e[x \mapsto v_2] \rightarrow^* v'_2 \), then \( v'_1 \approx v'_2 \), and if \( e[x \mapsto v_1] \rightarrow^* v'_1 \) and \( e[x \mapsto v_2] \rightarrow^* v'_2 \), then \( v'_1 \approx v'_2 \).

- \( p \) cannot open a value that hasn’t been received. Let \( H = p^- \land q^- \). For any \( e \) and \( \Gamma_{cro}; pc_p \vdash e : p^- \land q^- \) says \text{bool}, observe that \( \Pi; pc_p \vdash H \not\vdash p^- \) says \text{bool}, \( \Pi; pc_p; pc_p \not\vdash H^- \sqsubseteq q^- \), \( \Pi; pc_p; pc_p \not\vdash \nabla(H^-) \sqsubseteq (p \land q^-)^\ast \), and \( \Pi; pc_p; pc_p \not\vdash pc_p \not\vdash \nabla(H^-) \sqsubseteq (p \land q^-)^\ast \). Therefore, by Theorem 1, if \( e[x \mapsto v_1] \rightarrow^* v'_1 \) and \( e[x \mapsto v_2] \rightarrow^* v'_2 \), then \( v'_1 \approx v'_2 \).