Expressive Authorization Policies using Computation Principals

***Abstract*—In authorization logics, it is natural to treat computations as principals, since systems need to decide how much authority to give computations when they execute. But unlike other kinds of principals, the authority that we want to give to computations might be based on properties of the computation itself, such as whether the computation is differentially private, or whether the computation is memory safe.**

Existing authorization logics do not treat computation principals specially. Instead, they identify computation principals using a brittle hash-based naming scheme: minor changes to the code produce a distinct principal, even if the new computation is equivalent to the original one. Moreover, existing authorization logics typically treat computation principals as "black boxes," leaving any reasoning about the structure, semantics, or other properties of the computation out of the logic.

We introduce Coal, a novel programming-language calculus that embeds an authorization logic in its type system via the Curry-Howard isomorphism. A key innovation of Coal is *computation principals*: computations that can be treated like other principals but also allow reasoning about the computation itself. Critically, Coal allows equivalent computations to be treated as equivalent principals, avoiding the brittleness of identity-based approaches to computation principals. Coal enables us to cleanly express fine-grained access control policies that are dependent on the structure and semantics of computations, such as expressing trust in all computations that are analyzed to be differentially private by any program analyzer that has been verified correct.

***Index Terms*—Authorization logics, access control and authorization, language-based security**

I. Introduction

Authorization logics provide a rigorous framework to formally specify access control policies and reason about them in a principled way. Central to authorization logic is the notion of *principals*: entities that can express statements about access control policies and may also be the subjects of access control policies. Principals may represent users, public keys, secure channels, but also processes and other computations.

Computations are natural entities to treat as principals: systems need to decide how much authority to give computations when they execute. The authority that a computation possesses might be derived from the system's trust in other principals, such as who wrote the computation or who requested its execution. Moreover, the challenge of authenticating principals (i.e., verifying the identity of an entity) also applies to computations. This is particularly true in remote settings where we must verify the identity of an executing computation without direct access to the computation's code or binary image.

Although computations should be treated as principals in authorization logics, they are nonetheless different from other principals. Computations have structure, semantics, and other properties that are relevant to the trust that may be placed in them. For example, two semantically equivalent computations that differ only in unimportant aspects should perhaps be trusted equally. However, existing authorizations logics typically reason about computations based on the identity of computations, often identifying a computation by a hash calculated from the computation's code [1]. This is brittle, since minor changes to the representation of the computation results in a completely distinct principal. Another example is that the reason *why* a particular computation is trusted may be due to properties of the computation, as opposed to who authored or provided the code. Existing authorization logics do not expose the details of computation and reasoning that a computation is, say, differentially private, must be done completely outside the logic.

In this work, we introduce Coal, a programming-language calculus that embeds an authorization logic in its type system, such that the authorization logic can reason about computations written in Coal. Using Coal, we can write expressive and fine-grained access control policies involving computations. These *computation principals* can be treated like other principals but also allow reasoning about the computation itself. Critically, Coal allows equivalent computations to be treated as equivalent principals, avoiding the brittleness of identity-based approaches to computation principals.

Moreover, Coal includes limited reflection mechanisms, enabling programs to compute over representations of other programs, and thus allowing the Coal authorization logic to express policies such as delegating authority to any program that successfully passes a given program analysis.

Although we focus on the metatheory and expressiveness of Coal in this paper, we have designed Coal so that it is amenable to implementation using trusted execution environments (TEEs) such as Intel's SGX enclaves. In other words, we have designed computation principals so that they could be authenticated using remote attestation mechanisms (or other mechanisms for identifying computations).

To understand why existing approaches to incorporating computations into authorization logics do not provide sufficient expressiveness, consider the following policy and how to express it in logic: "If program P is differentially private then Alice authorizes P to read the file medical_report.csv." To express this policy in logic, note that we need some way of distinguishing which programs are differentially private. Authorization logics would typically approach this by identifying computations as principals based on, for example, a hash-

based code signature and then allowing predicates over these principals. For example, the predicate (isDP H_P) indicates that the principal H_P—the hash of program P, representing P itself—is differentially private. However, this approach has two limitations.

First, identity-based approaches for naming computations are brittle: they are sensitive to superficial changes to programs. For example, if P is transformed to P' by running a dead-code elimination optimization pass, programs P and P' are semantically equivalent. However, even though (isDP H_P) is true, (isDP $H_{P'}$) may not hold, because $H_{P'}$ (the hash of the new program P') is distinct from H_P. Thus, Alice would grant access to P but not to P', despite these programs being essentially the same computation. Second, existing approaches rely on a third party to establish the predicate (isDP H_P). For example, Apple may issue a digitally signed certificate indicating H_P is differentially private, and that certificate could represent (isDP H_P). However, Apple is trusted by fiat in this approach. But since computations like P can be analyzed, a more principled approach would be to place the trust in (certificates issued by) program analyzers and verifiers. Coal addresses both these limitations.

Coal is both a programming calculus and—by exploiting the well-known Curry-Howard correspondence—also an authorization logic. Access control policies are formulas in the authorization logic and are also types in Coal; proofs for authorization logic theorems are terms inhabiting the corresponding types. The novelty of Coal lies in representing computation principals using dependent types, thereby enabling a uniform interaction between computations and the corresponding principals. This approach offers multiple advantages. First, it enables equational reasoning about computation principals: equivalent computations are equivalent principals. Second, using reflection mechanisms, we can express computations that make statements about other computations. For example, we could write a program analysis for differential privacy that takes as input a Coal program P and asserts the predicate (isDP P) if P passes the analysis. Third, the approach allows the expression of access control policies that give authority to programs based on their relevant properties. That is, instead of giving authority to a program because of who wrote it or who requested its execution, Coal can give authority based on properties of the computation, such as being provably memory safe or differentially private. This is an instantiation of the *principle of minimal identity* [2]. Unlike identity-based authorization, which can impinge on the privacy of the requester by revealing more attributes, they offer the flexibility to define policies and credentials containing only the information that is needed for authorization.

Using Coal we can specify a wide variety of access control policies involving computations. The elegance of Coal becomes evident in the specification of access control policies that are conditioned on the properties of computation principals. Below we show policies in which Alice delegates her authority to computations, refining the restrictions on the delegations, in each subsequent policy. The policies are thus increasingly fine-grained and more expressive; all are expressible in Coal.

- Alice delegates her authority to a *specific* computation P.
- Alice delegates her authority to a *specific* P if P is differentially private.
- Alice delegates her authority to *any* computation that is differentially private.
- Alice delegates her authority to *any* computation that a *specific* program analyzer claims as differentially private.
- Alice delegates her authority to *any* computation that *any* program analyzer—proven correct by a verifier (e.g., Coq)—claims as differentially private.

The key contribution of this work is the introduction of Coal, a novel programming calculus and authorization logic that unifies reasoning about computations and principals. Importantly, Coal supports equational reasoning about computation principals, offering a robust alternative to hash-based principal naming schemes. It allows expressive access control policies involving computations, which we demonstrate by examples throughout the paper. Though we discuss the standard metatheory, we do not present any formal security results; enforcing security is orthogonal to our work.

In Section II we give an informal overview of Coal, showing the key ideas. We present the basic formalism, including examples, in Sections III and IV.

In Section V we extend the formalism with support for reflection, enabling the expression within Coal of computations that reason about computations, such as program analyzers.

Although Coal is a calculus to demonstrate the key idea of unifying computations and principals, we strive to ensure that it is useful. In Section VI we discuss various choices and issues in turning Coal into a practical programming language. We cover related work in Section VII.

II. Coal Overview

Authorization logics provide a formal approach to express and enforce access control policies. A *proposition* is a statement that is either true or false. A *principal* is an entity with privilege that can be authenticated. An access control policy—expressed as a proposition in the authorization logic—determines if a principal's request to access a resource can be granted. Principals request operations or assert their beliefs using the says connective [1], [3]. The formula A says s represents that principal A produces the statement s, for example, on a secure channel or using a digital certificate. If s is some action (e.g., open a file), then A says s is a request for the operation. Broadly, authorization logics are concerned with answering why a principal can access a resource. Coal is a constructive authorization logic: it provides evidence—in the form of access control proofs—for all authorization decisions.

The core of Coal is based on polymorphic Dependency Core Calculus (DCC [4]). DCC is an extension of computational lambda calculus designed to capture the notion of dependency using a monad indexed by security principals.

Coal is both an authorization logic and a programming language. The dual nature of Coal is due to the Curry-Howard correspondence, where terms in Coal are both programs (i.e., terms that can be evaluated for their computational content) and proofs. Besides providing support for evaluating programs using standard term and type application, Coal also supports $\eta_A\ e$, a monadic unit-protected expression, along with the corresponding bind $x = e_1$ in e_2 expression. If the expression e has the type τ, then the protected expression $\eta_A\ e$ has the type A says τ. Intuitively, the term e in $\eta_A\ e$ is evaluated to a value v and the value v is signed and encrypted with A's key (and hence protected at the level of A). This is an instance of a normal program evaluated to a value.

In addition to representing sets of values that programs can evaluate to, types can represent propositions. For example, the type Alice says (MayPlay Bob s) represents an access control policy stating that Alice authorizes Bob to play the song s and the term sign(Alice, (MayPlay Bob s)) representing a signed statement (e.g., digital certificate) from Alice is a proof of the proposition Alice says (MayPlay Bob s).

Though Coal can be used for both programs and proofs, it does not mix them. Such a separation ensures that the proofs are independent of the effects of computations. For example, if P is some proposition, then bind $x = \eta_A\ 42$ in $\eta_A\ x$ is allowed but not bind $x = \mathsf{sign}(A, P)$ in $\eta_A\ 42$ or bind $x = \eta_A\ 42$ in $\mathsf{sign}(A, P)$. This is because $\mathsf{sign}(A, P)$ represents a proof whereas $\eta_A\ 42$ represents a value possibly produced by evaluating some program. Proofs and programs serve different purposes; combining them does not yield a meaningful interpretation. Coal's type system enforces the proof-computation separation.

In Coal, a principal is a type. It supports two kinds of principals: *named principals* and *computation principals*. Named principals—such as Alice and Bob—are the real world entities with an identity and are the standard atomic principals used in authorization logics. Note that Alice and Bob are types. Computation principals, on the other hand, represent computations that can be treated as principals.

Coal uses a dependent type to express computation principals: for computation e, the type $\mathsf{code}\{e\}$ is the corresponding computation principal. Coal also provides a special expression, $\mu T :: \mathsf{Comp}.(e : \tau)$, that can be treated both as a program and a principal. It is given the type $\mathsf{code}\{\mu T :: \mathsf{Comp}.(e : \tau)\}$—a principal representing the program $\mu T :: \mathsf{Comp}.(e : \tau)$. When treated as a program, it can be executed. For example, it may represent a TEE or a smart contract that can be invoked. When treated as principal, the corresponding type $\mathsf{code}\{\mu T :: \mathsf{Comp}.(e : \tau)\}$ can be used to issue signed statements, for example, an attestation report signed by a TEE.

The type system ensures that equivalent computations are treated as equivalent computation principals. This is the key to avoid the brittleness of the traditional hash-based naming schemes for principals. The authentication of computation principals is outside the scope of logic but important. We discuss the authentication aspects of computation principals in Section VI.

Coal enables principals to express their trust in other principals using the speaks-for relation: A speaks-for B states that the principal A can speak on the behalf of the principal B. In other words, B says P whenever A says P. The speaks-for relation represents the delegation of authority: Coal uses DCC's encoding for speaks-for relation: the type $\forall X :: \kappa.\ A$ says $X \to B$ says X, abbreviated as $A \overset{\kappa}{\Rightarrow} B$ encodes A speaks-for B. Intuitively, the annotation κ describes the set of statements for which the speaks-for relation holds. We omit the annotation when it is not important.

Computation principals enable expressive access control policies that directly depend on code. For example, if tax is a function that computes tax, then $\mathsf{code}\{tax\}$ is the corresponding computation principal. Principal Alice can express her trust in the computation tax using Alice says $(\mathsf{code}\{tax\} \Rightarrow \mathsf{Alice})$.

Coal supports a limited form of reflection to enable reasoning about computations within the logic. Using reflection, a program analyzer can inspect and analyze computations, and issue statements such as $(\mathsf{isDP}\ tax)$. Instead of trusting a computation by fiat, Alice can now rely on a program analyzer Z to do the analysis, and trust the computation tax only if Z says it is differentially private (indicated by $(\mathsf{isDP}\ tax)$). Formally, the access control policy is expressed as follows: $(Z\ \mathsf{says}\ \mathsf{isDP}\ tax) \to (\mathsf{code}\{tax\} \Rightarrow \mathsf{Alice})$. Coupled with reflection, computation principals thus boost the expressiveness of Coal. Using the concepts introduced so far, we show how Coal can be used to express access control policies in a real world system.

Case Study: eBPF Authorization Engine. The extended Berkeley Packet Filter (eBPF) [5] is a general-purpose virtual machine instruction set and provides a programmable interface to adapt kernel components at run-time to user-specific behaviors. eBPF allows an arbitrary eBPF user program to be run inside kernel at a higher privilege provided that it passes the static analysis of eBPF verifier, which statically checks that an eBPF program terminates and satisfies the restrictions on the syscalls.

An authorization engine based on Coal could elegantly express the above access control policy.

$$\mathsf{Kernel\ says}\ \big(\forall U :: \mathsf{Comp}.\ \mathsf{V_{eBPF}\ says}\ (\mathsf{terminates}\ U \wedge \mathsf{safeSyscalls}\ U) \to (U \Rightarrow \mathsf{Kernel})\big)$$

The policy expresses that the principal Kernel trusts any user eBPF program U—indicated by $(U \Rightarrow \mathsf{Kernel})$—provided the eBPF verifier $\mathsf{V_{eBPF}}$ asserts that U is both terminating (indicated by terminates U) and safe (indicated by safeSyscalls U).

Here both U and $\mathsf{V_{eBPF}}$ are computation principals. Notably, $\mathsf{V_{eBPF}}$ is treated as a computation, when it analyzes the user programs, and as a principal, when it issues signed statements related to the termination and safety of the user programs.

III. Formalizing Coal

This section presents the formal syntax, operational semantics and type system of Coal, along with metatheoretic results.

A. Syntax

Figure 1 shows the grammar for Coal which is stratified into kinds, types and terms. Metavariables κ, ℓ, and e range over kinds, principals, and terms respectively. Metavariables τ and P range over types, though we tend to use P to range over propositions (i.e., types of kind Prop).

Kinds reason about the well-formedness of types. Coal combines a programming language with an authorization logic, and we use kinds to keep the different aspects of our calculus distinct. There are four simple kinds. Kind $\star$ represents so-called "proper types" that are part of normal computation. That is, $\star$ is the kind of types of normal programs. Kind Prop represents propositions of the authorization logic. Kind Prin represents principals, which are used in both the authorization logic and normal computations. Finally, Comp is the kind of computation principals: programs that can be treated as principals. Kind Comp is a subkind of $\star$ and Prin, that is, types belonging to this kind represent both programs and principals.

A brief note on terminology: in the remainder of the paper we generally use: "program" to refer to terms with proper types, i.e., normal computations; "proof" to refer to terms with types that represent propositions (i.e., types with kind Prop); and "computation expression" to refer to the terms that Coal treats as both programs and principals, i.e., terms with types of kind Comp.

We allow type-level applications and use kinds to ensure the well-formedness of type applications. Kind $x{:}\tau \to \kappa$ is the kind of a type that can be applied to an expression of type τ to produce a type of kind κ, and $(X :: \kappa) \to \kappa'$ is the kind of a type that can be applied to a type of simple kind κ to produce a type of kind κ'.

Types include the standard unit type, sum type $\tau_1 + \tau_2$, product type $\tau_1 \times \tau_2$, and dependent function type $(x{:}\tau_1) \to \tau_2$. Type variable X, type abstraction $\forall X :: \kappa.\ \tau$, and type application $\tau_1\ \tau_2$ are also standard. We also allow types to be applied to expressions: $\tau\ e$. As mentioned, kinds ensure that types are applied appropriately.

A named constant principal n, such as Alice or Bob, represents a standard security principal. A computation principal $\mathsf{code}\{e\}$ represents the computation e as a principal. Note that this is a dependent type, as it has an expression e appearing in the type. Metavariable ℓ ranges over principals. Though sum and product types enable compound principals such as Alice + Bob and Alice × Bob, we do not explore their interpretation; principal algebra is orthogonal to our work.

Type ℓ says τ, when treated as a proposition (i.e., when given kind Prop), means that the principal ℓ asserts or supports the proposition τ.[1] When treated as a program type (i.e., with kind $\star$), ℓ says τ can be thought of as a value of type τ that is signed and encrypted by principal ℓ.

Standard terms include variables x, unit $()$, pair $\langle e_1, e_2\rangle$, projection $\mathsf{proj}_i\ e$, injection $\mathsf{inj}_i\ e$, and case expression $\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inj}_1(x).\ e_1 \mid \mathsf{inj}_2(x).\ e_2$. Term and type abstractions, $\lambda x{:}\tau.\ e$ and $\Lambda X :: \kappa.\ e$ respectively, are mostly standard except that type variables in the type abstractions are annotated with kinds. Term and type applications, $e\ e'$ and $e\ \tau$ are also included.

Term $\eta_\ell\ e$ is a monadic unit term protecting expressions at the level of the principal ℓ. For computational types, it can be thought of evaluating e to a value and then signing and encrypting it with ℓ's cryptographic keys. For propositional types, it can be thought of as proof of a proposition ℓ says τ where e is a proof of τ. Term bind $x = e_1$ in e_2 is the corresponding bind term operating on the protected expression e_1.

The term $\mathsf{sign}(\ell, P)$ also has type ℓ says P and can be thought of as an assertion by principal ℓ that proposition P holds, i.e., a proof of the proposition ℓ says P. This is similar to the sign term used in Aura [6].

Novel to Coal are the terms for the construction and execution of computation principals. The computation expression $\mu T :: \mathsf{Comp}.(e{:}\tau)$ constructs the computation principal type $\mathsf{code}\{\mu T :: \mathsf{Comp}.(e{:}\tau)\}$. The term e is annotated with its type τ. We treat expressions $\mu T :: \mathsf{Comp}.(e{:}\tau)$ as both programs and principals, which is part of the key expressive power of Coal.

Computation expressions may need to refer to themselves as principals for producing self-signed statements. For example, TEEs such as Intel SGX enclaves refer to their identity to derive cryptographic keys used during the attestation. The type variable T addresses this self-referential issue. Type variable T (with kind Comp) is bound in the scope of e and τ, and refers to the type of the computation expression. The computation e can now refer to itself as a principal using T. For example, the term $\mathsf{sign}(T, P)$ might appear within e, expressing that the computation signs a certificate, modeling an enclave's attestation operations. Note, however, that computation expressions can not express recursive computations; we discuss this further when we discuss the type system.

B. Evaluation

The judgment $e \twoheadrightarrow e'$ says that the term e evaluates to e' in a single step. Figure 2 shows the small step rules. Coal uses call-by-name (CBN) evaluation.

Rules E-ProjI, E-App, E-TApp, E-Case are standard. Note that the small step semantics never reduces ℓ in $\eta_\ell\ e$. This is intentional—the index of the monad is not evaluated even if there is a redex (e.g., term application in the principal $\mathsf{code}\{(\lambda x.\ e)\ ()\}$), however, the variables in the index may get substituted due to β-reduction and type application. Arguably, this complicates the type preservation lemma that requires deciding type equivalences in a dependently typed system. We shall revisit this challenge in Section III-C.

[1] Syntactically, the construct has the form τ' says τ but the kinding rules restrict τ' to principals.

$$
\begin{array}{lll}
\text{Kinds} & \kappa & ::= \star \mid \mathsf{Prop} \mid \mathsf{Prin} \mid \mathsf{Comp} \\
& & \mid\ x{:}\tau \to \kappa \mid (X :: \kappa) \to \kappa \\
\text{Principals} & \ell & ::= n \mid \mathsf{code}\{e\} \mid X \\
\text{Types} & \tau, P & ::= \mathsf{unit} \mid \tau + \tau \mid \tau \times \tau \\
& & \mid\ (x{:}\tau) \to \tau \mid X \mid \forall X :: \kappa.\ \tau \\
& & \mid\ n \mid \mathsf{code}\{e\} \mid \tau\ \mathsf{says}\ \tau \\
& & \mid\ \tau\ e \mid \tau\ \tau \\
\text{Values} & v & ::= () \mid \langle v, v \rangle \mid \mathsf{inj}_i\ v \mid \lambda x{:}\tau.\ e \\
& & \mid\ \Lambda X :: \kappa.\ e \mid \eta_\ell\ v \mid \mathsf{sign}(\ell, P) \\
& & \mid\ \mu T :: \mathsf{Comp}.(e{:}\tau) \\
\text{Terms} & e & ::= () \mid x \mid \langle e, e \rangle \mid \mathsf{inj}_i\ e \mid \mathsf{proj}_i\ e \\
& & \mid\ \lambda x{:}\tau.\ e \mid \Lambda X :: \kappa.\ e \mid e\ e \mid e\ \tau \\
& & \mid\ \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inj}_1(x).\ e \mid \mathsf{inj}_2(x).\ e \\
& & \mid\ \mathsf{sign}(\ell, P) \mid \eta_\ell\ e \mid \mathsf{bind}\ x = e\ \mathsf{in}\ e \\
& & \mid\ \mu T :: \mathsf{Comp}.(e{:}\tau) \mid \mathsf{exec}(e) \\
& E & ::= [] \mid \langle E, e \rangle \mid \langle v, E \rangle \mid \mathsf{proj}_i\ E \\
& & \mid\ \mathsf{inj}_i\ E \mid E\ e \mid E\ \tau \mid \mathsf{exec}(E) \\
& & \mid\ \mathsf{case}\ E\ \mathsf{of}\ \mathsf{inj}_1(x).\ e \mid \mathsf{inj}_2(x).\ e \\
& & \mid\ \eta_\ell\ E \mid \mathsf{bind}\ x = E\ \mathsf{in}\ e
\end{array}
$$

Fig. 1: Coal Syntax and Evaluation Contexts

Rule E-BIND is a monadic operation that operates on protected values. Given the term $\mathsf{bind}\ x = \eta_\ell\ v\ \mathsf{in}\ e$, it binds x to v in e. Note that $\mathsf{bind}\ x = \mathsf{sign}(\ell, P)\ \mathsf{in}\ e$ cannot be reduced but such terms are still useful and legitimate proof terms. The type system, however, separates proof and computational terms; it ensures that the computational terms do not get stuck.

Rule E-APP shows call-by-name function application: $(\lambda x{:}\tau.\ e)\ e'$ binds x to e' inside e. Call-by-name semantics allow the function application to proceed even when the argument is not a value. This is useful in our setting as the function application always succeeds even when some terms (e.g., propositions) cannot be reduced. For example, $(\lambda x{:}\tau.\ ())\ \mathsf{bind}\ x = \mathsf{sign}(\ell, P)\ \mathsf{in}\ e$ takes a step even though the argument cannot be reduced further (see E-BIND).

A computation expression $\mu T :: \mathsf{Comp}.(e{:}\tau)$ can be invoked with term $\mathsf{exec}(\mu T :: \mathsf{Comp}.(e{:}\tau))$. If e is the code to execute in a TEE, $\mathsf{exec}(e)$ can be thought of as setting up and executing the TEE. Rule E-EXEC shows that $\mathsf{exec}(\mu T :: \mathsf{Comp}.(e{:}\tau))$ steps to e where the type variable T is replaced with the corresponding computation principal $\mathsf{code}\{\mu T :: \mathsf{Comp}.(e{:}\tau)\}$. Coal allows term $\mu T :: \mathsf{Comp}.(e{:}\tau)$ to be treated as both a program and a principal; $\mathsf{exec}(\mu T :: \mathsf{Comp}.(e{:}\tau))$ is how the term is used as a program, i.e., exec evaluates the program. Note that the type system ensures that computation expressions can not recurse.

C. Type System

Most of the technical complexity in Coal is in the typing and kinding rules, which we present here. In all of the typing and kinding judgments, we use typing context Γ that maps program variables and type variables to types and kinds.

$$
\text{E-PROJI}\quad \mathsf{proj}_i\ \langle v_1, v_2 \rangle \twoheadrightarrow v_i
\qquad
\text{E-APP}\quad (\lambda x{:}\tau.\ e)\ e' \twoheadrightarrow e[x \mapsto e']
$$

$$
\text{E-TAPP}\quad (\Lambda X :: \kappa.\ e)\ \tau \twoheadrightarrow e[X \mapsto \tau]
\qquad
\text{E-CONTEXT}\quad \frac{e \twoheadrightarrow e'}{E[e] \twoheadrightarrow E[e']}
$$

$$
\text{E-BIND}\quad \mathsf{bind}\ x = \eta_\ell\ v\ \mathsf{in}\ e \twoheadrightarrow e[x \mapsto v]
$$

$$
\text{E-CASE}\quad \mathsf{case}\ \mathsf{inj}_i\ v\ \mathsf{of}\ \mathsf{inj}_1(x).\ e_1 \mid \mathsf{inj}_2(x).\ e_2 \twoheadrightarrow e_i[x \mapsto v]
$$

$$
\text{E-EXEC}\quad \mathsf{exec}(\mu T :: \mathsf{Comp}.(e{:}\tau)) \twoheadrightarrow e[T \mapsto \mathsf{code}\{\mu T :: \mathsf{Comp}.(e{:}\tau)\}]
$$

Fig. 2: Small step semantics using CBN evaluation strategy.

$$
\text{WT-TYPE}\quad \frac{}{\Gamma \vdash \star\ \mathsf{ok}}
\qquad
\text{WT-PROP}\quad \frac{}{\Gamma \vdash \mathsf{Prop}\ \mathsf{ok}}
\qquad
\text{WT-PRIN}\quad \frac{}{\Gamma \vdash \mathsf{Prin}\ \mathsf{ok}}
$$

$$
\text{WT-COMP}\quad \frac{}{\Gamma \vdash \mathsf{Comp}\ \mathsf{ok}}
\qquad
\text{WT-KTERMABS}\quad \frac{\Gamma \vdash_{kind} \tau :: \kappa' \qquad \Gamma, x{:}\tau \vdash \kappa\ \mathsf{ok}}{\Gamma \vdash x{:}\tau \to \kappa\ \mathsf{ok}}
$$

$$
\text{WT-KTYPEABS}\quad \frac{\Gamma \vdash \kappa\ \mathsf{ok} \qquad \Gamma, X :: \kappa \vdash \kappa'\ \mathsf{ok} \qquad \kappa \in \{\star, \mathsf{Prop}, \mathsf{Prin}, \mathsf{Comp}\}}{\Gamma \vdash (X :: \kappa) \to \kappa'\ \mathsf{ok}}
$$

Fig. 3: Kind well-formedness rules $\Gamma \vdash \kappa\ \ \mathsf{ok}$

The judgment $\Gamma \vdash \kappa\ \mathsf{ok}$ means that the kind κ is well formed under the typing context Γ. Figure 3 shows inference rules for this judgment. The key restriction is that in rule WT-KTYPEABS, kind $(X :: \kappa) \to \kappa'$ is well formed only if κ is a simple kind, i.e., we do not allow higher-order kinds.

a) Kinding Rules: The judgment $\Gamma \vdash_{kind} \tau :: \kappa$ says that the type τ is well kinded under the typing context Γ. Figure 4 shows rules for well-kinded types. Rules K-UNIT, K-PRIN, K-TYPEVAR, K-PROD and K-SUM are straightforward.

Rules K-ABS and K-TABS say that the kind of a function type is same as that of the kind of the output type. Additionally, input and output types have simple kinds (i.e., one of $\star$, Prop, Prin or Comp) to disallow passing type constructors as arguments or results.

Rule K-SAYS that the kind of $\tau\ \mathsf{says}\ \tau'$ is either $\star$ or Prop and is same as the kind of τ'. The type is meaningful only if τ is a principal and so the premise $\Gamma \vdash_{kind} \tau :: \mathsf{Prin}$ restricts the kind of τ to Prin.

Type application with a term $(\tau\ e)$ or a type $(\tau\ \tau')$ constructs new types (and hence we refer to them as type constructors). Rules K-APP-TERM and K-APP-TYPE ensure that type application to terms and type, respectively, are well formed. Our type constructors are similar to Aura [6] and can

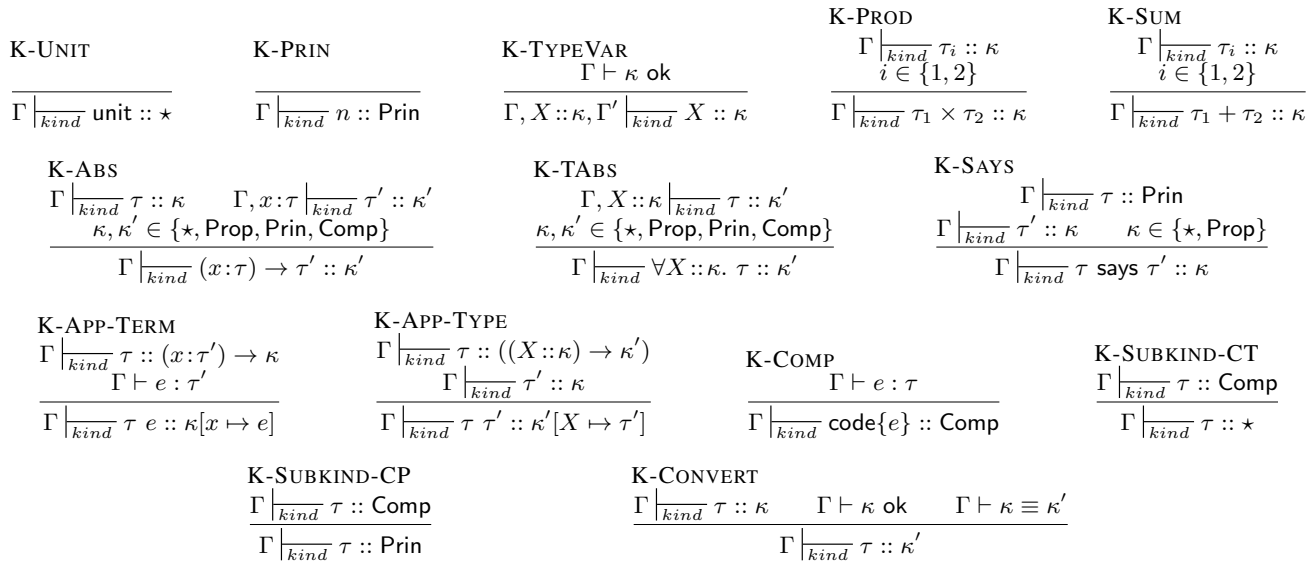

Fig. 4: Kinding rules $\Gamma \vdash_{kind} \tau :: \kappa$

be used to build new propositions. We demonstrate this with an example from Aura. Let MayPlay be a type variable with kind $(X :: \mathsf{Prin}) \to s : \mathsf{song} \to \mathsf{Prop}$. That is, it is a type constructor that takes a principal (a type X with kind Prin) and a term (with type song) as arguments. So MayPlay Bob freebird is a proposition, i.e., a type with kind Prop, that intuitively might represent that Bob has permission to stream the Lynyrd Skynyrd song Freebird.

Rule K-COMP says that the computation principal $\mathsf{code}\{e\}$ has kind Comp if the expression e is well typed. Rules K-SUBKIND-CP and K-SUBKIND-CT ensure that computation principals can be treated as both principals and as programs, by treating Comp as a subkind of the kinds Prin and $\star$, respectively.

Rule K-CONVERT says that if two kinds κ and κ' are equivalent under the typing context Γ, shown as $\Gamma \vdash \kappa \equiv \kappa'$, then a type having the kind κ can be converted to kind κ'. We do not present a definition of judgment $\Gamma \vdash \kappa \equiv \kappa'$ here, but discuss it later.

b) Typing Rules: Typing judgment, $\Gamma \vdash e : \tau$ says that the term e is well typed with type τ under the typing context Γ. Figure 5 shows the inference rules. Premises of the form $\Gamma \vdash_{kind} \tau :: \kappa$ appear in many rules and ensure that the type of the expression is well kinded.

Rules T-UNIT, T-VAR, T-PAIR, T-PROJI and T-INJI are standard. Rule T-CASE is standard except for requiring that the sum type and result type have the same kind. This allows terms having types of kind $\star$ to make progress. Otherwise, a case term with kind $\star$ may get stuck even if it is well typed (since in Coal, terms with kind Prop may get stuck). Note that the result type must be well formed under original typing context Γ to ensure that variable x does not escape.

For simplicity, named principal types are uninhabited. That is, no term can have the type n. This restriction is not fundamental and the language could be extended with term-level representations of named principals, in essence making named principals singleton types. However, doing so is orthogonal to our exploration of computational principals. Note that computation principals are inhabited and are singleton types.

Rule T-ABS is mostly straightforward except that it requires the resulting function type to be well kinded. This ensures that input and output types have simple kinds (see K-ABS). Similar restrictions apply to rule T-TABS: we restrict type abstraction to simple kinds, disallowing type constructors as arguments.

Rules T-APP and T-TAPP follow the standard dependent function and type application rules.

Rule T-UNITM requires that ℓ is a principal (i.e., has kind Prin) and expression e is well typed. Rule T-SIGN is similar to T-UNITM except that the proposition P has Prop kind. Note that no proof of P is required, and so propositions such as Bob says false can have proofs. But while a given principal might believe inconsistent propositions, this does not mean the logic fragment of Coal is inconsistent.

In a full-fledged programming language, term $\mathsf{sign}(\ell, P)$ would be a privileged operation and should require the approval or authority of principal ℓ. Such a programming language would need mechanisms to suitably restrict the use of a principal's authority or the use of a privileged operation that could damage a principal's security interests. We do not address this issue in our calculus, but discuss this and other implementation issues in Section VI.

Rule T-BIND requires that the kind of the result type is either $\star$ or Prop and is the same as the kind of the argument. This is required to ensure the separation of proofs and programs. Similar to T-CASE, we ensure that the result type does not allow the variable to escape its scope. The premise $\Gamma \vdash \tau'\ \mathsf{protects}\ (\ell\ \mathsf{says}\ \tau)$ says that the result type τ' is at least as restricted as the argument type $\ell\ \mathsf{says}\ \tau$.

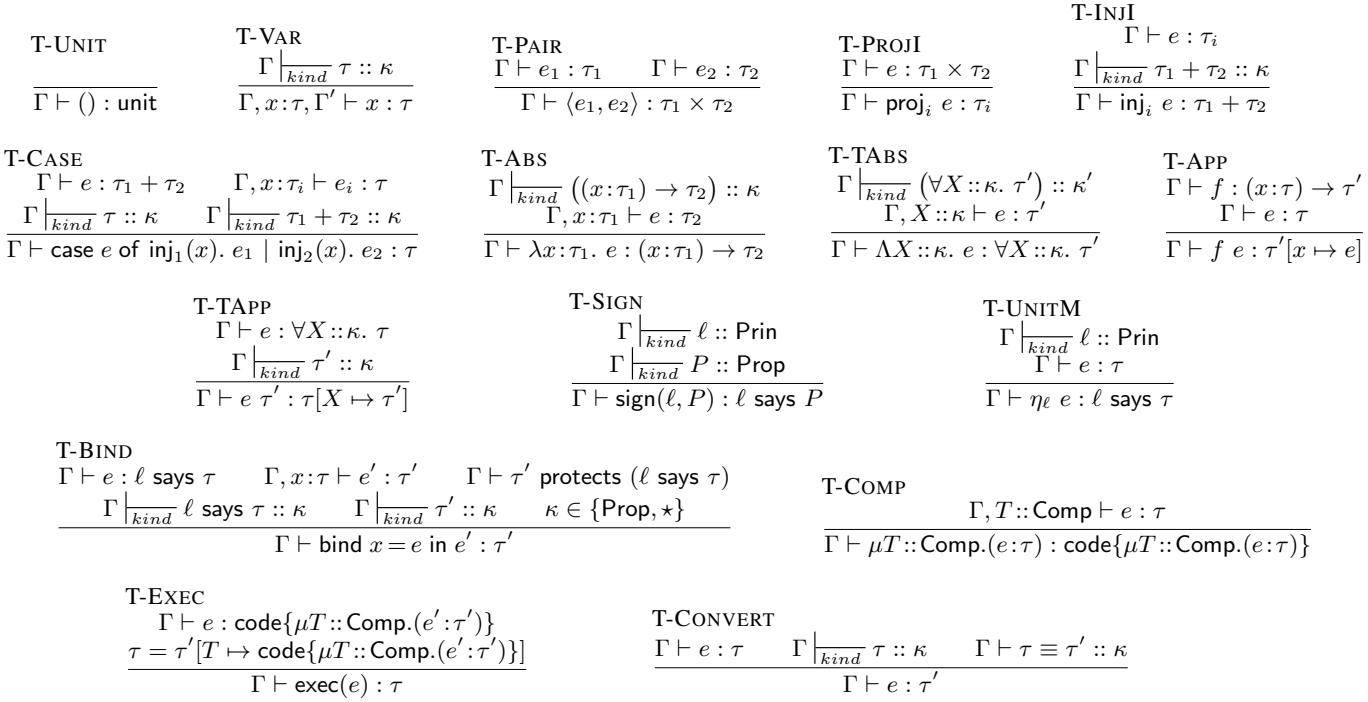

Fig. 5: Coal typing rules for expressions indicated by the judgment $\Gamma \vdash e : \tau$

One of the roles of the protects relation is to describe static trust relationships between principals, for example, maybe that principal Alice $\times$ Bob is at least as trusted as Bob, and thus type Alice $\times$ Bob says τ_1 protects type Bob says τ_2. Following Abadi [4], we leave the protects relation abstract but discuss possible instantiations in Section VI.

Rule T-COMP says computation expression $\mu T :: \mathsf{Comp}.(e:\tau)$ has type $\mathsf{code}\{\mu T :: \mathsf{Comp}.(e:\tau)\}$. The computation expression is well typed if e is well typed under Γ extended with T having Comp kind. The computation type $\mathsf{code}\{\mu T :: \mathsf{Comp}.(e:\tau)\}$ is a singleton type, as only the value $\mu T :: \mathsf{Comp}.(e:\tau)$ (and equivalent expressions) have that type. This allows the type system to reason precisely about the computation. It is also a recursive type: recall that when a computation expression is executed, T is replaced with the type of the computation expression, $\mathsf{code}\{\mu T :: \mathsf{Comp}.(e:\tau)\}$, giving the program e a way to refer to itself as a principal (e.g., $\mu T :: \mathsf{Comp}.(\mathsf{sign}(T, P) : T\ \mathsf{says}\ P)$). However, the type recursion does not allow diverging computations because terms with type T can not be used as arguments to exec. For example, $\mu T :: \mathsf{Comp}.(\lambda x : T.\ \mathsf{exec}(x)\ x : \tau)$ is ill-typed, since in expression $\mathsf{exec}(x)$, the variable x has type T which is not a suitable type for the argument to exec.

Indeed, rule T-EXEC says that $\mathsf{exec}(e)$ is well typed if e is a computation expression with type $\mathsf{code}\{\mu T :: \mathsf{Comp}.(e' : \tau')\}$. Note that in the result type, type variable T is replaced with $\mathsf{code}\{\mu T :: \mathsf{Comp}.(e' : \tau')\}$, consistent with the dynamic semantics.

Similar to K-CONVERT, T-CONVERT converts the type of an expression to an equivalent type. The type equivalence relation $\Gamma \vdash \tau \equiv \tau' :: \kappa$ says that type τ and τ' are equivalent at kind κ. Although we leave the general definition of type equivalence open, in Section III-D we define the equivalence of computation principals and show how Coal provides a principled framework to enable equational reasoning about programs (and hence computation principals).

EQUIV-COMP
$$\frac{\Gamma \vdash e_1 \equiv e_2 : \tau}{\Gamma \vdash \mathsf{code}\{e_1\} \equiv \mathsf{code}\{e_2\} :: \mathsf{Comp}}$$

Fig. 6: Computation Principal Equivalence

D. Principal Equivalence and Decidability

Coal's novelty lies in providing a flexible mechanism for establishing principal equivalence, even when programs can be treated as principals. Rule EQUIV-COMP in Figure 6 states the conditions under which two computation principals are equivalent: if terms e_1 and e_2 are equivalent then the computation principals $\mathsf{code}\{e_2\}$ and $\mathsf{code}\{e_2\}$ are equivalent.

Programming languages based on Coal can suitably instantiate the term equivalence definition to define equivalence classes of computation principals. Rule EQUIV-COMP enables Coal to offer a robust alternative to brittle hash-based naming schemes for computation principals.

In general, the notion of type equivalence is central to type checking a dependent-type system. Since terms appear

in types, checking if two types are equivalent may require checking if two terms are equivalent. However, the problem of general program equivalence is undecidable. To overcome this issue in dependent type systems, it is standard to define equality in way that makes type checking decidable. There are several options for equality ranging from syntactic equality to semantic equality with varying degrees of complexity and precision. We leave the term, type, and kind equivalences abstract as this is orthogonal to the expressiveness of authorization policies. We do note, however, that based on the motivation for this work, it may be useful to have definitions of term equivalence that could reason about the equivalence of source code and the compiled code that actually executes in a TEE; this may require advances to the state-of-the-art techniques for reasoning about term equivalence in dependent-type systems.

E. Metatheory

We prove the basic lemma that well-typed terms have well-kinded types as well as the standard type preservation lemma. Proofs are provided in Appendix D.

Lemma 1 (Well-typed terms have well-kinded types)**.** *For all well-formed typing contexts* Γ*, if* $\Gamma \vdash e : \tau$ *then* $\Gamma \vdash_{kind} \tau :: \kappa$ *for some kind* κ *such that* $\Gamma \vdash \kappa\ \mathsf{ok}$.

Lemma 2 (Well-kinded types have well-formed kinds)**.** *For all well-formed typing contexts* Γ*, if* $\Gamma \vdash_{kind} \tau :: \kappa$ *then* $\Gamma \vdash \kappa\ \mathsf{ok}$.

Lemma 3 (Type Preservation)**.** *If* $\emptyset \vdash e : \tau$ *and* $e \twoheadrightarrow e'$ *then* $\emptyset \vdash e' : \tau$.

Interestingly, progress holds only for well-typed terms with proper kind. A well-typed term having Prop kind may not make progress. For example, although bind $x = \mathsf{sign}(\ell, P)$ in $\mathsf{sign}(\ell, x)$ is well-typed (with type ℓ says P and kind Prop) under suitable typing environments, it gets stuck as there is no suitable term to use for variable x.

Lemma 4 (Progress)**.** *If* $\emptyset \vdash e : \tau$ *and* $\emptyset \vdash_{kind} \tau :: \star$ *then either* e *is a value or there exists* e' *such that* $e \twoheadrightarrow e'$.

Recall that Coal's type system separates propositions and proper types. Lemma 4 implies our separation of proofs and programs is correct. Otherwise, a program could depend on a proof term that is not guaranteed to reduce to a value and Lemma 4 would fail to hold.

Unsurprisingly, type soundness also holds only for well-typed terms having proper kinds.

Theorem 1 (Type Soundness)**.** *If* $\emptyset \vdash e : \tau$ *such that* $\emptyset \vdash_{kind} \tau :: \star$ *and* $e \twoheadrightarrow^* e'$ *then either* e' *is a value or exists* e'' *such that* $e' \twoheadrightarrow e''$.

Proof. Follows from Lemma 3 and Lemma 4. □

Logical Consistency It is important for the logical fragment of Coal to be consistent. Consistency would imply that there are no terms with types that are equivalent to false, such as $\forall X :: \mathsf{Prop}.\ X$.[2]

To ensure consistency, the language should be strongly normalizing, since a diverging term can inhabit any type. We conjecture that Coal does not have any diverging terms: Coal is based on the Calculus of Constructions which is strongly normalizing [7]. The only significant extension is the computation expression $\mu T :: \mathsf{Comp}.(e : \tau)$, and our type system ensures that they do not support recursion.

IV. Authority Delegations

Key to the expressive power of authorization logics is the ability for principals to delegate authority to other principals in carefully controlled ways, for example, delegating authority only under certain conditions or delegating authority only for certain statements. Coal is at least as expressive as most of these languages, allowing expressive delegations between principals. In this section, we show how to express a variety of delegation patterns in Coal. These patterns use principals generally; they are not specific to computation principals. In Section IV-A we present additional delegation patterns that use computation principals and demonstrate Coal's additional expressiveness.

Recall that we encode speaks-for relation using parametric polymorphism. That is, the type $\forall X :: \kappa.\ A$ says $X \to B$ says X, abbreviated as $A \overset{\kappa}{\Rightarrow} B$, encodes the relation A speaks-for B. A fundamental property of $A \overset{\kappa}{\Rightarrow} B$ is that it can be specialized with a specific statement.

Proposition 1 (Fundamental Property of speaks-for)**.** *Let* $\emptyset \vdash_{kind} \tau :: \kappa$*. Then, for all well-kinded principals* A *and* B*, there exists an expression* e *such that* $\emptyset \vdash e : (A \overset{\kappa}{\Rightarrow} B) \to (A\ \mathsf{says}\ \tau \to B\ \mathsf{says}\ \tau)$.

Proof. The required term is $\lambda x : A \overset{\kappa}{\Rightarrow} B.\ (x\ \tau)$. □

By constructing a term of the type $(A \overset{\kappa}{\Rightarrow} B) \to (A\ \mathsf{says}\ \tau \to B\ \mathsf{says}\ \tau)$, we have the authorization proof for the corresponding access control policy. This is an instance of Curry-Howard correspondence.

The speaks-for relation, $A \overset{\kappa}{\Rightarrow} B$, is a weaker form of delegation of authority: it does not require the support of B to vouch for $A \overset{\kappa}{\Rightarrow} B$. *Hand-off*, on the other hand, encoded as $(B\ \mathsf{says}\ A \overset{\kappa}{\Rightarrow} B) \to A \overset{\kappa}{\Rightarrow} B$, is a strong form of delegation. Informally, it says that B supports A to speak on the behalf of B. Similar to DCC, hand-off is a theorem in Coal, however, it is slightly different due to the protection relation. '

Proposition 2 (Hand-off)**.** *Let* A *and* B *be well-kinded principals. That is,* $\emptyset \vdash_{kind} A :: \mathsf{Prin}$ *and* $\emptyset \vdash_{kind} B :: \mathsf{Prin}$. *Then, for all* A *and* B*, there exists an expression* e *such that* $\emptyset \vdash e : (B\ \mathsf{says}\ A \overset{\kappa}{\Rightarrow} B) \to A \overset{\kappa}{\Rightarrow} B$.

[2]Note that our authorization logic is a modal logic and allows principals to believe inconsistent things. For example, $\mathsf{sign}(\mathsf{Bob}, \forall X :: \mathsf{Prop}.\ X)$ is a proof of the proposition Bob says $\forall X :: \mathsf{Prop}.\ X$, but this does not mean that the logic itself is inconsistent.

Proof. The required term is $\lambda x\!:\!(B\ \mathsf{says}\ A \overset{\kappa}{\Rightarrow} B).\ \Lambda X :: \kappa.\ \lambda y\!:\!A\ \mathsf{says}\ X.\ \mathsf{bind}\ z = x\ \mathsf{in}\ ((z\ X)\ y)$. □

The proof for Proposition 2 is a function that requires multiple arguments: a delegation $A \Rightarrow B$ signed by B, a type to instantiate the type variable X in the type abstraction, and a term signed by A. The function binds the first argument and applies it to the values signed by A to yield a term signed by B. For example, if the input is $\eta_A\ 42$ then the output is $\eta_B\ 42$. Note that to invoke this function, a proof of delegation (e.g., $\mathsf{sign}(B, A \overset{\star}{\Rightarrow} B)$) is required.

Hand-off applies uniformly to all principals, including computation principals. Corollary 1 in the next section instantiates Proposition 2 with computation principals.

Delegations can be restricted along two axes. First, delegations can be restricted along the axis representing the set of statements. A principal may only delegate its authority for few statements belonging to a set. Coal supports this naturally. The delegation $A \overset{\kappa}{\Rightarrow} B$ holds only for the set of statements having the kind κ. Second, delegations can be restricted along the principal axis. That is, besides restricting the statements for which A can speak for B, Coal also enables fine grained restrictions on principals that can speak for B. In the following example we show one such policy where any principal that satisfies isGood can speak for the principal B.

Example IV.1 (Restricted delegation)**.** Let $(\mathsf{isGood}\ L)$ be a proposition that the input principal L is trustworthy. Informally, principal B delegates its authority to any trustworthy principal Formal encoding of the policy is: $B\ \mathsf{says}\ (\forall L :: \mathsf{Prin}.\ \mathsf{isGood}\ L \to L \overset{\star}{\Rightarrow} B)$. Note that L is a type variable ranging over principals. □

Example IV.1 leads to an exciting possibility for computation principals: other principals can now specify which programs they trust. Moreover, predicates over programs naturally express the policy makers' intent. In the next two sections (Sections IV-A and V), we present examples that demonstrate delegations to computation principals.

A. Delegation to Computation Principals

The hand-off property and delegation policies from the previous section extend uniformly to computation principals. We first show a simple example in which a specific computation principal speaks on the behalf of a principal.

Example IV.2 (Delegation to a computation principal)**.** Let $\mathsf{code}\{\mu T :: \mathsf{Comp}.(genprime : \mathsf{int} \to \mathsf{prime})\}$ be a computation principal that generates a prime number given an input size. The delegation $A\ \mathsf{says}\ (\mathsf{code}\{\mu T :: \mathsf{Comp}.(genprime : \mathsf{int} \to \mathsf{prime})\} \overset{\star}{\Rightarrow} A)$ states that A delegates its authority to the computation principal $\mathsf{code}\{\mu T :: \mathsf{Comp}.(genprime : \mathsf{int} \to \mathsf{prime})\}$. □

Computation principals need not be closed and may contain open terms. Intuitively, a non-closed computation principal represents a family of principals indexed by open terms and a delegation to such a computation principal is a delegation to a family of computation principals indexed by open terms. For example, $\mu T :: \mathsf{Comp}.(x + 1 : \mathsf{int})$ represents a family of enclaves that increment the value of x.

We can also specify access control policies that range over computation principals. In the following example, principal A delegates its authority to all computation principals.

Example IV.3 (Delegation to all computation principals)**.** The policy $A\ \mathsf{says}\ (\forall L :: \mathsf{Comp}.\ L \overset{\star}{\Rightarrow} A)$ says that principal A allows any computation principal to speak on its behalf. In other words, A delegates its authority to any computation principal. Note that A could itself be another computation principal. □

Example IV.4 refines the policy further by restricting the delegation to computation principals that satisfy the predicate isGood. This is analogous to Example IV.1 but is ranged over computation principals.

Example IV.4 (Restricted Delegation to Computation Principals)**.** The delegation $A\ \mathsf{says}\ (\forall L :: \mathsf{Comp}.\ \mathsf{isGood}\ L \to L \overset{\star}{\Rightarrow} A)$ says that A delegates its authority to any trustworthy computation principal, indicated by the proposition $(\mathsf{isGood}\ L)$. □

Hand-off to a computation principal is no different from the hand-off to a named principal. Instantiating Proposition 2 with the kind for computation principals gives us Corollary 1. The proof term is the same as that of Proposition 2.

Corollary 1 (Hand-off to a Computation Principal)**.** *Let A and B be well-kinded principals such that B is a computation principal. That is, $\emptyset \vdash_{kind} A :: \mathsf{Prin}$ and $\emptyset \vdash_{kind} B :: \mathsf{Comp}$. Then, for all A and B, there exists an expression e such that $\emptyset \vdash e : (B\ \mathsf{says}\ A \overset{\kappa}{\Rightarrow} B) \to A \overset{\kappa}{\Rightarrow} B$.*

V. Reflection for More Expressive Policies

To enable reasoning about program properties directly within Coal, we need to extend it to a full-fledged program logic. One could imagine accomplishing this by embedding Coal in Coq, and taking full advantage of Coq's reasoning about computations. This is a completely reasonable approach. However, in this paper, we explore an approach that allows the explicit reasoning about program analyses, since program analyses are themselves programs in which we can express trust.

In general, the reason why a principal P might trust a given program e is because e has passed a program analysis, for example, a program analysis that verifies a program is differentially private. However, the analyzer is itself a program, and the reason why P trusts e is due to P's trust of the analyzer.

We can express such trust relationships in Coal, provided we extend Coal with a reflection mechanism to allow computations (such as program analyzers) to examine the representation of programs. Coal supports *quotation*: a reflection mechanism to convert terms to their representation. Reflective

$\tau ::= \ldots \mid \mathsf{AST}$

(a) Extended types

K-REFLECT
$\Gamma \vdash_{kind} \mathsf{AST} :: ((X :: \kappa) \to \star)$

(b) Kinding judgment for reflection

Fig. 7: Coal type extensions

reasoning enables expressing a whole new class of authorization policies that use program analyzers to specify the properties of the programs.

We extend the Coal grammar in Figure 1 with the type family AST and the function quote. Figure 7 shows the type extension and the corresponding kinding rule.

The type $\mathsf{AST}\ \tau$ is the type of representations of expressions of type τ. Rule K-REFLECT states this formally: the type $\mathsf{AST}\ \tau$ is well-kinded if τ is well-kinded. Also, since the representation of a term is available during runtime, $\mathsf{AST}\ \tau$ has proper kind.

The function $\mathsf{quote} : \forall X :: \star.\ X \to \mathsf{AST}\ X$ takes a term of type τ and returns the representation of that term. We leave open the mechanism for getting actual representation for a term. This is orthogonal to our work and there are a variety of approaches [8]–[11].

AST and quote give new expressive power to Coal: they enable reasoning about properties of the computation principals within Coal, thereby reducing the need to trust third-party credentials.

Armed with this new machinery, we can now specify policies that explicitly refer to program analyzers. We present two examples, one more complex than the other. The first example expresses a policy that delegates authority to any program that passes a specific program analyzer.

The second example refines the condition further by requiring that the program passes any verified program analyzer.

Example V.1 (Restricted delegation to a *statically analyzed* computation principal)**.** Let $(\mathsf{isDP}\ L)$ be a proposition expressing that the computation principal L is differentially private. That is, $\Gamma_0 \vdash_{kind} \mathsf{isDP} :: (L :: \mathsf{Comp}) \to \mathsf{Prop}$ where Γ_0 is our initial typing environment. Before we specify the policy involving the program analyzer and computation principals, we breakdown the terms and types for better readability.

$$Z \triangleq \mathsf{code}\{\mu Z :: \mathsf{Comp}.(\mathit{analyze} : \tau_z)\}$$
$$\tau_z \triangleq \forall L :: \mathsf{Comp}.\ \mathsf{AST}\ L \to (\mathsf{unit} + Z\ \mathsf{says}\ (\mathsf{isDP}\ L))$$

The program analyzer Z is a computation principal $\mathsf{code}\{\mu Z :: \mathsf{Comp}.(\mathit{analyze} : \tau_z)\}$; the corresponding computation expression $\mu Z :: \mathsf{Comp}.(\mathit{analyze} : \tau_z)$ uses the function $\mathit{analyze}$ to carry out the program analysis. Intuitively, the analyzer is run inside a TEE, and analyzes if the input program is differentially private.

Function $\mathit{analyze}$ has the type τ_z. The argument is an internal representation of a computation principal $\mathsf{AST}\ L$ (obtained by invoking quote). The function emits $(\mathsf{isDP}\ L)$ if the computation principal is differentially private, else it just returns unit.

Informally, the policy says that the principal A delegates to any computation principal L if the program analyzer Z indicates that L is differentially private. The formal encoding of the policy is: $A\ \mathsf{says}\ \forall L :: \mathsf{Comp}.\ Z\ \mathsf{says}\ (\mathsf{isDP}\ L) \to (L \overset{\star}{\Rightarrow} A)$

Interestingly, this policy captures the policy enforced by DuetSGX, a system that is executed inside Intel's SGX enclaves and analyzes if the input queries are differentially private [12]. □

We can further generalize the policy to allow A to delegate its authority based on *any* analyzer A that is verified to correctly check for differential privacy. We do of course need some root of trust: how does B know that A correctly checks for differential privacy? We can express this by relying on a verifier such as Coq.

Example V.2 (Restricted delegation to any *certified* program analyzer)**.** Let Z be a computation principal for analyzing a program and $(\mathsf{isDPAnalyzer}\ Z)$ be a proposition expressing that Z is an analyzer for differential privacy. That is, $\Gamma_0 \vdash_{kind} \mathsf{isDPAnalyzer} :: ((Z :: \mathsf{Comp}) \to \mathsf{Prop})$ where Γ_0 is our initial typing environment. Note the indirection here: Z is a program analyzer and $(\mathsf{isDPAnalyzer}\ Z)$ indicates if Z is a differential privacy analyzer.[3]

We reuse the predicate isDP from the previous example. Once again, we break down the terms and types for better readability.

$$V \triangleq \mathsf{code}\{\mu V :: \mathsf{Comp}.(\mathit{coq} : \tau_c)\}$$
$$\tau_c \triangleq \forall Z :: \mathsf{Comp}.\ \mathsf{AST}\ Z \to V\ \mathsf{says}\ (\mathsf{isDPAnalyzer}\ Z)$$

Verifier V is the root of trust and is the computation principal $\mathsf{code}\{\mu V :: \mathsf{Comp}.(\mathit{coq} : \tau_c)\}$; $\mu V :: \mathsf{Comp}.(\mathit{coq} : \tau_c)$ is the corresponding computation expression with function coq carrying out the actual verification. The argument is an internal representation of the computation principal. If Z passes the verification passes, V certifies it issuing the statement $(\mathsf{isDPAnalyzer}\ Z)$.

Informally, the policy says that the principal A delegates to a computation principal L only if the program analyzer Z—certified by the verifier V as functionally correct—indicates that L is differentially private. The formal encoding of the policy is shown below.

$$A\ \mathsf{says}\ \big(\forall Z :: \mathsf{Comp}.\ (V\ \mathsf{says}\ (\mathsf{isDPAnalyzer}\ Z)) \to \forall L :: \mathsf{Comp}.\ Z\ \mathsf{says}\ (\mathsf{isDP}\ L) \to L \overset{\star}{\Rightarrow} A\big)$$

Note that Z ranges over computation principals in this example whereas it is a fixed computation principal in the previous example (Example V.1). □

[3]Note that this is *different* from checking that Z itself is differentially private.

VI. Discussion

The key contribution of Coal is to enable specifying expressive access control policies that are dependent on the properties of the computations. To this end, we have so far focused on how to express various interesting policies using Coal, and intentionally left abstract several concepts that are not fundamental to the expressiveness of the logic. In this section, we discuss the open-ended nature of Coal, and how one can instantiate it to obtain a concrete programming language. Specifically, we focus on the choices for protects relation, realizing abstractions for computation principals as well as the useful security extensions.

A. Protection relation

The protects relation used in the rule T-Bind is left abstract. Intuitively, if $\Gamma \vdash \tau$ protects τ', then τ is entrusted to protect the information having type τ' under the contexts Γ. For example, unit does not reveal any new information and thus it protects information at any level. As another example, it might be reasonable to assume that when Alice makes a statement, then it is protected at the level of Alice. More generally if Alice trusts Bob, then Bob protects her statements. That is, Bob is entrusted to not act adversely with information owned by Alice.

Protection abstracts several design choices about the says modality that are orthogonal to the contributions; a real programming language would need to make the relation concrete. Several variants of protection relation can be found in works based on DCC [6], [13]–[16]. Most of these authorization logics define the protection relation between principals and types: a type τ is protected at the level of principal A.

A simpler protection relation is an identity relation where a type protects itself. Both CDD and Aura use a very basic protects relation that eschews the principal lattice [6], [17]. In the context of authorization logic, this is less interesting.

In DCC, the protection relation is slightly more complex and uses an underlying lattice of principals that are ordered by trust: more trusted principals protect the information influenced by less trusted principals. The protection relation does not distinguish between Alice says (Bob says τ) and Bob says (Alice says τ). This might be useful in settings where the order of influences of the principals is not important. Other works have restricted the commutativity of says [15], [16]. This enables a more fine-grained reasoning on the order in which principals influenced the information (e.g., traffic analyzers).

B. Towards a Real Language

Coal is a foundation for developing secure programming languages that unify programming and access control: a language chooses a specific point in the spectrum of design choices. In the minimal case, a real language has to define the protects relation, choose an implementation for computation principals, and enforce security. The instantiation of the protects relation has already been discussed at length in the previous section. We therefore focus on implementation and security extensions to the language.

Similar to other principals, a computation principal needs to prove its identity—that is, authenticated—before it can access resources. This happens outside the logic but is still an important detail in the implementation. For computation principals, this really means how do we know that the computation principal in the logic is actually the computation executing. The answer depends on the kind of the computation principal (e.g., trusted code or TEE) and the system in which it is executing.

In many cases, the source code is known beforehand. For example, if the computation is running on a trusted machine then a local check of the source code is sufficient. In the case of smart contracts executing on public blockchains, the source code is public, and thus the computation is known. In secure multiparty computation, garbled circuits, encoding the computation, are public, and are thus known to the parties. In all these cases, inspection of the computation is a form of authentication.

Authenticating computations is non-trivial for remote code execution, and may require some form of code attestation. In such scenarios, TEEs offer better integrity guarantees; authenticating TEEs reduces to remote code attestation. For example, in the case of Intel SGX enclaves, measurement of the enclave indicated in the attestation report is used. Coal computation principals model not only TEEs but also capture the information needed for authenticating TEEs: computation principal $\mathsf{code}\{\mu T :: \mathsf{Comp}.(e : \tau)\}$ models a TEE that executes the code e with T corresponding to the hash of the computation e.

Realizing Coal Abstractions. Our design ensures that the computation expressions and operations on them are meaningful, and that they correspond to concrete real world instances. Computation expressions can be implemented as Intel SGX enclaves. We describe one possible implementation using Intel SGX SDK, the standard framework to build enclave applications [18]. The SDK uses libraries to create a runtime instance of an enclave. The computation expression $\mu T :: \mathsf{Comp}.(e : \tau)$ corresponds to the enclave library. Recall that the annotation t is the identity of TEE, and it can be referred inside the expression e. This is similar to the SGX enclave computation referring its identity during attestation operations.[4] The SDK also provides the function sequence, sgx_create_enclave followed by ecall and destroy_enclave, to create, invoke and tear down an enclave. The expression $\mathsf{exec}(\mu T :: \mathsf{Comp}.(e : \tau))$ abstracts the entire enclave management into one single operation.

Security Extensions. It would be useful to enforce strong security guarantees such as noninterference of principals. One could extend Coal's type system to track the authority of the context in which signed terms and propositions are constructed using the techniques from language-based security

[4]Technically, measurement hash, designated by the value MRENCLAVE, is the identity of an SGX enclave. Instructions EINIT and EREPORT access MRENCLAVE value.

literature [6], [15], [16], [19]. Intuitively, these techniques associate the confidentiality and integrity of the protected values with encryption and signing. Thus, a protected value can be accessed only if the context has appropriate authority, that is, if it has access to the corresponding cryptographic key. Besides being an expressive authorization logic, these extensions could make Coal a secure programming language that can reason about both access and information flow control.

VII. Related Work

We discuss related work other than DCC under three broad categories.

Computations as Principals. *Measured principals* represent running programs [2], [20].[5] The defining characteristic of a measured principal is that it closes the gap between a running program and the program that was analyzed. They enforce authentication-based access control by authenticating the binaries. However, their naming scheme is brittle: minor modifications to the program (e.g., accessing a new resource) results in a new name for the binary, effectively making it a different principal. This is unsuitable for expressing access control polices that rely on the semantics, rather than the syntax, of the programs. Coal addresses this issue by enabling equational reasoning about computation principals (see Equiv-Comp in Figure 6) to check if two programs are equivalent.

It will be interesting to use a combination of measured principals and secure compilation techniques to establish the connection between Coal's computation principals and the binary programs. In that approach, all binaries compiled from equivalent programs get the same name, and thus are more robust to name changes.

Authorization Logics. There are several authorization logics that are based on DCC and its variants. We limit our discussion to logics that specify computations either directly or indirectly in the policies.

DFLATE, a calculus for reasoning and enforcing information flow control policies in distributed enclaves, first coined the term *computation principals* to identify TEEs [16]. Coal takes this further by treating any program as a principal, and establishing the principal equivalence.

Nexus Authorization logic (NAL) is based on CDD, a variant of DCC that eschews principal lattice and offers a simpler protection relation [1]. NAL explicitly supports programs as principals, and uses an analytic based approach in which properties of programs (and thus principals) are established using techniques such as program analysis and proof carrying code. However, it uses a hash-based naming scheme for programs. Also, it cannot express policies that reason about programs without relying on external program analyzers.

[5]Measured principals were first introduced in the late 1980's to build secure distributed systems. However, the authors could not find the first scholarly work that introduced them.

Aura is a language for access control and is based on CDD [6]. Both Coal and Aura share similar ideas such as unified language for programming and access control, and using dependent types for expressive access control policies. However, the goals are orthogonal. Aura focuses on security trail audit whereas Coal focuses on the expressive power of the authorization logic. Aura supports only named principals. It uses dependent types to specify expressive predicates whereas Coal uses dependent types to express computation principals.

Both Coal and Aura do not mix proofs and programs. Aura uses a pf monad for proofs, effectively separating them from computations. Coal relies on its stratified type system (Prop and $\star$ kinds) to enforce the separation.

Access Control Systems. Sirer *et al.* describe *logical attestation*—an authorization architecture based on NAL—in which programs act as principals and access control policies can be specified in terms of the run time behavior of the programs [21]. Their work has limitations similar to those of NAL.

Sadeghi *et al.* introduce *property-based attestation*, a privacy preserving access control mechanism, in which the name of a principal (e.g., software application) is computed from its properties [22]. The naming scheme is brittle and has the same drawbacks as that of measured principals. However, their cryptographic enforcement complements our work; it will be interesting to incorporate their techniques in Coal.

VIII. Conclusion

We introduced Coal, an expressive authorization logic that uses computation principals to specify fine grained access control policies involving computations. Computation principals, like any other principals, can be used as the subjects of the access control policies, however, Coal treats them specially. The representation of a computation principal exposes its structure and semantics, making equational reasoning possible. This avoids the brittleness of hash-based naming schemes for computations. Using Coal, we can express trust in computations conditioned on their semantic properties.

References

[1] 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. [Online]. Available: http://doi.acm.org/10.1145/1952982.1952990

[2] F. B. Schneider, https://www.cs.cornell.edu/fbs/publications/chptr.isolate.measPrins.pdf, measured Principals and Gating Functions.

[3] M. Abadi, M. Burrows, B. W. Lampson, and G. D. Plotkin, "A calculus for access control in distributed systems," *ACM Trans. on Programming Languages and Systems*, vol. 15, no. 4, pp. 706–734, 1993.

[4] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke, "A core calculus of dependency," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '99. ACM, 1999.

[5] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," ser. USENIX, 1993.

[6] 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.

[7] M. H. Sørensen and P. Urzyczyn, *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)*. USA: Elsevier Science Inc., 2006.

[8] F. Pfenning and P. Lee, "Leap: A language with eval and polymorphism," 03 1989, pp. 345–359.
[9] H. H. Barendregt, "Self-interpretations in lambda calculus." *J. Funct. Program.*, vol. 1, pp. 229–233, 01 1991.
[10] T. Rendel, K. Ostermann, and C. Hofer, "Typed self-representation," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 293–303.
[11] M. Brown and J. Palsberg, "Self-representation in girard's system u," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 471–484.
[12] P. Nguyen, A. Silence, D. Darais, and J. P. Near, "Duetsgx: Differential privacy with secure hardware," in *Workshop on Theory and Practice of Differential Privacy*, ser. TPDP '20, 2029.
[13] S. Tse and S. Zdancewic, "Translating dependency into parametricity," in *9th ACM SIGPLAN Int'l Conf. on Functional Programming*, 2004, pp. 115–125. [Online]. Available: http://doi.acm.org/10.1145/1016850.1016868
[14] 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.
[15] O. Arden and A. C. Myers, "A calculus for flow-limited authorization," in *29th IEEE Symp. on Computer Security Foundations (CSF)*, Jun. 2016, pp. 135–147. [Online]. Available: http://www.cs.cornell.edu/andru/papers/flac
[16] A. Gollamudi, S. Chong, and O. Arden, "Information flow control for distributed trusted execution environments," in *Proceedings of the 32nd IEEE Computer Security Foundations Symposium*. Hoboken, NJ, USA: IEEE Press, Jun. 2019.
[17] M. Abadi, "Variations in access control logic," in *Deontic Logic in Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.
[18] Intel, "Intel software guard extensions sdk," https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions/sdk.html, 2020.
[19] J. A. Vaughan, "Auraconf: A unified approach to authorization and confidentiality," in *Proceedings of the 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 2011, p. 45–58.
[20] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson, "The digital distributed system security architecture." National Institute of Standards and Technology, January 1989.
[21] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider, "Logical attestation: An authorization architecture for trustworthy computing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: Association for Computing Machinery, 2011.
[22] A.-R. Sadeghi, C. Stüble, and M. Winandy, "Property-based tpm virtualization," in *Information Security*, T.-C. Wu, C.-L. Lei, V. Rijmen, and D.-T. Lee, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.
[23] Intel, "Intel software guard extensions (Intel SGX) programming reference," https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, 2014.
[24] ARM, "ARM security technology — building a secure system using TrustZone technology."
[25] AMD, "Amd secure encrypted virtualization (sev)," https://developer.amd.com/sev/.
[26] Clarity, section "Smart Contracts as Principals". [Online]. Available: https://github.com/clarity-lang/reference

APPENDIX A
COMPUTATION PRINCIPALS IN PRACTICE

Trusted Execution Environments (TEE) such as Intel SGX [23], ARM TrustZone [24] and AMD Secure Virtualization [25] gave rise to a number of interesting trust policies that directly depend on the computation rather than the source of the computation. For example, it is now possible to express trust directly in the code downloaded from an external source because it can be run unmodified inside an enclave — a protected hardware container that offers code attestation and isolated execution. The user can now be confident that the code downloaded has run unmodified owing to the code attestation feature provided by these secure hardware architectures. TEEs thus provide the necessary infrastructure to place trust directly in the code.

Smart contracts, public programs having unique identity and residing on the blockchain, are yet another example of computation principals. Decentralized applications use smart contracts as principals to enforce authorization checks [26]. Smart contracts are identified by a combination of the publishing address and contract's name. As such the policies cannot explore the structure of the contract. On the other hand, using code of the contract itself as a principal will enable fine-grained policies (e.g., contracts may restrict using delegatecall on unsafe contracts) that explore the structure of the code.

In the UNIX family of operating systems, the SUID bit of a process is used to escalate its privileges. Typically, the root administrator sets the bit on trusted programs, and is thus errorprone. However, using programs as computation principals enables expressive access control policies such as the admin trusts only those programs that do not invoke execve. Other examples of computation principals include mobile code; garbled circuits — cryptographic circuits that compute a function — used in secure multiparty computation protocols; and eBPF filters that analyze the network traffic.

APPENDIX B
EQUIVALENCE RULES

Rule WT-KEQUIV in Figure 8 says that when two kinds κ and κ' are equivalent and κ is well-formed, then κ' is also well-formed.

$$\text{WT-KEQUIV}\quad \frac{\Gamma \vdash \kappa\ \mathsf{ok} \qquad \Gamma \vdash \kappa \equiv \kappa'}{\Gamma \vdash \kappa'\ \mathsf{ok}}$$

Fig. 8: Kind equivalence

APPENDIX C
HAND-OFF

We obtain a hand-off proof similar to that of DCC if the type B says $A \overset{\kappa}{\Longrightarrow} B$ protects the type $A \overset{\kappa}{\Longrightarrow} B$.

Proposition 3 (Hand-off proof-2)**.** *For all well-kinded principals A and B let $\emptyset \vdash A \overset{\kappa}{\Longrightarrow} B$ protects B says $A \overset{\kappa}{\Longrightarrow} B$. Then, for all A and B, there exists an expression e such that We have $\emptyset \vdash e : B$ says $A \overset{\kappa}{\Longrightarrow} B \to A \overset{\kappa}{\Longrightarrow} B$.*

Proof. The proof term is $\lambda x : B$ says $A \overset{\kappa}{\Longrightarrow} B$. bind $y = x$ in y. The key insight here is that the premise $\emptyset \vdash A \overset{\kappa}{\Longrightarrow} B$ protects B says $A \overset{\kappa}{\Longrightarrow} B$ ensures that bind $y = x$ in y is well typed (see T-BIND). □

APPENDIX D
TYPE SOUNDNESS

Lemma 5 (Variable Substitution in Kinds)**.** *If* $\Gamma, x : \tau, \Gamma' \vdash \kappa \ \mathsf{ok}$ *and* $\Gamma \vdash e : \tau$ *then* $\Gamma, [x \mapsto e]\Gamma' \vdash \kappa[x \mapsto e] \ \mathsf{ok}$.

Proof. Proof is by induction on the structure of the kinds. □

Lemma 6 (TypeVar Substitution in Kinds)**.** *If* $\Gamma, X :: \kappa, \Gamma' \vdash \kappa' \ \mathsf{ok}$ *and* $\Gamma \vdash_{kind} \tau :: \kappa$ *then* $\Gamma, [X \mapsto \tau]\Gamma' \vdash \kappa[X \mapsto \tau] \ \mathsf{ok}$.

Proof. Proof is by induction on the structure of the kinds. □

Lemma 7 (Variable Substitution in Types)**.** *If* $\Gamma, x : \tau', \Gamma' \vdash_{kind} \tau :: \kappa$ *and* $\Gamma \vdash e : \tau'$ *then* $\Gamma, [x \mapsto e]\Gamma' \vdash_{kind} \tau[x \mapsto e] :: \kappa[x \mapsto e]$.

Proof. Proof is by induction on the structure of the types. □

Lemma 8 (TypeVar Substitution in Types)**.** *If* $\Gamma, X :: \kappa', \Gamma' \vdash_{kind} \tau :: \kappa$ *and* $\Gamma \vdash_{kind} \tau' :: \kappa'$ *then* $\Gamma, [X \mapsto \tau']\Gamma' \vdash_{kind} \tau[X \mapsto \tau'] :: \kappa$.

Proof. Proof is by induction on the structure of the types. □

Lemma 9 (Variable Substitution in Terms)**.** *If* $\Gamma, x : \tau', \Gamma' \vdash e : \tau$ *and* $\Gamma \vdash v : \tau'$ *then* $\Gamma, [x \mapsto v]\Gamma' \vdash e[x \mapsto v] : \tau[x \mapsto v]$.

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

Lemma 10 (TypeVar Substitution in Terms)**.** *If* $\Gamma, X :: \kappa, \Gamma' \vdash e : \tau$ *and* $\Gamma \vdash_{kind} \tau' :: \kappa$ *then* $\Gamma, [X \mapsto \tau']\Gamma' \vdash e[X \mapsto \tau'] : \tau[X \mapsto \tau']$.

Proof. Proof is by induction on the structure of the terms. □

Lemma 11 (Equivalent Kinds)**.** *If* $\Gamma \vdash \kappa \equiv \kappa'$ *then* $\Gamma \vdash \kappa \ \mathsf{ok}$ *and* $\Gamma \vdash \kappa' \ \mathsf{ok}$.

Proof. Immediate from the inference rule WT-KEQUIV. □

Lemma 12 (Well-kinded types have well-formed kinds)**.** *For all* Γ*, if* $\Gamma \vdash_{kind} \tau :: \kappa$ *then* $\Gamma \vdash \kappa \ \mathsf{ok}$.

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

Lemma 13 (Well-typed terms have well-kinded types)**.** *For all* Γ*, if* $\Gamma \vdash e : \tau$ *then* $\Gamma \vdash_{kind} \tau :: \kappa$ *for some kind* κ *such that* $\Gamma \vdash \kappa \ \mathsf{ok}$.

Proof. Applying well-kinded types have well-formed kinds (Lemma 12) to $\Gamma \vdash_{kind} \tau :: \kappa$, we have $\Gamma \vdash \kappa \ \mathsf{ok}$. It remains to prove that if $\Gamma \vdash e : \tau$ then $\Gamma \vdash_{kind} \tau :: \kappa$. Proof is by induction on the derivation of the typing judgment. □

Lemma 14 (Closed Types)**.** *If* $\Gamma \vdash_{kind} \tau :: \kappa$ *then for all* $x \in \tau$, $x \in \mathtt{dom}(\Gamma)$.

Proof. The proof is by contradiction. Let $x \in \tau$ such that $x \notin \mathtt{dom}(\Gamma)$. Assume $\Gamma \vdash_{kind} \tau :: \kappa$. Consider the case K-COMP-PRIN. That is $\tau = \mathsf{code}\{e\}$. Without loss of generality, let $x \in e$. Then we are assuming that $\Gamma \vdash_{kind} \mathsf{code}\{e\} :: \kappa$. Thus, we have $\Gamma \vdash e : \tau$.

This is a contradiction since $x \notin \mathtt{dom}(\Gamma)$. Thus either $x \in \mathtt{dom}(\Gamma)$ or $x \notin e$. Both imply that τ is closed under $\mathtt{dom}(\Gamma)$. Hence proved. □

Lemma 15 (Type Preservation)**.** *If* $\Gamma \vdash e : \tau$ *and* $e \twoheadrightarrow e'$ *then* $\Gamma \vdash e' : \tau$.

Proof. The proof is a straight forward induction on the derivation of the small-step relation.

E-App: This is the interesting case. Let $e = \lambda x : \tau_1.\ e_2\ v$. Given $\Gamma \vdash \lambda x : \tau_1.\ e_2\ v : \tau_2[x \mapsto v]$ and $\lambda x : \tau_1.\ e_2\ v \twoheadrightarrow e_2[x \mapsto v]$. We have to prove

$$\Gamma \vdash e_2[x \mapsto v] : \tau_2[x \mapsto v]$$

Inverting the typing rule T-APP, we have $\Gamma \vdash \lambda x : \tau_1.\ e_2 : (x{:}\tau_1) \to \tau_2$ and $\Gamma \vdash v : \tau_1$. From T-ABS, we have $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$. Invoking variable substitution lemma (Lemma 9), we have $\Gamma \vdash e_2[x \mapsto v] : \tau_2[x \mapsto v]$.

E-TApp: Let $e = \Lambda X :: \kappa.\ e'\ \tau$. Given $\Gamma \vdash \Lambda X :: \kappa.\ e'\ \tau : \tau'[X \mapsto \tau]$. We have to prove that

$$\Gamma \vdash e'[X \mapsto \tau] : \tau'[X \mapsto \tau]$$

Inverting T-TAPP, we have $\Gamma, X :: \kappa \vdash e' : \tau'$ and $\Gamma \vdash_{kind} \tau :: \kappa$. Applying type substitution in terms (Lemma 10) we get the required judgment.

E-ProjI: Let $e = \mathsf{proj}_i\ \langle v_1, v_2 \rangle$. Given $\Gamma \vdash \mathsf{proj}_i\ \langle v_1, v_2 \rangle : \tau_i$. From T-PAIR, we have that, $\Gamma \vdash v_i : \tau_i$. Following E-PROJI, we have $e' = v_i$. Thus we have the required typing judgment.

E-Case: Let $e = \mathsf{case}\ \mathsf{inj}_i\ v\ \mathsf{of}\ \mathsf{inj}_1(x).\ e_1 \mid \mathsf{inj}_2(x).\ e_2$. Given $\Gamma \vdash \mathsf{case}\ \mathsf{inj}_i\ v\ \mathsf{of}\ \mathsf{inj}_1(x).\ e_1 \mid \mathsf{inj}_2(x).\ e_2 : \tau$. We have that $e' = e_i[x \mapsto v]$. We have to prove

$$\Gamma \vdash e_i[x \mapsto v] : \tau$$

Inverting T-CASE, we have

$$\Gamma \vdash \mathsf{inj}_i\ v : \tau_1 + \tau_2 \tag{1}$$
$$\Gamma, x : \tau_i \vdash e_i : \tau \tag{2}$$
$$\Gamma \vdash \tau : \kappa \tag{3}$$

From (1) and T-INJI, we have that $\Gamma \vdash v : \tau_i$. Applying variable substitution in terms (Lemma 9) to (2), we have $\Gamma \vdash e_i[x \mapsto v] : \tau[x \mapsto v]$. Applying Lemma 14 to (3), we have that $x \notin \tau$. That implies, $\tau[x \mapsto v] = \tau$. Thus we have the required typing judgment $\Gamma \vdash e_i[x \mapsto v] : \tau$.

E-Bind: Let $e = \mathsf{bind}\ x = \eta_\ell\ v\ \mathsf{in}\ e$. Given $\Gamma \vdash \mathsf{bind}\ x = \eta_\ell\ v\ \mathsf{in}\ e : \tau'$ and following E-BINDM, $\mathsf{bind}\ x = \eta_\ell\ v\ \mathsf{in}\ e' \twoheadrightarrow e'[x \mapsto v]$. We have to prove that

$$\Gamma \vdash e[x \mapsto v] : \tau'$$

Inverting T-BINDM, we have

$$\Gamma \vdash \eta_\ell\ v : \ell\ \mathsf{says}\ \tau \tag{4}$$
$$\Gamma, x : \tau \vdash e : \tau' \tag{5}$$
$$\Gamma \vdash_{kind} \tau' :: \kappa \tag{6}$$

From (4), we have $\Gamma \vdash v : \tau$. Applying variable substitution (Lemma 9) to (5), we have $\Gamma \vdash e[x \mapsto v] : \tau'[x \mapsto v]$. Note that the τ' in (6) is free of x. We get this by applying Lemma 14 (closed types) to (6), and so, $\tau'[x \mapsto v] = \tau'$. Thus we have the required typing judgment.

E-Exec: Similar to the case E-APP.

E-Context: Applying I.H to premise yields the result.

□

Lemma 16 (Progress)**.** *If* $\emptyset \vdash e : \tau$ *and* $\emptyset;\emptyset \vdash_{kind} \tau :: \star$ *then* e *is either a value or exists* e' *such that* $e \twoheadrightarrow e'$.

Proof. The proof is a straight forward induction on the derivation of the typing judgment.

T-Unit: Given $e = ()$ which is already a value.

T-Var: Given $e = x$ and $\emptyset;\emptyset \vdash x : \tau$. This is impossible.

T-Pair Given $e = \langle e_1, e_2 \rangle$ and $\emptyset;\emptyset \vdash e : \tau$. Inverting using rule T-PAIR, we have $\emptyset;\emptyset \vdash e_1 : \tau_1$ and $\emptyset;\emptyset \vdash e_2 : \tau_2$. Applying I.H to the premises we have that either e_1, e_2 take a step or both e_1 and e_2 are values. If e_1 and e_2 are values, then $\langle e_1, e_2 \rangle$.

T-InjI: Given $e = \mathsf{inj}_i\ e'$ and $\emptyset;\emptyset \vdash e : \tau$ for $\tau = \tau_1 + \tau_2$. Inverting using rule T-INJI, we have $\emptyset;\emptyset \vdash e' : \tau_i$. Applying I.H to the premises we have that either e' takes a step or e' is a value. If e' are values, then $\mathsf{inj}_i\ e'$ is a value.

T-ProjI: Given $e = \mathsf{proj}_i\ \langle e_1, e_2 \rangle$ and $\emptyset;\emptyset \vdash e : \tau$ for $\tau = \tau_1 \times \tau_2$. Inverting using rule T-PROJI, we have $\emptyset;\emptyset \vdash e_j : \tau_j$ for all $j \in \{1, 2\}$.
Applying I.H to the premises we have that either e_j takes a step or both e_1 and e_2 are values. If e_1 and e_2 are values, then from E-PROJI we have that $\mathsf{proj}_i\ \langle e_1, e_2 \rangle \twoheadrightarrow e_i$ is a value.

T-Case: Given $e = \mathsf{case}\ e'\ \mathsf{of}\ \mathsf{inj}_1(x).\ e_1 \mid \mathsf{inj}_2(x).\ e_2$. Also given that $\emptyset;\emptyset \vdash e : \tau$. Inverting from T-CASE, we have that $\emptyset;\emptyset \vdash \mathsf{inj}_i\ e' : \tau_1 + \tau_2$ and $\emptyset, x : \tau_i;\emptyset \vdash e_i : \tau$. We also have that if $\emptyset;\emptyset \vdash_{kind} \tau :: \star$ then $\emptyset;\emptyset \vdash_{kind} \tau_1 + \tau_2 :: \star$. Thus applying I.H. to e we have that either e' takes a step or is already a value. In the latter, e takes a step following E-CASE.

T-Abs: Given $e = \lambda x : \tau.\ e$ which is already a value.

T-TAbs: Given $e = \Lambda x : \tau.\ e$ which is already a value.

T-App: Given $e = e_1\ e_2$. Also given that $\emptyset;\emptyset \vdash e_1\ e_2 : \tau$ and $\emptyset;\emptyset \vdash_{kind} \tau :: \star$. Inverting the rule T-APP, we have that $\emptyset;\emptyset \vdash e_1 : \tau_1 \to \tau_2$ such that $\emptyset;\emptyset \vdash \tau_2 : \star$. Note that $\emptyset;\emptyset \vdash \tau_1 : \mathsf{Prop}$ is a valid case in which case we cannot apply I.H to $\emptyset;\emptyset \vdash e_2 : \tau_1$. However, applying I.H to the premise (abstraction) we have that either e_1 takes a step or is already a value. In the latter case, the entire expression takes a step using E-APP regardless of whether e_2 is already a value (because of CBN semantics).

T-TApp Given $e = e'\ \tau'$ and $\emptyset;\emptyset \vdash e : \tau$ such that $\emptyset;\emptyset \vdash_{kind} \tau :: \star$. Inverting T-TAPP, we have that $\emptyset;\emptyset \vdash e' : \forall X :: \kappa.\ \tau''$. Applying Lemma 13, we have that $\emptyset;\emptyset \vdash_{kind} \forall X :: \kappa.\ \tau'' :: \kappa'$. From K-TABS, wwe have $\kappa' = \star$. Thus applying I.H to $\emptyset;\emptyset \vdash e' : \forall X :: \kappa.\ \tau''$, we have that either e' takes a step or is already a value. If it is a value, then it takes a step as per E-TAPP.

T-UnitM: Given $e = \eta_\ell\ e$. Applying I.H to the premise of T-UNITM, we have that e can make progress or e is some value v. In the latter case, $\eta_\ell\ v$ is also a value.

T-Sign: Given $e = \mathsf{sign}(\ell, P)$ which is already a value.

T-Bind: Given $e = \mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2$ and $\emptyset;\emptyset \vdash \mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2 : \tau$. Inverting the rule T-BIND, we have that $\emptyset;\emptyset \vdash e_1 : \ell\ \mathsf{says}\ \tau'$. Also the premises imply that if $\emptyset, x : \tau';\emptyset \vdash_{kind} \tau :: \star$ then $\emptyset;\emptyset \vdash \ell\ \mathsf{says}\ \tau' : \star$. Applying I.H. to the premises we thus have that either e_1 takes a step or is already a value. In the latter case, rule E-BIND applies.

T-Comp: Given $e = \mu T :: \mathsf{Comp}.(e : \tau)$ which is already a value.

T-Exec: Given $e = \mathsf{exec}(e_1)$ and $\emptyset \vdash e : \tau'[T \mapsto \mathsf{code}\{\mu T :: \mathsf{Comp}.(e' : \tau')\}]$. Inverting T-EXEC, we have that $\emptyset \vdash e_1 : \mathsf{code}\{\mu T :: \mathsf{Comp}.(e' : \tau')\}$. Applying well-types are well-kinded, (Lemma 13), we have that $\emptyset \vdash_{kind} \tau'[T \mapsto \mathsf{code}\{\mu T :: \mathsf{Comp}.(e' : \tau')\}] :: \kappa$. If κ is $\star$, applying I.H, we thus have that either e_1 takes a step or it is a value. If e_1 is a value, $\mathsf{exec}(e_1)$ takes a step as per E-EXEC.

□

APPENDIX E
EXAMPLE POLICIES

A. Open Computation Principals

Example E.1 (Open Computation Principals)**.** Let Γ be a typing context that maps variable s (for size) to an integer. That is, $\Gamma = s \mapsto \mathsf{int}.$ and $\mathsf{code}\{\mu T :: \mathsf{Comp}.(genprime_\# : \mathsf{prime})\}$ be a well-kinded computation principal under the typing context Γ. That is, $\Gamma \vdash_{kind} \mathsf{code}\{\mu T :: \mathsf{Comp}.(genprime_\# : \mathsf{prime})\} :: \mathsf{Prin}$. Then, the hand-off from A to $\mathsf{code}\{\mu T :: \mathsf{Comp}.(genprime_\# : \mathsf{prime})\}$ is shown below.

$$\Gamma \vdash_{kind} (A\ \mathsf{says}\ (\mathsf{code}\{\mu T :: \mathsf{Comp}.(genprime_\# : \mathsf{prime})\} \overset{\kappa}{\Rightarrow} A)) \to \mathsf{code}\{\mu T :: \mathsf{Comp}.(genprime_\# : \mathsf{int})\} \overset{\kappa}{\Rightarrow} A :: \kappa$$

The proof term is a function that accepts a proof of delegation and a term signed by the computation principal, and outputs a term signed by A.

$$\lambda x : A\ \mathsf{says}\ (\mathsf{code}\{\mu T :: \mathsf{Comp}.(genprime_\# : \mathsf{prime})\} \overset{\star}{\Rightarrow} A).$$
$$\Lambda X :: \star.\ \lambda cp : \mathsf{code}\{\mu T :: \mathsf{Comp}.(genprime_\# : \mathsf{prime})\}\ \mathsf{says}\ X.$$
$$\mathsf{bind}\ z = x\ \mathsf{in}\ (z\ X\ cp)$$

Note that the above term is well-typed under Γ. □