λ-RBAC: Programming with Role-Based Access Control

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

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

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. JAAS and .NET enable two forms of access control mechanisms[3]. 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. Second, JAAS and .NET also permit *programmatic* mechanisms that permit access control code to be intertwined with functionality code, e.g. in the code of a component implementing interface I. Why commingle conceptually separate concerns? To enable the programmer to enforce access control that is sensitive to the control and dataflow of the code implementing the functionality.

There is extensive literature on policy languages to specify and implement policies (e.g. [14,25,13,3,26,12] 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:

⋆ Supported by NSF Cybertrust 0430175
⋆⋆ Supported by the DePaul University Research Council
⋆⋆⋆ Supported by NSF Career 0347542

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

- The user of a component faces the issue of safety: is a particular set of rights sufficient to use the component? (ie. 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. 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.

Related work. There is extensive literature on Role-Based Access-Control (RBAC) models including NIST standards for RBAC [22,11]; see [10] for a textbook survey.

The main motivation for RBAC, in software architectures (e.g. [19,18]) 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.

Unified frameworks encompassing RBAC and trust–management systems have also been studied [23], in part by incorporating history-sensitive ideas into the RBAC model [4]. Our work is close in spirit, if not in technical development, to edit automata [14], which use aspects to avoid the explicit intermingling of security and baseline code.

The papers most closely related to our work are those of Braghin, Gorla and Sassone [6] and Compagnoni, Garalda and Gunter [9]. [6] presents the first concurrent calculus with a notion of RBAC, whereas [9]'s language enables privileges depending upon location. This paper extends the lambda calculus with RBACand internalizes the lattice structure of roles. We present a more detailed comparison at the end of the paper.

An overview of our technical contributions. 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 JAASand .NET to determine the expressiveness required for the access control mechanisms

Roughly, the operation of λ-RBAC is as follows. 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. The set of roles used in a program is static: we do not allow the dynamic creation of roles. The only run-time operations on roles are as follows. There are combinators to check that the role-context is at least some minimum role: an exception is raised if the check fails. Rights modulation (c.f. "sessions" in RBAC) is achieved by impersonation: this enables an application to operate under the guise of different users at different times .

We assume that roles form a lattice: abstracting the concrete union/intersection operations of the motivating examples. Some of our results assume that the lattice is boolean, i.e. the lattice has a negation operation abstracting the concrete set complement of the motivating examples. Our study is parametric on the underlying role lattice. Our calculus includes a single combinator for role checking and two combinators for impersonation: one for rights weakening and the other for rights amplification. We internalize the right to amplify rights by considering role lattices with an explicit role constructor, *amplify*. This enables the reuse of the access control mechanisms of λ-RBAC to control rights amplification.

We demonstrate the expressiveness of the calculus 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. For both we prove preservation and progress properties.

Rest of the paper. We begin with a discussion of the dynamic semantics of λ-RBAC in section 2, illustrating the expressiveness of the language with examples in section 3. Section 4 describes the static analyses. The following section 5 provides types for the examples of section 3. We conclude with a summary of related work in section 6.

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 six constructors discussed below. We assume that each role constructor κ has an associated arity, $\mathsf{arity}(\kappa)$. Roles P,Q,R,S,T have the form $\kappa(R_1,\ldots,R_n)$.

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 be monotone with respect to $\geqslant$.

Further we require that roles form a boolean lattice. So, the role lattice is distributive: 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. For any R, S, we have $\vdash R \geqslant \bot$ and $\vdash \top \geqslant R$ and $\vdash R \sqcup S \geqslant R$ and $\vdash R \geqslant R \sqcap S$. In addition, there is a complement $R^\star$ for every role R, where R and S are complements if $R \sqcap S = \bot$ and $R \sqcup S = \top$.

Finally, in example 4 we require the unary constructor *amplify*, where $\mathit{amplify}(R)$ represents the right to store R in a piece of code; i.e. if $P = \mathit{amplify}(R)$, then role P stands for the right to provide the role R.

In summary, the syntax is as follows.

$$P,Q,R,S,T \ ::= \ \kappa(R_1,\ldots,R_n) \qquad \kappa \ ::= \ \cdots \mid \bot \mid \top \mid \sqcup \mid \sqcap \mid {}^\star \mid \mathit{amplify}$$

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[4] 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. Abstraction is written "$\{Q\}\lambda x.M$" where role Q is demanded to execute M. Application is written "$\downarrow P\ U\ V$" where the caller restricts its rights to P during the execution of function U.

We define evaluation using a small-step operational semantics; therefore, we include explicit syntax for the *frame* "$\downarrow P[M]$" which represents the execution of term M with rights restricted to P. We use frames additionally for rights escalation, with the form "$\uparrow P[M]$." The two forms of frames together allows code to assume any role, which — if entirely uncontrolled — allows code to circumvent an intended policy. We address this issue in example 4, where we describe the use of the *amplify* constructor to control rights escalation.

Let x, y, z, f, g range over variable names. The syntax of values and terms are as follows.

$$U,V \ ::= \ \cdots \mid x \mid \{Q\}\lambda x.M$$
$$M,N,L \ ::= \ \cdots \mid U \mid \mathsf{let}\, x{=}M;\, N \mid \downarrow P\ U\ V \mid \downarrow P[M] \mid \uparrow P[M]$$

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

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

$$\frac{\vdash R \geqslant Q}{R \vdash \downarrow P\ (\{Q\}\lambda x.M)\ U \rightarrow \downarrow P[M[{}^{U}\!/\!_{x}]]}$$

$$\frac{R \vdash M \rightarrow M'}{R \vdash \mathsf{let}\, x{=}M;\, N \rightarrow \mathsf{let}\, x{=}M';\, N} \qquad \frac{R \sqcap P \vdash M \rightarrow M'}{R \vdash \downarrow P[M] \rightarrow \downarrow P[M']} \qquad \frac{R \sqcup P \vdash M \rightarrow M'}{R \vdash \uparrow P[M] \rightarrow \uparrow P[M']}$$

$$\frac{}{R \vdash \mathsf{let}\, x{=}U;\, N \rightarrow N[{}^{U}\!/\!_{x}]} \qquad \frac{}{R \vdash \downarrow P[U] \rightarrow U} \qquad \frac{}{R \vdash \uparrow P[U] \rightarrow U}$$

The rules are straightforward, but for application; note only that a frame is discarded when the guarded term is fully evaluated. 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

[4] We have chosen an explicitly sequenced variant (with let). Implicit sequencing can be recovered as follows: $\downarrow R\ M\ N \triangleq \mathsf{let}\, x{=}M;\ \mathsf{let}\, y{=}N;\ \downarrow R\ x\ y$. 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.

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 $R \vdash M \Downarrow \mathsf{fail}$[5].

LEMMA 1. *If $S \vdash M \to M'$ and $\vdash R \geqslant S$ then $R \vdash M \to M'$.* □

3 Examples

EXAMPLE 2 (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 modeled as:

$$\begin{aligned} \mathit{filesystem} \overset{\text{def}}{=}\ & \lambda \mathit{name}.\mathsf{if}\ \mathit{name} = \text{"file1"}\ \mathsf{then}\ \mathsf{check}\ \mathit{ADMIN};\ \text{"content1"} \\ & \quad \mathsf{else\ if}\ \mathit{name} = \text{"file2"}\ \mathsf{then}\ \mathsf{check}\ \mathit{ALICE} \sqcap \mathit{BOB};\ \text{"content2"} \\ & \quad \mathsf{else}\ \text{"error: file not found"} \end{aligned}$$

Where $\mathsf{check}\ R \triangleq (\{R\}\lambda\_.())()$. Assuming incomparable roles *ALICE*, *BOB*, and *CHARLIE* each strictly dominated by *ADMIN*, code running in the *ADMIN* role can access both files:

$$\begin{aligned} & \mathit{ADMIN} \vdash \mathit{filesystem}\ \text{"file1"} \to^* \mathsf{check}\ \mathit{ADMIN};\ \text{"content1"} \to^* \text{"content1"} \\ & \mathit{ADMIN} \vdash \mathit{filesystem}\ \text{"file2"} \to^* \mathsf{check}\ \mathit{ALICE} \sqcap \mathit{BOB};\ \text{"content2"} \to^* \text{"content2"} \end{aligned}$$

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

$$\begin{aligned} & \mathit{ALICE} \vdash \mathit{filesystem}\ \text{"file1"} \to^* \mathsf{check}\ \mathit{ADMIN};\ \text{"content1"} \Downarrow \mathsf{fail} \\ & \mathit{BOB} \vdash \mathit{filesystem}\ \text{"file2"} \to^* \mathsf{check}\ \mathit{ALICE} \sqcap \mathit{BOB};\ \text{"content2"} \to^* \text{"content2"} \end{aligned}$$

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

$$\begin{aligned} & \mathit{CHARLIE} \vdash \mathit{filesystem}\ \text{"file1"} \to^* \mathsf{check}\ \mathit{ADMIN};\ \text{"content1"} \Downarrow \mathsf{fail} \\ & \mathit{CHARLIE} \vdash \mathit{filesystem}\ \text{"file2"} \to^* \mathsf{check}\ \mathit{ALICE} \sqcap \mathit{BOB};\ \text{"content2"} \Downarrow \mathsf{fail} \end{aligned}$$

[5] 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.

Now 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.

$$\begin{aligned} \mathit{webserver} \overset{\text{def}}{=} \lambda \mathit{name}.\,&\mathsf{if}\ \mathit{name} = \text{"file1"}\ \mathsf{then}\ \mathit{filesystem}\ \mathit{name} \\ &\mathsf{else\ if}\ \mathit{name} = \text{"file2"}\ \mathsf{then}\ \mathit{filesystem}\ \mathit{name} \\ &\mathsf{else\ check}\ \mathit{DEBUG};\ \text{"error: file not found"} \end{aligned}$$

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

$$\begin{aligned} \mathit{ALICE} \vdash \mathit{webserver}\ \text{"file2"} &\to^* \mathit{filesystem}\ \text{"file2"} \\ &\to^* \mathsf{check}\ \mathit{ALICE} \sqcap \mathit{BOB};\ \text{"content2"} \to^* \text{"content2"} \end{aligned} \qquad \square$$

Example 3 illustrates how the Domain-Type Enforcement (DTE) security mechanism [5,27], as found in the NSA's Security-Enhanced Linux (SELinux) [15], can be implemented in λ-RBAC. Further discussion of the relationship between RBAC and DTE can be found in [10,12].

EXAMPLE 3 (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 only changes when another image is executed. DTE facilitates least privilege by limiting domain transitions based upon the source and target domains, and type assigned to the invoked executable file.

The DTE domain transition from role R to role S (each acting as domains) can be modeled by the function $R \xrightarrow{E} S$ that allows code running at role R to apply a function at role S: $R \xrightarrow{E} S \overset{\text{def}}{=} \{R\}\lambda(f,x).\downarrow\perp f\ (\{E\}\lambda g.\uparrow S[g\ x])$.

However, the domain transition is only performed when the function is associated with role E, modeling 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 $\mathit{assignType}_E$ allows code running at *ADMIN* to assign DTE types to other code: $\mathit{assignType}_E \overset{\text{def}}{=} \{\mathit{ADMIN}\}\lambda g.\,\lambda h.\uparrow E[\downarrow\perp h\ g]$. For example, for a function value U:

$$\mathit{ADMIN} \vdash \downarrow\perp \mathit{assignType}_E\ U \to^* \lambda h.\uparrow E[\downarrow\perp h\ U]$$

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

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

With the $R \xrightarrow{E} S$ and $\mathit{assignType}_E$ functions we can adapt the login example from [27] 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:

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

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*. □

EXAMPLE 4 (CONTROLLING RIGHTS AMPLIFICATION). The *amplify* constructor provides a flexible dynamic way to control rights amplification. Suppose that M contains no direct rights amplification (subterms of the form $\uparrow P\,[\,\cdot\,]$). Then, in "$\mathsf{let}\ g = U\,;\ \downarrow R\,[M]$" we may view U as a Trusted Computing Base (TCB) — a privileged function which may escalate rights — and to N as restricted *user* code; g is then an entry point to the TCB and R is the user role. User code is therefore executed at the restricted role R, and rights amplification may only occur through invokation of g.

Non-trivial programs have larger TCBs with more entry points. 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. 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.

$$\mathsf{let}\ \mathit{do}_P = \{\mathit{amplify}(P)\}\lambda f.\,\lambda x.\,\uparrow P\,[\,f\ x\,]\,;$$

The privileged function do_P that performs rights amplification (for P) is justified by the check for $\mathit{amplify}(P)$ on any caller of do_P. One may also wish to explictly prohibit a term N from direct amplification of some right Q; with such a convention in place, this can be achieved using the frame $\downarrow(\mathit{amplify}(Q)^{\star})\,[N]$.

A formal and systematic general mechanism to enforce such a coding convention — by requiring code to use definable higher order combinators in place of unchecked frames — is omitted from this extended abstract for space reasons. □

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. In 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.

4.1 A type system to calculate caller roles

Our first type system attempts to calculate a caller role that guarantees that all execution paths are successful. The judgment $\Gamma \vdash M : \{R\}\ \tau$ asserts that R suffices to evaluate M, i.e. if M is executed with a role that dominates R, then all access checks in M are guaranteed to be successful. Values require no computation to evaluate, thus the value judgment $\Gamma \vdash U : \tau$ includes only the type τ. The syntax of types is as follows.

$$\sigma,\tau ::= \cdots \mid \sigma \to\!\{Q \triangleright R\}\ \tau \qquad \Gamma,\Delta ::= x_1 : \sigma_1, \ldots, x_n : \sigma_n$$

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. The typing judgments are as follows.

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

(VAL-VAR)
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

(VAL-ABS)
$$\frac{\Gamma, x{:}\sigma \vdash M : \{R\}\ \tau}{\Gamma \vdash \{Q\}\lambda x.M : \sigma \to\!\{Q \triangleright R\}\ \tau}$$

(TERM-VAL)
$$\frac{\Gamma \vdash U : \tau}{\Gamma \vdash U : \{\bot\}\ \tau}$$

(TERM-LET)
$$\frac{\Gamma \vdash M : \{R\}\ \sigma \qquad \Gamma, x{:}\sigma \vdash N : \{S\}\ \tau}{\Gamma \vdash \mathsf{let}\ x{=}M;\ N : \{R \sqcup S\}\ \tau}$$

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

(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)
$$\frac{\Gamma \vdash M : \{S\}\ \tau \qquad \vdash R \geqslant S}{\Gamma \vdash M : \{R\}\ \tau}$$

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.

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

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

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

The following example of Church booleans, illustrates the use of subtyping. The Church booleans, $\lambda t.\,\lambda f.t$ and $\lambda t.\,\lambda f.f$ can be given type $(\sigma \rightarrow\{R\}\ \tau) \rightarrow (\sigma \rightarrow\{S\}\ \tau) \rightarrow (\sigma \rightarrow\{R \sqcup S\}\ \tau)$.

The type system satisfies standard preservation and progress properties.

THEOREM 5. *If* $\Gamma \vdash M : \{R\}\ \tau$ *and* $S \vdash M \rightarrow M'$*, then* $\Gamma \vdash M' : \{R\}\ \tau$.
If $\Gamma \vdash M : \{R\}\ \tau$ *then either* M *is a value, or* $R \vdash M \rightarrow M'$*, for some* M'.

4.2 A type system to determine callee protection

Our second type system ("$\Vdash$") 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 deduce the minimum protection demanded by the callee on all execution paths. View $\Gamma \Vdash M : \{S\}\ \tau$ as asserting that it is not possible for M to evaluate to a value without using a role above S, i.e. the guaranteed protection for M is at least S.

The type system presented below has altered versions of TERM-APP, TERM-RESTRICT, TERM-PROVIDE and inverted versions of TERM-SUBEFFECT and SUBTYPING-BASE.

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

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

(TERM-APP)
$$\frac{\Gamma \Vdash U : \sigma \rightarrow\{Q \triangleright R\}\ \tau \qquad \Gamma \Vdash V : \sigma' \quad \vdash \sigma' <: \sigma}{\Gamma \Vdash {\downarrow}P\ U\ V : \{Q \sqcup R\}\ \tau}$$

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

(TERM-PROVIDE)
$$\frac{\Gamma \Vdash M : \{R\}\ \tau}{\Gamma \Vdash {\uparrow}P[M] : \{R \sqcap P^{\star}\}\ \tau}$$

In TERM-SUBEFFECT above, it is sound to weaken the role since the asserted protection is only reduced. ${\uparrow}P[M]$ adds P to the role context in the operational semantics. So, in TERM-PROVIDE, the guaranteed protection for ${\uparrow}P[M]$ removes P from the guaranteed protection for M.

As a consequence of the above rules, if M is a value, we can deduce that it must be typed at role $\perp$; values are already in normal form, so do not enforce any protection. Furthermore, in this system, the Church booleans may be given type: $(\sigma \rightarrow\{R\}\ \tau) \rightarrow (\sigma \rightarrow\{S\}\ \tau) \rightarrow (\sigma \rightarrow\{R \sqcap S\}\ \tau)$. illustrating the "minimum over all paths" principle via $R \sqcap S$ (to be contrasted with $R \sqcup S$ in the previous typing system.). More generally, the following theorem enables us to understand the invariants established by typing in this system [6].

THEOREM 6. *If* $\Gamma \Vdash M : \{S\}\ \tau$, $\vdash R \not\geqslant S$ *and* $R \vdash M \rightarrow M'$*, then* $\Gamma \Vdash M' : \{S\}\ \tau$.

If we start execution in a role context (R in the theorem) that does not suffice to pass the minimum protection guarantee (S in the theorem), 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.

[6] The usual form of the type preservation result does not hold for this system. For example $\Vdash (\{\top\}\lambda\_\,.\,())\ () : \{\top\}$ Unit and $\top \vdash (\{\top\}\lambda\_\,.\,())\ () \rightarrow ()$ but $\not\Vdash () : \{\top\}$ Unit.

5 Typing Examples

EXAMPLE 7. Recall the filesystem and web server from example 2. 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:

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

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 $\Vdash$ type:

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

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 $\Vdash$ 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 8. Recall the encoding of the DTE/SELinux domain transition mechanism from example 3. Define types for functions running at role S (acting as a domain) and functions that can prove their assigned DTE role is E by calling back with that role:

$$\begin{aligned} \mathsf{Func}(\sigma,\tau,S) &\overset{\text{def}}{=} \sigma \rightarrow\{S\}\ \tau \\ \mathsf{FuncDTEType}(\sigma,\tau,S,E) &\overset{\text{def}}{=} (\mathsf{Func}(\sigma,\tau,S) \rightarrow\{E \triangleright \bot\}\ \tau) \rightarrow\{\bot\}\ \tau \end{aligned}$$

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:

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

6 Conclusions

We have presented methods to aid the designer and use of components which include access control code (as permitted in the programmatic RBAC of JAAS/.NET). Our first analysis enables users of code to deduce the role at which code must be run. The other analysis method enables code designers to deduce the protection guarantees of their code by calculating the role that is verified on all execution paths.

In future work, we will explore extensions to role polymorphism and recursive roles following the techniques of [7,2].

Our paper falls into the broad area of research enlarging the scope of foundational, language-based security methods (see [24,16,1] for surveys). The papers that are most directly relevant to the current paper are [6,9]. 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. 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.

[6,9] 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 [9], and RBAC-schemes in [6]) to enforce control on rights activation.

Our paper deals with access control, so the extensive work on information flow, e.g., see [20] 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 [8,21,17] plays in the context of information flow; namely that of permitting access that would not have been possible otherwise. In addition, by internalizing the ability to amplify code rights into the role lattice, our system permits access control code to actively participate in managing rights amplification.

References

1. M. Abadi, G. Morrisett, and A. Sabelfeld. Language-based security. *J. Funct. Program.*, 15(2):129, 2005.
2. R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM TOPLAS*, 15(4):575–631, 1993.
3. 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.
4. E. Bertino, P. A. Bonatti, and E. Ferrari. TRBAC: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur.*, 4(3):191–233, 2001.
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. C. Braghin, D. Gorla, and V. Sassone. A distributed calculus for role-based access control. In *CSFW*, pages 48–60, 2004.
7. M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundam. Inf.*, 33(4):309–338, 1998.
8. S. Chong and A. C. Myers. Security policies for downgrading. In *ACM Conference on Computer and Communications Security*, pages 198–209, 2004.

9. A. Compagnoni, P. Garralda, and E. Gunter. Role-based access control in a mobile environment. In *Symposium on Trustworthy Global Computing*, 2005.
10. D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-Based Access Control*. Computer Security Series. Artech House, 2003.
11. 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.
12. J. Hoffman. Implementing RBAC on a type enforced system. In *13th Annual Computer Security Applications Conference (ACSAC '97)*, pages 158–163, 1997.
13. 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.
14. 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.
15. P. A. Loscocco and S. D. Smalley. Meeting critical security objectives with Security-Enhanced Linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, 2001.
16. J. C. Mitchell. Programming language methods in computer security. In *POPL*, pages 1–26, 2001.
17. A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *CSFW*, pages 172–186, 2004.
18. 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.
19. 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.
20. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
21. A. Sabelfeld and A. C. Myers. A model for delimited information release. In *ISSS*, pages 174–191, 2003.
22. R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *IEEE Computer*, 29(2), 1996.
23. R. S. Sandhu and J. Park. Usage control: A vision for next generation access control. *ACM Trans. Inf. Syst. Secur.*, 2004.
24. 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.
25. F. Siewe, A. Cau, and H. Zedan. A compositional framework for access control policies enforcement. In *FMSE*, pages 32–42, 2003.
26. E. G. Sirer and K. Wang. An access control language for web services. In *SACMAT '02: Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 23–30, 2002.
27. 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 *Proceedings of the Sixth USENIX UNIX Security Symposium*, 1996.