HARVARD UNIVERSITY
Graduate School of Arts and Sciences

DISSERTATION ACCEPTANCE CERTIFICATE

The undersigned, appointed by the

Harvard John A. Paulson School of Engineering and Applied Sciences

have examined a dissertation entitled:

"Software Contracts for Security"

presented by: Scott David Moore

candidate for the degree of Doctor of Philosophy and here by certify that it is worthy of acceptance.

Signature [illegible]

Typed name: Professor S. Chong

Signature [illegible]

Typed name: Professor E. Kohler

Signature [illegible]

Typed name: Professor J. Mickens

Date: April 25 2016

Software Contracts for Security

A DISSERTATION PRESENTED
BY
SCOTT DAVID MOORE
TO
THE HARVARD JOHN A. PAULSON SCHOOL OF ENGINEERING AND APPLIED SCIENCES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SUBJECT OF
COMPUTER SCIENCE

HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
MAY 2016

Dissertation advisor: Professor Stephen Chong Scott David Moore

Software Contracts for Security

Abstract

Component-based software engineering facilitates the design of complex systems by subdividing the programming task into separate components that interact via clearly defined interfaces. A component-based system is correct only when each component satisfies its specification and the interactions between different components satisfy their respective interfaces. "Design by Contract" is a programming methodology that enforces these requirements by attaching executable specifications to components. These contracts monitor the system's execution and interfere when a specification would be violated.

Designing a program this way gives assurance that the program is correct. It eliminates defensive programming by separating code that validates the correct use of a component from the component's implementation. It also simplifies debugging by localizing program errors within the component that violated its specification. However, existing work on software contracts focuses almost exclusively on contracts for functional correctness. This dissertation argues that

> *Higher-order software contracts are an effective mechanism to specify and enforce composable, easy-to-understand security properties.*

Whereas traditional software contracts describe what a component requires from its clients and provides in return, software contracts for security enforce limits on the contexts in which components can be used.

The first part of this dissertation shows how existing software contracts can be combined with language restrictions to write and enforce declarative security specifications. Based on this design, I develop a secure shell scripting language. The second part introduces authorization contracts, a new type of software contract that can implement a wide range of commonly used security mechanisms without requiring language modifications.

Contents

1 Introduction **1**

2 Behavioral Contacts **6**

2.1 Contracts by example . . . 7

2.2 A Model for Contracts . . . 13

2.3 Correct Blame and Complete Monitoring . . . 17

2.4 Related Work . . . 25

3 Capability Contracts **28**

3.1 Capability-safety and Complete Monitoring . . . 30

3.2 From Patterns to Contracts . . . 35

3.3 Related Work . . . 52

4 A Secure Shell Scripting Language **54**

4.1 Design . . . 58

4.2 Implementation . . . 73

4.3 Evaluation . . . 84

4.4 Related Work . . . 92

5 Authorization Contracts **95**

5.1 Authority Environments . . . 98

5.2 A Framework for Access Control . . . 105

5.3 Putting Authorization Contracts to Work . . . 128

5.4 Case studies . . . 139

5.5 Related Work . . . 146

6 Conclusion **149**

References **151**

To Jenn, who makes my heart leap and all things possible.

Acknowledgments

First, I must thank Stephen Chong, my advisor. He gave me support, helped me grow, and made this work possible. His encouragement and guidance has made me a better researcher and a better writer. His constant kindess is an inspiration.

I would like to thank my committee members Eddie Kohler and James Mickens. Their valuable comments pushed me to refine my thinking and better communicate my ideas.

The work in this dissertation could not have been completed without the help of my co-authors Aslan Askarov, Christos Dimoulas, Robby Findler, Matthew Flatt, and Dan King. Their hard work, insight, and companionship made the work a delight. I am especially grateful to Aslan and Christos, who, as well as collaborators, were great mentors.

I would also like to thank the many other people I have had the pleasure to work with over the years: Vikram Adve, Joshua Cranmer, David Darais, Brian DeVries, Will Dietz, Ashish Gehani, Andrew Johnson, Heiko Mantel, Gregory Malecha, Greg Morrisett, Lucas Waye, Natarajan Shankar, and Meera Sridhar. In particular, I would like to thank Kevin Hamlen, Gopal Gupta, and the rest of the faculty at the University of Texas at Dallas, who stirred my passion for research and supported me as an enthusiastic young student. Finally, I thank the inhabitants of the programming languages lab in MD309. They have made the past six years a wonderful combination of work, learning, and fun.

My family instilled the love of learning that set me on this path. Mom, Dad, Mark, Lilly, Andrew, Peter, and Gizem: thank you for always being there for me, even though you didn't always know what I was doing or why it was taking so long. Your support made this possible, and coming home was always a welcome respite. Jenn: you've been my steadfast companion, my partner in crime, and my inspiration; thank you for the years past and all the years to come.

1 Introduction

To simplify the monumental task of writing large computer systems, programmers decompose their code into components (variously called modules, functions, objects or units) that each handle a small part of the whole system. Modular decomposition allows the programmer to treat separate concerns of the whole system independently. Furthermore, it facilitates program reuse because different programs that share common sub-tasks may share components.

However, these benefits do not come free. For a system composed of many different parts to be correct, the interactions between different components must satisfy the assumptions that each component author made about the rest of the system. This has two implications: first, it suggests that components must come with formal or informal specifications for how they should be used, and second, it requires defensive programming to detect when an assumption is violated.

Software contracts, first popularized by the Eiffel programming langauge [74], are a programming language feature for writing and enforcing component specifications. Contracts are written in the same language as the components they describe and are checked during program execution to detect violations. If a violation occurs, the contract system signals an error which blames the component that did not satisfy its specification. This blame information greatly simplifies the task of debugging a large program comprising many different components.

Executable software contracts are available for a wide variety of programming languages including Ada [7, 68], C [96], C++ [88], Java [55, 59, 65], C# [8], Racket [37] and JavaScript [57]. These contract systems allow programmers to attach *behavioral contracts* to program components such as classes, methods, or functions [12]. A behavioral contract specifies functional properties of a component's behavior: properties relating the inputs and initial state to the outputs and final state of an operation on the component [47].

For example, attaching a behavioral contract to a function that writes nicely formatted log messages can ensure that its callers will invoke it with the appropriate arguments: a path designating a log file and a string with the message to be logged. However, such a contract says nothing about security concerns such as who may write log messages to which file or in what context.

In a simple program written for a single user, these questions may have uninteresting answers—there is a single log file chosen by the user and it is always safe to write to it. But as programs become more complex, so do these security concerns. Consider the famous article "The Confused Deputy" [46], in which Hardy recounts how confusion over who is doing what and why leads to security vulnerabilities in a time-sharing system. One of the programs running on the system was a compiler, which was deputized to perform two tasks: to compile programs for users and to record how much those users should be billed for the service. The billing information was stored in a special file called `(SYSX)BILL`. A particularly clever user found out about this file and instructed the compiler to use it for logging debugging information. The compiler happily compiled the user's program for free by over-writing all of the billing information.

To protect against this kind of misuse, a software contract must associate information with each component within the program that describes how it must be used and, before performing a sensitive action, must check that the context in which the action takes place is appropriate. But this is less simple than it seems, even for this simple example of controlling access to a file.

In existing efforts to use contracts to enforce security properties, programmers must explicitly record the information required to enforce a security property by adding code to contracts throughout the program that imperatively updates and checks additional "ghost" data associated with the security property [65]. Doing this correctly is challenging, so a number of tools have been developed to automatically generate annotations for particular classes of security policies [86, 105, 121]. However, because the contracts generated by these tools expose the details of a particular encoding strategy, they obscure the high-level security policy they are meant to enforce. This makes them difficult to understand and thus diminishes the benefits of software contracts as a form of documentation and an aid to debugging.

This dissertation presents an alternative approach to enforcing security properties using software contracts, based on *higher-order software contracts* [37]. In a language with higher-order features such as objects or closures, behavioral contracts must be able to specify functional properties of operations on these higher-order values. To address this issue, *nominal* contract systems for object-oriented languages such as Eiffel [74], Java [65], and C# [8] associate each object with a single contract that is determined by the object's class. Higher-order software contracts, introduced by Findler and Felleisen [37], go further by allowing behavioral contracts to attach additional contracts to higher-order values as they flow between components. The additional expressive power of higher-order software contracts means that like other properties of software components, security properties can be cleanly expressed as part of a component's interface.

Moreover:

> *Higher-order software contracts are an effective mechanism to specify and enforce composable, easy-to-understand security properties.*

To validate this claim, I present two novel approaches for writing software contracts that address security concerns. The first approach extends languages designed to support capabilty-based security with support for higher-order software contracts. In a capability-safe programming language, a component is permitted to access a sensitive resource (usually a program value) only if it posseses a corresponding "capability." By carefully controlling which components have access to capabilities, a wide range of access control abstractions can be implemented [79]. Because these languages achieve security by restricting how and when values can be accessed by different components, higher-order contracts are a natural way to formalize these policies. Moreover, contracts can replace ad-hoc design patterns that implement security abstractions with declarative, easy-to-understand security policies. To better express some design patterns, I introduce *bounded-polymorphic contracts*, which extend existing behavioral contract systems with contracts that behave similarly to bounded-polymorphic types.

While effective for enforcing security in a capability-safe programming language, existing behavioral contracts are ill-suited to enforcing security in less-restrictive languages where security depends not just on what values a component uses, but also in which context the component executes. Security contracts for these languages must express properties about the context in which particular operations are invoked. The second approach to software contracts for security presented by this dissertation introduces *context contracts* and *authorization contracts*, which allow programmers to develop custom-tailored security mechanisms for components.

A key insight in the development of authorization contracts is that components receive authority in different ways that have direct analogs in the scoping of variable environments: by inheriting authority from their surrounding execution context or by capturing authority where the component was created. Context contracts can cooperate with other contracts executing in their dynamic extent, and also capture information about their current context. Authorization contracts combine context contracts with a novel authorization logic to define a domain-specific language for writing contracts for security. This language is expressive enough to encode a wide range of existing language-level security mechanisms including stack inspection [124] and capability-based security [80]. Furthermore, it allows programmers to develop and use security mechanisms that compose cleanly and are tailored to the needs of specific components, rather than enforced centrally by the language implementation.

This dissertation is organized into six chapters. In Chapter 2, I review existing work on higher-order behavioral contracts, paying particular attention to contracts that prove useful for implementing security policies and to the foundations of reasoning about and validating the correctness of contract systems. In Chapter 3, I give a brief overview of capability-based security and show how software contracts can replace many existing design patterns from capability-safe programming languages. Chapter 4 describes Shill, a secure shell scripting language that combines a restrictive, capability-safe language, software contracts, and a system-level sandbox. Shill demonstates that the combination of contracts and capabilities results in easy-to-understand, effective security guarantees. Chapter 5 introduces context contracts and shows how they can be used to develop authorization contracts that enforce security properties on components. Chapter 6 concludes with some final remarks.

Chapter 3 includes material that was previously published in the paper "Declarative Policies for Capability Control" [24], which I authored with Christos Dimoulas, Aslan Askarov, and Stephen Chong. Chapter 4 was previously published as the paper "Shill: A Secure Shell Scripting Language" [81], which I authored with Christos Dimoulas, Dan King, and Stephen Chong. Chapter 5 is the result of an ongoing collaboration with Christos Dimoulas, Robert Bruce Findler, Matthew Flatt, and Stephen Chong.

2
Behavioral Contacts

Bertrand Meyer's "Design by Contract" paradigm directs software developers to couple every component with an enforceable contract that describes its interface [75]. These behavioral contracts describe the invariants and pre- and post-conditions a program must satisfy when using the component. Higher-order software contracts [37] extend previous implementations of behavioral contracts with flexible support for specifying higher-order properties of components.

This chapter provides a brief overview of higher-order behavioral contracts, first presenting contracts through a number of short examples (Section 2.1) and then describing their implementation and semantics using a formal model (Section 2.2 and Section 2.3).

```
(define (string-insert str index txt)
  (string-append (substring str 0 index)
                 txt
                 (substring str index)))
```

Figure 1: A simple component.

2.1 Contracts by example

As an example of a very simple component, consider the Racket [38] procedure in Figure 1 that inserts text into a string at a specified position. The informal specification of this procedure is that it takes three arguments: a string, an index indicating a position in the string, and another string to insert. It returns a new string where the first `index` characters are a prefix of the first argument, the next sequence of characters are the characters from the third argument, and the rest of the string is the remainder of the first argument.

This informal specification is precise enough for another programmer to start using this procedure in their own code:

```
> (string-insert "hello world" 6 "to the ")
"hello to the world"
```

Unfortunately, it is very easy for a programmer to misunderstand this specification or make a simple mistake, perhaps passing arguments in the wrong order:

```
> (string-insert "hello world" "to the " 6)
substring: contract violation
  expected: exact-nonnegative-integer?
  given: "to the "
  argument position: 3rd
  other arguments...:
   "hello world"
   0
```

```
(: string-insert (-> String Integer String String))
(define (string-insert str index txt)
  (string-append (substring str 0 index)
                 txt
                 (substring str index)))
```

Figure 2: A type interface for `string-insert`.

Because the implementation of the component assumed it would be called correctly, the result is unpredictable. In this case, reversing the arguments eventually causes the `substring` procedure to be invoked with inappropriate arguments, resulting in an error. To the client programmer, the error message is particularly unhelpful because it refers to implementation details of `string-insert` that were not part of its specification. Something went wrong, but it is not clear what.

To avoid these kinds of errors, many programming languages have features that allow programmers to express specifications more formally. A common tool for this purpose is types. Types are an excellent tool for specifying simple properties about the structure of inputs and outputs that a component expects. For example, using Typed Racket, a dialect of Racket with statically checked types [118], we can attach a type to the `string-insert` procedure, as seen in Figure 2.

The type annotation `(: string-insert (-> String Integer String String))` asserts that `string-insert` is a procedure that accepts three arguments (a string, an integer, and a string) and returns a string. Using this formal specification, the language can check that client programs at least invoke the component with the correct types of arguments, and prevent obviously wrong programs from compiling:

```
> (string-insert "hello world" "to the " 6)
Type Checker: type mismatch
  expected: Integer
  given: String
  in: "to the "
Type Checker: type mismatch
  expected: String
  given: Positive-Byte
  in: 6
```

However, because type systems enforce specifications statically, they must find a careful balance between supporting properties that are simple enough to reason about without running the program and more complex properties that force the programmer to structure their code (or worse, write proofs) to convince the type system of the program's correctness. As a result, programmers use types to express some, but not all, of a component's specifications. Properties that cannot be expressed in the type system are either ignored, or enforced by the implementation of the component.

In this case, the type of `string-insert` does not enforce that the index identifies a location within the string, so it is possible to call `string-insert` with arguments that don't satisfy its specification:

```
> (string-insert "a" 2 "b")
substring: ending index is out of range
  ending index: 2
  starting index: 0
  valid range: [0, 1]
  string: "a"
```

Here, the type of `string-insert` failed to capture the requirement that the index represents a valid position in the first string argument. In Figure 3, we show how the code can be made robust against this error by explicitly checking for it in the body of the procedure. This code improves on the previous version because it has predictable behavior when the component's assumptions aren't met, but it has a few downsides. First, the specification visible to the

```
(: string-insert (-> String Integer String String))
(define (string-insert str index txt)
  (unless (and (>= index 0) (< index (string-length str)))
    (raise-argument-error
      'string-insert
      "Integer in range [0,(string-length str)]" 1
      str index txt))
  (string-append (substring str 0 index)
                 txt
                 (substring str index)))
```

Figure 3: Checking for errors in `string-insert`.

caller doesn't communicate precisely what pre-conditions should be met to call the procedure. Second, the implementation is more complicated than it needs to be because it mixes code for the component's functionality with code for validating inputs.

An alternative for specifying and enforcing component specifications is behavioral contracts. Behavioral contracts allow programmers to attach pre- and post-conditions to component interfaces. These pre- and post-conditions are predicates written in the same language as the software components they govern, and thus can enforce complex, application-specific requirements.

Consider the version of the `string-insert` procedure in Figure 4. This version comes with a contract. The `define/contract` form works like `define`, but also attaches a contract to the newly defined value. In this case, it attaches the contract

```
(->i ([str string?]
      [index (str) (integer-in 0 (string-length str))]
      [txt string?])
     [result string?])
```

to procedure `string-insert`. This contract is a dependent function contract contract [25, 37] (non-dependent function contracts are defined using `->`). It specifies pre-conditions for

```
(define/contract (string-insert str index txt)
  (->i ([str string?]
        [index (str) (integer-in 0 (string-length str))]
        [txt string?])
       [result string?])
  (string-append (substring str 0 index)
                 txt
                 (substring str index)))
```

Figure 4: A contract for `string-insert`.

`string-insert` as predicates on the arguments: the first argument must be a string, the second must be an integer between zero and the length of the string, and the third must also be a string. The contract is *dependent* because the contract itself depends on program values, in this case, the length of the string passed as the first argument. The contract specifies a post-condition for the function as well: the result must be a string.

The predicates that appear in a contract are not restricted to a predefined set, but instead can be arbitrary code, or even additional contracts that have been defined by the programmer, as in the `integer-in` contract used by the contract for `string-insert`. The fact that contracts and predicates are defined within the language has two ramifications. First, it means that users are free to enforce whatever properties are important to their programs. Second, it means that users can define their own predicates and *contract combinators* (like `->`, `->i`, and `integer-in`) to express properties using a domain specific language customized to the task at hand. In this case, we use arrow combinators to give users a familiar, type-like language for expressing properties of functions.

This contracted procedure enforces the same restrictions on its use as the earlier version of the code which contained explicit checks that the arguments were suitable:

```
> (string-insert "a" 2 "b")
string-insert: contract violation
 expected: (integer-in 0 1)
  given: 2
  in: the index argument of
      (->i ([str string?]
           [index (str) (integer-in 0 (string-length str))]
           [txt string?])
           [result string?])
  contract from: (procedure string-insert)
  blaming: program
   (assuming the contract is correct)
  at: program:1.0
```

In addition to cleanly separating the defensive checks required to validate the uses of this code, using a contract for this purpose has a number of advantages. First, the contract automates the task of replacing the interface specfication with the code necessary to perform the checks, and even generates detailed error messages when the contract is violated. Second, the contract system can carefully track *blame* for a violation to help in debugging. In the previous example, the contract blames the incorrect invocation of `string-insert`. If instead the procedure failed to meet its specification, the contract would report that `string-insert` itself was to blame. In a large program with many interacting components, this debugging information is invaluable for quickly identifying the root cause of errors.

Because contracts may impose arbitrary restrictions on components, in general they cannot be checked immediately. For example, consider the following procedure and function contract:

```
(define/contract (twice f x)
  (-> (-> integer? integer?) integer? integer?)
  (f (f x)))
```

This procedure takes as input a function from integers to integers and an integer, and returns the result of applying the function to the integer twice. By Rice's theorem [95], the contract on the first argument `(-> integer? integer?)` cannot be checked by inspecting the procedure

$$
\begin{aligned}
v &::= n \mid \mathsf{\#t} \mid \mathsf{\#f} \mid \mathsf{()} \mid (\lambda\ (x : \tau)\ e) \mid (\mathsf{loc}\ r) \mid (\mathsf{loc/p}\ j\ k\ l\ ctc\ v) \mid ctc \\
ctc &::= (\mathsf{flat/c}\ v) \mid (\mathsf{ref/c}\ ctc) \mid (\rightarrow : \tau\ ctc\ ctc) \mid (\mathsf{contract/c} : \tau) \\
e &::= x \mid v \mid (\mathsf{ref}\ e) \mid (!\ e) \mid (e := e) \mid (e\ e) \mid (\mu\ (x : \tau)\ e) \\
&\quad \mid (\mathsf{let}\ (x\ e)\ e) \mid (\mathsf{if}\ e\ e\ e) \mid (e + e) \mid (e - e) \mid (e \le e) \\
&\quad \mid (\mathsf{flat/c}\ e) \mid (\mathsf{ref/c}\ e) \mid (\rightarrow : \tau\ e\ e) \\
&\quad \mid (\mathsf{mon}\ j\ k\ l\ e\ e) \mid (\mathsf{check}\ j\ k\ e\ e) \mid (\mathsf{error}\ j\ k) \\
B &::= \mathsf{Unit} \mid \mathsf{Int} \mid \mathsf{Bool} \\
\tau &::= B \mid (\tau \rightarrow \tau) \mid (\tau\ \mathsf{contract}) \mid (\tau\ \mathsf{ref})
\end{aligned}
$$

Figure 5: CtcPCF Syntax.

passed as an argument to this function. Instead, we follow Findler and Felleisen's semantics of contracts and check that all *uses* of the contracted function satisfy the contract [37]. To accomplish this, contracts are decomposed into their constituent parts and attached to the relevant values. Contract checks occur only when "flat" values (which, unlike functions, do not have higher-order behavior) are available to be checked by predicates appearing in the contract. In effect, the contract system creates "proxy" values that behave the same as the underlying value, but insert checks at opportune moments.

To give a clear semantics of this behavior, the next section presents a model of higher-order contracts as presented by Findler and Felleisen [37] and later extended by Dimoulas et al. [25].

2.2 A Model for Contracts

To model contracts, we use Plotkin's PCF [90] extended with contracts and references. PCF itself is an extension of the simply-typed λ-calculus with integers, booleans, and recursion. The syntax of our model, which we call CtcPCF, is presented in Figure 5.

The terms of this language are standard, except for forms related to contracts and contract checking. There are four types of contracts in the model: flat contracts ($(\mathsf{flat/c}\ e)$), contracts for references ($(\mathsf{ref/c}\ e)$), contracts for functions ($(\rightarrow : \tau\ e\ e)$), and a contract for contracts themselves (contract/c : τ). Contracts are attached to values in the language by the contract monitor term: (mon $j\ k\ l\ e_c\ e_v$). A contract monitor checks that values that flow from the "server" e_v to its "client" context satisfy the contract e_c. The labels j, k, and l identify the components whose interaction is monitored by the contract: the contract itself (j), the server (k), and the client (l). These labels will be used to verify that contracts are correctly enforced, and to assign blame to the component responsible for a contract violation. The value (loc/p $j\ k\ l\ ctc\ v$) is a proxy for a reference, and is described below.

Unlike the model presented by Dimoulas et al. [25], CtcPCF has first-class contracts. That is, the contract that a monitor attaches to a value is computed by the program, rather than appearing as a constant (hence it is represented by an expression e_c rather than from a separate grammar of contracts). Likewise, the contract combinators described above are composed from arbitrary expressions that may compute appropriate contracts at runtime. We include this extension to the previous model because it is necessary for building authorization contracts, which are presented in Chapter 5.

The term (check $j\ k\ (v_c\ v)\ v$) is used to check that a value v satisfies a predicate v_c imposed by some flat contract (flat/c v_c). If ($v_c\ v$) evaluates to false, the contract system uses the labels associated with the check to raise the error (error $j\ k$), which says that contract j was violated by component k. These terms do not appear in the surface syntax of the language, but are created during the evaluation of contracted values.

A type system for CtcPCF is shown in Figure 6. Again, it is mostly standard, other than the rules for typing contracts and contract monitors. These rules ensure that contracts are applied only to values whose type matches the type expected by the contract. For example, the function contract ($\rightarrow : \tau\ ctc_d\ ctc_r$) can only be applied to functions with the type τ. The contracts for the domain and range (ctc_d and ctc_r) of the function must be contracts for values of type τ_d and τ_r, where $\tau = (\tau_d \rightarrow \tau_r)$. Of course, the compatability between contracts and the values they are attached to could be enforced by the contracts themselves. We use a type system to eliminate the additional complications this would add to the semantics of the model.

To give meaning to the language, we extend the reduction semantics [34, 89] given by Dimoulas et al. [25]. The evaluation contexts used by the semantics are given in Figure 7. They are mostly standard and enforce strict, left-to-right evaluation of expressions. Of the reduction rules given in Figure 8, we focus on those relevant to the semantics of contracts: flat/c, contract/c, →/c, ref/c, loc/p-read, loc/p-write, check-true, and check-false.

Rule flat/c applies the predicate associated with the contract to a monitored value. The label *k* is used when checking the contract because satisfying the predicate is the responsibility of the server component.

Rule contract/c imposes no restrictions on the contract value provided by the server. A more sophisticated contract system could extend this contract to enforce more checks.

Rule →/c attaches a function contract to a function. It creates a proxy function that wraps the original function with additional monitors to check the contracts for the domain and range of the function. The labels of the components for the domain contract are swapped to reflect that the client of the function is responsible for supplying an acceptable argument, which will then be received by the server of the function.

Rule ref/c attaches a contract to a reference. Again, this creates a proxy value around the reference that will enforce the contract on values stored into or read from the reference. Unlike functions, we must use a special type of value (loc/p *j k l ctc v*) to implement this proxy because the language without contracts does not have a transparent interposition mechanism for references. Rules loc/p-read and loc/p-write replace occurances of reference proxies that appear in expressions for reading and writing references with expressions that read or write the reference only after checking that the value read from or written to the reference satisfies the reference's contract.

$$\frac{}{\Gamma; \Sigma \vdash n : \mathsf{Int}} \qquad \frac{}{\Gamma; \Sigma \vdash () : \mathsf{Unit}} \qquad \frac{}{\Gamma; \Sigma \vdash \#\mathsf{t} : \mathsf{Bool}} \qquad \frac{}{\Gamma; \Sigma \vdash \#\mathsf{f} : \mathsf{Bool}}$$

$$\frac{}{\Gamma; \Sigma \vdash (\mathsf{contract/c} : \tau) : ((\tau\ \mathsf{contract})\ \mathsf{contract})} \qquad \frac{\Gamma[x \mapsto \tau_d]; \Sigma \vdash e : \tau_r}{\Gamma; \Sigma \vdash (\lambda\ (x : \tau_d)\ e) : (\tau_d \to \tau_r)}$$

$$\frac{\Gamma; \Sigma \vdash e_1 : \tau_1 \quad \Gamma[x \mapsto \tau_1]; \Sigma \vdash e_2 : \tau_2}{\Gamma; \Sigma \vdash (\mathsf{let}\ (x\ e_1)\ e_2) : \tau_2} \qquad \frac{\Sigma(r) = \tau}{\Gamma; \Sigma \vdash (\mathsf{loc}\ r) : (\tau\ \mathsf{ref})} \qquad \frac{\Gamma; \Sigma \vdash e : \tau}{\Gamma; \Sigma \vdash (\mathsf{ref}\ e) : (\tau\ \mathsf{ref})} \qquad \frac{\Gamma; \Sigma \vdash e : (\tau\ \mathsf{ref})}{\Gamma; \Sigma \vdash (!\ e) : \tau}$$

$$\frac{\Gamma; \Sigma \vdash e_1 : (\tau\ \mathsf{ref}) \quad \Gamma; \Sigma \vdash e_2 : \tau}{\Gamma; \Sigma \vdash (e_1 := e_2) : \tau} \qquad \frac{\Gamma; \Sigma \vdash v : (\tau\ \mathsf{ref}) \quad \Gamma; \Sigma \vdash ctc : (\tau\ \mathsf{contract})}{\Gamma; \Sigma \vdash (\mathsf{loc/p}\ j\ k\ l\ ctc\ v) : (\tau\ \mathsf{ref})} \qquad \frac{\Gamma(x) = \tau}{\Gamma; \Sigma \vdash x : \tau} \qquad \frac{\Gamma; \Sigma \vdash e_1 : (\tau_d \to \tau_r) \quad \Gamma; \Sigma \vdash e_2 : \tau_d}{\Gamma; \Sigma \vdash (e_1\ e_2) : \tau_r}$$

$$\frac{\Gamma[x \mapsto \tau]; \Sigma \vdash e : \tau}{\Gamma; \Sigma \vdash (\mu\ (x : \tau)\ e) : \tau} \qquad \frac{\Gamma; \Sigma \vdash e_1 : \mathsf{Bool} \quad \Gamma; \Sigma \vdash e_2 : \tau \quad \Gamma; \Sigma \vdash e_3 : \tau}{\Gamma; \Sigma \vdash (\mathsf{if}\ e_1\ e_2\ e_3) : \tau} \qquad \frac{\Gamma; \Sigma \vdash e_1 : \mathsf{Int} \quad \Gamma; \Sigma \vdash e_2 : \mathsf{Int}}{\Gamma; \Sigma \vdash (e_1 + e_2) : \mathsf{Int}} \qquad \frac{\Gamma; \Sigma \vdash e_1 : \mathsf{Int} \quad \Gamma; \Sigma \vdash e_2 : \mathsf{Int}}{\Gamma; \Sigma \vdash (e_1 - e_2) : \mathsf{Int}}$$

$$\frac{\Gamma; \Sigma \vdash e_1 : \mathsf{Int} \quad \Gamma; \Sigma \vdash e_2 : \mathsf{Int}}{\Gamma; \Sigma \vdash (e_1 \leq e_2) : \mathsf{Bool}} \qquad \frac{\Gamma; \Sigma \vdash e : (B \to \mathsf{Bool})}{\Gamma; \Sigma \vdash (\mathsf{flat/c}\ e) : (B\ \mathsf{contract})} \qquad \frac{\Gamma; \Sigma \vdash e : (\tau\ \mathsf{contract})}{\Gamma; \Sigma \vdash (\mathsf{ref/c}\ e) : ((\tau\ \mathsf{ref})\ \mathsf{contract})}$$

$$\frac{\Gamma; \Sigma \vdash e_1 : (\tau_d\ \mathsf{contract}) \quad \Gamma; \Sigma \vdash e_2 : (\tau_r\ \mathsf{contract}) \quad \tau = (\tau_d \to \tau_r)}{\Gamma; \Sigma \vdash (\to : \tau\ e_1\ e_2) : (\tau\ \mathsf{contract})} \qquad \frac{\Gamma; \Sigma \vdash e_2 : \tau \quad \Gamma; \Sigma \vdash e_1 : (\tau\ \mathsf{contract})}{\Gamma; \Sigma \vdash (\mathsf{mon}\ j\ k\ l\ e_1\ e_2) : \tau} \qquad \frac{\Gamma; \Sigma \vdash e_1 : \mathsf{Bool} \quad \Gamma; \Sigma \vdash e_2 : \tau}{\Gamma; \Sigma \vdash (\mathsf{check}\ j\ k\ e_1\ e_2) : \tau}$$

Figure 6: Well-typed CtcPCF Terms.

$$\begin{aligned} E ::= {} & [\,] \mid (E\ e) \mid (v\ E) \mid (\mathsf{let}\ (x\ E)\ e) \mid (\mathsf{if}\ E\ e\ e) \\ & \mid (E + e) \mid (v + E) \mid (E - e) \mid (v - E) \mid (E \leq e) \mid (v \leq E) \\ & \mid (\mathsf{ref}\ E) \mid (E := e) \mid ((\mathsf{loc}\ r) := E) \mid (!\ E) \\ & \mid (\mathsf{flat/c}\ E) \mid (\mathsf{ref/c}\ E) \mid (\rightarrow : \tau\ E\ e) \mid (\rightarrow : \tau\ v\ E) \\ & \mid (\mathsf{mon}\ j\ k\ l\ E\ e) \mid (\mathsf{mon}\ j\ k\ l\ ctc\ E) \mid (\mathsf{check}\ j\ k\ E\ v) \end{aligned}$$

Figure 7: CtcPCF evaluation contexts.

Rules check-true and check-false give semantics to checking contract predicates. If the predicate returns true for the contract-monitored value, check passes the value to the client context. Otherwise, it halts the execution of the program and raises a contract error.

2.3 Correct Blame and Complete Monitoring

The semantics presented in the previous section gives a concrete meaning to contracts, but does not present a rationale for the design of its contract system. In this section, we consider two desirable properties of a contract system and how they influence its design: correct blame and complete monitoring.

Recall that one of the benefits of contracts over more ad-hoc approaches to checking the ineractions between components is the ability to identify the part of a program responsible for an error. But how should this part be determined, and what does it mean to be responsible for an error? While a number of answers to this question have been proposed [13, 37], we follow the criterion for correct blame given by Dimoulas et al. [21, 23, 25]: "a contract system should blame a party only if *the party controls the flow or return of values into the particular contract check that fails*."

The second criterion, complete monitoring, was also introduced by Dimoulas et al. [21, 25]. It strengthens the correct blame property to also require that every flow of values between components can be monitored by a contract. As we will see in the rest of this dissertation, this property

$\langle E[((\lambda\ (x : \tau)\ e_1)\ v_2)],\ \sigma\rangle$	$\longrightarrow \langle E[e_1[x \mapsto v_2]],\ \sigma\rangle$	[app]
$\langle E[(\mu\ (x : \tau)\ e)],\ \sigma\rangle$	$\longrightarrow \langle E[e[x \mapsto (\mu\ (x : \tau)\ e)]],\ \sigma\rangle$	[fix]
$\langle E[(\textsf{let}\ (x\ v)\ e)],\ \sigma\rangle$	$\longrightarrow \langle E[e[x \mapsto v]],\ \sigma\rangle$	[let]
$\langle E[(\textsf{if}\ \#\textsf{t}\ e_1\ e_2)],\ \sigma\rangle$	$\longrightarrow \langle E[e_1],\ \sigma\rangle$	[if-true]
$\langle E[(\textsf{if}\ \#\textsf{f}\ e_1\ e_2)],\ \sigma\rangle$	$\longrightarrow \langle E[e_2],\ \sigma\rangle$	[if-false]
$\langle E[(n_1 + n_2)],\ \sigma\rangle$	$\longrightarrow \langle E[n],\ \sigma\rangle$ where $n = n_1 + n_2$	[plus]
$\langle E[(n_1 - n_2)],\ \sigma\rangle$	$\longrightarrow \langle E[n],\ \sigma\rangle$ where $n = n_1 - n_2$	[minus]
$\langle E[(n_1 \leq n_2)],\ \sigma\rangle$	$\longrightarrow \langle E[v],\ \sigma\rangle$ where $v = n_1 \leq n_2$	[less-than-equal]
$\langle E[(\textsf{ref}\ v)],\ \sigma\rangle$	$\longrightarrow \langle E[(\textsf{loc}\ r)],\ \sigma[r \mapsto v]\rangle$ where r fresh	[ref]
$\langle E[((\textsf{loc}\ r) := v)],\ \sigma\rangle$	$\longrightarrow \langle E[v],\ \sigma[r \mapsto v]\rangle$	[update]
$\langle E[(!\ (\textsf{loc}\ r))],\ \sigma\rangle$	$\longrightarrow \langle E[v],\ \sigma\rangle$ where $\sigma(r) = v$	[deref]
$\langle E[(\textsf{mon}\ j\ k\ l\ (\textsf{flat/c}\ v_c)\ v)],\ \sigma\rangle$	$\longrightarrow \langle E[(\textsf{check}\ j\ k\ (v_c\ v)\ v)],\ \sigma\rangle$	[flat/c]
$\langle E[(\textsf{mon}\ j\ k\ l\ (\textsf{contract/c} : \tau)\ ctc)],\ \sigma\rangle$	$\longrightarrow \langle E[ctc],\ \sigma\rangle$	[contract/c]
$\langle E[(\textsf{mon}\ j\ k\ l\ (\rightarrow : \tau\ ctc_d\ ctc_r)\ v)],\ \sigma\rangle$	$\longrightarrow \langle E[(\lambda\ (x : \tau_d)\ (\textsf{mon}\ j\ k\ l\ ctc_r\ (v\ (\textsf{mon}\ j\ l\ k\ ctc_d\ x))))],\ \sigma\rangle$ where $(\tau_d \rightarrow \tau_r) = \tau$	[→/c]
$\langle E[(\textsf{mon}\ j\ k\ l\ (\textsf{ref/c}\ ctc)\ v)],\ \sigma\rangle$	$\longrightarrow \langle E[(\textsf{loc/p}\ j\ k\ l\ ctc)],\ \sigma\rangle$	[ref/c]
$\langle E[(!\ (\textsf{loc/p}\ j\ k\ l\ ctc\ v))],\ \sigma\rangle$	$\longrightarrow \langle E[(\textsf{mon}\ j\ k\ l\ ctc\ (!\ v))],\ \sigma\rangle$	[loc/p-read]
$\langle E[((\textsf{loc/p}\ j\ k\ l\ ctc\ v_r) := v)],\ \sigma\rangle$	$\longrightarrow \langle E[(v_r := (\textsf{mon}\ j\ l\ k\ ctc\ v))],\ \sigma\rangle$	[loc/p-write]
$\langle E[(\textsf{check}\ j\ k\ \#\textsf{t}\ v)],\ \sigma\rangle$	$\longrightarrow \langle E[v],\ \sigma\rangle$	[check-true]
$\langle E[(\textsf{check}\ j\ k\ \#\textsf{f}\ v)],\ \sigma\rangle$	$\longrightarrow \langle(\textsf{error}\ j\ k),\ \sigma\rangle$	[check-false]

Figure 8: CtcPCF Reduction Semantics.

$$\begin{aligned} v &::= \ldots. \mid (\mathsf{own}\ v\ l) \mid (\mathsf{obl}\ v\ ls) \\ e &::= \ldots. \mid (\mathsf{own}\ e\ l) \mid (\mathsf{obl}\ e\ j\ ks\ ls) \\ E &::= \ldots. \mid (\mathsf{own}\ E\ l) \mid (\mathsf{obl}\ E\ j\ ks\ ls) \\ O &::= [] \mid (\mathsf{own}\ O\ l) \end{aligned}$$

Figure 9: Ownership annotations.

provides a firm foundation on which to build enforcement mechanisms for a large class of properties.

To characterize these criteria and prove that a contract system satisfies them, Dimoulas et al. extend the contract system's syntax and semantics with annotations describing the ownership of terms and the obligations that contracts impose on values. These syntax extensions for CtcPCF are given in Figure 9.

The ownership annotations $(\mathsf{own}\ v\ l)$ and $(\mathsf{own}\ e\ l)$ indicate that the terms v and e are owned by the corresponding components l. The obligation annotation $(\mathsf{obl}\ v\ (l\ \ldots))$ says that components $(l\ \ldots)$ are responsible for satisfying the contract v. Obligation annotation $(\mathsf{obl}\ e\ j\ (k\ \ldots)\ (l\ \ldots))$ is a *delayed* obligation which denotes that expression e, which will evaluate to a contract, is owned by component j, and that components $(k\ \ldots)$ are responsible for contracts appearing in positive positions in the contract and components $(l\ \ldots)$ are responsible for contracts appearing in negative positions. Positive positions are those in which the contract applies to a value flowing from the server to the client. Negative positions are those in which the contract applies to a value flowing from the client to the server. An ownership context O is either a hole [] or an annotated value $(\mathsf{own}\ O\ l)$. Term $O[v]$ represents the value v wrapped in zero or more ownership annotations.

In order to make sense of ownership and obligations, the annotations in a program must be consistent with the component structure of the program. The judgment $\Gamma; \Sigma; l \Vdash e$, given in Figure 10 formalizes this idea. Ownership environments Γ and Σ associate owners with variables and

$$\frac{}{\Gamma; \Sigma; l \Vdash n} \quad \frac{}{\Gamma; \Sigma; l \Vdash ()} \quad \frac{}{\Gamma; \Sigma; l \Vdash \#\mathrm{t}} \quad \frac{}{\Gamma; \Sigma; l \Vdash \#\mathrm{f}} \quad \frac{}{\Gamma; \Sigma; l \Vdash (\mathsf{contract/c} : \tau)} \quad \frac{\Gamma[x \mapsto l]; \Sigma; l \Vdash e}{\Gamma; \Sigma; l \Vdash (\lambda\ (x : \tau)\ e)}$$

$$\frac{\begin{array}{c}\Gamma; \Sigma; l \Vdash e_1 \\ \Gamma[x \mapsto l]; \Sigma; l \Vdash e_2\end{array}}{\Gamma; \Sigma; l \Vdash (\mathsf{let}\ (x\ e_1)\ e_2)} \quad \frac{\Sigma(r) = l}{\Gamma; \Sigma; l \Vdash (\mathsf{loc}\ r)} \quad \frac{\Gamma; \Sigma; l \Vdash e}{\Gamma; \Sigma; l \Vdash (\mathsf{ref}\ e)} \quad \frac{\Gamma; \Sigma; l \Vdash e}{\Gamma; \Sigma; l \Vdash (!\ e)} \quad \frac{\begin{array}{c}\Gamma; \Sigma; l \Vdash e_1 \\ \Gamma; \Sigma; l \Vdash e_2\end{array}}{\Gamma; \Sigma; l \Vdash (e_1 := e_2)}$$

$$\frac{\begin{array}{c}\Gamma; \Sigma; k \Vdash v \\ \Gamma; \Sigma; (k\ l); (k\ l); j \triangleright ctc\end{array}}{\Gamma; \Sigma; l \Vdash (\mathsf{loc/p}\ j\ k\ l\ ctc\ v)} \quad \frac{\Gamma(x) = l}{\Gamma; \Sigma; l \Vdash x} \quad \frac{\begin{array}{c}\Gamma; \Sigma; l \Vdash e_1 \\ \Gamma; \Sigma; l \Vdash e_2\end{array}}{\Gamma; \Sigma; l \Vdash (e_1\ e_2)} \quad \frac{\Gamma[x \mapsto l]; \Sigma; l \Vdash e}{\Gamma; \Sigma; l \Vdash (\mu\ (x : \tau)\ e)} \quad \frac{\begin{array}{c}\Gamma; \Sigma; l \Vdash e_1 \\ \Gamma; \Sigma; l \Vdash e_2 \\ \Gamma; \Sigma; l \Vdash e_3\end{array}}{\Gamma; \Sigma; l \Vdash (\mathsf{if}\ e_1\ e_2\ e_3)}$$

$$\frac{\begin{array}{c}\Gamma; \Sigma; l \Vdash e_1 \\ \Gamma; \Sigma; l \Vdash e_2\end{array}}{\Gamma; \Sigma; l \Vdash (e_1 + e_2)} \quad \frac{\begin{array}{c}\Gamma; \Sigma; l \Vdash e_1 \\ \Gamma; \Sigma; l \Vdash e_2\end{array}}{\Gamma; \Sigma; l \Vdash (e_1 - e_2)} \quad \frac{\begin{array}{c}\Gamma; \Sigma; l \Vdash e_1 \\ \Gamma; \Sigma; l \Vdash e_2\end{array}}{\Gamma; \Sigma; l \Vdash (e_1 \leq e_2)} \quad \frac{\Gamma; \Sigma; l \Vdash e}{\Gamma; \Sigma; l \Vdash (\mathsf{flat/c}\ e)} \quad \frac{\Gamma; \Sigma; l \Vdash e}{\Gamma; \Sigma; l \Vdash (\mathsf{ref/c}\ e)}$$

$$\frac{\begin{array}{c}\Gamma; \Sigma; l \Vdash e_1 \\ \Gamma; \Sigma; l \Vdash e_2\end{array}}{\Gamma; \Sigma; l \Vdash (\rightarrow\ : \tau\ e_1\ e_2)} \quad \frac{\begin{array}{c}\Gamma; \Sigma; k \Vdash (\mathsf{own}\ e_2\ k) \\ \Gamma; \Sigma; (k); (l); j \triangleright e_1\end{array}}{\Gamma; \Sigma; l \Vdash (\mathsf{mon}\ j\ k\ l\ e_1\ (\mathsf{own}\ e_2\ k))} \quad \frac{}{\Gamma; \Sigma; l \Vdash (\mathsf{error}\ j\ k)}$$

$$\frac{\begin{array}{c}\Gamma; \Sigma; j \Vdash e_1 \\ \Gamma; \Sigma; l \Vdash e_2\end{array}}{\Gamma; \Sigma; l \Vdash (\mathsf{check}\ j\ k\ e_1\ e_2)} \quad \frac{\Gamma; \Sigma; l \Vdash e}{\Gamma; \Sigma; l \Vdash (\mathsf{own}\ e\ l)}$$

Figure 10: Well-formed terms.

$$\frac{\begin{array}{c} k \in ks \\ \Gamma;\ \Sigma;\ j \Vdash v \end{array}}{\Gamma;\ \Sigma;\ ks;\ ls;\ j \triangleright (\textsf{obl}\ (\textsf{flat/c}\ v)\ k)} \qquad \frac{\begin{array}{c} \Gamma;\ \Sigma;\ ls;\ ks;\ j \triangleright v_1 \\ \Gamma;\ \Sigma;\ ks;\ ls;\ j \triangleright v_2 \end{array}}{\Gamma;\ \Sigma;\ ks;\ ls;\ j \triangleright (\rightarrow : \tau\ v_1\ v_2)} \qquad \frac{}{\Gamma;\ \Sigma;\ ks;\ ls;\ j \triangleright (\textsf{contract/c} : \tau)}$$

$$\frac{\begin{array}{c} ls_2 = ks \cup ls \\ \Gamma;\ \Sigma;\ ls_2;\ ls_2;\ j \triangleright v \end{array}}{\Gamma;\ \Sigma;\ ks;\ ls;\ j \triangleright (\textsf{ref/c}\ v)} \qquad \frac{}{\Gamma;\ \Sigma;\ ks;\ ls;\ j \triangleright (\textsf{obl}\ e\ j\ ks\ ls)}$$

Figure 11: Well-formed contracts.

locations referenced by a term. An expression is well-formed if all expressions within a component are annotated with that component's ownership label. Within an expression, the ownership of sub-expressions may only differ from the expression where the sub-expression is separated by a contract monitor with the corresponding client and server labels. In this way, monitor expressions serve as component boundaries.

Assigning blame correctly also requires that contracts correctly assign responsibility for their constituent predicates to the correct components. To formalize this property, the well-formedness of contract monitors also requires that the monitor's contract is a well-formed contract. The judgment $\Gamma;\ \Sigma;\ (k\ ...);\ (l\ ...);\ j \triangleright e$ requires that ownership annotations within the contract are well-formed with respect to the contract's ownership label j. In addition, it requires that obligation annotations appearing within the contract are assigned to the correct party, giving a clear definition to the "positive" and "negative" parties described above. Because CtcPCF supports first-class contracts, the definition of well-formed contracts differs from the judgment given by Dimoulas et al. [25]: it has an additional rule for delayed obligation annotations that simply requires that they are well-formed with respect to the context. The semantics of annotated CtcPCF will ensure that these delayed obligations are correctly assigned to contracts as soon as the contract attached

$\langle E[(O_1[(\lambda\ (x : \tau)\ e_1)]\ O_2[v_2])],\ \sigma\rangle \longrightarrow$ [app]

$\langle E[e_1[x \mapsto (\text{own}\ v_2\ l)]],\ \sigma\rangle$

where l = context-label⟦E, l_0⟧, ownership-free⟦v_2⟧, owned⟦O_1, l⟧, owned⟦O_2, l⟧

$\langle E[(\mu\ (x : \tau)\ e)],\ \sigma\rangle \longrightarrow$ [fix]

$\langle E[e[x \mapsto (\text{own}\ (\mu\ (x : \tau)\ e)\ l)]],\ \sigma\rangle$

where l = context-label⟦E, l_0⟧

$\langle E[(\text{let}\ (x\ O[v])\ e)],\ \sigma\rangle \longrightarrow$ [let]

$\langle E[e[x \mapsto (\text{own}\ v\ l)]],\ \sigma\rangle$

where l = context-label⟦E, l_0⟧, ownership-free⟦v⟧, owned⟦O, l⟧

$\langle E[(\text{if}\ O[\#t]\ e_1\ e_2)],\ \sigma\rangle \longrightarrow$ [if-true]

$\langle E[e_1],\ \sigma\rangle$

where l = context-label⟦E, l_0⟧, owned⟦O, l⟧

$\langle E[(\text{if}\ O[\#f]\ e_1\ e_2)],\ \sigma\rangle \longrightarrow$ [if-false]

$\langle E[e_2],\ \sigma\rangle$

where l = context-label⟦E, l_0⟧, owned⟦O, l⟧

$\langle E[(O_1[n_1] + O_2[n_2])],\ \sigma\rangle \longrightarrow$ [plus]

$\langle E[n],\ \sigma\rangle$

where l = context-label⟦E, l_0⟧, $n = n_1 + n_2$, owned⟦O_1, l⟧, owned⟦O_2, l⟧

$\langle E[(O_1[n_1] - O_2[n_2])],\ \sigma\rangle \longrightarrow$ [minus]

$\langle E[n],\ \sigma\rangle$

where l = context-label⟦E, l_0⟧, $n = n_1 - n_2$, owned⟦O_1, l⟧, owned⟦O_2, l⟧

$\langle E[(O_1[n_1] \leq O_2[n_2])],\ \sigma\rangle \longrightarrow$ [less-than-equal]

$\langle E[v],\ \sigma\rangle$

where l = context-label⟦E, l_0⟧, $v = n_1 \leq n_2$, owned⟦O_1, l⟧, owned⟦O_2, l⟧

$\langle E[(\text{ref}\ O[v])],\ \sigma\rangle \longrightarrow$ [ref]

$\langle E[(\text{loc}\ r)],\ \sigma[r \mapsto v]\rangle$

where ownership-free⟦v⟧, l = context-label⟦E, l_0⟧, owned⟦O, l⟧, r fresh

$\langle E[(O_1[(\text{loc}\ r)] := O_2[v])],\ \sigma\rangle \longrightarrow$ [update]

$\langle E[(\text{own}\ v\ l)],\ \sigma[r \mapsto v]\rangle$

where l = context-label⟦E, l_0⟧, ownership-free⟦v⟧, owned⟦O_1, l⟧, owned⟦O_2, l⟧

$\langle E[(!\ O[(\text{loc}\ r)])],\ \sigma\rangle \longrightarrow$ [deref]

$\langle E[(\text{own}\ v\ l)],\ \sigma\rangle$

where $\sigma(r) = v$, l = context-label⟦E, l_0⟧, owned⟦O, l⟧

Figure 13: Annotated CtcPCF Reduction Semantics.

$\langle E[(\mathsf{obl}\ O[(\mathsf{flat/c}\ v)]\ j\ ks\ ls)],\ \sigma\rangle \longrightarrow$ [flat/c-obl]
$\langle E[(\mathsf{own}\ (\mathsf{obl}\ (\mathsf{flat/c}\ v)\ ks)\ j)],\ \sigma\rangle$
where $j = \text{context-label}[\![E, l_0]\!]$, $\text{owned}[\![O, j]\!]$

$\langle E[(\mathsf{obl}\ O[(\rightarrow : \tau\ v_1\ v_2)]\ j\ ks\ ls)],\ \sigma\rangle \longrightarrow$ [→-obl]
$\langle E[(\rightarrow : \tau\ (\mathsf{obl}\ v_1\ j\ ls\ ks)\ (\mathsf{obl}\ v_2\ j\ ks\ ls))],\ \sigma\rangle$
where $j = \text{context-label}[\![E, l_0]\!]$, $\text{owned}[\![O, j]\!]$

$\langle E[(\mathsf{obl}\ O[(\mathsf{contract/c} : \tau)]\ j\ ks\ ls)],\ \sigma\rangle \longrightarrow$ [contract/c-obl]
$\langle E[(\mathsf{own}\ (\mathsf{contract/c} : \tau)\ j)],\ \sigma\rangle$
where $j = \text{context-label}[\![E, l_0]\!]$, $\text{owned}[\![O, j]\!]$

$\langle E[(\mathsf{obl}\ O[(\mathsf{ref/c}\ v)]\ j\ ks\ ls)],\ \sigma\rangle \longrightarrow$ [ref/c-obl]
$\langle E[(\mathsf{own}\ (\mathsf{ref/c}\ (\mathsf{obl}\ v\ j\ ls_2\ ls_2))\ j)],\ \sigma\rangle$
where $j = \text{context-label}[\![E, l_0]\!]$, $ls_2 = ks \cup ls$, $\text{owned}[\![O, j]\!]$

$\langle E[(\mathsf{mon}\ j\ k\ l\ O_1[(\mathsf{obl}\ (\mathsf{flat/c}\ v_c)\ ls)]\ O_2[v])],\ \sigma\rangle \longrightarrow$ [flat/c]
$\langle E[(\mathsf{check}\ j\ k\ (\mathsf{own}\ (v_c\ v)\ j)\ v)],\ \sigma\rangle$
where $l = \text{context-label}[\![E, l_0]\!]$, $k \in ls$, $\text{ownership-free}[\![v]\!]$, $\text{owned}[\![O_1, j]\!]$, $\text{owned}[\![O_2, k]\!]$

$\langle E[(\mathsf{mon}\ j\ k\ l\ O_1[(\mathsf{contract/c} : \tau)]\ O_2[ctc])],\ \sigma\rangle \longrightarrow$ [contract/c]
$\langle E[(\mathsf{own}\ \text{strip}[\![ctc, k]\!]\ l)],\ \sigma\rangle$
where $l = \text{context-label}[\![E, l_0]\!]$, $\text{ownership-free}[\![ctc]\!]$, $\text{owned}[\![O_1, j]\!]$, $\text{owned}[\![O_2, k]\!]$

$\langle E[(\mathsf{mon}\ j\ k\ l\ O_1[(\rightarrow : \tau\ ctc_d\ ctc_r)]\ O_2[v])],\ \sigma\rangle \longrightarrow$ [→]
$\langle E[(\lambda\ (x : \tau_d)\ (\mathsf{mon}\ j\ k\ l\ ctc_r\ (v\ (\mathsf{mon}\ j\ l\ k\ ctc_d\ x))))],\ \sigma\rangle$
where $l = \text{context-label}[\![E, l_0]\!]$, $(\tau_d \rightarrow \tau_r) = \tau$, $\text{ownership-free}[\![v]\!]$, $\text{owned}[\![O_1, j]\!]$, $\text{owned}[\![O_2, k]\!]$

$\langle E[(\mathsf{mon}\ j\ k\ l\ O_1[(\mathsf{ref/c}\ O_2[ctc])]\ O_3[v])],\ \sigma\rangle \longrightarrow$ [ref/c]
$\langle E[(\mathsf{loc/p}\ j\ k\ l\ ctc\ v)],\ \sigma\rangle$
where $l = \text{context-label}[\![E, l_0]\!]$, $\text{ownership-free}[\![v]\!]$, $\text{owned}[\![O_1, j]\!]$, $\text{owned}[\![O_2, j]\!]$, $\text{owned}[\![O_3, k]\!]$

$\langle E[(!\ O[(\mathsf{loc/p}\ j\ k\ l\ ctc\ v)])],\ \sigma\rangle \longrightarrow$ [loc/p-read]
$\langle E[(\mathsf{own}\ (\mathsf{mon}\ j\ k\ l\ (\mathsf{own}\ ctc\ j)\ (!\ v))\ l)],\ \sigma\rangle$
where $l = \text{context-label}[\![E, l_0]\!]$, $\text{owned}[\![O, l]\!]$

$\langle E[(O_1[(\mathsf{loc/p}\ j\ k\ l\ ctc\ v_r)] := O_2[v])],\ \sigma\rangle \longrightarrow$ [loc/p-write]
$\langle E[(v_r := (\mathsf{mon}\ j\ l\ k\ ctc\ (\mathsf{own}\ v\ l)))],\ \sigma\rangle$
where $l = \text{context-label}[\![E, l_0]\!]$, $\text{ownership-free}[\![v]\!]$, $\text{owned}[\![O_1, l]\!]$, $\text{owned}[\![O_2, l]\!]$

$\langle E[(\mathsf{check}\ j\ k\ O[\#t]\ v)],\ \sigma\rangle \longrightarrow$ [check-true]
$\langle E[v],\ \sigma\rangle$
where $\text{owned}[\![O, j]\!]$

$\langle E[(\mathsf{check}\ j\ k\ O[\#f]\ v)],\ \sigma\rangle \longrightarrow$ [check-false]
$\langle (\mathsf{error}\ j\ k),\ \sigma\rangle$
where $\text{owned}[\![O, j]\!]$

$\langle E[(\mathsf{error}\ j\ k)],\ \sigma\rangle \longrightarrow$ [error]
$\langle (\mathsf{error}\ j\ k),\ \sigma\rangle$

Figure 13: Annotated CtcPCF Reduction Semantics (Continued).

by a contract monitor is reduced to a value.

Finally, the reduction semantics of CtcPCF is modified to explicitly track the ownership of terms and record contract obligations. The semantics, given in Figure 13 are mostly unchanged from Figure 8. However, the semantics now require that all sub-expressions of the current reducible expression have the same ownership annotation, unless the expression is an explicit boundary between components, that is, a contract monitor. The judgment $\text{owned}[\![O, l]\!]$ is true when all of the ownership annotations appearing in the stack of ownership annotations O have the same label, l. Meta-function $\text{context-label}[\![E, l_0]\!]$ evaluates to the ownership label of the annotated expression closest to the hole in the evaluation context E, or to label l_0, if there is no such annotation. For a contract monitor $(\textsf{mon}\ j\ k\ l\ v_c\ v)$, the semantics requires that the contract v_c is owned by component j and the value v is owned by the server component k. The rules for obligation annotations, flat/c-obl, →-obl, contract/c-obl, and ref/c-obl, propagate obligation annotations according to the rules of well-formed contracts.

With this annotated semantics in hand, we can now formally define blame correctness and complete monitoring in the style of Dimoulas et al. [21, 25].

Definition A contract system is *blame correct* if and only if for all terms e_0 such that $\varnothing; \varnothing; l_0 \Vdash e_0$, if $\langle e_0, \varnothing\rangle \longrightarrow^{\star} \langle E[(\textsf{mon}\ j\ k\ l\ (\textsf{obl}\ (\textsf{flat/c}\ v_c)\ ls)\ v)], \sigma\rangle$, then v is owned by component k and $k \in ls$.

Blame correctness requires that when a well-formed program checks a flat contract, the owner of the value and the server label of the monitor coincide, and furthermore, that component will be blamed if the contract fails.

Definition A contract system is a *complete monitor* if and only if for all well-typed terms e_0 such that $\varnothing; \varnothing; l_0 \Vdash e_0$, either

- $\langle e_0, \varnothing\rangle \longrightarrow^\star \langle v, \sigma\rangle$,
- for all e_1 and σ_1 such that $\langle e_0, \varnothing\rangle \longrightarrow^\star \langle e_1, \sigma_1\rangle$, there exist e_2 and σ_2 such that $\langle e_1, \sigma_1\rangle \longrightarrow \langle e_2, \sigma_2\rangle$, or
- $\langle e_0, \varnothing\rangle \longrightarrow^\star \langle e_1, \sigma_1\rangle \longrightarrow^\star \langle(\textsf{error}\ j\ k), \sigma_2\rangle$ and for all such terms e_1, e_1 has the form $E[(\textsf{mon}\ j\ k\ l\ (\textsf{own}\ (\textsf{obl}\ (\textsf{flat/c}\ v_c)\ ls)\ j)\ v)]$ where E is owned by component l, v is owned by k, and $k \in ls$.

This definition requires that the annotated semantics do not get stuck for well-formed terms. This, coupled with the ownership restrictions imposed by the annotated semantics, ensures that values flow between components only at explictly marked component boundaries that enforce contracts.

Using the subject reduction developed by Dimoulas et al. [25], it can be shown that the contract system presented here satisfies both correct blame and complete monitoring.

2.4 Related Work

There have been two, largely orthognal directions of research on behavioral contracts. The first has focused primarily on first-order or nominal higher-order contracts and investigates how contracts can be statically verified and used to statically prove properties of programs. The second direction extends Findler and Felleisen's work on structural higher-order contracts and investigates their semantics, extensions to other langauge features, and extensions to easily specify particular classes of program properties.

Verification with behavioral contracts   A number of program verification systems have been developed that use contracts as code annotations to express Hoare-style pre- and post-conditions on components [7, 8, 11, 15, 18, 51, 72]. Rather than execute contracts at runtime, these systems use automatic or interactive verification techniques to statically ensure that components satisfy their specifications and are used correctly.

Verification using contract specifications has been applied to a number of security problems. For example, SPARK Ada's contracts can express dataflow constraints on how component inputs may influence component outputs [7]. More recently, the Ironclad approach to full-system verification has used similar dataflow contraints to verify information-flow properties of applications [48]. Contracts have also been used to express and verify access control policies. Pavlova et al. [86] compile high-level policies restricting component access to particular APIs into software contracts. Using weakest-precondition propagation, the inserted software contracts are sufficient for static verification of the high-level policies. Similar approaches have been used to verify that stack inspection policies are never violated [105] and to verify that a program satisfies trace properties expressed in temporal logic [41].

The static security guarantees provided by these methods reduce the overhead of checking security properties at runtime and eliminate the possibility of runtime security violations. However, they require extensive annotations throughout the program to support verification. In contrast, the run-time enforced security contracts presented in this dissertation require annotations only on components directly related to security requirements. Furthermore, enforcing a desired property requires first encoding it using annotations supported by the verification tool. Doing this with the behavioral contract languages present in these systems may require encoding the property imperatively using additional "ghost" state [65]. For example, a temporal property could be enforced by translating the property into a finite state machine, recording the current state in a variable, and adding annotations throughout the program to update and check the state. For complex policies, the resulting contracts quickly become difficult to write and understand, reducing the usefulness of contracts for documentation [86]. One approach to combatting this complexity is to extend the contract specification language to support directly encoding certain properties. For example, Trentelman and Huisman [121] extend the JML contract language with support for properties expressed in temporal logic.

Higher-order contracts for additional language features Using the implementation techniques pioneered by Findler and Felleisen [37], contract systems have been developed to enforce behavioral properties of a wide range of language features including objects with first-class classes [112], delimited continuations [117], and parametric polymorphism [44]. Building on this work, contract systems have been used extensively in the development of gradual type systems, which blend typed and untyped languages by using contracts to ensure that untyped code cannot subvert the guarantees of the type system [43, 104, 115, 116, 118].

Higher-order contracts for specific properties In addition to the type-like properties enforced by the higher-order contracts described in this chapter, contract systems have also been proposed that allow programmers to succinctly express a wide range of desirable properties.

For example, Heidegger et al. [49] introduce access permission contracts, which restrict what paths through object fields a method may read or write to during execution. In a capability-based security setting, these access permission contracts are similar to the contracts we use to restrict the use of sensitive resources. This dissertation shows how by enriching contracts with additional features such as bounded polymorphism, more sophisticated security policies can be enforced.

Disney et al.'s higher-order temporal contracts [26] enforce trace properties on the order in which procedures are called and returned. These contracts can be used to enforce arbitrary execution-monitoring enforceable [99] trace properties of a program, but unlike authorization contracts, provide limited support for writing complex access control policies like stack inspection, capabilities, or discretionary access control.

3 Capability Contracts

As we have seen, software contracts provide an expressive language for specifying and enforcing interesting properties about how different components interact by exchanging values. For example, contracts can enforce that the arguments given to a procedure satisfy a precondition, or that calls to a procedure occur in a specified order. Unfortunately, it is not immediately clear how to use such properties to control which users or parts of a program are permitted to access particular resources or perform particular actions—in most langauges, a component may access resources or perform actions via channels that are not evident from the values it is passed as arguments.

It is possible, however, to design languages and systems where security can be expressed using restrictions on the flow of values. This *object-capability model* of security was first described by

Dennis and Van Horn [20] in their design of a "Programming Semantics for Multiprogrammed Computations," and has since been used as the basis of a wide range of secure systems (e.g., KeyKOS [45], seL4 [50], and Capsicum [126]) and programming languages (e.g., Gedanken [94], W7 [93], E [80], Joe-E [73], and Caja [78]).

At the core of this model is the notion of a *capability*, which is a value that corresponds to the right to access a resource or perform an action. Possessing a capability grants its holder the undisputed right to perform the underlying action. Security in a capability system is achieved by carefully controlling which components are given access to individual capabilities. Crucially, capabilities cannot be forged, but instead must be either granted to a component by some other component that already possessed the capability or derived from an existing capability.

The *object-capability* model takes this notion one step further, and mandates that the ability for components to communicate (and thereby to share capabilities) must also be mediated by capabilities. In object-capability languages, this is acheived by treating every reference to an object (or closure in a functional language) as a capability to invoke that object. Thus objects become both the components that may individually possess capabilities and the protected resources that require capabilities to access.

By enforcing restrictions on how objects may share references, these languages empower components to create their own *access abstractions* [80] that grant other components limited access to capabilities by instead granting access to specially-created proxy components that forward only acceptable requests. Building on this idea, appropriate design patterns can enforce fine-grained application-specific access control requirements, including confinement and selective revocation [79].

This chapter shows how software contracts can be used to express and enforce security policies in capability-safe programming languages. In Section 3.1, we examine the relationship between the interposition mechanisms provided by software contracts and the rules of the object-capability model. Building on this, in Section 3.2 we show how several object-capability design patterns can be expressed as contracts and how doing so can improve the readability and design of programs written in object-capability languages. In Chapter 4 we will use the ideas from this Chapter in the the design and implementation of a secure shell scripting language. The language, SHILL, uses capabilities to restrict the effects of running a script to those explicitly enabled by the capabilities it is passed as arguments. Furthermore, it uses contracts to give users fine-grained control over how scripts use capabilities and to provide easy-to-understand specifications of what a script might do.

3.1 Capability-safety and Complete Monitoring

To explore the object-capability model in more depth, we will consider small examples written in a small programming language inspired by E [80], dubbed E-on-Racket. Like E, E-on-Racket is a dynamically-typed, object-oriented language. The language does not implement all of the features of E, but captures the important features of E that enable object capability-based security. Here, we elide most of the details of the language and just explain those that are relevant to the examples that follow.

An object definition has the form

```
(def name
  (to (method arg ...) body ...)
  ...
  (recv (method-name args) body ...))
```

and creates a new object bound to the identifier `name`. Each method definition

```
(to (method arg ...) body ...)
```

defines a method of the new object with the name `method`. Sending a message to an object `(send obj method-name arg ...)` invokes the method of `obj` with the same name. If the object does not have a method with the given name, the `recv` method is invoked with the requested method's name as the first argument and a list of the arguments as the second argument. The `recv` method is not required in an object definition. If it was not included in the object's definition, the attempt to send the message fails, throwing an exception. The alternative operation `(call obj name args)` also sends a message to the object `obj`, but `name` is an expression that evaluates to a symbol corresponding to a method name, rather than a literal as in `send`.

A procedure definition `(def (name arg ...) body ...)` creates a new procedure and binds it to the name `name`. Procedures are represented as objects with a single method `run`. Invoking an object or procedure with the syntax `(obj arg ...)` is equivalent to sending a message: `(send obj run arg ...)`.

The language is strictly lexically scoped but supports mutation of lexically bound variables using the expression `(set! id expr)`.

Since E-on-Racket is an object-capability language like E, there are a limited number of ways that an object can acquire a reference to another object:

1. by initial conditions: two objects may reference each other before a computation begins,
2. by parenthood: the creator of an object is initially the only object with a reference to it,
3. by endowment: an object can close over references to objects available in its parent's environment, and
4. by introduction: an object can receive references to other objects passed as arguments to its methods or returned from methods it invokes.

These restrictions on the flow of objects within a program are sometimes referred to as *capability safety*, and they provide a foundation for reasoning about programs [80].

```
(def (makeCounter)
  (let ([count 0])
    (def counter
      (to (incr)
        (set! count (+ count 2))
        count)
      (to (decr)
        (set! count (- count 2))
         count))
    counter))

(define myCounter (makeCounter))


> (send myCounter incr)
2
> (send myCounter incr)
4
> (send myCounter decr)
2
```

Figure 14: A counter object implemented in E-on-Racket.

Consider the example program in Figure 14. It defines a procedure `makeCounter` that returns a new object encapsulating a variable `count`. The returned object provides two methods, `incr` and `decr`, that increment the counter by two and decrement the counter by two, respectively. Suppose we are interested in ensuring that the counter value is always even. According to the rules of capability safety, we can determine which parts of the program have access to the value of the counter `count`: the procedure `makeCounter` (by parenthood) and the object `counter` (by endowment). To verify that the count is always even, we must ensure that neither of these objects modify `count` in a way that violates the invariant and that neither leaks access to the counter to another object. Clearly, the code only changes the counter in increments of two, and so ensures that its value is even. Furthermore, we can observe that the methods of `counter` are the only way to modify the counter value, since the values returned by the method cannot be mutated.

Beyond facilitating this kind of local reasoning about programs, the capability-safety rules also provide a mechanism for creating access abstractions that enforce security guarantees. For example, suppose that we wish to share the counter with an untrusted piece of code, but ensure that that code can only increase the value of the counter. Simply sharing the `counter` object cannot ensure this property. Instead, we can create a new object that encapsulates the counter and ensures that `decr` cannot be invoked. The program in Figure 15 demonstrates how to compose programs safely using this pattern. The untrusted code is represented by a procedure `evil` that takes as argument a counter object and a flag `decr?`. If `decr?` is `#t`, it attempts to decrement the counter, violating the policy. Otherwise, it increments the counter. The procedure `compose`, which integrates the untrusted code and the counter, creates a new object `incr-only` that partially implements the counter interface, but hides the method `decr`. When its `incr` method is invoked, it simply forwards the message to the original counter. Because this interposition is transparent to the `evil` procedure, the program works as expected as long as the untrusted code

```
(def (evil counter decr?)
  (if decr?
      (send counter decr)
      (send counter incr)))

(def (compose fun counter arg)
  (def incr-only
    (to (incr) (send myCounter incr)))
  (fun incr-only arg))

(define myCounter (makeCounter))


> (compose evil myCounter #f)
2
> (compose evil myCounter #t)
no such method: decr
```

Figure 15: Safely sharing a counter object.

obeys the policy. On the other hand, if the untrusted code tries to decrement the counter, its attempt will fail.

Capability-safety ensures that it is always possible to protect sensitive behaviors of objects by creating these types of proxy objects—an object can interpose at any boundary between it and another object by replacing the value it would send across the boundary with a new object. In essense, capability safety gives the necesary conditions for *complete mediation* of the interactions between objects, in the same way that a contract system enforcing complete monitoring (Section 2.3) ensures that it is always possible to interpose between components. This connection between complete monitoring and capability-safety was first noted by Dimoulas et al. [24], and provides the footing on which we develop contracts for capability-safe programs. Capability-safety is, however, a stronger property than complete monitoring, since it requires not only complete mediation between objects, but that all interposition points correspond to where values are exchanged according to strict lexical scoping rules. In Chapter 5, we explore how contracts can enable security when these restrictions are relaxed.

3.2 From Patterns to Contracts

To demonstrate the effectiveness of contracts for capabilities, we extend E-on-Racket with contracts for objects and a syntactic form for attaching those contracts to objects. An object contract `(obj/c [method contract] ... [recv fun])` takes as arguments a list of pairs of permitted method names and procedure contracts, along with an optional pair of the keyword `recv` and a procedure. Contracts are attached to objects using the `def/ctc` form, which mirrors `def` but allows a contract to be specified immediately after the definition's header. When a method on an object with an object contract is invoked, the method's name is first found in the list of permitted methods. If the method appears in the list, the call is allowed, but the selected method is restricted by the corresponding contract. If the method is not in the list but a `recv` procedure `fun` was provided, `fun` is invoked with the name of the requested method as an argument. It can return either a contract or `#f`. If a `recv` procedure was not provided in the contract, or it returns `#f`, the contract raises an error stating that the method could not be invoked. If `fun` returns a contract, the call is allowed to proceed, but the method is restricted using the returned contract. Like the contracts presented in Chapter 2, contracts in E-on-Racket are first-class values.

In the rest of this section, we show how a number of security design patterns can be written as contracts and argue that doing so makes programs easier to understand and their security properties easier to verify.

3.2.1 Attenuated Sharing

Recall the program from Figure 15 that uses a proxy object to prevent untrusted code from decrementing a counter. While this program does enforce the desired security policy, it is not trivial to verify this fact. First, we must check that none of the arguments to the `evil` procedure give access to the counter directly. Second, we must verify that the proxy object `incr-only` is implemented correctly. That is, it must not leak direct access to the counter object, it must not grant access to any of the counter's functionality except for the `incr` method, *and* it must correctly forward allowed requests to the underlying counter object.

Contrast this implementation with the program in Figure 16. This version uses software contracts to enforce the policy instead of constructing a proxy object. In particular, it specifies a contract for the `compose` procedure that grants untrusted code limited access to the counter. Its contract has four parts:

1. `((obj/c [incr (-> void)]) boolean? . -> . integer?)`, a contract for the `fun` argument which will receive untrusted code;

2. `obj?`, a contract for the `counter` argument, which will receive the unprotected counter;

```
(def (evil counter decr?)
  (if decr?
      (send counter decr)
      (send counter incr)))

(def/ctc (compose fun counter arg)
  (((obj/c [incr (-> void)]) boolean? . -> . integer?)
   obj? boolean? . -> . integer?)
  (fun counter arg))

(define myCounter (makeCounter))


> (compose evil myCounter #f)
2
> (compose evil myCounter #t)
compose: contract violation
  cannot call hidden method: 'decr
  in: the 1st argument of
      the 1st argument of
      (->
       (->
        (obj/c (incr (-> void)))
        boolean?
        integer?)
       obj?
       boolean?
       integer?)
  contract from: (function compose)
  blaming: program
   (assuming the contract is correct)
  at: eval:3.0
```

Figure 16: Safely sharing a counter object with contracts.

3. `boolean?`, a contract for the `arg` argument, which will be passed to `fun`; and

4. `integer?`, the contract for `compose` 's result.

While this contract is complex, it is straightforward to verify that it satisfies the desired security policy, since the contract describes exactly which values will be shared with the untrusted code and how they will be used. In addition, it is no longer necessary to double check that the proxy object `incr-only` was written correctly, since we can rely on the `obj/c` contract to impose the desired restrictions without modifying the behavior of the underlying object.

Moving the enforcement of this policy to the interface of the `compose` procedure has additional benefits. The contract provides documentation of how client code is allowed to use the objects it receives, which makes it easier to write and maintain programs that will not fail due to security violations. Furthermore, the semantics of contracts mean that if security violations do occur, it will be easier to track down which code is to blame. In the original program, the misbehaving component causes the program to fail with the message `no such method: decr`, but this provides no help discovering what code tried to access this method, whether or not it was prevented from accessing the method to enforce a security policy, or what code put the policy in place. Contrast this with the error message generated by the contract system. It reports that the method is hidden rather than simplying missing and that the `compose` procedure imposed this restriction on the first argument to its `fun` argument.

3.2.2 Revocation

More sophisticated design patterns can enforce more interesting policies. An early example of capability design patterns is Redell's "Caretaker" pattern, which demonstrates how to create revokable capabilities in the object-capability model [92]. An adapation of Miller's E implementation of the Caretaker pattern is given in Figure 17.

```
(def (makeCaretaker target)
  (let ([enabled? #t])
    (def caretaker
      (recv (name args)
        (if enabled?
            (call target name args)
            (error "disabled"))))
    (def gate
      (to (enable)  (set! enable? #t))
      (to (disable) (set! enable? #f)))
    (values caretaker gate)))
```

Figure 17: Redell's "Caretaker" pattern [92], as presented by Miller [80].

The purpose of this pattern is to allow an object to grant another temporary access to the `target` capability. The `makeCaretaker` procedure returns two objects, a `caretaker` and a `gate`, that close over a shared variable `enabled?`. The `caretaker` object is a proxy object for the `target`. It intercepts all messages to `target`, and forwards them only if `enabled?` is set to `#t`. If `enabled?` is `#f`, the capability has been revoked, and the `caretaker` raises an error. The `gate` object is retained by the `target` object's owner, and provides two methods, `enable` and `disable`, which allow the owner to enable or disable access to the `target` via the `caretaker`.

Because of the first-class nature of contracts, reproducing this design pattern using contracts is straightforward. Figure 18 defines a procedure `makeCaretakerContract` that generates a contract that can be used in place of a caretaker object. Unlike `makeCaretaker`, this procedure does not take an object to protect as an argument and return a proxy to that object, but instead returns a contract that can be attached to any object. As before, this contract closes over a variable `enabled?` that is shared with a guard object that controls whether or not methods of the protected object can be invoked. It specifies that no methods can be called by default. Each time

```
(def (makeCaretakerContract)
  (let* ([enabled? #t]
         [caretaker/c
          (obj/c [recv (λ (name) (if enabled? any/c #f))])])
    (def gate
      (to (enable)  (set! enabled? #t))
      (to (disable) (set! enabled? #f)))
    (values caretaker/c gate)))
```

Figure 18: Adapting the "Caretaker" pattern to contracts.

a method of an object with the contract is invoked, the `recv` function checks whether or not `enabled?` is set to `#t`, raising a contract error if it is `#f`.

While reasoning about the correct use of this capability still requires careful consideration of how the `gate` object is used, we still retain the benefits provided by the contract system, since clearly the `caretaker/c` contract only interferes with the behavior of the objects it protects by preventing accesses when the capability has been revoked. Furthermore, we can again push the code that wraps objects with caretakers to the interface of the untrusted code.

3.2.3 Membranes

A deficiency of the Caretaker pattern and other design patterns that interpose between objects and untrusted code is that the `target` object must be trusted not to leak an unprotected reference to itself to the untrusted code. Otherwise, the untrusted code could invoke methods of the object even after its access has been revoked. The "Membrane" pattern [80] is designed to solve the deficiency by recursively wrapping objects as they flow between components.

An E-on-Racket translation of Miller's membrane implementation is given in Figure 19. It works in the same way as the Caretaker pattern, but the `caretaker` object must also wrap the

```
(def (makeCaretakerMembrane target)
  (let ([enabled? #t])
    (def (wrap wrapped)
      (if (not (obj? wrapped))
          wrapped
          (begin
            (def caretaker
              (recv (name args)
                (if enabled?
                    (let ([wrappedArgs (map wrap args)])
                      (wrap (call wrapped name wrappedArgs)))
                    (error "disabled"))))
            caretaker)))
    (def gate
      (to (enable)  (set! enable #t))
      (to (disable) (set! enable #f)))
    (values (wrap target) gate)))
```

Figure 19: The membrane pattern.

arguments and return values of any method invoked on it with an additional instance of the `caretaker` pattern. When the owner of the `target` capability revokes it by calling `disable` on the gate, all of the objects that have passed through the membrane are revoked, cutting off any communcation channel between the `target` object and the untrusted code.

The resulting code is very complicated and its correctness is difficult to reason about. To guarantee that all access to the `target` is revoked, we must ensure that every value flowing from the `target` object or any value it returned is wrapped with a `caretaker` proxy. To see the difficulty in reasoning about this type of property, consider a naive solution to this problem that simply recursively wraps the result of any method call:

```
(if (not (obj? wrapped))
    wrapped
    (begin
      (def caretaker
        (recv (name args)
          (if enabled?
              (wrap (call wrapped name args))
              (error "disabled"))))
      caretaker))
```

This simple solution is subtly incorrect. The problem is that an argument passed to the `target` object might itself be an object. In that case, the `target` object can subvert the `caretaker` by communicating with the caller by invoking the argument object, rather than through `target` object's return value. A correct solution to this problem is the code shown before in Figure 19.

One of the advantages of contracts in comparison to these capability design patterns is that they provide a framework for composing contracts together to enforce more intricate properties while allowing their correctness to be considered independently. To see this, compare the previous implementation of a caretaker membrane with the contract-based implementation in Figure 20.

This implementation decomposes the task of creating a caretaker membrane into two parts.

```
(def (membrane/c ctc-in ctc-out)
  (recursive-contract
    (and/c
      ctc-out
      (or/c
        (obj/c
          [recv (λ (name)
                  (->... (membrane/c ctc-out ctc-in)
                         (membrane/c ctc-in ctc-out)))])
          any/c))))

(def (makeCaretakerMembraneContract)
  (let-values ([(caretaker/c gate) (makeCaretakerContract)])
    (values (membrane/c any/c caretaker/c) gate)))
```

Figure 20: Adapting the membrane pattern to contracts.

The first is the implementation of the `caretaker/c` contract in Figure 18. The second is a new contract combinator `membrane/c` that recursively applies contracts to all values flowing across a contract boundary. This combinator generalizes the membrane design pattern shown above in two ways. First, it is not specialized to the caretaker pattern, but can instead recursively attach any contract. Second, it takes as arguments two contracts, `ctc-in` and `ctc-out`, instead of one. The first contract `ctc-in` is applied to any value flowing *into* the protected object (or objects that flowed out of it). The second contract `ctc-out` is applied to any value flowing *out* of the protected object (or objects that flowed out of it).

The definition of `membrane/c` uses several contracts that have not yet been introduced. Contract `(recursive-contract expr)` delays the evaluation of `expr`, which evaluates to a contract, until it is attached to a value. This allows the contract itself to appear in the body of `expr` without causing the evaluation of `expr` to diverge. The contract `(->... ctc-arg ctc-`

`result)` is a contract that accepts procedures of any arity. It wraps every argument to the contracted procedure with contract `ctc-arg` and every result with the contract `ctc-result`. When it is attached to a value, the `membrane/c` contract applies two contracts to the value using the `and/c` combinator: `ctc-out`, since the value is being passed out of the membrane; and a second `or/c` contract that ensures contracts are recursively added to the value if it is an object or procedure. The `(->... (membrane/c ctc-out ctc-in) (membrane/c ctc-in ctc-out))` contracts take care of these recursive contract applications. Crucially, the membrane contract attached to arguments has the `ctc-in` and `ctc-out` contracts reversed to reflect that values passed to those arguments in the body of the procedure flow from the inside to the outside.

Again, careful reasoning is required to verify that this contract correctly imposes restrictions on all of the values it is recursively attached to. Unlike the previous implementation, however, this reasoning can be separated from reasoning about the implementation of the Caretaker pattern. The final combination of the two guarantees is achieved by simply composing the `membrane/c` combinator with a caretaker contract: `(membrane/c any/c caretaker/c)`.

3.2.4 Dynamic Sealing

The final design pattern we consider is the use of "Sealer-Unsealer" pairs to allow for the controlled amplification of access rights; that is, for the combination of two different capabilities to permit access to more capabilities than can be accessed by using the two capabilities separately. This pattern was introduced by Morris [82] as a technique to allow components to authenticate the values as having a particular provenance. Morris' account presents the pattern in the Gedanken programming language [94]. Here, we describe its use in E-on-Racket.

The key primitive of this pattern is an operation `createSeal` that returns a pair of procedures `seal` and `unseal`. The `seal` procedure takes a value as an argument and returns a new value

```
(def (makeMint)
  (def mint
    (to (makePurse balance)
       (def purse
         (to (getBalance) balance)
         (to (deposit amount src)
            (send src decr amount)
            (set! balance (+ balance amount)))
         (to (decr amount)
            (set! balance (- balance amount))))
       purse))
  mint)
```

Figure 21: An insecure mint.

that is completely opaque to any code except the `seal` procedure's matching `unseal` procedure. If the `unseal` procedure is invoked on this opaque value, it returns the original value. Invoking `unseal` on any other value raises an error.

To see how this pattern allows components to amplify the rights they possess, consider the example program in Figure 21, adapted from Miller et al.'s paper "Capability-Based Financial Instruments" [77]. This program implements a simulation of a system of accounts. Each `mint` object controls a virtual currency and provides a single method `makePurse` that returns a `purse` object with a starting balance. A purse represents an account and has two public operations: `getBalance`, which returns the current balance, and `deposit`, which transfers a specified amount of currency from one purse to another. To facilitate the transfer of currency, the purses also have a `decr` method that subtracts an amount for the purses' balance.

The key security policy enforced by the `mint` and its associated `purse` objects is that only the `mint` can change the total amount of money: that is, at any point in time, the sum of all of

```
(def (makeMint)
  (let-values ([(seal unseal) (CreateSeal)])
    (def mint
      (to (makePurse balance)
         (def (decr amount)
            (set! balance (- balance amount)))
         (def purse
           (to (getBalance) balance)
           (to (deposit amount src)
              ((unseal (send src getDecr)) amount)
              (set! balance (+ balance amount)))
           (to (getDecr amount)
              (seal decr)))
         purse))
    mint))
```

Figure 22: A secure mint using sealers and unsealers.

the purse balances is exactly the sum of the initial balances of all the purses belonging to the mint. The implementation in Figure 21 does not satisfy this policy, since access to a purse allows money to be created or destroyed by invoking the purse's decr method.

Satisfying this policy presents a conundrum: the implementation of deposit method needs access to the decr method, but it must be protected against any other use. Sealers and unsealers solve this problem by allowing purse objects to authenticate that the objects they communicate with are purses belonging to the same mint. The revised program is given in Figure 22. In this program, each mint object closes over a unique seal and unseal pair. These procedures will be used to share values between purses belonging to the mint without exposing them to untrusted code. To achieve this, the decr method is moved out of the purse object's public interface and replaced with a new method getDecr. The getDecr method returns a reference to the decr procedure, but first seals it with the mint's `seal`. To any code that doesn't belong to the same mint, the resulting value is useless and can't be used to modify the balance of the purse. On the other hand, other purses from the same mint have access to the `unseal` procedure and thus are able to unseal the procedure in the body of the `deposit` method, successfully implementing the original protocol.

This pattern is powerful, but comes at a high cost—retrofitting the mint to enforce the desired security policy required significant modifications to the structure of the original program and complicated reasoning to justify both that it is secure and that it preserves the desired functionality. In contrast, contracts make it possible to preserve the clean implementation of the mint program while enforcing this property. Before demonstrating this, we need to introduce a new type of contracts: *bounded polymorphic contracts*.

Bounded polymorphic contracts generalize the parametric-polymorphic contracts that have been previosly proposed [44], and are inspired by bounded parametric-polymorphic types and bounded existential types [17]. Before describing bounded polymorphic contracts for objects, we present the corresponding contracts for functions. A bounded polymorphic contract quantifier is either universally quantified (as in `(∀ X <: ctc-bound)`) or existentially quanitified (as in `(∃ X <: ctc-bound)`). The form

```
(bounded-> (quantifier ...) ctc-arg ... ctc-result)
```

creates a procedure contract `(-> ctc-arg ... ctc-result)`, where the contracts appearing in quantifier bounds, arguments, or the result can refer to quantifier variables `X`. Each time the contract is attached to a function, a fresh contract is created for each quantifier variable `X`.

Values flowing into the polymorphic function through a universal quanitified contract variable (i.e., values protected by some univerisally quantified `X` in negative position with respect to `bounded->`) are wrapped with the corresponding bound contract. Values flowing out of the polymorphic function through a universally quantified contract variable (i.e., values protected by some universally quantified X in positive position with respect to bounded->) are checked to ensure they were wrapped by the corresponding X contract in a negative position. If so, the bound contract is removed from the value; if not, a contract violation is signaled.

Like bounded universally quantified types, bounded universally quantified contracts are useful because they provide a mechanism to grant a procedure a limited interface by which to interact with its arguments, but allow the caller to recover the full interface when the procedure returns. For example, the following definition creates a function example that may use its first argument f only according to the contract (-> integer? integer?):

```
(def/ctc (example f x)
  (bounded-> ((∀ X <: (-> integer? integer?))) X any/c X)
  (f x)
  f)
```

Invoking example with an integer works as expected:

```
> (example (λ (x) x) 0)
#<procedure>
```

Supplying a non-integer value causes example to invoke f with an non-integer argument, violating the bound attached to the contract variable X:

```
> (example (λ (x) x) #f)
example: broke its own contract
  promised: integer?
  produced: #f
  in: the 1st argument of
      the 1st argument of
      (-> X any/c X)
  contract from: (function example)
  blaming: (function example)
   (assuming the contract is correct)
  at: eval:1.0
```

However, the restriction is lifted when f is returned through another instance of the X contract, allowing the following to succeed:

```
> ((example (λ (x) x) 0) #f)
#f
```

Existentially quantified contract variables have the opposite effect. Values flowing out of the polymorphic function through an existentially quanitified contract variable (i.e., values protected by some existentially quantified `X` in positive position with respect to `bounded->`) are wrapped with the corresponding bound contract. Values flowing into the polymorphic function through an existentially quantified contract variable (i.e., values protected by some existentially quantified `X` in negative position with respect to `bounded->`) are checked to ensure they were wrapped by the corresponding `X` contract in a positive position. If so, the bound contract is removed from the value; if not, a contract violation is signaled.

Existential types are useful because they allow programs to define abstract data types that can only be manipulated via a prescribed interface. Bounded existential types relax this, and allow programs to define partially abstract data types, where some operations are publically available and some are hidden. Bounded existentially quantified contracts afford a similar idea. Consider this example program which returns two values:

```
(def/ctc (example)
  (bounded-> ((∃ X <: (-> integer? integer?)))
    (values X (-> X any/c any/c)))
  (values (λ (x) x) (λ (f x) (f x))))
```

The first value is the identity function with the bound contract `(-> integer? integer?)` and the second is function that consumes a function with the existentially quantified contract along with a second value and applies the function to the value. We know from the contract bound that it is safe to call the first value with an integer:

```
> (let-values ([(f apply) (example)])
    (f 0))
0
```

However, despite the fact that this function is the identify function, the bound prevents us from invoking it with non-integer arguments:

```
> (let-values ([(f apply) (example)])
    (f #f))
example: contract violation
  expected: integer?
  given: #f
  in: the 1st argument of
      the range of
      (-> (values X (-> X any/c any/c)))
  contract from: (function example)
  blaming: program
   (assuming the contract is correct)
  at: eval:1.0
```

The second function returned by `example` has the contract `(-> X any/c any/c)`, and thus can access the underlying value ignoring the bound:

```
> (let-values ([(f apply) (example)])
    (apply f #f))
#f
```

Moreover, invoking the second function requires that its first argument witnesses the existential parameter `X`, and so it cannot be called with any value other than the corresponding return value of `example`:

```
> (let-values ([(f apply) (example)])
    (apply (λ (x) x) #f))
example: contract violation
  not X: #<procedure>
  in: the 1st argument of
      the range of
      (-> (values X (-> X any/c any/c)))
  contract from: (function example)
  blaming: program
   (assuming the contract is correct)
  at: eval:1.0
```

```
(def (makeMint)
  (def/ctc mint
    (obj/c
      (∃ P <: (obj/c
                [getBalance (-> amount?)]
                [deposit    (-> amount? P void)]))
      [makePurse (-> amount? P)])
    (to (makePurse balance)
        (def purse
          (to (getBalance) balance)
          (to (deposit amount src)
              (send src deduct amount)
              (set! balance (+ balance amount)))
          (to (deduct amount)
              (set! balance (- balance amount))))
        purse))
  mint)
```

Figure 23: A secure mint using contracts.

In fact, using this property we can provide a succinct implementation of the `CreateSeal` operator:

```
(def/ctc (CreateSeal)
  (bounded-> ((∃ X <: opaque/c))
    (values (-> any/c X) (-> X any/c)))
  (values (λ (x) x) (λ (x) x)))
```

where `opaque/c` is a contract that hides all of the functionality of its contracted value.

We extend object contracts with an additional list of quantifiers to yield bounded polymorphic object contracts. The new syntax is:

```
(obj/c quantifier ...
  [method contract] ...
  [recv fun])
```

As with bounded polymorphic function contracts, the variables declared in `quantifier ...` may appear as contracts anywhere in the bounds or method contracts that follow.

Figure 23 shows a new version of the mint program using a bounded polymorphic contract. The use of contracts is able to clearly separate the functionality of mints and purses from the code that preserves the integrity of mints. Each `mint` object has a unique existentially quantified contract `P` which represents purses belonging to that mint. Purses created by the mint's `makePurse` method are each wrapped with this contract. The bound on `P` enforces two key properties: first, that code without direct access to the underlying purse value can invoke only the purse's `get-Balance` and `deposit` methods, and second, that deposit can be invoked only with another purse from the same mint. Together, these properties guarantee the desired security property.

3.3 Related Work

Object-capability design patterns In addition to the design patterns we have presented in this chapter, object-capability design patterns have designed to enforce a wide range of security properties. Murray and Grove [84] show how to create non-delegatable authorities, which separate the right to invoke a capability from the right to grant the capability to another component. Dimoulas et al. [24] implement contracts for restricting the delegation of capabilities that enforce a similar property. Miller et al. [76] give a design pattern for associating objects with users that allows programs to implement traditional access control mechanisms based on access control lists. We conjencture that contracts implementing a similar pattern could be implemented in the style of option contracts [22].

Correctness of capability-based security Preventing security abstractions from leaking sensitive capabilities is a recognized challenge for capability-based security. In early capability-based operating systems [56], the confinement problem [63] led to the combination of capabilities and access control policies. The ICAP system [42] uses access control policies on capabilities to limit their propagation in distributed systems. Dimoulas et al. [24] use contracts to enforce similar policies in a capability-safe language.

Maffeis et al. [70] show that capability-safe languages such as Caja are suitable for enforcing isolation properties as long as components do not share capabilities. Others have studied how to verify the security of capability-based abstractions where components must communicate. For instance, Politz et al. [91] use a type system to verify the confinement guarantees provided by ADsafe. Murray et al. [83], Speissens [106], and Drossopoulou et al. [28] apply formal methods to verify the security of specific object-capability design patterns.

4
A Secure Shell Scripting Language

Up to this point, we have demonstrated the advantages of contracts for capabilities only through a series of small examples. In this chapter, we explore this connection further through the design and implementation of a secure shell scripting language.*

A scripting language is an ideal test of an approach to developing secure software, since users of commodity operating systems often need to execute untrustworthy software. In fact, this is the common case: due to errors or malicious intent, software regularly does not behave as expected. The Principle of Least Privilege (POLP) [98] requires that software should be given only the authority it needs to accomplish its functionality. If adhered to, this principle (also known as the

*This chapter previously appeared in the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation [81].

Principle of Least Authority) can help protect systems from erroneous or malicious software.

However, commodity systems and their secure tools fail to adequately support POLP. First, it is difficult for the user of a commodity system to determine what authority a given piece of software requires to execute correctly. Second, current mechanisms for limiting authority are difficult to use: they are either coarse-grained or require significant changes to existing software, and are often not available to all users [60]. For both of these reasons, users tend to execute software with more authority than is necessary.

For example, consider scripts to grade homework submissions in a computer science course. Students submit source code, and a script `grade.sh` is run on each submission to compile it and run it against a test suite. The submission server must execute `grade.sh` with sufficient authority to accomplish its task, but should also restrict its authority to protect the server from student-submitted code and ensure the integrity of grading. At a coarse grain, the server should allow `grade.sh` to access files and directories necessary to compile, run, and record the scores of homework submissions, and deny access to other files or resources. This ensures, for example, that a careless student's code won't corrupt the server and a cheating student's code won't modify or leak the test suite. At a fine grain, each call to `grade.sh` to grade a single submission should be isolated from the grading of other submissions. This ensures, for example, that a cheating student cannot copy solutions from another submission.

Securing a script such as `grade.sh` is difficult, as it requires balancing functional and security requirements. To begin with, it is a priori unclear what authority `grade.sh` needs to execute correctly. While the author of the script may know, the user must examine the code to try to determine what authority it requires. If the user can identify the required resources, she can use existing tools for sandboxing program execution (e.g., [52, 58, 67]) to achieve the coarse-grained security requirements. However, it is difficult to use the same tools to enforce the fine-grained security requirements described above. This is because achieving these requirements requires that each invocation of `grade.sh` is given different privileges, i.e., it must be executed in a differently configured sandbox. Configuring all of these sandboxes correctly is error prone, so users often forgo fine-grained security and violate POLP.

To address these issues, we introduce the SHILL programming language. SHILL is a secure shell scripting language with features that help apply POLP in commodity operating systems.[†] At the core of SHILL are declarative security policies that describe and limit the effects of script execution, including effects of arbitrary programs invoked by the script.

These declarative security policies can be used by producers of software to provide fine-grained descriptions of the authority the software needs to execute. This, in turn, allows consumers of software to inspect the software's required authority, and make an informed decision to execute the software, reject the software, or apply a more restrictive policy on the software.} The SHILL runtime system ensures that script execution adheres to the declared security policy, providing a simple mechanism to restrict the authority of software.

Two key features enable SHILL's declarative security policies: language-level capabilities for system resources and contracts. SHILL scripts access system resources only through capabilities: unforgeable tokens that confer privileges on resources. SHILL scripts receive capabilities only from the script invoker; SHILL scripts cannot store or arbitrarily create capabilities. Moreover, SHILL uses *capability-based sandboxes* to control the execution of arbitrary software. Thus, the capabilities that a user passes to a SHILL script limit the script's authority, including any programs it invokes. SHILL's contracts specify what capabilities a script requires and how it intends to use them. SHILL's runtime and sandboxes enforce these contracts, hence they serve as fine-grained, expressive, declarative security policies that bound the effects of a script.

For example, Figure 24 shows a SHILL contract for a script to grade a single student submission

[†]SHILL is not an interactive shell, but rather a language that presents operating system abstractions to the programmer and is used primarily to launch programs. Other languages currently used for this purpose include Perl, Python, and the scripting portion of Bash.

```
(provide
  [grade (-> [submission (and/c file? readonly/c)]
             [tests      (and/c dir?  readonly/c)]
             [working    (dir/c (+create-dir +all))]
             [grade-log  (and/c file? writeable/c)]
             [wallet     ocaml-wallet/c]
             void)])
```

Figure 24: SHILL contract for a grading script.

(corresponding to the `grade.sh` script described above). It is a declarative security specification for the function `grade`, which takes 5 arguments: a read-only file `submission` (i.e., the student's source code), a read-only directory `tests` (containing the test suite), a "working directory" in which the script may create subdirectories with full privileges, a writeable file `grade-log` for recording the student's grade, and a "wallet" that provides sufficient capabilities to invoke the OCaml compiler. This contract serves two purposes: it clearly describes what `grade` needs to execute correctly and it also provides guarantees about what `grade` may do when invoked. Given this contract, a user can be confident that `grade` satisfies the security requirements described above, even though `grade` will compile and execute student-submitted code. Specifically: `grade` will not read any other student's submission; `grade` will not communicate over the network (as it has no capability for network access); `grade` will not corrupt the test suite nor write any files other than the grade log and subdirectories it creates within the working directory. The implementation of `grade` (not shown) focuses solely on the functionality for grading, and is not concerned with enforcing security requirements.

SHILL offers language abstractions for reasoning about the authority of pieces of software and their composition. Specifically, SHILL (1) introduces a capability-based scripting language with language abstractions (such as contracts and wallets) to use capabilities effectively, and (2) implements, on a commodity operating system, capability-based sandboxes that extend the guarantees of the scripting language to binary executables and legacy applications. These language abstractions, and the enforcement of these abstractions, make it possible to manage authority and follow POLP, even when using and combining untrusted programs.

The rest of this chapter is structured as follows. In Section 4.1 we present the design of SHILL. Our implementation of SHILL in FreeBSD 9.2 is described in Section 4.2. We evaluate SHILL by using it to implement several case studies, and measure the overhead of SHILL's security mechanisms. We present the evaluation results in Section 4.3. Section 4.4 describes related work.

4.1 Design

SHILL aims to meet the following five goals:

1. Script users can control the authority of a script, i.e., what system resources it can access or modify.

2. Script users can understand what authority a script needs in order to accomplish its functionality.

3. Security guarantees of scripts apply transitively to other programs the script may invoke, including arbitrary executables.

4. SHILL separates the security aspects of scripts from functional aspects, reducing the impact of security concerns on the effort required to write scripts.

5. SHILL is compatible with commodity operating system abstractions.

To meet these goals, SHILL uses a combination of language design and mandatory access control-based sandboxing.

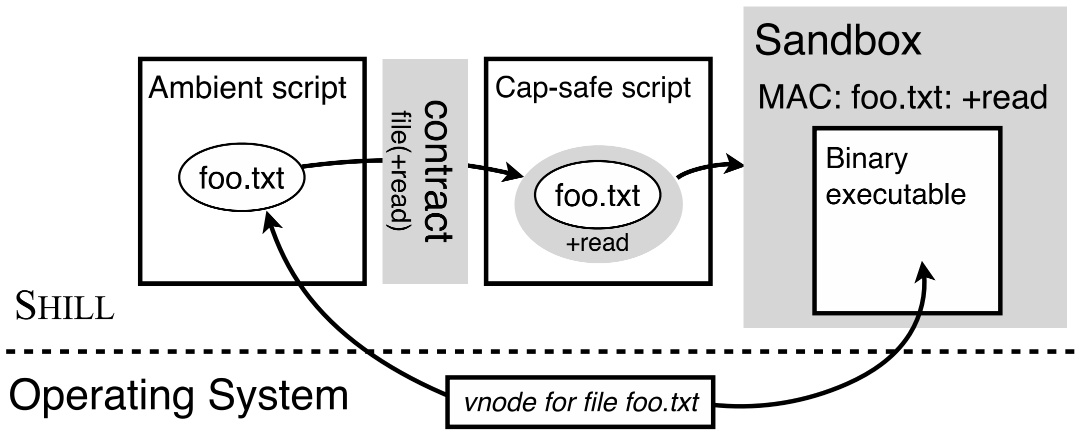

Figure 25: SHILL in a nutshell.

In most scripting languages, scripts can access a resource (such as a file) using the resource's well-known global name. Access control is based on the user on whose behalf the script executes. Thus, a script's authority is *ambient* (i.e., it derives from the script's execution context) [79], and a script may access any and all resources that the invoking user may access. SHILL's security is based on capabilities instead of ambient authority.

There are two kinds of SHILL scripts: *capability-safe SHILL scripts*, and *ambient SHILL scripts*. Capability-safe SHILL scripts play the same role as regular shell scripts, but do not have ambient authority and must be given capabilities to access resources. Ambient SHILL scripts are used to create the initial set of capabilities to give to capability-safe scripts. They do have ambient authority, but are very restricted: ambient scripts can only create capabilities for system resources and invoke capability-safe SHILL scripts.

Each capability-safe SHILL script comes with a contract that is enforced by the language runtime. A capability-safe SHILL script can use the capabilities it possesses to access resources using SHILL's built-in functions, if allowed by the contract. SHILL scripts can also invoke arbitrary executables in *capability-based sandboxes*. A capability-based sandbox is created with a set of capabilities, and enforces a mandatory access control policy that restricts the executable's behavior based on those capabilities and their contracts.

Figure 25 depicts the life cycle of a capability for a file named `foo.txt`. First, an ambient script acquires a capability for the file from the operating system using the user's ambient authority. This capability is then passed to a capability-safe script via a contract, which restricts the privileges on the capability to `+read` (i.e., the capability can be used only to read `foo.txt`, not to write to it, etc.). The capability-safe script then runs an executable in a sandbox, granting it the capability to read the file.

THREAT MODEL In SHILL's threat model, some capability-safe scripts (and the executables they invoke) are not trusted. However, their behavior is restricted by their contracts and the capabilities they are given: a capability-safe script (and any executables it invokes) can access resources only as permitted by its contract and the capabilities it possesses. Of course, the contract that accompanies a script may also be untrustworthy: a user should inspect the contract and understand its security implications before passing capabilities to the script. The benefit of SHILL's approach is that it is much easier to inspect and understand the declarative contract than to examine the script itself.

SHILL's trusted computing base includes the operating system kernel and SHILL runtime. SHILL does not explicitly defend against malicious scripts or executables that exploit security flaws in the kernel or SHILL itself.

The rest of this section describes how SHILL's design and features contribute towards these goals, and provides an introduction to SHILL via several small examples.

4.1.1 Controlling script authority

Ambient authority makes writing scripts easy: if a script needs to access a resource, it can simply use the resource's name to access it. However, ambient authority makes it difficult to understand and control the potential effect of executing a script. First, the authority of a script is not easily deducible from its code, a problem that is exacerbated when the script invokes other scripts or executables. Second, commodity operating systems do not provide easy mechanisms to limit authority of an execution context, for example, by allowing a user to temporarily restrict permissions in a fine-grained way.

Authority in SHILL is controlled by capabilities. In order to access a resource, a SHILL script must have a capability for that resource. SHILL scripts can only acquire capabilities as arguments provided by the user, or by deriving them from other capabilities (e.g., using a directory capability to acquire a capability for a file in the directory). These restrictions, which correspond to capability safety, lie at the heart of the security of SHILL scripts. Capability safety makes it possible for users to control the authority of SHILL scripts they invoke (Goal 1).

Figure 26 presents a snippet of SHILL code that demonstrates how SHILL scripts use capabilities. It defines a function `find-jpg` for recursively finding all the files with extension `.jpg` within a given directory. Argument `cur` is a capability for either a file or a directory. In contrast with standard scripting languages, `cur` is not a string that names a file, but is a capability that denotes it, much like a file descriptor. If `cur` is a file capability and the name of the file ends with `.jpg`, then the script uses the built-in function `path` to get the string for the path to the file,[‡] and appends it to the pipe or file capability `out` (lines 5–6).

If `cur` is a directory capability, then the built-in function `contents` is used to get the list of names of children of `cur`. For each child, the script calls `(lookup cur name)` to obtain a

[‡]The library function `has-ext?` also uses `path`.

```
(define (find-jpg cur out)
  (cond
    ; if cur is a file with extension jpg,
    ; output its path to out.
    [(and (file? cur) (has-ext? cur "jpg"))
     (append out (path cur))]
    ; if cur is a directory, recur on its contents
    [(dir? cur)
     (for ([name (contents cur)])
       (let ([child (lookup cur name)])
         (unless (error? child)
           (find-jpg child out))))]))
```

Figure 26: SHILL script snippet to find `.jpg` files.

capability for the child (line 10), which is then used in a recursive call to `find-jpg` (line 12).

Conceptually, SHILL capabilities correspond to operating system representations of resources, such as file descriptors, and built-in functions such as `append` and `lookup` are wrappers for the corresponding system calls.

SHILL enforces capability safety by restricting the expressiveness of the scripting language. While SHILL offers full-fledged language features and rich libraries, comparable to other scripting languages, the built-in functions for using resources require capabilities as arguments. In addition, SHILL does not have mutable variables and capabilities are not serializable. This means that SHILL scripts cannot store or share capabilities through memory, the filesystem, or the network. For controlled sharing of capabilities, SHILL provides *wallets*, capabilities for packaging and managing collections of capabilities. We discuss wallets further in Section 4.1.4.

SHILL scripts provide the same protection from confused deputy attacks [46] as traditional capability systems. Furthermore, filesystem operations that produce new capabilities (such as lookup) do not allow scripts to arbitrarily traverse the filesystem. For instance, a script cannot use the capability for the current directory cur and (lookup cur "..") to obtain the parent directory of cur.

4.1.2 Contracts

Capability safety makes it possible to limit the authority granted to a SHILL script by carefully selecting what capabilities to pass as arguments. Unfortunately, needing to pass capabilities explicitly makes it harder for script users to deduce how to use scripts and compose them to complete more complicated tasks. At its core, this is a problem of defining the script's interface: how does the script communicate what resources it requires and how it will use those resources?[§]

SHILL addresses these issues by providing expressive, fine-grained and enforceable interfaces for scripts (Goal 2) following the *Design by Contract* paradigm [71, 75]. Every function that a SHILL script exports (i.e., makes available to users of the script) is accompanied by a *contract* that describes the arguments the function expects and the result it returns. For example, the following snippet is a contract for the find-jpg function from Figure 26:

```
(provide [find-jpg (-> [cur (or/c dir? file?)] [out file?] void)])
```

The provide keyword indicates that the function find-jpg is exported. The contract for the function is (-> [cur (or/c dir? file?)] [out file?] void). Each function contract has two parts: the precondition and the postcondition. The precondition of our example states that find-jpg takes two arguments: a capability cur that is either a directory or a file capability, and a file capability out. Following Unix convention, file capabilities include capabilities for files, pipes, and devices. The postcondition void means that no value is returned.

[§]Traditional shell scripting languages such as Bash or Python also suffer from these issues, but the use of ambient authority masks them: scripts typically receive much more authority than needed.

The precondition of the contract above describes what kind of capabilities `find-jpg` needs, but does not indicate how the function intends to use these capabilities. SHILL allows us to give a more precise contract for `find-jpg`:

```
(provide
  [find-jpg (-> [cur (or/c (dir/c +contents +lookup +path)
                           (file/c +path))]
                [out (file/c +append)]
                void)])
```

This version specifies not only what kind of capabilities the function consumes but also what privileges it requires on these capabilities. Each privilege, such as `+path` or `+contents`, corresponds to an operation on a capability. A capability contract with a set of privileges restricts what operations that capability can be used for.

Some operations on capabilities, such as `lookup`, produce more capabilities. Capability contracts can specify the privileges a script should have on these derived capabilities. For example, privilege `(+lookup +path +stat)` indicates that any capabilities derived using the `lookup` operation should only have the `+path` and `+stat` privileges. When a privilege confers the right to derive new capabilities but does not come with a modifier (such as the `+lookup` privilege in the contract for `find-jpg`), the derived capability has the same privileges as its parent capability.

Each contract establishes an agreement between two parties: the provider of the value with the contract and the value's consumer. As part of the agreement, each party promises to live up to its contractual obligations. In this way, a contract both describes a guarantee one party provides and a requirement the other party demands. For function contracts, the consumer's obligations are to supply function arguments that satisfy the precondition, and the provider must produce a result that satisfies the postcondition. For capability contracts, the provider agrees to provide a capability of the appropriate kind with *at least* the specified privileges while the consumer promises to use the capability as if it has *at most* the specified privileges. For example, according to the `find-jpg` contract, users of `find-jpg` must supply a file capability that permits the `append` operation for the `out` argument, while `find-jpg` itself promises not to call other operations on the capability, such as `read`.

The SHILL runtime checks whether parties live up to their obligations by monitoring execution and checking that values are used in accordance with their contracts. For example, when `find-jpg` is called with a capability for a directory and a capability for the output file, the body of `find-jpg` does not receive the capabilities themselves. Instead, each contract wraps the underlying capability with a *proxy*. These proxies enforce the contracts for `cur` and `out` by intercepting calls to operations on the capabilities and allowing them only if permitted by the contract. If the body of `find-jpg` attempts to perform an operation that isn't permitted—such as reading the contents of `out` or unlinking `cur`—the proxy will indicate that a contract violation has occurred. If a contract is violated, the SHILL runtime aborts execution and, to help with auditing and debugging, indicates which part of the script failed to meet its obligations.

4.1.3 Securing arbitrary executables

SHILL security guarantees must be completely enforced: even if a script calls other scripts or runs arbitrary executables, its authority should be restricted to its capabilities, and it should meet its contract obligations (Goal 3). When SHILL scripts invoke only other SHILL scripts, we achieve SHILL's security guarantees easily because of the language's semantics. However, scripts also invoke executable programs.

To ensure that these programs cannot violate SHILL's security guarantees, SHILL scripts may only invoke executables inside a *capability-based sandbox*. When a sandbox is created, it is given a set of capabilities. The SHILL sandbox limits the authority of the sandboxed executable to the authority implied by the set of capabilities.

Scripts can invoke an executable in a sandbox by calling the built-in function `exec`. For example, the following snippet executes the file `jpeginfo` in a sandbox with the arguments `"-i"` and a given file:

```
(exec jpeginfo (list "jpeginfo" "-i" file)
  #:stdout out #:extras (list libc libjpeg))
```

The `exec` function has two required arguments. The first is a file capability with the `+exec` privilege. The second is a list of string arguments to provide to the executable. SHILL programmers can also provide as arguments to executables capabilities for files or directories instead of string representations of their paths. In this case, the path to the given file is passed to the executable as an argument. The `exec` function also takes some optional arguments, including capabilities to use for standard input, output, or error (`#:stdout out`), and extra capabilities needed by the program (`#:extras (list libc libjpeg)`). This set of extra capabilities is often quite large. In Section 4.1.4, we describe abstractions to help manage capabilities for sandboxes.

SHILL sandboxes enforce a capability-based mandatory access control (MAC) policy on the sandboxed execution. For example, the sandbox for `jpeginfo` allows access only to resources indicated by capabilities passed as arguments to `exec` (which, for the `jpeginfo` example above, are the `jpeginfo`, `file`, `out`, `libc`, and `libjpeg` files and directories). Moreover, if any of these capabilities comes with a contract, the MAC policy further limits access to the resource according to the capability's contract.

This capability-based MAC policy is enforced *in addition* to the operating system's discretionary access control (DAC) policies: an operation on a resource by a sandboxed execution is permitted only if it passes the checks performed by the operating system based on the user's ambient authority and is also permitted by the capabilities possessed by the sandbox. Note that sandboxed executables never possess capabilities that allow them to use ambient authority to circumvent the MAC policy. For example, no sandboxed executable has a capability to unload kernel modules, including the module that enforces the MAC policy. Section 4.2.2 describes how we implement capability-based sandboxes using the TrustedBSD MAC framework.

4.1.4 Writing shill scripts

shill's security benefits come at the cost of extra effort to write scripts. Nonetheless, we strive to make it easy to write shill scripts while obtaining stronger security guarantees than traditional shell scripting languages. To make it easier to write scripts, shill offers security abstractions such as *capability wallets* and pushes security concerns to the interfaces between scripts.

Security abstractions

shill requires that any access of a protected resource requires an appropriate capability. However, even simple executable programs require access to a surprising number of files. For example, executing `cat` in a sandbox requires providing eight capabilities to libraries and configuration files in addition to capabilities for the executable itself and the input and output.

Consider a shill script that executes `cat` in a sandbox. One can imagine a contract that requires a separate argument for each of the eight capabilities that `cat` requires. While precise, such a contract imposes a significant burden on both the script writer (since the need for these capabilities will be exposed in the interface for the script) and the script user (who will need to supply these capabilities individually).

Another possibility is a contract that takes important capabilities separately (e.g., for the executable and the input and output) and takes all other capabilities in a list. Although succinct, this contract burdens the script's user, who has no idea what capabilities should be in this list.

We introduce *capability wallets* as a mechanism to automate and simplify the discovery, packaging, and management of capabilities that sandboxes need to run executables. Conceptually, a capability wallet is a map from strings to lists of capabilities. To reduce the burden on script

```
(provide
  [jpeginfo (-> [wallet native-wallet?]
                [out    (file/c +write +append)]
                [arg    (file/c +read +path)]
                void)])
(define (jpeginfo wallet out arg)
  (let ([jpeg-wrapper (pkg-native "jpeginfo" wallet)])
    (jpeg-wrapper (list "-i" arg) #:stdout out)))
```

Figure 27: Executing `jpeginfo` in a sandbox using wallets.

writers, SHILL provides *wallet contracts*, which describe contracts for the capabilities associated with individual keys or groups of keys. To reduce the burden on script users, SHILL provides library functions to automate the collection and packaging of capabilities into wallets.

Figure 27 shows a script that uses a capability wallet to create a sandbox for the program `jpeginfo`. The first argument to the `jpeginfo` function has the contract `native-wallet?` (line 2). A `native-wallet` is a particular kind of capability wallet that can be built using functions from SHILL's standard library. It collects together the capabilities needed to invoke executables and can be used with other functions from the SHILL standard library that present a familiar path-based interface for identifying and running executables. The capabilities in a wallet are derived from capabilities the user explicitly grants to the script. Thus despite its path-based interface, a native wallet is still capability safe.

This script uses one of the standard library functions, `pkg-native`, to create a wrapper containing all of the capabilities needed to run the `jpeginfo` executable in a sandbox (line 7). The script then calls the wrapper, supplying the executable arguments and input and output capabilities (line 8).

SHILL's standard library comes with a rich collection of functions that construct and manipulate wallets, wallet contracts and wallet-derived sandboxes. Section 4.2.1.4 presents these utilities in further detail.

Pushing security to interfaces

SHILL's contracts allow the programmer to separate the security specification of a script from the implementation of its functionality (Goal 4). The SHILL runtime ensures that contracts are enforced, removing the need for defensive code that checks and protects the use of capabilities. Consider the `find-jpg` function from Figure 26: the implementation is simple, and the security guarantee is provided by its contract. This separation makes it possible to strengthen or relax a script's security guarantees by modifying its contract. Indeed, in Section 4.1.2 we saw two different contracts for the `find-jpg` function, one of which provides a more precise security guarantee.

SHILL's contract system is rich and expressive, allowing precise specifications of security guarantees. For example, users can define their own contracts by creating contract combinators and user-defined predicates written in SHILL itself.

SHILL's contracts can also be used to write security specifications that provide different guarantees to different script users. Consider the script in Figure 28. This script recursively finds files and performs an action on these files. (It is more general than the `find-jpg` script of Figure 26.) The function `find` takes three arguments: a file or directory capability `cur`, a function `filter` that is used to select files, and a function `cmd` to apply to all selected files. Lines 6–14 implement `find`'s functionality. Note that this code is straightforward, and does not directly address security concerns.

Lines 1–4 define the contract for `find`, using a bounded polymorphic contract. The polymorphic contract declares that for any contract `X`, the function `find` can be called with arguments `cur`, `filter`, and `cmd` such that `cur` satisfies contract `X`, `filter` satisfies contract

```
(provide
  [find (bounded-> ([∀ X <: (cap/c +lookup +contents)])
          [cur X] [filter (-> X boolean?)] [cmd (-> X void)]
          void)])

(define (find cur filter cmd)
  (cond
   [(and (file? cur) (filter cur))
    (cmd cur)]
   [(dir? cur)
    (for ([name (contents cur)])
      (let ([child (lookup cur name)])
        (unless (error? child)
          (find child filter cmd))))]))
```

Figure 28: A find script with a polymorphic contract.

(-> X boolean?) (i.e., filter is a function that expects a value that satisfies X and returns a boolean), and cmd satisfies contract (-> X void) (i.e., cmd is a function that expects a value that satisfies X and returns no value).

The polymorphic contract is *bounded* because the contract X on capability cur that the caller provides must have at least the privileges +lookup and +contents. Moreover, the contract requires that find can use only the +lookup and +contents privileges of the cur argument or derived capabilities, even though contract X may specify more privileges. Importantly, the contracts for arguments filter and cmd allow these functions to use all of the privileges that X specifies. In essence, the contract of find dynamically seals [82] the argument cur as it flows into the body of the function through contract X, and unseals it as it flows out to the functions filter and cmd.

The contract on find allows clients to use find in different ways. For example, one client may use it with a `filter` that examines file creation times (which requires the `+stat` privilege). Another client may use `find` with a `filter` that inspects a file's name (which requires `+path`, but not `+stat`). For both clients, the contract guarantees that the implementation of `find` itself cannot use either the `+stat` or `+path` privileges, even though it invokes the functions `filter` and `cmd`.

4.1.5 Interaction with ambient authority

Figure 26, Figure 27, and Figure 28 show SHILL scripts that consume and use capabilities. But where do capabilities come from? SHILL is intended for use with commodity operating systems, and so we must provide a mechanism to transition from the ambient world of the operating system to SHILL's capability-safe world (Goal 5).

To that end, in addition to the capability-safe scripts we have described so far, users of SHILL scripts write *ambient scripts* which inherit the authority of the invoking user and are *not* capability safe. Ambient scripts are used to create capabilities and pass them to functions that capability-safe scripts provide. Consequently, the language of ambient scripts is extremely restricted: ambient scripts contain straight line code that can import capability-safe scripts, create capabilities for resources using file paths and other global names, and call functions exported by capability-safe scripts. Ambient scripts are brief and delegate all interesting tasks to the capability-safe scripts they import. Also, capability-safe scripts cannot import ambient scripts, which ensures that capability-safe scripts cannot use ambient scripts to obtain additional capabilities. Ambient scripts must reason carefully about their interaction with untrusted scripts. Contracts and capabilities help with this.

Figure 29 shows an ambient script that creates appropriate capabilities and then invokes the `jpeginfo` function from the script in Figure 27. The annotation `#lang shill/ambient` on

```
#lang shill/ambient

  (require shill/native)
  (require "jpeginfo.cap")

  (define root   (open-dir "/"))
  (define wallet (create-wallet))
  (populate-native-wallet wallet root
     "~/Downloads/jpeginfo"
     "/lib:/usr/local/lib"
     pipe-factory)

  (define dog    (open-file "~/Documents/dog.jpg"))
  (jpeginfo wallet stdout dog)
```

Figure 29: Ambient script to call `jpeginfo`.

line 1 indicates that this is an ambient script.[5] Line 3 loads a SHILL library script that helps create capability wallets. Line 4 loads the capability-safe script from Figure 27.

Lines 8–11 create an appropriate capability wallet to run `jpeginfo` by calling the trusted standard library function `populate-native-wallet`. Line 13 creates a capability for the file `~/Documents/dog.jpg`. The capability has all privileges that the invoking user is allowed for this file; when the capability passes through a capability contract, it loses all privileges except those stated in the contract. Line 14 invokes `jpeginfo` with the capability wallet, a capability to standard out, and the capability to `dog.jpg`.

[5] Capability-safe scripts have the annotation `#lang shill/cap` on the first line; we omitted this annotation in Figure 26, Figure 27, and Figure 28.

4.2 Implementation

We have implemented a prototype of SHILL as a kernel module and set of userspace tools for FreeBSD 9.2. The userspace tools include the SHILL compiler, runtime, and standard library. The kernel module implements capability-based sandboxes and provides capability-safe versions of several POSIX system calls.

4.2.1 Language

We implement the SHILL language as an extension to Racket [38] using Racket's macro system and tools for building languages [119]. Prototyping SHILL in this way allows us to use Racket functionality where it meets our security requirements. In particular, we used Racket's contract mechanism to implement SHILL contracts.

A distinguishing feature of SHILL is capability safety: access to resources occurs only through capabilities, and creation of capabilities is limited. To achieve capability safety at the language level, we (1) provide language-level capabilities and capability contracts; (2) restrict the expressiveness of the language; and (3) provide a capability-based language runtime for SHILL.

Capabilities and their Contracts

Capabilities in the SHILL language are object-like values that encapsulate low-level capabilities such as file descriptors or sockets. Each operation on a capability is implemented by calling the corresponding operation on the low-level capability. Different kinds of capabilities support different operations. For example, supported operations on files and pipes include reading, writing, and changing modes. Directories also have capabilities for listing, adding, or removing directory entries. Each operation has a corresponding privilege that can be present or absent on a given capability. In total, SHILL has twenty-four different privileges for filesystem capabilities and seven different privileges for sockets. Socket privileges are further refined by connection type.

We chose privileges and operations to align closely with the operations that our capability-based sandbox can interpose on, so that we can ensure that giving a capability to a sandbox conveys the same authority as giving that capability to a SHILL script. There are two kinds of SHILL capabilities that do not encapsulate a system resource directly: the `pipe-factory` and `socket-factory` capabilities. These capabilities encapsulate, respectively, the right to create new pipes or sockets. The `pipe-factory` capability has a `create` operation that returns a pair of pipe ends. Each pipe end is a file capability. In our prototype implementation, SHILL scripts cannot create or manipulate sockets directly (which can be addressed by adding built-in functions for socket operations to the language). We do restrict a sandbox's permitted socket operations: a sandbox must possess a `socket-factory` capability to be allowed to create and use sockets.

We implement SHILL contracts using Racket contract combinators [36, 37] that create proxies [113] for capabilities, allowing us to interpose on operations and check privileges before allowing an operation. These proxies also store information about the privilege restrictions each contract imposes.

Restricting the SHILL language

To achieve capability safety in SHILL, we carefully choose which language features and libraries of Racket are available in SHILL. We allow access to certain Racket libraries, such as the regular expression library, but prevent access to all others, including Racket's system library and Racket's macro system. SHILL scripts are allowed to import only SHILL capability-safe scripts.

The ambient SHILL language (see Section 4.1.5) has further restrictions: it may not do anything other than import capability-safe SHILL scripts, create strings and other base values, define (immutable) variables, and invoke functions. However, unlike the capability-safe SHILL language, it

Resource	**Language**	**Sandbox**
Directories, files, links	Capabilities	Capabilities
Pipes	Capabilities	Capabilities
Character Devices	Capabilities	Capabilities†
Sockets (IP,Unix)	Capabilities	Capabilities
Sockets (other)	Denied	Denied
Processes	ulimit‡	Confinement
Sysctl	Denied	Read-only
Kernel environment	Denied	Denied
Kernel modules	Denied	Denied
POSIX IPC	Denied	Denied
System V IPC	Denied	Denied

Figure 30: System resources and how each is protected in the SHILL language and capability-based sandboxes.
†: In our prototype, character devices are only partially controlled by capabilities, see Section 4.2.2.3.
‡: SHILL allows calls to the `exec` function to specify `ulimit` parameters for the child process.

may create capabilities using ambient authority.

Capability-based runtime

We implemented a capability-based language runtime for SHILL that provides operations to access files and other resources through file descriptors. (The Racket libraries for accessing files and other resources rely on ambient authority, and are thus not suitable for our use.) File descriptors provide unforgeable tokens that can serve as low-level capabilities for directories, files, links, pipes, sockets, and devices. Our capability-based runtime provides wrappers for the `*at` family of system calls which provide a file-descriptor based interface to common operations like opening, reading, and writing files. Our runtime further restricts these system calls by requiring that arguments that specify sub-paths contain only a single component. For example, the path-name argument to `openat` may be `alice` but not `alice/dog.jpg` or `../bob`. Our runtime also provides wrappers for standard system calls which can be used by SHILL's ambient language to create capabilities for system resources.

New system calls Most but not all FreeBSD system calls that manipulate the filesystem have a version that consumes file descriptors rather than paths. The `linkat`, `unlinkat`, and `renameat` system calls use file descriptors to designate target directories, but rely on paths to designate files. Thus, a call to `linkat` can not be guaranteed to link to the correct file without risking a time-of-check-to-time-of-use vulnerability. Our kernel module adds three system calls to address these deficiencies: `flinkat`, which installs a link to a file in a directory given file descriptors for both the file and the directory; `funlinkat`, which takes a name and file descriptors for a file and a directory and removes the link at the given name if it refers to the file; and `frenameat`, which is similar to `funlinkat` but also installs a link to the file in a target directory. The module also provides a version of `mkdirat` that returns a file descriptor for the newly created directory.

We also add a new `path` system call that attempts to retrieve an accessible path for a file descriptor from the filesystem's lookup cache. SHILL uses this system call to provide a relatively robust mechanism to translate SHILL capabilities into paths to provide as arguments to sandboxed executables. If the `path` system call fails, SHILL uses the last known path at which the file was accessible.

Our prototype implementation of SHILL does not provide support for all system resources. Interaction with resources that do not correspond to capabilities is either restricted or denied entirely. Figure 30 lists system resources and how SHILL controls access to these resources in the language and in capability-based sandboxes. There is no fundamental obstacle to providing capability support for all resources, though doing so would require additional modifications to the system call interface. For example, we would need to provide a low-level capability for processes, similar to Capsicum's *process descriptors* [126].

Standard Library

SHILL's standard library provides a number of capability-safe scripts that help programmers write SHILL scripts. The `filesys` script provides capability-based functions that emulate common tasks such as resolving paths and symlinks. The `io` script provides printf-like wrappers around `write` and `append` for formatted output. The `contracts` script provides abbreviated definitions of common contracts. For example, a programmer can specify the contract `readonly/c` rather than the more verbose

```
(or/c (dir/c  +read-symlink +contents +lookup +stat +read +path)
      (file/c +stat +read +path))
```

Capability wallets Recall that capability wallets are maps from strings to lists of capabilities that help automate and simplify the discovery, packaging, and use of capabilities to invoke executables in sandboxes. SHILL provides functions for creating and using capability wallets. For example, the `native` script in the standard library provides two functions for using native wallets to invoke executables (as in Figure 27 and Figure 29): `populate-native-wallet` and `pkg-native`.

Function `populate-native-wallet` helps create a native wallet. Its arguments include path specifications for where to search for executables and libraries (i.e., colon-separated strings, analogous to environment variables `$PATH` and `$LD_LIBRARY_PATH`), and a directory capability to use as a root for the path specifications. In addition, it takes a map (of strings to lists of strings) from known libraries to the file resources those libraries depend on. Function `populate-native-wallet` uses the directory capability to resolve the path specifications (i.e., converts the lists of strings to lists of capabilities), and places these capabilities in a native wallet. It also resolves the known dependencies (i.e., the map from known libraries to the file resource path names) into a map from strings to lists of capabilities, and places the resolved map into the native wallet.

Function `pkg-native` takes a native wallet and a file name (of an executable file) and searches the path capabilities in the native wallet for a capability for the executable. The function then invokes `ldd` to obtain a list of libraries that the executable depends on, and searches the library-path capabilities for capabilities for the required libraries. Once these capabilities are gathered, `pkg-native` uses the map of known dependencies to gather additional capabilities needed to run the executable. Function `pkg-native` then returns a function that encapsulates a call to `exec` with all capabilities needed to run the executable. Figure 27 shows an example script that uses `pkg-native`.

4.2.2 Capability-based sandbox

The SHILL sandbox is implemented as a policy module for the TrustedBSD MAC Framework [127] (hereafter, "the MAC framework"). The MAC framework allows FreeBSD's access control mechanisms to be extended with third-party mandatory access control policies by mediating access to sensitive kernel objects and invoking access control checks specified by third-party policy modules. The framework also provides a policy-agnostic mechanism for attaching security labels to kernel objects. Mechanisms with similar functionality are available on Linux and Apple's OS X.

Session lifecycle

Each process executing in a SHILL sandbox is associated with a *session*. Processes in the same session share the same set of capabilities and can communicate via signals. Processes spawned by a process in a session are by default placed in the same session. However, sessions are hierarchical: a sandboxed process inside session S_1 can spawn a process inside a new session S_2, which has fewer capabilities that S_1. This allows SHILL-aware executables to further attenuate

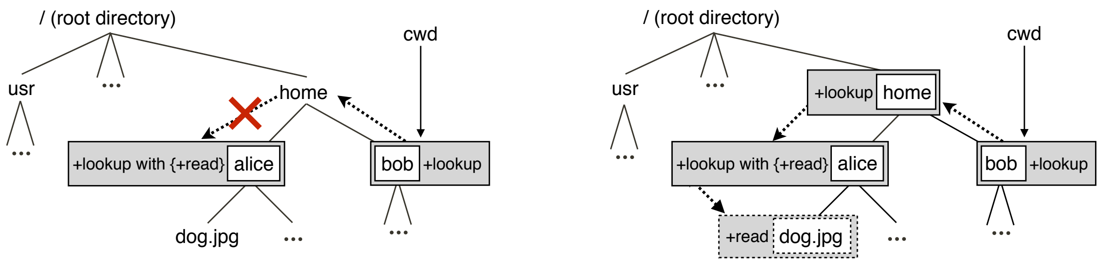

Figure 31: Resolving system call `open("../alice/dog.jpg", O_RDONLY)` in a capability-based sandbox. Left: the session has privileges for `/home/alice` and `/home/bob`, but not `/home`, so the operation fails. Right: the session also has a lookup privilege for `/home`, so the operation succeeds and the lookup privilege on `/home/alice` is propagated to `/home/alice/dog.jpg`.

their privileges.

New sessions are created by invoking the system call `shill_init`, which creates a session and associates it with the current process. A new session initially has no capabilities of its own. Capabilities possessed by the parent session can be granted to the new session until the process invokes the `shill_enter` system call. Once `shill_enter` is called, the session allows only operations permitted by capabilities it was granted explicitly.

From capabilities to MAC labels

Each system resource protected by a SHILL capability corresponds to an underlying kernel object: a filesystem vnode, pipe, device, or socket. Using the MAC framework's ability to attach labels to kernel objects, SHILL labels these kernel objects with a *privilege map*: a map from sessions to sets of privileges. A privilege map records the privileges that each session has for the given kernel object. Privileges in the privilege map correspond directly to privileges of SHILL capabilities.

When a SHILL script calls `exec`, the SHILL runtime sets up a sandbox by forking a new process, creating a new session, and granting the session the capabilities passed to `exec`. It then calls `shill_enter` before transferring control to the executable.

When a sandboxed process invokes a system call relevant to a resource protected by SHILL, we use the privilege map for that resource to check whether the process's session has sufficient privileges for the operation. If there are insufficient privileges, the system call aborts with an error but the process is otherwise allowed to continue.

DERIVED CAPABILITIES In the SHILL language, some operations on SHILL capabilities yield derived capabilities. For example, using a directory capability, a script might obtain capabilities for children of the directory, or might obtain a capability for a new file created in that directory. In the sandbox, we track these derived capabilities by updating privilege maps in response to operations on kernel objects. To enable this, we extended the MAC framework with two additional hooks: `mac_vnode_post_lookup` and `mac_vnode_post_create`. These entry points are invoked after a lookup or create operation completes successfully, and allow the SHILL policy module to update the privilege map on the resulting vnode. For example, if session *S* has privilege `(+lookup +stat +path)` on a vnode for a directory `d`, and a process in that session successfully invokes system call `openat(d, "child", flags)`, then the SHILL policy module updates the privilege map for the vnode for file `child` to add privileges `+stat` and `+path` for session S.

PATH TRAVERSAL To achieve fine-grained confinement in the filesystem, SHILL scripts are not permitted to follow the "`..`" entry of a file or directory capability. However, simply disallowing use of "`..`" in SHILL's capability-based sandboxes would break many existing programs. Instead, the sandbox allows any lookup operation on a directory if the session has the `+lookup` privilege, but only propagates privileges when the lookup would have been permitted in the SHILL language, that is, when the directory entry requested is not “`..`”.[⊠]

Example Consider a sandboxed process attempting to call `open("../alice/dog.jpg", O_RDONLY)` from the current working directory `/home/bob`. This system call invokes a series of low-level `lookup` operations on filesystem objects to resolve the path and create a file descriptor for the designated resource.

Figure 31 depicts the process of completing these operations in a shill sandbox. Shaded boxes around nodes in the file system denote privileges held by the current session. The current working directory is indicated with a solid arrow. Dashed arrows represent low-level lookup operations, and a dashed box around a node represents privileges propagated in response to a lookup operation.

In the left diagram, the current session has a capability to the vnode corresponding to the directory `/home/alice` and a capability to the current working directory. The first operation (lookup “`..`” in `/home/bob`) is permitted because the process has the `+lookup` privilege, but privileges are not propagated to the vnode for `/home`. Thus, the second operation (lookup `alice` in `/home`) fails because the session does not have the necessary privileges. The `open` system call returns `EACCES` to indicate that the process had insufficient privileges.

The right diagram considers the same scenario, but where the session also has a `+lookup` privilege to the directory `/home`. In this case, the session is permitted to look up `alice` in `/home`. The final operation (lookup `dog.jpg` in `/home/alice`) also succeeds. These two lookups propagate privileges from the parent nodes to the results of the lookup. Looking up `dog.jpg` in `/home/alice` grants the session the privilege `+read` on the vnode representing `dog.jpg`, since the session had privilege `(+lookup +read)` on the vnode for `/home/alice`.

[⊠]We also do not propagate privileges when the directory entry is “`.`”, since this can lead to privilege amplification. For example, if session *S* has only the privilege `(+lookup +stat)` on directory `d`, then calling `openat(d, ".", flags)` would give *S* the `+stat` privilege on `d`.

Thus, the call `open("../alice/dog.jpg", O_RDONLY)` succeeds.

Note that unlike SHILL scripts, sandboxed executables are vulnerable to confused deputy attacks if they allow clients to specify resources with paths rather than, e.g., file descriptors. However, the authority of the sandboxed execution is still limited by the capabilities it is granted.

AVOIDING PRIVILEGE AMPLIFICATION In the SHILL language, capabilities both designate resources and confer privileges. As a consequence, it is possible to have two separate capabilities to the same resource with different privileges. These separate capabilities may confer less privilege than a single capability with the combined privileges. For example, consider a pair of capabilities to create a network socket, one with sufficient privileges to send but not receive messages at a particular port, and one with sufficient privileges to receive but not send messages on the same port. Because only a single socket can be bound to a port, a program with these capabilities must choose to either send or receive messages.

Since in the SHILL language, scripts cannot combine capabilities, possessing multiple capabilities for the same resource does not lead to privilege amplification. In the capability-based sandbox, however, processes designate resources using the traditional POSIX API and authorization decisions are based separately on the capabilities associated with the target object. Thus, to avoid privilege amplification the sandbox must prevent two separate capabilities to the same object from being combined to allow additional operations.

For network sockets, the privilege map is actually a map from sessions to sets of sets of privileges. That is, for privilege map m, if $P \in m(S)$, then P is a set of privileges that is consistent with how session S has used the socket so far. For example, if session S has privileges to create read-only sockets or write-only sockets, then when it creates a new socket, the privileges for the session $m(S)$ will contain the sets {`+read`} and {`+write`}, as it is not yet clear whether the process intended to create a read-only or write-only socket. When session S attempts a send or receive operation, if P is not consistent with the attempted operation, it is removed from the privilege map. In this way, the sandbox lazily discovers the "intended" set of privileges for the resource.

For file system operations that create new objects (e.g., creating new files or directories), SHILL requires that a session is never granted conflicting privileges to the same object. For example, if session S currently has privilege `(+create-file +read +stat +path)` for a directory d, (i.e., the privilege to create read-only files), and due to a lookup from the parent directory we want to propagate privilege `(+create-file +write)`, we would *not* merge these privileges, i.e., we would *not* give S the privilege `(+create-file +write +read +stat +path)`. While more sophisticated techniques to track privileges are possible, we have found that this conservative approach to prevent privilege amplification works well in practice, and does not break functionality of any of our case studies.

PROCESS INTERACTION The SHILL language provides limited support for operations on processes: SHILL does not have capabilities to control the creation of processes, process synchronization, interprocess communication, etc.

Within capability-based sandboxes, we enforce a simple security policy for operations related to processes: processes in a session can only interact with processes in the same session or a descendent session. A process in a sandbox cannot debug, send signals to, or wait for a process outside of its session.

DEBUGGING SHILL provides several tools for debugging processes running in SHILL sandboxes. First, there is a command-line tool for running a single shell command with capabilities specified in a policy file. Second, for all SHILL sandboxes, logging can be enabled and viewed by privileged users. The log records all of the capabilities and privileges granted during a session in addition to all operations that were denied because of insufficient privileges. Using the command-line tool, a session can be created in debugging mode, which automatically grants the necessary privileges if an operation would fail. We found that running programs in a debugging sandbox and then viewing the logs was a useful starting point for identifying necessary capabilities to provide to a SHILL script. However, as we developed additional standard library support to run common executables, this became less necessary. In most cases, the utilities in the standard library automate the retrieval and collection of capabilities needed to run an executable.

Limitations

SHILL's capability-based sandboxes rely on the MAC framework to implement access control checks based on capabilities. Thus, the granularity of the MAC framework's mechanism determines the granularity at which our sandboxes protect resources. For example, the MAC framework exposes a single entry point for operations that write to filesystem objects, so we cannot distinguish write and append operations. Conservatively, we enforce that to write (or append) to a file, a session must have both the `+write` and `+append` privileges for the file. (Note that in SHILL scripts, privileges can be enforced at fine granularity, since capability safety in scripts relies on language abstractions, not on the MAC framework.)

The MAC framework does not interpose on `read` or `write` operations on character devices. Thus, while the SHILL language exposes `stdin`, `stdout`, and `stderr` as file capabilities and enforces restrictions on how they can be used, sandboxed processes can bypass these restrictions if one of these capabilities abstracts a pseudo-terminal or other device. This limitation is not fundamental and can be resolved by adding entry points to the MAC framework around unprotected operations. It can be mitigated by not granting capabilities to such devices to sandboxes.

4.3 Evaluation

We evaluate the expressiveness of SHILL through four case studies: a grading script for a programming assignment, a package management script for the GNU Emacs editor, sandboxing the Apache web server, and a find and execute task similar to the example in Section 4.1. We measure the performance of SHILL via case studies and microbenchmarks. Our evaluation indicates that (1) SHILL is a practical security tool for typical system tasks}, (2) SHILL can provide fine-grained security guarantees when scripts are used to compose untrusted software and, (3) its performance cost is pay-as-you-go, i.e., weak security guarantees incur little overhead.

4.3.1 Case studies

Grading submissions We used SHILL to securely grade student submissions written in OCaml for an undergraduate programming languages course. As a baseline, we wrote a 61-line Bash script that compiles the OCaml source code of each submission and runs the compiled program against a test suite. Results of executing student submissions against the test suite are recorded in a grading directory, one file per student.

With minimal effort, we secured this Bash script in a SHILL sandbox. The capability-safe script that executes the Bash script in a sandbox is 22 lines, of which 14 are the contract for the script. The ambient script that invokes capability-safe script is also 22 lines. The contract guarantees that the grading script can at most: read files in directories containing student submissions and tests; create, modify, and delete new files in a working directory and the output directory; and access the system resources needed to run the compiler and compiled programs.

To demonstrate the finer-grained guarantees of SHILL, we also wrote a version of the grading script exclusively in SHILL. The capability-safe grading script is 78 lines of code, of which six are the script's contract. The ambient script that invokes it is 16 lines. The SHILL script provides all the security guarantees of the sandboxed Bash script, and also ensures that while grading a student's submission, no other student's submission, working-directory files, or results file can be accessed.

The capability-safe SHILL script was developed by manually translating and modifying the original Bash script. String-based references to files were replaced with appropriate capabilities. Calls to programs like `gmake`, `diff`, and `ocamlrun` were replaced with calls to the SHILL standard library to package and execute those programs. To enable this, the ambient script creates a `native-wallet` initialized with a standard `PATH` and `LD_LIBRARY_PATH`. Contracts for the capability-safe SHILL script ensure that each student's grading file is isolated from other students and that students' programs can't directly modify their grade file. These fine-grained guarantees—which the Bash script does not provide—are achieved by ensuring that the contract on the grading directory allows only the creation of new append-only files, and the functions that compile and execute a student's submission are given no capabilities to other students' grading files.

In developing this script, we debugged several cases where the script had too few privileges to run successfully. In one case, we wrote too restrictive a contract for the submissions directory, forgetting the `+lookup` privilege. The resulting contract failure indicated which argument had insufficient privileges. After verifying that this privilege was necessary and did not compromise the security guarantees, we fixed the script. We encountered two issues with sandboxed executables. First, the wallet used to launch executables was missing some necessary capabilities: when trying to compile students' submissions, `ocamlc` reported that it was unable to read a file in `/usr/local/lib/ocaml`. Investigating, we realized that OCaml searches for libraries in this directory. Adding the directory to the wallet as a dependency for OCaml executables fixed the issue but revealed another: `ocamlyacc` could not write to `/tmp`. After adding a capability to `/tmp` when invoking `gmake`, the script ran successfully. To ensure isolation between different invocations of `gmake`, we used a contract on the `/tmp` capability to specify that sandboxed processes can only read, modify, or delete files or directories they create.

Package Management We used shill to write an installation script for GNU Emacs (similar to what may be found in a package manager). The script provides functions to download, compile, install, and uninstall Emacs. Unlike a typical package manager, the script has a detailed security interface for each function. For example, only the function for downloading the source code can access the network, and only the install function can write to the intended installation directory. In addition, the install function is restricted from reading, altering, or removing any existing files in the installation directory, and the uninstall function's contract gives a list of files that it is permitted to remove. The package manager comprises 114 lines of ambient code, and 91 lines of capability-safe code, of which 45 specify contracts.

Apache web server To showcase how shill handles networking applications, we used shill to develop a sandbox for the Apache webserver, version 2.2. We tested the performance of the web server by using the Apache Benchmark tool to download a 50MB file served by Apache five thousand times using up to 100 concurrent connections. In addition to its required libraries, the script's contract gives the webserver read-only access to configuration files and web content directories, the ability to create and use sockets, and write-only access to log files. The ambient script is 27 lines, and the capability-safe script is 30 lines, of which 20 lines are contracts.

Find As another example of how programmers can use shill to gradually strengthen the guarantees of scripts, we developed two versions of a shill script for a find and execute task. Our scripts find all files with extension `.c` in the BSD source tree that contain the string "`mac_`", the prefix on entry points for the MAC framework. Completing this task requires visiting 57,817 files and invoking `grep` on the 15,376 files with extension `.c`.

The simpler version is a shill script that launches a sandbox for the command `find /usr/src -name "*.c" -exec grep -H mac_ {} \;`. The ambient script is 11 lines and calls a 27-line capability-safe script, of which 5 lines are contracts. The contract ensures that the sandbox

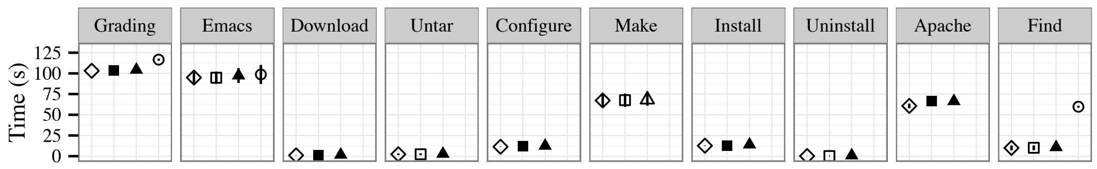

Figure 32: Performance of SHILL for a variety of tasks. Running time is given for the "Baseline" (◇), "SHILL installed" (□), "Sandboxed" (△), and "SHILL version" (◯) configurations. 95% confidence %intervals are indicated by vertical bars. Bars may be hidden by %plotting symbols when confidence intervals are small. Configurations that differ significantly from "Baseline" are filled (e.g., ■).

has access only to `/usr/src` and files necessary to run `find` and `grep`.

The second version uses the `find` function (Figure 28) to find files with the extension `.c` and invokes `grep` in a sandbox for each matching file. In addition to the guarantees of the previous version, this script provides the fine-grained guarantee that the files that `grep` operates on are exactly the files selected by the `find` function. Note that our first script does not provide this guarantee: paths passed to `grep` may resolve to different files. The ambient script is 9 lines, and the capability-safe script is 60 lines, of which 11 are contracts.

4.3.2 Performance Analysis

Our prototype implementation focuses on providing fine-grained security guarantees, and we have not yet optimized performance. However, to verify that the performance costs of SHILL are commensurate with the security guarantees, we use the case studies as benchmarks. We also develop benchmarks for sub-tasks of the Emacs installation script (download, untar, configure, make, make install, make uninstall). For each benchmark, we derive a command line invocation to achieve the same task as the case study outside of SHILL (if such a command was not already part of the case study).

We measured the performance of each benchmark in three different configurations. The "Baseline" configuration executes the command on FreeBSD without the SHILL kernel module installed. The "SHILL installed" configuration executes the command with the kernel module installed (but not active). The "Sandboxed" configuration uses a SHILL script to create a sandbox for the command. Where applicable, we also executed a "SHILL version" of the case study that replaces the command.

We ran each configuration of each benchmark 50 times and computed the mean time to completion along with a 95% confidence interval. The performance measurements were conducted on a six core, 3.33GHz Xeon server with 6GB of RAM running FreeBSD 9.2. Figure 32 presents the results. We compare performance with "Baseline" using a two-sided t-test on the difference in mean run time. Statistical significance was determined at the 0.05 level after a Bonferroni correction for multiple hypothesis testing within each benchmark.

First, observe that the overhead of our system for programs that are not secured by SHILL scripts is negligible. Second, the slowdown for "Sandboxed" and "SHILL version" configurations ranges from negligible to 1.21×, except for a few extreme cases: the "Sandboxed" configurations of the Download and Uninstall benchmarks and the "SHILL version" of the Find benchmark. These tasks are 1.73×, 6.61×, and 6.01× slower than the baseline, respectively. We explore these high overheads below. Third, the SHILL version of the package management benchmark has no significant overhead and the SHILL version of the grading script is only 1.13× slower, despite the finer-grained guarantees these scripts provide.

PROFILING To better understand the performance of SHILL, we profiled the "SHILL version" configurations of the Grading and Find benchmarks, and the "Sandboxed" configurations of Download and Uninstall. We inserted instrumentation to measure the total execution time, Racket startup (which includes script compilation, and starting the runtime), setup of sandboxes, and sandboxed execution for each benchmark. Figure 33 shows the results. Remaining time (i.e., time not spent on Racket startup, sandbox setup, or sandboxed execution) is time spent executing SHILL scripts, including contract checking. We used a Racket profiler [108] to estimate how SHILL's features affect the running time. Most time spent executing SHILL scripts is in capability-safe scripts (more than 99% for both Find and Grading) and in particular checking contracts (86% for Find and 87% for Grading). The contract on the result of `pkg-native` accounts for almost all contract checking time (92% and 93% of contract checking time for Find and Grading respectively) because it is checked once per sandbox. (The remaining time for the Download and Uninstall benchmarks was insufficient for the profiler to produce meaningful data.)

For these benchmarks, most time outside of sandboxed execution is spent enforcing security guarantees: checking contracts and setting up sandboxes. The Grading benchmark creates 5,371 sandboxes, Find creates 15,292, Uninstall creates one, and Download creates two (one for `pkg-native` and one for the executable, `curl`). Grading and Find create many sandboxes, each of which takes a relatively small amount of time to set up and a relatively small amount of time to check the contract from `pkg-native`. Racket startup cost is responsible for the high overhead of Download and Uninstall. The high overhead of Find is due to contract checking and sandbox setup, but also due to high sandboxed execution time. A small portion of the latter cost is due to overhead on system call interposition for privilege checking (see microbenchmarks below). We conjecture that the remaining cost stems from the high number of short-lived sandboxes that Find creates, which causes contention between threads using privilege maps and the kernel's asynchronous cleanup of expired SHILL sandbox sessions.

MICROBENCHMARKS To understand the overhead added to system calls due to privilege checking during sandboxed execution (see Section 4.2.2.2), we evaluated microbenchmarks for sev-

	Uninstall	Download	Grading	Find
Total time	0.82 s	1.66 s	116.38 s	61.20 s
Racket startup	0.65 s	0.63 s	0.92 s	0.65 s
Sandbox				
setup	0.01 s	0.01 s	6.98 s	18.04 s
execution	0.14 s	0.96 s	104.09 s	27.61 s
Remaining time	0.03 s	0.07 s	4.39 s	14.90 s

Figure 33: Performance breakdown of four benchmarks.

Operation	SHILL Installed	Sandboxed	Difference
`pread-1B`	516 ± 80 ns	560 ± 64 ns	44 ± 102 ns
`pread-1MB`	199 ± 4 ms	202 ± 6 ms	3 ± 7 ms
`create-unlink`	13 ± 3 ms	14 ± 4 ms	1 ± 4 ms
`open-read-close`			
1 lookup	3.7 ± 0.4 ms	4.0 ± 0.4 ms	0.3 ± 0.6 ms
5 lookups	5.3 ± 0.3 ms	6.4 ± 0.5 ms	1.1 ± 0.6 ms

Figure 34: Overhead of SHILL for microbenchmarks.

eral representative system calls under both the "SHILL installed" and "Sandboxed" configurations. The `pread-1B` microbenchmark reads one byte from an opened file; `pread-1MB` reads 1 megabyte. The `create-unlink` microbenchmark creates a new file, closes, and unlinks it. The `open-read-close` benchmarks open a file, reads one byte, and closes it. In one version of this benchmark, the path argument to `open` has length one, and in the other it has length five (i.e., the file is nested in 4 subdirectories).

We timed one million iterations of each microbenchmark, except for `pread-1MB`, which was executed one thousand times. Figure 34 shows the mean execution time and 95% confidence intervals. All differences were statistically significant. The overhead of executing system calls in a SHILL sandbox ranges between 18% (`open-read-close`, 5 lookups) and 1% (`pread-1MB`). For the `open-read-close` benchmarks, further experiments (not shown) indicate that overhead increases linearly in the length of the path (i.e., linearly with the number of lookup system calls required).

4.4 Related Work

There is extensive research on capability-based security that has resulted in a plethora of systems, programming languages, design patterns, and reasoning techniques.

Operating Systems Capability-based operating systems [20] such as PSOS [85], KeyKOS [14, 45], EROS [103], and Coyotos [102] use operating system and hardware capabilities to limit the authority of users and processes. Numerous microkernels inspired by the L4 family [66] employ capabilities as an access control mechanism [10, 50, 62]. SHILL is not an operating system and is built on a commodity operating system. However, it shares similar goals and draws inspiration from these novel systems. For instance, the source of certain kinds of capabilities in KeyKOS is the *command system*: the only program in the system with ambient access to a user's directory.

SHILL's ambient scripts serve the same purpose.

Capsicum [126] extends the FreeBSD operating system with capabilities but requires programs to be rewritten to use the capability-based interfaces in order to make use of capability mode. By contrast, SHILL's capability-based sandbox does not require executables to be aware of capabilities. In addition, SHILL capabilities are more expressive than Capsicum capabilities; for example, a SHILL capability can express the permission to create files in a directory and delete only files that were created with the capability.

PROGRAMMING LANGUAGES The use of language-level capabilities has a long history [82]. The E programming language [80] is a seminal *object capability language*, where capabilities are object references. CapDesk [111, 122] is a desktop shell for launching applications written in E. Applications are granted limited authority initially and can gain more capabilities through *powerboxes*, which mediate requests for authority from the application to the user. In contrast to SHILL, CapDesk does not have a scripting interface and applications launched by CapDesk must be capability-aware and designed to work with the CapDesk framework.

Joe-E [73] restricts Java to an object-capability-safe subset. Similarly, Caja [78] introduces an object-capability-safe subset of JavaScript. Maffeis et al. [70] prove that these subsets are indeed capability safe. Unlike other capability-safe languages, SHILL targets a particular domain (shell scripting) instead of general programming and uses contracts to manage capabilities instead of capability-based design patterns [80].

OTHER APPROACHES TO SYSTEMS SECURITY Laminar [97] integrates operating system and programming language abstractions to enforce decentralized information flow control (DIFC). Its high-level architecture resembles that of SHILL. However, Laminar provides fine-grained security only for programs that use Laminar's security abstractions, and does not provide declarative security specifications. Hails [40] uses declarative information-flow control policies as a mechanism for composing mutually distrusting web applications. Unlike SHILL, it provides limited support for securing legacy applications. HiStar [128], Asbestos [30], and Flume [61] track information flow to enforce fine-grained security policies. While all of these methods can enforce security restrictions on untrusted applications, SHILL uses capabilities and contracts rather than DIFC labels.

A wide variety of sandboxing tools have been developed for commodity operating systems, including SELinux [67], Seatbelt [125], AppArmor [4], Grsecurity [107], LXC [69], and Docker [27]. Unlike SHILL, these sandboxes deny or grant access based on a profile rather than a programmable capability-based interface. Mbox [58] and TxBOX [52] create sandboxes with transactional semantics that can reverse the effects of misbehaving processes, but enforce strong isolation between sandboxed processes and the rest of the system. Notably, programs running in a SHILL sandbox are not isolated from the rest of the system. For example, in our Apache case study, concurrently executing programs can dynamically add new web content or view logs as they are generated. Many of these sandboxes require root privileges, but some are available to all users [58]. PLASH [101] is a capability-based interactive shell for creating sandboxes in which to execute shell commands, similar to SHILL's `exec`. All of these tools lack the reasoning principles SHILL provides for composing multiple sandboxes together.

5 Authorization Contracts

In the previous chapter, we have seen how one approach to access control, capabilities, can be improved by using behavioral contracts to express and enforce security properties. This approach is limiting because it forces programmers to write programs that are explicitly structured to meet the requirements of capability-based security. This is not just a problem of capability-based security. In general, 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 [123] 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 [80] and Caja [78]) 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 chapter we propose a new, extensible access control framework based on software contracts that allows component authors to design access control monitors that suit their needs. The framework supports the design and implementation of many different custom 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.

In designing such a framework, we first must consider the tasks that an access control monitor must perform, and how those tasks reveal choices in the design of such monitors. 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 was `setuid`. 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 a user could not otherwise access.

Similar to operating systems, software components also need access control mechanisms to prevent certain 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 as access control mechanisms for operating systems: which rights are necessary for the call and which rights the request possesses.

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 granted 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 (Section 5.2). We implement our framework as a library for Racket [38] without changes to the language's runtime. We use higher-order contracts [37] to specify where an access control monitor should interpose on a program and how it should manage authority environments. In the same way that existing behavioral contracts support separation of concerns by removing defensive checks from programs, 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 (Section 5.1),
2. the introduction of *context contracts* to check and enforce properties of execution contexts (Section 5.2.1),
3. a novel *authorization logic* for representing and querying authority in authority environments (Section 5.2.2), and
4. *authorization contracts* that specialize context contracts for managing authority environments and enforcing access control policies expressed in the logic (Section 5.2.3).

We have used the framework to implement diverse access control mechanisms: discretionary access control, stack inspection, history-based access control, and object-capabilities (Section 5.3). We demonstrate the practicality of our approach with three realistic case studies (Section 5.4). Finally, we discuss related work (Section 5.5).

5.1 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 (Section 5.1.1). Then we describe the connection between lexical and dynamic scoping and access control (Section 5.1.2) and show how we can use scoping in the design of a framework for writing access control monitors (Section 5.1.3). Throughout, we use small examples in the Racket programming language [38].

5.1.1 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 [39, 109, 110].

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))))
0
```

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))))
42
```

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 use the file.

5.1.2 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 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 Privilege [98], 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, calling 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.

```
(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 35: Defining a simple access control monitor.

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 5.1.3 shows how authority environments help enforce access control in our running example, including how to create a secure `login` function. Section 5.2 generalizes authority environments so that we can express a wide variety of access control mechanisms.

5.1.3 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 5.3. Here we explain only the features relevant to the example.

Figure 35 shows our example monitor. This monitor specifies three actions: `setuid/c`, `chuser/c`, and `checkuser/c`. Each action defines a higher-order function contract [37]. 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* [37] 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 #:auth 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:

```
(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 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.

5.2 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 (Section 5.2.1). 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 (Section 5.2.2). 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 (Section 5.2.3). Authorization contracts are specializations of context contracts that use the authorization logic to succinctly describe how they manage authority environments.

5.2.1 A contract system with context contracts

We model higher-order contracts and context contracts as extensions to an applied lambda calculus with parameters to support dynamic binding. This model extends the model presented in Chapter 2. Extensions to its syntax are given in Figure 36. Figure 38 gives the additional rules of reduction semantics for the extended model. Metafunction do-parameterization, which is used in the semantics of context contracts, is shown in Figure 39.

Evaluation contexts are standard and enforce call-by-value, left-to-right evaluation. The additional rules of the typing judgment for the language are given in Figure 40. The extension

```
  v ::= .... | (param r) | (t : τ) | (param/p j k l ctc v) | (tag/p j k l ctc v) | (ctx/p : τ j l v g ... v)
  g ::= (v ⇒ v ← v)
ctc ::= .... | (param/c ctc) | (tag/c ctc) | (→a : τ ctc v ctc) | (ctx/c : τ v g ... v g ...)
  e ::= .... | (make-parameter e) | (parameterize (e e) e) | (? e) | (e ← e)
      | (make-tag : τ) | (reset e e) | (shift e x e)
      | (param/c e) | (tag/c e) | (→a : τ e e e) | (ctx/c : τ e ge ... e ge ...)
 ge ::= (e ⇒ e ← e)
  τ ::= .... | (τ param) | (τ tag)
  E ::= .... | (make-parameter E) | (parameterize (E e) e)
      | (parameterize ((param r) E) e) | (parameterize ((param r) v) E)
      | (param/c E) | (tag/c E) | (→a : τ E e e) | (→a : τ ctc E e) | (→a : τ ctc v E)
      | (E ← e) | ((param r) ← E) | (? E) | (reset E e) | (reset v E) | (shift E x e)
      | (ctx/c : τ E ge ... e ge ...) | (ctx/c : τ v g ... (E ⇒ e ← e) ge ... e ge ...)
      | (ctx/c : τ v g ... (v ⇒ E ← e) ge ... e ge ...) | (ctx/c : τ v g ... (v ⇒ v ← E) ge ... e ge ...)
      | (ctx/c : τ v g ... E ge ...) | (ctx/c : τ v g ... v g ... (E ⇒ e ← e) ge ...)
      | (ctx/c : τ v g ... v g ... (v ⇒ E ← e) ge ...) | (ctx/c : τ v g ... v g ... (v ⇒ v ← E) ge ...)
      | (check j k E e)
  T ::= [] | (tag/p j k l ctc T)
```

Figure 36: Syntax extensions for context contracts.

supports first class delimited continuations [19] because our implementation language supports them and they interact in interesting ways with our framework.

The semantics of common language features was presented in Chapter 2. Below, we explain the semantics of parameters, continuations, higher-order contracts, and context contracts.

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 (param 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 ((param r) e_1) e_2) installs the result of e_1 as the new value of the parameter (param r) for the dynamic extent of e_2. Access-

$\langle E[(\text{make-parameter } v)], \sigma\rangle \longrightarrow$ [make-param]
$\langle E[(\text{param } r)], \sigma[r \mapsto v]\rangle$
where r fresh

$\langle E[((\text{param } r) \leftarrow v)], \sigma\rangle \longrightarrow$ [set-param-default]
$\langle E[v], \sigma[r \mapsto v]\rangle$
where #t = $\text{param-free}[\![E, (\text{param } r)]\!]$

$\langle E_1[(\text{parameterize } ((\text{param } r)\ v_1)\ E_2[((\text{param } r) \leftarrow v_2)])], \sigma\rangle \longrightarrow$ [set-param]
$\langle E_1[(\text{parameterize } ((\text{param } r)\ v_2)\ E_2[v_2])], \sigma\rangle$
where #t = $\text{param-free}[\![E_2, (\text{param } r)]\!]$

$\langle E[(?\ (\text{param } r))], \sigma\rangle \longrightarrow$ [read-param-default]
$\langle E[v], \sigma\rangle$
where #t = $\text{param-free}[\![E, (\text{param } r)]\!]$, $\sigma(r) = v$

$\langle E_1[(\text{parameterize } ((\text{param } r)\ v)\ E_2[(?\ (\text{param } r))])], \sigma\rangle \longrightarrow$ [read-param]
$\langle E_1[(\text{parameterize } ((\text{param } r)\ v)\ E_2[v])], \sigma\rangle$
where #t = $\text{param-free}[\![E_2, (\text{param } r)]\!]$

$\langle E[(\text{parameterize } ((\text{param } r)\ v_p)\ v)], \sigma\rangle \longrightarrow$ [parameterize]
$\langle E[v], \sigma\rangle$

$\langle E[(\text{make-tag} : \tau)], \sigma\rangle \longrightarrow$ [make-tag]
$\langle E[(t : \tau)], \sigma\rangle$
where t fresh

$\langle E[(\text{reset } T[(t : \tau)]\ v)], \sigma\rangle \longrightarrow$ [reset]
$\langle E[v], \sigma\rangle$

$\langle E_1[(\text{reset } T_1[(t : (\tau_d \to \tau_r))]$ $\longrightarrow$ [shift]
$\quad E_2[(\text{shift } T_2[(t : (\tau_d \to \tau_r))]\ x\ e)])], \sigma\rangle$
$\langle E_1[(\text{reset } T_1[(t : (\tau_d \to \tau_r))]\ e[x \mapsto \text{wrap-pos}[\![T_1[(t : (\tau_d \to \tau_r))], \text{wrap-neg}[\![T_2[(t : (\tau_d \to \tau_r))], v_k]\!]]\!]])], \sigma\rangle$
where #t = $\text{tag-free}[\![E_2, t]\!]$, $v_k = (\lambda\ (x_f : \tau_d)\ (\text{reset } T_1[(t : (\tau_d \to \tau_r))]\ E_2[x_f]))$, x_f fresh

Figure 38: Reduction semantics for extensions.

$$\langle E[(\text{mon}\ j\ k\ l\ (\text{→a} : \tau\ ctc_d\ v_c\ ctc_r)\ v)],\ \sigma\rangle \longrightarrow \qquad [\text{→a}]$$
$$\langle E[(\lambda\ (x : \tau)\ (\text{mon}\ j\ k\ l\ ctc_r$$
$$((\text{mon}\ j\ k\ l\ (v_c\ (\text{mon}\ j\ l\ j\ ctc_d\ x))\ v)$$
$$(\text{mon}\ j\ l\ k\ ctc_d\ x))))],\ \sigma\rangle$$

$$\langle E[(\text{mon}\ j\ k\ l\ (\text{ctx/c} : (\tau_d \to \tau_r)\ v_c\ g_c\ \ldots\ v_a\ g_a\ \ldots)\ v)],\ \sigma\rangle \longrightarrow \qquad [\text{ctx/c}]$$
$$\langle E[(\text{check}\ j\ l\ (v_c\ ())\ e_p)],\ \sigma\rangle$$
$$\text{where } e_p = \text{do-parameterization}[\![\tau_d,\ (\text{ctx/p} : (\tau_d \to \tau_r)\ j\ l\ v_a\ g_a\ \ldots\ v),\ \ g_c,\ \ldots]\!]$$

$$\langle E[((\text{ctx/p} : (\tau_d \to \tau_r)\ j\ l\ v_a\ g_a\ \ldots\ v_f)\ v)],\ \sigma\rangle \longrightarrow \qquad [\text{ctx/p}]$$
$$\langle E[(\text{check}\ j\ l\ (v_a\ ())\ (e_p\ v))],\ \sigma\rangle$$
$$\text{where } e_p = \text{do-parameterization}[\![\tau_d,\ v_f,\ g_a,\ \ldots]\!]$$

$$\langle E[(\text{mon}\ j\ k\ l\ (\text{tag/c}\ ctc)\ v)],\ \sigma\rangle \longrightarrow \qquad [\text{tag/c}]$$
$$\langle E[(\text{tag/p}\ j\ k\ l\ ctc\ v)],\ \sigma\rangle$$

$$\langle E[(\text{mon}\ j\ k\ l\ (\text{param/c}\ ctc)\ v)],\ \sigma\rangle \longrightarrow \qquad [\text{param/c}]$$
$$\langle E[(\text{param/p}\ j\ k\ l\ ctc\ v)],\ \sigma\rangle$$

$$\langle E[(?\ (\text{param/p}\ j\ k\ l\ ctc\ v))],\ \sigma\rangle \longrightarrow \qquad [\text{read-param/p}]$$
$$\langle E[(\text{mon}\ j\ k\ l\ ctc\ (?\ v))],\ \sigma\rangle$$

$$\langle E[(\text{parameterize}\ ((\text{param/p}\ j\ k\ l\ ctc\ v_p)\ v)\ e)],\ \sigma\rangle \longrightarrow \quad [\text{parameterize-param/p}]$$
$$\langle E[(\text{parameterize}\ (v_p\ (\text{mon}\ j\ l\ k\ ctc\ v))\ e)],\ \sigma\rangle$$

$$\langle E[((\text{param/p}\ j\ k\ l\ ctc\ v_p) \leftarrow v)],\ \sigma\rangle \longrightarrow \qquad [\text{set-param/p}]$$
$$\langle E[(v_p \leftarrow (\text{mon}\ j\ l\ k\ ctc\ v))],\ \sigma\rangle$$

$$\langle E[(\text{check}\ j\ k\ \text{\#t}\ e)],\ \sigma\rangle \longrightarrow \qquad [\text{check-true}]$$
$$\langle E[e],\ \sigma\rangle$$

$$\langle E[(\text{check}\ j\ k\ \text{\#f}\ e)],\ \sigma\rangle \longrightarrow \qquad [\text{check-false}]$$
$$\langle (\text{error}\ j\ k),\ \sigma\rangle$$

Figure 38: Reduction semantics for extensions (Continued).

$$\text{do-parameterization}[\![\tau,\ v]\!] =$$
$$v$$

$$\text{do-parameterization}[\![\tau,\ e,\ (v_g \Rightarrow v_p \leftarrow v_v),\ g,\ \ldots]\!] =$$
$$(\text{let}\ (x\ \text{do-parameterization}[\![\tau,\ e,\ g,\ \ldots]\!])$$
$$(\text{if}\ (v_g\ ())$$
$$(\text{let}\ (x_v\ (v_v\ ()))$$
$$(\lambda\ (x_a : \tau)\ (\text{parameterize}\ (v_p\ x_v)\ (x\ x_a))))$$
$$x))$$

Figure 39: Metafunction implementing guarded parameterizations.

$$\frac{\Gamma; \Sigma \vdash e : \tau}{\Gamma; \Sigma \vdash (\textsf{make-parameter}\ e) : (\tau\ \textsf{param})} \qquad \frac{\Sigma(r) = \tau}{\Gamma; \Sigma \vdash (\textsf{param}\ r) : (\tau\ \textsf{param})}$$

$$\frac{\begin{array}{c}\Gamma; \Sigma \vdash e_1 : (\tau_p\ \textsf{param}) \\ \Gamma; \Sigma \vdash e_2 : \tau_p \\ \Gamma; \Sigma \vdash e_3 : \tau\end{array}}{\Gamma; \Sigma \vdash (\textsf{parameterize}\ (e_1\ e_2)\ e_3) : \tau} \qquad \frac{\Gamma; \Sigma \vdash e : (\tau\ \textsf{param})}{\Gamma; \Sigma \vdash (?\ e) : \tau} \qquad \frac{\begin{array}{c}\Gamma; \Sigma \vdash e_1 : (\tau\ \textsf{param}) \\ \Gamma; \Sigma \vdash e_2 : \tau\end{array}}{\Gamma; \Sigma \vdash (e_1 \leftarrow e_2) : \tau}$$

$$\frac{}{\Gamma; \Sigma \vdash (\textsf{make-tag} : \tau) : (\tau\ \textsf{tag})} \qquad \frac{}{\Gamma; \Sigma \vdash (t : \tau) : (\tau\ \textsf{tag})} \qquad \frac{\begin{array}{c}\Gamma; \Sigma \vdash e_1 : ((\tau_1 \rightarrow \tau_2)\ \textsf{tag}) \\ \Gamma; \Sigma \vdash e_2 : \tau_2\end{array}}{\Gamma; \Sigma \vdash (\textsf{reset}\ e_1\ e_2) : \tau_2}$$

$$\frac{\begin{array}{c}\Gamma; \Sigma \vdash e_1 : ((\tau_1 \rightarrow \tau_2)\ \textsf{tag}) \\ \Gamma[x \mapsto (\tau_1 \rightarrow \tau_2)]; \Sigma \vdash e_2 : (\tau_2)\end{array}}{\Gamma; \Sigma \vdash (\textsf{shift}\ e_1\ x\ e_2) : \tau_1} \qquad \frac{\Gamma; \Sigma \vdash e : (\tau\ \textsf{contract})}{\Gamma; \Sigma \vdash (\textsf{param/c}\ e) : ((\tau\ \textsf{param})\ \textsf{contract})}$$

$$\frac{\Gamma; \Sigma \vdash e : (\tau\ \textsf{contract})}{\Gamma; \Sigma \vdash (\textsf{tag/c}\ e) : ((\tau\ \textsf{tag})\ \textsf{contract})} \qquad \frac{\begin{array}{c}\Gamma; \Sigma \vdash e_d : (\tau_d\ \textsf{contract}) \\ \Gamma; \Sigma \vdash e_a : (\tau_d \rightarrow ((\tau_d \rightarrow \tau_r)\ \textsf{contract})) \\ \Gamma; \Sigma \vdash e_r : (\tau_r\ \textsf{contract})\end{array}}{\Gamma; \Sigma \vdash (\rightarrow\textsf{a} : \tau_d\ e_d\ e_a\ e_r) : ((\tau_d \rightarrow \tau_r)\ \textsf{contract})}$$

$$\frac{\begin{array}{c}\Gamma; \Sigma \vdash e_c : (\textsf{Unit} \rightarrow \textsf{Bool}) \\ \Gamma; \Sigma \vdash ge_c \quad \ldots \\ \Gamma; \Sigma \vdash e_a : (\textsf{Unit} \rightarrow \textsf{Bool}) \\ \Gamma; \Sigma \vdash ge_a \quad \ldots\end{array}}{\Gamma; \Sigma \vdash (\textsf{ctx/c} : \tau\ e_c\ ge_c \ldots\ e_a\ ge_a \ldots) : (\tau\ \textsf{contract})} \qquad \frac{\begin{array}{c}\Gamma; \Sigma \vdash ctc : (\tau\ \textsf{contract}) \\ \Gamma; \Sigma \vdash v : (\tau\ \textsf{param})\end{array}}{\Gamma; \Sigma \vdash (\textsf{param/p}\ j\ k\ l\ ctc\ v) : (\tau\ \textsf{param})}$$

$$\frac{\begin{array}{c}\Gamma; \Sigma \vdash ctc : (\tau\ \textsf{contract}) \\ \Gamma; \Sigma \vdash v : (\tau\ \textsf{tag})\end{array}}{\Gamma; \Sigma \vdash (\textsf{tag/p}\ j\ k\ l\ ctc\ v) : (\tau\ \textsf{tag})} \qquad \frac{\begin{array}{c}\Gamma; \Sigma \vdash v_f : \tau \\ \Gamma; \Sigma \vdash v : (\textsf{Unit} \rightarrow \textsf{Bool}) \\ \Gamma; \Sigma \vdash g \quad \ldots\end{array}}{\Gamma; \Sigma \vdash (\textsf{ctx/p} : \tau\ j\ l\ v\ g \ldots\ v_f) : \tau}$$

Figure 40: Well-typed terms for extensions.

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

First-class continuations

Our model includes first-class delimited continuations using Danvy and Filinski's shift and reset operators [19]. Term (reset (*t* : *τ*) *e*) adds a marker with tag *t* to the current continuation and then executes *e*. Term (shift *t x e*) reifies its continuation up to the nearest reset with a matching tag *t* as a value *v*, removes the captured continuation, and invokes expression *e* with the captured continuation bound to *x*. To demonstrate these semantics, consider the following term:

```
(let (t (make-tag : (Int → Int)))
  (1 + (reset t (10 + (shift t f (f (f 100)))))))
```

This term evalutes to 121. The key step in evaluating this term occurs when the reducible expression is the shift operator: E[(shift (*t* : (Int → Int)) f (f (f 100)))]. The continuation E can be decomposed into two parts, seperated by the delimiter reset: E_1[(reset (*t* : (Int → Int)) E_2)], where E_1 is (1 + []) and E_2 is (10 + []). The shift operator binds f to a function that reifies the continuation up to the delimiter (reset (*t* : (Int → Int)) E_2): (λ (*x* : Int) (reset (*t* : (Int → Int)) (10 + *x*))). It then removes this part of the continuation, leaving just the evaluation context E_1. The resulting term (1 + ((λ (*x* : Int) (reset (*t* : (Int → Int)) (10 + *x*))) ((λ (*x* : Int) (reset (*t* : (Int → Int)) (10 + *x*))) 100))) evaluates to 121.

Of course, this example is somewhat contrived. In practice, first-class continuations are used to implement a wide-range of control abstractions including exceptions, back-tracking search, and event-based programming. While our model includes just shift and reset operators, a number of alternative formulations of first-class continuations have been proposed. Takikawa et al. [117] show how the style of contracts used in our model can be applied to these alternatives.

Higher-order contracts

Contract ($\rightarrow$a : τ ctc_d v_c ctc_r) is a generalization of indy-dependent[*] contracts for higher-order functions [25]. This contract corresponds to the `->a` contracts from Section 5.1. Attaching the contract to a function returns a new value that enforces contracts on the argument and the result of the function. The contract is *dependent* since the contract uses the argument of a contracted function to construct an additional contract for the function. This generalizes previous dependent function contracts, which construct a contract for the function's result, but not for the function itself. This extension makes it easier to reuse context and authorization contracts by defining contract combinators that are specialized to particular arguments when the contracted function is applied. Without dependent function contracts, a function whose contract should depend on its arguments would need to be duplicated for each possible contract, and different uses of the function would need to invoke the appropriately contracted copy.

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 v_c to construct a new contract. Third, the resulting 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.

The parameter contract (param/c ctc) is a higher-order contract that restricts uses of a parameter. A contracted parameter v reduces to a proxy (param/p j k l ctc v) which records the necessary

[*]An indy-dependent contract is a dependent function contract that uses the "indy" strategy for blame assignment [25].

blame information and intercepts uses of the parameter to enforce that values bound to the parameter meet contract *ctc*.

Our contracts for continuations follow Takikawa et al. [117]. Contracts for continuations are attached to the tags that connect shift operators to their corresponding reset delimiters. A tag contract (tag/c *ctc*) is a higher-order contract that restricts how delimited continuations created by shift expressions are used. Attaching a contract to a tag creates a proxy value (tag/p *j k l ctc v*) which records blame information and a contract. When a shift operator is invoked, the tags at both the shift and reset operators for the tag may be wrapped in one or more proxies attaching contracts. Meta-functions wrap-pos and wrap-neg collect the contracts from these proxies and apply them to the freshly reified continuation. Contracts origininating from the reset tag are given positive blame. That is, the responsibility for satisfying contracts on arguments is from these contract is assigned to the expression supplied by the shift. Contracts originating from the shift tag have their blame swapped, to reflect that the shift expression is obligated to use the continuation as specified by the tag's contract.

Context contracts

This work introduces *context contracts*, which are novel higher-order contracts that enforce restrictions on the execution context of function calls. 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 (ctx/c : τ v_c ($v_{c\_g} \Rightarrow v_{c\_p} \leftarrow v_{c\_v}$) ... v_a ($v_{a\_g} \Rightarrow v_{a\_p} \leftarrow v_{a\_v}$) ...) has four parts:

1. v_c, a predicate that checks whether the context is appropriate when the contract is attached,

2. $(v_{c\_g} \Rightarrow v_{c\_p} \leftarrow v_{c\_v})$ ..., a list of guarded parameterizations 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. $(v_{a\_g} \Rightarrow v_{a\_p} \leftarrow v_{a\_v})$ ..., 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 from part 2 is evaluated in turn. If applying the guard v_{c_g} returns #t, the corresponding thunk v_{c_v} is executed to produce a new value. This value is "closed over" and re-installed for parameter v_{c_p} when the contracted function is applied. The predicate v_a and the remaining parameterizations are recorded in a proxy (ctx/p : $\tau\ j\ l\ v_a\ (v_{a\_g} \Rightarrow v_{a\_p} \leftarrow v_{a\_v})\ \ldots\ v$).

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.

Figure 41 demonstrates context contracts with a simple example. The example involves two context contracts, outer/c and inner/c, that communicate via parameter p. The contracts en-

```
(let (p (make-parameter #f))
  (let (true (λ (x : Unit) #t))
    (let (outer/c (ctx/c : (Int → Int) true true (true ⇒ p ← true)))
      (let (inner/c (ctx/c : (Int → Int) true (λ (x : Unit) (? p))))
        (let (inner (mon inner-ctc in top inner/c (λ (x : Int) x)))
          (let (outer (mon outer-ctc out top outer/c (λ (x : Int) (inner x))))
            (inner 42)))))))
```

Figure 41: Context contracts enforcing nested applications.

```
(let (cp (make-parameter #f))
(let (capture/c (ctx/c : Int true (true ⇒ cp ← (λ (x : Unit) (? p))) true (true ⇒ p ← (λ (x : Unit) (? cp)))))
...))
```

Figure 42: A context contract that closes over parameter p.

sure that function inner can only be applied in the dynamic extent of function outer. Evaluating (inner 42) results in a contract error (error inner-ctc top) blaming the context that applied inner. Replacing this expression with (outer 42) evaluates to 42.

Context contracts can also close over the values of parameters. Consider extending the example in Figure 41 with the contract capture/c from Figure 42. 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. The ability to close over an environment in this way is a key feature required to implement authorization contracts.

Complete monitoring

Our contract system satisfies *complete monitoring* [25], 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 interposes 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 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 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.

5.2.2 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) [64]. In practice, subjects and

$$
\begin{aligned}
a, b, c, d &::= (\mathsf{prim}\ P) \mid \top \mid \bot \mid (a \triangleright \alpha) \mid (a \vee \mathsf{b}) \mid (a \wedge \mathsf{b}) \mid (W \rightarrow a) \mid (W \leftarrow a) \\
del &::= (v \succcurlyeq v\ @\ v) \\
W &::= \{\ del\ \ldots\ \}
\end{aligned}
$$

Figure 43: Syntax of principals, delegations, and worlds.

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 and reason about principals, the authority of principals, and how principals delegate and restrict their authority. For this purpose, we use an authorization logic [16] based on the Flow-Limited Authorization Model (FLAM) [6]. We briefly describe our logic, highlighting where it differs from FLAM. In Section 5.2.3, we use this logic to manage authority using specialized context contracts, dubbed authorization contracts.

Figure 43 presents the syntax of our logic. We assume an enumerable set of *primitive principals*, ranged over by the metavariable 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 a and b, the conjunctive principal $(a \wedge b)$ is a principal with the authority of *both* a and b. Similarly, the disjunctive principal $(a \vee b)$ has the authority of *either* a or b.

If principal a trusts principal b, we write $(b \succcurlyeq a)$, and say that b *acts for* a. 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* $(a \succcurlyeq b\ @\ c)$ means that principal c asserts that a acts for b (or, equivalently, that b delegates its authority to a). Of course, whether a principal d believes the assertion depends on whether d trusts principal c. (We differ

$$\frac{}{W\,;\,d \vdash a \succcurlyeq \bot}\,[\text{bot}] \qquad \frac{}{W\,;\,d \vdash \top \succcurlyeq a}\,[\text{top}] \qquad \frac{}{W\,;\,d \vdash a \succcurlyeq (a \triangleright D)}\,[\text{proj}]$$

$$\frac{W\,;\,d \vdash a \succcurlyeq c}{W\,;\,d \vdash (a \wedge b) \succcurlyeq c}\,[\text{conj-left-1}] \qquad \frac{W\,;\,d \vdash b \succcurlyeq c}{W\,;\,d \vdash (a \wedge b) \succcurlyeq c}\,[\text{conj-left-2}]$$

$$\frac{W\,;\,d \vdash a \succcurlyeq b \qquad W\,;\,d \vdash a \succcurlyeq c}{W\,;\,d \vdash a \succcurlyeq (b \wedge c)}\,[\text{conj-right}] \qquad \frac{W\,;\,d \vdash a \succcurlyeq c \qquad W\,;\,d \vdash b \succcurlyeq c}{W\,;\,d \vdash (a \vee b) \succcurlyeq c}\,[\text{disj-left}]$$

$$\frac{W\,;\,d \vdash a \succcurlyeq b}{W\,;\,d \vdash a \succcurlyeq (b \vee c)}\,[\text{disj-right-1}] \qquad \frac{W\,;\,d \vdash a \succcurlyeq c}{W\,;\,d \vdash a \succcurlyeq (b \vee c)}\,[\text{disj-right-2}]$$

$$\frac{(a \succcurlyeq b \mathbin{@} c) \in W \qquad W\,;\,d \vdash c \succcurlyeq d}{W\,;\,d \vdash a \succcurlyeq b}\,[\text{del}] \qquad \frac{W_2\,;\,c \vdash a \succcurlyeq b \qquad W_1\,;\,d \vdash (W_2 \leftarrow c) \succcurlyeq d}{W_1\,;\,d \vdash a \succcurlyeq (W_2 \leftarrow b)}\,[\text{closure-left}]$$

$$\frac{W_2\,;\,c \vdash a \succcurlyeq b \qquad W_1\,;\,d \vdash (W_2 \leftarrow c) \succcurlyeq d}{W_1\,;\,d \vdash (W_2 \rightarrow a) \succcurlyeq b}\,[\text{closure-right}] \qquad \frac{a = b}{W\,;\,d \vdash a \succcurlyeq b}\,[\text{refl}]$$

$$\frac{W\,;\,d \vdash a \succcurlyeq c \qquad W\,;\,d \vdash c \succcurlyeq b}{W\,;\,d \vdash a \succcurlyeq b}\,[\text{trans}]$$

Figure 44: Inference rules for judgment $W\,;\,c \vdash a \succcurlyeq b$.

from FLAM in that we describe only the integrity of delegations, not their confidentiality.)

Judgment $W\,;c \vdash a \succcurlyeq b$ denotes that given the set of delegations W, principal c believes that principal a acts for principal b. Intuitively, c believes that a acts for b if that trust relationship can be derived using only delegations asserted by principals that c trusts.

Figure 44 presents the inference rules for the judgment $W\,;c \vdash a \succcurlyeq b$. 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 c trusts only delegations asserted by principals that it trusts, that is delegations $(a \succcurlyeq b \ @ \ d)$ where $W\,;c \vdash d \succcurlyeq c$.

We have three additional principal constructors. Principal $(a \triangleright \alpha)$ is the *projection* of the authority of principal a on dimension α[†]. Projection is commutative, so $((a \triangleright \alpha) \triangleright \beta) = ((a \triangleright \beta) \triangleright \alpha)$. We use projections to limit or attenuate the authority of a principal, and to identify access rights. For example, $(a \triangleright \mathsf{files})$ may refer to principal a's authority restricted to a's rights to access the file system. Similarly, principal $((a \triangleright \mathsf{obj}) \triangleright \mathsf{invoke})$ (equivalently $((a \triangleright \mathsf{invoke}) \triangleright \mathsf{obj})$) might refer to the right to invoke a particular object belonging to principal a. Principal a can grant this right to another principal b by asserting a delegation: $(b \succcurlyeq ((a \triangleright \mathsf{obj}) \triangleright \mathsf{invoke}) \ @ \ a)$.

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

Novel to this work, we introduce *closure principals* $(W \leftarrow a)$ and $(W \rightarrow a)$. Given a set of delegations W and principal a, the *left-closure principal* $(W \leftarrow a)$ represents a with all of the trust relationships derivable from W where a delegates its authority to other principals. The *right-closure principal* $(W \rightarrow a)$ represents a with all of the trust relationships derivable from W where a acts

[†]FLAM considers *basis projections* and *ownership projections*. The projections we use here are more general and have less structure than either.

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 $W_1\,;\, d \vdash a \succcurlyeq (W_2 \leftarrow b)$ holds when there is some principal c such that at the time of closure creation (i.e., with delegation set W_2), c believed that a acted for b (premise $W_2\,;\, c \vdash a \succcurlyeq b$, and moreover, right now (i.e., with delegation set W_1) principal d trusts principal $(W_2 \leftarrow c)$ (premise $W_1\,;\, d \vdash (W_2 \leftarrow c) \succcurlyeq d$). Typically, c and d are the same principal, meaning that d-at-time-W_1 trusts the decisions made by d-at-time-W_2. Rule closure-right is similar and $W_1\,;\, d \vdash (W_2 \rightarrow a) \succcurlyeq b$ holds when there is some principal c such that at the time the closure was taken c believed that a acted for b (premise $W_2\,;\, c \vdash a \succcurlyeq b$), and principal d trusts c-at-time-W_2 (premise $W_1\,;\, d \vdash (W_2 \leftarrow c) \succcurlyeq d$).

To query whether a particular set of delegations satisfies an acts-for relation, we use a proof search algorithm adapted from FLAM [6]. We give examples of using delegations to implement different authorization mechanisms in Section 5.3.

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.

```
         v ::= .... | a | b | c | d | α | β | del | W
         e ::= .... | (actions (act ...) e) | new-principal | new-dimension
               | (singleton e) | (e ∪ e) | (e ▹ e) | (e ≽ e @ e) | (e e ⊢ e ≽ e) | (fold e e e)
               | (top? e) | (bottom? e) | (proj? e) | (pcpl e) | (dim e) | (left e) | (right e) | (label e)
         τ ::= .... | Prin | Dim | Del | DelSet
       act ::= (action x : τ ([y : τ] ...) ce ae)
        ce ::= (check: cee add: cee remove: cee set!-principal: cee
                closure-principal: cee closure-delegations: cee)
        ae ::= (check: aee add: aee remove: aee scope: aee
                set!-principal: aee set-principal?: aee principal: aee)
       cee ::= e | (authlet (x cee) cee) | current-principal | current-delegations
       aee ::= e | (authlet (x aee) aee) | current-principal | current-delegations
               | closure-principal | closure-delegations
a, b, c, d ::= (prim P) | ⊤ | ⊥ | (a ▹ α) | (a ∨ b) | (a ∧ b) | (W → a) | (W ← a)
       del ::= (v ≽ v @ v)
         W ::= { del ... }
         E ::= .... | (singleton E) | (E ∪ e) | (v ∪ E) | (E ▹ e) | (v ▹ E) | (E ∨ e) | (v ∨ E) | (E ∧ e) | (v ∧ E)
               | (E e ⊢ e ≽ e) | (v E ⊢ e ≽ e) | (v v ⊢ E ≽ e) | (v v ⊢ v ≽ E)
               | (fold E e e) | (fold v E e) | (fold v v E) | (top? E) | (bottom? E) | (proj? E)
               | (pcpl E) | (dim E) | (left E) | (right E) | (label E)
```

Figure 45: Syntax extensions for authorization contracts.

5.2.3 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 (Section 5.1). In this section, we model a pared-down version of `define-monitor` as an extension to the language model from Section 5.2.1. The extension, given in Figure 45, introduces new types, constructors, and operations for principals, delegations, and delegations sets, includ-

$$\frac{}{\Gamma;\Sigma \vdash \textsf{new-principal} : \textsf{Prin}} \qquad \frac{}{\Gamma;\Sigma \vdash (\textsf{prim}\ P) : \textsf{Prin}} \qquad \frac{}{\Gamma;\Sigma \vdash \top : \textsf{Prin}} \qquad \frac{}{\Gamma;\Sigma \vdash \bot : \textsf{Prin}}$$

$$\frac{}{\Gamma;\Sigma \vdash \textsf{new-dimension} : \textsf{Dim}} \qquad \frac{}{\Gamma;\Sigma \vdash (\textsf{dim}\ D) : \textsf{Dim}} \qquad \frac{\Gamma;\Sigma \vdash e_1 : \textsf{Prin} \quad \Gamma;\Sigma \vdash e_2 : \textsf{Dim}}{\Gamma;\Sigma \vdash (e_1 \triangleright e_2) : \textsf{Prin}}$$

$$\frac{\Gamma;\Sigma \vdash e_1 : \textsf{Prin} \quad \Gamma;\Sigma \vdash e_2 : \textsf{Prin}}{\Gamma;\Sigma \vdash (e_1 \vee e_2) : \textsf{Prin}} \qquad \frac{\Gamma;\Sigma \vdash e_1 : \textsf{Prin} \quad \Gamma;\Sigma \vdash e_2 : \textsf{Prin}}{\Gamma;\Sigma \vdash (e_1 \wedge e_2) : \textsf{Prin}} \qquad \frac{\Gamma;\Sigma \vdash e_1 : \textsf{DelSet} \quad \Gamma;\Sigma \vdash e_2 : \textsf{Prin}}{\Gamma;\Sigma \vdash (e_1 \leftarrow e_2) : \textsf{Prin}} \qquad \frac{\Gamma;\Sigma \vdash e_1 : \textsf{DelSet} \quad \Gamma;\Sigma \vdash e_2 : \textsf{Prin}}{\Gamma;\Sigma \vdash (e_1 \rightarrow e_2) : \textsf{Prin}}$$

$$\frac{\Gamma;\Sigma \vdash e : \textsf{Prin}}{\Gamma;\Sigma \vdash (\textsf{top?}\ e) : \textsf{Bool}} \qquad \frac{\Gamma;\Sigma \vdash e : \textsf{Prin}}{\Gamma;\Sigma \vdash (\textsf{bottom?}\ e) : \textsf{Bool}} \qquad \frac{\Gamma;\Sigma \vdash e : \textsf{Prin}}{\Gamma;\Sigma \vdash (\textsf{proj?}\ e) : \textsf{Bool}}$$

$$\frac{\Gamma;\Sigma \vdash e : \textsf{Prin}}{\Gamma;\Sigma \vdash (\textsf{pcpl}\ e) : \textsf{Prin}}$$

Figure 47: Types for authorization contracts.

ing an expression that evaluates an acts-for judgment: $(e\ e \vdash e \succcurlyeq e)$. Additional typing rules for the language extension are given in Figure 47 and Figure 48. The `define-monitor` form corresponds to the (actions $(act\ \ldots)\ e$) form of the extended model, where act is an action specification. Each action specification (action $x : \tau$ ([y : τ_y] ...) $ce\ ae$) has a name x, a set of arguments ([y : τ_y] ...), a type (τ), 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

$$\frac{\Gamma;\Sigma\vdash e : \mathsf{Prin}}{\Gamma;\Sigma\vdash (\mathsf{dim}\ e) : \mathsf{Dim}} \qquad \frac{\Gamma;\Sigma\vdash e : \mathsf{Del}}{\Gamma;\Sigma\vdash (\mathsf{left}\ e) : \mathsf{Prin}} \qquad \frac{\Gamma;\Sigma\vdash e : \mathsf{Del}}{\Gamma;\Sigma\vdash (\mathsf{right}\ e) : \mathsf{Prin}} \qquad \frac{\Gamma;\Sigma\vdash e : \mathsf{Del}}{\Gamma;\Sigma\vdash (\mathsf{label}\ e) : \mathsf{Prin}}$$

$$\frac{\Gamma;\Sigma\vdash e : \mathsf{Del} \quad \ldots}{\Gamma;\Sigma\vdash \{\ e\ \ldots\ \} : \mathsf{DelSet}} \qquad \frac{\Gamma;\Sigma\vdash e : \mathsf{Del}}{\Gamma;\Sigma\vdash (\mathsf{singleton}\ e) : \mathsf{DelSet}} \qquad \frac{\begin{array}{c}\Gamma;\Sigma\vdash e_1 : \mathsf{DelSet}\\ \Gamma;\Sigma\vdash e_2 : \mathsf{DelSet}\end{array}}{\Gamma;\Sigma\vdash (e_1 \cup e_2) : \mathsf{DelSet}}$$

$$\frac{\begin{array}{c}\Gamma;\Sigma\vdash e_1 : \mathsf{DelSet}\\ \Gamma;\Sigma\vdash e_2 : \mathsf{DelSet}\end{array}}{\Gamma;\Sigma\vdash (e_1 - e_2) : \mathsf{DelSet}} \qquad \frac{\begin{array}{c}\Gamma;\Sigma\vdash e_1 : \mathsf{DelSet}\\ \Gamma;\Sigma\vdash e_2 : \mathsf{Prin}\\ \Gamma;\Sigma\vdash e_3 : \mathsf{Prin}\\ \Gamma;\Sigma\vdash e_4 : \mathsf{Prin}\end{array}}{\Gamma;\Sigma\vdash (e_1\ e_2 \vdash e_3 \succcurlyeq e_4) : \mathsf{Bool}}$$

$$\frac{\begin{array}{c}\Gamma;\Sigma\vdash ce \quad \ldots\\ \Gamma;\Sigma\vdash ae \quad \ldots\\ \Gamma[x \mapsto (\tau_x\ \mathsf{contract}), \ldots];\Sigma\vdash e : \tau\end{array}}{\Gamma;\Sigma\vdash (\mathsf{actions}\ ((\mathsf{action}\ x : \tau_x\ ([\mathsf{y} : \tau_y]\ \ldots)\ ce\ ae)\ \ldots)\ e) : \tau} \qquad \frac{\begin{array}{c}\Gamma;\Sigma\vdash e_1 : (\mathsf{Del} \to (\tau \to \tau))\\ \Gamma;\Sigma\vdash e_2 : \mathsf{DelSet}\\ \Gamma;\Sigma\vdash e_3 : \tau\end{array}}{\Gamma;\Sigma\vdash (\mathsf{fold}\ e_1\ e_2\ e_3) : \tau}$$

$$\frac{\begin{array}{c}\Gamma;\Sigma\vdash e_1 : (\mathsf{Unit} \to \mathsf{Bool})\\ \Gamma;\Sigma\vdash e_2 : (\tau\ \mathsf{param})\\ \Gamma;\Sigma\vdash e_3 : (\mathsf{Unit} \to \tau)\end{array}}{\Gamma;\Sigma\vdash (e_1 \Rightarrow e_2 \leftarrow e_3)}$$

Figure 47: Types for authorization contracts (Continued).

$$\frac{\begin{array}{c}\Gamma; \Sigma \vdash \mathit{cee}_1 : \textsf{Del} \\ \Gamma; \Sigma \vdash \mathit{cee}_2 : \textsf{DelSet} \\ \Gamma; \Sigma \vdash \mathit{cee}_3 : \textsf{DelSet} \\ \Gamma; \Sigma \vdash \mathit{cee}_4 : \textsf{Prin} \\ \Gamma; \Sigma \vdash \mathit{cee}_5 : \textsf{Prin} \\ \Gamma; \Sigma \vdash \mathit{cee}_6 : \textsf{DelSet}\end{array}}{\begin{array}{l}\Gamma; \Sigma \vdash (\textsf{check: } \mathit{cee}_1 \textsf{ add: } \mathit{cee}_2 \textsf{ remove: } \mathit{cee}_3 \textsf{ set!-principal: } \mathit{cee}_4 \\ \qquad \textsf{closure-principal: } \mathit{cee}_5 \textsf{ closure-delegations: } \mathit{cee}_6)\end{array}}$$

$$\frac{\begin{array}{c}\Gamma; \Sigma \vdash \mathit{aee}_1 : \textsf{Del} \\ \Gamma; \Sigma \vdash \mathit{aee}_2 : \textsf{DelSet} \\ \Gamma; \Sigma \vdash \mathit{aee}_3 : \textsf{DelSet} \\ \Gamma; \Sigma \vdash \mathit{aee}_4 : \textsf{DelSet} \\ \Gamma; \Sigma \vdash \mathit{aee}_5 : \textsf{Prin} \\ \Gamma; \Sigma \vdash \mathit{aee}_6 : \textsf{Bool} \\ \Gamma; \Sigma \vdash \mathit{aee}_7 : \textsf{Prin}\end{array}}{\begin{array}{l}\Gamma; \Sigma \vdash (\textsf{check: } \mathit{aee}_1 \textsf{ add: } \mathit{aee}_2 \textsf{ remove: } \mathit{aee}_3 \textsf{ scope: } \mathit{aee}_4 \\ \qquad \textsf{set!-principal: } \mathit{aee}_5 \textsf{ set-principal?: } \mathit{aee}_6 \textsf{ principal: } \mathit{aee}_7)\end{array}} \qquad \frac{\begin{array}{c}\Gamma; \Sigma \vdash \mathit{cee}_1 : \tau_1 \\ \Gamma[x \mapsto \tau_1]; \Sigma \vdash \mathit{cee}_2 : \tau_2\end{array}}{\Gamma; \Sigma \vdash (\textsf{authlet } (x\ \mathit{cee}_1)\ \mathit{cee}_2) : \tau_2}$$

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

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

Figure 48: Types for authorization contracts (Continued).

particular it describes how to modify each part of the authority environment. Its field check: accepts a delegation ($a \succcurlyeq b$ @ c) which will be used to construct an acts-for judgment $W\,;\, c \vdash a \succcurlyeq b$, where W is the current delegation set. 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 term for the `#:on-apply` hook has the form

(check: *aee* add: *aee* remove: *aee* scope: *aee* set!-principal: *aee* set-principal?: *aee* principal: *aee*)

and specifies what the authorization contract should do upon a call of the contracted function. Similar to *ce* terms, it allows the configuration of the contract's behavior with seven fields. As before, check: accepts an delegation and raises a contract error if the corresponding acts-for judgment 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 ac-

```
translate⟦(actions (act ...) e), $x_p$, $x_d$, $x_s$, $x_{cp}$, $x_{cd}$, $x_{curp}$, $x_{curd}$⟧ =
  (let ($x_p$ (make-parameter ⊤))
    (let ($x_d$ (make-parameter { }))
      (let ($x_s$ (make-parameter { }))
        (let ($x_{cp}$ (make-parameter ⊤))
          (let ($x_{cd}$ (make-parameter { }))
            (let ($x_{curp}$ (ref ⊤))
              (let ($x_{curd}$ (ref { }))
                translate-actions⟦(act ...), $x_p$, $x_d$, $x_s$, $x_{cp}$, $x_{cd}$, $x_{curp}$, $x_{curd}$, e⟧)))))))
translate-actions⟦(), $x_p$, $x_d$, $x_s$, $x_{cp}$, $x_{cd}$, $x_{curp}$, $x_{curd}$, e⟧ =
  e
translate-actions⟦((action x : $\tau$ ([$x_a$ : $\tau_a$] ...) ce ae) act ...), $x_p$, $x_d$, $x_s$, $x_{cp}$, $x_{cd}$, $x_{curp}$, $x_{curd}$, e⟧ =
  (let (x translate-action⟦(action x : $\tau$ ([$x_a$ : $\tau_a$] ...) ce ae), $x_p$, $x_d$, $x_s$, $x_{cp}$, $x_{cd}$, $x_{curp}$, $x_{curd}$⟧)
    translate-actions⟦(act ...), $x_p$, $x_d$, $x_s$, $x_{cp}$, $x_{cd}$, $x_{curp}$, $x_{curd}$, e⟧)
translate-action⟦(action x : $\tau$ ([$x_a$ : $\tau_a$] [$x_{a\_2}$ : $\tau_{a\_2}$] ...) ce ae), $x_p$, $x_d$, $x_s$, $x_{cp}$, $x_{cd}$, $x_{curp}$, $x_{curd}$⟧ =
  (λ (y : $\tau_y$)
    translate-action⟦(action x : $\tau$ ([$x_{a\_2}$ : $\tau_{a\_2}$] ...) ce ae), $x_p$, $x_d$, $x_s$, $x_{cp}$, $x_{cd}$, $x_{curp}$, $x_{curd}$⟧)
translate-action⟦(action x : $\tau$ () ce ae), $x_p$, $x_d$, $x_s$, $x_{cp}$, $x_{cd}$, $x_{curp}$, $x_{curd}$⟧ =
  (ctx/c : $\tau$
        translate-ce-check⟦ce, $x_p$, $x_d$, $x_s$, $x_{curp}$, $x_{curd}$⟧
        translate-ce-cp⟦ce, $x_p$, $x_d$, $x_{cp}$, $x_{curp}$, $x_{curd}$⟧
        translate-ce-cd⟦ce, $x_{cd}$, $x_{curp}$, $x_{curd}$⟧
        translate-ae-check⟦ae, $x_p$, $x_d$, $x_s$, $x_{cp}$, $x_{cd}$, $x_{curp}$, $x_{curd}$⟧
        translate-ae-s⟦ae, $x_p$, $x_s$, $x_d$, $x_{curp}$, $x_{curd}$, $x_{cp}$, $x_{cd}$⟧
        translate-ae-p⟦ae, $x_p$, $x_{curp}$, $x_{curd}$, $x_{cp}$, $x_{cd}$⟧)
```

Figure 49: Compiling authorization contracts to context contracts.

cess the principal and delegations closed over by the contract with terms closure-principal and closure-delegations.

To give a detailed semantics for actions terms, we use the compilation function given in Figure 49 that replaces actions 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 actions term generates a fresh set of parameters, preventing separately defined monitors from interfering with each other. Each action term within the actions term compiles to a single context contract that closes over the fresh parameters. The hooks provided for each action are compiled

```
translate-ce-check⟦(check: $cee_1$ add: $cee_2$ remove: $cee_3$ set!-principal: $cee_4$ =
                      closure-principal: $cee_5$ closure-delegations: $cee_6$),
                     $x_p$, $x_d$, $x_s$, $x_{curp}$, $x_{curd}$⟧
    (λ ($x$ : Unit)
      (let ($x$ ($x_{curp}$ := (? $x_p$)))
        (let ($x$ ($x_{curd}$ := ((? $x_d$) ∪ (? $x_s$))))
          (let (tocheck translate-cee⟦$cee_1$, $x_{curp}$, $x_{curd}$⟧)
            (let ($x_{c\_l}$ (left tocheck))
              (let ($x_{c\_r}$ (right tocheck))
                (let ($x_{c\_p}$ (label tocheck))
                  ((! $x_{curd}$) $x_{c\_p} \vdash x_{c\_l} \succcurlyeq x_{c\_r}$))))))))
translate-ce-cp⟦(check: $cee_1$ add: $cee_2$ remove: $cee_3$ set!-principal: $cee_4$ =
                   closure-principal: $cee_5$ closure-delegations: $cee_6$),
                  $x_p$, $x_d$, $x_{cp}$, $x_{curp}$, $x_{curd}$⟧
    ((λ ($x$ : Unit) #t) ⇒
                      $x_{cp}$
                      ← (λ ($x$ : Unit)
                           (let (to-add translate-cee⟦$cee_2$, $x_{curp}$, $x_{curd}$⟧)
                             (let (to-remove translate-cee⟦$cee_3$, $x_{curp}$, $x_{curd}$⟧)
                               (let (setprin translate-cee⟦$cee_4$, $x_{curp}$, $x_{curd}$⟧)
                                 (let ($x$ ($x_d$ ← (((! $x_{curd}$) ∪ to-add) - to-remove)))
                                   (let ($x$ ($x_p$ ← setprin))
                                     translate-cee⟦$cee_5$, $x_{curp}$, $x_{curd}$⟧)))))))
translate-ce-cd⟦(check: $cee_1$ add: $cee_2$ remove: $cee_3$ set!-principal: $cee_4$ =
                   closure-principal: $cee_5$ closure-delegations: $cee_6$),
                  $x_{cd}$, $x_{curp}$, $x_{curd}$⟧
    ((λ ($x$ : Unit) #t) ⇒
                      $x_{cd}$
                      ← (λ ($x$ : Unit)
                           translate-cee⟦$cee_6$, $x_{curp}$, $x_{curd}$⟧))
translate-cee⟦$e$, $x_{curp}$, $x_{curd}$⟧ =
    $e$
translate-cee⟦(authlet ($x$ $cee_1$) $cee_2$), $x_{curp}$, $x_{curd}$⟧ =
    (let ($x$ translate-cee⟦$cee_1$, $x_{curp}$, $x_{curd}$⟧)
      translate-cee⟦$cee_2$, $x_{curp}$, $x_{curd}$⟧)
translate-cee⟦current-principal, $x_{curp}$, $x_{curd}$⟧ =
    (! $x_{curp}$)
translate-cee⟦current-delegations, $x_{curp}$, $x_{curd}$⟧ =
    (! $x_{curd}$)
```

Figure 50: Compiling *ce* expressions.

```
translate-ae-check⟦(check: $aee_1$ add: $aee_2$ remove: $aee_3$ scope: $aee_4$                 =
                    set!-principal: $aee_5$ set-principal?: $aee_6$ principal: $aee_7$),
                    $x_p$, $x_d$, $x_s$, $x_{cp}$, $x_{cd}$, $x_{curp}$, $x_{curd}$⟧
    (λ ($x$ : Unit)
      (let ($x$ ($x_{curp}$ := (? $x_p$)))
        (let ($x$ ($x_{curd}$ := ((? $x_d$) ∪ (? $x_s$))))
          (let (clop (? $x_{cp}$))
            (let (clod (? $x_{cd}$))
              (let (tocheck translate-aee⟦$aee_1$, $x_{curp}$, $x_{curd}$, $x_{cp}$, $x_{cd}$⟧)
                (let ($x_{c\_l}$ (left tocheck))
                  (let ($x_{c\_r}$ (right tocheck))
                    (let ($x_{c\_p}$ (label tocheck))
                      ((! $x_{curd}$) $x_{c\_p} \vdash x_{c\_l} \succcurlyeq x_{c\_r}$))))))))))
translate-ae-p⟦(check: $aee_1$ add: $aee_2$ remove: $aee_3$ scope: $aee_4$                 =
                set!-principal: $aee_5$ set-principal?: $aee_6$ principal: $aee_7$),
                $x_p$, $x_{curp}$, $x_{curd}$, $x_{cp}$, $x_{cd}$⟧
    ((λ ($x$ : Unit)
       translate-aee⟦$aee_6$, $x_{curp}$, $x_{curd}$, $x_{cp}$, $x_{cd}$⟧) ⇒
                                                   $x_p$
                                                   ← (λ ($x$ : Unit)
                                                        translate-aee⟦$aee_7$, $x_{curp}$, $x_{curd}$, $x_{cp}$, $x_{cd}$⟧))
translate-ae-s⟦(check: $aee_1$ add: $aee_2$ remove: $aee_3$ scope: $aee_4$                 =
                set!-principal: $aee_5$ set-principal?: $aee_6$ principal: $aee_7$),
                $x_p$, $x_s$, $x_d$, $x_{curp}$, $x_{curd}$, $x_{cp}$, $x_{cd}$⟧
    ((λ ($x$ : Unit) #t) ⇒
                  $x_s$
                  ← (λ ($x$ : Unit)
                       (let (to-add translate-aee⟦$aee_2$, $x_{curp}$, $x_{curd}$, $x_{cp}$, $x_{cd}$⟧)
                         (let (to-remove translate-aee⟦$aee_3$, $x_{curp}$, $x_{curd}$, $x_{cp}$, $x_{cd}$⟧)
                           (let (news translate-aee⟦$aee_4$, $x_{curp}$, $x_{curd}$, $x_{cp}$, $x_{cd}$⟧)
                             (let ($x$ ($x_p$ ← translate-aee⟦$aee_5$, $x_{curp}$, $x_{curd}$, $x_{cp}$, $x_{cd}$⟧))
                               (let ($x$ ($x_d$ ← (((! $x_{curd}$) ∪ to-add) - to-remove)))
                                 news)))))))
translate-aee⟦$e$, $x_{curp}$, $x_{curd}$, $x_{cp}$, $x_{cd}$⟧ =
    $e$
translate-aee⟦(authlet ($x$ $aee_1$) $aee_2$), $x_{curp}$, $x_{curd}$, $x_{cp}$, $x_{cd}$⟧ =
    (let ($x$ translate-aee⟦$aee_1$, $x_{curp}$, $x_{curd}$, $x_{cp}$, $x_{cd}$⟧)
      translate-aee⟦$aee_2$, $x_{curp}$, $x_{curd}$, $x_{cp}$, $x_{cd}$⟧)
translate-aee⟦current-principal, $x_{curp}$, $x_{curd}$, $x_{cp}$, $x_{cd}$⟧ =
    (! $x_{curp}$)
translate-aee⟦current-delegations, $x_{curp}$, $x_{curd}$, $x_{cp}$, $x_{cd}$⟧ =
    (! $x_{curd}$)
translate-aee⟦closure-princpal, $x_{curp}$, $x_{curd}$, $x_{cp}$, $x_{cd}$⟧ =
    (? $x_{clop}$)
translate-aee⟦closure-delegations, $x_{curp}$, $x_{curd}$, $x_{cp}$, $x_{cd}$⟧ =
    (? $x_{clod}$)
```

Figure 51: Compiling *ae* expressions.

into a set of guarded parameterization expressions that access and update the parameters that make up the authority environment.

The hooks for defining actions are sufficiently flexible to implement a variety of access control mechanisms (Section 5.3). 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 discretionary access control policy would be forgotten when a `setuid` function executes.

5.3 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 inspection [123], history-based access control [2], and object capabilities [80]. Before delving into

```
(define-monitor monitor-name
  (monitor-interface
    action-name ... extra-name ...)
  (monitor-syntax-interface
    syntax-name ...)
  (action
    [action-name (action-var ...)
      #:on-create on-create-hook
      #:on-apply  on-apply-hook]
      ...)
  (extra
    (define extra-name extra-body)
    ...)
  (syntax
    (define-syntax syntax-name syntax-body)
    ...))
```

Figure 52: The `define-monitor` form.

these monitors, we further explain `define-monitor`, the main linguistic tool that our framework provides.

5.3.1 The `define-monitor` form

Figure 52 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 5.2.3. 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 35 is an acts-for query that is always satisfied.

5.3.2 A stack inspection monitor

In stack inspection [123], 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* [123], 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. For example, Java's stack inspection implementation provides a primitive `AccessController.getContext()` that encapsulates the access rights of the context where it is called.

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. A monitor enforcing stack inspection is given in Figure 53.

In the 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 contains permissions enabled for this frame. The `active` projection represents permissions that would satisfy a privilege check, that is, those permissions which are both enabled and statically granted. An access control check for a particular permission succeeds if the `active` projection of the current frame acts for the corresponding projection of the `⊤` 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:

```
(define-monitor stack-inspection
  (monitor-interface make-permission permission? check-permission/c do-privileged/c
                     context/c unprivileged/c privileged/c coerce-to-unprivileged)
  (monitor-syntax-interface define/rights)
  (action #:search (list use-static search-delegates-left)
    [check-permission/c (perm)
     #:on-create (do-create)
     #:on-apply
     (do-apply #:check (≥@ (▷ current-principal active) (▷ ⊤ perm) (▷ ⊤ perm)))]
    [privileged/c (perms)
     #:on-create (do-create #:check (≥@ current-principal ⊤ ⊤))
     #:on-apply
     (let* ([callee (pcpl (gensym 'frame))]
            [static-principal
             (normalize (disj (list->set (map (λ (p) (▷ ⊤ p)) perms))))]
            [scoped-delegations
             (list (≥@ (▷ callee static) static-principal ⊤)
                   (≥@ (▷ callee enable) (▷ current-principal active) current-principal)
                   (≥@ (▷ callee active) (∨ (▷ callee enable) (▷ callee static)) callee))])
       (do-apply #:add-scoped scoped-delegations #:set-principal callee))]
    [do-privileged/c
     #:on-create (do-create)
     #:on-apply
     (let ([enable (≥@ (▷ current-principal enable) (▷ current-principal static)
                       current-principal)])
       (do-apply #:add-scoped (list enable)))]
    [context/c
     #:on-create (do-create #:add-lifetime (list (≥@ (← ⊤ current-delegations) ⊤ ⊤)))
     #:on-apply
     (let* ([callee (pcpl (gensym 'frame))]
            [closure-pcpl (→ closure-principal closure-delegations)]
            [scoped-delegations
             (list (≥@ (▷ callee static) (▷ closure-pcpl active) ⊤)
                   (≥@ (▷ callee enable) (▷ current-principal active) current-principal)
                   (≥@ (▷ callee active) (∨ (▷ callee enable) (▷ callee static)) callee))])
       (do-apply #:add-scoped scoped-delegations #:set-principal callee))]

    [unprivileged/c
     #:on-create (do-create)
     #:on-apply  (do-apply #:set-principal ⊥)])
  (extra
    ...
    (define coerce-to-unprivileged
      (make-contract #:name "coerce-to-unprivileged"
                     #:projection
                     (λ (blame) (λ (val)
                       (cond
                        [((disjoin privileged/c? unprivileged/c? context/c?) val) val]
                        [(procedure? val) (((contract-projection unprivileged/c) blame) val)]
                        [else val]))))))

  (syntax
    ...
    (define-syntax define/rights ...)))
```

Figure 53: A stack inspection monitor.

```
(≥@ (▷ 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, 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 mirrors the information that can be found by walking the stack in a language with built-in support for stack inspection. Action `check-permission/c` checks only that the `active` projection of the current principal acts for all of the requested permissions. However, unlike security-passing style implementations of stack inspection [124], checking this acts-for relation requires validating delegations for each frame on the call stack between when the permission was enabled and when `check-permission/c` was called.

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 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` defined in its `syn-tax` 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 $\perp$ 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.

Figure 54 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`.

```
(require authorization-contracts authorization-
contracts/monitor/stack-inspection)

(run stack-inspection)

(define filesys (make-permission 'filesys))
(define net     (make-permission 'net))

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

(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;
 (▷ frame98139 active) ⋡ (▷ ⊤ filesys) @ (▷ ⊤ filesys)
  in: the 1st conjunct of
      the 1st conjunct of
      (and/c
       (and/c check-permission/c privileged/c)
       (membrane/c
        "coerce-to-unprivileged"
        "coerce-to-unprivileged"))
  contract from: (definition read-file)
  blaming: top-level
   (assuming the contract is correct)
  at: eval:6.0
```

Figure 54: Using the stack inspection monitor.

5.3.3 A history-based access control monitor

Abadi and Fournet [2] observe that stack inspection fails to protect against attacks where the influence of untrusted code is no longer apparent from the call stack. As a remedy, they propose history-based access control (HBAC). In HBAC, the rights of an execution context depend not just on the rights of code currently on the execution stack, but also on the rights of all code that has previously been executed. HBAC has two primitives: `grant` and `accept`. The `grant` primitive has the same behavior as the `do-privileged` operator we implement for stack inspection. `accept` allows a component to take responsibility for code in its dynamic extent. After `accept` returns, it restores any privileges that were present before it was invoked.

Our implementation of a monitor for HBAC is very similar to the monitor for stack inspection. The primary difference between the two monitors is the definition of action `privileged/c`. In the HBAC monitor, in addition to initializing the three projections for the new frame, its `#:on-apply` event walks the call stack by reading the current delegations. For every frame on the call stack, it adds a disjunct with the callee's static permissions to the delegation that granted permissions from that frame's caller. For example, the delegation:

```
(≥@ (▷ parent enable) (▷ grandparent active) grandparent)
```

is replaced with the delegation:

```
(≥@ (▷ parent enable) (∨ (▷ grandparent active) (▷ callee static))
    grandparent)
```

This means that future attempts to use the `parent` frame's permissions will be restricted to whatever rights the `callee` had, unless the `parent` frame specifically vouches for an action by enabling its own permissions.

We define `accept/c` in the `extra` section. `accept/c` uses the `accept-context/c` action to create an authority closure around its continuation. When the function `accept/c` wraps returns, `accept/c` invokes this continuation, restoring the authority environment before the wrapped function was called.

5.3.4 An object capability monitor

Authority closures are closely related to the idea of capabilities. A capability both designates a resource and confers the authority necessary to use it [20]. An authority closure designates resources (those accessed by the wrapped function) and captures the authority that should be used to access those resources. Any code that can invoke an authority closure can exercise the authority of the closure, though that authority is attenuated by the functionality of the closure itself. For example, in our web application example, `login` is a capability that allows any code that invokes it to use the "root" user's authority; however, the implementation of the `login` function ensures that this authority can only be used to switch to a different user after supplying the correct password.

Recall that in an object-capability language, all sensitive resources are represented as objects and access to those objects is controlled by structuring the language to limit the ways that objects acquire references to other objects [80]. Specifically, an object-capability language allows an object to acquire a reference to another object only in the following ways:

1. by initial conditions: two objects may reference each other before a computation begins,
2. by parenthood: the creator of an object is initially the only object with a reference to it,
3. by endowment: an object can close over references to objects available in its parent's environment, and
4. by introduction: an object can receive references to other objects passed as arguments to its methods or returned from methods it invokes.

```
(define-monitor ocap
  (monitor-interface capability/c unprivileged-capability/c capability/c?
                     unprivileged-capability/c? coerce-to-unprivileged-capability)
  (action #:search (list search-caps search-delegates-left)
    [capability/c
     #:on-create
     (let* ([child (pcpl (gensym 'capability))]
            [parent current-principal]
            [parenthood (≥@ (▻ parent caps) (▻ child invoke) child)]
            [endowment
             (≥@ (▻ child caps) (→ (▻ parent caps) current-delegations) parent)]
            [validity   (≥@ (← parent current-delegations) parent parent)])
       (do-create #:add-lifetime (list parenthood endowment validity)
                  #:closure-principal child))
     #:on-apply
     (let ([introductions
            (map (λ (arg) (let ([arg-cap (cdr arg)])
              (≥@ (▻ closure-principal caps) (▻ arg-cap invoke) current-principal)))
              (filter id closure-args))]
           [return-hook (λ (results)
             (let ([result-introductions
                    (map (λ (res)
                      (make-lifetime (≥ (▻ current-principal caps) (▻ (cdr res) invoke))
                                     closure-principal (car res)))
                      (filter id results))])
               (do-return #:add result-introductions)))])
       (do-apply
        #:check (≥@ current-principal (▻ closure-principal invoke)
                (▻ closure-principal invoke))
        #:add-lifetime introductions
        #:set-principal closure-principal
        #:on-return return-hook))]
    [unprivileged-capability/c
     #:on-create
     (let* ([child (pcpl (gensym 'unprivileged))]
            [parent current-principal]
            [parenthood (≥@ (▻ parent caps) (▻ child invoke) child)])
       (do-create #:add-lifetime (list parenthood) #:closure-principal child))
     #:on-apply ; the same as #:on-apply for capability/c])
  (extra
   (define coerce-to-unprivileged-capability
     (make-contract #:name "coerce-to-unprivileged-capability"
                    #:projection (λ (blame) (λ (val)
                       (cond
                         [((disjoin capability/c? unprivileged-capability/c?) val) val]
                         [(procedure? val)
                          (((contract-projection unprivileged-capability/c) blame) val)]
                         [else val])))))))
```

Figure 55: An object capability monitor.

We built a monitor to enforce these restrictions. This monitor, shown in Figure 55, defines two actions, `capability/c` and `unprivileged-capability/c`, which is the same as the `capability/c`, but does not grant any initial authority. These actions enforce capability safety by creating a new principal for each capability and using delegations to specify when one capability has the authority to invoke another. Their `#:on-create` hooks handle parenthood and endowment and their `#:on-apply` hooks handle introduction. Parenthood amounts to a delegation:

```
(≥@ (▻ parent caps) (▻ child invoke) child)
```

Similarly, endowment closes over the current authority of the parent and grants it to the child:

```
(≥@ (▻ child caps) (→ (▻ parent caps) current-delegations) parent)
```

We must also assert that the parent authorizes the use of its closed-over delegations for the rest of the execution:

```
(≥@ (← parent current-delegations) parent parent)
```

Introduction is implemented by adding additional delegations granting callees authority over arguments they are passed, and vice versa for return values.

5.4 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 5.1.2, replaces authorization checks in a web application with authorization contracts. In addition to evaluating the expressiveness of authorization contracts in each case study, we evaluated the performance of our framework. 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.

5.4.1 Preventing cheating in a card game

We have used authorization contracts to enforce a security policy for an implementation of the card game Dominion[‡]. 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

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

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`.

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. The contract says that the function has three arguments: a game structure, a turn structure describing the actions taken by the player so far, and a play structure indicating the desired move. The function returns two results: an updated game structure and an updated turn structure. The #:auth keyword applies an additional contract to the entire function depending on the game argument, which is an authorization contract that changes the current player to whichever player appears first in the game structure.

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$\times$ at both the median and 99th percentile.

5.4.2 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 only be invoked 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 reinstates 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.

5.4.3 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 invoke a procedure to process the request, like the `login` function from Section 5.1.2. 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 mark particularly robust packages.

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.

5.5 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' seminal paper "Protection in Programming Languages" [82] 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 [80]. Wallach and Felten [124] 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. [87] use dynamic and lexical scoping to associate principals with executing code and closures in order to correctly enforce security policies on programs that mix JavaScript and ActionScript code.

Language-level security mechanisms Numerous language-level security mechanisms have been proposed to support extensible access control including stack inspection [123], history-based access control [2], capabilities [78, 80]. In Section 5.3, we show how several of these mechanisms relate to our framework by demonstrating how they can be implemented using authorization contracts.

Reference Monitors An alternative approach to language-level access control is 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 [99]. Inlined reference monitoring (IRM) weaves the implementation of a reference monitor into the program being monitored [31]. Many implementations of inlined reference monitoring rely on aspects to identify security relevant actions during program execution [9, 32, 33, 35, 54]. 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 [32], policies where the authority of code depends on application state require duplicating code, since relevant changes to the application's state must also be reflected in the state of the reference monitor. 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 for expressing access control policies [1, 16]. Authorization logics have been used to understand existing access control mechanisms, including Java stack inspection [124]. Aura [53] and Fine [114] implement access control using proof-carrying authentication, where proofs of formulas in an authorization logic are used as capabilities [5]. Our access control logic is inspired by the Flow-Limited Authorization Model [6], 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. [81] use contracts to constrain the use of capabilities in a secure shell scripting language. Dimoulas et al. [24] use contracts to control the flow of capabilities between components in object-capability languages. Heidegger et al. [49] 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. [26] introduce temporal higher-order contracts, which enforce that sequences of function calls and returns match a specification. Their system supports arbitrarily powerful monitors, but like inlined reference monitoring, provides limited support for writing complex access control policies like stack inspection or discretionary access control. Scholliers et al.'s computational contracts [100] 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.

Scoped aspects for security Dutchyn et al. [29] enforce simple access control policies with lexically and dynamically-scoped aspects. With additional aspect scoping mechanisms, Toledo et al. [120] 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.

6 Conclusion

Writing programs that correctly address security requirements is difficult, since those requirements cut across many different parts of the program and are intrinsically tied to the program's structure. This dissertation argues that

> *Higher-order software contracts are an effective mechanism to specify and enforce composable, easy-to-understand security properties.*

Moreover, it argues that higher-order software contracts make it easier to write and debug secure programs by expressing security requirements of components at their interfaces using languages designed explicitly for this purpose.

To validate these claims, I present two separate approaches to the use of software contracts for security. In the first, I demonstrate how contracts improve on existing design patterns for languages whose security is based on the object-capability paradigm. Using this approach, we developed a secure shell scripting language that offers its users a high assurance of security without unduly burdening script authors. In the second, I develop a new type of software contract called an authorization contract. Authorization contracts allow component authors to devise custom access control mechanisms that address application-specific security concerns.

References

[1] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A Calculus for Access Control in Distributed Systems. *ACM Transactions on Programming Languages and Systems* 15(4), pp. 706–734, 1993.

[2] Martín Abadi and Cédric Fournet. Access Control Based on Execution History. In *Proc. Network and Distributed System Security Symposium*, 2003.

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

[4] AppArmor. 1998. http://apparmor.net

[5] Andrew W. Appel and Edward W. Felten. Proof-carrying Authentication. In *Proc. ACM Conference on Computer and Communications Security*, 1999.

[6] Owen Arden, Jed Liu, and Andrew C. Myers. Flow-Limited Authorization. In *Proc. IEEE Computer Security Foundations Symposium*, 2015.

[7] John Barnes. *High Integrity Ada: the SPARK Approach*. Addison-Wesley Professional, 1997.

[8] Michael Barnett, Rustan K. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Proc. Construction and analysis of safe, secure, and interoperable smart devices*, pp. 49–69, 2004.

[9] Lujo Bauer, Jay Ligatti, and David Walker. Composing Security Policies with Polymer. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[10] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proc. ACM SIGOPS Symposium on Operating Systems Principles*, pp. 29–44, 2009.

[11] Joachim Berg and Bart Jacobs. The LOOP Compiler for Java and JML. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 299–312, 2001.

[12] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer* 32(7), pp. 38–45, 1999.

[13] Matthias Blume and David McAllester. Sound and complete models of contracts. *Journal of Functional Programming* 16(4), pp. 375–414, 2006.

[14] Alan C. Bomberger, A. Peri Frantz, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. The KeyKOS Nanokernel Architecture. In *Proc. USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, 1992.

[15] Lilian Burdy, Antoine Requet, and Jean-Louis Lanet. Java applet correctness: A developer-oriented approach. In *Proc. International Symposium of Formal Methods Europe*, pp. 422–439, 2003.

[16] Michael Burrows, Martín Abadi, and Roger M. Needham. A Logic of Authentication. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 426(1871), pp. 233–271, 1989.

[17] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys* 17(4), 1985.

[18] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevostos, Julien Signoles, and Boris Yakobowski. *Frama-c. Software Engineering and Formal Methods*, pp. 233–247, 2012.

[19] Olivier Danvy and Andrzej Filinski. Abstracting Control. In *Proc. ACM Conference on LISP and Functional Programming*, 1990.

[20] Jack B. Dennis and Earl C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM* 9(3), pp. 143–155, 1966.

[21] Christos Dimoulas. Foundations for Behavioral Higher-Order Contracts. PhD dissertation, Northeastern University, 2012.

[22] Christos Dimoulas, Robert Bruce Findler, and Matthias Felleisen. Option Contracts. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2013.

[23] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct Blame for Contracts: No More Scapegoating. In *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2011.

[24] Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. Declarative Policies for Capability Control. In *Proc. IEEE Computer Security Foundations Symposium*, 2014, © 2014 IEEE.

[25] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete Monitors for Behavorial Contracts. In *Proc. European Symposium on Programming*, pp. 211–230, 2012.

[26] Tim Disney, Cormac Flanagan, and Jay McCarthy. Temporal Higher-order Contracts. In *Proc. ACM SIGPLAN International Conference on Functional Programming*, pp. 176–188, 2011.

[27] Docker. 2013. https://www.docker.com

[28] Sophia Drossopoulou, James Noble, and Mark S. Miller. Swapsies on the Internet – First steps towards Reasoning about Risk and Trust in the Open World. In *Proc. ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2015.

[29] Christopher Dutchyn, David B. Tucker, and Shriram Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming* 63(3), pp. 207–239, 2006.

[30] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and Event Processes in the Asbestos Operating System. In *Proc. ACM Symposium on Operating Systems Principles*, 2005.

[31] Úlfar Erlingsson. The Inlined Reference Monitor Approach to Security Policy Enforcement. PhD dissertation, Cornell University, 2004.

[32] Úlfar Erlingsson and Fred B. Schneider. IRM Enforcement of Java Stack Inspection. In *Proc. IEEE Symposium on Security and Privacy*, 2000.

[33] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *Proc. IEEE Symposium on Security and Privacy*, 1999.

[34] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[35] Chen Feng and Grigore Rosu. Java-MOP: A Monitoring Oriented Programming Environment for Java. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2005.

[36] Robert Bruce Findler and Matthias Blume. Contracts as Pairs of Projections. In *Proc. International Symposium on Functional and Logic Programming*, pp. 226–241, 2006.

[37] Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In *Proc. ACM SIGPLAN International Conference on Functional Programming*, pp. 48–59, 2002.

[38] Matthew Flatt and PLT. Reference: Racket. PLT Inc., PLT-TR-2010-1, 2010. http://racket-lang.org/tr1/

[39] Martin Gasbichler and Michael Sperber. Processes vs. user-level threads in Scsh. In *Proc. Workshop on Scheme and Functional Programming*, 2002.

[40] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. Hails: Protecting Data Privacy in Untrusted Web Applications. In *Proc. USENIX Symposium on Operating Systems Design and Implementation*, pp. 47–60, 2012.

[41] Alain Giorgetti and Julien Groslambert. JAG: JML Annotation Generation for Verifying Temporal Properties. In *Proc. International Conference on Fundamental Approaches to Software Engineering*, pp. 373–376, 2006.

[42] Li Gong. A Secure Identity-Based Capability System. In *Proc. IEEE Symposium on Security and Privacy*, pp. 56–63, 1989.

[43] Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-Grained Interoperability through Contracts and Mirrors. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 609–624, 2005.

[44] Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-parametric Polymorphic Contracts. In *Proc. Symposium on Dynamic Languages*, pp. 29–40, 2007.

[45] Norman Hardy. KeyKOS architecture. *ACM SIGOPS Operating Systems Review* 19(4), pp. 8–25, 1985.

[46] Norman Hardy. The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review* 22(4), pp. 35–38, 1998.

[47] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. *ACM Computing Surveys* 44(3), 2012.

[48] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *Proc. USENIX Symposium on Operating Systems Design and Implementation*, pp. 165–181, 2014.

[49] Phillip Heidegger, Annette Bieniusa, and Peter Thiemann. Access Permission Contracts for Scripting Languages. In *Proc. AVM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.

[50] Gernot Heiser, Kevin Elphinstone, Ihor Kuz, Gerwin Klein, and Stefan M. Petters. Towards Trustworthy Computing Systems: Taking Microkernels to the Next Level. *ACM SIGOPS Operating Systems Review* 41(4), pp. 3–11, 2007.

[51] Bart Jacobs and Frank Piessens. The VeriFast program verifier. Department of Computer Science, KU Leuven, 2008.

[52] Suman Jana, Donald E. Porter, and Vitaly Shmatikov. TxBox: Building Secure, Efficient Sandboxes with System Transactions. In *Proc. IEEE Symposium on Security and Privacy*, 2011.

[53] Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. AURA: A Programming Language for Authorization and Audit. In *Proc. ACM SIGPLAN Conference on Functional Programming*, 2008.

[54] Micah Jones and Kevin W. Hamlen. Enforcing IRM Security Policies: Two Case Studies. In *Proc. IEEE Intelligence and Security Informatics Conference*, 2009.

[55] Murat Karaorman, Urs Hölzle, and John Bruno. jContractor: A Reflective Java Library to Support Design by Contract. University of California, Santa Barbara, TRCS98-31, 1998.

[56] Paul Ashley Karger. Improving security and performance of capability systems. PhD dissertation, University of Cambridge, 1988.

[57] Matthias Keil and Peter Thiemann. TreatJS: Higher-order contracts for JavaScript. arXiv:1504.08110, 2015.

[58] Taesoo Kim and Nickolai Zeldovich. Practical and Effective Sandboxing for Non-root Users. In *Proc. USENIX Annual Technical Conference*, pp. 139–144, 2013.

[59] Reto Kramer. iContract - The Java(tm) Design by Contract(tm) Tool. In *Proc. International Conference on Technology of Object-Oriented Languages and Systems*, 1998.

[60] Maxwell Krohn, Petros Efstathopoulos, Cliff Frey, Frans Kaashoek, Eddie Kohler, David Mazières, Robert Morris, Michelle Osborne, Steve VanDeBogart, and David Ziegler. Make Least Privilege a Right (Not a Privilege). In *Proc. Conference on Hot Topics in Operating Systems*, 2005.

[61] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, Frans Kaashoek, Eddie Kohler, and Robert Morris. Information Flow Control for Standard OS Abstractions. In *Proc. ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[62] Adam Lackorzynski and Alexander Warg. Taming Subsystems: Capabilities As Universal Access Control in L4. In *Proc. Workshop on Isolation and Integration in Embedded Systems*, pp. 25–30, 2009.

[63] Butler W. Lampson. A Note on the Confinement Problem. *Communications of the ACM* 16(10), pp. 613–615, 1973.

[64] Butler W. Lampson. Protection. *ACM SIGOPS Operating System Review* 8(1), pp. 18–24, 1974.

[65] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In *Proc. Behavioral Specifications of Businesses and Systems*, pp. 175–188, 1999.

[66] Jochen Liedtke. On μ-kernel construction. In *Proc. ACM Symposium on Operating Systems Principles*, pp. 237–250, 1995.

[67] Peter Loscocco and Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proc. FREENIX Track, USENIX Annual Technical Conference*, 2001.

[68] David Luckham and Friedrich W. Henke. An overview of ANNA - a specification language for ADA. *IEEE Software* 2(9), pp. 9–22, 1985.

[69] LXC. 2008. https://linuxcontainers.org

[70] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Object Capabilities and Isolation of Untrusted Web Applications. In *Proc. IEEE Symposium on Security and Privacy*, pp. 125–140, 2010.

[71] Dino Mandrioli and Bertrand Meyer. *Advances in Object-Oriented Software Engineering*. Prentice Hall, 1992.

[72] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming* 58(1), pp. 89–106, 2004.

[73] Adrian Mettler, David Wagner, and Tyler Close. Joe-E: A Security-Oriented Subset of Java. In *Proc. USENIX Symposium on Networked Systems Design and Implementation*, 2010.

[74] Bertrand Meye. *Eiffel: The Language*. Prentice Hall, 1992.

[75] Bertrand Meyer. Applying "Design by Contract". *Computer* 25(10), pp. 40–51, 1992.

[76] Mark S. Miller, James E. Donnelley, and Alan H. Karp. Delegating Responsibility in Digital Systems: Horton's "Who Done It?". In *Proc. USENIX Workshop on Hot Topics in Security*, 2007.

[77] Mark S. Miller, Chip Morningstar, and Bill Frantz. Capability-based Financial Instruments. In *Proc. International Conference on Financial Cryptography*, pp. 349–378, 2000.

[78] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized JavaScript. Google, 2008.

[79] Mark S. Miller, Ka-Ping Yee, and Jonathan Shapiro. Capability Myths Demolished. Johns Hopkins University, SRL2003-02, 2003.

[80] Mark Samuel Miller. Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. PhD dissertation, Johns Hopkins University, 2006.

[81] Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. Shill: A Secure Shell Scripting Language. In *Proc. USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[82] James H. Morris Jr. Protection in Programming Languages. *Communications of the ACM* 16(1), 1973.

[83] Toby Murray. Analysing the security properties of object-capability patterns. PhD dissertation, University of Oxford, 2010.

[84] Toby Murray and Duncan Grove. Non-Delegatable Authorities in Capability Systems. *Journal of Computer Security* 16(6), pp. 743–759, 2008.

[85] Peter G. Neumann and Richard J. Feiertag. PSOS revisited. In *Proc. Computer Security Applications Conference*, pp. 208–216, 2003.

[86] Mariela Pavlova, Gilles Barthe, Lilian Burdy, Marieke Huisman, and Jean-Louis Lanet. Enforcing High-Level Security Properties for Applets. In *Proc. International Conference on Smart Card Research and Advanced Applications, World Computer Congress, TC8/WG8.8 & TC11/WG11.2*, pp. 1–16, 2004.

[87] Phu H. Phung, Maliheh Monshizadeh, Meera Sridhar, Kevin W. Hamlen, and V. N. Venkatakrishnan. Between Worlds: Securing Mixed JavaScript/ActionScript Multi-party Web Content. *IEEE Transactions on Dependable and Secure Computing* 12(4), pp. 443–457, 2015.

[88] Reinhold Ploesch and Josef Pichler. Contracts: From Analysis to C++ Implementation. In *Proc. International Conference on Technology of Object-Oriented Languages and Systems*, 1999.

[89] Gordon D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science* 1(2), pp. 125–159, 1975.

[90] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science* 5(3), pp. 223–255, 1977.

[91] Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. Type-based Verification of Web Sandboxes. *Journal of Computer Security* 22(4), pp. 511–565, 2014.

[92] David D. Redell. Naming and Protection in Extensible Operating Systems. PhD dissertation, University of California at Berkeley, 1974.

[93] Jonathan Allen Rees. A Security Kernel Based on the Lambda-Calculus. PhD dissertation, Massachusetts Institute of Technology, 1995.

[94] John C. Reynolds. GEDANKEN - a simple typeless language based on the principle of completeness and the reference concept. *Communications of the ACM* 13(5), pp. 308–319, 1970.

[95] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society* 74(2), pp. 358–366, 1953.

[96] David S. Rosenblum. A Practical Approach to Programming with Assertions. *IEEE Transactions on Software Engineering* 21(1), pp. 19–31, 1995.

[97] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical Fine-grained Decentralized Information Flow Control. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 63–74, 2009.

[98] Jerome H. Saltzer. Protection and the Control of Information Sharing in Multics. *Communications of the ACM* 17(7), pp. 388–402, 1974.

[99] Fred B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security* 3(1), pp. 30–50, 2000.

[100] Christophe Schollier, Éric Tanter, and Wolfgang De Meuter. Computational Contracts. *Science of Computer Programming* 98(3), pp. 360–375, 2015.

[101] Mark Seaborn. PLASH: the Principle of Least Authority Shell. 2007. http://www.cs.jhu.edu/~seaborn/plash/html/

[102] Jonathan S. Shapiro, Michael Scott Doerrie, Eric Northup, Swaroop Sridhar, and Mark S. Miller. Towards a Verified, General-Purpose Operating System Kernel. In *Proc. NICTA Invitational Workshop on Operating System Verification*, pp. 1–19, 2004.

[103] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Proc. ACM Symposium on Operating Systems Principles*, pp. 170–185, 1999.

[104] Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. In *Proc. Workshop on Scheme and Functional Programming*, 2006.

[105] Jan Smans, Bart Jacobs, and Frank Piessens. Static verification of code access security policy compliance of .NET applications. In *Proc. International Conference on .NET*, pp. 1–12, 2005.

[106] Alfred Speissens. Patterns of Safe Collaboration. PhD dissertation, Catholic University of Louvain, 2007.

[107] Brad Spengler. Grsecurity. 2003. https://grsecurity.net

[108] Vincent St-Amour, Leif Andersen, and Matthias Felleisen. Feature-specific profiling. In *Proc. International Conference on Compiler Construction*, pp. 49–68, 2015.

[109] Guy Lewis Steele Jr. Macaroni is Better Than Spaghetti. In *Proc. Symposium on Artificial Intelligence and Programming Languages*, 1977.

[110] Guy Lewis Steele Jr. and Gerald Jay Sussman. The Revised Report on SCHEME: A Dialect of LISP. Massachusetts Institute of Technology Artificial Intelligence Laboratory, AIM-452, 1978.

[111] Marc Stiegler and Mark S. Miller. A Capablity Based Client: The DarpaBrowser. Combex, Inc., BAA-00-06-SNK, 2002.

[112] T. Stephen Strickland, Christos Dimoulas, Asumu Takikawa, and Matthias Felleisen. Contracts for First-Class Classes. *Transactions on Programming Languages and Systems* 35(3), pp. 11–58, 2013.

[113] T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and Impersonators: Run-time Support for Contracts on Higher-Order, Stateful Values. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 943–962, 2012.

[114] Nikhil Swamy, Juan Chen, and Ravi Chugh. Enforcing Stateful Authorization and Information Flow Policies in Fine. In *Proc. European Conference on Programming Languages and Systems*, 2010.

[115] Asumu Takikawa, Daniel Feltey, Earl Dean, Robert Bruce Findler, Matthew Flatt, Sam Tobin-Hochstadt, and Matthias Felleisen. Towards Practical Gradual Typing. In *Proc. European Conference on Object-Oriented Programming*, 2015.

[116] Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual Typing for First-Class Classes. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 793–810, 2012.

[117] Asumu Takikawa, T. Stephen Strickland, and Sam Tobin-Hochstadt. Constraining Delimited Control with Contracts. In *Proc. European Conference on Programming Languages and Systems*, pp. 229–248, 2013.

[118] Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 395–406, 2008.

[119] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as Libraries. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 132–141, 2011.

[120] Rodolfo Toledo, Angel Nunez, Eric Tanter, and Jacques Noye. Aspectizing Java Access Control. *IEEE Transactions on Software Engineering* 38(1), 2012.

[121] Kerry Trentelman and Marieke Huisman. *Extending JML specifications with temporal logic*. Algebraic Methodology and Software Technology, 2002.

[122] David Wagner and Dean Tribble. A Security Analysis of the Combex DarpaBrowser Architecture. Combex, Inc., 2002. http://www.combex.com/papers/darpa-review/

[123] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward Felten. Extensible Security Architectures for Java. In *Proc. ACM Symposium on Operating Systems Principles*, 1997.

[124] Dan S. Wallach and Edward W. Felten. Understanding Java Stack Inspection. In *Proc. IEEE Symposium on Security and Privacy*, pp. 52–63, 1998.

[125] Robert N. M. Watson. A Decade of OS Access-Control Extensibility. *Communications of the ACM* 56(2), pp. 52–63, 2013.

[126] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical Capabilities for UNIX. In *Proc. USENIX Security Symposium*, pp. 29–46, 2010.

[127] Robert N. M. Watson and Chris Vance. The TrustedBSD MAC framework: Extensible kernel access control for FreeBSD 5.0. In *Proc. USENIX Annual Technical Conference*, pp. 285–296, 2003.

[128] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making Information Flow Explicit in HiStar. In *Proc. USENIX Symposium on Operating Systems Design and Implementation*, 2006.