Logical Methods in Computer Science
Volume 00, Number 0, Pages 000–000
S 0000-0000(XX)0000-0

λ-RBAC: PROGRAMMING WITH ROLE-BASED ACCESS CONTROL

RADHA JAGADEESAN, ALAN JEFFREY, CORIN PITCHER, AND JAMES RIELY

CTI, DEPAUL UNIVERSITY
E-mail address: rjagadeesan@cti.depaul.edu

BELL LABS
E-mail address: ajeffrey@bell-labs.com

CTI, DEPAUL UNIVERSITY
E-mail address: cpitcher@cti.depaul.edu

CTI, DEPAUL UNIVERSITY
E-mail address: jriely@cti.depaul.edu

ABSTRACT. We study mechanisms that permit program components to express role constraints on clients, focusing on programmatic security mechanisms, which permit access controls to be expressed, *in situ*, as part of the code realizing basic functionality. In this setting, two questions immediately arise:

- The user of a component faces the issue of safety: is a particular role sufficient to use the component?
- The component designer faces the dual issue of protection: is a particular role demanded in all execution paths of the component?

We provide a formal calculus and static analysis to answer both questions.

1. INTRODUCTION

This paper addresses programmatic security mechanisms as realized in systems such as Java Authentication and Authorization Service (JAAS) and .NET. These systems enable two forms of access control mechanisms[1]. First, they permit *declarative* access control to describe security specifications that are orthogonal and separate from descriptions of functionality, e.g., in an interface I, a declarative access control mechanism could require the caller to possess a minimum set of rights. While conceptually elegant, such specifications do not directly permit the enforcement of access control that is sensitive to the control and dataflow of the code implementing the functionality — consider for example history sensitive security policies that require runtime monitoring of relevant events. Consequently, JAAS and .NET also include *programmatic* mechanisms that permit access

Radha Jagadeesan and Corin Pitcher were supported in part by NSF CyberTrust 0430175.
James Riely was supported in part by NSF CAREER 0347542.

[1] In this paper, we discuss only authorization mechanisms, ignoring the authentication mechanisms that are also part of these infrastructures.

control code to be intertwined with functionality code, e.g., in the code of a component implementing interface I. On the one hand, such programmatic mechanisms permit the direct expression of access control policies. However, the programmatic approach leads to the commingling of the conceptually separate concerns of security and functionality.

There is extensive literature on policy languages to specify and implement policies (e.g., [16, 28, 15, 5, 29, 13] to name but a few). This research studies security policies as separate and orthogonal additions to component code, and is thus focused on declarative security in the parlance of JAAS/.NET.

In contrast, we study programmatic security mechanisms. Our motivation is to *extract* the security guarantees provided by access control code which has been written inline with component code. We address this issue from two viewpoints:

- The user of a component faces the issue of safety: is a particular set of rights sufficient to use the component? (ie. with that set of rights, there is no possible execution path that would fail a security check. Furthermore, any greater set of rights will also be allowed to use the component)
- The component designer faces the dual issue of protection: is a particular set of rights demanded in all execution paths of the component? (ie. every execution path requires that set of rights. Furthermore, any lesser set of rights will not be allowed to use the component)

The main contribution of this paper is separate static analyses to calculate approximations to these two questions. An approximate answer to the first question is a set of rights, perhaps bigger than necessary, that is *sufficient* to use the component. On the other hand, an approximate answer to the second question, is a set of rights, perhaps smaller than what is actually enforced, that is *necessary* to use the component.

1.1. **An overview of our technical contributions.** There is extensive literature on Role-Based Access-Control (RBAC) models including NIST standards for RBAC [26, 12]; see [11] for a textbook survey. The main motivation for RBAC, in software architectures (e.g., [22, 21]) and frameworks such as JAAS/.NET, is that it enables the enforcement of security policies at a granularity demanded by the application. In these examples, RBAC allows permissions to be de-coupled from users: Roles are the unit of administration for users and permissions are assigned to roles. Roles are often arranged in a hierarchy for succinct representation of the mapping of permissions. Component programmers design code in terms of a static collection of roles. When the application is deployed, administrators map the roles defined in the application to users in the particular domain.

In this paper, we study a lambda calculus enriched with primitives for access control, dubbed λ-RBAC. The underlying lambda calculus serves as an abstraction of the ambient programming framework in a real system. We draw inspiration from the programming idioms in JAAS and .NET, to determine the expressiveness required for the access control mechanisms. In a sequence of .NET examples[2], closely based on [18], we give the reader a flavor of the basic programming idioms.

Example 1 ([18])**.** In the .NET Framework CLR, every thread has a `Principal` object that carries its role. This `Principal` object can be viewed as representing the user executing the thread. In programming, it often needs to be determined whether a specific `Principal` object belongs to a familiar role. The code performs checks by making a security call for a `PrincipalPermission` object. The `PrincipalPermission` class denotes the role that a specific principal needs to match. At the time of a security check, the CLR checks whether the role of the `Principal` object of the

[2]In order to minimize the syntactic barrage on the unsuspecting reader, our examples to illustrate the features are drawn solely from the .NET programming domain. At the level of our discussion, there are no real distinctions between JAAS and .NET security services.

caller matches the role of the `PrincipalPermission` object being requested. If the role values of the two objects do not match, an exception is raised. The following code snippet illustrates the issues:

```
PrincipalPermission usrPerm  =
     new PrincipalPermission (null,"Manager");
usrPerm.Demand()
```

If the current thread is associated with a principal that has the the role of manager, the `PrincipalPermission` objects are created and security access is given as required. If the credentials are not valid, a security exception is raised. □

In this vein, the intuitive operation of λ-RBAC is as follows. λ-RBAC program execution takes place in the context of a role, say *r*, which can be viewed concretely as a set of permissions. The set of roles used in a program is static: we do not allow the dynamic creation of roles. λ-RBAC supports run-time operations to create objects (i.e. higher-order functions) that are wrapped with protecting roles. The use of such guarded objects is facilitated by operations that check that the role-context *r* is at least as strong as the guarding role: an exception is raised if the check fails.

The next example illustrates that boolean combinations of roles are permitted in programs. In classical RBAC terms, this is abstracted by a lattice or boolean structure on roles.

Example 2 ([18])**.** The `Union` method of the `PrincipalPermission` class combines multiple `PrincipalPermission` objects. The following code represents a security check that succeeds only if the `Principal` object represents a user in the `CourseAdmin` or `BudgetManager` roles:

```
PrincipalPermission Perm1 =
    new PrincipalPermission (null,"CourseAdmin");
PrincipalPermission Perm2 =
    new PrincipalPermission(null,"BudgetManager');

// Demand at least one of the roles using Union
perm1.Union (perm2).Demand ()
```

Similarly, there is an `Intersect` method to represent a "join" operation in the role lattice. □

In λ-RBAC, we assume that roles form a lattice: abstracting the concrete union/intersection operations of these examples. In the concrete view of a role as a set of permissions, role ordering is given by supersets, ie. a role is stronger than another role if it has more permissions; join of roles corresponds to the union of the sets of permissions and meet of roles corresponds to the intersection of the sets of permissions. Some of our results assume that the lattice is boolean, i.e. the lattice has a negation operation. In the concrete view of the motivating examples, the negation operation is interpreted by set complement with respect to a maximum collection of permissions

Our study is parametric on the underlying role lattice.

The key operation in such programming is *rights modulation*. From a programming viewpoint, it is convenient, indeed sometimes required, for an application to operate under the guise of different users at different times. Rights modulation of course comes in two flavors: rights weakening is overall a safe operation, since the caller chooses to execute with fewer rights. On the other hand, rights amplification is clearly a more dangerous operation. In the .NET framework, rights modulation is achieved via a technique called impersonation.

Example 3. Impersonation of an account is achieved using the account's token, as shown in the following code snippet:

```
WindowsIdentity stIdentity = new WindowsIdentity (StToken);
   // StToken is the token associated with the Windows acct being impersonated
WindowsImpersonationContext stImp = stIdentity.Impersonate();
   // now operating under the new identity
stImp.Undo();  // revert back
```

□

λ-RBAC has combinators to perform scoped rights weakening and amplification.

We demonstrate the expressiveness of λ-RBAC by building a range of useful combinators and a variety of small illustrative examples. We discuss type systems to perform the two analyses alluded to earlier: (a) an analysis to detect and remove unnecessary role-checks in a piece of code for a caller at a sufficiently high role, and (b) an analysis to determine the (maximal) role that is guaranteed to be required by a piece of code. The latter analysis acquires particular value in the presence of rights modulation. For both we prove preservation and progress properties.

1.2. **Related work.** Our paper falls into the broad area of research enlarging the scope of foundational, language-based security methods (see [27, 19, 3] for surveys).

Our work is close in spirit, if not in technical development, to edit automata [16], which use aspects to avoid the explicit intermingling of security and baseline code.

The papers that are most directly relevant to the current paper are those of Braghin, Gorla and Sassone [7] and Compagnoni, Garalda and Gunter [10]. [7] presents the first concurrent calculus with a notion of RBAC, whereas [10]'s language enables privileges depending upon location.

Both these papers start off with a mobile process-based computational model. Both calculi have primitives to activate and deactivate roles: these roles are used to prevent undesired mobility and/or communication, and are similar to the primitives for role restriction and amplification in this paper. The ambient process calculus framework of these papers provides a direct representation of the "sessions" of RBAC— in contrast, our sequential calculus is best thought of as modeling a single session.

[7, 10] develop type systems to provide guarantees about the minimal role required for execution to be successful — our first type system occupies the same conceptual space as this static analysis. However, our second type system that calculates minimum access controls does not seem to have an analogue in these papers.

More globally, our paper has been influenced by the desire to serve loosely as a metalanguage for programming RBAC mechanisms in examples such as the JAAS/.NET frameworks. Thus, our treatment internalizes rights amplification by program combinators and the amplify role constructor in role lattices. In contrast, the above papers use external — i.e. not part of the process language — mechanisms (namely, user policies in [10], and RBAC-schemes in [7]) to enforce control on rights activation. We expect that our ideas can be adapted to the process calculi framework. In future work, we also hope to integrate the powerful bisimulation principles of these papers.

Our paper deals with access control, so the extensive work on information flow, e.g., see [24] for a survey, is not directly relevant. However, we note that rights amplification plays the same role in λ-RBAC that declassification and delimited release [9, 25, 20] plays in the context of information flow; namely that of permitting access that would not have been possible otherwise. In addition, by supporting the internalizing of the ability to amplify code rights into the role lattice, our system permits access control code to actively participate in managing rights amplification.

1.3. **Rest of the paper.** We present the language in Section 2, the type system in Section 3 and illustrate its expressiveness with examples in Section 4. We discuss methods for controlling rights amplification in Section 5. Section 6 provides proofs of the theorems from Section 3.

2. The Language

After a discussion of roles, we present an overview of the language design. The remaining subsections present the formal syntax, evaluation semantics, typing system, and some simple examples.

2.1. **Roles.** The language of roles is built up from *role constructors*. The choice of role constructors is application dependent, but must include the lattice constructors discussed below. Each role constructor, κ, has an associated arity, $\mathit{arity}(\kappa)$. Roles A–E have the form $\kappa(A_1,\ldots,A_n)$.

We require that roles form a boolean lattice; that is, the set of constructors must include the nullary constructors $\mathbf{0}$ and $\mathbf{1}$, binary constructors $\sqcup$ and $\sqcap$ (written infix), and unary constructor * (written postfix). $\mathbf{0}$ is the least element of the role lattice. $\mathbf{1}$ is the greatest element. $\sqcap$ and $\sqcup$ are idempotent, commutative, associative, and mutually distributive meet and join operations respectively. * is the complement operator.

A role may be thought of as a set of permissions. Under this interpretation, $\mathbf{0}$ is the empty set, while $\mathbf{1}$ is the set of all permissions.

The syntax of terms uses *role modifiers*, ρ, which may be of the form $\uparrow A$ or $\downarrow A$. We use role modifiers as functions from roles to roles, with $\rho(\!|A|\!)$ defined as follows:

$$\uparrow A(\!|B|\!) = A \sqcup B \qquad\qquad \downarrow A(\!|B|\!) = A \sqcap B$$

In summary, the syntax of roles is as follows.

$\kappa ::= \mathbf{0} \mid \mathbf{1} \mid \sqcup \mid \sqcap \mid {}^\star \mid \cdots$	Role constructors
A–$E ::= \kappa(A_1,\ldots,A_n)$	Roles
$\rho ::= \uparrow A \mid \downarrow A$	Role modifiers

Throughout the paper, we assume that all roles (and therefore all types) are well-formed, in the sense that role constructors have the correct number of arguments.

The semantics of roles is defined by the relation "$A \doteq B$" stating that A and B are provably equivalent. In addition to any application-specific axioms, we assume the standard axioms of boolean algebra. We say that A *dominates* B (notation $A \geqslant B$) if $A \doteq A \sqcup B$ (equivalently $B \doteq A \sqcap B$) is derivable. Thus we can conclude $\mathbf{1} \geqslant A \sqcup B \geqslant A \geqslant A \sqcap B \geqslant \mathbf{0}$, for any A, B.

The role modifier $\downarrow A$ creates a weaker role (closer to $\mathbf{0}$), thus we refer to it as a *restriction*. Dually, the modifier $\uparrow A$ creates a stronger role (closer to $\mathbf{1}$), and thus we refer to it as an *amplification*. While this ordering follows that of the NIST RBAC standard [12], it is dual to the normal logical reading; it may be helpful to keep in mind that, viewed as a logic, $\mathbf{1}$ is "false", $\mathbf{0}$ is "true", $\sqcup$ is "and", $\sqcap$ is "or" and $\geqslant$ is "implies."

2.2. **Language overview.** Our goal is to capture the essence of role-based systems, where roles are used to regulate the interaction of components of the system. We have chosen to base our language on Moggi's monadic metalanguage because it is simple and well understood, yet rich enough to capture the key concepts. By design, the monadic metalanguage is particularly well suited to studying computational side effects (or simply *effects*), which are central to our work. (We expect that our ideas can be adapted to both process and object calculi.)

The "components" in the monadic metalanguage are terms and the contexts that use them. To protect terms, we introduce guards of the form $\{A\}\,[M]$, which can only be discharged by a context whose role dominates A. The notion of *context role* is formalized in the definition of evaluation, where $A \triangleright M \longrightarrow N$ indicates that context role A is sufficient to reduce M to N. The term $\mathsf{check}\,M$ discharges the guard on M. The evaluation rule allows $A \triangleright \mathsf{check}\,\{B\}\,[M] \longrightarrow [M]$ only if $A \geqslant B$.

The context role may vary during evaluation: given context role A, the term $\rho(M)$ evaluates M with context role $\rho(\!|A|\!)$. Thus, when $\downarrow B(M)$ is evaluated with context role A, M is evaluated with context role $A \sqcap B$. A context may protect itself from a term by placing the use of the term in such a restricted context. (The syntax enforces a stack discipline on role modifiers.) By combining upwards and downwards modifiers, code may assume any role and thus circumvent an intended policy. We address this issue in Section 5.

These constructs are sufficient to allow protection for both terms and contexts: terms can be protected from contexts using guards, and contexts can be protected from terms using (restrictive) role modifiers.

2.3. **Syntax.** Let x, y, z, f, g range over variable names, and let bv range over base values. Our presentation is abstract with respect to base values; we use the types String, Int and Unit (with value unit) in examples. We use the standard encodings of booleans and pairs (see Example 14). The syntax of values and terms are as follows.

$V, U, W ::=$	$M, N, L ::=$	Values; Terms
$bv \mid x$	V	Base Value
$\lambda x.\, M$	$M\, N \mid \mathsf{fix}\, M$	Abstraction
$\{A\}[M]$	$\mathsf{check}\, M$	Guard
$[M]$	$\mathsf{let}\, x = M\, ;\, N$	Computation
	$\rho(M)$	Role Modifier

Notation. In examples, we write $A(M)$ to abbreviate $\downarrow \mathsf{0}(\uparrow A(M))$, which executes M at exactly role A.

The variable x is bound in the value "$\lambda x.\, M$" (with scope M) and in the term "$\mathsf{let}\, x = M\, ;\, N$" (with scope N). If x does not appear free in M, we abbreviate "$\lambda x.\, M$" as "$\lambda\, .\, M$". Similarly, if x does not appear free in N, we abbreviate "$\mathsf{let}\, x = M\, ;\, N$" as "$M\, ;\, N$". We identify syntax up to renaming of bound variables and write $N\{x := M\}$ for the capture-avoiding substitution of M for x in N. □

In the presentation of the syntax above, we have paired the constructors on values on the left with the destructors on computations on the right. For example, the monadic metalanguage distinguishes 2 from [2] and [1+1]: the former is an integer, whereas the latter are computations that, when bound, produce an integer. The computation value $[M]$ must be discharged in a binding context — see the reduction rule for let, below. Similarly, the function value $\lambda x.\, M$ must be discharged by application; in the reduction semantics that follows, evaluation proceeds in an application till the term in function position reduces to a lambda abstraction. $\{A\}[M]$ constructs a guarded value; the associated destructor is check .

The monadic metalanguage distinguishes computations from the values they produce and treats computations as first class entities. (Any term may be treated as a value via the unit constructor $[M]$.) Both application and the let construct result in computations; however, the way that they handle their arguments is different. The application "$(\lambda x.\, N)\ [M]$" results in $N\{x := [M]\}$, whereas the binding "$\mathsf{let}\, x = [M]\ ;\, M$" results in $N\{x := M\}$.

2.4. **Evaluation and role error.** The small-step evaluation relation $A \triangleright M \to M'$ is defined inductively by the following reduction and context rules.

$$\frac{}{A \rhd (\lambda x.\, M)\, N \to M\{x := N\}}\ \text{(R-APP)} \qquad \frac{A \rhd M \to M'}{A \rhd M\, N \to M'\, N}\ \text{(C-APP)}$$

$$\frac{}{A \rhd \mathsf{fix}\, (\lambda x.\, M) \to M\{x := \mathsf{fix}\, (\lambda x.\, M)\}}\ \text{(R-FIX)} \qquad \frac{A \rhd M \to M'}{A \rhd \mathsf{fix}\, M \to \mathsf{fix}\, M'}\ \text{(C-FIX)}$$

$$\frac{}{A \rhd \mathsf{check}\ \{B\}\, [M] \to [M]}\ A \geqslant B\ \text{(R-CHK)} \qquad \frac{A \rhd M \to M'}{A \rhd \mathsf{check}\, M \to \mathsf{check}\, M'}\ \text{(C-CHK)}$$

$$\frac{}{A \rhd \mathsf{let}\, x = [M]\, ;\, N \to N\{x := M\}}\ \text{(R-BIND)} \qquad \frac{A \rhd M \to M'}{A \rhd \mathsf{let}\, x = M\, ;\, N \to \mathsf{let}\, x = M'\, ;\, N}\ \text{(C-BIND)}$$

$$\frac{}{A \rhd \rho\,(V) \to V}\ \text{(R-MOD)} \qquad \frac{\rho\,(\!|A|\!) \rhd M \to M'}{A \rhd \rho\,(M) \to \rho\,(M')}\ \text{(C-MOD)}$$

The rules R/C-APP for application, R/C-FIX for fixed points and R/C-BIND for let are standard. R-CHK ensures that the context role is sufficient before discharging the relevant guard. C-MOD modifies the context role until the relevant term is reduced to a value, at which point R-MOD discards the modifier.

The evaluation semantics is designed to ensure a role-monotonicity property. Increasing the available role-context cannot invalidate transitions, it can only enable more evolution.

Lemma 4. *If $B \rhd M \to M'$ and $A \geqslant B$ then $A \rhd M \to M'$.* □

Proof. (Sketch) The context role is used only in R-CHK. Result follows by induction on the evaluation judgement. □

Via a series of consecutive small steps, the final value for the program can be determined. Successful termination is written $A \rhd M \twoheadrightarrow V$ which indicates that A is authorized to run the program M to completion, with result V. Viewed as a role-indexed relation on terms, $\twoheadrightarrow$ is reflexive and transitive.

Definition 5. (a) M_0 *evaluates to* M_n at A (notation $A \rhd M_0 \twoheadrightarrow M_n$) if there exist terms M_i such that $A \rhd M_i \to M_{i+1}$, for all i ($0 \leq i \leq n-1$). (b) M *diverges* at A (notation $A \rhd M \to^{\omega}$) if there exist terms M_i such that $A \rhd M_i \to M_{i+1}$, for all $i \in \mathbb{N}$. □

Evaluation can fail because a term diverges, because a destructor is given a value of the wrong shape, or because an inadequate role is provided at some point in the computation. We refer to the latter as a *role error* (notation $A \rhd M \lightning \mathsf{err}$), defined inductively as follows.

$$\frac{}{A \rhd \mathsf{check}\ \{B\}\, [M] \lightning \mathsf{err}}\ A \not\geqslant B$$

$$\frac{A \rhd M \lightning \mathsf{err}}{A \rhd M\, N \lightning \mathsf{err}} \qquad \frac{A \rhd M \lightning \mathsf{err}}{A \rhd \mathsf{fix}\, M \lightning \mathsf{err}} \qquad \frac{A \rhd M \lightning \mathsf{err}}{A \rhd \mathsf{let}\, x = M\, ;\, N \lightning \mathsf{err}} \qquad \frac{A \rhd M \lightning \mathsf{err}}{A \rhd \mathsf{check}\, M \lightning \mathsf{err}} \qquad \frac{\rho\,(\!|A|\!) \rhd M \lightning \mathsf{err}}{A \rhd \rho\,(M) \lightning \mathsf{err}}$$

Example 6. Recall from Section 2.3 that $B(M)$ abbreviates $\downarrow\mathbf{0}(\uparrow B(M))$, and define test<B> as follows[3].

$$\texttt{test<B>} \triangleq \textsf{check}\ \{\texttt{B}\}[\texttt{unit}]$$

test<B> is a computation that requires context role B to evaluate. For example, $\downarrow\texttt{B}^\star(\texttt{test<B>})$ produces a role error in any context, since $\downarrow\texttt{B}^\star$ restricts any role-context to the negation of the role B. □

Example 7. We now illustrate how terms can provide roles for themselves. Consider the following guarded function:

$$\texttt{from<A,B>} \triangleq \{\texttt{A}\}[\lambda\texttt{y}.\,\texttt{B}(\texttt{y})]$$

from<A,B> is a guarded value that may only be discharged by A, resulting in a function that runs any computation at B. Let test<B> $\triangleq$ **check** {B}[unit]. No matter what the relationship is between A and B, the following evaluation succeeds:

$$\texttt{A} \triangleright \textsf{let}\ \texttt{z} = \textsf{check}\ \texttt{from<A,B>};\ \texttt{z}\ \texttt{test<B>} \twoheadrightarrow \texttt{B}(\texttt{test<B>}) \twoheadrightarrow [\texttt{unit}]$$

from<A,B> is far too powerful to be useful. After the A-guard is discharged, the resulting function will run *any* code at role B. One can provide specific code, of course, as in $\lambda\texttt{y}.\,\texttt{B}(\texttt{M})$. Such functions are inherently dangerous and therefore it is desirable constrain the way in which such functions are created. The essential idea is to attach suitable checks to a function such as $\lambda\texttt{g}.\,\lambda\texttt{y}.\,\texttt{B}(\texttt{g}\,\texttt{y})$, which takes a non-privileged function and runs it under B. There are a number of subtleties to consider in providing a general purpose infrastructure to create terms with rights amplification. When should the guard be checked? What functions should be allowed to run, and in what context? In Example 21, we discuss the treatment of these issues using the Domain and Type Enforcement access control mechanism. □

3. TYPING

We present two typing systems that control role errors in addition to shape errors.

The first typing system determines a context role *sufficient* to avoid role errors; that is, with this role, there is no possible execution path that causes a role error. This system enables the removal of unnecessary role-checks in a piece of code for a caller at a sufficiently high role.

The second system determines a context role *necessary* to avoid role errors; that is, any role that does not dominate this role will cause every execution path to result in a role error. Stated differently, the second system calculates the role that is checked and tested on every execution path and thus determines the amount of protection that is enforced by the callee.

Technically, the two systems differ primarily in their notions of subtyping. In the absence of subtyping, the typing system determines a context role that is both necessary and sufficient to execute a term without role errors.

Because it clearly indicates the point at which computation is performed, the monadic metalanguage is attractive for reasoning about security properties, which we understand as computational effects. The type $[T]$ is the type of computations of type T. We extend the computation type $[T]$ to include an effect that indicates the guards that are discharged during evaluation of a term. Thus the term check $\{A\}[1+1]$ has type $\langle A\rangle[\mathsf{Int}]$ — this type indicates that the reduction of the term to a value (at type Int) requires A. Guarded values inhabit types of the form $\{A\}[T]$ — this type indicates the protection of A around an underlying value at type T. These may be discharged with a check, resulting in a term inhabiting the computation type $\langle A\rangle[T]$.

[3]We do not address parametricity here; the brackets in the names test<B> and from<A,B> are merely suggestive.

The syntax of types is given below, with the constructors and destructors at each type recalled from Section 2.3.

T,S ::=	V,U,W ::=	M,N,L ::=	Types; Values; Terms
Base	$bv \mid x$	V	Base Value
$T \to S$	$\lambda x.M$	$M\,N \mid \mathsf{fix}\,M$	Abstraction
$\{A\}[T]$	$\{A\}[M]$	$\mathsf{check}\,M$	Guard
$\langle A\rangle[T]$	$[M]$	$\mathsf{let}\,x = M; N$	Computation
		$\rho(M)$	Role Modifier

3.1. **Subtyping.** The judgments of the subtyping and typing relations are indexed by α which ranges over $\{1,2\}$. The subtyping relation for $\langle A\rangle[T]$ reflects the difference between the two type systems.

If role A suffices to enable a term to evaluate without role errors, then any higher role context also avoids role errors (using Lemma 4). This explains the subtyping rule for the first type system — in particular, $\vdash_1 \langle A\rangle[T] <: \langle \mathbf{1}\rangle[T]$, reflecting the fact that the top role is sufficient to run any computation.

On the other hand, if a role A of the role-context is checked and tested on every execution path of a term, then so is any smaller role. This explains the subtyping rule for the first type system — in particular, $\vdash_2 \langle A\rangle[T] <: \langle \mathbf{0}\rangle[T]$, reflecting the fact that the bottom role is vacuously checked in any computation.

$$\frac{}{\vdash_\alpha \mathsf{Base} <: \mathsf{Base}} \qquad \frac{\vdash_\alpha T <: T'}{\vdash_\alpha \{A\}[T] <: \{A'\}[T']} \begin{array}{l}\text{if } \alpha = 1 \text{ then } A' \geqslant A \\ \text{if } \alpha = 2 \text{ then } A \geqslant A'\end{array}$$

$$\frac{\vdash_\alpha T' <: T \quad \vdash_\alpha S <: S'}{\vdash_\alpha T \to S <: T' \to S'} \qquad \frac{\vdash_\alpha T <: T'}{\vdash_\alpha \langle A\rangle[T] <: \langle A'\rangle[T']} \begin{array}{l}\text{if } \alpha = 1 \text{ then } A' \geqslant A \\ \text{if } \alpha = 2 \text{ then } A \geqslant A'\end{array}$$

Lemma 8. *The relations $\vdash_\alpha T <: S$ are reflexive and transitive.* □

3.2. **Type systems.** Typing is defined using environments. An *environment*,

$$\Gamma ::= x_1 : T_1, \ldots, x_n : T_n$$

is a finite partial map from variables to types.

As usual, there is one typing rule for each syntactic form plus the rule T-SUB for subsumption, which allows the use of subtyping. Upwards and downwards role modifiers have separate rules, discussed below. The typing rules for the two systems differ only in their notion of subtyping and in the side condition on T-MOD-DN; we discuss the latter in Example 15.

$$\frac{}{\Gamma \vdash_\alpha bv : \mathsf{Base}}\ \text{(T-BASE)} \qquad \frac{}{\Gamma, x:T, \Gamma' \vdash_\alpha x : T}\ \text{(T-VAR)} \qquad \frac{\Gamma \vdash_\alpha M : T}{\Gamma \vdash_\alpha M : T'} \vdash_\alpha T <: T'\ \text{(T-SUB)}$$

$$\frac{\Gamma, x:T \vdash_\alpha M : S}{\Gamma \vdash_\alpha \lambda x.\, M : T \to S}\ x \notin dom(\Gamma)\ \text{(T-ABS)} \qquad \frac{\Gamma \vdash_\alpha M : T \to S \quad \Gamma \vdash_\alpha N : T}{\Gamma \vdash_\alpha M\, N : S}\ \text{(T-APP)} \qquad \frac{\Gamma \vdash_\alpha M : T \to T}{\Gamma \vdash_\alpha \mathsf{fix}\, M : T}\ \text{(T-FIX)}$$

$$\frac{\Gamma \vdash_\alpha M : T}{\Gamma \vdash_\alpha \{A\}\,[M] : \{A\}[T]}\ \text{(T-GRD)} \qquad \frac{\Gamma \vdash_\alpha M : \{A\}[T]}{\Gamma \vdash_\alpha \mathsf{check}\, M : \langle A\rangle[T]}\ \text{(T-CHK)}$$

$$\frac{\Gamma \vdash_\alpha M : T}{\Gamma \vdash_\alpha [M] : \langle \mathbf{0}\rangle[T]}\ \text{(T-UNIT)} \qquad \frac{\Gamma \vdash_\alpha M : \langle A\rangle[T] \quad \Gamma, x:T \vdash_\alpha N : \langle B\rangle[S]}{\Gamma \vdash_\alpha \mathsf{let}\, x = M;\, N : \langle A \sqcup B\rangle[S]}\ x \notin dom(\Gamma)\ \text{(T-BIND)}$$

$$\frac{\Gamma \vdash_\alpha M : \langle B\rangle[T]}{\Gamma \vdash_\alpha \uparrow A\,(M) : \langle B \sqcap A^\star\rangle[T]}\ \text{(T-MOD-UP)} \qquad \frac{\Gamma \vdash_\alpha M : \langle B\rangle[T]}{\Gamma \vdash_\alpha \downarrow A\,(M) : \langle B\rangle[T]}\ \text{if } \alpha = 1 \text{ then } A \geqslant B\ \text{(T-MOD-DN)}$$

The rules T-BASE, T-VAR, T-SUB, T-ABS, T-APP and T-FIX are standard. For example, the identity function has the expected typing, $\vdash_\alpha \lambda x.\, x : T \to T$, for any T. Nonterminating computations can also be typed; for example, $\vdash_\alpha \mathsf{fix}\ (\lambda x.\, x) : T$, for any T.

Any term may be injected into a computation type at the least role using T-UNIT. Thus, in the light of the earlier discussion on subtyping, if $\vdash_\alpha M : T$ then, in the first system, $[M]$ inhabits $\langle A\rangle[T]$ for every role A; in the second system, the term inhabits only type $\langle \mathbf{0}\rangle[T]$, indicating that no checks are required to successfully evaluate the value $[M]$.

Computations may be combined using T-BIND[4]. If M inhabits $\langle A\rangle[T]$ and N inhabits $\langle B\rangle[S]$, then "$M\,; N$" inhabits $\langle A \sqcup B\rangle[S]$. More generally, we can deduce:

$$\vdash_\alpha \lambda x.\, \mathsf{let}\, x' = x;\, x' : \langle A\rangle[\langle B\rangle[T]] \to \langle A \sqcup B\rangle[T]$$

In the first type system, this rule is motivated by noting that the role context $A \sqcup B$ suffices to successfully avoid role errors in the combined computation if A (resp. B) suffices for M (resp. N). For the second type system, consider a role C that is not bigger than $A \sqcup B$ — thus C is not bigger than at least one of A, B. If it is not greater than A, by assumption on typing of M, every computation path of M in role context C leads to a role-error. Similarly for B. Thus, in role context C, every computation path in the combined computation leads to a role error. Furthermore, using the earlier subtyping discussion, the sequence also inhabits $\langle \mathbf{1}\rangle[S]$ in the first system and $\langle \mathbf{0}\rangle[S]$ in the second.

The rule T-GRD types basic values with their protection level. The higher-order version of $\{A\}\,[\,]$ has the natural typing:

$$\vdash_\alpha \lambda x.\, \{A\}\,[x] : T \to \{A\}[T]$$

Recall that in the transition relation, $\mathsf{check}\ \{A\}\,[N]$ checks the role context against A. The typing rule T-CHK mirrors this behavior by converting the protection level of values into constraints on role contexts. For example, we have the typing:

$$\vdash_\alpha \lambda x.\, \mathsf{check}\, x : \{A\}[T] \to \langle A\rangle[T]$$

[4]The distinction between our system and dependency-based systems can be see in T-BIND, which in DCC [1, 2, 30] states that $\vdash \mathsf{let}\, x = M\,; N : \langle B\rangle[S]$ if $B \geqslant A$, where $\vdash M : \langle A\rangle[T]$ and $x : T \vdash N : \langle B\rangle[S]$.

In the special case of typing $\Gamma \vdash_\alpha \mathsf{check}\ \{A\}\,[N] : \langle A\rangle[T]$, we can further justify in the two systems as follows. In terms of the first type system, the role context passes this check if it is at least A. In terms of the second type system, any role context that does not include A will cause a role-error.

Role modifiers are treated by separate rules for upwards and downwards modifiers.

The rule for T-MOD-UP is justified for the first type system as follows. Under assumption that B suffices to evaluate M without role-errors, consider evaluation of $\uparrow A\,(M)$ in role context $B \sqcap A^\star$. This term contributes A to role context yielding $A \sqcup (B \sqcap A^\star) = (A \sqcup B) \sqcap (A \sqcup A^\star) = B$ for the evaluation of M. For the second type system, assume that if a role is not greater than B, then the evaluation of N leads to a role error. Consider the evaluation of $\uparrow A\,(M)$ in a role context C that does not exceed $B \sqcap A^\star$. Then, the evaluation of M proceeds in role context $C \sqcup A$ which does not exceed B and hence causes a role error by assumption.

The rule for T-MOD-DN is justified for the first type system as follows. Under assumption that B suffices to evaluate M without role-errors, and A is greater than B consider evaluation of $\downarrow A\,(M)$ in role context B. This term alters role-context B to $B \sqcap A = B$ for the evaluation of M, which suffices. For the second type system, assume that if a role is not greater than B, then the evaluation of N leads to a role error. Consider the evaluation of $\downarrow A\,(M)$ in a role context C that does not exceed B. Then, $C \sqcap A$ certainly does not exceed B and so the evaluation of M causes a role error by assumption.

Example 16 and Example 15 discuss alternate presentations for the rules of typing for the role modifiers.

In stating the results, we distinguish computations from other types. Lemma 10 holds trivially from the definitions.

Definition 9. Role A *dominates* type T (notation $A \geq T$) if T is not a computation type, or T is a computation type $\langle B\rangle[S]$ and $A \geqslant B$. □

Lemma 10. *(a) If $A \geqslant B$ and $B \geq T$ then $A \geq T$. (b) If $\vdash_1 T <: S$ and $A \geq S$ then $A \geq T$. (c) If $\vdash_2 T <: S$ and $A \geq T$ then $A \geq S$.* □

The following theorems formalize the guarantees provided by the two systems. The proofs may be found in Section 6.

Theorem 11. *If $\vdash_1 M : T$ and $A \geq T$, then either $A \triangleright M \to^\omega$ or $A \triangleright M \twoheadrightarrow V$ for some V.*

Theorem 12. *If $\vdash_2 M : T$ and $A \not\geqslant T$, then either $A \triangleright M \to^\omega$ or there exists N such that $A \triangleright M \twoheadrightarrow N$ and $A \triangleright N \not\downarrow \mathsf{err}$.*

For the first system, we have a standard type-safety theorem. For the second system, such a safety theorem does not hold; for example $\vdash_2 \mathsf{check}\ \{\mathbf{1}\}\,[\mathsf{unit}] : \langle \mathbf{1}\rangle[\mathsf{Unit}]$ and $\mathbf{1} \triangleright \mathsf{check}\ \{\mathbf{1}\}\,[\mathsf{unit}] \to [\mathsf{unit}]$ but $\not\vdash_2 [\mathsf{unit}] : \langle \mathbf{1}\rangle[\mathsf{Unit}]$. Instead Theorem 12 states that a term run with an insufficient context role is guaranteed either to diverge or to produce a role error.

3.3. **Simple examples.**

Example 13. We illustrate combinators of the language with some simple functions. The identity function may be given its usual type:

$$\vdash_\alpha \lambda \mathsf{x}.\,\mathsf{x} : \mathsf{T} \to \mathsf{T}$$

The unit of computation can be used to create a computation from any value:

$$\vdash_\alpha \lambda \mathsf{x}.\ [\mathsf{x}] : \mathsf{T} \to \langle \mathbf{0}\rangle[\mathsf{T}]$$

The let construct evaluates a computation. In this following example, the result of the computation x' must itself be a computation because it is returned as the result of the function:

$$\vdash_\alpha \lambda \mathsf{x}.\,\mathsf{let}\ \mathsf{x}' = \mathsf{x}\,;\,\mathsf{x}' : \langle \mathsf{A}\rangle[\langle \mathsf{B}\rangle[\mathsf{T}]] \to \langle \mathsf{A} \sqcup \mathsf{B}\rangle[\mathsf{T}]$$

The guard construct creates a guarded term:

$$\vdash_\alpha \lambda \mathsf{x}.\, \{\mathsf{A}\}[\mathsf{x}] : \mathsf{T} \to \{\mathsf{A}\}[\mathsf{T}]$$

The check construct discharges a guard, resulting in a computation:

$$\vdash_\alpha \lambda \mathsf{x}.\, \mathbf{check}\ \mathsf{x} : \{\mathsf{A}\}[\mathsf{T}] \to \langle \mathsf{A} \rangle[\mathsf{T}]$$

The upwards role modifier reduces the role required by a computation.

$$\vdash_\alpha \lambda \mathsf{x}.\, \uparrow \mathsf{B}\,(\mathsf{x}) : \langle \mathsf{A} \rangle[\mathsf{T}] \to \langle \mathsf{A} \sqcap \mathsf{B}^\star \rangle[\mathsf{T}]$$

The first typing system requires that any computation performed in the context of a downward role modifier $\downarrow \mathsf{B}\,()$ must not require more than role B:

$$\vdash_\alpha \lambda \mathsf{x}.\, \downarrow \mathsf{B}\,(\mathsf{x}) : \langle \mathsf{A} \rangle[\mathsf{T}] \to \langle \mathsf{A} \rangle[\mathsf{T}] \quad (\text{where } \mathsf{B} \geqslant \mathsf{A} \text{ if } \alpha = 1)$$

In the first type system, the last two judgments may be generalized as follows:

$$\vdash_1 \lambda \mathsf{x}.\, \rho(\mathsf{x}) : \langle \rho(\!| \mathsf{A} |\!) \rangle[\mathsf{T}] \to \langle \mathsf{A} \rangle[\mathsf{T}]$$

Thus a role modifier may be seen as transforming a computation that requires the modifier into one that does not. For further discussion see Example 16. □

Example 14 (Booleans)**.** The Church Booleans, $\mathsf{tru} \triangleq \lambda \mathtt{t}.\, \lambda \mathtt{f}.\, \mathtt{t}$ and $\mathsf{fls} \triangleq \lambda \mathtt{t}.\, \lambda \mathtt{f}.\, \mathtt{f}$, illustrate the use of subtyping. In the two systems, these may be given the following types.

$$\mathsf{Bool}_1 \triangleq \langle A \rangle[T] \to \langle B \rangle[T] \to \langle A \sqcup B \rangle[T] \qquad \vdash_1 \mathsf{tru}, \mathsf{fls} : \mathsf{Bool}_1$$
$$\mathsf{Bool}_2 \triangleq \langle A \rangle[T] \to \langle B \rangle[T] \to \langle A \sqcap B \rangle[T] \qquad \vdash_2 \mathsf{tru}, \mathsf{fls} : \mathsf{Bool}_2$$

These types reflect the intuitions underlying the two type systems. The first type system reflects a "maximum over all paths" typing, whereas the second reflects a "minimum over all paths" typing. The conditional may be interpreted using the following derived rules.

$$\frac{\Gamma \vdash_1 L : \mathsf{Bool}_1 \quad \Gamma \vdash_1 M : \langle A \rangle[T] \quad \Gamma \vdash_1 N : \langle B \rangle[T]}{\Gamma \vdash_1 \mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N : \langle A \sqcup B \rangle[T]} \qquad \frac{\Gamma \vdash_2 L : \mathsf{Bool}_2 \quad \Gamma \vdash_2 M : \langle A \rangle[T] \quad \Gamma \vdash_2 N : \langle B \rangle[T]}{\Gamma \vdash_2 \mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N : \langle A \sqcap B \rangle[T]}$$

□

Example 15 (T-MOD-DN)**.** The side condition on T-MOD-DN does not effect typability in second typing system, but may unnecessarily decrease the accuracy of the analysis, as can be seen from the following concrete example.

Let M, N be terms of type $\langle B \rangle[T]$.

$$\dfrac{\dfrac{\Gamma \vdash_\alpha M : \langle B \rangle[T]}{\Gamma \vdash_\alpha M : \langle A \sqcap B \rangle[T]}\ (\text{T-SUB})}{\Gamma \vdash_\alpha \downarrow A\,(M) : \langle A \sqcap B \rangle[T]}\ (\text{T-MOD-DN})$$

With the side condition, the term $\mathsf{let}\ x = \downarrow A\,(M)\,;\, N$ would have to be given a type of the form $\langle A \sqcap B \rangle[T]$, even though both M and N have type $\langle B \rangle[T]$. Without the side condition, the "better" type $\langle B \rangle[T]$ may be given to the entire let expression. □

Example 16 (Alternative rule for role modifiers)**.** In the first typing system, T-MOD-UP and T-MOD-DN may be replaced with the following rule, which we call T-MOD-*.

$$\frac{\Gamma \vdash_{\mathsf{T}} M : \langle \rho(\!|B|\!)\rangle[T]}{\Gamma \vdash_{\mathsf{T}} \rho(M) : \langle B\rangle[T]}$$

Consider $\rho = \uparrow A$. Because $C \geqslant (A \sqcup C) \sqcap A^\star$, the following are equivalent.

$$\frac{\Gamma \vdash_{\mathsf{T}} M : \langle A \sqcup C\rangle[T]}{\Gamma \vdash_{\mathsf{T}} \uparrow A\,(M) : \langle C\rangle[T]}\ \text{(T-MOD-*)} \qquad \frac{\dfrac{\Gamma \vdash_{\mathsf{T}} M : \langle A \sqcup C\rangle[T]}{\Gamma \vdash_{\mathsf{T}} \uparrow A\,(M) : \langle (A \sqcup C) \sqcap A^\star\rangle[T]}\ \text{(T-MOD-UP)}}{\Gamma \vdash_{\mathsf{T}} \uparrow A\,(M) : \langle C\rangle[T]}\ \text{(T-SUB)}$$

Because $(D \sqcap A^\star) \sqcup A \geqslant D$, the following are equivalent.

$$\frac{\dfrac{\dfrac{\Gamma \vdash_{\mathsf{T}} M : \langle D\rangle[T]}{\Gamma \vdash_{\mathsf{T}} M : \langle (D \sqcap A^\star) \sqcup A\rangle[T]}\ \text{(T-SUB)}}{\Gamma \vdash_{\mathsf{T}} \uparrow A\,(M) : \langle D \sqcap A^\star\rangle[T]}\ \text{(T-MOD-*)} \qquad \frac{\Gamma \vdash_{\mathsf{T}} M : \langle D\rangle[T]}{\Gamma \vdash_{\mathsf{T}} \uparrow A\,(M) : \langle D \sqcap A^\star\rangle[T]}\ \text{(T-MOD-UP)}$$

Consider $\rho = \downarrow A$. Because $A \geqslant A \sqcap C$ and $C \geqslant A \sqcap C$, the following are equivalent.

$$\frac{\Gamma \vdash_{\mathsf{T}} M : \langle A \sqcap C\rangle[T]}{\Gamma \vdash_{\mathsf{T}} \downarrow A\,(M) : \langle C\rangle[T]}\ \text{(T-MOD-*)} \qquad \frac{\dfrac{\Gamma \vdash_{\mathsf{T}} M : \langle A \sqcap C\rangle[T]}{\Gamma \vdash_{\mathsf{T}} \downarrow A\,(M) : \langle A \sqcap C\rangle[T]}\ \text{(T-MOD-DN)}}{\Gamma \vdash_{\mathsf{T}} \downarrow A\,(M) : \langle C\rangle[T]}\ \text{(T-SUB)}$$

Suppose $A \geqslant D$. Then $D \sqcap A \geqslant D$, and the following are equivalent.

$$\frac{\dfrac{\dfrac{\Gamma \vdash_{\mathsf{T}} M : \langle D\rangle[T]}{\Gamma \vdash_{\mathsf{T}} M : \langle D \sqcap A\rangle[T]}\ \text{(T-SUB)}}{\Gamma \vdash_{\mathsf{T}} \downarrow A\,(M) : \langle D\rangle[T]}\ \text{(T-MOD-*)} \qquad \frac{\Gamma \vdash_{\mathsf{T}} M : \langle D\rangle[T]}{\Gamma \vdash_{\mathsf{T}} \downarrow A\,(M) : \langle D\rangle[T]}\ \text{(T-MOD-DN)}$$ □

Example 17 (A sublanguage)**.** The following proper sublanguage is sufficient to encode the computational lambda calculus. Here values and terms are disjoint, with values assigned value types T and terms assigned computation types $\langle A\rangle[T]$.

$$\begin{aligned} T,S &::= \mathsf{Base} \mid T \to \langle A\rangle[S] \mid \{A\}[T] \\ V,U,W &::= bv \mid x \mid \lambda x.M \mid \{A\}[V] \\ M,N,L &::= [V] \mid V\ U \mid \mathsf{fix}\,V \mid \mathsf{check}\,V \mid \mathsf{let}\,x = M; N \mid \rho(M) \end{aligned}$$

Encoding the Church Booleans in this sublanguage is slightly more complicated than in Example 14; tru and fls must accept thunks of type $\mathsf{Unit} \to \langle A\rangle[S]$ rather than the simpler blocks of type $\langle A\rangle[S]$.

Operations on base values that have no computational effect are placed in the language of values rather than the language of terms. The resulting terms may be simplified at any time without affecting the computation (e.g., `[1+2 == 3]` may be simplified to `[tru]`). □

Example 18 (Relation to conference version)**.** The language presented here is much simpler than that of the conference version of this paper [14]. In particular, the conference version collapsed guards and abstractions into a single form $\{A\}[\lambda x.M]$ with types of the form $T \to \{A \triangleright B\}[S]$, which translates here as $\{A\}\big[T \to \langle B\rangle[S]\big]$: the immediate guard of the abstraction is A, whereas the effect of applying the abstraction is B.

In addition, the conference version collapsed role modification and application: the application $\downarrow C\ V\ U$ first checked the guard of V, then performed the application in a context modified by $\downarrow C$. In the current presentation, this translates as "$\mathsf{let}\,x = \mathsf{check}\,V\,;\downarrow C\,(x\,U)$." □

4. EXAMPLES

In this section we assume nullary role constructors for user roles, such as Alice, Bob, Charlie, Admin, and Daemon.

Example 19 (ACLs)**.** Consider a read-only filesystem protected by Access Control Lists (ACLs). One can model such a system as:

$$\begin{aligned}\mathsf{filesystem} \triangleq \lambda \mathsf{name}.\ &\mathbf{if}\ \mathsf{name}{=}{=}\texttt{"file1"}\ \mathbf{then}\ \mathbf{check}\ \{\mathsf{Admin}\}[\texttt{"data1"}]\\ &\mathbf{else}\ \mathbf{if}\ \mathsf{name}{=}{=}\texttt{"file2"}\ \mathbf{then}\ \mathbf{check}\ \{\mathsf{Alice}\sqcap\mathsf{Bob}\}[\texttt{"data2"}]\\ &\mathbf{else}\ [\texttt{"error: file not found"}]\end{aligned}$$

If $\mathsf{Admin} \geq \mathsf{Alice}\sqcap\mathsf{Bob}$ then code running in the Admin role can access both files:

$$\begin{aligned}&\mathsf{Admin} \triangleright \mathsf{filesystem}\ \texttt{"file1"} \twoheadrightarrow \mathbf{check}\ \{\mathsf{Admin}\}[\texttt{"data1"}] \twoheadrightarrow [\texttt{"data1"}]\\ &\mathsf{Admin} \triangleright \mathsf{filesystem}\ \texttt{"file2"} \twoheadrightarrow \mathbf{check}\ \{\mathsf{Alice}\sqcap\mathsf{Bob}\}[\texttt{"data2"}] \twoheadrightarrow [\texttt{"data2"}]\end{aligned}$$

If $\mathsf{Alice} \not\geq \mathsf{Admin}$ then code running as Alice cannot access the first file but can access the second:

$$\begin{aligned}&\mathsf{Alice} \triangleright \mathsf{filesystem}\ \texttt{"file1"} \twoheadrightarrow \mathbf{check}\ \{\mathsf{Admin}\}[\texttt{"data1"}] \not\twoheadrightarrow \mathbf{err}\\ &\mathsf{Alice} \triangleright \mathsf{filesystem}\ \texttt{"file2"} \twoheadrightarrow \mathbf{check}\ \{\mathsf{Alice}\sqcap\mathsf{Bob}\}[\texttt{"data2"}] \twoheadrightarrow [\texttt{"data2"}]\end{aligned}$$

Finally, if $\mathsf{Charlie} \not\geq \mathsf{Alice}\sqcap\mathsf{Bob}$ then code running as Charlie cannot access either file:

$$\begin{aligned}&\mathsf{Charlie} \triangleright \mathsf{filesystem}\ \texttt{"file1"} \twoheadrightarrow \mathbf{check}\ \{\mathsf{Admin}\}[\texttt{"data1"}] \not\twoheadrightarrow \mathbf{err}\\ &\mathsf{Charlie} \triangleright \mathsf{filesystem}\ \texttt{"file2"} \twoheadrightarrow \mathbf{check}\ \{\mathsf{Alice}\sqcap\mathsf{Bob}\}[\texttt{"data2"}] \not\twoheadrightarrow \mathbf{err}\end{aligned}$$

The filesystem code can be assigned the following type, meaning that a caller must possess a role from each of the ACLs in order to guarantee that access checks will not fail. If, in addition, $\mathsf{Admin} \geq \mathsf{Alice}\sqcap\mathsf{Bob}$ then the final role is equal to Admin.

$$\vdash_1 \mathsf{filesystem} : \mathsf{String} \to \langle \mathsf{Admin} \sqcup (\mathsf{Alice}\sqcap\mathsf{Bob}) \sqcup \mathbf{0}\rangle[\mathsf{String}]$$

In the above type, the final role $\mathbf{0}$ arises from the "unknown file" branch that does not require an access check. The lack of an access check explains the weaker $\vdash_2$ type:

$$\vdash_2 \mathsf{filesystem} : \mathsf{String} \to \langle \mathsf{Admin} \sqcap (\mathsf{Alice}\sqcap\mathsf{Bob}) \sqcap \mathbf{0}\rangle[\mathsf{String}]$$

This type indicates that filesystem has the potential to expose some information to unprivileged callers with role $\mathsf{Admin}\sqcap(\mathsf{Alice}\sqcap\mathsf{Bob})\sqcap\mathbf{0} \doteq \mathbf{0}$, perhaps causing the code to be flagged for security review. □

Example 20 (Web server)**.** Consider a web server that provides remote access to the filesystem described above. The web server can use the role assigned to a caller to access the filesystem (unless the web server's caller withholds its role). To prevent an attacker determining the non-existence of files via the web server, the web server fails when an attempt is made to access an unknown file unless the Debug role is activated.

$$\begin{aligned}\mathsf{webserver} \triangleq \lambda \mathsf{name}.\ &\mathbf{if}\ \mathsf{name}{=}{=}\texttt{"file1"}\ \mathbf{then}\ \mathsf{filesystem}\ \mathsf{name}\\ &\mathbf{else}\ \mathbf{if}\ \mathsf{name}{=}{=}\texttt{"file2"}\ \mathbf{then}\ \mathsf{filesystem}\ \mathsf{name}\\ &\mathbf{else}\ \mathbf{check}\ \{\mathsf{Debug}\}[\texttt{"error: file not found"}]\end{aligned}$$

For example, code running as Alice can access "file2" via the web server:

$$\mathsf{Alice} \triangleright \mathsf{webserver}\ \texttt{"file2"} \twoheadrightarrow \mathsf{filesystem}\ \texttt{"file2"} \twoheadrightarrow [\texttt{"data2"}]$$

The access check in the web server does prevent the "file not found" error message leaking unless the Debug role is active, but, unfortunately, it is not possible to assign a role strictly greater

than $\mathbf{0}$ to the web server using the second type system. The filesystem type does not record the different roles that must be checked depending upon the filename argument.

$$\vdash_2 \mathsf{webserver} : \mathsf{String} \to \langle \mathsf{Admin} \sqcap (\mathsf{Alice} \sqcap \mathsf{Bob}) \sqcap \mathbf{0} \rangle [\mathsf{String}] \qquad \text{(derivable)}$$

$$\nvdash_2 \mathsf{webserver} : \mathsf{String} \to \langle \mathsf{Admin} \sqcap (\mathsf{Alice} \sqcap \mathsf{Bob}) \sqcap \mathsf{Debug} \rangle [\mathsf{String}] \qquad \text{(not derivable)} \quad \square$$

Example 21 illustrates how the *Domain-Type Enforcement* (DTE) access control mechanism [6, 31], found in Security-Enhanced Linux (SELINUX) [17], can be modelled in λ-RBAC. Further discussion of the relationship between RBAC and DTE can be found in [11, 13].

Example 21 (Domain-Type Enforcement)**.** The DTE access control mechanism grants or denies access requests according to the current *domain* of running code. The current domain changes as new programs are executed, and transitions between domains are restricted in order to allow, and also force, code to run with an appropriate domain. The restrictions upon domain transitions are based upon a DTE type associated with each program to execute. For example, the DTE policy in [31] only permits transitions from a domain for daemon processes to a domain for login processes when executing the login program, because code running in the login domain is highly privileged. This effect is achieved by allowing transitions from the daemon domain to the login domain only upon execution of programs associated with a particular DTE type, and that DTE type is assigned only to the login program.

The essence of DTE can be captured in λ-RBAC, using roles to model both domains and DTE types, and the context role to model the current domain of a system. We start by building upon the code fragment $\lambda \mathsf{g}.\, \lambda \mathsf{y}.\, \mathsf{B}\,(\mathsf{g}\,\mathsf{y})$, discussed in Example 7, that allows a function checking role B to be executed in the context of code running at a different role. We have the typing (for emphasis we use extra parentheses that are not strictly necessary given the usual right associativity for the function type constructor):

$$\vdash_\alpha \lambda \mathsf{g}.\, \lambda \mathsf{y}.\, \mathsf{B}\,(\mathsf{g}\,\mathsf{y}) : (\mathsf{T} \to \langle \mathsf{B} \rangle [\mathsf{S}]) \to (\mathsf{T} \to \langle \mathbf{0} \rangle [\mathsf{S}])$$

To aid readability, and fixing types T and S for the remainder of this example, define:

$$\overline{\mathsf{R}} \triangleq \mathsf{R} \to (\mathsf{T} \to \langle \mathbf{0} \rangle [\mathsf{S}])$$

So that the previous typing becomes:

$$\vdash_\alpha \lambda \mathsf{g}.\, \lambda \mathsf{y}.\, \mathsf{B}\,(\mathsf{g}\,\mathsf{y}) : \overline{\mathsf{T} \to \langle \mathsf{B} \rangle [\mathsf{S}]}$$

To restrict the use of the privileged function $\lambda \mathsf{g}.\, \lambda \mathsf{y}.\, \mathsf{B}\,(\mathsf{g}\,\mathsf{y})$, it can be guarded by a role E acting as a DTE type, where the association of the DTE type E with a function is modelled in the sequel by code that can activate role E. The guarded function can be typed as:

$$\vdash_\alpha \{\mathsf{E}\}[\lambda \mathsf{g}.\, \lambda \mathsf{y}.\, \mathsf{B}\,(\mathsf{g}\,\mathsf{y})] : \{\mathsf{E}\}[\overline{\mathsf{T} \to \langle \mathsf{B} \rangle [\mathsf{S}]}]$$

We now define a function domtrans<A, E, B> for a domain transition from domain (role) A to domain (role) B upon execution of a function associated with DTE type (also a role) E. The function first verifies that the context role dominates A, and then permits use of the privileged function $\lambda \mathsf{g}.\, \lambda \mathsf{y}.\, \mathsf{B}\,(\mathsf{g}\,\mathsf{y})$ by code that can activate role E. The function domtrans<A, E, B> is defined by:

$$\mathsf{domtrans{<}A, E, B{>}} \triangleq \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, \mathbf{check}\ \{\mathsf{A}\}[\mathsf{unit}]\ ;\ \mathsf{f}\ \{\mathsf{E}\}[\lambda \mathsf{g}.\, \lambda \mathsf{y}.\, \mathsf{B}\,(\mathsf{g}\,\mathsf{y})]\ \mathsf{x}$$

We have the typing:

$$\vdash_\alpha \mathsf{domtrans{<}A, E, B{>}} : \overline{\{\mathsf{E}\}[\overline{\mathsf{T} \to \langle \mathsf{B} \rangle [\mathsf{S}]}]} \to (\mathsf{T} \to \langle \mathsf{A} \rangle [\mathsf{S}])$$

The above type shows that domtrans<A, E, B> can be used to turn a function checking role B into a function checking role A, but only when the role E is available—in contrast to the type $\overline{(\mathsf{T} \to \langle \mathsf{B} \rangle[\mathsf{S}])} \to (\mathsf{T} \to \langle \mathsf{A} \rangle[\mathsf{S}])$ that does not require E.

In order to make use of domtrans<A, E, B>, we must also consider code that can activate E. We define a function assign<E> that takes a function f and activates E in order to access the privileged code $\lambda \mathsf{g}.\, \lambda \mathsf{y}.\, \mathsf{B}\,(\mathsf{g}\ \mathsf{y})$ from domtrans<A, E, B>. The function assign<E> is defined by:

$$\mathsf{assign{<}E{>}} \triangleq \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, \lambda \mathsf{y}.\, \mathbf{let}\ \mathsf{g} = \mathsf{E}\,(\mathbf{check}\ \mathsf{x})\ ;\ \mathsf{g}\ \mathsf{f}\ \mathsf{y}$$

And we have the typing:

$$\vdash_{\alpha} \mathsf{assign{<}E{>}} : (\mathsf{T} \to \langle \mathsf{B} \rangle[\mathsf{S}]) \to \overline{\{\mathsf{E}\}[\overline{\mathsf{T} \to \langle \mathsf{B} \rangle[\mathsf{S}]}]}$$

Therefore the functional composition of assign<E> and domtrans<A, E, B> has type:

$$(\mathsf{T} \to \langle \mathsf{B} \rangle[\mathsf{S}]) \to (\mathsf{T} \to \langle \mathsf{A} \rangle[\mathsf{S}])$$

To show that in the presence of both assign<E> and domtrans<A, E, B>, code running with context A can execute code checking for role context B, we consider the following reductions in role context A, where we take $\mathscr{F} \triangleq \lambda \mathsf{z}.\, \mathbf{check}\ \{\mathsf{B}\}[\mathsf{unit}]$ and underline terms to indicate the redex:

$$\begin{aligned}
&\underline{\mathsf{domtrans{<}A,\ E,\ B{>}}}\ (\mathsf{assign{<}E{>}}\ \mathscr{F})\ \mathsf{unit} \\
=\ &(\underline{\lambda \mathsf{f}.\, \lambda \mathsf{x}.\, \mathbf{check}\ \{\mathsf{A}\}[\mathsf{unit}]\ ;\ \mathsf{f}\ (\{\mathsf{E}\}[\lambda \mathsf{g}.\, \lambda \mathsf{y}.\, \mathsf{B}\,(\mathsf{g}\ \mathsf{y})]\,)\ \mathsf{x})}\ \underline{(\mathsf{assign{<}E{>}}\ \mathscr{F})}\ \mathsf{unit} \\
\to\ &(\underline{\lambda \mathsf{x}.\, \mathbf{check}\ \{\mathsf{A}\}[\mathsf{unit}]\ ;\ (\mathsf{assign{<}E{>}}\ \mathscr{F})\ (\{\mathsf{E}\}[\lambda \mathsf{g}.\, \lambda \mathsf{y}.\, \mathsf{B}\,(\mathsf{g}\ \mathsf{y})]\,)\ \mathsf{x})}\ \underline{\mathsf{unit}} \\
\to\ &\underline{\mathbf{check}\ \{\mathsf{A}\}[\mathsf{unit}]}\ ;\ (\mathsf{assign{<}E{>}}\ \mathscr{F})\ (\{\mathsf{E}\}[\lambda \mathsf{g}.\, \lambda \mathsf{y}.\, \mathsf{B}\,(\mathsf{g}\ \mathsf{y})]\,)\ \mathsf{unit} \\
\to\ &(\underline{\mathsf{assign{<}E{>}}}\ \mathscr{F})\ (\{\mathsf{E}\}[\lambda \mathsf{g}.\, \lambda \mathsf{y}.\, \mathsf{B}\,(\mathsf{g}\ \mathsf{y})]\,)\ \mathsf{unit} \\
=\ &\big((\underline{\lambda \mathsf{f}.\, \lambda \mathsf{x}.\, \lambda \mathsf{y}.\, \mathbf{let}\ \mathsf{g} = \mathsf{E}\,(\mathbf{check}\ \mathsf{x})\ ;\ \mathsf{g}\ \mathsf{f}\ \mathsf{y})}\ \underline{\mathscr{F}}\big)\ (\{\mathsf{E}\}[\lambda \mathsf{g}.\, \lambda \mathsf{y}.\, \mathsf{B}\,(\mathsf{g}\ \mathsf{y})]\,)\ \mathsf{unit} \\
\to\ &(\underline{\lambda \mathsf{x}.\, \lambda \mathsf{y}.\, \mathbf{let}\ \mathsf{g} = \mathsf{E}\,(\mathbf{check}\ \mathsf{x})\ ;\ \mathsf{g}\ \mathscr{F}\ \mathsf{y}})\ (\underline{\{\mathsf{E}\}[\lambda \mathsf{g}.\, \lambda \mathsf{y}.\, \mathsf{B}\,(\mathsf{g}\ \mathsf{y})]})\ \mathsf{unit} \\
\to\ &(\underline{\lambda \mathsf{y}.\, \mathbf{let}\ \mathsf{g} = \mathsf{E}\,(\mathbf{check}\ \{\mathsf{E}\}[\lambda \mathsf{g}.\, \lambda \mathsf{y}.\, \mathsf{B}\,(\mathsf{g}\ \mathsf{y})]\,)\ ;\ \mathsf{g}\ \mathscr{F}\ \mathsf{y}})\ \underline{\mathsf{unit}} \\
\to\ &\mathbf{let}\ \mathsf{g} = \mathsf{E}\,(\underline{\mathbf{check}\ \{\mathsf{E}\}[\lambda \mathsf{g}.\, \lambda \mathsf{y}.\, \mathsf{B}\,(\mathsf{g}\ \mathsf{y})]}\,)\ ;\ \mathsf{g}\ \mathscr{F}\ \mathsf{unit} \\
\to\ &\mathbf{let}\ \mathsf{g} = \underline{\mathsf{E}\,([\lambda \mathsf{g}.\, \lambda \mathsf{y}.\, \mathsf{B}\,(\mathsf{g}\ \mathsf{y})]\,)}\ ;\ \mathsf{g}\ \mathscr{F}\ \mathsf{unit} \\
\twoheadrightarrow\ &\underline{\mathbf{let}\ \mathsf{g} = [\lambda \mathsf{g}.\, \lambda \mathsf{y}.\, \mathsf{B}\,(\mathsf{g}\ \mathsf{y})]\ ;\ \mathsf{g}\ \mathscr{F}\ \mathsf{unit}} \\
\to\ &(\underline{\lambda \mathsf{g}.\, \lambda \mathsf{y}.\, \mathsf{B}\,(\mathsf{g}\ \mathsf{y})})\ \underline{\mathscr{F}}\ \mathsf{unit} \\
\to\ &(\underline{\lambda \mathsf{y}.\, \mathsf{B}\,(\mathscr{F}\ \mathsf{y})})\ \underline{\mathsf{unit}} \\
\to\ &\mathsf{B}\,(\underline{\mathscr{F}}\ \mathsf{unit}) \\
=\ &\mathsf{B}\,((\underline{\lambda \mathsf{z}.\, \mathbf{check}\ \{\mathsf{B}\}[\mathsf{unit}]})\ \underline{\mathsf{unit}}) \\
\to\ &\mathsf{B}\,(\underline{\mathbf{check}\ \{\mathsf{B}\}[\mathsf{unit}]}) \\
\to\ &\underline{\mathsf{B}\,([\mathsf{unit}]\,)} \\
\twoheadrightarrow\ &[\mathsf{unit}]
\end{aligned}$$

The strength of DTE lies in the ability to factor access control policies into two components: the set of permitted domain transitions and the assignment of DTE types to code. We illustrate this by adapting the aforementioned login example from [31] to λ-RBAC. In this example, the DTE mechanism is used to force every invocation of user code (running at role User) from daemon code (running at role Daemon) to occur via trusted login code (running at role Login). This is achieved by providing domain transitions from Login to User, and Daemon to Login, but no others. Moreover, code permitted to run at Login must be assigned DTE type LoginEXE, and similarly for User and

UserEXE. Thus a full program running daemon code M has the following form, where neither M nor N contain direct rights amplification.

```
let dtLoginToUser = domtrans<Login, UserEXE, User>;
let dtDaemonToLogin = domtrans<Daemon, LoginEXE, Login>;
let shell = assign<UserEXE> (λ. M);
let login = assign<LoginEXE> (λpwd. if pwd=="secret" then dtLoginToUser shell unit else ...);
Daemon(N)
```

Because login provides the sole gateway to the role User, the daemon code N must provide the correct password in order to execute the shell at User (in order to access resources that are available at role User but not at role Daemon). In addition, removal of the domain transition dtDaemonToLogin makes it impossible for the daemon code to execute any code at User. □

5. CONTROLLING RIGHTS AMPLIFICATION

Example 22. Suppose that M contains no direct rights amplification, that is, no subterms of the form ↑A(·). Then, in

```
let priv = [λx. ↑A(V x)]; ↓User(M)
```

we may view V as a *Trusted Computing Base* (TCB) — a privileged function which may escalate rights — and view M as restricted *user code*. The function priv is an entry point to the TCB which is accessible to user code; that is, user code is executed at the restricted role User, and rights amplification may only occur through invocation of priv.

Non-trivial programs have larger TCBs with more entry points. As the size of the TCB grows, it becomes difficult to understand the security guarantees offered by a system when rights amplification is unconstrained, even if only in the TCB. To manage this complexity, one may enforce a coding convention that requires rights increases be justified by earlier checks. As an example, consider the following, where *amplify* is a unary role constructor.

```
let at<A> = [λf. check {amplify(A)}[λx. ↑A(f x)]];
let priv = at<A> V;
↓User(M)
```

In a context with role *amplify*(A), this reduces (using R-BIND, R-APP and R-CHK) to

```
let priv = [λx. ↑A(V x)]; ↓User(M)
```

In a context without role *amplify*(A), evaluation becomes stuck when attempting to execute R-CHK. The privileged function returned by at<A> (which performs rights amplification for A) is justified by the check for *amplify*(A) on any caller of at<A>.

One may also wish to explicitly prohibit a term from direct amplification of some right B; with such a convention in place, this can be achieved using the role modifier ↓*amplify*(B). □

One may formalize the preceding example by introducing the unary role constructor *amplify*, where *amplify*(*A*) stands for the right to provide the role *A* by storing ↑*A* in code.

We require that *amplify* distribute over ⊔ and ⊓ and obey the following absorption laws:

$$A \sqcup \mathit{amplify}(A) \doteq \mathit{amplify}(A) \qquad\qquad A \sqcap \mathit{amplify}(A) \doteq A$$

Thus $\mathit{amplify}(A) \geqslant A$ for any role *A*.

To distinguish justified use of role modifiers from unjustified use, we augment the syntax with *checked role modifiers*.

$$M, N ::= \cdots \mid \rho_A(M)$$

Whenever a check is performed on role M we *mark* role modifiers in the consequent to indicate that these modifiers have been justified by a check. Define the function $mark_A$ homomorphically over all terms but for role modifiers:

$$mark_A(\rho(M)) = \rho_A(mark_A(M))$$
$$mark_A(\rho_B(M)) = \rho_{A \sqcup B}(mark_A(M))$$

Modify the reduction rule for check as follows.

$$\frac{}{A \triangleright \mathsf{check}\ \{B\}[M] \longrightarrow [mark_B(M)]}\ A \geqslant B$$

Thus, the check in the example above will execute as follows.

$$amplify(A) \triangleright \mathsf{check}\ \{amplify(A)\}[\lambda x.\uparrow A(f\,x)] \longrightarrow \lambda x.\uparrow A_{amplify(A)}(f\,x)$$

In the residual, the abstraction contains a checked role modifier, indicating that the role amplification has been provided by code that had the right to do so.

We now define *role modification errors* so that $\uparrow A_B(M)$ produces an error if B does not dominate $amplify(A)$.

$$\frac{}{\uparrow B(M)\ \lightning\ \mathsf{moderr}} \qquad \frac{}{\uparrow B_C(M)\ \lightning\ \mathsf{moderr}}\ C \not\geqslant amplify(B)$$

$$\frac{M\ \lightning\ \mathsf{moderr}}{M\,N\ \lightning\ \mathsf{moderr}} \qquad \frac{M\ \lightning\ \mathsf{moderr}}{\mathsf{let}\ x = M;\ N\ \lightning\ \mathsf{moderr}} \qquad \frac{M\ \lightning\ \mathsf{moderr}}{\mathsf{check}\ M\ \lightning\ \mathsf{moderr}} \qquad \frac{M\ \lightning\ \mathsf{moderr}}{\rho(M)\ \lightning\ \mathsf{moderr}}$$

Using this augmented language, unjustified rights amplification is noted as an error. To prevent such errors, we modify the typing system to have judgments of the form $\Gamma; C \vdash_\alpha M : T$, where C indicates the accumulated guards on a term which must be discharged before the term may be executed; since M is guarded by C, it may include subterms of the form $\uparrow A(\cdot)$ when $C \geqslant amplify(A)$. In addition to adding rules for checked role modifiers, we also modify T-GRD and T-MOD-UP. The rule T-MOD-UP ensures that any amplification is justified by C. The rule T-GRD allows guards to be used in checking guarded terms; the rule is sound since guarded terms must be checked before they are executed.

(T-GRD′)
$$\frac{\Gamma; C \sqcup A \vdash_\alpha M : T}{\Gamma; C \vdash_\alpha \{A\}[M] : \{A\}[T]}$$

(T-MOD-DN-CHECKED)
$$\frac{\Gamma; C \vdash_\alpha M : \langle B\rangle[T]}{\Gamma; C \vdash_\alpha \downarrow A_D(M) : \langle B\rangle[T]}\ \text{if } \alpha = 1 \text{ then } A \geqslant B$$

(T-MOD-UP′)
$$\frac{\Gamma; C \vdash_\alpha M : \langle B\rangle[T]}{\Gamma; C \vdash_\alpha \uparrow A(M) : \langle B \sqcap A^\star\rangle[T]}\ C \geqslant amplify(A)$$

(T-MOD-UP-CHECKED)
$$\frac{\Gamma; C \vdash_\alpha M : \langle B\rangle[T]}{\Gamma; C \vdash_\alpha \uparrow A_D(M) : \langle B \sqcap A^\star\rangle[T]}\ C \sqcup D \geqslant amplify(A)$$

One may not assume that top level terms have been guarded; therefore, let $\Gamma \vdash_\alpha M : T$ be shorthand for $\Gamma; \mathbf{0} \vdash_\alpha M : T$.

Example 23. The functions domtrans and assign from Example 21 are not typable using this more restrictive system. Recall the definitions:

$$\mathsf{domtrans<A, E, B>} \triangleq \lambda \mathsf{f}.\,\lambda \mathsf{x}.\,\mathbf{check}\ \{\mathsf{A}\}[\mathsf{unit}];\ \mathsf{f}\ \{\mathsf{E}\}[\lambda \mathsf{g}.\,\lambda \mathsf{y}.\,\mathsf{B}\,(\mathsf{g}\,\mathsf{y})]\ \mathsf{x}$$
$$\mathsf{assign<E>} \triangleq \lambda \mathsf{f}.\,\lambda \mathsf{x}.\,\lambda \mathsf{y}.\,\mathsf{let}\ \mathsf{g} = \mathsf{E}(\mathbf{check}\ \mathsf{x});\ \mathsf{g}\ \mathsf{f}\ \mathsf{y}$$

The amplification of B in domtrans is not justified; neither is the amplification of E in assign. The required form is:

$$\mathsf{domtrans<A, E, B>} \triangleq \{amplify(\mathsf{B})\}[\lambda \mathsf{f}.\,\lambda \mathsf{x}.\,\mathbf{check}\ \{\mathsf{A}\}[\mathsf{unit}];\ \mathsf{f}\ \{\mathsf{E}\}[\lambda \mathsf{g}.\,\lambda \mathsf{y}.\,\mathsf{B}\,(\mathsf{g}\,\mathsf{y})]\ \mathsf{x}]$$

$$\mathsf{assign\langle E\rangle} \triangleq \{\mathit{amplify}(\mathsf{E})\}[\lambda \mathsf{f}\,.\,\lambda \mathsf{x}\,.\,\lambda \mathsf{y}\,.\,\mathsf{let\ g = E\,(check\ x)\,;\,g\ f\ y}]$$

The login example must now be modified in order to discharge the guards. Again the modifications are straightforward:

```
let dtLoginToUser = check domtrans<Login, UserEXE, User>;
let dtDaemonToLogin = check domtrans<Daemon, LoginEXE, Login>;
let assignXUser = check assign<UserEXE>;
let assignXLogin = check assign<LoginEXE>;
let shell = assignXUser (λ . M);
let login = assignXLogin (λpwd . if pwd=="secret" then dtLoginToUser shell unit else ...);
Daemon(N)
```

Thus modified, the program types correctly, but will only execute in a context that dominates the four roles *amplify*(User), *amplify*(UserEXE), *amplify*(Login), and *amplify*(LoginEXE). This ensures that domain transitions and assignments are created by authorized code. □

Proposition 25 establishes that the typing system is sufficient to prevent role modification errors. The proof of Proposition 25 relies on the following lemma, which establishes the relation between typing and *mark*.

Lemma 24. *If* $\Gamma; C \sqcup A \vdash_\alpha M : T$ *then* $\Gamma; C \vdash_\alpha \mathit{mark}_A(M) : T$.

Proof. By induction on the derivation of the typing judgment, appealing to the definition of *mark*. □

Proposition 25. *If* $\vdash_\alpha M : T$ *and* $A \triangleright M \twoheadrightarrow N$ *then* $\neg(N \not\downarrow \mathsf{moderr})$

Proof Sketch. That $\neg(M \not\downarrow \mathsf{moderr})$ follows immediately from the definition of role modification error, combined with T-MOD-UP′ and T-MOD-UP-CHECKED. It remains only to show that typing is preserved by reduction. We prove this for the type systems of Section 3 in the next section. The proof extends easily to the type system considered here. The only wrinkle is the evaluation rule for check, which is handled using the previous lemma. □

6. Proof of Type Safety Theorems

The proofs for the first and second systems are similar, both relying on well-studied techniques [23]. We present proofs for the second system, which is the more challenging of the two.

Definition 26 (Compatibility)**.** Types T and S are *compatible* (notation $T \approx S$) if $T = S$ or $T = \langle A\rangle[R]$ and $S = \langle B\rangle[R]$, for some type R. □

The following lemmas have straightforward inductive proofs.

Lemma 27 (Compatibility)**.** *If* $\vdash_\alpha T <: T'$ *then* $T \approx S$ *iff* $T' \approx S$. □

Lemma 28 (Substitution)**.** *If* $\Gamma \vdash_\alpha M : T$ *and* $\Gamma, x : T \vdash_\alpha N : S$, *then* $\Gamma \vdash_\alpha N\{x := M\} : S$. □

Lemma 29 (Bound Weakening)**.** *If* $\Gamma, x : S \vdash_\alpha M : T$ *and* $\vdash_\alpha S' <: S$, *then* $\Gamma, x : S' \vdash_\alpha M : T$. □

Lemma 30 (Canonical Forms)**.**

(1) If $\vdash_2 V : T \to S$ *then* V *has form* $(\lambda x.\, M)$ *where* $x{:}T \vdash_2 M : S$*.*
(2) If $\vdash_2 V : \langle A\rangle[T]$ *then* V *has form* $[M]$ *where* $\vdash_2 M : T$ *and* $A = \mathbf{0}$*.*
(3) If $\vdash_2 V : \{A\}[T]$ *then* V *has form* $\{B\}[M]$ *where* $\vdash_2 M : T$ *and* $B \geqslant A$*.*

Proof.

(1) By induction on derivation of $\vdash_2 V : T \to S$. The only applicable cases are T-SUB and T-ABS.
(T-SUB) We know $\vdash_2 V : T' \to S'$, where $\vdash_2 V : T \to S$ and $\vdash_2 T \to S <: T' \to S'$, so $\vdash_2 T' <: T$ and $\vdash_2 S <: S'$. By the IH, V has form $(\lambda x.\, M)$ where $x{:}T \vdash_2 M : S$. By Lemma 29 and subsumption, $x{:}T' \vdash_2 M : S'$.
(T-ABS) Immediate.

(2) By induction on derivation of $\vdash_2 V : \langle A\rangle[T]$. The only applicable cases are T-SUB and T-UNIT.
(T-SUB) We know $\vdash_2 V : \langle A'\rangle[T']$, where $\vdash_2 V : \langle A\rangle[T]$ and $\vdash_2 \langle A\rangle[T] <: \langle A'\rangle[T']$, so $\vdash_2 T <: T'$ and $A \geqslant A'$. By the IH, V has form $[M]$ where $\vdash_2 M : T$ and $A = \mathbf{0}$, so $A' = \mathbf{0}$. By subsumption, $\vdash_2 M : T'$.
(T-UNIT) Immediate.

(3) By induction on derivation of $\vdash_2 V : \{A\}[T]$. The only applicable cases are T-SUB and T-GRD.
(T-SUB) We know $\vdash_2 V : \{A'\}[T']$, where $\vdash_2 V : \{A\}[T]$ and $\vdash_2 \{A\}[T] <: \{A'\}[T']$, so $\vdash_2 T <: T'$ and $A \geqslant A'$. By the IH, V has form $\{B\}[M]$ where $\vdash_2 M : T$ and $B \geqslant A$, so $B \geqslant A'$. By subsumption, $\vdash_2 M : T'$.
(T-GRD) Immediate. □

Proposition 31 (Preservation)**.** *If* $\vdash_2 M : T$ *and* $A \triangleright M \to N$ *then there exists* S *such that* $S \approx T$ *and* $\vdash_2 N : S$ *and if* $A \geq S$ *then* $A \geq T$*.*

Proof. By induction on the derivation of $\vdash_2 M : T$. The induction hypothesis includes the quantification over A, N. For values, the result is trivial; thus we consider only the rules for non-values.

(T-SUB) We know $\vdash_2 M : T'$, where $\vdash_2 M : T$ and $\vdash_2 T <: T'$, and $A \triangleright M \to N$. Applying the IH to $\vdash_2 M : T$ and $A \triangleright M \to N$ yields S such that $\vdash_2 N : S$ and $S \approx T$ and if $A \geq S$ then $A \geq T$. By Lemma 10c, this extends to if $A \geq S$ then $A \geq T'$. In addition, by Lemma 27, we have $S \approx T'$.

(T-APP) We know $\vdash_2 M\,N : T_2$, where $\vdash_2 M : T_1 \to T_2$ and $\vdash_2 N : T_1$, and $A \triangleright M\,N \to L$. There are two subcases depending on the reduction rule used in $A \triangleright M\,N \to L$.
(M is a value) By Lemma 30, $M = \lambda x.\, M'$ and $x{:}T_1 \vdash_2 M' : T_2$. The reduction yields $L = M'\{x := N\}$. By Lemma 28, $\vdash_2 L : T_2$. The remaining requirements on T_2 are immediate.
(M has a reduction) Therefore $A \triangleright M \to M'$ and $L = M'\,N$. Applying the IH to $\vdash_2 M : T_1 \to T_2$ and $A \triangleright M \to M'$ yields S such that $\vdash_2 M' : S$ and $S \approx T_1 \to T_2$, which implies that $S = T_1 \to T_2$. Hence $\vdash_2 L : T_2$. The remaining requirements on T_2 are immediate.

(T-FIX) We know $\vdash_2 \mathsf{fix}\, M : T$, where $\vdash_2 M : T \to T$ and $A \triangleright \mathsf{fix}\, M \to L$. There are two subcases depending on the reduction rule used in $A \triangleright \mathsf{fix}\, M \to L$.
(M is a value) By Lemma 30, $M = \lambda x.\, M'$ and $x{:}T \vdash_2 M' : T$. The reduction yields $L = M'\{x := M\}$. By Lemma 28, $\vdash_2 L : T$. The remaining requirements on T are immediate.
(M has a reduction) Therefore $A \triangleright M \to M'$ and $L = \mathsf{fix}\, M'$. Applying the IH to $\vdash_2 M : T \to T$ and $A \triangleright M \to M'$ yields S such that $\vdash_2 M' : S$ and $S \approx T \to T$, which implies that $S = T \to T$. Hence $\vdash_2 L : T$. The remaining requirements on T are immediate.

(T-CHK) We know $\vdash_2 \mathsf{check}\, M : \langle A_1\rangle[T]$, where $\vdash_2 M : \{A_1\}[T]$, and $A \triangleright \mathsf{check}\, M \to L$. There are two subcases depending on the reduction rule used in $A \triangleright \mathsf{check}\, M \to L$.
(M is a value) By Lemma 30, $M = \{A_2\}[M']$ and $\vdash_2 M' : T$ and $A_2 \geqslant A_1$. The reduction yields $L = [M']$ and from the reduction we deduce $A \geqslant A_2$, so $A \geqslant A_1$ always holds.

We assign type $\vdash_2 L : \langle \mathbf{0} \rangle [T]$, where $\langle \mathbf{0} \rangle [T] \approx \langle A_1 \rangle [T]$, and we have already shown that $A \geqslant \mathbf{0}$ implies $A \geqslant A_1$.

(M has a reduction) Therefore $A \triangleright M \to M'$ and $L = \mathsf{check}\, M'$. Applying the IH to $\vdash_2 M : \{A_1\}[T]$ and $A \triangleright M \to M'$ yields S such that $\vdash_2 M' : S$ and $S \approx \{A_1\}[T]$, so $S = \{A_1\}[T]$. Hence $\vdash_2 L : \langle A_1 \rangle [T]$. The remaining requirements on $\langle A_1 \rangle [T]$ are immediate.

(T-BIND) We know $\vdash_2 \mathsf{let}\, x = M; N : \langle A_1 \sqcup A_2 \rangle [T_2]$, where $\vdash_2 M : \langle A_1 \rangle [T_1]$ and $x : T_1 \vdash_2 N : \langle A_2 \rangle [T_2]$, and $A \triangleright \mathsf{let}\, x = M; N \to L$. There are two subcases depending on the reduction rule used in $A \triangleright \mathsf{let}\, x = M; N \to L$.

(M is a value) By Lemma 30, $M = [M']$ and $\vdash_2 M' : T_1$ and $A_1 = \mathbf{0}$, so $A_1 \sqcup A_2 = A_2$. The reduction yields $L = N\{x := M'\}$. By Lemma 28, $\vdash_2 L : \langle A_2 \rangle [T_2]$. The remaining requirements on T_2 are immediate.

(M has a reduction) Therefore $A \triangleright M \to M'$ and $L = \mathsf{let}\, x = M'; N$. Applying the IH to $\vdash_2 M : \langle A_1 \rangle [T_1]$ and $A \triangleright M \to M'$ yields S such that $\vdash_2 M' : S$ and $S \approx \langle A_1 \rangle [T_1]$ and if $A \geq S$ then $A \geq \langle A_1 \rangle [T_1]$. Hence $S = \langle A_3 \rangle [T_1]$, for some A_3, and $A \geqslant A_3$ implies $A \geqslant A_1$. We deduce $\vdash_2 L : \langle A_3 \sqcup A_2 \rangle [T_2]$, where $\langle A_3 \sqcup A_2 \rangle [T_2] \approx \langle A_1 \sqcup A_2 \rangle [T_2]$. Finally, suppose $A \geq \langle A_3 \sqcup A_2 \rangle [T_2]$, i.e., $A \geqslant A_3 \sqcup A_2$, so $A \geqslant A_3$ and $A \geqslant A_2$. By the above, this entails $A \geqslant A_1$, so $A \geqslant A_1 \sqcup A_2$. Therefore $A \geq \langle A_1 \sqcup A_2 \rangle [T_2]$, as required.

(T-MOD-UP) We know $\vdash_2 \uparrow A_1\, (M) : \langle A_2 \sqcap A_1^\star \rangle [T]$, where $\vdash_2 M : \langle A_2 \rangle [T]$, and $A \triangleright \uparrow A_1\, (M) \to L$. There are two subcases depending on the reduction rule used in $A \triangleright \uparrow A_1\, (M) \to L$.

(M is a value) Therefore $L = M$ and $\vdash_2 L : \langle A_2 \rangle [T]$, where $\langle A_2 \rangle [T] \approx \langle A_2 \sqcap A_1^\star \rangle [T]$. By Lemma 30, $M = [M']$ and $\vdash_2 M' : T$ and $A_2 = \mathbf{0}$, and the remaining requirement on $\langle A_2 \rangle [T]$, that $A \geq \langle A_2 \rangle [T]$ implies $A \geq \langle A_2 \sqcap A_1^\star \rangle [T]$, is immediate.

(M has a reduction) Therefore $\uparrow A_1 (\!| A |\!) \triangleright M \to M'$ and $L = \uparrow A_1\, (M')$. Applying the IH to $\vdash_2 M : \langle A_2 \rangle [T]$ and $\uparrow A_1 (\!| A |\!) \triangleright M \to M'$ yields S such that $\vdash_2 M' : S$ and $S \approx \langle A_2 \rangle [T]$, so $S = \langle A_3 \rangle [T]$ for some A_3, and if $\uparrow A_1 (\!| A |\!) \geq S$ then $\uparrow A_1 (\!| A |\!) \geq \langle A_2 \rangle [T]$, i.e., $A \sqcup A_1 \geqslant A_3$ implies $A \sqcup A_1 \geqslant A_2$. We have $\vdash_2 \uparrow A_1\, (M') : \langle A_3 \sqcap A_1^\star \rangle [T]$ and $\langle A_3 \sqcap A_1^\star \rangle [T] \approx \langle A_2 \sqcap A_1^\star \rangle [T]$. Finally, if $A \geq \langle A_3 \sqcap A_1^\star \rangle [T]$, then $A \geqslant A_3 \sqcap A_1^\star$, so $A \sqcup A_1 \geqslant (A_3 \sqcap A_1^\star) \sqcup A_1 = (A_3 \sqcup A_1) \sqcap \mathbf{1} = A_3 \sqcup A_1$. Hence $A \sqcup A_1 \geqslant A_3$, so $A \sqcup A_1 \geqslant A_2$, and $A \sqcap A_1^\star = (A \sqcap A_1^\star) \sqcup \mathbf{0} = (A \sqcup A_1) \sqcap A_1^\star \geqslant A_2 \sqcap A_1^\star$. Therefore $A \geqslant A_2 \sqcap A_1^\star$ and $A \geq \langle A_2 \sqcap A_1^\star \rangle [T]$, as required.

(T-MOD-DN) We know $\vdash_2 \downarrow A_1\, (M) : \langle A_2 \rangle [T]$, where $\vdash_2 M : \langle A_2 \rangle [T]$, and $A \triangleright \downarrow A_1\, (M) \to L$. There are two subcases depending on the reduction rule used in $A \triangleright \downarrow A_1\, (M) \to L$.

(M is a value) Therefore $L = M$ and $\vdash_2 L : \langle A_2 \rangle [T]$, and we are done.

(M has a reduction) Therefore $\downarrow A_1 (\!| A |\!) \triangleright M \to M'$ and $L = \downarrow A_1\, (M')$. By $A \geqslant \downarrow A_1 (\!| A |\!)$ and Lemma 4, we have $A \triangleright M \to M'$. Applying the IH to $\vdash_2 M : \langle A_2 \rangle [T]$ and $A \triangleright M \to M'$ yields S such that $\vdash_2 M' : S$ and $S \approx \langle A_2 \rangle [T]$, so $S = \langle A_3 \rangle [T]$ for some A_3, and if $A \geq S$ then $A \geq \langle A_2 \rangle [T]$. Hence $\vdash_2 \downarrow A_1\, (M') : \langle A_3 \rangle [T]$, which completes the subcase. □

Corollary 32. *If $\vdash_2 M : T$ and $A \triangleright M \twoheadrightarrow V$, then $A \geq T$.*

Proof. By induction on the length of the reduction sequence $A \triangleright M \twoheadrightarrow V$. For the base case, $M = V$ and Lemma 30 implies that $A \geq T$, because every non-computation type is dominated by any role, and in a computation type $T = \langle B \rangle [S]$ Lemma 30 tells us that $B = \mathbf{0}$. For the inductive step, there exists N such that $A \triangleright M \to N$ and $A \triangleright N \twoheadrightarrow V$. By Proposition 31, there exists S such that $\vdash_2 N : S$ and if $A \geq S$ then $A \geq T$. Applying the IH to $\vdash_2 N : S$ and $A \triangleright N \twoheadrightarrow V$ yields $A \geq S$, hence $A \geq T$ as required. □

Proposition 33 (Progress)**.** *For all A, if $\vdash_2 M : T$ then either M is a value, $A \triangleright M \not\downarrow \mathsf{err}$, or there exists N such that $A \triangleright M \to N$.*

Proof. By induction on the derivation of $\vdash_? M : T$. We need only consider the cases when M is not a value.

(T-SUB) We know $\vdash_? M : T'$, where $\vdash_? M : T$ and $\vdash_? T <: T'$. Immediate by the IH.

(T-APP) We know $\vdash_? M\,N : T_2$, where $\vdash_? M : T_1 \to T_2$ and $\vdash_? N : T_1$. Apply the IH to $\vdash_? M : T_1 \to T_2$ and role A. If M is a value, then, by Lemma 30, M has form $(\lambda x.\, L)$, so $A \triangleright M\,N \to L\{x := N\}$. If $A \triangleright M \not\downarrow \mathsf{err}$, then $A \triangleright M\,N \not\downarrow \mathsf{err}$. Finally, if $A \triangleright M \to L$, then $A \triangleright M\,N \to L\,N$.

(T-FIX) We know $\vdash_? \mathsf{fix}\,M : T$, where $\vdash_? M : T \to T$. Apply the IH to $\vdash_? M : T \to T$ and role A. If M is a value, then, by Lemma 30, M has form $(\lambda x.\, L)$, so $A \triangleright \mathsf{fix}\,M \to L\{x := (\lambda x.\, L)\}$. If $A \triangleright M \not\downarrow \mathsf{err}$, then $A \triangleright \mathsf{fix}\,M \not\downarrow \mathsf{err}$. Finally, if $A \triangleright M \to L$, then $A \triangleright \mathsf{fix}\,M \to \mathsf{fix}\,L$.

(T-CHK) We know $\vdash_? \mathsf{check}\,M : \langle A_1 \rangle[T]$, where $\vdash_? M : \{A_1\}[T]$. Apply the IH to $\vdash_? M : \{A_1\}[T]$ and role A. If M is a value, then, by Lemma 30, there exists B, L such that $M = \{B\}[L]$, so either $A \triangleright \mathsf{check}\,M \to [L]$ or $A \triangleright \mathsf{check}\,M \not\downarrow \mathsf{err}$ depending on whether $A \geqslant B$ holds or not. If $A \triangleright M \not\downarrow \mathsf{err}$, then $A \triangleright \mathsf{check}\,M \not\downarrow \mathsf{err}$. Finally, if $A \triangleright M \to L$, then $A \triangleright \mathsf{check}\,M \to \mathsf{check}\,L$.

(T-BIND) We know $\vdash_? \mathsf{let}\,x = M\,;\,N : \langle A_1 \sqcup A_2 \rangle[T_2]$, where $\vdash_? M : \langle A_1 \rangle[T_1]$ and $x : T_1 \vdash_? N : \langle A_2 \rangle[T_2]$. Apply the IH to $\vdash_? M : \langle A_1 \rangle[T_1]$ and role A. If M is a value, then, by Lemma 30, M has form $[L]$, so $A \triangleright \mathsf{let}\,x = M\,;\,N \to N\{x := L\}$. If $A \triangleright M \not\downarrow \mathsf{err}$, then $A \triangleright \mathsf{let}\,x = M\,;\,N \not\downarrow \mathsf{err}$. Finally, if $A \triangleright M \to L$, then $A \triangleright \mathsf{let}\,x = M\,;\,N \to \mathsf{let}\,x = L\,;\,N$.

(T-MOD-UP) We know $\vdash_? \uparrow\!A_1\,(M) : \langle A_2 \sqcap A_1^\star \rangle[T]$, where $\vdash_? M : \langle A_2 \rangle[T]$. Apply the IH to $\vdash_? M : \langle A_2 \rangle[T]$ and role $\uparrow\!A_1(\!|A|\!)$. If M is a value, then $A \triangleright \uparrow\!A_1\,(M) \to M$. If $\uparrow\!A_1(\!|A|\!) \triangleright M \not\downarrow \mathsf{err}$, then $A \triangleright \uparrow\!A_1\,(M) \not\downarrow \mathsf{err}$. Finally, if $\uparrow\!A_1(\!|A|\!) \triangleright M \to L$, then $A \triangleright \uparrow\!A_1\,(M) \to \uparrow\!A_1\,(L)$.

(T-MOD-DN) We know $\vdash_? \downarrow\!A_1\,(M) : \langle A_2 \rangle[T]$, where $\vdash_? M : \langle A_2 \rangle[T]$. Apply the IH to $\vdash_? M : \langle A_2 \rangle[T]$ and role $\downarrow\!A_1(\!|A|\!)$. If M is a value, then $A \triangleright \downarrow\!A_1\,(M) \to M$. If $\downarrow\!A_1(\!|A|\!) \triangleright M \not\downarrow \mathsf{err}$, then $A \triangleright \downarrow\!A_1\,(M) \not\downarrow \mathsf{err}$. Finally, if $\downarrow\!A_1(\!|A|\!) \triangleright M \to L$, then $A \triangleright \downarrow\!A_1\,(M) \to \downarrow\!A_1\,(L)$. □

Theorem (12)**.** *If $\vdash_? M : T$ and $A \not\geq T$, then either $A \triangleright M \to^\omega$ or there exists N such that $A \triangleright M \twoheadrightarrow N$ and $A \triangleright N \not\downarrow \mathsf{err}$.*

Proof. We use a coinductive argument to construct a reduction sequence that is either infinite or terminates with a role check failure. When $\vdash_? M : T$ and $A \not\geq T$, we know that M is not a value by Lemma 30. By Proposition 33, either $A \triangleright M \not\downarrow \mathsf{err}$ or there exists N such that $A \triangleright M \to N$. In the former case, we are done. In the latter case, using Proposition 31, there exists S such that $\vdash_? N : S$ and if $A \geq S$ then $A \geq T$. However, we know that $A \not\geq T$, so $A \not\geq S$, as required. □

7. Conclusions

The focus of this paper is programmatic approaches, such as JAAS/.NET, that use RBAC. From a software engineering approach to the design of components, RBAC facilitates a separation of concerns: the design of the system is carried out in terms of a role hierarchy with an associated assignment of permissions to roles, whereas the actual assignment of users to roles takes place at the time of deployment.

We have presented two methods to aid the design and use of components that include such access control code. The first — admittedly standard — technique enables users of code to deduce the role at which code must be run. The main use of this analysis is to optimize code by enabling the removal of some dynamic checks. The second — somewhat more novel — analysis calculates the role that is verified on all execution paths. This analysis is potentially useful in validating architectural security requirements by enabling code designers to deduce the protection guarantees of their code.

We have demonstrated the use of these methods by modeling Domain Type Enforcement, as used in SELinux. As future work, we will explore extensions to role polymorphism and recursive roles following the techniques of [8, 4].

ACKNOWLEDGMENTS

The presentation of the paper has greatly improved thanks to the comments of the referees.

REFERENCES

[1] M. Abadi. Access control in a core calculus of dependency. *SIGPLAN Not.*, 41(9):263–273, 2006.
[2] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *POPL '99*, pages 147–160. ACM Press, 1999.
[3] M. Abadi, G. Morrisett, and A. Sabelfeld. Language-based security. *J. Funct. Program.*, 15(2):129, 2005.
[4] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM TOPLAS*, 15(4):575–631, 1993.
[5] S. Barker and P. J. Stuckey. Flexible access control policy specification with constraint logic programming. *ACM Trans. Inf. Syst. Secur.*, 6(4):501–546, 2003.
[6] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *CSS '85*, 1985.
[7] C. Braghin, D. Gorla, and V. Sassone. A distributed calculus for role-based access control. In *CSFW*, pages 48–60, 2004.
[8] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundam. Inf.*, 33(4):309–338, 1998.
[9] S. Chong and A. C. Myers. Security policies for downgrading. In *CCS '04*, pages 198–209, 2004.
[10] A. Compagnoni, P. Garralda, and E. Gunter. Role-based access control in a mobile environment. In *Symposium on Trustworthy Global Computing*, 2005.
[11] D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-Based Access Control*. Computer Security Series. Artech House, 2003.
[12] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, 4(3):224–274, 2001.
[13] J. Hoffman. Implementing RBAC on a type enforced system. In *Computer Security Applications (ACSAC '97)*, pages 158–163, 1997.
[14] R. Jagadeesan, A. Jeffrey, C. Pitcher, and J. Riely. λ-RBAC: Programming with role-based access control. In *ICALP '06*, volume 4052 of *Lecture Notes in Computer Science*, pages 456–467. Springer, 2006.
[15] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Trans. Database Syst.*, 26(2):214–260, 2001.
[16] J. Ligatti, L. Bauer, and D. Walker. Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Sec.*, 4(1-2):2–16, 2005.
[17] P. A. Loscocco and S. D. Smalley. Meeting critical security objectives with Security-Enhanced Linux. In *Ottawa Linux Symposium*, 2001.
[18] S. Malhotra. *Microsoft .NET Framework Security*. Premier Press, 2002.
[19] J. C. Mitchell. Programming language methods in computer security. In *POPL '01*, pages 1–26, 2001.
[20] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *CSFW*, pages 172–186, 2004.
[21] S. Osborn, R. Sandhu, and Q. Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Trans. Inf. Syst. Secur.*, 3(2):85–106, 2000.
[22] J. S. Park, R. S. Sandhu, and G.-J. Ahn. Role-based access control on the web. *ACM Trans. Inf. Syst. Secur.*, 4(1):37–71, 2001.
[23] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.
[24] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
[25] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *ISSS*, pages 174–191, 2003.
[26] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *IEEE Computer*, 29(2), 1996.
[27] F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics—10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*, pages 86–101, 2000.
[28] F. Siewe, A. Cau, and H. Zedan. A compositional framework for access control policies enforcement. In *FMSE '03*, pages 32–42, 2003.

[29] E. G. Sirer and K. Wang. An access control language for web services. In *SACMAT '02*, pages 23–30, 2002.
[30] S. Tse and S. Zdancewic. Translating dependency into parametricity. In *ICFP*, pages 115–125, 2004.
[31] K. M. Walker, D. F. Sterne, M. L. Badger, M. J. Petkac, D. L. Shermann, and K. A. Oostendorp. Confining root programs with Domain and Type Enforcement (DTE). In *USENIX Security Symposium*, 1996.

CTI, DEPAUL UNIVERSITY
E-mail address: rjagadeesan@cti.depaul.edu

BELL LABS
E-mail address: ajeffrey@bell-labs.com

CTI, DEPAUL UNIVERSITY
E-mail address: cpitcher@cti.depaul.edu

CTI, DEPAUL UNIVERSITY
E-mail address: jriely@cti.depaul.edu