Extensible Access Control with Authorization Contracts

Scott Moore
Harvard University (USA)
sdmoore@fas.harvard.edu

Christos Dimoulas
Harvard University (USA)
chrdimo@seas.harvard.edu

Robert Bruce Findler
Northwestern University (USA)
robby@eecs.northwestern.edu

Matthew Flatt
University of Utah (USA)
mflatt@cs.utah.edu

Stephen Chong
Harvard University (USA)
chong@seas.harvard.edu

Abstract

Existing programming language access control frameworks do not meet the needs of all software components. We propose an expressive framework for implementing access control monitors for components. The basis of the framework is a novel concept: the *authority environment*. An authority environment associates rights with an execution context. The building blocks of access control monitors in our framework are *authorization contracts*: software contracts that manage authority environments. We demonstrate the expressiveness of our framework by implementing a diverse set of existing access control mechanisms and writing custom access control monitors for three realistic case studies.

Categories and Subject Descriptors D.3.1 [*PROGRAMMING LANGUAGES*]: Formal Definitions and Theory—Semantics; D.2.4 [*SOFTWARE ENGINEERING*]: Software/Program Verification—Programming by contract

Keywords access control; contracts; authorization logic

1. Introduction

An access control monitor mediates requests to call sensitive operations and allows each call if and only if the request possesses the necessary rights to call the operation. Broadly speaking, when an access control mechanism is presented with a call to a sensitive operation, it must be able to answer two questions. First, which rights are required for the call? And second, which rights does the request possess? The design of an access control mechanism specifies, implicitly or explicitly, the answers to these questions.

For example, Unix file permissions describe which users are allowed to call which operations on a file. The access control mechanism uses file permissions to determine what rights are necessary to call different sensitive operations. Each Unix process executes on behalf of a specific user, and a request to call an operation possesses the same rights as the user of the process that issues the request. Thus, file permissions answer the first question, and the rights of the user associated with a process answer the second question. Importantly, Unix associates users and processes in two different ways. By default, a new process runs on behalf of the same user as the process that spawned it. But a process can run on behalf of a different user if it runs an executable that has the `setuid` bit set. When a process invokes a `setuid` executable, the operating system launches a new process to run the executable and associates the new process with the user that owns the executable, rather than the user that invoked it. Hence, this feature creates services that provide restricted access to resources that an invoking user could not otherwise access.

Similar to operating systems, software components also need access control mechanisms to prevent unauthorized clients from calling sensitive operations while allowing authorized ones to do so. Thus, when responding to a request to call a sensitive operation, access control mechanisms for components must be able to answer the same two questions: which rights are necessary for the call and which rights the request possesses.

However, access control needs of components vary, and it is impossible to choose a single answer to these questions that satisfies all component authors. To make things worse, access control mechanisms for general purpose programming languages have made design choices that are not suitable for all application domains and are typically mutually incompatible. For example, Java stack inspection [40] determines the rights associated with a call site by walking the stack from the current stack frame. In contrast, object-capability languages (e.g., E [25] and Caja [26]) determine rights by the lexical structure of the program: a code may call operations on exactly those resources that are reachable from variables in the code's text.

In this paper we propose a new, extensible access control framework that allows component authors to design access control monitors that suit their needs. The framework supports the design and implementation of many different novel and existing access control monitors for software components. Moreover, because different monitors are implemented using a common framework, different software components within the same application can use different access control mechanisms.

The framework builds on a novel concept: the *authority environment*. Just as each execution context has a variable environment that maps variable identifiers to values, each execution context has an authority environment that associates the context with its rights to call operations. The rights that a call to a sensitive operation possesses are those possessed by the authority environment of the call's execution context.

By analogy with dynamic and lexical scoping of variable environments, we identify two ways in which an execution context can receive authority:

1. *dynamically,* by inheriting the authority environment of the surrounding execution context, and
2. *lexically,* by capturing the authority environment of the execution context where it is defined.

Returning to the Unix file system example, a process receives authority dynamically when it inherits the user of the process that launched it. A process receives authority "lexically" when it runs a `setuid` executable.

Based on the correspondence with variable scoping, we define a framework for designing access control monitors as sets of monitor actions that manipulate authority environments (§3). We implement our framework as a library for Racket [16] without changes to the language's runtime. We use *higher-order contracts* [15] to specify where an access control monitor should interpose on a program and how it should manage authority environments. Contracts are executable specifications attached to software components that support separation of concerns by removing defensive checks from code implementing functionality [22–24]. In the same way, our *authorization contracts* separate the task of access control from the program's functionality.

The design of this framework presents four major contributions:

1. the introduction of *authority environments* as a unifying concept for access control mechanisms (§2),
2. the introduction of *context contracts* to check and enforce properties of execution contexts (§3.1),
3. a novel *authorization logic* for representing and querying authority in authority environments (§3.2), and
4. *authorization contracts* that specialize context contracts for managing authority environments and enforcing access control policies expressed in the logic (§3.3).

We have used the framework to implement diverse access control mechanisms: discretionary access control, stack inspection, history-based access control, and object-capabilities (§4). We demonstrate the practicality of our approach with three realistic case studies (§5).

2. Authority Environments

In this section, we introduce *authority environments* as a unifying concept for access control. First, we review the differences between lexical and dynamic scoping (§2.1). Then we describe the connection between lexical and dynamic scoping and access control (§2.2) and show how we can use scoping in the design of a framework for writing access control monitors (§2.3). Throughout, we use small examples in the Racket programming language [16].

2.1 Lexical and Dynamic Scoping

The scope of a variable binding is the spatial and temporal part of the program in which it is visible. A common way to categorize strategies for assigning scopes to bindings is as either *lexical* or *dynamic*. Earlier work distinguishes between the *scope* of a binding, which describes where the binding is visible in the program text, and the *extent* of a binding, which describes when the binding is visible during execution. Dynamic scope often refers to bindings that have dynamic extent and "indefinite" scope. Here, we use dynamic scope to refer to bindings that have dynamic extent and lexical scope, also called "fluid" scope [17, 34, 35].

Under lexical scoping, a variable refers to the binding from its closest binder in the textual structure of the program. For example, in the Racket expression below, the variable `x` in function `f` refers to the binding in the outer-most `let` statement. The evaluation of this expression returns `0` since the inner-most `let` statement has no effect on the value `x` binds within `f`.

```
(let ([x 0])
  (let ([f (lambda () x)])
    (let ([x 42])
      (f))))
```

In a programming language with fluid scoping, programmers can instead associate a binding with the dynamic extent of an expression. That binding is visible to any code that runs in the dynamic extent of the expression. For example, the following Racket expression defines a new fluidly-scoped variable `x` with default value `0`. The `parameterize` expression binds `x` to the value `42` in the dynamic extent of its body. The variable `x` in the body of `f` refers to the most recent binding rather than the closest one in the program text. Since `f` is invoked within the `parameterize` expression, the program evaluates to `42` instead of `0`.

```
(let ([x (make-parameter 0)])
  (let ([f (lambda () (x))])
    (parameterize ([x 42])
      (f))))
```

Fluid scoping is a useful programming construct because it allows the context of an expression to communicate with its callees without explicitly threading arguments through the program. For example, a library function for printing may offer a parameter that determines the standard output file. Instead of threading that file as an argument through every function call leading to the `printf` routine, a client program can instead set the parameter once and all calls to `printf` in the body of the program see the client-specified file.

2.2 Scoping for Access Control

This ability to pass contextual information from an execution context to an eventual callee closely matches the problem of correctly determining the authority of a request to call a sensitive operation. To demonstrate this relationship, consider the design of a web application with multiple users. A key component of this application is a login function that authenticates users and executes code on their behalf:

```
(define (login user guess onSuccess)
  (if (check-password? user guess)
      (run-as-user user onSuccess)
      (error "Wrong password!")))
```

This `login` function takes three arguments: the `user` attempting to authenticate, the password `guess`, and a callback `onSuccess` to invoke with the user's rights if the password is correct. After checking the password, the login function changes the state of the program to indicate that the current user is now `user` and then calls `onSuccess`.

The body of `onSuccess` may attempt to access sensitive resources. For example, it may try to update a user's profile. To avoid an unauthorized update, the `update-profile` function checks whether the current user has sufficient rights:

```
(define (update-profile profileUser text)
  (if (can-update? currentUser profileUser)
      ...
      (error "Unauthorized!")))
```

Function `can-update?` compares the current user with the user who owns the profile to determine whether the update is authorized. This code thus implicitly uses the authority of its context, i.e., the current user, in much the same way that code accesses the dynamically scoped bindings from its context. By managing authority as an implicit context in this way, we can avoid modifying the code between the decision to run a computation with particular authority and the call to the sensitive operation. This has two advantages. First, threading authority explicitly through the program reduces extensibility, since third party code would need to be aware of and correctly handle authority explicitly. Second, if the code is untrustworthy, it might attempt to subvert the access control checks that protect the sensitive operation by fabricating its own authority.

Another requirement of the security of this application is that only code running with the authority of the main loop is allowed to switch users. According to the Principle of Least

```
(define-monitor users
 (monitor-interface
   setuid/c chuser/c checkuser/c)
 (action
  [chuser/c (user)
  #:on-create (do-create)
  #:on-apply  (do-apply
    #:check (≽@ current-principal user
                user)
    #:set-principal user)]
  [checkuser/c (user)
  #:on-create (do-create)
  #:on-apply  (do-apply
    #:check (≽@ current-principal user
                user))]
  [setuid/c
  #:on-create (do-create)
  #:on-apply  (do-apply
    #:set-principal closure-principal)]))
```

Figure 1. Defining a simple access control monitor

Privilege [31], we should further limit the code that is allowed to switch users to just the `login` function, and switch to an unprivileged user for the rest the program. Crucially, the body of the `login` function must still use the authority that was in its environment when it was created, i.e., the authority of the main loop. In a sense, for `login`, we wish to close over the authority of the main loop, in the same way that closures capture lexically scoped bindings.

To achieve this, we build on the analogy between scoping and access control and introduce the concept of an *authority environment*. An authority environment associates rights with an execution context, just as a variable environment associates bindings with an execution context. Just like variable environments, authority environments can be captured and associated with code, updated, and extended with new bindings for the dynamic extent of a computation. In this application, the authority environment of an execution context records the user on whose behalf the code executes. Section 2.3 shows how authority environments help enforce access control in our running example, including how to create a secure `login` function. Section 3 generalizes authority environments so that we can express a wide variety of access control mechanisms.

2.3 From Access Control to Authorization Contracts

Using the concept of an authority environment, we build an access control monitor that manipulates and inspects the authority environments of the example web application. The monitor consists of *actions* that describe how events in the execution of the application interact with its authority environment. We describe our framework for defining monitors in detail in Section 4. Here we explain only the features relevant to the example.

Figure 1 shows our example monitor. The monitor specifies three actions: `setuid/c`, `chuser/c`, and `checkuser/c`. Each action defines a higher-order function contract [15]. When one of these contracts is attached to a function, the contract captures the current authority environment and associates it with the function. When the function is called, the contract has access to both the authority environment at the call site and the authority environment that it has captured. The monitor configures each action-contract with two hooks: `#:on-create` and `#:on-apply`. By changing these hooks, monitor designers can implement actions that implement different forms of "lexically" and dynamically scoped authority environments.

Action `chuser/c` is parameterized with an argument `user` that identifies the user whose authority should be used during the execution of the body of a contracted function. The `#:on-apply` hook for `chuser/c` ignores the authority it has closed over and sets the active principal to `user` for the dynamic extent of the body of the contracted function, but only if the `#:check` holds, that is, the `current-principal` has authority over user `user`. Otherwise, it raises a security violation as a contract violation. Monitor action `checkuser/c` is also parameterized with a `user`. Upon a call of its contracted function, it checks that the `current-principal` has authority over `user`. If the check succeeds, the action does not change the authority environment. If the check fails, the action raises a security violation. The final monitor action, `setuid/c`, creates an authority closure: calling a function with this contract changes the current principal in the authority environment to the closed-over principal `closure-principal` for the dynamic extent of the function's body.

Using the monitor, we can now reimplement the web application. First, we can replace code that defensively performs authorization checks with contracts that enforce authority requirements:

```
(define/contract
  (update-profile someUser text)
  (->a ([user principal?] [text string?])
       #:auth (user) (checkuser/c user)
       any))
  ...)
```

This revised implementation of `update-profile` uses the Racket form **`define/contract`** to attach a contract to the `update-profile` function. This contract is a *dependent contract* [15] for a function. It says that `update-profile` takes two arguments: `user`, which must be a `principal?`, and `text`, which must be a `string?`. The keyword argument[1] `#:auth (user)(checkuser/c user)` says that the authorization contract for this function depends on the `user` argument and attaches the contract `(checkuser/c user)` to the function. Finally, the range of this contract is `any`, making no requirements on the return values. The definition of the function can now elide the authorization check.

We also revise the implementation of `login`:

[1] In Racket, a keyword argument is a (possibly optional) argument passed by keyword instead of position. A keyword is a symbol starting with `#:`.

```
(define/contract
  (login user guess onSuccess)
  (->a ([user principal?] [guess string?]
        [onSuccess (user) (chuser/c user)])
       #:auth () setuid/c any)
  (if (check-password? user guess)
      (onSuccess)
      (error "Wrong password")))
```

The keyword argument `#:auth () setuid/c` attaches the `setuid/c` action to the login function, capturing the authority of the program context where it is created. This allows the `login` function to execute with the captured authority and thus allows the application to switch to a less privileged principal without losing the ability to safely authenticate as a different user. In the original implementation, `login` uses a hand-rolled function `run-as-user` to confine `onSuccess` within the authority of `user`. In the revised code, `login` can invoke `onSuccess` directly. The contract on the `onSuccess` argument attaches the action `(chuser/c user)` to the function. This ensures that any call to `onSuccess` has the correct authority.

Rewriting this application to use authorization contracts makes the authorization requirements of each function clear, while also simplifying its implementation by removing authorization code that was spread throughout the program.

3. A Framework for Access Control

In this section, we present the general design of our framework with a formal model. First, we show how we extend existing higher-order function contracts to *context contracts* that check and modify information about their execution context (§3.1). Context contracts are expressive enough to enforce a wide range of properties. However, this flexibility makes it difficult to use them to implement and reason about access control. To free users from this burden, our framework provides a specialized interface for defining *authorization contracts*. The interface simplifies the definition of context contracts for access control in two ways. First, it specifies a common representation for authorization environments (§3.2). At the core of this representation is a novel authorization logic that describes how authority captured in a closure may be used. Second, it defines combinators for building authorization contracts (§3.3). Authorization contracts are specializations of context contracts that use the authorization logic to succinctly describe how they manage authority environments.

3.1 A Contract System with Context Contracts

We model higher-order contracts and context contracts as extensions to an applied lambda calculus with modules and parameters, which implement dynamic binding. Figure 2 describes the syntax of our model. Figure 3 gives the reduction semantics of the model. Evaluation contexts that are not presented in Figure 2 are standard and enforce call-by-value, left-to-right evaluation. The type system and the definition of

Surface syntax

$$\begin{array}{lcl}
p & ::= & m\,;p \mid e \\
m & ::= & \textsf{module}\,\ell\,\textsf{exports}\,x\,\textsf{with}\,x,\ldots\,\textsf{where}\,x = e,\ldots \\
e & ::= & v \mid e\;e \mid \mu\,x:\tau.\,e \mid \textsf{let}\,x = e\,\textsf{in}\,e \mid e \oplus e \\
 & \mid & e \leq e \mid \textsf{if}\,e\,\textsf{then}\,e\,\textsf{else}\,e \mid \textsf{make-parameter}\;e \\
 & \mid & \textsf{parameterize}\,e = e\,\textsf{in}\,e \mid ?e \mid e := e \\
 & \mid & \textsf{flat/c}(e) \mid \textsf{param/c}(e) \mid e:\tau \rightarrow (e)\;e \\
 & \mid & e:\tau \rightarrow_{\textsf{a}} (\lambda\,x:\tau.\,e)\;e \\
 & \mid & \textsf{ctx/c}(e,(e \Rightarrow e \leftarrow e),\ldots,e,(e \Rightarrow e \leftarrow e),\ldots) \\
v & ::= & () \mid n \mid \#\textsf{t} \mid \#\textsf{f} \mid \lambda\,x:\tau.\,e \mid c \\
c & ::= & \textsf{flat/c}(v) \mid \textsf{param/c}(c) \mid c:\tau \rightarrow (c)\;c \\
 & \mid & c:\tau \rightarrow_{\textsf{a}} (\lambda\,x:\tau.\,e)\;c \\
 & \mid & \textsf{ctx/c}(v,(v \Rightarrow v \leftarrow v)\ldots,v,(v \Rightarrow v \leftarrow v)\ldots) \\
\tau & ::= & \beta \mid \tau \rightarrow \tau \mid \tau\;\textsf{param} \mid \tau\;\textsf{ctc} \mid \textsf{ctx ctc} \\
\beta & ::= & \textsf{Unit} \mid \textsf{Int} \mid \textsf{Bool}
\end{array}$$

Expanded syntax

$$\begin{array}{lcl}
e & ::= & \ldots \mid {}^{\ell}\textsf{mon}_j^k(e,e) \mid \textsf{check}_j^k(e,e) \mid \textsf{error}_j^k \\
 & \mid & \textsf{guard}_j\;(e,v,v,e) \mid \textsf{install/p}_j\;(v,e,e) \\
v & ::= & \ldots \mid \textsf{p}(r) \mid {}^{\ell}\textsf{param/p}_j^k(c,v) \\
 & \mid & {}^{\ell}\textsf{ctx/p}_j^k\;(v,(v \Rightarrow v \leftarrow v),\ldots,v) \\
 & \mid & \textsf{install/p}_j\;(v,v,v)
\end{array}$$

Selected evaluation contexts

$$\begin{array}{lcl}
E & ::= & \ldots \mid \textsf{guard}_j\;(E,v,v,e) \\
 & \mid & \textsf{install/p}_j\;(v,e,E) \mid \textsf{install/p}_j\;(v,E,v)
\end{array}$$

Figure 2. Syntax

evaluation contexts are given in Appendix A. Though we omit the extension for clarity, we have also extended the model with first class continuations, following Takikawa et al. [37], because our implementation language supports them and they interact in interesting ways with our framework.

The semantics of common language features is standard. Below, we explain the semantics of modules, parameters, higher-order contracts, and context contracts.

3.1.1 Modules

A program p is a sequence of modules followed by a top-level expression. A module simultaneously defines a collection of values owned by a single component and a set of contracts for those values. Each module

$$\textsf{module}\,\ell\,\textsf{exports}\,x_{v_1}\,\textsf{with}\,x_{c_1},\ldots\,\textsf{where}\,y_1 = e_1,\ldots$$

has a label ℓ and defines a set of values $y_1, \ldots$. Values within the module are visible only to subsequent modules in the program if they are exported. An export declaration x_{v_i} with x_{c_i} binds x_{v_i} in the rest of the program to the value defined as x_{v_i} in the where clause, but only after attaching to it the contract defined as x_{c_i} in the where clause. Metafunction import, shown in Figure 3, substitutes occurrences of x_{v_i} in the rest of the program with monitored values ${}^{\ell}\textsf{mon}_j^k(c,v)$ that enforce the contract.

3.1.2 Higher-order contracts

Term ${}^{\ell}\textsf{mon}_j^k(c,v)$ attaches contract c to value v and monitors whether uses of v satisfy the contract. Labels j, k, and ℓ, identify, respectively, the module that imposed the contract, the module that provided the value, and the module that is the client of the value. The top-level expression of a program is identified by the distinguished label ℓ_0. These labels are used to assign *blame* when a contract is violated. The simplest contract is a flat contract $\textsf{flat/c}(v_c)$ that takes a predicate v_c as an argument. Flat contracts can be applied only to values of base types Int, Bool, and Unit. When the contract is attached to a value, the predicate is applied to the value. If the predicate returns true, the value passes to its context. Otherwise, the contract system stops the program and raises an error blaming the provider of the value.

Contract $c_d:\tau \rightarrow (c_c)\;c_r$ is a higher-order function contract. It specifies a contract c_d for the domain of the function and a contract c_r for the range of the function. In addition, it specifies a *context contract* c_c for the function. Context contracts, novel to this work, are higher-order contracts that enforce restrictions on the execution context of function calls. They are described in detail below. Attaching contract $c_d:\tau \rightarrow (c_c)\;c_r$ to a function returns a new value that enforces contracts on the argument and results of the function and applies the context contract c_c.

Contract $c_d:\tau \rightarrow_{\textsf{a}} (\lambda\,x:\tau.\,e_c)\;v_r$ is an indy-dependent[2] contract for higher-order functions. This contract corresponds to the `->a` contracts from Section 2. The contract is *dependent* since the contract uses the argument of a contracted function to choose a context contract for the function. Applying a function v with this contract has four steps. First, the argument is wrapped with the contract for the domain, c_d. Second, the wrapped argument is passed to the function $\lambda\,x:\tau.\,e_c$ to construct a context contract. Third, the resulting context contract is attached to v, which is applied to the wrapped argument. Finally, the contract for the range, c_r, is attached to the result of the application.

3.1.3 Parameters

Parameters are first-class values that can be used to access and install dynamic bindings. Parameters implement fluid scope because access to their dynamic bindings is controlled lexically by access to the parameter itself. The expression make-parameter e creates a new parameter $\textsf{p}(r)$ with default value the result of e, where r is a fresh tag uniquely identifying the parameter. The default value is recorded in the store σ. Term parameterize $\textsf{p}(r) = e_1$ in e_2 installs the result of e_1 as the new value of the parameter $\textsf{p}(r)$ for the dynamic extent of e_2. Accessing the value of a parameter

[2] An indy-dependent contract is a dependent function contract that uses the "indy" strategy for blame assignment [8].

$$\begin{aligned}
&\langle \mathsf{module}\,\ell\,\mathsf{exports}\;x_1\;\mathsf{with}\,x_{c_1},\ldots\;\mathsf{where}\;y_1 = v_1,\ldots,y_n = v_n,y_{e_1} = e_1,\ldots y_{e_m} = e_m;\,p,\sigma\rangle\\
&\qquad\to\quad \langle \mathsf{module}\,\ell\,\mathsf{exports}\;x_1\;\mathsf{with}\,x_{c_1},\ldots\;\mathsf{where}\;y_1 = v_1,\ldots,y_n = v_n,y_{e_1} = \{{}^{v_i}/_{y_i}\}e_1,y_{e_2} = e_2,\ldots y_{e_m} = e_m;\,p,\sigma\rangle
\end{aligned}$$

$$\begin{aligned}
&\langle \mathsf{module}\,\ell\,\mathsf{exports}\;x_1\;\mathsf{with}\,x_{c_1},\ldots\;\mathsf{where}\;\ldots,x_1 = v_1,\ldots,x_{c_1} = c_1,\ldots;\,p,\sigma\rangle\\
&\qquad\to\quad \langle \mathrm{import}\llbracket \ell,(x_1,\ldots,x_n),(v_1,\ldots,v_n),(c_1,\ldots,c_n),p\rrbracket,\sigma\rangle
\end{aligned}$$

$$\begin{array}{lcl}
\langle E[(\lambda\,x:\tau.\;e)\;v],\sigma\rangle & \to & \langle E[\{{}^{v}/_{x}\}e],\sigma\rangle\\
\langle E[\mu\,x:\tau.\;e],\sigma\rangle & \to & \langle E[\{{}^{\mu\,x:\tau.\;e}/_{x}\}e],\sigma\rangle\\
\langle E[\mathsf{let}\,x = v\;\mathsf{in}\;e],\sigma\rangle & \to & \langle E[\{{}^{v}/_{x}\}e],\sigma\rangle\\
\langle E[\mathsf{if}\;\#\mathsf{t}\;\mathsf{then}\;e_1\;\mathsf{else}\;e_2],\sigma\rangle & \to & \langle E[e_1],\sigma\rangle\\
\langle E[\mathsf{if}\;\#\mathsf{f}\;\mathsf{then}\;e_1\;\mathsf{else}\;e_2],\sigma\rangle & \to & \langle E[e_2],\sigma\rangle\\
\langle E[v_1 \oplus v_2],\sigma\rangle & \to & \langle E[v],\sigma\rangle\ \text{where}\ v = v_1 \oplus v_2\\
\langle E[v_1 \leq v_2],\sigma\rangle & \to & \langle E[v],\sigma\rangle\ \text{where}\ v = v_1 \leq v_2\\
\langle E[\mathsf{make\text{-}parameter}\;v],\sigma\rangle & \to & \langle E[\mathsf{p}(r)],\sigma[r \mapsto v]\rangle\ \text{where}\ r\ \text{is fresh}\\
\langle E[?\mathsf{p}(r)],\sigma\rangle & \to & \langle v,\sigma\rangle\ \text{where}\ \sigma(r) = v\ \text{and}\\
 & & E\ \text{does not contain}\ \mathsf{parameterize}\,\mathsf{p}(r) = v'\ \mathsf{in}\ E'\\
\langle E[\mathsf{parameterize}\,\mathsf{p}(r) = v\;\mathsf{in}\;E'[?r]],\sigma\rangle & \to & \langle E[\mathsf{parameterize}\,\mathsf{p}(r) = v\;\mathsf{in}\;E'[v]],\sigma\rangle\\
 & & \text{where}\ E'\ \text{does not contain}\ \mathsf{parameterize}\,\mathsf{p}(r) = v'\ \mathsf{in}\ E''\\
\langle E[\mathsf{p}(r) := v],\sigma\rangle & \to & \langle E[()],\sigma[r \mapsto v]\rangle\\
 & & \text{where}\ E\ \text{does not contain}\ \mathsf{parameterize}\,\mathsf{p}(r) = v'\ \mathsf{in}\ E'\\
\langle E[\mathsf{parameterize}\,\mathsf{p}(r) = v\;\mathsf{in}\;E'[\mathsf{p}(r) := v']],\sigma\rangle & \to & \langle E[\mathsf{parameterize}\,\mathsf{p}(r) = v'\;\mathsf{in}\;E'[v']],\sigma\rangle\\
 & & \text{where}\ E'\ \text{does not contain}\ \mathsf{parameterize}\,\mathsf{p}(r) = v''\ \mathsf{in}\ E''\\
\langle E[\mathsf{parameterize}\,\mathsf{p}(r) = v\;\mathsf{in}\;v'],\sigma\rangle & \to & \langle E[v'],\sigma\rangle\\
\langle E[{}^{\ell}\mathsf{mon}^{k}_{j}(\mathsf{flat/c}(v_c),v)],\sigma\rangle & \to & \langle E[\mathsf{check}^{k}_{j,k}((v_c\;v),v)],\sigma\rangle\\
\langle E[{}^{\ell}\mathsf{mon}^{k}_{j}(\mathsf{param/c}(c),v)],\sigma\rangle & \to & \langle E[{}^{\ell}\mathsf{param/p}^{k}_{j}(c,v)],\sigma\rangle\\
\langle E[{}^{\ell}\mathsf{mon}^{k}_{j}(c_d : \tau \to (c_c)\;c_r,v)],\sigma\rangle & \to & \langle E[({}^{\ell}\mathsf{mon}^{k}_{j}(c_c,\lambda\,x:\tau_d.\;{}^{\ell}\mathsf{mon}^{k}_{j}(c_r,v_f\;{}^{k}\mathsf{mon}^{\ell}_{j}(c_d,x)))\;v)],\sigma\rangle\\
 & & \text{where}\ x\ \text{is fresh}\\
\langle E[{}^{\ell}\mathsf{mon}^{k}_{j}(c_d : \tau \to_{\mathsf{a}} (\lambda\,x:\tau.\;e_c)\;c_r,v)],\sigma\rangle & \to & \langle E[\lambda\,y:\tau_d.\;{}^{\ell}\mathsf{mon}^{k}_{j}(\{{}^{{}^{k}\mathsf{mon}^{\ell}_{j}(c_d,y)}/_{x}\}e_c,{}^{\ell}\mathsf{mon}^{k}_{j}(c_r,v\;{}^{k}\mathsf{mon}^{\ell}_{j}(c_d,y)))],\sigma\rangle\\
 & & \text{where}\ y\ \text{is fresh}
\end{array}$$

$$\begin{aligned}
&\langle E[{}^{\ell}\mathsf{mon}^{k}_{j}(\mathsf{ctx/c}(v_c,(v_{c_{g_1}} \Rightarrow v_{c_{p_1}} \leftarrow v_{c_{v_1}}),\ldots,v_a,(v_{a_{g_1}} \Rightarrow v_{a_{p_1}} \leftarrow v_{a_{v_1}}),\ldots),v)],\sigma\rangle \quad\to\quad \langle E[\mathsf{check}^{k}_{j}((v_c\;()),e)],\sigma\rangle\\
&\qquad\text{where}\ e = \mathsf{guard}_j\;((v_{c_{g_1}}\;()),v_{c_{p_1}},v_{c_{v_1}},\ldots\;\mathsf{guard}_j\;((v_{c_{g_n}}\;()),v_{c_{p_n}},v_{c_{v_n}},{}^{\ell}\mathsf{ctx/p}^{k}_{j}\;(v_a,(v_{a_{g_1}} \Rightarrow v_{a_{p_1}} \leftarrow v_{a_{v_1}}),\ldots,v)))\\
&\langle E[({}^{\ell}\mathsf{ctx/p}^{k}_{j}\;(v_a,(v_{a_{g_1}} \Rightarrow v_{a_{p_1}} \leftarrow v_{a_{v_1}}),\ldots,v_f)\;v)],\sigma\rangle \quad\to\quad \langle E[(\mathsf{check}^{\ell}_{j}((v_a\;()),e)\;v)],\sigma\rangle\\
&\qquad\text{where}\ e = \mathsf{guard}_j\;((v_{a_{g_1}}\;()),v_{a_{p_1}},v_{a_{v_1}},\ldots\;\mathsf{guard}_j\;((v_{a_{g_n}}\;()),v_{a_{p_n}},v_{a_{v_n}},v_f))
\end{aligned}$$

$$\begin{array}{lcl}
\langle E[\mathsf{guard}_j\;(\#\mathsf{f},v_p,v_v,e_f)],\sigma\rangle & \to & \langle E[e_f],\sigma\rangle\\
\langle E[\mathsf{guard}_j\;(\#\mathsf{t},v_p,v_v,e_f)],\sigma\rangle & \to & \langle E[\mathsf{install/p}_j\;(v_p,(v_v\;()),e_f)],\sigma\rangle\\
\langle E[(\mathsf{install/p}_j\;(v_p,v_v,v_f)\;v)],\sigma\rangle & \to & \langle E[\mathsf{parameterize}\;v_p = v_v\;\mathsf{in}\;(v_f\;v)],\sigma\rangle\\
\langle E[?{}^{\ell}\mathsf{param/p}^{k}_{j}(c,v_p)],\sigma\rangle & \to & \langle E[{}^{\ell}\mathsf{mon}^{k}_{j}(c,?v_p)],\sigma\rangle\\
\langle E[\mathsf{parameterize}\;{}^{\ell}\mathsf{param/p}^{k}_{j}(c,v_p) = v\;\mathsf{in}\;e],\sigma\rangle & \to & \langle E[\mathsf{parameterize}\;v_p = {}^{k}\mathsf{mon}^{\ell}_{j}(c,v)\;\mathsf{in}\;e],\sigma\rangle\\
\langle E[{}^{\ell}\mathsf{param/p}^{k}_{j}(c,v_p) := v],\sigma\rangle & \to & \langle E[v_p := {}^{k}\mathsf{mon}^{\ell}_{j}(c,v)],\sigma\rangle\\
\langle E[\mathsf{check}^{k}_{j}(\#\mathsf{t},v)],\sigma\rangle & \to & \langle E[v],\sigma\rangle\\
\langle E[\mathsf{check}^{k}_{j}(\#\mathsf{f},v)],\sigma\rangle & \to & \langle \mathsf{error}^{k}_{j},\sigma\rangle
\end{array}$$

$$\begin{aligned}
&\mathrm{import}\llbracket k,(x_1,\ldots,x_n),(v_1,\ldots,v_n),(c_1,\ldots,c_n),m_1;\ldots;m_n;e\rrbracket =\\
&\qquad \mathrm{import}\llbracket k,(x_1,\ldots,x_n),(v_1,\ldots,v_n),(c_1,\ldots,c_n),m_1\rrbracket;\\
&\qquad \ldots;\\
&\qquad \mathrm{import}\llbracket k,(x_1,\ldots,x_n),(v_1,\ldots,v_n),(c_1,\ldots,c_n),m_n\rrbracket;\\
&\qquad \mathrm{import}\llbracket k,(x_1,\ldots,x_n),(v_1,\ldots,v_n),(c_1,\ldots,c_n),e\rrbracket\\
&\mathrm{import}\llbracket k,(x_1,\ldots,x_n),(v_1,\ldots,v_n),(c_1,\ldots,c_n),\mathsf{module}\,\ell\,\mathsf{exports}\;x_{v_1}\;\mathsf{with}\,x_{c_1},\ldots\;\mathsf{where}\;y_1 = e_1,\ldots,y_n = e_n\rrbracket =\\
&\qquad \mathsf{module}\,\ell\,\mathsf{exports}\;x_{v_1}\;\mathsf{with}\,x_{c_1},\ldots\;\mathsf{where}\;y_1 = \{{}^{{}^{\ell}\mathsf{mon}^{k}_{k}(c_i,v_i)}/_{x_i}\}e_1,\ldots,y_n = \{{}^{{}^{\ell}\mathsf{mon}^{k}_{k}(c_i,v_i)}/_{x_i}\}e_n\\
&\mathrm{import}\llbracket k,(x_1,\ldots,x_n),(v_1,\ldots,v_n),(c_1,\ldots,c_n),e\rrbracket \quad=\quad \{{}^{{}^{\ell 0}\mathsf{mon}^{k}_{k}(c_i,v_i)}/_{x_i}\}e
\end{aligned}$$

Figure 3. Reduction semantics

with term ?p(r) returns the value of the closest enclosing parameterize for p(r) in the current evaluation context. If there is no such term, it returns the current value for r in the store. Similarly, term p$(r) := v$ mutates the current binding for the parameter, updating the parameter associated with either the closest enclosing parameterize for p(r) in the current evaluation context or the value for r in the store, if there is no such expression.

The parameter contract param/c(c) is a higher-order contract that restricts uses of a parameter. A contracted parameter v reduces to a proxy ${}^{\ell}\text{param/p}_j^k(c, v)$ that records the labels of the contract, provider, and client modules and intercepts uses of the parameter to enforce that values bound to the parameter meet contract c.

3.1.4 Context contracts

To track properties of execution contexts, context contracts use parameters to install and access relevant state. A context contract interposes on programs at two key times: when the contract is attached to a function and when the contracted function is applied. At both times, the contract can inspect the current values of parameters to check that the current environment is satisfactory, capture the current value for later use, or change the parameterization of a call to the contracted function.

A context contract

$$\begin{array}{l}\text{ctx/c}(v_c,(v_{g_c} \Rightarrow v_{p_c} \leftarrow v_{v_c}),\ldots,\\ \quad v_a,(v_{g_a} \Rightarrow v_{p_a} \leftarrow v_{v_a}),\ldots)\end{array}$$

has four parts:

1. v_c, a predicate that checks whether the context is appropriate when the contract is attached,
2. $(v_{g_c} \Rightarrow v_{p_c} \leftarrow v_{v_c}),\ldots$, a list of *guarded parameterizations*, described below, to close over when the contract is attached,
3. v_a, a predicate that checks whether the context is appropriate when the contract function is called, and
4. $(v_{g_a} \Rightarrow v_{p_a} \leftarrow v_{v_a}),\ldots$, a list of guarded parameterizations to be installed around the body of the contracted function if the contract check succeeds.

The first two parts are evaluated when the contract is attached to a value. First, the predicate v_c is executed to allow the contract to check the current context. If the predicate returns #f, a contract error is raised blaming the client of the contract. Otherwise, each guarded parameterization $(v_{g_c} \Rightarrow v_{p_c} \leftarrow v_{v_c})$ from part 2 is evaluated in turn. Each guarded parameterization specifies a guard function v_{g_c}, a parameter v_{p_c}, and a value function v_{v_c}. If invoking the guard thunk v_{g_c} returns #t, the corresponding value thunk v_{v_c} is executed to produce a new value. This value is "closed over" and re-installed for parameter v_{p_c} when the contracted function is applied. The predicate v_a and the remaining parameterizations are recorded in a proxy ${}^{\ell}\text{ctx/p}_j^k\,(v_a,(v_{g_a} \Rightarrow v_{p_a} \leftarrow v_{v_a}),\ldots,v)$.

The proxy enforces additional checks and parameterizations when the contracted function is called. First, the parameter values captured when the contract was attached are reinstalled. This gives the evaluation of the proxy and the function call access to some bindings from when the contract was attached, in addition to any bindings that are present in the current evaluation context. With these captured bindings in place, the proxy first evaluates the predicate v_a, which checks whether the current context is satisfactory. If the predicate returns false, a contract error is raised blaming the client ℓ. Otherwise, the guarded parameterizations of the proxy are evaluated in a similar fashion as before. However, any new bindings are installed just for the dynamic extent of the contracted function's call.

```
module ℓ exports inner with inner/c,
                 outer with outer/c
where inner = λ x : Int. x,
      outer = λ f : (Int → Int). λ x : Int. (f x),
      true = λ _ : Unit. #t,
      int/c = flat/c(λ _ : Int. #t),
      fun/c = λ ctx : ctx ctc. (int/c : Int → (ctx) int/c),
      any/ctx = ctx/c(true,true),
      any/c = (fun/c any/ctx),
      p = make-parameter #f,
      check/ctx = ctx/c(true,λ _ : Unit. ?p),
      enable/ctx = ctx/c(true,true,((true ⇒ p ← true))),
      enable/c = (fun/c enable/ctx),
      inner/c = (fun/c check/ctx),
      outer/c = (any/c : (Int → Int) → (any/ctx) enable/c);
(inner 42)
```

Figure 4. Context contracts enforcing nested applications.

Figure 4 demonstrates context contracts with a small example. The example involves two context contracts, $outer/ctx$ and $inner/ctx$, that communicate via parameter p. The contracts ensure that function $inner$ can be applied only in the dynamic extent of the function returned by $outer$. Evaluating $(inner\ 42)$ results in a contract error $\text{error}_{\ell}^{top}$ blaming the context that applied $inner$. Replacing this expression with $((outer\ inner)\ 42)$ evaluates to 42.

The ability to close over an environment is a key feature of authorization contracts. To see that context contracts can close over some part of the environment when a contract is applied, consider extending the example in Figure 4 with the contract $capture/c$ from Figure 5. This contract captures the value of parameter p when the contract is applied, and reinstates that value for the dynamic extent of subsequent applications of the contracted value.

3.1.5 Complete monitoring

Our contract system satisfies *complete monitoring* [8], an important correctness criterion for contract systems. Complete monitoring guarantees that a contract system correctly assigns blame to components that violate their contracts and, crucially, that the contract system can interpose on all uses of a value in a component that did not create that value. This property makes contracts suitable for interposing on programs to enforce access control policies. Moreover, because

```
module ℓ exports ...
where ...,
        cp = make-parameter #f,
        capture/ctx = ctx/c(true,
                            ((true ⇒ cp ← λ_ : Unit. ?p)),
                            true,
                            ((true ⇒ p ← λ_ : Unit. ?cp)))
        ...;
...
```

Figure 5. A context contract that closes over parameter p.

the interposition is local to individual components, an access control monitor can be installed around a component without a global enforcement mechanism or the cooperation of other components.

Put differently, complete monitoring guarantees that contracts can enforce the same set of properties as reference monitors: an arbitrary prefix-closed property of a sequence of events. For contracts, these events are the attachment of contracts to values and the use of contracted values. In contrast, for inlined reference monitors built with aspects, this set of events is determined by the point-cuts selected by the policy. In either case, the programmer must correctly identify relevant events and specify the policy, but can assume the policy is enforced.

The formal definition and proof of complete monitoring for our contract system can be found in the accompanying technical report [28].

3.2 Representing Authority

In principle, a programmer can use context contracts to enforce arbitrary properties of execution contexts such as access control, but in practice this requires the careful design of an appropriate representation of the relevant information as an environment, i.e., a set of parameters. In particular, for access control this requires a representation of the authority of an execution context.

The authority of an execution context describes the rights it has to perform sensitive operations. In different access control mechanisms, the form and organization of these rights varies. For example, in a web application, a session executes on behalf of a particular user whose rights may change over time in accordance with the access control policies attached to the application's resources. In the Java stack inspection framework, rights are sets of "permissions" possessed by activation records that can be queried with the `checkPermission` operation.

A common way to describe the structure of authority in an access control system involves a mapping from *subjects* (users, processes, or security domains) to access rights for objects (resources that require the protection of the access control system) [21]. In practice, subjects and objects may comprise the same entities, so we refer to both as *security principals* (or, simply, *principals*).

To build a framework that supports many different access control mechanisms, we need a general way to express

$$\begin{array}{lrcl}
\text{Primitives} & \mathcal{P} & ::= & a \mid b \mid \dots \\
\text{Projection dimensions} & d & ::= & \alpha \mid \beta \mid \dots \\
\text{Principals} & p, q, r, s & ::= & \mathcal{P} \mid \top \mid \bot \\
 & & \mid & p \wedge p \mid p \vee p \\
 & & \mid & p \triangleright d \mid \underset{D}{\overleftarrow{\ }}p \mid \underset{D}{\overrightarrow{\ }}p \\
\text{Delegations} & A & ::= & p \succeq p \mathbin{@} p \\
\text{Delegation sets} & D & ::= & \{A, A, \dots\}
\end{array}$$

Figure 6. Syntax of principals, delegations, and worlds

and reason about principals, the authority of principals, and how principals delegate and restrict their authority. For this purpose, we use an authorization logic [2] based on the Flow-Limited Authorization Model (FLAM) [5]. We briefly describe our logic, highlighting where it differs from FLAM. In Section 3.3, we employ this logic to represent authority as a set of parameters that are managed by specialized context contracts, dubbed authorization contracts.

Figure 6 presents the syntax of our logic. We assume an enumerable set of *primitive principals* $\mathcal{P}$. Primitive principals represent program entities that possess rights, such as users, modules, or activation records. We assume a most trusted principal $\top$ and a least trusted principal $\bot$. For principals p and q, the conjunctive principal $p \wedge q$ is a principal with the authority of *both* p and q. Similarly, the disjunctive principal $p \vee q$ has the authority of *either* p or q.

If principal p trusts principal q, we write $q \succeq p$, and say that q *acts for* p. The acts-for relation is reflexive and transitive, and induces a lattice structure over the set of principals, with conjunction as join, disjunction as meet, and $\top$ and $\bot$ as the top and bottom elements of the lattice.

Principals may assert the existence of trust relationships. A *delegation* $p \succeq q \mathbin{@} r$ means that principal r asserts that p acts for q (or, equivalently, that q delegates its authority to p). Of course, whether a principal s believes the assertion depends on whether s trusts principal r. (We differ from FLAM in that we describe only the integrity of delegations, not their confidentiality.)

Judgment $D\,;r \vdash p \succeq q$ denotes that given the set of delegations D, principal r believes that principal p acts for principal q. Intuitively, r believes that p acts for q if that trust relationship can be derived using only delegations asserted by principals that r trusts.

Figure 7 presents the inference rules for the judgment $D\,;r \vdash p \succeq q$. Rules Bot, Top, Refl, Trans, Conj-Left, Conj-Right, Disj-Left, and Disj-Right are standard and provide the underlying lattice structure for the acts-for relation. Rule Del captures the intuition that principal r trusts only delegations asserted by principals that it trusts, that is delegations $p \succeq q \mathbin{@} s$ where $D\,;r \vdash s \succeq r$

We have three additional principal constructors. Principal $p \triangleright \alpha$ is the *projection* of the authority of principal p on dimension α[3]. We use projections to limit or attenuate the

[3] FLAM considers *basis projections* and *ownership projections*. The projections we use here are more general and have less structure than either. In addition to preserving the acts-for lattice, the only structure we impose is that projections are commutative: $p \triangleright \alpha \triangleright \beta = p \triangleright \beta \triangleright \alpha$.

$$\textsc{Bot}\quad \frac{}{D\,;r \vdash p \succeq \bot} \qquad \textsc{Top}\quad \frac{}{D\,;r \vdash \top \succeq p} \qquad \textsc{Proj}\quad \frac{}{D\,;r \vdash p \succeq p \triangleright \alpha}$$

$$\textsc{Conj-Left}\quad \frac{D\,;r \vdash p_k \succeq q \qquad k \in \{1,2\}}{D\,;r \vdash p_1 \wedge p_2 \succeq q} \qquad \textsc{Conj-Right}\quad \frac{D\,;r \vdash p \succeq q_1 \qquad D\,;r \vdash p \succeq q_2}{D\,;r \vdash p \succeq q_1 \wedge q_2}$$

$$\textsc{Disj-Left}\quad \frac{D\,;r \vdash p_1 \succeq q \qquad D\,;r \vdash p_2 \succeq q}{D\,;r \vdash p_1 \vee p_2 \succeq q} \qquad \textsc{Disj-Right}\quad \frac{D\,;r \vdash p \succeq q_k \qquad k \in \{1,2\}}{D\,;r \vdash p \succeq q_1 \vee q_2} \qquad \textsc{Del}\quad \frac{p \succeq q \mathbin{@} s \in D \qquad D\,;r \vdash s \succeq r}{D\,;r \vdash p \succeq q}$$

$$\textsc{Closure-Left}\quad \frac{D'\,;s \vdash p \succeq q \qquad D\,;r \vdash \overleftarrow{\scriptstyle D'}s \succeq r}{D\,;r \vdash p \succeq \overleftarrow{\scriptstyle D'}q} \qquad \textsc{Closure-Right}\quad \frac{D'\,;s \vdash p \succeq q \qquad D\,;r \vdash \overleftarrow{\scriptstyle D'}s \succeq r}{D\,;r \vdash \overrightarrow{\scriptstyle D'}p \succeq q}$$

$$\textsc{Refl}\quad \frac{}{D\,;r \vdash p \succeq p} \qquad \textsc{Trans}\quad \frac{D\,;r \vdash p \succeq q \qquad D\,;r \vdash q \succeq s}{D\,;r \vdash p \succeq s}$$

Figure 7. Inference rules for judgment $D\,;l \vdash p \succeq q$

authority of a principal, and to identify access rights. For example, $p \triangleright files$ may refer to principal p's authority restricted to p's rights to access the file system. Similarly, principal $p \triangleright obj \triangleright invoke$ (equivalently $p \triangleright invoke \triangleright obj$) might refer to the right to invoke a particular object belonging to principal p. Principal p can grant this right to another principal q by asserting a delegation: $q \succeq p \triangleright obj \triangleright invoke \mathbin{@} p$.

We leave projection dimensions underspecified, and access control mechanisms can define their own dimensions. For any projection dimension α, principal p acts for principal $p \triangleright \alpha$, as captured in Rule Proj. Typically the converse does not hold, and so $p \triangleright \alpha$ has strictly less authority than p.

Novel to this work, we introduce *closure principals* $\overleftarrow{\scriptstyle D}p$ and $\overrightarrow{\scriptstyle D}p$. Given a set of delegations D and principal p, the *left-closure principal* $\overleftarrow{\scriptstyle D}p$ represents p with all of the trust relationships derivable from D where p delegates its authority to other principals. The *right-closure principal* $\overrightarrow{\scriptstyle D}p$ represents p with all of the trust relationships derivable from D where p acts for other principals. In our framework, delegations may change over time. Closure principals are useful because they allow us to capture trust relationships as they exist at particular moments in time. In particular, closure principals are a principled mechanism to describe how authority captured by a context contract should be combined with the current authority environment based on which parts of the closed over authority environment are trusted by principals in the current authority environment.

Rule Closure-Left shows that $D\,;r \vdash p \succeq \overleftarrow{\scriptstyle D'}q$ holds when there is some principal s such that at the time of closure creation (i.e., with delegation set D'), s believed that p acted for q (premise $D'\,;s \vdash p \succeq q$), and moreover, right now (i.e., with delegation set D) principal r trusts principal $\overleftarrow{\scriptstyle D'}s$ (premise $D\,;r \vdash \overleftarrow{\scriptstyle D'}s \succeq r$). Typically, s and r are the same principal, meaning that r-at-time-D trusts the decisions made by r-at-time-D'. Rule Closure-Right is similar and $D\,;r \vdash \overrightarrow{\scriptstyle D'}p \succeq q$ holds when there is some principal s such that at the time the closure was taken s believed that p acted for q (premise $D'\,;s \vdash p \succeq q$), and principal r trusts s-at-time-D' (premise $D\,;r \vdash \overleftarrow{\scriptstyle D'}s \succeq r$).

To query whether a particular set of delegations satisfies an acts-for relation, we use a proof search algorithm adapted from FLAM [5]. We give examples of using delegations to implement different authorization mechanisms in Section 4.

Based on this logic, we represent an authority environment as:

1. a *principal*, who is responsible for the current execution context, and
2. a *delegation set*, which records the current trust relationships between principals.

The latter has two sub-parts: a global, mutable delegation set, and a set of delegations that are in place only for a currently executing context.

3.3 Authorization Contracts

Using authority environments, we can now introduce authorization contracts. Authorization contracts specialize context contracts in two ways. First, they prevent interference from untrustworthy code by using parameters that the rest of the program does not have access to. Second, they use a high-level representation of authority environments rather than directly manipulating parameters. Authorization contracts provide a structured way to describe how the underlying context contracts should manipulate authority environments.

Authorization contracts are defined as monitor actions using the `define-monitor` form (§2). In this section, we model a pared-down version of `define-monitor` as an extension to the language model from Section 3.1. Figure 8 displays the syntax of the extension. The extension introduces new types, constructors, and operations for principals, delegations, and delegations sets, including an expression that evaluates an acts-for judgment: $e\,;e \vdash e \succeq e$. The `define-monitor` form corresponds to the monitor $(a \ldots)$ form that can appear in the where clause of a module definition in the extended model. Each a in monitor $(a \ldots)$ is an action specification. An action specification action x $(y : \tau, \ldots)$ (ce,ae) has a name (x), a set of arguments $(y : \tau, \ldots)$, and two terms (ce and ae) that define the action's `#:on-create` and `#:on-apply` hooks.

The term for the `#:on-create` hook has the form

check: cee add: cee remove: cee set!-principal: cee
closure-principal: cee closure-delegations: cee

and specifies what the authorization contract should do when the contract is applied to a value. In particular it describes

$$
\begin{array}{rcl}
m & ::= & \textsf{module}\ \ell\ \textsf{exports}\ x\ \textsf{with}\ x,\ldots\ \textsf{where}\ x = e,\ldots,\textsf{monitor}\ (a,\ldots),x = e,\ldots \\
v & ::= & \ldots \mid \top \mid \bot \mid \mathcal{P} \mid \mathcal{D} \mid v \succeq v\ @\ v \mid \{v,\ldots\} \\
e & ::= & \ldots \mid \textsf{new-principal} \mid \textsf{new-dimension} \mid e \rhd e \mid e\,;e \vdash e \succeq e \mid e \succeq e\ @\ e \mid \textsf{let}\ x \succeq x\ @\ x = e\ \textsf{in}\ e \\
 & \mid & \{\} \mid \{e\} \mid e \cup e \mid e \setminus e \mid (\textsf{fold}\ e\ e\ e) \\
\beta & ::= & \ldots \mid \textsf{Prin} \mid \textsf{Dim} \mid \textsf{Del} \mid \textsf{DelSet} \\
a & ::= & \textsf{action}\ x\ (y : \tau,\ldots)\ (ce,ae) \\
ce & ::= & \textsf{check:}\ cee\ \textsf{add:}\ cee\ \textsf{remove:}\ cee\ \textsf{set!-principal:}\ cee\ \textsf{closure-principal:}\ cee\ \textsf{closure-delegations:}\ cee \\
ae & ::= & \textsf{check:}\ aee\ \textsf{add:}\ aee\ \textsf{remove:}\ aee\ \textsf{scope:}\ aee\ \textsf{set-principal?:}\ aee\ \textsf{principal:}\ aee\ \textsf{set!-principal:}\ aee \\
cee & ::= & e \mid \textsf{let}\ x = cee\ \textsf{in}\ cee \mid \textsf{current-principal} \mid \textsf{current-delegations} \\
aee & ::= & e \mid \textsf{let}\ x = aee\ \textsf{in}\ aee \mid \textsf{current-principal} \mid \textsf{current-delegations} \mid \textsf{closure-principal} \mid \textsf{closure-delegations}
\end{array}
$$

Figure 8. Syntax extensions for authorization contracts.

how to modify each part of the authority environment. Its field check accepts an acts-for judgment. If this judgement does not hold, a contract error is raised blaming the client of the contract. Field add accepts a set of delegations to add to the global delegation set. Field remove accepts a set of delegations to remove from the global delegation set, if present. Field set!-principal changes the current principal to the given principal. Fields closure-principal and closure-delegations accept a principal and a set of delegations, respectively, and record the principal and delegations for use upon a call to the contracted function. Terms in each of these six fields can access the pieces of the current authority environment using current-principal and current-delegations.

The second term corresponds to the `#:on-apply` hook, which specifies what the authorization contract should do upon a call of the contracted function. It has the form

check: cee add: cee remove: cee scope: cee
set-principal?: cee principal: cee set!-principal: cee.

Similar to a ce term, it allows the configuration of the contract's behavior. As before, check accepts an acts-for judgment and raises a contract error if it does not hold. Likewise, fields add and remove mutate the global delegation set, and field set!-principal changes the current principal. The scope field accepts a set of delegations, but this set is installed only for the dynamic extent of the current function call, rather than added to the global delegation set. The set-principal? field requires a boolean value. If that value is #t, the current authorization environment is extended with a principal for the dynamic extent of the function call. This (1) allows changing the principal visible within the extent of the function call and (2) prevents contracts that change the principal during the extent of the function call from modifying the principal of the enclosing context. In addition to accessing the current principal and delegations from the authority environment, the seven fields of an ae term can access the principal and delegations closed over by the contract with terms closure-principal and closure-delegations.

To give a detailed semantics for monitor terms, we use a compilation function that replaces monitor expressions with terms that explicitly construct context contracts. The compilation uses five parameters: one each for the current principal, global delegation set, and scoped delegation set, plus a pair to record the closed-over principal and delegations. Each monitor term generates a fresh set of parameters, preventing separately defined monitors from interfering with each other. Each action term compiles to a single context contract that closes over the fresh parameters. The full compilation function is listed in Appendix A, along with typing judgments for authorization contracts and the semantics of expressions that operate on principals, dimensions, delegations, and delegation sets.

The hooks for defining actions are sufficiently flexible to implement a variety of access control mechanisms (§4). Here, we briefly describe some of the ways programmers can configure authorization contracts.

Mutable Authority Many access control mechanisms have a global policy that changes over time. For example, in discretionary access control, users can grant or revoke access to their resources. We can implement this with a contract that adds or removes (global) delegations.

Dynamically-scoped Authority An authority closure can inherit the authority environment from its calling context by ignoring the authority environment it closes over.

"Lexically"-scoped Authority An authority closure can isolate itself from the authority of its calling context by replacing the authority environment at a call site with the authority that it closes over.

Moreover, different access control mechanisms may require authorization contracts that blend these different strategies. For example, implementing `setuid`-like authority closures requires capturing the principal but not the delegations the closures close over. Otherwise, updates to the global, mutable discretionary access control policy would be forgotten when a `setuid` function runs.

4. Putting Authorization Contracts to Work

As evidence of the usefulness and expressiveness of the framework, we implemented a variety of existing access control mechanisms including discretionary access control, stack

```
(define-monitor monitor-name
  (monitor-interface
    action-name ... extra-name ...)
  (monitor-syntax-interface
    syntax-name ...)
  (action ; definitions of monitor actions
    [action-name (action-var ...)
      #:on-create on-create-hook
      #:on-apply  on-apply-hook]
      ... )
  (extra  ; additional monitor abstractions
    (define extra-name extra-body)
    ... )
  (syntax ; syntactic monitor abstractions
    (define-syntax syntax-name syntax-body)
    ... ))
```

Figure 9. The `define-monitor` form

inspection [40], history-based access control [1], and object capabilities [25]. Here, we focus on stack inspection to demonstrate how to use authorization contracts to design a complex access control monitor. The remaining access control mechanisms are described in the accompanying technical report [28]. Before delving into stack inspection, we further explain `define-monitor`, the main linguistic tool that our framework provides.

4.1 The `define-monitor` Form

Figure 9 shows the complete syntax of `define-monitor`. It has two sections in addition to the `action` section we have seen before: `extra` and `syntax`. The first defines extra functions and contracts that the programmer wants to include in the interface of a monitor. These are usually contracts that combine two or more actions together or contracts that fix the arguments of an action. The `syntax` section defines macros that serve as syntactic abstractions over the monitor's interface, for example, to automate the placement of authority contracts when defining a function. We give examples of definitions in the `extra` and `syntax` sections later. The `monitor-interface` and `monitor-syntax-interface` clauses specify which elements are available to users of the monitor. After defining a monitor `monitor-name`, a client can instantiate it with `(run monitor-name)`. This creates a fresh monitor, i.e., one with a fresh authority environment and contracts.

The most complicated part of defining a monitor is writing the two hooks for each monitor action. To facilitate this, we provide two functions, `do-create` and `do-apply`, that simplify this process. Each function has optional keyword arguments corresponding to the fields of an action form in Section 3.3. The functions provide default values for any argument not specified. Thus, the programmer need only specify the results of the hooks they care about. For instance, the default value for the argument with keyword `#:check` seen in Figure 1 is an acts-for relation that is always true.

4.2 A Stack Inspection Monitor

In stack inspection [40], code obtains permissions based on static properties such as the package it belongs to. At run time, code can choose to enable its static permissions making them eligible for satisfying an access control check. Before a sensitive operation, stack inspection checks for the presence of a particular permission by walking the run-time call stack until a frame from code that has enabled the permission is found. To prevent *luring attacks* [40], stack inspection additionally requires that all execution contexts between the enabled permission and the authorization check have the required static permission. Despite this protection, untrusted code may be able to influence the program even if its frames are no longer on the stack. As a result, modern adaptations of stack inspection provide additional support for capturing the permissions of the stack at some point in an execution and reinstating them for a later check.

Implementations of stack inspection provide the following primitives: `checkPermission`, which checks that a frame on the stack has the required permission enabled and that all intervening frames have the required static permission; `doPrivileged`, which enables the static permissions of the current code for its dynamic extent, possibly using captured permissions instead of the current permissions; and `getContext`, which captures the permissions of the stack at some point in execution. In addition, the implementation must provide a mechanism to associate static permissions with code.

To realize stack inspection using authorization contracts, a monitor must provide (1) actions that implement these primitives and (2) a way to grant static permissions to code. In our monitor, the actions for (1) are `check-permission/c`, `do-privileged/c`, and `context/c`. To track which permissions are held by code on the stack, we use the authority environment to grant permissions to individual frames, each represented by a distinct principal. Each stack frame has three projections that are used to manage its authority. The `static` projection indicates the permissions granted to the code statically. The `enable` projection has the authority of the permissions enabled for this frame. The `active` projection represents permissions that would satisfy a privilege check. We say a principal has a particular permission if it acts for the corresponding projection of the $\top$ principal.

We use one additional monitor action, `privileged/c`, to indicate the static permissions a piece of code possesses and to enforce that a stack frame's `active` projection acts for exactly those permissions for which `checkPermission` should succeed. Action `privileged/c` takes a list of permissions (each of which is a projection of the $\top$ principal). On an `#:on-apply` event, it creates a new principal `callee` to represent the new stack frame and adds delegations initializing these projections for the dynamic extent of the function:

```
(⪰ @ (▷ callee static) permissions ⊤)
(⪰ @ (▷ callee enable)
     (▷ current-principal active)
      current-principal)
(⪰ @ (▷ callee active)
     (∨ (▷ callee enable) (▷ callee static))
     callee).
```

These delegations give callee the specified static permissions (by asserting that the callee's static dimension acts for the conjunctive principle permissions), assert that the new frame inherits the active permissions from the previous frame, and require that the callee has both static and enabled permissions to make them active.

Tracking the authority of each frame in this way makes walking the stack unnecessary. Action check-permission/c only checks that the active projection of the current principal acts for all of the requested permissions.

Action do-privileged/c enables the current frame's static permissions by adding a delegation from the frame's static projection to its enable projection for the dynamic extent of the wrapped function.

Action context/c is used to capture the permissions of the current stack for future permission checks. It captures the current authorization environment when it is attached to a function. When it is invoked, it installs the same set of delegations as privileged/c, except that the first delegation that grants static permissions gets replaced with a delegation that derives permissions from the active permissions of the captured frame at the time they were captured:

```
(⪰ @ (▷ callee static)
   (▷ (→ closure-principal
         closure-delegations)
   active) ⊤).
```

The right-closure principal on the right hand side of this delegation acts for all of the principals that closure-principal acted for when the closure was created.

The monitor must also provide (2) a way to grant static permissions to code. Because Racket does not have class-loading facilities that would allow permissions to be granted to code at load-time, we use macros to attach authorization contracts to code that should have static permissions. In particular, the monitor provides a new definition form define/rights in its **syntax** section. This form works like the **define** form, but takes two additional arguments: a set of permissions and a contract to apply to the definition. It defines a function wrapped with the given contract and a privileged/c contract. In addition, the macro define/rights coerces any function arguments or free-variables appearing in the body of the function to authority closures by applying an additional contract unprivileged/c, which is defined in the **extra** section of the monitor. Action unprivileged/c switches to the ⊥ principal for the dynamic extent of the closure it wraps, preventing any check-permission/c actions from succeeding. Thus, these contracts prevent functions that were not defined with define/rights from using code that requires permissions.

```
(define/rights (read-file file) (filesys)
  (check-permission/c filesys)
  ...)

(define/rights (read-privileged file)
    (filesys)
  do-privileged/c
  (if (safe? file) (read-file file) #f))

(define/rights (malicious) (net)
  any/c
  (read-file "/etc/passwd"))

> (malicious)
; read-file: contract violation;
;  (▷ frame10247 active) ⋡ (▷ ⊤ filesys)
;    @ (▷ ⊤ filesys)
;   contract from: (definition read-file)
;   blaming: top-level
```

Figure 10. Using the stack inspection monitor

Figure 10 shows an example program using the stack inspection monitor. There are three functions defined using define/rights. Two of these functions are trusted to access the filesystem: read-file and read-privileged. However, read-file should not be used directly, so it checks that the filesys permission has been enabled by one of its callers. Function read-privileged enables the filesys permission, but only calls read-file if the file is safe to read. Function malicious does not have the filesys permission but attempts to read "/etc/passwd" anyway, so invoking this function results in a contract violation. The contract violation says that the stack frame corresponding to the call to read-file does not have the necessary permission filesys.

5. Case studies

To evaluate the use of our framework in practical applications, we developed three case studies. The first adds simple authorization contracts to the implementation of a card game to ensure that player's moves affect only the parts of the game state they control. The second secures a plugin interface of the DrRacket development environment and demonstrates how the flexibility of the framework can support complex security mechanisms. The third, which mirrors the example from Section 2.2, replaces authorization checks in a web application with authorization contracts.

We evaluated the performance of our framework on each case study. The experiments were conducted on a MacBook Pro with a 2.6 GHz Intel Core i5 and 16GB of RAM running Mac OS X 10.11 and Racket 6.4.0.9. In the first two case studies, authorization contracts have significant impact on the performance of the benchmarks. However, both case studies are worst case scenarios: they have no existing code implementing access control (and so we are strictly adding functionality), and after adding contracts, they invoke many access control checks (tens of thousands in the case of the card game) while performing cheap operations. Moreover, in the DrRacket case study, the absolute overhead for each benchmark due to authorization contracts is less than 45ms, but the relative overhead is high since the baseline running time is less than 15ms. The third case study replaces existing access control checks with authorization contracts, with negligible impact on performance. Our implementation is a prototype, and we anticipate that optimizations in the implementation of our contracts can further reduce their overhead.

Preventing Cheating in a Card Game We have used authorization contracts to enforce a security policy for a functional implementation of the card game Dominion[4]. The exact rules of Dominion do not matter for our purpose, except that each player collects cards in a local deck and attempts to outscore the rest of the players by playing cards from their deck. During each turn, players can play cards from their deck to either purchase additional cards or attack other players, forcing them to discard some of their cards.

In this implementation, each player is a program that runs in its own process and responds automatically to messages from a central broker. The broker maintains the shared inventory of cards and a mirror of each player's local deck. Players perform moves by sending messages to the broker describing the move.

To perform a move, the player sends a message to the broker identifying a card to play. In response, the broker updates its copy of the game state to reflect the move and, if the move involves an attack on another player, informs the other player of the attack. The other player then has an opportunity to defend by choosing which card to discard and the broker again updates the game state.

The broker represents the local deck of each player as an immutable record `player` and the state of the game as an immutable structure `game` that holds a list of `player` records. The first element in this list corresponds to the player who makes the next move. The broker is implemented as a core `drive` function that delegates to two functions: `move` and `defend`. Both functions perform functional updates to the relevant structures.

We enforce the policy that the broker only updates the current player's deck or a defending player's deck. The monitor that enforces this policy specifies three authorization contracts: `deprivilege/c`, which sets the principal for the dynamic extent of a function to $\perp$; `(switch-player/c name)`, which sets the principal for the dynamic extent of a function to the player with name `name`; and `(check-player/c name)`, which checks before calling a function if the current principal is the player with name `name`.

[4] The implementation is part of the teaching material of a long running undergraduate Functional Programming course.

To enforce the policy, we attach contract `deprivilege/c` to the function `drive` so that only authorized code can modify the game state during the game. The contract for the `game` structure, `game/c`, gives the accessor functions of each field of the player records in the `game` the contract`(check-player/c name)`, where `name` is the name of the corresponding player. The contract for the `move` function is

```
(->a
  ([game game/c] [turn any/c] [play any/c])
  #:auth (game)
    (switch-player/c
      (player-name
        (first (game-players game))))
  (values [game-result-game game/c]
          [turn-result any/c]))
```

and it authorizes the `move` function to act on behalf of the current player, i.e., `(first (game-players game))`. The contract for `defend` is

```
(->a ([player player/c] [defense any/c])
     #:auth (player)
       (switch-player/c
         (player-name player))
     [result player/c])
```

which similarly allows the function to update the state of the player who was attacked.

We created 10 benchmarks for the Dominion case study that each consists of a simulated game with 2-7 players. Adding authorization contracts increases running time by 1.3–1.7× at both the median and 99th percentile.

Securing a Plugin Interface We wrote a monitor to protect DrRacket from malicious or buggy third-party key bindings. First, we explain aspects of DrRacket's design related to key bindings. Keystrokes sent to DrRacket are dispatched as method calls to a `text%` object which encapsulates the state of the editor. This object has methods that access and modify parts of DrRacket. For instance, the `get-text` method returns the content of the editor, while the `set-padding` method changes the inset padding used to display the editor's content. Each `text%` object has a `keymap%` object that stores registered key bindings and maps sequences of keystrokes to the action they trigger. A keybinding action is an arbitrary Racket function of two arguments: the current `text%` object and an `event%` object, which describes the event that triggered the action. On startup, DrRacket populates its `text%` object's key map with built-in key bindings. In addition, DrRacket registers user-defined key bindings from configuration files. Keybinding actions can inspect and modify almost any aspect of DrRacket through the `text%` object. This gives users a powerful interface for customizing DrRacket but makes key bindings a source of vulnerabilities. For instance, a key binding could accidentally erase the user's code or snoop on the editing session.

Our monitor restricts which `text%` object methods a keybinding action can invoke. We group methods of `text%` that can access or modify similar parts of DrRacket. For instance, methods that write to the clipboard (e.g. cut and copy) belong to the same group while methods that change how DrRacket displays content (e.g. set-max-width and set-line-spacing) belong to a second group. Each group has a corresponding privilege that is required to invoke the group's methods. For example, the privileges ReadClipboard and ChangeEditorView grant access to the methods mentioned above. Methods can belong to multiple groups. Access control checks around each method verify that the authority of a calling execution context has the necessary privileges.

In addition to methods that require specific privileges to invoke, text% has sensitive methods that should be invoked only by another method of the text% object. For example, the on-delete method should never be invoked directly as its correctness depends on DrRacket's state. Instead, key bindings should invoke the delete method that subsequently calls on-delete. To support this use case, we require an additional privilege to call on-delete that is granted during the dynamic extent of delete.

The stack-inspection-like access control mechanism we have described so far is not sufficient. Some methods of text% install callbacks that are triggered by subsequent events. For example, add-undo registers a callback that runs when the user wishes to undo the action of a key binding. This callback should not run with the authority of its calling context, but instead should use the privileges of the action that created it. To achieve this, we create authority closures around any callbacks registered by an action.

Our monitor represents each privilege as a unique principal and represents sets of principals as conjunctions and disjunctions of principals. It defines three actions: check/c, enable/c, and closure/c. Upon an #:on-apply event, the first action consumes principal perms and checks if the current principal has permissions that imply perms. Then the action switches the current principal to a principal that only has permissions perms. When a function wrapped with enable/c is applied, it switches the current principal to a principal that has the same permissions as the current principal augmented with perms. The #:on-create event of the third action creates an authority closure. When the authority closure is applied, it installs the closed-over principal.

We use the actions of the monitor to define an authorization contract for the keybinding interface:

```
(->a ([t text/c] [e (is-a?/c event%)])
     #:auth () (check/c perms) any)
```

where perms is a principal which encodes the privileges we grant to the key binding and text/c is the object contract we define for the editor's text% object. text/c applies a contract to each method of text% specifying whether the method enables some permission, requires some permissions, or creates an authority closure around one of its arguments. For example, text/c gives the method blink-caret the contract (check/c ChangeEditorView). In essence, text/c defines a security policy for the editor.

To assess the monitor's performance, we ran a series of 30 benchmarks, adapted from DrRacket's test suite, that simulate a sequence of keystrokes that trigger built-in key bindings. We ran these benchmarks with the monitor off and on. When the monitor is on, the prototype grants the minimum set of privileges necessary for each key binding. For each benchmark, we measured the time required to retrieve and execute each key binding. Our measurements show that the authority monitor increases median response time by 3–7$\times$ and increases response time at the 99th percentile by 3–5$\times$. However, for an IDE, a response time fast enough for interactive use is more important. Our prototype achieves this goal with a maximum response time of 53ms.

Authentication in a Web Application The Racket package system allows users to discover and install packages from a public index service. Individual users can add new packages or update old ones by logging into the index service web application, which is implemented using the Racket web-server. Requests to add or modify packages are issued to the application as asynchronous http requests. The baseline implementation of the application uses macros to authenticate the user and perform any required access control checks before processing the request. For example, the jsonp/pkg/modify endpoint authenticates the current user and checks that they are an author of the package they are attempting to modify. This approach to access control is brittle, since it requires that the checks included for each endpoint accurately capture the privileges required when processing the response.

Using authorization contracts, we are able to separate the tasks of authentication and authorization in the index service web application. Rather than performing a different set of access control checks for each endpoint, all endpoints now simply invoke an authenticate function that checks whether the current session is valid and which user is logged in, then invokes a procedure to process the request, like the login function from Section 2.2. The access control policies for sensitive operations like updating a package are enforced by adding authorization contracts that implement the necessary checks to the web application's data model. There are two types of checks: (is-author/c pkg) which checks that the logged in user is an author of package pkg, and is-curator/c, which checks whether the logged in user has "curator" status, which allows them to tag packages with information about their quality.

To evaluate the new implementation's performance, we measured the latency of 1,000 repeated requests to modify a package record. Replacing inline checks with authorization contracts has minimal impact on performance. Median latency was 283ms for the baseline implementation versus 281ms with authorization contracts. At the 99th percentile, using authorization contracts latency was 338ms versus 330ms with the baseline implementation.

6. Related Work

The connection between scoping and access control has been implicit in prior work on security in programming languages but has never been a central concept for extensible access control. Morris's seminal paper "Protection in Programming Languages" [29] describes how lexical scope can be used to create security abstractions within a program. More recently, the object-capability paradigm has embraced lexical scope as an organizing security principle [25]. Wallach and Felten [39] note that "in some ways, [stack inspection] resembles dynamic variables (where free variables are resolved from the caller's environment rather than from the environment in which the function is defined)." Phung et al. [30] use dynamic and lexical scoping to associate principals with executing code in order to correctly enforce security policies on programs that mix JavaScript and ActionScript code.

Inlined Reference Monitors An alternative approach to language-level access control is inlined reference monitoring. Reference monitors observe the actions taken by a system and intercede to prevent violations of a security policy [3]. They can enforce a large class of policies [32]. Inlined reference monitoring (IRM) weaves the implementation of a reference monitor into the program being monitored [12]. Many implementations of inlined reference monitoring rely on aspects to identify security relevant actions during program execution [6, 7, 13, 14, 20]. Policies supported by these tools typically focus on access patterns for sensitive resources. While policies supported by our framework can be encoded this way, as in Erlingsson and Schneider's IRM implementation of Java stack inspection [13], policies where the authority of code depends on application state require duplicating code. A further disadvantage of IRMs is that they require a global transformation of the program to inline the security monitor. Because authorization contracts are applied at component boundaries, our framework requires only local modifications.

Authorization Logics Authorization logics give a formal language to express access control policies [1]. Authorization logics have been used to understand existing access control mechanisms, including Java stack inspection [39]. Aura [19] and Fine [36] implement access control using proof-carrying authentication, where proofs of formulas in an authorization logic are used as capabilities [4]. Our access control logic is inspired by the Flow-Limited Authorization Model [5], which uses projections to describe attenuated authority without requiring additional constructs such as roles or groups.

Contracts for Security Previous work has used contracts to enforce limited access control policies. Moore et al. [27] use contracts to constrain the use of capabilities in a secure shell scripting language. Dimoulas et al. [9] use contracts to control the flow of capabilities between components in object-capability languages. Heidegger et al. [18] use contracts to specify which fields of an object may be accessed by a component. However, each of these systems is specialized to enforce a specific type of access control policy. Disney et al. [10] introduce temporal higher-order contracts that enforce that sequences of function calls and returns match a specification. Schollier et al.'s computational contracts [33] can enforce a wide range of trace properties on programs. Unlike authorization contracts and temporal higher-order contracts, computational contracts use aspects to interpose on program events. Both of these systems support arbitrarily powerful monitors, but like inlined reference monitoring, provide limited support for writing complex access control policies like stack inspection or discretionary access control.

Scoped Aspects for Security Dutchyn et al. [11] enforce simple access control policies with lexically and dynamically-scoped aspects. With additional aspect scoping mechanisms, Toledo et al. [38] encode full Java stack inspection. While aspects can enforce a wide range of access control mechanisms, authorization contracts offer linguistic support for implementing diverse (and customized) access control mechanisms with ease. Doing the same with aspects, if possible, requires brittle and complex encodings.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Numbers CCF-1438271, CNS-1524052, and CCF-1526324. This research is also supported by the Air Force Research Laboratory and a Google Faculty Research Award.

References

[1] M. Abadi and C. Fournet. Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS)*, pages 107–121, February 2003.

[2] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(4): 706–734, Sept. 1993.

[3] J. P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, U.S. Air Force Electronic Systems Division, Deputy for Command and Management Systems, HQ Electronic Systems Division, 1972.

[4] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *Proceedings of the 6th ACM Conference on Computer and Communications Security (CCS)*, pages 52–62, November 1999.

[5] O. Arden, J. Liu, and A. C. Myers. Flow-limited authorization. In *Proceedings of the 28th IEEE Computer Security Foundations Symposium (CSF)*, pages 569–583, July 2015.

[6] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 305–314, June 2005.

[7] F. Chen and G. Roşu. Java-MOP: A monitoring oriented programming environment for Java. In *Proceedings of the*

Eleventh International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pages 546–550, April 2005.

[8] C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Complete monitors for behavioral contracts. In *Proceedings of the 21st European Symposium on Programming (ESOP)*, pages 211–230, March 2012.

[9] C. Dimoulas, S. Moore, A. Askarov, and S. Chong. Declarative policies for capability control. In *Proceedings of the 27th IEEE Computer Security Foundations Symposium (CSF)*, pages 3–17, 2014.

[10] T. Disney, C. Flanagan, and J. McCarthy. Temporal higher-order contracts. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 176–188, September, 2011.

[11] C. Dutchyn, D. B. Tucker, and S. Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239, December 2006.

[12] U. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 Workshop on New Security Paradigms (NSPW)*, pages 87–95, 1999.

[13] U. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy (S&P)*, pages 246–255, May 2000.

[14] D. Evans and A. Twyman. Flexible policy-directed code safety. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy (S&P)*, pages 32–45, May 1999.

[15] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 48–59, October 2002.

[16] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. `http://racket-lang.org/tr1/`.

[17] M. Gasbichler and M. Sperber. Processes vs. user-level threads in Scsh. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Scheme and Functional Programming*, 2002.

[18] P. Heidegger, A. Bieniusa, and P. Thiemann. Access permission contracts for scripting languages. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 111–122, January 2012.

[19] L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. AURA: A programming language for authorization and audit. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 27–38, September 2008.

[20] M. Jones and K. W. Hamlen. Enforcing IRM security policies: Two case studies. In *Proceedings of the 7th IEEE Intelligence and Security Informatics Conference (ISI)*, pages 214–216, June 2009.

[21] B. W. Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, Jan. 1974.

[22] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[23] B. Meyer. Design by contract. In *Advances in Object-Oriented Software Engineering*, pages 1–50. Prentice Hall, 1991.

[24] B. Meyer. Applying “Design by Contract”. *Computer*, 25(10): 40–51, October 1992.

[25] M. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, May 2006.

[26] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript, 2008. Google white paper.

[27] S. Moore, C. Dimoulas, D. King, and S. Chong. Shill: A secure shell scripting language. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 183–199. USENIX, Oct. 2014.

[28] S. Moore, C. Dimoulas, R. B. Findler, M. Flatt, and S. Chong. Extensible access control with authorization contracts. Technical Report TR-03-16, Harvard University, 2016.

[29] J. H. Morris, Jr. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, Jan. 1973.

[30] P. H. Phung, M. Monshizadeh, M. Sridhar, K. W. Hamlen, and V. N. Venkatakrishnan. Between worlds: Securing mixed javascript/actionscript multi-party web content, 2015.

[31] J. H. Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7):388–402, July 1974.

[32] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1): 30–50, Feb. 2000.

[33] C. Schollier, É. Tanter, and W. D. Meuter. Computational contracts, 2013.

[34] G. L. Steele, Jr. Macaroni is better than spaghetti. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 60–66, August 1977.

[35] G. L. Steele Jr and G. J. Sussman. The revised report on SCHEME: A dialect of LISP. Technical Report AIM-452, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 1978.

[36] N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in Fine. In *Proceedings of the 19th European Conference on Programming Languages and Systems (ESOP)*, pages 529–549, March 2010.

[37] A. Takikawa, T. S. Strickland, and S. Tobin-Hochstadt. Constraining delimited control with contracts. In *Proceedings of the 22nd European Conference on Programming Languages and Systems (ESOP)*, pages 229–248, March 2013.

[38] R. Toledo, A. Nunez, E. Tanter, and J. Noye. Aspectizing Java access control. *IEEE Transactions on Software Engineering*, 38(1):101–117, Jan. 2012.

[39] D. Wallach and E. Felten. Understanding Java stack inspection. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy (S&P)*, pages 52–63, May 1998.

[40] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for Java. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 116–128, October 1997.

A. Details of model

A.1 Evaluation contexts

$$
\begin{array}{rcl}
E & ::= & [\cdot] \mid E\ e \mid v\ E \mid \mathsf{let}\,x = E\,\mathsf{in}\ e \mid \mathsf{if}\ E\ \mathsf{then}\ e\ \mathsf{else}\ e \mid E \oplus e \mid v \oplus E \mid E \leq e \mid v \leq E \\
& \mid & \mathsf{make\text{-}parameter}\ E \mid ?E \mid E := e \mid \mathsf{p}(r) := E \\
& \mid & \mathsf{parameterize}\,E = e\,\mathsf{in}\ e \mid \mathsf{parameterize}\,\mathsf{p}(r) = E\,\mathsf{in}\ e \mid \mathsf{parameterize}\,\mathsf{p}(r) = v\,\mathsf{in}\ E \\
& \mid & \mathsf{flat/c}(E) \mid \mathsf{param/c}(E) \mid E : \tau \rightarrow (e)\ e \mid c : \tau \rightarrow (E)\ e \mid c : \tau \rightarrow (c)\ E \\
& \mid & E : \tau \rightarrow_{\mathsf{a}} (\lambda\,x : \tau.\ e)\ e \mid c : \tau \rightarrow_{\mathsf{a}} (\lambda\,x : \tau.\ e)\ E \mid \mathsf{ctx/c}(E,(e \Rightarrow e \leftarrow e),\ldots,e,(e \Rightarrow e \leftarrow e),\ldots) \\
& \mid & \mathsf{ctx/c}(v,(v \Rightarrow v \leftarrow v),\ldots,(E \Rightarrow e \leftarrow e),(e \Rightarrow e \leftarrow e),\ldots,e,(e \Rightarrow e \leftarrow e),\ldots) \\
& \mid & \mathsf{ctx/c}(v,(v \Rightarrow v \leftarrow v),\ldots,(v \Rightarrow E \leftarrow e),(e \Rightarrow e \leftarrow e),\ldots,e,(e \Rightarrow e \leftarrow e),\ldots) \\
& \mid & \mathsf{ctx/c}(v,(v \Rightarrow v \leftarrow v),\ldots,(v \Rightarrow v \leftarrow E),(e \Rightarrow e \leftarrow e),\ldots,e,(e \Rightarrow e \leftarrow e),\ldots) \\
& \mid & \mathsf{ctx/c}(v,(v \Rightarrow v \leftarrow v),\ldots,E,(e \Rightarrow e \leftarrow e),\ldots) \\
& \mid & \mathsf{ctx/c}(v,(v \Rightarrow v \leftarrow v),\ldots,v,(v \Rightarrow v \leftarrow v),\ldots,(E \Rightarrow e \leftarrow e),(e \Rightarrow e \leftarrow e),\ldots) \\
& \mid & \mathsf{ctx/c}(v,(v \Rightarrow v \leftarrow v),\ldots,v,(v \Rightarrow v \leftarrow v),\ldots,(v \Rightarrow E \leftarrow e),(e \Rightarrow e \leftarrow e),\ldots) \\
& \mid & \mathsf{ctx/c}(v,(v \Rightarrow v \leftarrow v),\ldots,v,(v \Rightarrow v \leftarrow v),\ldots,(v \Rightarrow v \leftarrow E),(e \Rightarrow e \leftarrow e),\ldots) \\
& \mid & {}^{\ell}\mathsf{mon}^{k}_{j}(E,e) \mid {}^{\ell}\mathsf{mon}^{k}_{j}(c,E) \mid \mathsf{check}^{k}_{j}(E,e) \\
& \mid & \mathsf{guard}_{j}\ (E,v,v,e) \mid \mathsf{install/p}_{j}\ (v,e,E) \mid \mathsf{install/p}_{j}\ (v,E,v) \\
& \mid & \mathsf{module}\,\ell\,\mathsf{exports}\ x\ \mathsf{with}\,x,\ldots\ \mathsf{where}\ x = v,\ldots,x = E,x = e,\ldots;\ p
\end{array}
$$

A.2 Typing judgments

$\boxed{\Sigma \vdash p : \tau}$

$$\frac{\varnothing;\Sigma \vdash m_1 \triangleright \Gamma_1 \qquad \ldots \qquad \Gamma_{n-1};\Sigma \vdash m_n \triangleright \Gamma_n \qquad \Gamma_n;\Sigma \vdash e : \tau}{\vdash m_1\,;\ \ldots m_n\,; e : \tau}$$

$\boxed{\Gamma;\Sigma \vdash m \triangleright \Gamma'}$

$$\frac{\begin{array}{c}\varnothing;\Sigma \vdash e_1 : \tau_1 \qquad \ldots \qquad \{y_1 : \tau_1,\ldots,y_{n-1} : \tau_{n-1}\};\Sigma \vdash e_n : \tau_n \qquad \Gamma' = \{y_1 : \tau_1,\ldots,y_n : \tau_n\} \restriction_{\{x_1,\ldots,x_{n'}\}} \\ \forall_{1\leq i\leq n'}.\exists_{1\leq h\leq n}.x_i \equiv y_h \wedge \tau_h \neq \tau\ \mathsf{ctc} \wedge \tau_h \neq \mathsf{ctx\ ctc} \qquad \forall_{1\leq i\leq n'}.\exists_{1\leq h\leq n}.x_{c_i} \equiv y_h \wedge \tau_h = \tau\ \mathsf{ctc}\end{array}}{\Gamma;\Sigma \vdash \mathsf{module}\,\ell\,\mathsf{exports}\ x_1\ \mathsf{with}\,x_{c_1},\ldots,x_{n'}\ \mathsf{with}\,x_{c_{n'}}\ \mathsf{where}\ y_1 = e_1,\ldots,y_n = e_n \triangleright \Gamma \uplus \Gamma'}$$

$\boxed{\Gamma;\Sigma \vdash e : \tau}$

$$\frac{}{\Gamma;\Sigma \vdash () : \mathsf{Unit}} \qquad \frac{}{\Gamma;\Sigma \vdash n : \mathsf{Int}} \qquad \frac{}{\Gamma;\Sigma \vdash \mathsf{\#t} : \mathsf{Bool}} \qquad \frac{}{\Gamma;\Sigma \vdash \mathsf{\#f} : \mathsf{Bool}} \qquad \frac{\Gamma(x) = \tau}{\Gamma;\Sigma \vdash x : \tau} \qquad \frac{\Gamma;\Sigma \vdash e_i : \mathsf{Int} \qquad i \in \{1,2\}}{\Gamma;\Sigma \vdash e_1 \oplus e_2 : \mathsf{Int}}$$

$$\frac{\Gamma;\Sigma \vdash e_i : \mathsf{Int} \qquad i \in \{1,2\}}{\Gamma;\Sigma \vdash e_1 \leq e_2 : \mathsf{Bool}} \qquad \frac{\Gamma;\Sigma \vdash e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1];\Sigma \vdash e_2 : \tau}{\Gamma;\Sigma \vdash \mathsf{let}\,x = e_1\ \mathsf{in}\ e_2 : \tau} \qquad \frac{\Gamma;\Sigma \vdash e_c : \mathsf{Bool} \qquad \Gamma;\Sigma \vdash e_i : \tau \qquad i \in \{1,2\}}{\Gamma;\Sigma \vdash \mathsf{if}\ e_c\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau}$$

$$\frac{\Gamma[x \mapsto \tau_1];\Sigma \vdash e : \tau_2}{\Gamma;\Sigma \vdash \lambda\,x : \tau_1.\ e : \tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma[x \mapsto \tau];\Sigma \vdash e : \tau}{\Gamma;\Sigma \vdash \mu\,x : \tau.\ e : \tau} \qquad \frac{\Gamma;\Sigma \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma;\Sigma \vdash e_2 : \tau_1}{\Gamma;\Sigma \vdash e_1\ e_2 : \tau_2} \qquad \frac{\Sigma(r) = \tau}{\Gamma;\Sigma \vdash \mathsf{p}(r) : \tau\ \mathsf{param}}$$

$$\frac{\Gamma;\Sigma \vdash e : \tau\ \mathsf{param}}{\Gamma;\Sigma \vdash ?e : \tau} \qquad \frac{\Gamma;\Sigma \vdash e_1 : \tau\ \mathsf{param} \qquad \Gamma;\Sigma \vdash e_2 : \tau}{\Gamma;\Sigma \vdash e_1 := e_2 : \mathsf{Unit}} \qquad \frac{\Gamma;\Sigma \vdash e : \tau}{\Gamma;\Sigma \vdash \mathsf{make\text{-}parameter}\ e : \tau\ \mathsf{param}}$$

$$\frac{\Gamma;\Sigma \vdash e_1 : \tau_p\ \mathsf{param} \qquad \Gamma;\Sigma \vdash e_2 : \tau_p \qquad \Gamma;\Sigma \vdash e_3 : \tau}{\Gamma;\Sigma \vdash \mathsf{parameterize}\,e_1 = e_2\ \mathsf{in}\ e_3 : \tau} \qquad \frac{\Gamma;\Sigma \vdash e_1 : \tau\ \mathsf{ctc} \qquad \Gamma;\Sigma \vdash e_2 : \tau}{\Gamma;\Sigma \vdash {}^{\ell}\mathsf{mon}^{k}_{j}(e_1,e_2) : \tau}$$

$$\frac{\Gamma;\Sigma \vdash e_1 : \mathsf{ctx\ ctc} \qquad \Gamma;\Sigma \vdash e_2 : (\tau_d \rightarrow \tau_r)}{\Gamma;\Sigma \vdash {}^{\ell}\mathsf{mon}^{k}_{j}(e_1,e_2) : (\tau_d \rightarrow \tau_r)} \qquad \frac{\Gamma;\Sigma \vdash e_1 : \mathsf{Bool} \qquad \Gamma;\Sigma \vdash v_2 : \tau}{\Gamma;\Sigma \vdash \mathsf{check}^{k}_{j}(e_1,v_2) : \tau} \qquad \frac{\Gamma;\Sigma \vdash e : \beta \rightarrow \mathsf{Bool}}{\Gamma;\Sigma \vdash \mathsf{flat/c}(e) : \beta\ \mathsf{ctc}}$$

$$\frac{\Gamma;\Sigma \vdash e : \tau\ \mathsf{ctc}}{\Gamma;\Sigma \vdash \mathsf{param/c}(e) : (\tau\ \mathsf{param})\ \mathsf{ctc}} \qquad \frac{\Gamma;\Sigma \vdash e_d : \tau_d\ \mathsf{ctc} \qquad \Gamma;\Sigma \vdash e_r : \tau_r\ \mathsf{ctc} \qquad \Gamma;\Sigma \vdash e_c : \mathsf{ctx\ ctc}}{\Gamma;\Sigma \vdash e_d : \tau_d \rightarrow (e_c)\ e_r : \tau_d \rightarrow \tau_r\ \mathsf{ctc}}$$

$$\frac{\Gamma;\Sigma \vdash e_d : \tau_d\ \mathsf{ctc} \qquad \Gamma;\Sigma \vdash e_r : \tau_r\ \mathsf{ctc} \qquad \Gamma[x \mapsto \tau_d];\Sigma \vdash e : \mathsf{ctx\ ctc}}{\Gamma;\Sigma \vdash e_d : \tau_d \rightarrow_{\mathsf{a}} (\lambda\,x : \tau_d.\ e)\ e_r : \tau_d \rightarrow \tau_r\ \mathsf{ctc}}$$

$$\frac{\begin{array}{c}\Gamma;\Sigma \vdash e_1 : \mathsf{Unit} \to \mathsf{Bool} \qquad \Gamma;\Sigma \vdash e_2 : \mathsf{Unit} \to \mathsf{Bool}\\ \Gamma;\Sigma \vdash e_{g_{c_i}} : \mathsf{Unit} \to \mathsf{Bool} \qquad \Gamma;\Sigma \vdash e_{p_{c_i}} : \tau_{c_i}\ \mathsf{param} \qquad \Gamma;\Sigma \vdash e_{v_{c_i}} : \mathsf{Unit} \to \tau_{c_i}\\ \Gamma;\Sigma \vdash e_{g_{a_i}} : \mathsf{Unit} \to \mathsf{Bool} \qquad \Gamma;\Sigma \vdash e_{p_{a_i}} : \tau_{a_i}\ \mathsf{param} \qquad \Gamma;\Sigma \vdash e_{v_{a_i}} : \mathsf{Unit} \to \tau_{a_i}\end{array}}{\Gamma;\Sigma \vdash \mathsf{ctx/c}(e_1,(e_{g_{c_1}} \Rightarrow e_{p_{c_1}} \leftarrow e_{v_{c_1}}),\ldots,e_2,(e_{g_{a_1}} \Rightarrow e_{p_{a_1}} \leftarrow e_{v_{a_1}}),\ldots) : \mathsf{ctx}\ \mathsf{ctc}}$$

$$\frac{\Gamma;\Sigma \vdash e_c : \tau\ \mathsf{ctc} \qquad \Gamma;\Sigma \vdash e : \tau\ \mathsf{param}}{\Gamma;\Sigma \vdash {}^{\ell}\mathsf{param/p}_j^k(e_c, e) : \tau\ \mathsf{param}}$$

$$\frac{\Gamma;\Sigma \vdash e_g : \mathsf{Bool} \qquad \Gamma;\Sigma \vdash v_p : \tau\ \mathsf{param} \qquad \Gamma;\Sigma \vdash v_v : \mathsf{Unit} \to \tau \qquad \Gamma;\Sigma \vdash e_f : \tau_d \to \tau_r}{\Gamma;\Sigma \vdash \mathsf{guard}_j\ (e_g,v_p,v_v,e_f) : \tau_d \to \tau_r}$$

$$\frac{\Gamma;\Sigma \vdash v_p : \tau\ \mathsf{param} \qquad \Gamma;\Sigma \vdash e_v : \tau \qquad \Gamma;\Sigma \vdash e_f : \tau_d \to \tau_r}{\Gamma;\Sigma \vdash \mathsf{install/p}_j\ (v_p,e_v,e_f) : \tau_d \to \tau_r}$$

A.3 Typing judgments for authorization contract extensions

$\boxed{\Gamma;\Sigma \vdash m \triangleright \Gamma'}$

$$\frac{\begin{array}{c} a_1,\ldots,a_o = \mathsf{action}\, x_{a_1}\ (y_{a_1},\ldots)\ (ce,ae),\ldots,\mathsf{action}\, x_{a_o}\ (y_{a_o},\ldots)\ (ce,ae)\\ \varnothing;\Sigma \vdash e_1 : \tau_1 \qquad \ldots \qquad \{y_1 : \tau_1,\ldots,y_{n-1} : \tau_{n-1}\};\Sigma \vdash e_n : \tau_n\\ \{y_1 : \tau_1,\ldots,y_n : \tau_n\};\Sigma \vdash a_1 : \tau_{a_1} \qquad \ldots \qquad \{y_1 : \tau_1,\ldots,y_n : \tau_n,\ldots,x_{a_1} : \tau_{a_1},\ldots,x_{a_{o-1}} : \tau_{a_{o-1}}\};\Sigma \vdash a_o : \tau_{a_o}\\ \{y_1 : \tau_1,\ldots,y_n : \tau_n,\ldots,x_{a_1} : \tau_{a_1},\ldots,x_{a_o} : \tau_{a_o}\};\Sigma \vdash e_{n+1} : \tau_{n+1}\\ \ldots \qquad \{y_1 : \tau_1,\ldots,y_n : \tau_n,\ldots,x_{a_1} : \tau_{a_1},\ldots,x_{a_o} : \tau_{a_o},y_{n+1} : \tau_{n+1},\ldots,y_{m-1} : \tau_{n-1}\};\Sigma \vdash e_m : \tau_m\\ \Gamma' = \{y_1 : \tau_1,\ldots,y_m : \tau_n\} \restriction_{\{x_1,\ldots,x_{n'}\}}\\ \forall_{1\le i\le n'}.(\exists_{1\le h\le m}.x_i \equiv y_h \wedge \tau_h \ne \tau\ \mathsf{ctc} \wedge \tau_h \ne \mathsf{ctx}\ \mathsf{ctc}) \vee (\exists_{1\le h\le o}.x_i \equiv x_{a_h} \wedge \tau_{a_h} \ne \tau\ \mathsf{ctc} \wedge \tau_{a_h} \ne \mathsf{ctx}\ \mathsf{ctc})\\ \forall_{1\le i\le n'}.\exists_{1\le h\le m}.x_{c_i} \equiv y_h \wedge \tau_h = \tau\ \mathsf{ctc}\end{array}}{\Gamma;\Sigma \vdash \mathsf{module}\,\ell\,\mathsf{exports}\ x_1\ \mathsf{with}\, x_{c_1},\ldots,x_{n'}\ \mathsf{with}\, x_{c_{n'}}\ \mathsf{where}\ y_1 = e_1,\ldots,y_n = e_n, \mathsf{monitor}\ (a_1,\ldots,a_o), y_k = e_k, y_m = e_m \triangleright \Gamma \uplus \Gamma'}$$

$\boxed{\Gamma;\Sigma \vdash e : \tau}$

$$\frac{}{\Gamma;\Sigma \vdash \top : \mathsf{Prin}} \qquad \frac{}{\Gamma;\Sigma \vdash \bot : \mathsf{Prin}} \qquad \frac{}{\Gamma;\Sigma \vdash \mathcal{P} : \mathsf{Prin}} \qquad \frac{}{\Gamma;\Sigma \vdash \mathcal{D} : \mathsf{Dim}} \qquad \frac{}{\Gamma;\Sigma \vdash \mathsf{new\text{-}principal} : \mathsf{Prin}}$$

$$\frac{}{\Gamma;\Sigma \vdash \mathsf{new\text{-}dimension} : \mathsf{Dim}} \qquad \frac{\Gamma;\Sigma \vdash e_p : \mathsf{Prin} \qquad \Gamma;\Sigma \vdash e_d : \mathsf{Dim}}{\Gamma;\Sigma \vdash e_p \triangleright e_d : \mathsf{Prin}} \qquad \frac{}{\Gamma;\Sigma \vdash \{\} : \mathsf{DelSet}} \qquad \frac{\Gamma;\Sigma \vdash e : \mathsf{Del}}{\Gamma;\Sigma \vdash \{e\} : \mathsf{DelSet}}$$

$$\frac{\Gamma;\Sigma \vdash e_1 : \mathsf{DelSet} \qquad \Gamma;\Sigma \vdash e_2 : \mathsf{DelSet}}{\Gamma;\Sigma \vdash e_1 \cup e_2 : \mathsf{DelSet}} \qquad \frac{\Gamma;\Sigma \vdash e_1 : \mathsf{DelSet} \qquad \Gamma;\Sigma \vdash e_2 : \mathsf{DelSet}}{\Gamma;\Sigma \vdash e_1 \setminus e_2 : \mathsf{DelSet}}$$

$$\frac{\Gamma;\Sigma \vdash e_w : \mathsf{DelSet} \qquad \Gamma;\Sigma \vdash e_f : (\tau \to (\mathsf{Del} \to \tau)) \qquad \Gamma;\Sigma \vdash e_i : \tau}{\Gamma;\Sigma \vdash (\mathsf{fold}\ e_w\ e_f\ e_i) : \tau}$$

$$\frac{\Gamma;\Sigma \vdash e_w : \mathsf{DelSet} \qquad \Gamma;\Sigma \vdash e_s : \mathsf{Prin} \qquad \Gamma;\Sigma \vdash e_l : \mathsf{Prin} \qquad \Gamma;\Sigma \vdash e_r : \mathsf{Prin}}{\Gamma;\Sigma \vdash e_w\,;e_s \vdash e_l \succeq e_r : \tau}$$

$$\frac{\Gamma;\Sigma \vdash e_s : \mathsf{Prin} \qquad \Gamma;\Sigma \vdash e_l : \mathsf{Prin} \qquad \Gamma;\Sigma \vdash e_r : \mathsf{Prin}}{\Gamma;\Sigma \vdash e_s \succeq e_l\ @\ e_r : \mathsf{Del}} \qquad \frac{\Gamma;\Sigma \vdash e_a : \mathsf{Del} \qquad \Gamma[x_s \mapsto \mathsf{Prin}, x_l \mapsto \mathsf{Prin}, x_r \mapsto \mathsf{Prin}];\Sigma \vdash e : \tau}{\Gamma;\Sigma \vdash \mathsf{let}\, x_s \succeq x_l\ @\ x_r = e_a\ \mathsf{in}\ e : \tau}$$

$$\frac{\Gamma[y_1 \mapsto \tau_1,\ldots,y_n \mapsto \tau_n];\Sigma \vdash ce \qquad \Gamma[y_1 \mapsto \tau_1,\ldots,y_n \mapsto \tau_n];\Sigma \vdash ae}{\Gamma;\Sigma \vdash \mathsf{action}\, x\ (y_1 : \tau_1,\ldots,y_n : \tau_n)\ (ce,ae) : (\tau_1 \to (\ldots \to (\tau_n \to \mathsf{ctx}\ \mathsf{ctc})))}$$

$\boxed{\Gamma;\Sigma \vdash ce}$

$$\frac{\Gamma;\Sigma \vdash cee_1 : \mathsf{Del} \qquad \Gamma;\Sigma \vdash cee_2 : \mathsf{DelSet} \qquad \Gamma;\Sigma \vdash cee_3 : \mathsf{DelSet} \qquad \Gamma;\Sigma \vdash cee_4 : \mathsf{Prin} \qquad \Gamma;\Sigma \vdash cee_5 : \mathsf{Prin} \qquad \Gamma;\Sigma \vdash cee_6 : \mathsf{DelSet}}{\Gamma;\Sigma \vdash \mathsf{check:}\ cee_1\ \mathsf{add:}\ cee_2\ \mathsf{remove:}\ cee_3\ \mathsf{set!\text{-}principal:}\ ceee_4\ \mathsf{closure\text{-}principal:}\ cee_5\ \mathsf{closure\text{-}delegations:}\ cee_6}$$

$\boxed{\Gamma;\Sigma \vdash ae}$

$$\frac{\begin{array}{c}\Gamma;\Sigma \vdash aee_1 : \mathsf{Del} \qquad \Gamma;\Sigma \vdash aee_2 : \mathsf{DelSet} \qquad \Gamma;\Sigma \vdash aee_3 : \mathsf{DelSet} \qquad \Gamma;\Sigma \vdash aee_4 : \mathsf{DelSet}\\ \Gamma;\Sigma \vdash aee_5 : \mathsf{Prin} \qquad \Gamma;\Sigma \vdash aee_6 : \mathsf{Prin} \qquad \Gamma;\Sigma \vdash aee_7 : \mathsf{Prin}\end{array}}{\Gamma;\Sigma \vdash \mathsf{check:}\ aee_1\ \mathsf{add:}\ aee_2\ \mathsf{remove:}\ aee_3\ \mathsf{scope:}\ aeee_4\ \mathsf{set\text{-}principal?:}\ aee_5\ \mathsf{principal:}\ aee_6\ \mathsf{set!\text{-}principal:}\ aee_7}$$

$\boxed{\Gamma;\Sigma \vdash cee : \tau}$ $\boxed{\Gamma;\Sigma \vdash aee : \tau}$

$$\frac{\Gamma;\Sigma \vdash aee_v : \tau_x \qquad \Gamma[x \mapsto \tau_x];\Sigma \vdash aee : \tau}{\Gamma;\Sigma \vdash \mathsf{let}\, x = aee_v \,\mathsf{in}\, aee : \tau} \qquad \frac{\Gamma;\Sigma \vdash cee_v : \tau_x \qquad \Gamma[x \mapsto \tau_x];\Sigma \vdash cee : \tau}{\Gamma;\Sigma \vdash \mathsf{let}\, x = cee_v \,\mathsf{in}\, cee : \tau} \qquad \frac{}{\Gamma;\Sigma \vdash \mathsf{current\text{-}principal} : \mathsf{Prin}}$$

$$\frac{}{\Gamma;\Sigma \vdash \mathsf{current\text{-}delegations} : \mathsf{DelSet}} \qquad \frac{}{\Gamma;\Sigma \vdash \mathsf{closure\text{-}principal} : \mathsf{Prin}} \qquad \frac{}{\Gamma;\Sigma \vdash \mathsf{closure\text{-}delegations} : \mathsf{DelSet}}$$

A.4 Compiling authorization contract extensions

$\mathrm{compile}\llbracket \mathsf{module}\,\ell\, \mathsf{exports}\, x_1\, \mathsf{with}\, x_{c_1}, \ldots\, \mathsf{where}\; y_1 = e_2, \ldots; p\rrbracket =$
$\quad \mathrm{compile\text{-}monitor}\llbracket \mathsf{module}\,\ell\, \mathsf{exports}\, x_1\, \mathsf{with}\, x_{c_1}, \ldots\, \mathsf{where}\; y_1 = e_2, \ldots; \mathrm{compile}\llbracket p\rrbracket\rrbracket$
$\mathrm{compile}\llbracket e\rrbracket = e$

$\mathrm{compile\text{-}monitor}\llbracket \mathsf{module}\,\ell$
$\qquad \mathsf{exports}\, x_1\, \mathsf{with}\, x_{c_1}, \quad \ldots$
$\qquad \mathsf{where}\; y_1 = e_1, \ldots, y_n = e_n,$
$\qquad\quad \mathsf{monitor}\; (\mathsf{action}\, x_{a_1}\, (y_{a_1} : \tau_{y_{a_1}}, \ldots)\, (ce_1, ae_1), \ldots, \mathsf{action}\, x_{a_n}\, (y_{a_n} : \tau_{y_{a_n}}, \ldots)\, (ce_n, ae_n)),$
$\qquad\quad y_{n+1} = e_{n+1}, \ldots, y_m = e_m;\rrbracket =$

$\mathsf{module}\,\ell$
$\mathsf{exports}\, x_1\, \mathsf{with}\, x_{c_1}, \quad \ldots$
$\mathsf{where}\; y_1 = e_1, \ldots, y_n = e_n,$
$\quad p = \mathsf{make\text{-}parameter}\;\top, d = \mathsf{make\text{-}parameter}\;\{\}, s = \mathsf{make\text{-}parameter}\;\{\},$
$\quad cp = \mathsf{make\text{-}parameter}\;\top, cd = \mathsf{make\text{-}parameter}\;\{\},$
$\quad curp = \mathsf{make\text{-}parameter}\;\top, curd = \mathsf{make\text{-}parameter}\;\{\},$
$\quad x_{a_1} = \mathrm{compile\text{-}action}\llbracket \mathsf{action}\, x_{a_1}\, (y_{a_1} : \tau_{y_{a_1}}, \ldots)\, (ce_1, ae_1), p, d, s, cp, cd, curp, curd\rrbracket,$
$\quad \ldots,$
$\quad x_{a_n} = \mathrm{compile\text{-}action}\llbracket \mathsf{action}\, x_{a_n}\, (y_{a_n} : \tau_{y_{a_n}}, \ldots)\, (ce_n, ae_n), p, d, s, cp, cd, curp, curd\rrbracket,$
$\quad y_{n+1} = e_{n+1}, \ldots, y_m = e_m;$

where p, d, s, cp, cd, $curp$, and $curd$ are fresh

$\mathrm{compile\text{-}action}\llbracket \mathsf{action}\, x\, (y : \tau_y, \ldots)\, (ce, ae), p, d, s, cp, cd, curp, curd\rrbracket =$
$\quad \lambda\, y : \tau_y. \ldots .$
$\qquad \mathsf{ctx/c}(\mathrm{compile\text{-}ce}_{\mathsf{check}}\llbracket ce, p, d, s, curp, curd\rrbracket,$
$\qquad\quad \mathrm{compile\text{-}ce}_{\mathsf{cp}}\llbracket ce, cp, curp, curd\rrbracket,$
$\qquad\quad \mathrm{compile\text{-}ce}_{\mathsf{cd}}\llbracket ce, cd, curp, curd\rrbracket,$
$\qquad\quad \mathrm{compile\text{-}ae}_{\mathsf{check}}\llbracket ae, p, d, s, cp, cd, curp, curd\rrbracket,$
$\qquad\quad \mathrm{compile\text{-}ae}_{\mathsf{s}}\llbracket ae, s, cp, cd, curp, curd\rrbracket,$
$\qquad\quad \mathrm{compile\text{-}ae}_{\mathsf{p}}\llbracket ae, p, cp, cd, curp, curd\rrbracket)$

$\mathrm{compile\text{-}ce}_{\mathsf{check}}\llbracket \mathsf{check{:}}\; e_1\; \mathsf{add{:}}\; e_2\; \mathsf{remove{:}}\; e_3\; \mathsf{set!\text{-}principal{:}}\; e_4\; \mathsf{closure\text{-}principal{:}}\; e_5\; \mathsf{closure\text{-}delegations{:}}\; e_6, p, d, s, curp, curd\rrbracket =$
$\quad \lambda\,\_ : \mathsf{Unit}.$
$\qquad \mathsf{let}\,\_ = curp := ?p\; \mathsf{in}$
$\qquad \mathsf{let}\,\_ = curd := ?d \cup ?s\; \mathsf{in}$
$\qquad \mathsf{let}\, p_1 \succeq p_2 \,@\, p_3 = \mathrm{compile\text{-}cee}\llbracket e_1, curp, curd\rrbracket\; \mathsf{in}$
$\qquad \mathsf{if}\; (?curd)\,;\, p_3 \vdash p_1 \succeq p_2\; \mathsf{then}$
$\qquad\quad \mathsf{let}\, add = \mathrm{compile\text{-}cee}\llbracket e_2, curp, curd\rrbracket\; \mathsf{in}$
$\qquad\quad \mathsf{let}\, remove = \mathrm{compile\text{-}cee}\llbracket e_3, curp, curd\rrbracket\; \mathsf{in}$
$\qquad\quad \mathsf{let}\, setprin = \mathrm{compile\text{-}cee}\llbracket e_4, curp, curd\rrbracket\; \mathsf{in}$
$\qquad\quad \mathsf{let}\,\_ = d := (?d \cup add)/remove\; \mathsf{in}$
$\qquad\quad \mathsf{let}\,\_ = p := setprin\; \mathsf{in}$
$\qquad\quad \mathsf{\#t}$
$\qquad \mathsf{else}\; \mathsf{\#f}$

$\mathrm{compile\text{-}ce}_{\mathsf{cp}}\llbracket \mathsf{check{:}}\; e_1\; \mathsf{add{:}}\; e_2\; \mathsf{remove{:}}\; e_3\; \mathsf{set!\text{-}principal{:}}\; e_4\; \mathsf{closure\text{-}principal{:}}\; e_5\; \mathsf{closure\text{-}delegations{:}}\; e_6, cp, curp, curd\rrbracket =$
$\quad ((\lambda\,\_ : \mathsf{Unit}.\; \mathsf{\#t}) \Rightarrow cp \leftarrow (\lambda\,\_ : \mathsf{Unit}.\; \mathrm{compile\text{-}cee}\llbracket e_5, curp, curd\rrbracket))$

$\mathrm{compile\text{-}ce}_{\mathsf{cd}}\llbracket \mathsf{check{:}}\; e_1\; \mathsf{add{:}}\; e_2\; \mathsf{remove{:}}\; e_3\; \mathsf{set!\text{-}principal{:}}\; e_4\; \mathsf{closure\text{-}principal{:}}\; e_5\; \mathsf{closure\text{-}delegations{:}}\; e_6, cd, curp, curd\rrbracket =$
$\quad ((\lambda\,\_ : \mathsf{Unit}.\; \mathsf{\#t}) \Rightarrow cd \leftarrow (\lambda\,\_ : \mathsf{Unit}.\; \mathrm{compile\text{-}cee}\llbracket e_6, curp, curd\rrbracket))$

```
$\text{compile-ae}_{\textsf{check}}[\![\textsf{check: } e_1 \textsf{ add: } e_2 \textsf{ remove: } e_3 \textsf{ scope: } e_4 \textsf{ set-principal?: } e_5 \textsf{ principal: } e_6 \textsf{ set!-principal: } e_7, p, d, s, cp, cd, curp, curd]\!]$ =
      $\lambda\_ : \textsf{Unit}.$
        let $\_ = curp := ?p$ in
        let $\_ = curd := ?d \cup ?s$ in
        let $p_1 \succeq p_2 \mathbin{@} p_3 = \text{compile-aee}[\![e_1, curp, curd, cp, cd]\!]$ in
        if $(?curd) \,;\, p_3 \vdash p_1 \succeq p_2$ then
          let $add = \text{compile-aee}[\![e_2, curp, curd, cp, cd]\!]$ in
          let $remove = \text{compile-aee}[\![e_3, curp, curd, cp, cd]\!]$ in
          let $setprin = \text{compile-aee}[\![e_7, curp, curd, cp, cd]\!]$ in
          let $\_ = d := (?d \cup add)/remove$ in
          let $\_ = p := setprin$ in
          #t
        else #f
$\text{compile-ae}_{\textsf{p}}[\![\textsf{check: } e_1 \textsf{ add: } e_2 \textsf{ remove: } e_3 \textsf{ scope: } e_4 \textsf{ set-principal?: } e_5 \textsf{ principal: } e_6 \textsf{ set!-principal: } e_7, p, cp, cd, curp, curd]\!]$ =
      $(\text{compile-aee}[\![e_5, curp, curd, cp, cd]\!] \Rightarrow p \leftarrow \text{compile-aee}[\![e_6, curp, curd, cp, cd]\!])$
$\text{compile-ae}_{\textsf{s}}[\![\textsf{check: } e_1 \textsf{ add: } e_2 \textsf{ remove: } e_3 \textsf{ scope: } e_4 \textsf{ set-principal?: } e_5 \textsf{ principal: } e_6 \textsf{ set!-principal: } e_7, s, cp, cd, curp, curd]\!]$ =
      $((\lambda\_ : \textsf{Unit}.\ \#\textsf{t}) \Rightarrow s \leftarrow \text{compile-aee}[\![e_4, curp, curd, cp, cd]\!])$
```

$$\begin{aligned}
\text{compile-cee}[\![\textsf{let } x = cee_1 \textsf{ in } cee_2, curp, curd]\!] &= \textsf{let } x = \text{compile-cee}[\![cee_1, curp, curd]\!] \\
&\quad\ \textsf{in } \text{compile-cee}[\![cee_2, curp, curd]\!] \\
\text{compile-aee}[\![\textsf{let } x = aee_1 \textsf{ in } aee_2, curp, curd, cp, cd]\!] &= \textsf{let } x = \text{compile-aee}[\![aee_1, curp, curd, cp, clod]\!] \\
&\quad\ \textsf{in } \text{compile-aee}[\![aee_2, curp, curd, cp, cd]\!] \\
\text{compile-cee}[\![\textsf{current-principal}, curp, curd]\!] &= ?curp \\
\text{compile-aee}[\![\textsf{current-principal}, curp, curd, cp, cd]\!] &= ?curp \\
\text{compile-cee}[\![\textsf{current-delegations}, curp, curd]\!] &= ?curd \\
\text{compile-aee}[\![\textsf{current-delegations}, curp, curd, cp, cd]\!] &= ?curd \\
\text{compile-aee}[\![\textsf{closure-principal}, curp, curd, cp, cd]\!] &= ?cp \\
\text{compile-aee}[\![\textsf{closure-delegations}, curp, curd, cp, cd]\!] &= ?cd \\
\text{compile-cee}[\![e, curp, curd]\!] &= e \\
\text{compile-aee}[\![e, curp, curd, cp, cd]\!] &= e
\end{aligned}$$

A.5 Authorization contract extension evaluation contexts

$$\begin{aligned}
E \ ::=\ & \ldots \mid E \triangleright e \mid v \triangleright c \mid E\,;e \vdash e \succeq e \mid v\,;E \vdash e \succeq e \mid v\,;v \vdash E \succeq e \mid v\,;v \vdash v \succeq E \\
\mid\ & E \succeq e \mathbin{@} e \mid v \succeq E \mathbin{@} e \mid v \succeq v \mathbin{@} E \mid \{E\} \mid E \cup e \mid v \cup E \mid E \setminus e \mid v \setminus E \\
\mid\ & (\textsf{fold } E\ e\ e) \mid (\textsf{fold } v\ E\ e) \mid (\textsf{fold } v\ v\ E)
\end{aligned}$$

A.6 Reduction semantics for authorization contract extensions

$$\begin{aligned}
\langle E[\textsf{new-principal}], \sigma\rangle &\to \langle E[p], \sigma\rangle \text{ where } p \text{ is fresh} \\
\langle E[\textsf{new-dimension}], \sigma\rangle &\to \langle E[d], \sigma\rangle \text{ where } d \text{ is fresh} \\
\langle E[\{p_{1_s} \succeq p_{1_l} \mathbin{@} p_{1_r}, \ldots\}\,; p_s \vdash p_l \succeq p_r], \sigma\rangle &\to \langle E[\#\textsf{t}], \sigma\rangle \text{ if } \{p_{1_l} \succeq p_{1_r} \mathbin{@} p_{1_s}, \ldots\}\,; p_s \vdash p_l \succeq p_r \\
\langle E[\{p_{1_s} \succeq p_{1_l} \mathbin{@} p_{1_r}, \ldots\}\,; p_s \vdash p_l \succeq p_r], \sigma\rangle &\to \langle E[\#\textsf{f}], \sigma\rangle \text{ if } \{p_{1_l} \succeq p_{1_r} \mathbin{@} p_{1_s}, \ldots\}; p_s \nvdash p_l \succeq p_r \\
\langle E[\{v_{1_1}, \ldots, v_{1_n}\} \cup \{v_{2_1}, \ldots, v_{2_m}\}], \sigma\rangle &\to \langle E[\{v_{3_1}, \ldots, v_{3_k}\}], \sigma\rangle \\
&\quad \text{where } \{v_{3_1}, \ldots, v_{3_k}\} = \{v_{1_1}, \ldots, v_{1_n}\} \cup \{v_{2_1}, \ldots, v_{2_m}\} \\
\langle E[\{v_{1_1}, \ldots, v_{1_n}\} \setminus \{v_{2_1}, \ldots, v_{2_m}\}], \sigma\rangle &\to \langle E[\{v_{3_1}, \ldots, v_{3_k}\}], \sigma\rangle \\
&\quad \text{where } \{v_{3_1}, \ldots, v_{3_k}\} = \{v_{1_1}, \ldots, v_{1_n}\} \setminus \{v_{2_1}, \ldots, v_{2_m}\} \\
\langle E[(\textsf{fold } \{\}\ v_f\ v)], \sigma\rangle &\to \langle E[v], \sigma\rangle \\
\langle E[(\textsf{fold } \{v_1, v_2, \ldots\}\ v_f\ v)], \sigma\rangle &\to \langle E[(\textsf{fold } \{v_2, \ldots\}\ v_f\ ((v_f\ v)\ v_1))], \sigma\rangle \\
\langle E[\textsf{let } x_s \succeq x_l \mathbin{@} x_r = v_s \succeq v_l \mathbin{@} v_r \textsf{ in } e], \sigma\rangle &\to \langle E[\{^{v_l, v_r, v_s}/_{x_l, x_r, x_s}\}e], \sigma\rangle
\end{aligned}$$