# λ-RBAC: Programmatic Role-Based Access Control (Extended Abstract)

Radha Jagadeesan[1,⋆], Alan Jeffrey[2,⋆], Corin Pitcher[1], and James Riely[1,⋆⋆]

[1] DePaul University, CTI
[2] Bell Labs, Lucent Technologies

## 1 Introduction

The context of this paper is enlarging the scope of foundational, language-based security methods (see [31,19,3] for surveys) to the realm of programming role-based access control (RBAC). Our investigations are motivated by extant mechanisms that permit components to express a form of role constraints on clients enabling the designer to aid in access control — e.g. Java Authentication and Authorization Service (JAAS), .NET and SELinux already permit programmatic use of role-based access control. Thus, this paper falls squarely in the corner of the programmatic implementation and enforcement of security policies, rather than the specification of security policies.

The main motivation for RBAC, e.g. see [29], in software architectures is that it enables the enforcement of security policies at a granularity demanded by the application. Roughly speaking, roles are the unit of administration for users and privileges are assigned to "roles". Roles are often arranged in a hierarchy for succinct representation of the mapping of privileges to facilitate the desired implementation of the "Principle of Least Privilege" [28]. This argument is supported by diverse applications, e.g. [24,23].

EXAMPLE 1. In traditional UNIX systems the root user and the programs that run as root ("setuid" programs [25,8]) are all-powerful. More modern Solaris systems [33] allow a user to assume a role that provides only *some* of superuser's capabilities. For example, the system administrator role has the solaris.admin.usermgr.read and the solaris.admin.usermgr.write authorizations for making changes to user files. However, without the solaris.admin.usermgr.pswd authorization, the system administrator cannot change passwords.

The Solaris role hierarchy provides an explicit model for the ability to delegate roles. An authorization that ends with the suffix grant enables a user or a role to delegate to other users any assigned authorizations that begin with the same prefix. For example, a role with the authorizations solaris.admin.usermgr.grant and solaris.admin.usermgr.read can delegate the solaris.admin.usermgr.read authorization to another user. □

*Our contributions.* We initiate the study of RBAC in the programming language setting by studying a small lambda calculus, λ-RBAC that embodies the key features of RBAC.

⋆ Supported in part by NSF Cybertrust 0430175
⋆⋆ Supported by NSF Cybertrust 0430175 and NSF Career 0347542

What is the expressiveness that is required of a programming language framework that supports RBAC? We draw inspiration from the programming idioms in extant architectures such as JAAS and .NET — for completeness, appendix A contains code excerpts exemplifying the arguments made in the rest of this paragraph. In these frameworks, program execution takes place in the context of a role, which can be viewed concretely as a set of permissions. Roles are closed under union and intersection operations. There are combinators to check that the role-context is at least some minimum role: an exception is raised if the check fails. In the .NET framework, rights modulation is achieved via a technique called impersonation: this enables an application to operate under the guise of different users at different times. In this light, example 1 can be viewed as providing a `grant` role constructor (written in suffix in this example) standing for the rights to enable impersonation.

These observations inspire the design of λ-RBAC. We study a call-by-value lambda calculus where execution occurs in the context of a role. We assume that roles form a lattice: abstracting the concrete union/intersection operations of the motivating examples. Our study is parametric on the underlying role lattice. Our calculus includes combinators for role checks. We separate the impersonation combinators into two pieces: one for rights weakening and the other for rights amplification. Inspired by example 1, we internalize the right to amplify rights by considering role lattices with an explicit role constructor, *amplify*.

We demonstrate the expressiveness of the calculus by building a range of useful combinators and a variety of small illustrative examples. We consider two kinds of *static analysis*.

- Analysis to determine a (minimal) role that is guaranteed to run a piece of code without role-errors.
- Analysis to determine the (maximal) role that is guaranteed to be required by a piece of code.

The first analysis enables removal of unnecessary role-checks in a piece of code for a caller at a sufficiently high role. The second analysis determines the amount of protection that is enforced by a piece of code. We formalize both analysis as type systems and prove preservation and progress properties.

*Rest of the paper.* We begin with a discussion of the dynamic semantics of λ-RBAC in section 2 and follow up in section 3 to clarify the expressiveness with examples. Section 4 describes the static analysis. The following section 5 provides types for the examples of section 3. We conclude with a summary of related work in section 6.

*For completeness, some material is presented in the appendices that may be read at the discretion of the reader.*

## 2 Language and Operational Semantics

### 2.1 Roles

The language of roles is built up from role constructors. The choice of role constructors is application dependent, but must include at least the five constructors listed below. We assume that each role constructor $\kappa$ has an associated arity, $\mathsf{arity}(\kappa)$.

ROLES

| | |
|---|---|
| $\kappa ::= \cdots$ | *Role Constructors* |
| $\bot$ | *least role (arity 0)* |
| $\top$ | *greatest role (arity 0)* |
| $\sqcup$ | *join (arity 2)* |
| $\sqcap$ | *meet (arity 2)* |
| *amplify* | *amplification (arity 1)* |
| $P,Q,R,S,T ::=$ | *Roles* |
| $\kappa(R_1,\ldots,R_n)$ | *constructor* |

The semantics of roles is defined by the relation "$\vdash R \geqslant S$" which states that $R$ dominates $S$. We do not define this relation, but rather assume that it has a suitable, application-specific definition; we impose only the following requirements. We require that all constructors are monotone with respect to $\geqslant$. Further we require that roles form a distributive lattice: we require that the set of constructors include the nullary constructors $\bot$ and $\top$ and binary constructors $\sqcup$ and $\sqcap$ (which we write infix). $\bot$ is the least element; $\top$ is the greatest element; $\sqcup$ and $\sqcap$ are idempotent, commutative, associative, and mutually distributive meet and join operations on the lattice of roles. Note that, for any $R$, $S$, we have $\vdash R \geqslant \bot$ and $\vdash \top \geqslant R$ and $\vdash R \sqcup S \geqslant R$ and $\vdash R \geqslant R \sqcap S$. Finally, we require the unary constructor *amplify*, discussed below: *amplify*$(R)$ will stand for the right to store $R$ in a piece of code. Its use is demonstrated in section 2.3.

### 2.2 Terms

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 the call-by-value lambda calculus[3] because it is simple and well understood, yet rich enough to capture the key concepts. (We expect that our ideas can be adapted to both process and object calculi.) The "components" in a lambda term are abstractions and their calling contexts. Thus it is function calls and returns that we seek to regulate, and, therefore, the language has roles decorating abstractions and applications.

We first present the calculus of runtime terms, which include syntactic forms for *frames*.

TERMS

| | |
|---|---|
| $x,y,z,f,g$ | *Value Variables* |
| $U,V ::= \cdots$ | *Values* |
| $x$ | *value variable* |

[3] We have chosen an explictly sequenced variant (with let). Implicit sequencing can be recovered using the following syntax sugar: $\downarrow R\,M\,N \triangleq \mathsf{let}\ x{=}M\,;\ \mathsf{let}\ y{=}N\,;\ \downarrow R\,x\,y$. When $x$ does not appear free in $N$, we abbreviate $\mathsf{let}\ x{=}M\,;\ N$ as $M\,;\ N$ . To focus the presentation, we elide base types, indicating them in the syntax using ellipses. In examples, we use base types with the usual operators and semantics, including Int (with values 0, 1, etc), Bool (with values true, false) and Unit (with value ()). We write $M[^{U}/_{x}]$ for the capture-avoiding substitution of $U$ for $x$ in $M$.

| | |
|---|---|
| $\{Q\}\lambda x.M$ | *value abstraction, requiring $Q$ ($x$ bound with scope $M$)* |

| | |
|---|---|
| $M,N,L ::= \cdots$ | *Terms* |
| $U$ | *value* |
| $\mathsf{let}\ x{=}M\,;\ N$ | *let ($x$ bound with scope $N$)* |
| $\downarrow P\ U\ V$ | *value application, restricting to $P$* |
| $\downarrow P[M]$ | *frame, restricting to $P$* |
| $\uparrow P[M]$ | *frame, providing $P$* |

Evaluation is defined in terms of a *role context*. Formally, we define a judgment $R \vdash M \to N$, which indicates $R$ is authorized to compute a single step of the initial program $M$, resulting in the new program $N$.

EVALUATION $(R \vdash M \to N)$

$$\frac{\vdash R \geqslant Q}{R \vdash \downarrow P\ (\{Q\}\lambda x.M)\ U \to \downarrow P[M[{}^{U}/_{x}]]}\ \text{(EVAL-APP)}$$

$$\frac{R \vdash M \to M'}{R \vdash \mathsf{let}\ x{=}M\,;\ N \to \mathsf{let}\ x{=}M'\,;\ N}\ \text{(EVAL-LET1)}$$

$$\frac{R \sqcap P \vdash M \to M'}{R \vdash \downarrow P[M] \to \downarrow P[M']}\ \text{(EVAL-RESTRICT1)}$$

$$\frac{R \sqcup P \vdash M \to M'}{R \vdash \uparrow P[M] \to \uparrow P[M']}\ \text{(EVAL-PROVIDE1)}$$

$$\frac{}{R \vdash \mathsf{let}\ x{=}U\,;\ N \to N[{}^{U}/_{x}]}\ \text{(EVAL-LET2)}$$

$$\frac{}{R \vdash \downarrow P[U] \to U}\ \text{(EVAL-RESTRICT2)}$$

$$\frac{}{R \vdash \uparrow P[U] \to U}\ \text{(EVAL-PROVIDE2)}$$

Most of the rules are straightforward: let-bindings provide sequencing, and frames affect the role context. We distinguish two types of frames: *restricting* frames reduce the role context, and *providing* frames enhance it.

The rule EVAL-APP for application is slightly more complex. Application involves two participants: the caller (or calling context) and the callee (or abstraction). Each participant may wish to protect itself from the other. When the caller $\downarrow P\ V\ U$ transfers control to $V$, it may protect itself by restricting the role context to $P$ while executing $V$. Symmetrically, the callee $(\{Q\}\lambda x.M)$ may protect itself by demanding that the role context before the call dominates $Q$. Significantly, the restricting frame created by the caller does not take effect until after the guard is satisfied. In brief, the protocol is "call-test-restrict," with the callee controlling the middle step. This alternation explains why restriction is syntactically fused into application.

A role is *trivial* if it has no effect on evaluation. Thus $\top$ is trivial in restricting frames and applications, whereas $\bot$ is trivial in providing frames and abstractions. We often elide trivial roles and trivial frames; thus, $\lambda x.M$ is read as $\{\bot\}\lambda x.M$ (the check always succeeds), and $U\ V$ is read as $\downarrow\top\ U\ V$ (the role context is unaffected by the resulting frame). In our semantics, these terms evaluate like ordinary lambda terms.

By stringing together a series of small steps, the final value for the program can be determined. Successful termination is written $R \vdash M \Downarrow U$ which indicates that $R$ is authorized to run the program $M$ to completion, with result $U$. Evaluation can fail

because the term diverges or because an inadequate role is provided at some point in the computation; we write the latter as $R \vdash M \Downarrow \mathsf{fail}$[4].

LEMMA 2. *If* $S \vdash M \to M'$ *and* $\vdash R \geqslant S$ *then* $R \vdash M \to M'$. □

### 2.3 The Trusted Computing Base and User Code

The ability to amplify rights must be carefully controlled to achieve security goals. Access checks have no value if code can arbitrarily raise its role, but the opposite extreme of disallowing all occurrences of $\uparrow R[\,\cdot\,]$ is overly restrictive. In this subsection we motivate, and describe a mechanism to achieve, systematic control over rights amplification.

Consider the following program where $M$ contains no direct rights amplification (subterms of the form $\uparrow R[\,\cdot\,]$) but $U$ has no such restriction:

$$\mathsf{let}\ g = U\ ;\ \downarrow T\,[M]$$

We sometimes refer to $g$ or $U$ as part of the Trusted Computing Base (TCB) or as privileged functions, and to $M$ as user code. When the entire program is run in the top role, the user code will initially run in role $T$ but may invoke $g$, possibly leading to rights amplification. However, as the size of the TCB grows, it becomes too difficult to understand the security guarantees offered by a system allowing arbitrary rights amplification in all TCB code.

Access control in the presence of the *amplify*$(\,\cdot\,)$ constructor provides a flexible dynamic way to control rights amplification. The following coding uses the derived form $M[\![N]\!]$, which means roughly "run $N$ in the context provided by $M$." The definition ensures that $(\lambda f.\,\lambda x.\,f\,x)\,[\![N]\!]$ evaluates to $N$. (It is straightforward to define polymorphic variants using the extensions discussed in Appendix B.)

DERIVED FORMS

$$M,N,L ::= \cdots \mid M[\![N]\!] \mid \{\!|P|\!\} \mid \Uparrow P \mid \Downarrow P$$

$$M[\![N]\!] \triangleq \mathsf{let}\ x = M\ ;\ x\,(\lambda\_\,.N)\,() \qquad \Downarrow P \triangleq \lambda f.\,\lambda x.\,\downarrow P\,f\,x$$

$$\{\!|P|\!\} \triangleq \lambda f.\,\{P\}\lambda x.\,f\,x \qquad \Uparrow P \triangleq \{\!|\mathit{amplify}(P)|\!\}\,[\![\lambda f.\,\lambda x.\,\uparrow P[f\,x]\,]\!]$$

One can easily verify that these derived forms behave as expected:

- $R \vdash \{\!|P|\!\}\,[\![N]\!] \to^* N$ if $\vdash R \geqslant P$, otherwise it fails.
- $R \vdash \{\!|P|\!\}\,\lambda x.M \to^* \{P\}\lambda y.\,\lambda x.M\ y \overset{\beta,\alpha}{=} \{P\}\lambda x.M$ (using β- and α-equivalence).
- $R \vdash \Uparrow P\,[\![N]\!] \to^* \uparrow P[N]$ if $\vdash R \geqslant \mathit{amplify}(P)$, otherwise it fails.
- $R \vdash \Downarrow P\,[\![N]\!] \to^* \downarrow P[N]$.
- $R \vdash \Downarrow P\,U\,V \to^* \downarrow P\,U\,V$.

We are now in a position to define user code.

[4] Write "$R \vdash M_0 \to^* M_n$" if there exist terms $M_i$ such that $R \vdash M_i \to M_{i+1}$, for all $i$ $(0 \leq i \leq n-1)$. Write "$R \vdash M \Downarrow U$" if $R \vdash M \to^* U$. Write "$R \vdash M \Downarrow$" if $R \vdash M \to^* U$ for some $U$. Write "$R \vdash M \Downarrow \mathsf{fail}$" if $R \vdash M \to^* M'$ where $R \vdash M' \not\to$ and $M'$ is not a value.

DEFINITION 3. A term is *user code* if occurrences of $\uparrow R[\cdot]$ only appear in subterms of the form $\Uparrow R$. □

Note that user code is not preserved by evaluation. Also note that any runtime term can be translated into a user code term by replacing $\uparrow$ with $\Uparrow$, $\downarrow$ with $\Downarrow$, $\{\ \}$ with $\lhd\ \rhd$, and $[]$ with $[\![]\!]$. This translation is also not preserved by evaluation.

## 3 Examples

EXAMPLE 4 (ACCESS CONTROL LISTS). Consider a web server that provides remote access to files protected by Access Control Lists (ACLs) at the filesystem layer. A read-only filesystem can be modelled as:

$$\begin{array}{l} \mathit{filesystem} \stackrel{\text{def}}{=} \\ \quad \lambda \mathit{name}. \\ \qquad \mathsf{if}\ \mathit{name} = \text{"file1"}\ \mathsf{then}\ \lhd \mathit{ADMIN} \rhd [\![\text{"content1"}]\!] \\ \qquad \mathsf{else\ if}\ \mathit{name} = \text{"file2"}\ \mathsf{then}\ \lhd \mathit{ALICE} \sqcap \mathit{BOB} \rhd [\![\text{"content2"}]\!] \\ \qquad \mathsf{else}\ \text{"error: file not found"} \end{array}$$

Assuming incomparable roles *ALICE*, *BOB*, and *CHARLIE* each strictly dominated by *ADMIN*, code running in the *ADMIN* role can access both files:

$$\begin{array}{l} \mathit{ADMIN} \vdash \mathit{filesystem}\ \text{"file1"} \rightarrow^* \lhd \mathit{ADMIN} \rhd [\![\text{"content1"}]\!] \rightarrow^* \text{"content1"} \\ \mathit{ADMIN} \vdash \mathit{filesystem}\ \text{"file2"} \rightarrow^* \lhd \mathit{ALICE} \sqcap \mathit{BOB} \rhd [\![\text{"content2"}]\!] \rightarrow^* \text{"content2"} \end{array}$$

Code running as *ALICE* or *BOB* cannot access the first file but can access the second:

$$\begin{array}{l} \mathit{ALICE} \vdash \mathit{filesystem}\ \text{"file1"} \rightarrow^* \lhd \mathit{ADMIN} \rhd [\![\text{"content1"}]\!] \Downarrow \mathsf{fail} \\ \mathit{BOB} \vdash \mathit{filesystem}\ \text{"file2"} \rightarrow^* \lhd \mathit{ALICE} \sqcap \mathit{BOB} \rhd [\![\text{"content2"}]\!] \rightarrow^* \text{"content2"} \end{array}$$

Finally, assuming that $\mathit{CHARLIE} \not\geq \mathit{ALICE} \sqcap \mathit{BOB}$, code running as *CHARLIE* cannot access either file:

$$\begin{array}{l} \mathit{CHARLIE} \vdash \mathit{filesystem}\ \text{"file1"} \rightarrow^* \lhd \mathit{ADMIN} \rhd [\![\text{"content1"}]\!] \Downarrow \mathsf{fail} \\ \mathit{CHARLIE} \vdash \mathit{filesystem}\ \text{"file2"} \rightarrow^* \lhd \mathit{ALICE} \sqcap \mathit{BOB} \rhd [\![\text{"content2"}]\!] \Downarrow \mathsf{fail} \end{array}$$

Now the web server can use the role assigned to a caller to access the filesystem (unless the web server's caller withholds their 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.

$$\begin{array}{l} \mathit{webserver} \stackrel{\text{def}}{=} \\ \quad \lambda \mathit{name}. \\ \qquad \mathsf{if}\ \mathit{name} = \text{"file1"}\ \mathsf{then}\ \mathit{filesystem}\ \mathit{name} \\ \qquad \mathsf{else\ if}\ \mathit{name} = \text{"file2"}\ \mathsf{then}\ \mathit{filesystem}\ \mathit{name} \\ \qquad \mathsf{else} \lhd \mathit{DEBUG} \rhd [\![\text{"error: file not found"}]\!] \end{array}$$

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

$$\begin{array}{l} \mathit{ALICE} \vdash \mathit{webserver}\ \text{"file2"} \\ \rightarrow^* \mathit{filesystem}\ \text{"file2"} \\ \rightarrow^* \lhd \mathit{ALICE} \sqcap \mathit{BOB} \rhd [\![\text{"content2"}]\!] \\ \rightarrow^* \text{"content2"} \end{array}$$

□

Sollins [34] describes an access control mechanism for distributed systems that "cooperate in the absence of complete trust of each other". Example 5 explains how the same concerns are addressed in λ-RBAC.

EXAMPLE 5 (TRAVEL AGENT - SIMPLIFIED FROM [34]). Suppose that company employees must purchase airline tickets from a travel agent. The travel agent must contact the accounting department at the same company to verify and receive payment. The accounts payable department at the company must check that the payment request received from the travel agent is a request from an employee of their own company.

This scenario can be modelled by requiring an employee, or programs running on their behalf, to delegate their role to the travel agent. The travel agent adds their own role before calling the accounts payable department, and the accounts payable department verifies both roles using the join, not the meet, of the *EMPL* and *AGENT* roles (we assume functions to calculate the cost of a ticket, issue a ticket, and authorize payment):

$$
\begin{array}{l}
\mathsf{let}\ \mathit{accountsPayable} = \Downarrow\!\bot\ [\![ \{\!|\ \mathit{EMPL} \sqcup \mathit{AGENT}\ |\!\}\ \lambda x.\mathit{authorizePayment}\ (\mathsf{snd}\ x)]\!]\ ; \\
\mathsf{let}\ \mathit{agent} = \Downarrow\!\mathit{amplify}(\mathit{AGENT})\ [\![ \\
\qquad\qquad \mathsf{let}\ t = \Uparrow\!\mathit{AGENT}\ ; \\
\qquad\qquad \lambda \mathit{destination}. \\
\qquad\qquad\quad \mathsf{let}\ \mathit{cost} = \mathit{calculateCost}\ \mathit{destination}\ ; \\
\qquad\qquad\quad \mathsf{let}\ \mathit{payment} = t\ [\![\mathit{accountsPayable}\ (\mathit{destination}, \mathit{cost})]\!]\ ; \\
\qquad\qquad\quad \mathit{issueTicket}\ \mathit{destination} \\
\qquad\qquad ]\!]\ ; \\
\Downarrow\!\mathit{EMPL}\ [\![\mathit{agent}\ \text{"New York"}]\!]
\end{array}
$$

Here $\mathit{amplify}(\cdot)$ controls rights amplification. When the entire program is executed in the top role, $\Uparrow\!\mathit{AGENT}$ runs with role context $\mathit{amplify}(\mathit{AGENT})$ during the definition of the travel agent. The result of running $\Uparrow\!\mathit{AGENT}$ is used to amplify the role when the employee, running with role context *EMPL*, subsequently calls the travel agent. □

*Subsequent examples are written directly in the runtime language for the sake of economy of mechanism.*

EXAMPLE 6 (MULTIPLE OWNERS). The Decentralized Label Model (DLM) [20] is an information-flow model that allows multiple principals to impose, and update, a policy on a piece of data. It is straightforward to program a comparable form of access control in λ-RBAC, where resources are protected by multiple access control lists, and the owner of an ACL may update their own ACL.

The ACLs are encoded as a single policy function of type $\mathsf{Unit} \to \mathsf{Unit}$ that is stored with each value, and the pair of the policy and value are protected by an additional access check to ensure exclusive access by privileged code, thus a protected value has type $\mathsf{Prot}(\sigma) \stackrel{\text{def}}{=} \mathsf{Unit} \to ((\mathsf{Unit} \to \mathsf{Unit}) \times \sigma)$, where $\sigma$ is the type of the data to be protected. A typical protected value has the form $\{\mathit{System}\}\lambda().(g, U)$ where the guard $g$ performs access checks against zero or more ACLs, each coded as a role as in example 4. Following the DLM each ACL is expected to have an owner, drawn from a family

of roles $Owner_1, \ldots, Owner_n$. Ownership is indicated by including the owner role in the ACL itself. For example, to check two ACLs $R_1 \leq Owner_1$ and $R_2 \leq Owner_2$ we use the guard:

$$\lambda().\{\!|R_1|\!\} \triangleright [\![\{\!|R_2|\!\} \triangleright [\![()]\!]]\!]$$

Protected values are accessed via a privileged function $access : \mathsf{Prot}(\sigma) \to \sigma$ that retrieves and checks the policy—representing the multiple ACLs—stored with the data:

$$\begin{array}{l} access \stackrel{\text{def}}{=} \lambda f : \mathsf{Prot}(\sigma). \\ \quad \mathsf{let}\ x : (\mathsf{Unit} \to \mathsf{Unit}) \times \sigma = \uparrow System[\downarrow_\perp f\ ()]\ ; \\ \quad (\mathsf{fst}\ x)\ ()\ ; \\ \quad \mathsf{snd}\ x \end{array}$$

Note that that value for $f$ need not be trustworthy from *access*'s point of view, and, consequently, the role *System* is not delegated to $f$. However, *access*'s caller must delegate its role to *access*, because the latter in turn must delegate the same role to the policy function. This is illustrated in the following trace where $S \geq R_1 \sqcup R_2$ and $V = \lambda().\{\!|R_1|\!\} \triangleright [\![\{\!|R_2|\!\} \triangleright [\![()]\!]]\!]$:

$$\begin{array}{l} S \vdash access\ (\{System\}\lambda().(V,U)) \\ \to \mathsf{let}\ x = \uparrow System[\downarrow_\perp (\{System\}\lambda().(V,U))\ ()]\ ;\ (\mathsf{fst}\ x)\ ()\ ;\ \mathsf{snd}\ x \\ \to \mathsf{let}\ x = \uparrow System[\downarrow_\perp [(V,U)]]\ ;\ (\mathsf{fst}\ x)\ ()\ ;\ \mathsf{snd}\ x \\ \to \mathsf{let}\ x = \uparrow System[(V,U)]\ ;\ (\mathsf{fst}\ x)\ ()\ ;\ \mathsf{snd}\ x \\ \to \mathsf{let}\ x = (V,U);\ (\mathsf{fst}\ x)\ ()\ ;\ \mathsf{snd}\ x \\ \to (\mathsf{fst}\ (V,U))\ ()\ ;\ \mathsf{snd}\ (V,U) \\ \to (\lambda().\{\!|R_1|\!\} \triangleright [\![\{\!|R_2|\!\} \triangleright [\![()]\!]]\!])\ ()\ ;\ \mathsf{snd}\ (V,U) \\ \to (\{\!|R_1|\!\} \triangleright [\![\{\!|R_2|\!\} \triangleright [\![()]\!]]\!])\ ;\ \mathsf{snd}\ (V,U) \\ \to^* (\{\!|R_2|\!\} \triangleright [\![()]\!])\ ;\ \mathsf{snd}\ (V,U) \\ \to^* ()\ ;\ \mathsf{snd}\ (V,U) \\ \to^* U \end{array}$$

Clearly, this system does not control information flow because a protected value can be accessed according to the policy and then left unprotected. Moreover, it is not possible for an owner to modify their own ACL without extracting the protected value and creating a new protected resource—that will not possess the ACLs of other owners. As an alternative, the privileged $declassify_i : \mathsf{Prot}(\sigma) \to \mathsf{Prot}(\sigma)$ functions allow an owner to remove their own ACL without disturbing other ACLs on the protected value:

$$\begin{array}{l} declassify_i \stackrel{\text{def}}{=} \{Owner_i\}\lambda f : \mathsf{Prot}(\sigma). \\ \quad \mathsf{let}\ x : (\mathsf{Unit} \to \mathsf{Unit}) \times \sigma = \uparrow System[\downarrow_\perp f\ ()]\ ; \\ \quad \mathsf{let}\ g : \mathsf{Unit} \to \mathsf{Unit} = \lambda().\uparrow Owner_i[(\mathsf{fst}\ x)\ ()]\ ; \\ \quad \{System\}\lambda().(g, \mathsf{snd}\ x) \end{array}$$

The function $declassify_i$ replaces the existing policy function, adding $Owner_i$ before the policy is checked which overrides ACLs owned by $Owner_i$. □

Example 7 illustrates how the Domain-Type Enforcement (DTE) security mechanism [5,35], as found in the NSA's Security-Enhanced Linux (SELinux) [17], can be implemented in λ-RBAC. Further discussion of the relationship between RBAC and DTE can be found in [12,16].

EXAMPLE 7 (DOMAIN-TYPE ENFORCEMENT / SELINUX). DTE grants or denies access requests according to the *domain* of the requesting process and the *type* assigned to the object, e.g., a file or port. The domain of a process may only change when another image is executed. DTE facilitates use of least privilege by limiting domain transitions based upon the source domain, target domain, and type assigned to the invoked executable file. The DTE domain transition from role $R$ to role $S$ (each acting as domains) can be modelled by the function $R \xrightarrow{E} S$ that allows code running at role $R$ to apply a function at role $S$:

$$R \xrightarrow{E} S \stackrel{\text{def}}{=} \{R\}\lambda(f,x).\downarrow\!\perp f\ (\{E\}\lambda g.\uparrow\! S[g\ x])$$

However, the domain transition is only performed when the function is associated with role $E$, modelling assignment of DTE type $E$ to an executable file. Association of a function $g$ with role $E$ is achieved by accepting a continuation that is called back at role $E$ with the function $g$. The function $assignType_E$ allows code running at $ADMIN$ to assign DTE types to other code:

$$assignType_E \stackrel{\text{def}}{=} \{ADMIN\}\lambda g.\,\lambda h.\uparrow\! E\,[\downarrow\!\perp h\ g]$$

For example, for a function value $U$:

$$ADMIN \vdash \downarrow\!\perp assignType_E\ U \rightarrow^* \lambda h.\uparrow\! E\,[\downarrow\!\perp h\ U]$$

Then given a value $V$ such that $S \vdash U\ V \rightarrow^* W$ we have:

$$R \vdash \downarrow\!\perp (R \xrightarrow{E} S)\ (\lambda h.\uparrow\! E\,[\downarrow\!\perp h\ U], V) \rightarrow^* W$$

With the $R \xrightarrow{E} S$ and $assignType_E$ functions we can adapt the login example from [35] to λ-RBAC. In this example, the DTE mechanism is used to force every invocation of administrative code (running at $ADMIN$) from daemon code (running at $DAEMON$) to occur via trusted login code (running at $LOGIN$). This is achieved by providing domain transitions from $DAEMON$ to $LOGIN$, and $LOGIN$ to $ADMIN$, but no others. Moreover, code permitted to run at $LOGIN$ must be assigned DTE type $LOGINEXE$, and similarly for $ADMIN$ and $ADMINEXE$. Thus a full program running daemon code $M$ has the following form, where neither $M$ nor the code assigned to $g$ variables contain rights amplification:

$$\begin{array}{l}
\mathsf{let}\ daemonToLogin = DAEMON \xrightarrow{LOGINEXE} LOGIN\,; \\
\mathsf{let}\ loginToAdmin = LOGIN \xrightarrow{ADMINEXE} ADMIN\,; \\
\mathsf{let}\ shell = \mathsf{let}\ g = \ldots\,;\ \downarrow\!\perp assignType_{ADMINEXE}\ (g)\,; \\
\mathsf{let}\ login = \mathsf{let}\ g = \lambda(password, cmd). \\
\qquad\qquad \mathsf{if}\ password = \text{"secret"}\ \mathsf{then} \\
\qquad\qquad\quad \downarrow\!\perp loginToAdmin\ (shell, cmd) \\
\qquad\qquad \mathsf{else} \\
\qquad\qquad\quad \ldots\,; \\
\qquad\quad \downarrow\!\perp assignType_{LOGINEXE}\ (g)\,; \\
\downarrow DAEMON\,[M]
\end{array}$$

In the above program, the daemon code $M$ must provide the correct password in order to execute the shell at *ADMIN* because the *login* provides the sole gateway to *ADMIN*. In addition, removal of the domain transition *daemonToLogin* makes it impossible for the daemon code to execute any code at *ADMIN*. □

## 4 Statics

We consider two kinds of *static analysis*: (i) a type system to enable removal of unnecessary role-checks in a piece of code for a caller at a sufficiently high role, and (ii) a type system to determine the amount of protection that is enforced by the callee.

To make the issues as clear as possible, we treat a simply-typed calculus with subtyping in the main text; we discuss bounded role polymorphism in Appendix B. Throughout the rest of this section, we assume that all roles (and therefore all types) are well-formed, in the sense that role constructors have the correct number of arguments (for a more explicit treatment, see the appendix).

In this section, we only consider toy examples. In section 5 we revisit types, in both type systems, for the examples from section 3.

### 4.1 A type system to calculate caller roles

Values require no computation to evaluate, thus the value judgment $\Gamma \vdash U : \tau$ includes only the type $\tau$. The judgment for terms $\Gamma \vdash M : \{R\}\ \tau$ does include an effect $R$, indicating that $R$ is sufficient to evaluate $M$ without role errors, i.e., EVAL-APP will always succeed.

The type language includes base types (which we elide in the formal presentation) and function types. The function type $\sigma \to \{Q \triangleright R\}\ \tau$ is decorated with two latent effects; roughly, these indicate that the role context must dominate $Q$ in order to pass the function's guard, and that the caller must provide a role context of at least $R$ for execution of the function body to succeed. The least role $\bot$ is trivial in function types; thus $\sigma \to \tau$ abbreviates $\sigma \to \{\bot \triangleright \bot\}\ \tau$. We also write $\sigma \to \{R\}\ \tau$ for $\sigma \to \{\bot \triangleright R\}\ \tau$. If all roles occurring in a term are trivial, our typing rules degenerate to those of the standard simply-typed lambda calculus.

TYPES

| | |
|---|---|
| $\sigma, \tau ::= \cdots$ | *Types* |
| $\quad \sigma \to \{Q \triangleright R\}\ \tau$ | *value abstraction* |
| $\Gamma, \Delta ::= x_1 : \sigma_1, \ldots, x_n : \sigma_n$ | *Environments* |

VALUE AND TERM TYPING $(\Gamma \vdash U : \tau)$ $(\Gamma \vdash M : \{R\}\ \tau)$

$$\frac{}{\Gamma \vdash x : \tau}\ \text{(VAL-VAR)}\ \Gamma(x) = \tau \qquad \frac{\Gamma, x : \sigma \vdash M : \{R\}\ \tau}{\Gamma \vdash \{Q\}\lambda x.M : \sigma \to \{Q \triangleright R\}\ \tau}\ \text{(VAL-ABS)} \qquad \frac{\Gamma \vdash U : \tau}{\Gamma \vdash U : \{\bot\}\ \tau}\ \text{(TERM-VAL)} \qquad \frac{\Gamma \vdash M : \{R\}\ \sigma \quad \Gamma, x : \sigma \vdash N : \{S\}\ \tau}{\Gamma \vdash \mathsf{let}\ x = M\ ;\ N : \{R \sqcup S\}\ \tau}\ \text{(TERM-LET)}$$

(TERM-APP)
$$\frac{\Gamma \vdash U : \sigma \to \{Q \triangleright R \sqcap P\}\ \tau \quad \Gamma \vdash V : \sigma}{\Gamma \vdash \downarrow P\ U\ V : \{Q \sqcup R\}\ \tau}$$

(TERM-RESTRICT)
$$\frac{\Gamma \vdash M : \{R \sqcap P\}\ \tau}{\Gamma \vdash \downarrow P[M] : \{R\}\ \tau}$$

(TERM-PROVIDE)
$$\frac{\Gamma \vdash M : \{R \sqcup P\}\ \tau}{\Gamma \vdash \uparrow P[M] : \{R\}\ \tau}$$

(TERM-SUBEFFECT)
$$\frac{\Gamma \vdash M : \{S\}\ \tau \quad \vdash R \geqslant S}{\Gamma \vdash M : \{R\}\ \tau}$$

VAL-ABS simply records the effects that the abstraction will incur when run. By TERM-VAL, a value can be treated as a term that evaluates without error in every role context. TERM-LET indicates that two expressions must succeed sequentially if the current role guarantees success of each individually. TERM-RESTRICT (resp. TERM-PROVIDE) captures the associated rights weakening (resp. amplification). TERM-APP incorporates the role required to evaluate the function to an abstraction and the role required to evaluate the body of the function (while allowing for rights weakening).

**Subtyping.** A natural notion of subtyping is induced from the role ordering. Formally, subtyping is the least precongruence on types induced by SUBTYPING-BASE. There are subsumption rules for values and terms.

SUBTYPING ($\vdash \sigma <: \sigma'$)

(SUBTYPING-BASE)
$$\frac{\vdash \sigma' <: \sigma \quad \vdash Q' \geqslant Q \quad \vdash S' \geqslant S \quad \vdash \tau <: \tau'}{\vdash (\sigma \to \{Q \triangleright S\}\ \tau) <: (\sigma' \to \{Q' \triangleright S'\}\ \tau')}$$

(VAL-SUBTYPE)
$$\frac{\Gamma \vdash U : \sigma \quad \vdash \sigma <: \sigma'}{\Gamma \vdash U : \sigma'}$$

(TERM-SUBTYPE)
$$\frac{\Gamma \vdash M : \{R\}\ \tau \quad \vdash \tau <: \tau'}{\Gamma \vdash M : \{R\}\ \tau'}$$

The following example of Church booleans, illustrates the typing system. The Church booleans, $\lambda t.\,\lambda f.t$ and $\lambda t.\,\lambda f.f$ can be given type $(\sigma \to \{R\}\ \tau) \to (\sigma \to \{S\}\ \tau) \to (\sigma \to \{R \sqcup S\}\ \tau)$. The type system satisfies standard preservation and progress properties.

THEOREM 8. *If $\Gamma \vdash M : \{R\}\ \tau$ and $S \vdash M \to M'$, then $\Gamma \vdash M' : \{R\}\ \tau$.*
*If $\Gamma \vdash M : \{R\}\ \tau$ then either $M$ is a value, or $R \vdash M \to M'$, for some $M'$.*

**Algorithmic version.** In order to facilitate bottom-up deduction and lead into the next type system, we now describe an alternate algorithmic presentation of the type system. Here we insist that the role lattice is Boolean, i.e., a distributive lattice with a complement $R^\star$ for every role $R$, where $R$ and $S$ are complements if $R \sqcap S = \bot$ and $R \sqcup S = \top$.

ALGORITHMIC VERSION (VAL-VAR, VAL-ABS, TERM-VAL, TERM-LET AS BEFORE)

$$\frac{\Gamma \vdash^{\mathbb{A}} U : \sigma \to \{Q \triangleright R\}\ \tau \quad \Gamma \vdash^{\mathbb{A}} V : \sigma' \quad \vdash \sigma' <: \sigma \quad \vdash P \geqslant R}{\Gamma \vdash^{\mathbb{A}} \downarrow P\ U\ V : \{Q \sqcup R\}\ \tau}$$

$$\frac{\Gamma \vdash^{\mathbb{A}} M : \{R\}\ \tau \quad \vdash P \geqslant R}{\Gamma \vdash^{\mathbb{A}} \downarrow P[M] : \{R\}\ \tau}$$

$$\frac{\Gamma \vdash^{\mathbb{A}} M : \{R\}\ \tau}{\Gamma \vdash^{\mathbb{A}} \uparrow P[M] : \{R \sqcap P^\star\}\ \tau}$$

PROPOSITION 9. *If $\Gamma \vdash^{\mathbb{A}} M : \{R\}\ \tau$ then $\Gamma \vdash M : \{R\}\ \tau$.*
*If $\Gamma \vdash M : \{R\}\ \tau$ then $\Gamma \vdash^{\mathbb{A}} M : \{R'\}\ \tau'$ for some $R'$ and $\tau'$ such that $\vdash R \geqslant R'$ and $\vdash \tau' <: \tau$.*

### 4.2 A type system to determine callee protection

Our second type system has aims "dual" to the previous type system. Rather than attempting to calculate a caller role that guarantees that all execution paths are successful, we now calculate the minimum protection demanded by the callee on all execution paths.

Formally, we present the type system by building on the algorithmic presentation. We use the algorithmic system, with the following changes: we use altered versions of TERM-APP, TERM-RESTRICT and inverted versions of TERM-SUBEFFECT and SUBTYPING-BASE, with the resulting system denoted using "$\Vdash$".

TYPING FOR CALLEE PROTECTION. OTHER RULES FROM ALGORITHMIC VERSION

$$\frac{\Gamma \Vdash M : \{S\}\ \tau \quad \vdash S \geqslant R}{\Gamma \Vdash M : \{R\}\ \tau} \qquad \frac{\Vdash \sigma' <: \sigma \quad \vdash Q \geqslant Q' \quad \vdash S \geqslant S' \quad \Vdash \tau <: \tau'}{\Vdash (\sigma \to \{Q \triangleright S\}\ \tau) <: (\sigma' \to \{Q' \triangleright S'\}\ \tau')}$$

(TERM-APP)

$$\frac{\Gamma \Vdash U : \sigma \to \{Q \triangleright R\}\ \tau \quad \Gamma \Vdash V : \sigma' \quad \vdash \sigma' <: \sigma}{\Gamma \Vdash \downarrow P\ U\ V : \{Q \sqcup R\}\ \tau}$$

(TERM-RESTRICT)

$$\frac{\Gamma \Vdash M : \{R\}\ \tau}{\Gamma \Vdash \downarrow P\,[M] : \{R\}\ \tau}$$

In the new system, the Church booleans may be given type: $(\sigma \to \{R\}\ \tau) \to (\sigma \to \{S\}\ \tau) \to (\sigma \to \{R \sqcap S\}\ \tau)$. This type illustrates the "minimum over all paths" principle via $R \sqcap S$ (to be contrasted with $R \sqcup S$ in the previous typing system.)

The following lemma captures the idea that values are already in normal form, so do not enforce any protection.

LEMMA 10. *$\Vdash U : \{S\}\ \tau$ implies $S = \bot$.*

Unsurprisingly, considering the motivation behind this type system, the standard form of the type preservation result does not hold for this system. For example $\Vdash (\{\top\}\lambda\_.()) \ () : \{\top\}$ Unit and $\top \vdash (\{\top\}\lambda\_.()) \ () \to ()$ but $\not\Vdash () : \{\top\}$ Unit. The following theorem captures the invariants preserved by reduction.

THEOREM 11. *If $\Gamma \Vdash M : \{S\}\ \tau$ and $\vdash R \not\geqslant S$ and $R \vdash M \to M'$, then $\Gamma \Vdash M' : \{S\}\ \tau$.*

In combination with lemma 10, theorem 11 yields that if $\Gamma \Vdash M : \{S\}\ \tau$, then it is not possible for $M$ to evaluate to a value without using a role above $S$. We see this as follows. If we start execution in a role context ($R$ in the theorem statement) that does not suffice to pass the minimum protection guarantee ($S$ in the theorem statement), then a single step of reduction has only two possibilities: (i) a check, e.g., for $S$, that is not passed by $R$ occurs and the term gets stuck, or (ii) the check for $S$ does not happen at this step but the invariant that the minimum protection is $S$ continues to get preserved.

## 5 Typing Examples

EXAMPLE 12. Recall the filesystem and web server from example 4. 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:

$$\vdash \mathit{filesystem} : \mathsf{String} \to \{\bot \triangleright \mathit{ADMIN} \sqcup (\mathit{ALICE} \sqcap \mathit{BOB}) \sqcup \bot\}\ \mathsf{String}$$

In the above type, the final role $\bot$ arises from the "unknown file" branch that does not require an access check. The lack of an access check explains the weaker type in the dual system:

$$\Vdash \mathit{filesystem} : \mathsf{String} \rightarrow\{\bot \triangleright \mathit{ADMIN} \sqcap (\mathit{ALICE} \sqcap \mathit{BOB}) \sqcap \bot\}\ \mathsf{String}$$

This type indicates that *filesystem* has the potential to expose some information to unprivileged callers with role $\mathit{ADMIN} \sqcap (\mathit{ALICE} \sqcap \mathit{BOB}) \sqcap \bot = \bot$, perhaps causing the code to be flagged for security review.

The access check in the web server does prevent the "unknown file" error message leaking unless the *DEBUG* role is active, but, unfortunately, it is not possible to assign a role strictly greater than $\bot$ to the web server using the second type system because the *filesystem* type does not record the different roles that must be checked depending upon the filename argument, and hence:

$$\nVdash \mathit{webserver} : \mathsf{String} \rightarrow\{\bot \triangleright \mathit{ADMIN} \sqcap (\mathit{ALICE} \sqcap \mathit{BOB}) \sqcap \mathit{DEBUG}\}\ \mathsf{String} \qquad \square$$

EXAMPLE 13. Unlike example 12, the DLM-inspired operations on values protected by ACLs with different owners in example 6 can be assigned the same type in both type systems. First, define the type of values protected by role $R$:

$$\mathsf{Prot}(R,\sigma) \stackrel{\text{def}}{=} \mathsf{Unit} \rightarrow\{\mathit{System} \triangleright \bot\}\ ((\mathsf{Unit} \rightarrow\{R\}\ \mathsf{Unit}) \times \sigma)$$

Now, for any role $R$, we have:

$$\vdash \mathit{access} : \mathsf{Prot}(R,\sigma) \rightarrow\{R\}\ \sigma$$
$$\Vdash \mathit{access} : \mathsf{Prot}(R,\sigma) \rightarrow\{R\}\ \sigma$$

The first type assignment ensures that the protected value can be accessed at role $R$, on the assumption that access to the pair is possible with role *System* and the guard function with role $R$. In contrast, the second type assignment ensures that access to the protected value is guaranteed to require $R$, on the assumption that the guard function is guaranteed to require $R$.

The declassification operation that removes an owner's ACL from the guard can be assigned the following type:

$$\vdash \mathit{declassify}_i : \mathsf{Prot}(R \sqcup \mathit{Owner}_i,\sigma) \rightarrow\{\mathit{Owner}_i \triangleright \bot\}\ \mathsf{Prot}(R,\sigma)$$

Thus, partitioning the role required to pass a protected value's guard as $R \sqcup \mathit{Owner}_i$ ensures that the declassified protected value can be accessed with $R$. In particular, note that an ACL owned by $\mathit{Owner}_i$ with the form $S = S_1 \sqcap S_2 \sqcap \ldots \sqcap S_n \sqcap \mathit{Owner}_i$ satisfies $R \sqcup S \leq R \sqcup \mathit{Owner}_i$ and so $\vdash \mathsf{Prot}(R \sqcup S,\sigma) <: \mathsf{Prot}(R \sqcup \mathit{Owner}_i,\sigma)$.

The second type system can assign the same type to declassification if the role lattice is Boolean and satisfies $R \sqcap \mathit{Owner}_i = \bot$:

$$\Vdash \mathit{declassify}_i : \mathsf{Prot}(R \sqcup \mathit{Owner}_i,\sigma) \rightarrow\{\mathit{Owner}_i \triangleright \bot\}\ \mathsf{Prot}(R,\sigma)$$

The condition $R \sqcap \mathit{Owner}_i = \bot$ ensures that guaranteed checks against $R$ are not negated by the addition of $\mathit{Owner}_i$ to the role context at runtime. For example, the type assignment would clearly be incorrect when $R = \mathit{Owner}_i$.

Subtyping in the second type system makes the above type for declassification less useful. Fortunately, a much more useful type can be assigned for all roles $R$ and $S$ such that $R \sqcap Owner_i = \bot$:

$$\Vdash declassify_i : \mathsf{Prot}(R \sqcup (S \sqcap Owner_i), \sigma) \rightarrow\{Owner_i \triangleright \bot\} \mathsf{Prot}(R, \sigma)$$

Here the role $S$ corresponds to other entries in the ACL owned by $Owner_i$. □

EXAMPLE 14. Recall the encoding of the DTE/SELinux domain transition mechanism from example 7. Define types for functions running at role $S$ (acting as a domain) and functions that can prove their assigned DTE type[5] is $E$ by calling back with that role:

$$\mathsf{Func}(\sigma, \tau, S) \stackrel{\text{def}}{=} \sigma \rightarrow\{S\}\ \tau$$
$$\mathsf{FuncDTEType}(\sigma, \tau, S, E) \stackrel{\text{def}}{=} (\mathsf{Func}(\sigma, \tau, S) \rightarrow\{E \triangleright \bot\}\ \tau) \rightarrow\{\bot\}\ \tau$$

A domain transition will certainly succeed if the caller possesses role $R$ and the function invoked after the domain transition requires at most role $S$:

$$\vdash R \xrightarrow{E} S : \mathsf{FuncDTEType}(\sigma, \tau, S, E) \times \sigma \rightarrow\{R \triangleright \bot\}\ \tau$$

In contrast, the following type guarantees that role $R$ will be demanded from the caller:

$$\Vdash R \xrightarrow{E} S : \mathsf{FuncDTEType}(\sigma, \tau, S, E) \times \sigma \rightarrow\{R \triangleright \bot\}\ \tau$$

Similarly, the function that assigns DTE types has the same λ-RBAC type in both systems.

$$\vdash assignType_E : \mathsf{Func}(\sigma, \tau, S) \rightarrow\{ADMIN \triangleright \bot\}\ \mathsf{FuncDTEType}(\sigma, \tau, S, E)$$
$$\Vdash assignType_E : \mathsf{Func}(\sigma, \tau, S) \rightarrow\{ADMIN \triangleright \bot\}\ \mathsf{FuncDTEType}(\sigma, \tau, S, E)$$ □

## 6 Related work

We have already referred to several related pieces of research in the earlier text and examples.

Recently, there has been renewed interest in the subject of access control — both formal, e.g., see [1] for a broad survey of literature in logical methods in access control; and pragmatic, e.g., addressing the use of software components by requiring that access control take into account the history of execution, either the entire execution history (e.g., [2]) or under the discipline of stack inspection (e.g., [36,14]) where the history is restricted to all unfinished method calls

Role-based access control introduces a level of indirection between subjects and objects [29,13]. Usage control [30] provides a unified framework encompassing RBAC and trust–management systems, in part by incorporating history-sensitive ideas into the RBAC model.

[5] Recall that DTE types are modelled as roles, and should not be confused with λ-RBAC types.

The papers that are most directly relevant to to the current paper are [6,10]. 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. Static type systems are used to provide guarantees about the minimal role required for execution to be successful — our first type system occupies the same conceptual space as the static analysis in these papers.

In contrast to this paper, the underlying computation model is more expressive in the above papers. 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 [6]) to enforce control on rights activation.

In future work, we hope to integrate into λ-RBAC the powerful bisimulation principles that are explored in these papers.

Our paper deals with access control, so the work on information flow, e.g., see [26] 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 [9,27,21] plays in the context of information flow; namely that of permitting access that would not have been possible otherwise.

## References


1. Martín Abadi. Logic in access control. In *LICS*, pages 228–233. IEEE Computer Society, 2003.
2. Martín Abadi and Cedric Fournet. Access control based on execution history. In *Proceedings of the Network and Distributed System Security Symposium Conference*, 2003.
3. Martín Abadi, Greg Morrisett, and Andrei Sabelfeld. Language-based security. *J. Funct. Program.*, 15(2):129, 2005.
4. Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM TOPLAS*, 15(4):575–631, 1993.
5. W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the Eighth National Computer Security Conference*, 1985.
6. Chiara Braghin, Daniele Gorla, and Vladimiro Sassone. A distributed calculus for role-based access control. In Computer Security Foundations Workshop [11], pages 48–60.
7. Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundam. Inf.*, 33(4):309–338, 1998.
8. Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *Proceedings of the 11th USENIX Security Symposium*, 2002.
9. Stephen Chong and Andrew C. Myers. Security policies for downgrading. In Vijayalakshmi Atluri, Birgit Pfitzmann, and Patrick McDaniel, editors, *ACM Conference on Computer and Communications Security*, pages 198–209. ACM, 2004.
10. Adriana Compagnoni, Pablo Garralda, and Elsa Gunter. Role-based access control in a mobile environment. In *Symposium on Trustworthy Global Computing*, 2005. To appear in LNCS.

11. *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*. IEEE Computer Society, 2004.
12. David F. Ferraiolo, D. Richard Kuhn, and Ramaswamy Chandramouli. *Role-Based Access Control*. Computer Security Series. Artech House, 2003.
13. David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, 4(3):224–274, 2001.
14. Cédric Fournet and Andrew D. Gordon. Stack inspection: Theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, May 2003.
15. Robert M. Graham. Protection in an information processing utility. *Communications of the ACM*, 11(5):365–369, May 1968.
16. J. Hoffman. Implementing RBAC on a type enforced system. In *13th Annual Computer Security Applications Conference (ACSAC '97)*, pages 158–163, 1997.
17. P. A. Loscocco and S. D. Smalley. Meeting critical security objectives with Security-Enhanced Linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, 2001.
18. Surbhi Malhotra. *Microsoft .NET Framework Security*. Premier Press, 2002.
19. John C. Mitchell. Programming language methods in computer security. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–26. ACM Press, 2001.
20. Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.
21. Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification. In Computer Security Foundations Workshop [11], pages 172–186.
22. Elliott Organick. *The Multics System: An Examination of its Structure*. MIT Press, 1972.
23. Sylvia Osborn, Ravi Sandhu, and Qamar Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Trans. Inf. Syst. Secur.*, 3(2):85–106, 2000.
24. Joon S. Park, Ravi S. Sandhu, and Gail-Joon Ahn. Role-based access control on the web. *ACM Trans. Inf. Syst. Secur.*, 4(1):37–71, 2001.
25. Dennis M. Ritchie. Protection of data file contents. Technical Report United States Patent 4,135,240, United States Patent and Trademark Office, 1979.
26. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
27. Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In Kokichi Futatsugi, Fumio Mizoguchi, and Naoki Yonezaki, editors, *ISSS*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191. Springer, 2003.
28. J. Saltzer and M. Schroeder. The protection of information in computer systems. In *IEEE*, volume 9(63), 1975.
29. Ravi Sandhu, Edward Coyne, Hal Feinstein, and Charles Youman. Role-based access control models. *IEEE Computer*, 29(2), 1996.
30. Ravi S. Sandhu and Jaehong Park. Usage control: A vision for next generation access control. *ACM Trans. Inf. Syst. Secur.*, 2004. To appear.
31. 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. Springer-Verlag, 2000.
32. Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. *Communications of the ACM*, 15(3):157–170, March 1972.
33. Solaris 10 System Administration Guide: Security Services, January 2005. http://docs-pdf.sun.com/816-4557/816-4557.pdf.

34. Karen R. Sollins. Cascaded authentication. In *IEEE Symposium on Security and Privacy*, 1988.
35. Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L. Shermann, and Karen A. Oostendorp. Confining root programs with Domain and Type Enforcement (DTE). In *Proceedings of the Sixth USENIX UNIX Security Symposium*, 1996.
36. Dan S. Wallach, Andrew W. Appel, and Edward W. Felten. SAFKASI: a security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):341–378, October 2000.

## A Examples from .NET

What expressiveness is required of a programming language framework that supports RBAC? In a sequence of .NET examples[6], closely based on [18], we give the reader a flavor of the basic programming idioms.

EXAMPLE 15 ( [18]). In the .NET Framework CLR, every thread has a Principal object that carries its role. 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 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, the PrincipalPermission objects are not created and a security exception is raised. □

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

EXAMPLE 16 ( [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 =
```

[6] 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.

```
    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. □

The key operation in such programming is *rights modulation*. Rights modulation of course comes in two flavors: rights weakening is overall a safe operation, since the caller choses 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 17. From a programming viewpoint, it is convenient, indeed sometimes required, for an application to operate under the guise of different users at different times. In the .NET framework, this is called impersonation. Programmatically, impersonation of an account is achieved by retrieving and using the account's token, as done by 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
```

□

## B Extensions of Typing

We can formalize the assumption that all roles are well-formed by incorporating this requirement into the type system. To achieve this, we must provide definitions for good roles, types and environments.

GOOD ENVIRONMENT $(\Gamma \vdash \diamond)$ $(\Gamma \vdash R)$ $(\Gamma \vdash \sigma)$

$$\frac{}{\emptyset \vdash \diamond}\ (\text{ENV-EMPTY}) \qquad \frac{\Gamma \vdash \diamond \quad \Gamma \vdash \sigma \quad x \notin dom(\Gamma)}{\Gamma, x:\sigma \vdash \diamond}\ (\text{ENV-VAR}) \qquad \frac{\mathsf{arity}(\kappa) = n \quad \Gamma \vdash R_i \ ^{(\forall i \in \{1,\ldots,n\})}}{\Gamma \vdash \kappa(R_1,\ldots,R_n)}\ (\text{ROLE-CONSTR})$$

$$\frac{\Gamma \vdash \sigma \quad \Gamma \vdash Q \quad \Gamma \vdash \tau \quad \Gamma \vdash S}{\Gamma \vdash \sigma \to \{Q \triangleright S\}\ \tau}\ (\text{TYPE-ABS}) \qquad \frac{\sigma \text{ is a base type}}{\Gamma \vdash \sigma}\ (\text{TYPE-BASE})$$

We must then back-patch the well-formedness requirement into the typing system. We require that each synactically mentioned role be well formed. We must also insist that the environment be well formed; this is achieved by requiring that environments be well-formed (written $\Gamma \vdash \diamond$) for any base term. For example, the rules for variables, unit values and abstractions become the following.

$$\frac{\Gamma \vdash \diamond \quad \Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash () : \mathsf{Unit}} \qquad \frac{\Gamma \vdash Q \qquad \Gamma, x:\sigma \vdash M : \{S\}\ \tau}{\Gamma \vdash \{Q\}\lambda x.M : \sigma \to \{Q \triangleright S\}\ \tau}$$

Role parametricity can be added as follows.

ROLE PARAMETRICITY — SYNTAX AND EVALUATION

| | |
|---|---|
| $p,q,r,s$ | Role Variables |
| $P,Q,R,S ::= \cdots \mid r$ | Roles |
| $\sigma,\tau ::= \cdots \mid \forall p \geqslant P.\tau$ | Types |
| $\Gamma,\Delta ::= \cdots \mid \Gamma, p \geqslant P$ | Environments |
| $U,V ::= \cdots \mid \Lambda r.\, U$ | Values |
| $M,N,L ::= \cdots \mid U\ \langle R \rangle$ | Terms |

$$\frac{}{R \vdash (\Lambda s.\, M)\ \langle S \rangle \longrightarrow M[S/s]}\ \text{(EVAL-ROLE-APP)}$$

Typing requires that the role order be lifted to *open* roles (those with free occurences of role variables). We require that the resulting relation, $\Gamma \vdash S \geqslant R$, satisfies the following (where all roles, types and environments in the antecedents are assumed to be well formed):

(WEAKENING) If $\Gamma \vdash S \geqslant R$ then $\Gamma, p \geqslant P \vdash S \geqslant R$.
(BOUND WEAKENING) If $\Gamma, p \geqslant P \vdash S \geqslant R$ and $\Gamma \vdash Q \geqslant P$ then $\Gamma, p \geqslant Q \vdash S \geqslant R$.
(SUBSTITUTIVITY) If $\Gamma, p \geqslant P \vdash S \geqslant R$ and $\Gamma \vdash Q \geqslant P$ then $\Gamma \vdash S[Q/p] \geqslant R[Q/p]$.

ROLE PARAMETRICITY — STATICS

$$\frac{p \in dom(\Gamma)}{\Gamma \vdash p}\ \text{(ROLE-VAR)} \qquad \frac{\Gamma, p \geqslant P \vdash \tau}{\Gamma \vdash \forall p \geqslant P.\tau}\ \text{(TYPE-ROLE-ABS)}$$

$$\frac{\Gamma \vdash P' \geqslant P \qquad \Gamma, p \geqslant P' \vdash \tau <: \tau'}{\Gamma \vdash (\forall p \geqslant P\ .\tau\ ) <: (\forall p \geqslant P'.\tau')}\ \text{(SUBTYPE-ROLE-ABS)}$$

$$\frac{\Gamma \vdash \diamond \quad \Gamma \vdash P \qquad p \notin dom(\Gamma)}{\Gamma, p \geqslant P \vdash \diamond}\ \text{(ENV-ROLE-VAR)}$$

$$\frac{\Gamma \vdash P \qquad \Gamma, p \geqslant P \vdash U : \tau}{\Gamma \vdash \Lambda p.\, U : \forall p \geqslant P.\tau}\ \text{(VAL-ROLE-ABS)}$$

$$\frac{\Gamma \vdash U : \forall p \geqslant P.\tau \qquad \Gamma \vdash Q \quad \Gamma \vdash Q \geqslant P}{\Gamma \vdash U\ \langle Q \rangle : \{\bot\}\ \sigma}\ \text{(TERM-ROLE-APP)}$$

Following [7,4], we can also add equirecursive types.

## C Multics

EXAMPLE 18 (MULTICS). The Multics system [22,15,32] uses *protection rings*, or rings for short, to protect interacting but mutually suspicious subsystems. Each data segment is protected with an access bracket $(a,b)$, where $a \leq b$ are ring numbers that control access to the data segment, and ring 0 is the most privileged ring. A process running in ring $i$ can write to the data segment if $0 \leq i \leq a$, and can read from the data segment if $a \leq i \leq b$. Each procedure segment is protected with an access and call bracket $(a,b,c)$. Suppose a process running in ring $i$ attempts to invoke such a procedure:

**Outward Call** If $0 \leq i < a$ then a ring-crossing fault is generated. The procedure will execute in ring $a$, reducing privileges.

**Regular Call** If $a \leq i \leq b$ then the procedure will execute. The procedure will execute in ring $i$.

**Inward Call** If $b < i \leq c$ and the procedure segment is marked as a gate (a designated entrypoint for less privileged callers) then the procedure will execute. The procedure will execute in ring $b$, increasing privileges.

**Failed Call** If $b < i \leq c$ and the procedure segment is not a gate, or if $c < i$, then the invocation fails.

To implement rings in λ-RBAC, take the role lattice to be the free Boolean lattice over a set that includes ring numbers $\{0,\ldots,MAX\}$. Although lower-numbered rings are considered more privileged than higher-numbered rings, this is not reflected in the role order because of the need to differentiate outward calls—that cause ring-crossing faults in Multics—from regular calls, and hence roles representing different ring numbers are incomparable. Programs use trusted code to invoke other procedures (functions representing procedure segments). The trusted invocation code modifies the ring if necessary, and maintains the invariant that code runs with no more than one ring. For example, an inward call from code running in ring $i$ will cause the trusted invocation code to switch from ring $i$ to ring $b$ using $(\downarrow\neg i[\uparrow b[\ldots]])$. In order to perform the correct test, and ring change if necessary, the trusted invocation code must know the caller's ring. In this implementation, the current ring number is passed as an integer between function calls. The trusted invocation code can both branch upon the current ring number and check that a claimed current ring number matches with the ring number determined by the role.

As an example, consider a function *proc* acting as a gate of type $\mathsf{Int} \times \sigma \to \tau$, where the integer component of the argument is expected to be the ring number in which the function is running. To assign *proc* the access and call bracket $(a,b,c)$, add trusted invocation code to form a new function *PROC*:

$$
\begin{array}{l}
PROC \stackrel{\text{def}}{=} \\
\quad \lambda r : \mathsf{Int}. \\
\qquad \mathsf{if}\ r = 0\ \mathsf{then}\ AUX_0 \\
\qquad \mathsf{else\ if}\ r = 1\ \mathsf{then}\ AUX_1 \\
\qquad \ldots \\
\qquad \mathsf{else\ if}\ r = c\ \mathsf{then}\ AUX_c \\
\qquad \mathsf{else}\ \ldots error \ldots
\end{array}
$$

$$
AUX_i \stackrel{\text{def}}{=} \begin{cases} \{i\}\lambda x.\downarrow\neg i[\uparrow a[proc\ (x,a)]] & 0 \leq i < a \\ \{i\}\lambda x.proc\ (x,i) & a \leq i \leq b \\ \{i\}\lambda x.\downarrow\neg i[\uparrow b[proc\ (x,b)]] & b < i \leq c \end{cases}
$$

Code can invoke *proc* directly, but no ring change will take place. By invoking *PROC*, code can run *proc* with an appropriate ring. Such an invocation has the form $\mathsf{let}\, x = \downarrow\bot\ PROC\ r; x\ V$, where the current ring is bound to variable $r$ as an integer, and the argument to the function is $V$. □