Declarative Policies for Capability Control

Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong
School of Engineering and Applied Sciences
Harvard University
Cambridge, MA, USA
{chrdimo,sdmoore,aslan,chong}@seas.harvard.edu

***Abstract*—In capability-safe languages, components can access a resource only if they possess a capability for that resource. As a result, a programmer can prevent an untrusted component from accessing a sensitive resource by ensuring that the component never acquires the corresponding capability. In order to reason about which components may use a sensitive resource it is necessary to reason about how capabilities propagate through a system. This may be difficult, or, in the case of dynamically composed code, impossible to do before running the system.**

To counter this situation, we propose extensions to capability-safe languages that restrict the use of capabilities according to declarative policies. We introduce two independently useful semantic security policies to regulate capabilities and describe language-based mechanisms that enforce them. *Access control policies* restrict which components may use a capability and are enforced using higher-order contracts. *Integrity policies* restrict which components may influence (directly or indirectly) the use of a capability and are enforced using an information-flow type system. Finally, we describe how programmers can dynamically and soundly combine components that enforce access control or integrity policies with components that enforce different policies or even no policy at all.

***Keywords*-Capabilities; Capability policies; Information-flow control; Language-based security.**

I. Introduction

Capabilities are a popular mechanism for managing authority in both historical and modern systems and languages (e.g., [1, 2, 3]). *Authority* is the right to use resources, and capabilities reify authority as objects or nonces (unforgeable tokens). *Capability safety* requires that a component must possess an appropriate capability in order to access a resource. *Capability-safe languages* enforce capability safety at the language level, ensuring that authority of language-level components is conveyed exclusively via capabilities. Appropriate design patterns for capabilities can enforce fine-grained application-specific access control requirements, including confinement and selective revocation [4].

However, in order to reason about which components may exercise a given authority it is necessary to reason about how capabilities and references to capabilities propagate through a system. This in turn requires reasoning about the functionality and implementation of the system. In the extreme, reasoning about the use of a single capability may be as complicated as reasoning about the implementation of an entire system [5].

This paper proposes two extensions to capability-safe languages that restrict the use of capabilities according to two independently useful kinds of declarative fine-grained policies: access control policies and integrity policies. Declarative policies for capabilities simplify reasoning about the correct use of capabilities by separating policy from implementation: the components that may use or influence the use of a given capability can be determined by examining the policy associated with the capability, without needing to examine the implementation of the entire system.

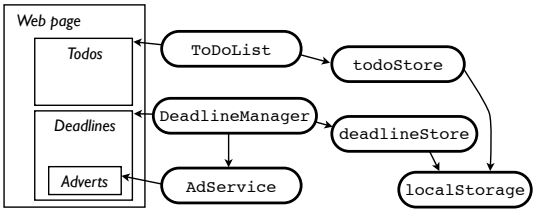

Fig. 1. A mashup for writing scientific papers. The page layout is shown on the left (which also depicts the DOM representation of the page). Rounded boxes are components; arrows indicate object references.

More specifically, access control policies restrict which components may use a capability; integrity policies restrict which components may influence the use of a capability. The two extensions cover non-overlapping security requirements and readily interoperate. In fact, the security guarantees can be soundly enforced even when components that use our extensions are composed with those that do not.

A. Example: securing a web mashup

Web mashups embed third-party code into a host web page. For mashups to be useful, the embedded code needs to share resources with the host page. Yet to avoid jeopardizing the security of the host page, the programmer of the host page needs to carefully restrict the authority of embedded code. To help programmers resolve this tension, tools for "taming" third-party JavaScript code, such as Caja [1] and JSand [6], use JavaScript proxy technology [7, 8, 9] to enforce the principles of object capabilities on mashups. Object-capability systems [10] treat all object references as capabilities and achieve capability safety by restricting object interactions to message passing. However, capability safety alone is not sufficient to guarantee security requirements of a mashup as we demonstrate with the following example.

Figure 1 illustrates the page layout and object references of a JavaScript mashup for authoring scientific papers. The

```
var deadlineDOM = ...;
var todoDOM = ...;
var vStore = new VirtualStore(localStorage);
var deadlineStore = vStore.make('deadlines');
var todoStore = vStore.make('todo');
var deadlineManager =
  new DeadlineManager(deadlineStore,deadlineDOM);
var todoList = new ToDoList(todoStore,todoDOM);

function VirtualStore(localStore) {
  var store = localStore;
  this.make = function (id) {
   return {
     put : function (key,value)
       { store.put(id + '.' + key, value); },
     get : function (key)
       { store.get(id + '.' + key); },
     del : function (key)
       { store.removeItem(id + '.' + key); }
   }; }
}
```

Listing 1. The host page delegates initial authority to the plugins.

mashup incorporates two plugins: a to-do list plugin, called `ToDoList`, and a plugin for managing upcoming conference deadlines, called `DeadlineManager`. The deadline manager also displays third-party advertisements via a component called `AdService`. Both plugins generate content on the web page, and thus have references to DOM nodes corresponding to portions of the page they use. `DeadlineManager` gives a reference to a descendant DOM node to `AdService` to allow it display advertisements.

Both plugins also need to maintain persistent state: the list of to-do items and upcoming deadlines. HTML5 offers a persistent local store via a simple key-value API, exposed in JavaScript as the `localStorage` object. To avoid giving the two components access to each other's entries in the local store, we can apply a standard object-capability design pattern that provides `ToDoList` and `DeadlineManager` with limited access to `localStorage`. Listing 1 sketches code to do so.[1] Constructor `VirtualStore` takes a capability for `localStorage` and returns an object that can make a new "namespace": an object that provides limited access to the local store by ensuring that all keys start with a common prefix. The host page makes one separate namespace (`deadlineStore` and `todoStore`) for each component.

Access control requirements. Consider the security requirement that neither `ToDoList` nor `DeadlineManager` has unrestricted access to the local store. Capability safety helps us reason about this security requirement, but by itself is not enough. What code must we trust in order to achieve this requirement? To start with, we rely on the implementation of `VirtualStore` to not propagate capability `localStorage` to its clients. In addition, if there is other code in the system that possesses capability `localStorage` and interacts with `ToDoList` or `DeadlineManager`, then we must also trust that this other code does not propagate the capability inappropriately. In sum, to prohibit `ToDoList` and `DeadlineManager` from using `localStorage` we may need to trust large portions of the code base.

From a high-level viewpoint, this is a problem of access control: components `ToDoList` and `DeadlineManager` should not be allowed to directly use (a reference of) the `localStorage` capability. In fact, the 2013 OWASP Top Ten project ranks direct uses of objects without appropriate access control checks the fourth most common security risk for web applications [11]. To mitigate these pervasive vulnerabilities, rather than *trusting large portions of the code base*, we introduce and enforce declarative access control policies that restrict which components may use a capability.

Integrity requirements. Consider an additional security requirement that `AdService` must not affect the content of the local store. If it could do so, it could perhaps create "supercookies" that uniquely identify a computer. Although enforcing the access control policy from the previous paragraph suffices to establish that `AdService` never directly accesses the local store, `AdService` interacts with `DeadlineManager`, which has limited access to the local store. Therefore `AdService` may indirectly affect the local store's contents. As before, in order to exclude this possibility, we may need to trust large amounts of code in the system: specifically any code that can access `localStorage` and can be influenced (directly or indirectly) by `AdService`.

This is a problem of information-flow control for integrity: component `AdService` should not influence key-value pairs in the local store, even though it needs to interact with components that use the store. Thus, we introduce and enforce information-flow control policies that ensure the integrity of uses of capabilities (such as the integrity of `localStorage` against the influence of `AdService`) *without needing to trust large portions of the code base*.

While in general integrity policies are stronger than access control policies, in many cases they may be too coarse because they conflate direct and indirect uses of capabilities. For instance, our mashup's security requires that `localStorage` adheres to both an access control and an integrity policy: `ToDoList` or `DeadLineManager` should not use `localStorage` and `AdService` should not influence uses of `localStorage`. Separating the two kinds of policies allows the specification of flexible security requirements for capabilities.

B. Overview of approach

We propose two extensions to capability-safe languages that allow programmers to annotate capabilities with declarative policies restricting their use. An *access control policy* is a white-list of components; when associated with a capability c, only white-listed components are allowed to *use* c. An *integrity policy* is also a white-list of components; when associated with a capability c, only white-listed components are allowed to (directly or indirectly) *influence the use* of c. For both access

[1] For clarity, we do not show calls to an API such as Caja that enforces capability safety. These calls do not change the structure of the mashup or the interaction pattern between plugins and the host web page. We assume that capability safety is enforced.

control and integrity, we give a semantic definition and present an enforcement mechanism that provably achieves security.

We enforce access control policies with *higher-order contracts* [12]. Contracts provide a run-time mechanism to enforce access control policies without modifying existing components of the program. In the event of an attempted policy violation (i.e., a component C tries to use a capability but C is not on that capability's white-list), the contract mechanism intervenes and prevents the use. Moreover, the contract mechanism accurately detects which component violates the policy [13], simplifying debugging and auditing.

To enforce integrity policies, we must track and control information-flow within the system. This requirement is beyond the grasp of higher-order contracts, so we use a security type-system [14, 15]: a well-typed component is guaranteed to use capabilities according to their integrity policies (i.e., only information from a capability's white-listed components influences the use of the capability).

These two independent enforcement mechanisms yield a gradual path for increasing the security of programs component by component. Programmers can start with a basic program that meets capability safety and then add access control policies to important capabilities and components. If beneficial, the programmer can invest more time to isolate the use of sensitive capabilities from external influences by adding integrity policies to sensitive capabilities and type annotations to sensitive components.

Our work makes the following contributions:

- We formalize capability safety using a simple calculus. This general model is a sound basis for exploring capabilities in a language-agnostic setting.
- We extend our model with declarative access control and integrity policies for capabilities. This greatly simplifies reasoning about the use of capabilities without needing to trust large portions of the code base.
- We prove that we can soundly enforce these declarative policies using higher-order contracts and a security type system. Moreover, we show that these mechanisms can be soundly composed.
- We use standard programming language abstractions and techniques to both model and enforce the policies. This bodes well for the practicality of both the security guarantees and enforcement mechanisms.

The rest of this paper is structured as follows. In Section II, we present a core model of a capability-safe language. Section III introduces access control policies for capabilities, states the semantic guarantee, and shows how contracts can enforce it. Section IV introduces integrity policies as a noninterference guarantee [16] and gives a mechanism for specifying and enforcing them. In Section V, we describe how higher-order contracts allow the sound composition of components with different policies, thus allowing correct enforcement of both access control policies and integrity policies within a single system, despite the presence of untrusted code. In Section VI we discuss how our model applies to real languages and how to implement our extensions to capability-safe languages. Section VII reviews related work and Section VIII concludes.

Terms	e	$=$	$v \mid x \mid e\ e \mid \mu x{:}\tau.e$
		$\mid$	$e+e \mid e-e \mid e \wedge e \mid e \vee e$
		$\mid$	$\texttt{zero?}(e) \mid \texttt{if}\ e\ e\ e$
		$\mid$	$\texttt{K}^{l}(^{k}\, e) \mid \texttt{new} \mid \texttt{use}(e)$
Values	v	$=$	$\texttt{b} \mid \lambda x{:}\tau.e$
	$\texttt{b}$	$=$	$0 \mid 1 \mid -1 \mid \ldots \mid \texttt{tt} \mid \texttt{ff}$
Labels	k, l, p, q	$\in$	$\mathbb{L}$
Types	τ	$=$	$\mathsf{Int} \mid \mathsf{Bool} \mid \tau \to \tau \mid \mathsf{Cap}$

Fig. 2. CapPCF: source syntax

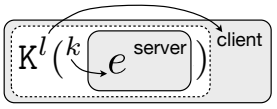

Fig. 3. CapPCF: components, component boundaries and labels

II. A Core Calculus for Capabilities

In this section we introduce CapPCF, an extension of Plotkin's strongly typed call-by-value PCF [17]. We show that CapPCF is capability-safe. This makes CapPCF a suitable foundation on which to develop further extensions in Sections III–V.

Figure 2 shows the source syntax of CapPCF. In addition to the standard constructs of PCF, CapPCF includes constructs to designate boundaries between components and to create and use capabilities.

Construct $\texttt{K}^{l}(^{k}\, e)$ defines a boundary between component e and the context in which it appears. Label k is the server label and identifies the component. Label l is the client label and identifies the context in which the term appears. Server and client labels on component boundaries represent security principals in CapPCF. Figure 3 depicts component composition and component boundaries in CapPCF.

We use component boundaries both to syntactically identify the origin of code (i.e., the component to which code belongs) and (in following sections) to help restrict the use of capabilities. In real languages, components may correspond to functions or objects, and labels may correspond to names or source locations of modules, classes, scripts or packages to which the component belongs. We further discuss components in real languages in Section VI but note here that component boundaries in CapPCF are a modeling technique. In particular, programmers do *not* need to explicitly add component boundary labels to their programs, as they can be constructed automatically from the structure of the code.

Capabilities in CapPCF are abstract objects drawn from an enumerable set. Term `new` creates a new abstract capability and term $\texttt{use}(e)$ uses a capability. To reason about how capabilities are used in a CapPCF program, we record the uses of capabilities in the *usage trace* of the program. Since our

$$
\begin{array}{lrcl}
\textbf{Values} & v & = & \ldots \mid \gamma \\
\textbf{Capabilities} & \gamma & = & c \mid \mathtt{G}^{l}\{^{k}\,\gamma\}
\end{array}
$$

Fig. 4. CapPCF: intermediate syntax

$$
\begin{array}{rcl}
E & = & E\ e \mid v\ E \mid E + e \mid v + E \mid E - e \mid v - E \\
& \mid & E \wedge e \mid v \wedge E \mid E \vee e \mid v \vee E \mid \mathtt{use}(E) \\
& \mid & \mathtt{zero?}(E) \mid \mathtt{if}\ E\ e\ e \mid \mathtt{K}^{l}(^{k}\,E\,)
\end{array}
$$

Fig. 5. CapPCF: evaluation contexts

$$
\begin{array}{lll}
\langle U, E[\cdots]\rangle & \rightarrow & \langle U, E[\cdots]\rangle \\
\hline
\mathtt{K}^{l}(^{k}\,v\,) & . & \lambda x.\mathtt{K}^{l}(^{k}\,v\ \mathtt{K}^{k}(^{l}\,x\,)\,) \text{ if } v = \lambda x.e \\
\mathtt{K}^{l}(^{k}\,\gamma\,) & . & \mathtt{G}^{l}\{^{k}\,\gamma\,\} \\
\mathtt{K}^{l}(^{k}\,\mathtt{b}\,) & . & \mathtt{b} \\
\hline
\langle U, E[\mathtt{new}]\rangle & \rightarrow & \langle U, E[c]\rangle \quad \text{where } c \text{ is fresh} \\
\langle U, E[\mathtt{use}(\gamma)]\rangle & \rightarrow & \langle U :: c, E[\gamma]\rangle \\
\text{where } \gamma = \mathtt{G}^{l_n}\{^{k_n} \ldots \mathtt{G}^{l_1}\{^{k_1}\,c\,\}\ldots\} & & \\
\hline
\end{array}
$$

Fig. 6. CapPCF: reduction semantics

goal is to investigate which parts of a program affect the use of capabilities, abstract capabilities associated with a general use operation are sufficient and reduce unnecessary clutter. Abstract capabilities in CapPCF can be thought of as modeling resources such as file-system entities, DOM nodes, or APIs that interact with the external environment.

Note that CapPCF is a strongly typed language with a standard type system, omitted here for conciseness. Types do not have any runtime significance and type annotations can be erased after type checking. Thus we ignore them throughout the paper when discussing examples or the semantics of CapPCF and its extensions unless needed.

A. Semantics of CapPCF

We define the behavior of CapPCF programs with a reduction semantics [18]. Figure 4 displays the intermediate syntax of the language as an extension of its source syntax and Figure 5 defines the evaluation contexts. A program state is a pair $\langle U, e\rangle$ of a usage trace U and a term e.

Reduction relation $\rightarrow$ for CapPCF (Figure 6) extends the standard rules of PCF with rules for the additional features of our language: component boundaries and capabilities.

When a component calls a function f that belongs to another component, the argument crosses the component boundary. To express this, component boundary $\mathtt{K}^{l}(^{k}\,f\,)$ around function f reduces to the term $\lambda x.\mathtt{K}^{l}(^{k}\,f\ \mathtt{K}^{k}(^{l}\,x\,)\,)$. Note that the argument x is now wrapped in a component boundary that marks x as originating from the client component: $\mathtt{K}^{k}(^{l}\,x\,)$. This ensures that when the function is applied, we correctly track the origin of the actual argument.

When a component boundary wraps around a capability $\mathtt{K}^{l}(^{k}\,\gamma\,)$, it reduces to a *guard* $\mathtt{G}^{l}\{^{k}\,\gamma\,\}$. Guards are values, and as such can migrate from one component to another. In essence, they establish a proxy that mediates uses of an abstract capability between the capability's server and its client. Thus, in CapPCF, capabilities include both abstract capabilities c and guards $\mathtt{G}^{l}\{^{k}\,\gamma\,\}$. In the remainder of the paper, unless otherwise indicated, we use the term "capability" to refer to both abstract capabilities and guards.

Unlike component boundaries around functions and capabilities, a component boundary around a base value, $\mathtt{K}^{l}(^{k}\,\mathtt{b}\,)$, lets client l absorb $\mathtt{b}$, i.e., the boundary disappears. This is because we are not concerned with tracking the origin of base values such as integers and booleans.

Reduction of term `new` creates a new capability c. Use of a capability ($\mathtt{use}(\gamma)$) records the use of capability c at the bottom of the stack of guards of γ (if any) by appending it to the usage trace U, and evaluates to γ.

B. Ownership annotations

As mentioned above, we want to prove capability safety for CapPCF, i.e., that a component can add a capability to the usage trace only if it uses a capability the component created or received from other components. To formalize this property, we apply the standard technique of Dimoulas et al. [19] for defining and establishing the correctness of contract systems. In particular, we use *ownership annotations* as a mechanism to describe which components possess which capabilities at every step of program execution. An ownership annotation on a term specifies the term's originating component. By establishing that ownership is invariant during execution of well-formed programs, we show that components can possess only capabilities that they create or obtain from other components through component boundaries. Note that ownership annotations are a modeling technique to establish capability safety, and do not place any burden on either the programmer or the runtime of a real language. The remainder of this section gives a brief overview of the proof technique.[2]

As a first step, we extend the syntax of CapPCF with ownership annotations $|e|^{l}$, which indicates that component l is the owner of term e. The extended syntactic categories of terms, values, and capabilities each include their annotated versions. Ownership annotations describe the ownership of terms independently of component boundaries. Yet, in order for these annotations to be meaningful, they need to agree with the notion of ownership derived from component boundaries in the source code. We express this as a well-formedness relation on source terms, $G; l \Vdash e$. Here, G is an environment that associates variables to ownership labels. Term e is well formed under environment G and owner l if its sub-terms are also well-formed under the same owner and environment, with two exceptions. The rules corresponding to these exceptions appear in Figure 7.

First, the owner of a term changes when checking the well-formedness of a term inside a component boundary. This is

[2]We refer the interested reader to Dimoulas et al. [13, 19] for a more detailed explanation.

$$\frac{G; k \Vdash e \qquad k \neq l}{G; l \Vdash \mathtt{K}^{l(k}\,|e|^{k}\,)}$$

$$\frac{G \uplus \{x : l\}; l \Vdash e}{G; l \Vdash \lambda x.e} \qquad \frac{G \uplus \{x : l\}; l \Vdash e}{G; l \Vdash \mu x.e}$$

Fig. 7. CapPCF with annotations: well-formed source terms

$$\begin{array}{rcl}
E^{l} & = & E^{l}\ e \mid v\ E^{l} \mid E^{l} + e \mid v + E^{l} \mid E^{l} - e \\
& \mid & v - E^{l} \mid E^{l} \wedge e \mid v \wedge E^{l} \mid E^{l} \vee e \mid v \vee E^{l} \\
& \mid & \mathtt{zero?}(E^{l}) \mid \mathtt{if}\ E^{l}\ e\ e \mid \mathtt{K}^{k(l}\,E^{l_o}\,) \\
& \mid & \mathtt{K}^{k(p}\,E^{l}\,), l \neq l_o \mid |E^{l_o}|^{l} \mid |E^{l}|^{p}, l \neq l_o \\
E^{l_o} & = & [\ ] \mid E^{l_o}\ e \mid v\ E^{l_o} \mid E^{l_o} + e \mid v + E^{l_o} \\
& \mid & E^{l_o} - e \mid v - E^{l_o} \mid E^{l_o} \wedge e \mid v \wedge E^{l_o} \\
& \mid & E^{l_o} \vee e \mid v \vee E^{l_o} \mid \mathtt{zero?}(E^{l_o}) \mid \mathtt{if}\ E^{l_o}\ e\ e
\end{array}$$

Fig. 8. CapPCF with annotations: evaluation contexts

$$\begin{array}{lll}
\langle U, E^{l}[\cdots]\rangle & \rightarrow \langle U, E^{l}[\cdots]\rangle & \\
\hline
\|\mathtt{n}_1\|^{l} + \|\mathtt{n}_2\|^{l} & .\ \mathtt{n} & \text{where } n_1 + n_2 = n \\
\mathtt{if}\ \|\mathtt{tt}\|^{l}\ e_1\ e_2 & .\ e_1 & \\
\|\lambda x.e\|^{l}\ \|v\|^{l} & .\ |\{|v|^{l}/x\}e|^{l} & \\
\mu x.e & .\ \{|\mu x.e|^{l}/x\}e & \\
\hline
\mathtt{K}^{l(k}\,\|\lambda x.e\|^{p}\,) & .\ \lambda x.\mathtt{K}^{l(k}\,\|\lambda x.e\|^{p}\ \mathtt{K}^{k(l}\,x\,)\,) & \\
\mathtt{K}^{l(k}\,\gamma\,) & .\ \mathtt{G}^{l}\{^{k}\,\gamma\,\} & \\
\mathtt{K}^{l(k}\,\|\mathtt{b}\|^{p}\,) & .\ \mathtt{b} & \\
\hline\hline
\langle U, E^{l}[\mathtt{new}]\rangle & \rightarrow \langle U, E^{l}[|c|^{l}]\rangle & \text{where } c \text{ is fresh} \\
\langle U, E^{l}[\mathtt{use}(\|\gamma\|^{l})]\rangle & \rightarrow \langle U :: c, E[\|\gamma\|^{l}]\rangle & \\
\text{where } \gamma = \mathtt{G}^{l_n}\{^{k_n}\,\|\ldots\mathtt{G}^{l_1}\{^{k_1}\,\|c\|^{m_1}\,\}\ldots\|^{m_n}\,\} & &
\end{array}$$

Fig. 9. CapPCF with annotations: reduction semantics

reflected in the first rule in Figure 7. The owner of the term that resides in the component boundary is the component with server label k. The rule also requires that the term inside the component comes with an explicit k ownership annotation. This enforces that ownership annotations and labels on component boundaries are in agreement.

Second, the bodies of (recursive) function abstractions are checked for well-formedness under an environment extended with the bound variable of the abstraction. This is reflected in the last two rules in Figure 7. In the new environment, the owner of the variable is the owner of the abstraction, reflecting that the context of the abstraction provides the argument applied to it. That is, whoever has the function can decide what arguments to provide to it.

Finally, the well-formedness rules forbid ownership annotations in any place except directly inside boundary terms, which tightens the correspondence between annotations and components in source code.

The second step for turning ownership annotations into a mechanism for reasoning about ownership is to modify the reduction relation of CapPCF so that it propagates ownership annotations. To achieve that, we must also annotate evaluation contexts with the label of the owner of the hole, i.e., the server label of the component boundary or the ownership annotation label that is closest to the hole (Figure 8). We use label l_o to denote the owner of the whole program and E^{l_o} for evaluation contexts without either a component boundary or an ownership annotation along the path to the hole.

With the annotated evaluation contexts in hand, we define the reduction relation of CapPCF with annotations in Figure 9. We use notation $\|e\|^{l}$ to indicate that e is wrapped with zero or more ownership annotations, all with label l, and e itself does not have an ownership annotation:

$$\|e\|^{l} \text{ iff } |\ldots|e|^{l}\ldots|^{l} \text{ and } e \neq |e'|^{k} \text{ for any } k.$$

We write $\bar{l}$ to denote a set of labels. Note that the reduction rules require that the redex have the same owner as the evaluation context, either implicitly (through the lack of annotations) or explicitly (with an annotation). Thus, program states where terms have more than one owner (i.e., ownership annotations with different labels) may get stuck. The absence of such stuck states during the evaluation of well-formed terms indicates that the semantics of CapPCF respects a "single-owner policy": each term is owned by a single label.

The rules for primitive operators, conditionals, capability creation and capability use are straightforward. One point worth mentioning is that when these terms produce base values, these values are without ownership annotations and implicitly acquire the owner of their context, which coincides with the implicit owner of the operator. Also, new abstract capabilities are explicitly annotated with the label of their creator so that we can easily track their origin.

The rules for function application and the fix-point operator are the most involved. They are the only rules in our model where a term e flows from one context (the evaluation context) into another (the abstraction body). We make this flow explicit by wrapping e with the owner of the evaluation context before installing e in the abstraction body.

The rules for component boundaries around functions and capabilities do not manipulate ownership annotations, because these rules do not cause values to cross component boundaries. Since ownership annotations and labels on component boundaries both express the notion of ownership, keeping one separate from the other makes reasoning about ownership in terms of ownership annotations independent of component boundaries. Therefore ownership annotations become a specification against which we can validate the way component boundaries mark code ownership.

In contrast to the other rules for component boundaries, the rule for component boundaries around base values, $\mathtt{K}^{l(k}\,\|\mathtt{b}\|^{p}\,)$, removes any ownership annotations around `b` and the surrounding context implicitly adopts `b`. This is the only rule that modifies the ownership of a term. Base values, unlike functions or capabilities, do not encapsulate the right to use a resource and thus the surrounding context can safely absorb them.

C. Properties of CapPCF

Using ownership annotations, we can define a security property for CapPCF. CapPCF is capability safe if and only if a component can directly cause a capability to be recorded in the usage trace only if it owns the capability or a guard for the capability. We define capability safety formally as a property of the Cap languages family, which includes CapPCF and extended versions defined in later sections.

Definition 1 (Capability Safety)**.** *A Cap language is* capability safe *iff for all terms e_0 such that $\varnothing; l_o \Vdash e_0$ and $\langle \varnothing, e_0 \rangle \xrightarrow{*} \langle U :: c, e_1 \rangle$, there exists v such that $\langle \varnothing, e_0 \rangle \xrightarrow{*} \langle U, E^l[\mathtt{use}(v)] \rangle \xrightarrow{*} \langle U :: c, e_1 \rangle$, where $v = \|\mathtt{G}^{l_n}\{^{k_n} \| ... \mathtt{G}^{l_1}\{^{k_1} \| c \|^{m_1} \}^{...} \|^{m_n} \}\|^{l}$ and $l_n = l$ (if it exists).*

The definition states that whenever evaluation of a program e_0 reaches a state where it records the use of a capability c, the previous state is a state $\langle U, e \rangle$ where a component with label l uses capability c, either directly or through a guard $\mathtt{G}^l\{^k \gamma\}$. If used directly, the component with label l owns the capability; if used through a guard, the component with label l owns the guard and the client label on the guard is l.

An important prerequisite to showing that CapPCF satisfies capability safety is the definition and proof of complete mediation for CapPCF programs. In well-formed programs, component boundaries are the only points where the owner of an embedded term may differ from that of its containing context. If the evaluation of a well-formed program does not preserve this invariant, programs may get stuck. Thus the absence of stuck states for all well-formed programs establishes that component boundaries and guards separate components throughout evaluation and completely mediate the flow of values between components:

Definition 2 (Complete Mediation)**.** *A Cap language satisfies* complete mediation *iff for all terms e_0 such that $\varnothing; l_o \Vdash e_0$ either 1) $\langle \varnothing, e_0 \rangle \xrightarrow{*} \langle U, v \rangle$ or, 2) for all terms e_1 and usage trace U_1 such that $\langle \varnothing, e_0 \rangle \xrightarrow{*} \langle U_1, e_1 \rangle$, there exists term e_2 and usage trace U_2 such that $\langle U_1, e_1 \rangle \rightarrow \langle U_2, e_2 \rangle$.*

We can now prove that well-formed CapPCF programs do not reach stuck states:[3]

Theorem 3. *CapPCF satisfies complete mediation.*

Complete mediation is sufficient to derive that CapPCF meets capability safety:

Theorem 4. *CapPCF is capability safe.*

III. Controlling Who Can Use Capabilities

In this section we extend the capability-safe language CapPCF with declarative access control policies that restrict who can use a capability. Recall the web mashup example from the Introduction, and consider the following CapPCF term virtualStore, which is a model of a `VirtualStore` function that intends to limit access to the local store.[4]

$$\mathsf{virtualStore} \equiv \mathtt{let}\ \mathsf{localStore} = ... \\ \mathtt{in}\ \lambda x.\mathtt{if}\ (\mathsf{test}\ x)\ \ \mathtt{use}(\mathsf{localStore})\ \ ...$$

In this term, variable localStore represents the local store, and virtualStore is a function that checks test to restrict the use of localStore. Unfortunately, even though this code looks like a reasonable attempt to restrict access to the local store, it allows localStore to escape to clients of virtualStore (such as the `ToDoList` and `DeadlineManager` components). This is because use(localStore) evaluates to localStore, and thus function virtualStore returns the localStore capability. As a result, toDoList can use localStore directly once it provides an argument that satisfies test:

$$\mathsf{toDoList} \equiv \mathtt{use}(\mathtt{K}^{\mathsf{toDoList}}(^{\mathsf{virtualStore}}\,\mathsf{virtualStore})\,42)$$

As the example demonstrates, capability safety alone is not sufficient to guarantee restrictions on who can use capabilities. Suppose we extend CapPCF with access control policies on capabilities, and write $c_{\bar{q}}$ for abstract capability c with access control policy $\bar{q}$: the set of labels of components that are allowed to use the capability. Intuitively, enforcing access control means that if c has access control policy $\bar{q}$, then the only components that use $c_{\bar{q}}$ or a guard for $c_{\bar{q}}$ are components with labels in the whitelist $\bar{q}$. Formally:

Definition 5 (Access Control)**.** *A Cap language enforces* access control *iff for every term e such that $\varnothing; l_o \Vdash e$, if*

$$\langle \varnothing, e \rangle \xrightarrow{*} \langle U, E^l[\mathtt{use}(v)] \rangle \rightarrow \langle U :: c_{\bar{q}}, E^l[v] \rangle$$

then $v = \|\mathtt{G}^{l_n}\{^{l_{n-1}} ... \|\mathtt{G}^{l_2}\{^{l_1} \| c_{\bar{q}} \|^{l_1} \}\|^{l_2} ... \}\|^{l_n}$ and $l \in \bar{q}$.

CapPCF does not enforce access control. We extend CapPCF to the language ac-CapPCF, which provides constructs to attach and enforce access control policies on capabilities.

A. Semantics of ac-CapPCF

Figure 10 presents the source syntax of ac-CapPCF. We replace CapPCF's `new` construct with $\mathtt{new}_{\bar{q}}$ that creates a new capability with access control policy $\bar{q}$. Term $\mathtt{new}_{\bar{q}}$ evaluates to an abstract capability $c_{\bar{q}}$, i.e., an abstract capability with the access control policy attached.

As we established in Section II, component boundaries are sufficient to track the flow of capabilities (Definition 1 and Theorem 4). Thus we use component boundaries, and the complete mediation they provide, to enforce access control policies in ac-CapPCF. In addition, each component boundary $\mathtt{K}^l(^k \kappa, e)$ is augmented with a *contract* κ. Contracts [20, 21] are executable specifications that regulate the exchange of values between components. In ac-CapPCF, contracts allow components to specify additional access control policies on capabilities they consume or return.

[3]The proof is a simplification of the progress-and-preservation complete monitoring proof of Dimoulas et al. [19]. We omit the details for conciseness. It is also easy to prove that ownership annotations do not change the meaning of well-formed programs, and thus results proved for CapPCF with annotations can be transferred to CapPCF.

[4]The $\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$ construct is syntactic sugar for $(\lambda x.e_2)\ e_1$.

$$
\begin{array}{lrcl}
\textbf{Types} & \tau & = & \ldots \mid \mathtt{con}(\tau) \\
\textbf{Contracts} & \kappa & = & \mathit{base} \mid \kappa \mapsto \kappa \mid \mathtt{cap}_{\bar{p}} \\
& \mathit{base} & = & \mathtt{int} \mid \mathtt{bool} \\
\textbf{Terms} & e & = & \ldots \mid \mathtt{K}^{l}(^{k}\, \kappa, e\,) \mid \mathtt{new}_{\bar{q}}
\end{array}
$$

Fig. 10. ac-CapPCF: syntax

$$
\begin{array}{lrcl}
\textbf{Terms} & e & = & \ldots \mid \mathtt{error}^{l} \\
\textbf{Capabilities} & \gamma & = & c_{\bar{q}} \mid \mathtt{G}^{l}\{^{k}\, \gamma, \bar{p}\,\}
\end{array}
$$

Fig. 11. ac-CapPCF: intermediate syntax

There are three kinds of contracts. Contract $\kappa_1 \mapsto \kappa_2$ is a contract for a function with contract κ_1 for the argument and contract κ_2 for the result. Base contracts, ranging over base, check base values. Capability contract $\mathtt{cap}_{\bar{p}}$ specifies an access control policy on capabilities that flow through it.

With the exception of specifying capability policies, our contracts are limited to type-like properties. It is straightforward to extend the contract language with arbitrary behavioral contracts but we opt for a simple language to avoid clutter from features that are orthogonal to our goals. An alert reader may wonder why we choose a contract system instead of an access control type system. As we show in Sections IV and V, contracts offer advantages over type systems in our setting: they allow the dynamic composition of components that enforce different kinds of security policies on capabilities. In addition, contracts require minimum modifications to source code, i.e, annotations on `new` constructs, and otherwise treat components as black boxes.

The intermediate syntax of ac-CapPCF is shown in Figure 11. As previously mentioned, abstract capabilities now include an access control policy annotation: $c_{\bar{q}}$. We also add access control policies to guards: $\mathtt{G}^{l}\{^{k}\, \gamma, \bar{p}\,\}$. If a contract fails during execution, it evaluates to an error term $\mathtt{error}^{k}$, where label k identifies the component that is responsible for the contract breach in the blame assignment tradition of contract systems [12].

Figures 12 and 13 show the changes to the evaluation contexts and reduction rules of CapPCF that are necessary to support access control policies and contract checking in ac-CapPCF. The reduction rule for creating an abstract capability now annotates the resulting capability with its access control policy. Note that the policy annotation on an abstract capability does not change after its creation.

Rules for component boundaries create new boundaries and guards, as in CapPCF. A component boundary for a function, $\mathtt{K}^{l}(^{k}\, \kappa_1 \mapsto \kappa_2, v\,)$ applies contracts κ_1 and κ_2 to the argument and the result of a function, respectively. A capability contract $\mathtt{cap}_{\bar{p}}$ on a component boundary evaluates to a guard $\mathtt{G}^{l}\{^{k}\, \gamma, \bar{p}\,\}$ with access control policy $\bar{p}$. This policy further restricts who may use the enclosed capability γ to components in $\bar{p}$.

When a capability wrapped in a stack of guards is used, the reduction rule checks that the client l_n is allowed to use capability γ according to the policies $\bar{p}_i$ on the guards

$$E = \ldots \mid \mathtt{K}^{l}(^{k}\, \kappa, E\,)$$

Fig. 12. ac-CapPCF: evaluation contexts

$$
\begin{array}{l}
\langle U, E[\cdots]\rangle \quad \to \langle U, E[\cdots]\rangle \\
\hline
\mathtt{K}^{l}(^{k}\, \kappa_1 \mapsto \kappa_2, v\,) \;.\; \lambda x.\mathtt{K}^{l}(^{k}\, \kappa_2, v\ \mathtt{K}^{k}(^{l}\, \kappa_1, x\,)\,) \\
\mathtt{K}^{l}(^{k}\, \mathtt{cap}_{\bar{p}}, v\,) \;.\; \mathtt{G}^{l}\{^{k}\, v, \bar{p}\,\} \\
\mathtt{K}^{l}(^{k}\, \mathit{base}, \mathtt{b}\,) \;.\; \mathtt{b} \\
\hline
\hline
\langle U, E[\mathtt{new}_{\bar{p}}]\rangle \quad \to \langle U, E[c_{\bar{p}}]\rangle \qquad \text{where } c \text{ is free} \\
\langle U, E[\mathtt{use}(\gamma)]\rangle \;\to \langle U :: c_{\bar{p}}, E[\gamma]\rangle \\
\quad \text{where } \gamma = \mathtt{G}^{l_n}\{^{k_n} \ldots \mathtt{G}^{l_1}\{^{k_1}\, c_{\bar{p}}, \bar{p}_1\,\}\ldots, \bar{p}_n\,\} \\
\quad \text{if for all } 1 \le i \le n,\ l_n \in \bar{p}_i \text{ and } l_n \in \bar{p} \\
\langle U, E[\mathtt{use}(\gamma)]\rangle \;\to \mathtt{error}^{q} \\
\quad \text{where } \gamma = \mathtt{G}^{l_n}\{^{k_n} \ldots \mathtt{G}^{l_1}\{^{k_1}\, c_{\bar{p}}, \bar{p}_1\,\}\ldots, \bar{p}_n\,\} \text{ and} \\
\quad q = l_j, \text{ if for all } j < i \le n,\ l_n \in \bar{p}_i \text{ and } l_n \notin \bar{p}_j, \text{ or} \\
\quad q = k_1, \text{if for all } 1 \le i \le n,\ l_n \in \bar{p}_i \text{ and } l_n \notin \bar{p}
\end{array}
$$

Fig. 13. ac-CapPCF: reduction semantics

and the policy $\bar{p}$ on the enclosed abstract capability: l_n must appear in all of the policies. Put differently, the access control policy for the capability at the bottom of a stack of guards is the intersection of all the policies in the stack. If the check fails, the contract system raises a contract error blaming either the client l_i of the topmost guard whose policy $\bar{p}_i$ does not contain l_n or the server k_1 if l_n appears in each $\bar{p}_i$ but not in $\bar{p}$. In the remainder of this section, we explain and establish the correctness of this blame assignment strategy as part of a complete monitoring property for ac-CapPCF, which in turn is sufficient to show that ac-CapPCF correctly enforces access control.

B. Obligation annotations in ac-CapPCF

To prove that ac-CapPCF satisfies the access control property, we develop a variant of ac-CapPCF with ownership annotations, and establish a new complete mediation property. The revised property also guarantees the correctness of blame assignment [13]. For this, we borrow *obligation annotations* from Dimoulas et al. [19] in addition to ownership annotations. Obligation annotations $\lfloor \mathtt{cap}_{\bar{p}} \rfloor^{k}$ decorate capability contracts in the source code, and indicate that component with label k is responsible for uses of the corresponding capability according to policy $\bar{p}$. When the semantics reduces component boundaries for capabilities to guards, it propagates the obligation annotation on the capability contract to the policy of the guard:

$$E^{l}[\mathtt{K}^{l}(^{k}\, \lfloor \mathtt{cap}_{\bar{p}} \rfloor^{q}, v\,)] \to E^{l}[\mathtt{G}^{l}\{^{k}\, v, \lfloor \bar{p} \rfloor^{q}\,\}]$$

Since the creation site of a capability also imposes an access control policy on the resulting capability, we decorate the policy with an obligation annotation too. Similar to capability contracts, the reduction rule for `new` propagates the obligation

$$\frac{l \in \bar{q}}{G; l \Vdash \texttt{new}_{\lfloor \bar{q} \rfloor^\iota}} \qquad \frac{l; k \rhd \kappa \quad G; k \Vdash e \quad k \neq l}{G; l \Vdash \mathtt{K}^l(^k\, \kappa, |e|^k)}$$

Fig. 14. ac-CapPCF with annotations: well-formed source terms

$$\frac{}{l; k \rhd base} \qquad \frac{}{l; k \rhd \lfloor \texttt{cap}_{\bar{p}} \rfloor^l} \qquad \frac{k; l \rhd \kappa_1 \quad l; k \rhd \kappa_2}{l; k \rhd \kappa_1 \mapsto \kappa_2}$$

Fig. 15. ac-CapPCF with annotations: well-formed contracts

annotation to the policy on the abstract capability:

$$E^l[\texttt{new}_{\lfloor \bar{p} \rfloor^k}] \to E^l[c_{\lfloor \bar{p} \rfloor^k}]$$

We use obligation annotations to prove that in the event of a contract breach, the correct component is blamed. In order to program effectively with contracts, a programmer must understand which contracts a component is responsible for satisfying; obligation annotations are proof mechanisms that express these responsibilities explicitly.

With the exception of obligation annotations, we can easily adapt the well-formedness relation, evaluation contexts, and reduction rules of CapPCF with annotations for ac-CapPCF with annotations. We focus here only on the interesting parts of the well-formedness relation that concern contracts, obligation annotations and capability policies, and omit the other rules.

Figure 14 displays the new well-formedness rules for capability creation and component boundaries. The first rule requires that the owner of a $\texttt{new}_{\bar{p}}$ construct is listed in the access control policy, i.e., $l \in \texttt{new}_{\bar{p}}$. In addition, it expects the owner of the creation site to appear as the obligation annotation on the policy of `new`. This reflects that the creator of a capability imposes the policy on it and must treat the capability in a manner consistent with the policy. The second rule, for component boundaries, is similar to the corresponding rule in CapPCF with an additional requirement that contract κ is well-formed: $l; k \rhd \kappa$.

Figure 15 defines well-formedness for contracts $l; k \rhd \kappa$. Here, k is the label of the provider of the value that the contract checks, and l is the label of the context that consumes the value. Base contracts are trivially well-formed. Well-formedness of capability contracts requires that the obligation annotation must match the client l. This indicates that l receives a capability under a particular policy and agrees to treat it accordingly. Function contracts are well-formed when their subcontracts are well-formed. Note that the rule for function contract $\kappa_1 \mapsto \kappa_2$ flips the positions of k and l when checking κ_1. This switches responsibility between the client and the server for the different subcontracts. This is consistent with the label flipping that the reduction rule for function component boundaries uses when constructing the component boundary for the argument of the function.

C. Security of ac-CapPCF

We can now define and prove an extended complete mediation property. The definition extends the complete mediation property for CapPCF (Definition 2) with an additional case that guarantees correct blame assignment for contract failures. Namely, the contract system blames a component only for uses of a capability that violate one of its obligations.

Definition 6 (Complete Mediation, revisited)**.** *A Cap language satisfies* complete mediation *iff for all terms e_0 such that $G; l_o \Vdash e_0$ either*

1) $\langle \varnothing, e_0 \rangle \xrightarrow{*} \langle U, v \rangle$ *or,*
2) *for all e_1 and U_1 such that $\langle \varnothing, e_0 \rangle \xrightarrow{*} \langle U_1, e_1 \rangle$, there exists e_2 and U_2 such that $\langle U_1, e_1 \rangle \to \langle U_2, e_2 \rangle$ or,*
3) $\langle \varnothing, e_0 \rangle \xrightarrow{*} \langle U_1, E^{k_{n+1}}[\texttt{use}(\|v\|^{k_{n+1}})] \rangle \to \langle U_2, \texttt{error}^q \rangle$ *where* $v =$

$$\mathtt{G}^{k_{n+1}}\{^{k_n} \|...\mathtt{G}^{k_2}\{^{k_1} \|c_{\lfloor \bar{p} \rfloor^{k_1}}\|^{k_1}, \lfloor \bar{p}_1 \rfloor^{k_2}\}...\|^{k_n}, \lfloor \bar{p}_n \rfloor^{k_{n+1}}\},$$

$n \geq 1$ and $q = k_j$, $1 \leq j \leq n+1$, if forall $j < i \leq n$, $k_{n+1} \in \bar{p}_i$ and $k_{n+1} \notin \bar{p}_j$ or, $q = k_1$, if forall $1 \leq i \leq n$, $k_{n+1} \in \bar{p}_i$ and $k_{n+1} \notin \bar{p}$.

Notice that complete monitoring entails that the contract system blames a component q in only two cases: 1) q uses a capability via a guard while it agreed on a policy (that of the guard) that does not include q or, 2) q obtains a capability with a policy that *does not* include component m, passes it to another component through a contract with a policy that *does* include m and eventually, m uses this capability. The latter also clarifies blame assignment when the policy on an abstract capability, such as $\bar{p}$ above, is violated: the creator of the capability is blamed if they apply inconsistent policies when creating and propagating the capability. Towards the end of this section, we illustrate the blame behavior with concrete examples.

The proof of complete mediation for ac-CapPCF is a straight-forward application of the standard complete monitoring proof for contracts [19].

Theorem 7. *ac-CapPCF satisfies complete mediation.*

Using the complete mediation theorem, we establish that ac-CapPCF is capability safe:

Theorem 8. *ac-CapPCF is capability safe.*

In addition, we define and prove for ac-CapPCF a revised access control property for well-formed programs. The revised version of access control (Definition 9) is stronger than the initial one (Definition 5) and allows components to refine the access control policies on capabilities: in addition to requiring that the component l that uses a capability $c_{\bar{p}}$ appears in $\bar{p}$, l must also appear in each policy $\bar{p}_i$ of the guards around c.

Definition 9 (Access Control, revisited)**.** *A Cap language enforces* access control *iff for every term e such that $\varnothing; l_o \Vdash e$, if $\langle \varnothing, e \rangle \xrightarrow{*} \langle U, E^l[\texttt{use}(v)] \rangle \to \langle U :: c, E^l[v] \rangle$ then $v = \|\mathtt{G}^{k_{n+1}}\{^{k_n} \|...\mathtt{G}^{k_2}\{^{k_1} \|c_{\lfloor \bar{p} \rfloor^{k_1}}\|^{l_1}, \lfloor \bar{p}_1 \rfloor^{k_2}\}...\|^{k_n}, \lfloor \bar{p}_n \rfloor^{k_n}\}\|^{k_{n+1}}$, $l \in \bar{p}$, and for all $1 \leq i \leq n$, we have $l \in \bar{p}_i$.*

Theorem 10. *ac-CapPCF enforces access control.*

Revisiting the example from the beginning of this Section, recall that function virtualStore uses capability localStore, but

incorrectly propagates it to clients. Assume that capability localStore has access control policy {virtualStore} on it (i.e., only component virtualStore, the component that function virtualStore belongs to, can access the capability). Consider the following term, which shows component toDoList using the capability it extracts from the insecure virtualStore function. Note that there is a component boundary between components virtualStore and toDoList, i.e., the example term belongs to toDoList.

$$\texttt{use}((\texttt{K}^{\mathsf{toDoList}}(^{\mathsf{virtualStore}}\,\texttt{int} \mapsto \texttt{cap}_{\{\mathsf{virtualStore}\}},\ \mathsf{virtualStore}))\ 42)$$

Under ac-CapPCF semantics, this term raises a contract violation $\texttt{error}^{\mathsf{toDoList}}$, because when component virtualStore passes the localStore capability to component toDoList, it imposes the access control policy {virtualStore}, which forbids toDoList to use the localStore capability.

The following term also evaluates to a contract violation.

$$\texttt{use}((\texttt{K}^{\mathsf{toDoList}}(^{\mathsf{virtualStore}}\,\texttt{int} \mapsto \texttt{cap}_{\{\mathsf{virtualStore},\mathsf{toDoList}\}},\ \mathsf{virtualStore}))\ 42)$$

This term is the same as the previous one, except that the access control policy on the capability contract for the value returned by virtualStore is {virtualStore, toDoList} instead of {virtualStore}. This time the capability contract fails and blames virtualStore because virtualStore mislead toDoList into violating the original policy on the capability by providing it under a more permissive contract.

As a final remark in this section, notice that the blame strategy of ac-CapPCF is not the only correct option. For instance, instead of blaming the last component that disagrees on the access control policy, a different semantics could blame the first or all such components. We can easily modify the reduction rule for use and the definition of complete monitoring to establish the correctness of the alternative semantics. These changes do not affect the access control property. The decision of which strategy is the most preferable is an issue of design, rather than correctness, to be settled with programming experience. In addition, it is worth mentioning that capability contracts can accommodate different interpretations of the access control policy than the one we adopt. For example, we can obtain an interpretation reminiscent of history-based access control [22] by requiring, upon the use of a capability through a stack of guards, that not only the owner of the use appears in all related policies but also the owners of the guards. Similarly, imposing an inconsistent access control policy could cause a contract failure at the component boundary. This gives rise to a less permissive interpretation of access control but detects failures more quickly. Finally, by having the component boundary for a capability check whether its client label appears in the policy of the capability, capability contracts can enforce the confidentiality of capabilities, i.e, serve as a dynamic monitor for (explicit) information flow for capabilities.

IV. Controlling Who Influences Capabilities

Access control restricts which components may use a given capability. However, even if a component is not allowed to use a capability directly, it can still influence its use. Recall the web mashup example from Section I, and that `DeadlineManager` needed both to communicate with the untrusted `AdService` and to use the local persistent store via the `VirtualStore` component. Security of this web mashup relied on `AdService` being unable to influence what key-value pairs were stored, which intuitively requires restricting the flow of information from `AdService` to the local store.

The following terms model `DeadlineManager`'s interaction with both `AdService` and `VirtualStore`. First, we redefine virtualStore so that it does not violate the access control policy that localStore is not used outside virtualStore:

$$\mathsf{virtualStore} \equiv \texttt{let}\ \mathsf{localStore} = \ldots \\ \texttt{in}\ \lambda x.\texttt{if}\ (\mathsf{test}\ x)\ (\texttt{use}(\mathsf{localStore}); \texttt{tt})\ \ldots$$

The deadline manager term deadlineManager composes virtualStore and an adService plugin:

$$\mathsf{deadlineManager} \equiv \\ (\texttt{K}^{\mathsf{deadlineManager}}(^{\mathsf{virtualStore}}\,\texttt{bool} \mapsto \texttt{bool}, \mathsf{virtualStore}))\\ \texttt{K}^{\mathsf{deadlineManager}}(^{\mathsf{adService}}\,\texttt{bool}, \mathsf{adService})$$

We assume that the code that defines abstract capability localStore gives it an access control policy of {virtualStore}, which prevents adService from using it directly. However, assuming that the evaluation of test depends on x, a boolean value supplied by adService influences whether virtualStore uses abstract capability localStore. Even though the access control policy is not violated, we do not achieve the desired security goal: ensuring the integrity of the uses of localStore.

In this section we present i-CapPCF, which extends ac-CapPCF with *integrity policies* that restrict which components may influence the use of a capability and uses an information-flow type system [14, 15] to enforce these integrity policies.

An integrity policy $\bar{q}$ is a whitelist of labels of components that are allowed to influence the use of a capability. We extend the new construct with integrity policy annotations: term $\texttt{new}_{\bar{p},\bar{q}}$ creates a capability with access control policy $\bar{p}$ and integrity policy $\bar{q}$. The i-CapPCF type system uses integrity policy annotations to reject programs which use capabilities inconsistently with respect to the associated integrity policies.

A. Integrity for capabilities

We formalize integrity for capabilities as an instance of termination-insensitive noninterference [23]. Before diving in to the formal definitions, we examine the intuition behind integrity policies.

Given capability c with integrity policy $\bar{q}$, we refer to the components that are allowed to influence the use of c (i.e., components with labels $\bar{q}$) as *trusted*, and refer to all other components as *untrusted*. Given a collection of capabilities $c_1 \ldots c_n$, the trusted components are the union of the trusted

components of all the capabilities (i.e., a component is untrusted only if it is untrusted by all the capabilities).

Suppose we have a collection of capabilities $c_1 \ldots c_n$ and two programs that differ only in the behavior of the untrusted components. If both programs enforce integrity policies, and both programs terminate, then their usage of capabilities $c_1 \ldots c_n$ should be identical. This is because only trusted components should influence the use of capabilities, and the trusted components of both programs behave identically.

To formalize the notion of two programs being equivalent in the behavior of trusted components, we first define term equivalence up to a set of labels $\bar{l}$.

Definition 11 (Term Equivalence up to $\bar{l}$)**.** Term equivalence up to $\bar{l}$ *is the least congruence relation on terms* $\overset{\bar{l}}{\sim}$ *such that* $e_1 \overset{\bar{l}}{\sim} e_2$ *if* $e_1 = \mathtt{K}^{p(k}\, e_1'\,)$ *and* $e_2 = \mathtt{K}^{p(k}\, e_2'\,)$ *and* $k \in \bar{l}$.

That is, the term equivalence relation relates two terms if they are different implementations of an untrusted component ($\mathtt{K}^{p(k}\, e_1'\,) \overset{\bar{l}}{\sim} \mathtt{K}^{p(k}\, e_2'\,)$ where $k \in \bar{l}$), and is otherwise a homomorphism on term constructors (e.g., $e_1 + e_2 \overset{\bar{l}}{\sim} e_1' + e_2'$ if $e_1 \overset{\bar{l}}{\sim} e_1'$ and $e_2 \overset{\bar{l}}{\sim} e_2'$).

Recall that a usage trace records the use of abstract capabilities during program execution. Two usage traces are equivalent up to a set of labels of untrusted components $\bar{l}$ if the two traces are equal after removing use of capabilities that are allowed to be influenced by $\bar{l}$.

Definition 12 (Usage Trace Equivalence up to $\bar{l}$)**.** *Consider usage traces* U *such that recorded capabilities are of the form* $c_{\bar{p},\bar{q}}$ *where* $\bar{p}$ *is the access control policy and* $\bar{q}$ *the integrity policy associated with the capability. Let* $[\![U]\!]^{\bar{l}}$ *be the filter that removes from* U *all capabilities* $c_{\bar{p},\bar{q}}$ *such that* $\bar{q} \subseteq \bar{l}$*. Two traces are* equivalent up to $\bar{l}$*, denoted* $U_1 \overset{\bar{l}}{\sim} U_2$*, iff* $[\![U_1]\!]^{\bar{l}} = [\![U_2]\!]^{\bar{l}}$.

Using these definitions, we can now formally specify our non-interference security property, that if two programs differ only in the behavior of untrusted components, then their usage of the trusted components' capabilities should be identical.

Definition 13 (Integrity for Capabilities)**.** *Term* e_1 *satisfies* integrity for capabilities *if for all sets of component labels* $\bar{l}$ *and for all terms* e_2 *such that* $e_1 \overset{\bar{l}}{\sim} e_2$*, if* $\langle \varnothing, e_1 \rangle \overset{*}{\to} \langle U_1, v_1 \rangle$ *and* $\langle \varnothing, e_2 \rangle \overset{*}{\to} \langle U_2, v_2 \rangle$ *then* $U_1 \overset{\bar{l}}{\sim} U_2$.

A term e_1 satisfies integrity for capabilities if for all sets of labels of untrusted components $\bar{l}$ and pairs of terms e_1 and e_2 that are equivalent up to $\bar{l}$, e_1 and e_2 behave equivalently with respect to the use of capabilities that do not trust $\bar{l}$.

The deadlineManager example from the beginning of this section does not satisfy Definition 13. Consider another expression that replaces the code of component adService such that test returns `ff`. The two expressions are equivalent up to {adService}, and both terminate. However, the original term produces a trace that records the use of localStore, whereas the modified term does not.

$$\begin{array}{lrcl} \textbf{Types} & \tau & = & \sigma^{\bar{l}} \\ & \sigma & = & \mathsf{Int} \mid \mathsf{Bool} \mid \tau_1 \xrightarrow{\bar{p}} \tau_2 \mid \mathsf{Cap}_{\bar{q}} \end{array}$$

Fig. 16. i-CapPCF: types

B. Type-based enforcement

Language i-CapPCF extends ac-CapPCF with integrity policies, which are enforced with a standard type-and-effect information flow type system [24, 25, 26]. The syntax and semantics of i-CapPCF are essentially the same as ac-CapPCF, though functions $\lambda_{[\bar{p}]}x{:}\tau.e$ come with an extra annotation $\bar{p}$ and $\mathtt{new}_{\bar{p},\bar{q}}$ terms specify both an access control policy $\bar{p}$ and an integrity policy $\bar{q}$. Similarly, a capability value $c_{\bar{p},\bar{q}}$ is annotated with both the access control policy $\bar{p}$ and integrity policy $\bar{q}$ from the term $\mathtt{new}_{\bar{p},\bar{q}}$ that created it. Integrity policy annotations are necessary only for type checking, do not affect computation, and can be erased before evaluation. In fact, the annotations on capabilities are necessary only for proving that the type system correctly enforces the policies.

Note that integrity policies and access control policies are distinct, as are their enforcement mechanisms. Using contracts together with security types allows us to use a precise dynamic enforcement mechanism with accurate blame assignment for access control together with a precise static enforcement mechanism for information flow control.

The type system is based on an integrity security lattice $(L, \subseteq)$ with sets of program component labels as the elements, set union as the join and set intersection as the meet. The top element $\top$ is the set of all component labels in a program; the bottom element $\bot$ is the empty set.

The syntax for types in i-CapPCF is shown in Figure 16. Base types σ include integers, booleans, function types, and capability types $\mathsf{Cap}_{\bar{q}}$, where $\bar{q}$ lists the component labels that are allowed to influence the use of capabilities of this type. Function types, $\tau_1 \xrightarrow{\bar{p}} \tau_2$ also have an extra annotation $\bar{p}$ that is a lower bound on the policies of any capabilities used by the function body. The type-checker can extract $\bar{p}$ from the source code of functions $\lambda_{[\bar{p}]}x{:}\tau.e$. Types τ in i-CapPCF are annotated base types $\sigma^{\bar{l}}$, where superscript $\bar{l}$ is an upper bound on the components that have influenced the values of this type.

The typing judgment has the form $\Gamma, \bar{l} \vdash e : \tau$. Here, Γ is a typing environment and $\bar{l}$ is the program counter level: an upper bound on the labels on components that influence the decision to evaluate e. The typing rules are mostly standard [26]. Figure 17 presents the most interesting rules of our type system and we discuss four of them here: the rules for type checking capabilities and their creation and use, and the rules for type checking component boundaries and guards.

The rule for using a capability, $\mathtt{use}(e)$, requires that any component that could have influenced either the decision to use the capability or which capability to use is allowed to do so. That is, both the program counter level and the upper bound on components that influence the result of e must be a subset of integrity policy $\bar{p}$. This prevents untrusted components from using capabilities that originate from trusted contexts as well

$$\frac{\Gamma, \bar{l} \vdash e_1 : (\tau_a \xrightarrow{\bar{p}} \sigma^{\bar{r}})^{\bar{q}} \quad \Gamma, \bar{l} \vdash e_2 : \tau_a \quad \bar{l} \cup \bar{q} \subseteq \bar{p}}{\Gamma, \bar{l} \vdash e_1\ e_2 : \sigma^{\bar{r} \cup \bar{q}}} \qquad \frac{\Gamma, \bar{l} \vdash e : \mathsf{Cap}^{\bar{p}}_{\bar{q}} \quad \bar{p} \cup \bar{l} \subseteq \bar{q}}{\Gamma, \bar{l} \vdash \mathtt{use}(e) : \mathsf{Cap}^{\bar{p}}_{\bar{q}}}$$

$$\frac{\Gamma \uplus \{x : \tau_a\}, \bar{p} \vdash e_{:}\tau_r}{\Gamma, \bar{l} \vdash \lambda_{[\bar{p}]} x{:}\tau_a.e : (\tau_a \xrightarrow{\bar{p}} \tau_r)^{\bar{l}}} \qquad \frac{\bar{l} \subseteq \bar{q}}{\Gamma, \bar{l} \vdash \mathtt{new}_{\bar{p},\bar{q}} : \mathsf{Cap}^{\bar{l}}_{\bar{q}}}$$

$$\frac{\Gamma, \bar{l} \cup \{k\} \vdash e : \tau \quad \vdash \kappa : \tau' \quad \vdash \tau \leq \tau'}{\Gamma, \bar{l} \vdash \mathtt{K}^{l(k}\, \kappa, e\,) : \tau}$$

$$\frac{\bar{l} \subseteq \bar{q}}{\Gamma, \bar{l} \vdash c_{\bar{p},\bar{q}} : \mathsf{Cap}^{\bar{l}}_{\bar{q}}} \qquad \frac{\Gamma, \bar{l} \cup \{k\} \vdash e : \mathsf{Cap}^{\bar{q}}_{\bar{p}}}{\Gamma, \bar{l} \vdash \mathtt{G}^{l\{k}\, e, \bar{l}'\,\} : \mathsf{Cap}^{\bar{q}}_{\bar{p}}}$$

Fig. 17. i-CapPCF: selected type-checking rules for terms

$$\frac{}{\vdash \mathtt{int} : \mathsf{Int}^{\top}} \qquad \frac{}{\vdash \mathtt{int} : \mathsf{Bool}^{\top}} \qquad \frac{}{\vdash \mathtt{cap}_{\bar{l}} : \mathsf{Cap}^{\top}_{\bot}}$$

$$\frac{}{\vdash_{-} \mathtt{cap}_{\bar{l}} : \mathsf{Cap}^{\bot}_{\top}} \qquad \frac{}{\vdash_{-} \mathtt{int} : \mathsf{Int}^{\bot}} \qquad \frac{}{\vdash_{-} \mathtt{int} : \mathsf{Bool}^{\bot}}$$

$$\frac{\vdash_{-} \kappa_1 : \tau_1 \quad \vdash \kappa_2 : \tau_2}{\vdash \kappa_1 \mapsto \kappa_2 : (\tau_1 \xrightarrow{\bot} \tau_2)^{\top}} \qquad \frac{\vdash \kappa_1 : \tau_1 \quad \vdash_{-} \kappa_2 : \tau_2}{\vdash_{-} \kappa_1 \mapsto \kappa_2 : (\tau_1 \xrightarrow{\top} \tau_2)^{\bot}}$$

Fig. 18. i-CapPCF: contracts type-checking rules

as preventing the use of trusted capabilities in ways that are influenced by untrusted values.

A new capability, $\mathtt{new}_{\bar{p},\bar{q}}$, has type $\mathsf{Cap}^{\bar{l}}_{\bar{q}}$, where $\bar{q}$ is the specified integrity policy, and $\bar{l}$ is the program counter level. This captures the intuition that a component k that influences the decision to create a capability (i.e., k is in the program counter level $\bar{l}$) influences the use of that capability. Moreover, we require that $\bar{l}$ is a subset of $\bar{q}$, enforcing that the integrity policy is an upper bound on which components may influence the use of a capability. Similarly, the rule for an abstract capability value $c_{\bar{p},\bar{q}}$ gives it type $\mathsf{Cap}^{\bar{l}}_{\bar{q}}$ and requires that $\bar{l}$ is a subset of $\bar{q}$.

Typing a component boundary $\mathtt{K}^{l(k}\, \kappa, e\,)$ requires typing the body e. Server label k is added to the program counter level used to type e, reflecting that the code of e is determined by component k. The contract κ is given a type via relation $\vdash \kappa : \tau$, which assigns to κ the most permissive type that is consistent with the structure of κ. Figure 18 presents this relation; it uses a helper relation $\vdash_{-} \kappa : \tau$ to type contracts in negative positions of function contracts. The type of the component body e must be a subtype of the type of contract κ, indicated by the relation $\vdash \tau \leq \tau'$. The rule for guards is a specialized version of the rule for component boundaries that takes into account the capability type of the guarded term. Notice that the access control policy does not affect the typing judgment in accordance with our decision to separate enforcement of access control and integrity policies.

Subtyping in our type system is mostly standard. Figure 19 presents the only non-standard rule in our system: subtyping for capability types. The interesting part is the contravariance of integrity policies: $\mathsf{Cap}^{\bar{p}}_{\bar{q}}$ is a subtype of $\mathsf{Cap}^{\bar{p}'}_{\bar{q}'}$ if integrity

$$\frac{\bar{q}' \subseteq \bar{q} \qquad \bar{p} \subseteq \bar{p}'}{\vdash \mathsf{Cap}^{\bar{p}}_{\bar{q}} \leq \mathsf{Cap}^{\bar{p}'}_{\bar{q}'}}$$

Fig. 19. i-CapPCF: subtyping rules

policy $\bar{q}'$ is at least as restrictive as $\bar{q}$, i.e., if $\bar{q}' \subseteq \bar{q}$. Intuitively, this means that if a capability's integrity policy allows only components $\bar{q}$ to influence its use, it is sound to allow only a subset of those components to influence its use. As a consequence, $\mathsf{Cap}^{\bot}_{\top}$ is the most permissive capability type, and $\mathsf{Cap}^{\top}_{\bot}$ is the most restrictive (but uninhabited) type.

C. Security of i-CapPCF

As for ac-CapPCF, we prove that i-CapPCF is capability safe and enforces access control:

Theorem 14. *i-CapPCF is capability safe.*

Theorem 15. *i-CapPCF enforces access control.*

Moreover, the type system of i-CapPCF enforces the integrity property for well-typed programs:

Theorem 16 (Integrity for Capabilities)**.** *If* $\varnothing, \{l_o\} \vdash e_1 : \tau$*, for all terms* e_2 *such that* $\varnothing, \{l_o\} \vdash e_2 : \tau$*, and* $e_1 \overset{\bar{l}}{\sim} e_2$*, if* $\langle \varnothing, e_1 \rangle \xrightarrow{*} \langle U_1, v_1 \rangle$ *and* $\langle \varnothing, e_2 \rangle \xrightarrow{*} \langle U_2, v_2 \rangle$ *then* $U_1 \overset{\bar{l}}{\sim} U_2$*.*

Note that we consider only well-typed terms, since ill-typed terms will not be executed. We prove this theorem using a straightforward adaptation of Pottier and Simonet's [25] noninterference proof technique.

Revisiting the example from the beginning of the section (now decorated with types), we can see that it does not type check if localStore is annotated with the integrity policy $\{\mathsf{virtualStore}, \mathsf{deadlineManager}\}$:

$$\begin{array}{l} \qquad\qquad \mathsf{virtualStore} \equiv \\ \quad \mathtt{let}\ \mathsf{localStore}{:}\mathsf{Cap}^{\{\mathsf{virtualStore},\mathsf{deadlineManager}\}}_{\{\mathsf{virtualStore},\mathsf{deadlineManager}\}} = \ldots \\ \mathtt{in}\ \ \lambda_{[\{\mathsf{virtualStore},\mathsf{deadlineManager}\}]} x{:}\mathsf{Bool}^{\{\mathsf{deadlineManager}\}}. \\ \qquad\quad \mathtt{if}\ (\mathsf{test}\ x)\ \ (\mathtt{use}(\mathsf{localStore}); \mathtt{tt})\ \ \ldots \\ \qquad\quad \mathsf{deadlineManager} \equiv \\ (\mathtt{K}^{\mathsf{deadlineManager}(\mathsf{virtualStore}}\, \mathtt{bool} \mapsto \mathtt{bool}, \mathsf{virtualStore}\,)) \\ \quad\ \mathtt{K}^{\mathsf{deadlineManager}(\mathsf{adService}}\, \mathtt{bool}, \mathsf{adService}\,) \end{array}$$

The culprit is the result of adService, which has type $\mathsf{Bool}^{\bar{l}}$, where $\mathsf{adService} \in \bar{l}$. This type is incompatible with the type of variable x, which may only be influenced by deadlineManager.

V. Combining Languages

The languages of the previous three sections enforce increasingly stronger policies about the use of capabilities: CapPCF provides capability safety; ac-CapPCF adds access control policies, dynamically enforced using contracts; and i-CapPCF adds integrity policies, statically enforced using an information-flow type system. However, a programmer must apply significant effort to transform a CapPCF program into an

$$\frac{}{\vdash \mathtt{cap}_? : \mathsf{Cap}^{\top}_{\bot}} \qquad \frac{}{\vdash_{-} \mathtt{cap}_? : \mathsf{Cap}^{\bot}_{\top}}$$

$$\frac{\langle U, E[\cdots]\rangle \quad \to \langle U, E[\cdots]\rangle}{\begin{array}{ll} \mathtt{K}^{l}\langle\!\langle^{k}\, \mathtt{cap}_?, c_{\bar{p},}\rangle & .\ \mathtt{G}^{l}\{\!\{^{k}\, c_{\bar{p},}, \bar{p}\}\!\} \\ \mathtt{K}^{l}\langle\!\langle^{k}\, \mathtt{cap}_?, \mathtt{G}^{l'}\{\!\{^{k'}\, \gamma, \bar{p}\}\!\}\rangle & .\ \mathtt{G}^{l}\{\!\{^{k}\, \mathtt{G}^{l'}\{\!\{^{k'}\, \gamma, \bar{p}\}\!\}, \bar{p}\}\!\} \end{array}}$$

Fig. 20. mix-CapPCF: the wild-card capability contract

$$\frac{l \in \text{i-CapPCF} \qquad k \in \text{ac-CapPCF} \qquad \Gamma \vdash e}{\Gamma, \bar{l} \vdash \mathtt{K}^{l}\langle\!\langle^{k}\, \kappa, e\rangle : \mathtt{T}[\![\kappa]\!]}$$

$$\frac{l \in \text{ac-CapPCF} \qquad k \in \text{i-CapPCF} \qquad \Gamma, \top \vdash e : \mathtt{T}[\![\kappa]\!]}{\Gamma \vdash \mathtt{K}^{l}\langle\!\langle^{k}\, \kappa, e\rangle}$$

$$\frac{l \in \text{ac-CapPCF} \qquad k \in \text{ac-CapPCF} \qquad \Gamma \vdash e}{\Gamma \vdash \mathtt{K}^{l}\langle\!\langle^{k}\, \kappa, e\rangle}$$

Fig. 21. mix-CapPCF: type-checking rules for mixed components

$$\begin{array}{rclcrcl} \mathtt{T}[\![\mathtt{int}]\!] & = & \mathsf{Int}^{\top} & & \mathtt{T}[\![\mathtt{bool}]\!] & = & \mathsf{Bool}^{\top} \\ \mathtt{T}[\![\mathtt{cap}_{\bar{l}}]\!] & = & \mathsf{Cap}^{\top}_{\top} & & \mathtt{T}[\![\mathtt{cap}_?]\!] & = & \mathsf{Cap}^{\top}_{\top} \end{array}$$

$$\mathtt{T}[\![\kappa_1 \mapsto \kappa_2]\!] \;=\; (\mathtt{T}[\![\kappa_1]\!] \xrightarrow{\top} \mathtt{T}[\![\kappa_2]\!])^{\top}$$

Fig. 22. mix-CapPCF: contracts to types translation

i-CapPCF program that enforces access control and integrity policies. In particular for integrity, this typically requires prolific security annotations and an all-or-nothing trial and error process until the type system admits the program [27]. As a middle ground, the contracts of ac-CapPCF offer a smoother transition: the programmer immediately obtains a working ac-CapPCF program by adding contracts only on component boundaries.

This section shows how the transition from CapPCF to ac-CapPCF and i-CapPCF can be done gradually on a per component basis: we demonstrate how components written in any of the three languages can be composed to form a complete system without violating the security policies of the individual components.

First observe that we can rewrite any CapPCF component as an ac-CapPCF component with the same behavior. Recall that ac-CapPCF extends CapPCF with access control policy annotations on capabilities and contracts on component boundaries. We consider a CapPCF capability c to have access control policy $\top$, the set of labels of all components in a program. That is, the default access control policy for a CapPCF capability allows any component to use it. We can construct a contract for every CapPCF component from its type. The only interesting piece of this construction is the choice of white-list policies on capability contracts. We handle this corner case by adding a wild-card policy $?$ to ac-CapPCF that never leads to a contract violation.

Let mix-CapPCF be a language that allows us to mix components of ac-CapPCF and i-CapPCF. Except for type annotations and integrity policies on capabilities, ac-CapPCF and i-CapPCF share the same syntax, and thus the two languages share the same reduction relation after type erasure. Therefore a mix-CapPCF program has a well defined meaning. We do need to extend the reduction rules for component boundaries with wild-card capability contracts, as shown in Figure 20, in addition to typing rules for wild-card capabilities.

A. Typing component mixes

To ensure that mix-CapPCF satisfies integrity we must extend the i-CapPCF type system to handle mixed components. Figure 21 displays new typing rules that allow secure mixing of components. These rules describe how to type component boundaries $\mathtt{K}^{l}\langle\!\langle^{k}\, \kappa, e\rangle$ where the client label l indicates an i-CapPCF component and the server label k indicates an ac-CapPCF component. An ac-CapPCF component k embedded in an i-CapPCF context l is well-typed as long as its i-CapPCF sub-components are well-typed. The judgment $\Gamma \vdash e$ holds for an ac-CapPCF term e if all i-CapPCF sub-components are well-typed. The type of an ac-CapPCF component embedded in an i-CapPCF context is obtained using the contract-to-type translation operator $\mathtt{T}[\![\kappa]\!]$ that is presented in Figure 22. This translation operator conservatively maps contracts to types in a way that prevents capabilities with non-trivial integrity policies propagating from i-CapPCF components to ac-CapPCF components. Specifically, the translation operator ensures that only capabilities with an integrity policy of $\top$ (i.e., all components may influence the use of the capability) may propagate across the component boundary. For functions, it ensures that only capabilities with an integrity policy of $\top$ may be used in the function body.

All ac-CapPCF terms are well-typed according to $\Gamma \vdash e$ so long as their subterms are well-typed, with the exception of component boundaries where the component is an i-CapPCF component. In that case, the i-CapPCF component e must be well-typed according to judgment $\Gamma, \top \vdash e : \mathtt{T}[\![\kappa]\!]$. The program counter level for the i-CapPCF component is $\top$, indicating that the decision to execute e may be influenced by any and all components, since the ac-CapPCF component does not track integrity. Note that the type of e uses the contract-to-type translation operator $\mathtt{T}[\![\kappa]\!]$, again ensuring that only capabilities with an integrity policy of $\top$ may propagate across the component boundary.

B. Security of mix-CapPCF

The language mix-CapPCF inherits capability safety and access control enforcement from ac-CapPCF and i-CapPCF:

Theorem 17. *mix-CapPCF is authority safe.*

Theorem 18. *mix-CapPCF enforces access control.*

Based on the extended type-system, we can also show that mix-CapPCF satisfies integrity for capabilities:

Theorem 19 (Integrity for Capabilities)**.** *If* $\varnothing, \{l_o\} \vdash e_1 : \tau$*, for all terms* e_2 *such that* $\varnothing, \{l_o\} \vdash e_2 : \tau$*, and* $e_1 \overset{\bar{l}}{\sim} e_2$*, if* $\langle \varnothing, e_1\rangle \overset{*}{\to} \langle U_1, v_1\rangle$ *and* $\langle \varnothing, e_2\rangle \overset{*}{\to} \langle U_2, v_2\rangle$ *then* $U_1 \overset{\bar{l}}{\sim} U_2$*.*

In summary, CapPCF components can be easily converted to ac-CapPCF components, and ac-CapPCF and i-CapPCF components can be easily composed to conservatively enforce both access control and integrity policies. This enables the gradual addition of policies to a program.

VI. Capability Control in Real Languages

We have shown how to extend capability safe languages with declarative policies to restrict the use of capabilities. We have done so in the context of a series of simple calculi: CapPCF, ac-CapPCF, i-CapPCF, and mix-CapPCF. In this section we discuss the connection between CapPCF and existing capability-safe languages, and describe how standard, practical language techniques can be used to extend these existing capability-safe languages with access control and integrity policies.

A. CapPCF as a model for capability-safe languages

Existing capability-safe languages, such as Caja, E, and Joe-E, are object-oriented and capabilities are object references. In contrast to traditional protection systems where the subjects and objects of access are distinct [28], in these object-capability languages the entities that use capabilities and the targets of such uses are all objects. Objects play three distinct roles in object-capability languages. First, they provide services via their methods. Second, they act as consumers of services. Third, they implement application-specific security abstractions, which can be viewed as specialized consumers that aggregate capabilities and expose restricted facets of their services. Note that an object may perform all three roles.

Capability-safety gives some minimal security guarantees for objects as services: invoking a service requires a reference to the object and references propagate in the program only via message passing or object initialization. Objects as security abstractions provide additional guarantees by further mediating accesses to services.

CapPCF concisely captures the salient details of capability-safe languages even though we use a lambda-calculus based model without objects. We don't account for objects with internal state, behavioral subtyping, or object extension, but these features are orthogonal to our goals. To emphasize the distinction between critical services of interest and other services, security abstractions, and clients, CapPCF represents the former as capabilities γ invoked with $\mathtt{use}(\gamma)$, and the latter by lambda abstractions. This reduces clutter and allows us to focus on these critical services. We can easily remove this simplification by treating lambda abstractions as capabilities.

CapPCF and our security policies assume that programs are composed of components with associated principals (the component labels). In a capability-safe language, we can view collections of objects as components associated with principals or domains that indicate their origin. For example, in Caja principals might denote source URLs, whereas in E or Joe-E principals might denote packages or source files. Capability-safe languages already provide mechanisms (such as loaders) to create component boundaries and to attach principals to components [29], and thus our technique of using syntactic elements $\mathsf{K}^l(^k\, e\,)$ to explicitly mark component boundaries is a reasonable model of real languages.

Although our Cap family of calculi captures intuitive ideas (components with boundaries between them, capabilities, access control, integrity), a significant amount of technical machinery is required to model them. We believe that this is not an artifact of our formalism, but is innate to precisely modeling these concepts: we use standard modeling techniques from higher-order contracts and our formalism is of similar complexity to previous formal models of language-based capabilities (e.g., [30]). Indeed, we believe that the use of standard language techniques for modeling and enforcement of novel security guarantees is a benefit of our approach, and speaks to the practicality of both the security guarantees and enforcement mechanisms.

B. Implementing capability control

Well-known programming language constructs and techniques can be used to extend existing capability-safe languages with access control policies and integrity policies.

Interposition for Access Control. Enforcing access control policies in ac-CapPCF requires contracts that completely mediate access to a component. Object-capability-safe languages typically already have constructs that achieve complete and transparent mediation and interposition, e.g., membranes [10] in E and Caja. Recent research [7] explores the addition of such a feature to existing languages, and contract systems have been implemented using membrane-like constructs [19, 31]. In addition, membranes have been used to equip Javascript objects with contracts for path-based access control for method invocation [32].

Enforcing Information Flow. Enforcement of integrity policies in i-CapPCF relies on an information-flow type system. The addition of information-flow type systems to existing programming languages is well studied (e.g., [33, 34]). Extensions to information-flow type systems to improve practicality (such as endorsement [35] and flow-sensitivity [36]) can be easily incorporated into the type system of i-CapPCF. For dynamic languages such as Javascript and Caja, run-time enforcement mechanisms for information-flow control (e.g., [37, 38, 39]) offer an alternative enforcement mechanism for integrity policies.

VII. Related Work

Capability-based security. Capabilities have been widely used in operating systems to provide confinement and isolation (e.g., [40, 2, 3]). Capability-safe languages and their design patterns can enforce a variety of security properties [10].

A number of capability-based languages and mechanisms have been proposed to increase the security of browsers and web applications. For example, early proposals for securing mashups required third party code to conform to secure subsets of JavaScript, such as Google's Caja [1] and Yahoo's ADsafe [41] languages. More recently, Agten et al. [6] introduce

JSand, which sandboxes third-party scripts via client-side enforcement of object-capability principles.

Other web application security tools rely on browser primitives such as the same origin policy and iframes to provide isolation [42, 43, 44, 45]. The use of browser mechanisms for isolation is complementary to the use of capability-based patterns for building fine-grained security abstractions. For example, Meyerovich et al. [46] use iframes to provide isolation between components, but introduce object views to enable fine-grained sharing between components.

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 [47], the confinement problem [48] led to the combination of capabilities and access control policies. The ICAP system [49] uses access control policies on capabilities to limit their propagation in distributed systems. Our work is inspired by these early results, but focuses on capability-safe languages.

Maffeis et al. [30] 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. [50] use a type system to verify the confinement guarantees provided by ADsafe. Our work also considers the security of communicating components, but does so via declarative policies for the use of capabilities rather than excluding language features. In contrast to our language-based approach, Murray et al. [51, 52] and Spiessens [53] apply formal methods to verify the security of specific object-capability design patterns.

Closer to our work, Taly et al. [54] develop a static analysis for checking whether capability-based sandboxes properly confine access to sensitive resources. However, their analysis for access control depends on availability of the source code and does not consider integrity requirements on the use of capabilities. In a similar spirit, Barth et al. [55] and Finifter et al. [56] add restricted access control checks to JavaScript objects, but only for enforcing the same origin policy.

VIII. Conclusion

Capability-safe languages are a powerful tool for managing resources. However, reasoning about the correctness of applications built using these languages requires reasoning about implicit policies concerning the use of capabilities.

We extend capability-safe languages with declarative policies to simplify reasoning about the correct use of capabilities. *Access control policies* restrict which components may use a given capability. *Integrity policies* restrict which components may influence the use of a capability. We demonstrate that standard language-based techniques can soundly enforce these policies (contracts and a security type system respectively). Moreover, these enforcement mechanisms can be easily composed, allowing the gradual incorporation of capability policies. Thus, this work provides firm theoretical foundations for practical security extensions to capability-safe languages.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. This research is supported by the National Science Foundation under Grants 1054172 and 1237235, and by the Air Force Research Laboratory.

References

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

[2] J. S. Shapiro, J. M. Smith, and D. J. Farber, "EROS: a fast capability system," in *ACM Symposium on Operating Systems Principles*, 1999, pp. 170–185.

[3] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, "A taste of Capsicum: practical capabilities for UNIX," *Communications of the ACM*, vol. 55, no. 3, pp. 97–104, 2012.

[4] M. Miller, K.-P. Yee, and J. Shapiro, "Capability myths demolished," Johns Hopkins University, Tech. Rep. SRL2003-02, 2003.

[5] S. Drossopoulou and J. Noble, "The need for capability policies," in *Formal Techniques for Java-like Programs Workshop*, 2013, pp. 6:1–6:7.

[6] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens, "JSand: Complete client-side sandboxing of third-party JavaScript without browser modifications," in *Annual Computer Security Applications Conference*, 2012, pp. 1–10.

[7] T. H. Austin, T. Disney, and C. Flanagan, "Virtual values for language extension," in *ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, Languages, and Applications*, 2011, pp. 921–938.

[8] T. Van Cutsem and M. Miller, "Proxies: Design principles for robust object-oriented intercession APIs," in *Dynamic Languages Symposium*, 2010, pp. 59–72.

[9] T. Van Cutsem and M. S. Miller, "Trustworthy proxies: Virtualizing objects with invariants," in *European Conference on Object-Oriented Programming*, 2013.

[10] M. Miller, "Robust composition: Towards a unified approach to access control and concurrency control," Ph.D. dissertation, Johns Hopkins University, 2006.

[11] "OWASP top 10 - 2013. The ten most critical web application security risks," OWASP The Open Web Application Security Project, Tech. Rep., 2013. [Online]. Available: http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013.pdf

[12] R. B. Findler and M. Felleisen, "Contracts for higher-order functions," in *International Conference on Functional Programming*, 2002, pp. 48–59.

[13] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen, "Correct blame for contracts: No more scapegoating," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2011, pp. 215–226.

[14] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[15] D. Volpano, G. Smith, and C. Irvine, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 3, pp. 167–187, 1996.

[16] J. A. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[17] G. D. Plotkin, "LCF considered as a programming language," *Theoretical Computer Science*, vol. 5, no. 3, pp. 223–255, 1977.

[18] M. Felleisen, R. B. Findler, and M. Flatt, *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[19] C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen, "Complete monitors for behavioral contracts," in *European Symposium on Programming*, 2012, pp. 211–230.

[20] B. Meyer, "Design by contract," in *Advances in Object-Oriented Software Engineering*. Prentice Hall, 1991, pp. 1–50.

[21] ——, "Applying design by contract," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[22] M. Abadi and C. Fournet, "Access control based on execution history," in *Network and Distributed System Security Symposium*, 2003, pp. 107–121.

[23] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, "Termination-insensitive noninterference leaks more than just a bit," in *European Symposium on Research in Computer Security*, 2008, pp. 333–348.

[24] N. Heintze and J. G. Riecke, "The SLam calculus: programming with secrecy and integrity," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998, pp. 365–377.

[25] F. Pottier and V. Simonet, "Information flow inference for ML," *ACM Transactions on Programming Languages and Systems*, vol. 25, no. 1, pp. 117–158, 2003.

[26] S. A. Zdancewic, "Programming languages for information security," Ph.D. dissertation, University of Pennsylvania, 2002.

[27] A. Askarov and A. Sabelfeld, "Security-typed languages for implementation of cryptographic protocols: A case study," in *European Symposium on Research in Computer Security*, 2005, pp. 197–221.

[28] B. Lampson, "Protection," in *International Conference on Information Systems Security*, 1971, pp. 437–443.

[29] M. S. Miller, J. E. Donnelley, and A. H. Karp, "Delegating responsibility in digital systems: Horton's "who done it?"," in *HotSec*, 2007, pp. 2:1–2:5.

[30] S. Maffeis, J. C. Mitchell, and A. Taly, "Object capabilities and isolation of untrusted web applications," in *IEEE Symposium on Security and Privacy*, 2010, pp. 125–140.

[31] T. S. Strickland, S. Tobin-Hochstadt, R. Findler, and M. Flatt, "Chaperones and impersonators," in *ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, Languages, and Applications*, 2012, pp. 943–962.

[32] M. Kiel and P. Thiemann, "Efficient dynamic access analysis using JavaScript proxies," in *Dynamic Languages Symposium*, 2013.

[33] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom, "Jif: Java information flow," 2001–2008, http://www.cs.cornell.edu/jif.

[34] V. Simonet, "The Flow Caml System: documentation and user's manual," INRIA, Technical Report, 2003.

[35] A. Sabelfeld and D. Sands, "Dimensions and principles of declassification," in *IEEE Computer Security Foundations Symposium*, 2005, pp. 255–269.

[36] S. Hunt and D. Sands, "On flow-sensitive security types," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006, pp. 79–90.

[37] T. H. Austin and C. Flanagan, "Permissive dynamic information flow analysis," in *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2010, pp. 3:1–3:12.

[38] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner, "Staged information flow for JavaScript," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 50–62.

[39] D. Hedin and A. Sabelfeld, "Information-flow security for a core of JavaScript," in *IEEE Computer Security Foundations Symposium*, 2012, pp. 3–18.

[40] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, 1966.

[41] D. Crockford. (2013) ADSafe. [Online]. Available: http://www.adsafe.org

[42] D. Akhawe, P. Saxena, and D. Song, "Privilege separation in HTML5 applications," in *IEEE Symposium on Security and Privacy*, 2012.

[43] L. Ingram and M. Walfish, "Treehouse: JavaScript sandboxes to help web developers help themselves," in *USENIX ATC*, 2012.

[44] M. Ter Louw, K. T. Ganesh, and V. Venkatakrishnan, "AdJail: Practical enforcement of confidentiality and integrity policies on web advertisements." in *USENIX Security*, 2010, pp. 371–388.

[45] H. J. Wang, X. Fan, J. Howell, and C. Jackson, "Protection and communication abstractions for web browsers in MashupOS," in *ACM Symposium on Operating Systems Principles*, 2007, pp. 1–16.

[46] L. A. Meyerovich, A. P. Felt, and M. S. Miller, "Object views: Fine-grained sharing in browsers," in *WWW*, 2010, pp. 721–730.

[47] P. A. Karger, "Improving security and performance of capability systems," Ph.D. dissertation, University of Cambridge, 1988.

[48] B. W. Lampson, "A note on the confinement problem," *Communications of the ACM*, vol. 16, no. 10, pp. 613–615, 1973.

[49] L. Gong, "A secure identity-based capability system," in *IEEE Symposium on Security and Privacy*, 1989, pp. 56–63.

[50] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi, "Type-based verification of JavaScript sandboxing," in *USENIX Security*, 2011.

[51] T. Murray, "Analysing object-capability security," in *Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security*, 2008, pp. 177–194.

[52] T. Murray and G. Lowe, "Analysing the information flow properties of object-capability patterns," in *IEEE Symposium on Security and Privacy*, 2009, pp. 81–95.

[53] A. Spiessens, "Patterns of safe collaboration," Ph.D. dissertation, Catholic University of Louvain, 2007.

[54] A. Taly, J. C. Mitchell, M. S. Miller, and J. Nagra, "Automated analysis of security-critical JavaScript APIs," in *IEEE Symposium on Security and Privacy*, 2011, pp. 363–378.

[55] A. Barth, J. Weinberger, and D. Song, "Cross-origin JavaScript capability leaks: Detection, exploitation, and defense," in *USENIX Security*, 2009, pp. 187–198.

[56] M. Finifter, J. Weinberger, and A. Barth, "Preventing capability leaks in secure JavaScript subsets," in *Network and Distributed System Security Symposium*, 2010.