Dynamic Access Control in a Concurrent Object Calculus

Avik Chaudhuri

Computer Science Department
University of California, Santa Cruz
avik@cs.ucsc.edu

Abstract. We develop a variant of Gordon and Hankin's concurrent object calculus with support for flexible access control on methods. We investigate safe administration and access of shared resources in the resulting language. Specifically, we show a static type system that guarantees safe manipulation of objects with respect to dynamic specifications, where such specifications are enforced via access changes on the underlying methods at runtime. By labeling types with secrecy groups, we show that well-typed systems preserve their secrets amidst dynamic access control and untrusted environments.

1 Introduction

Systems that share resources almost always exercise some control on access to those resources. The access control typically relies on a set of rules that decide which access requests to accept; furthermore, those rules may be subject to change to reflect changing requirements on resource access at runtime. While access control mechanisms are often easy to deploy—they are both common and various—they are surprisingly difficult to marshal towards achieving higher safety goals. For instance, users who have access to a file with sensitive content can often share the content, intentionally or by mistake, with those who do not have access; even if the privileged users are careful, a change in the access rules can allow other users to read that content, or write over it.

A convenient view of access control results from its characterization in terms of capabilities: a resource may be accessed if and only if a corresponding capability is shown for its access. For one, this view provides an immediate low-level abstraction of access control "by definition"; two, the view is independent of higher level specifications on resource usage (say, in terms of types, or identities of principals). The separation facilitates higher level proofs, since it suffices to guarantee that the flow of a capability that protects a resource respects the corresponding high-level intention on resource usage. We develop methods to provide such guarantees in this paper. The methods in turn rely on a sound low-level implementation of access control in terms of capabilities. Fortunately, to that end, a capability for a resource can be identified with a pointer to that resource. Exporting a direct link to a resource, however, poses problems for dynamic access control, as discussed by Redell in his dissertation (1974). Redell suggests a simple alternative that uses indirection: export indirect pointers to a local, direct reference to the resource, and overwrite this local pointer to modify access to that resource [30]. We revisit that idea in this paper.

We study safe dynamic access control in a concurrent object language. Resources are often built over other resources; dependencies between resources may entail dependencies on their access assumptions for end-to-end safety. For example, suppose two users read the same file to obtain what they believe is a shared secret key that they then use to encrypt secret messages between themselves; it does not help if a third user can write a Trojan key on that file and then decrypt the "secret" messages. A natural way to capture such dependencies is to group the related resources into objects. (In the example above, the object would be the file in question, and the resources would be a "key" field and "read" and "write" methods that manipulate that field.)

We develop a variant of Gordon and Hankin's concurrent object calculus $\mathbf{conc}_\varsigma$ [14] for our study. In $\mathbf{conc}_\varsigma$, as in most previous object calculi (*e.g.*, [2, 4, 32]), a method is accessed by providing the name of the parent object and a label that identifies the method. For example, for a timer object t in the calculus, with two methods, tick and set, knowing the name t is sufficient to call (or even redefine) both methods ($t.\mathsf{tick}$, $t.\mathsf{set}$). We may, however, want to restrict access to set to the owner of t, while allowing other users to access tick; such requirements are not directly supported by $\mathbf{conc}_\varsigma$. In languages like Java, there is limited support for method-level access control via access modifiers—however, such modifiers can only sometimes be changed in a very restricted way, and moreover they implement a policy design that is fixed by the language.

Our calculus provides *veils* for implementing flexible access control of methods. Veils are similar to Redell's indirect access pointers. More specifically, a veil is an alias (or "handle") for the label that identifies a particular method inside an object definition. A method is invoked by sending a message on its veil; method access is modified by re-exporting a different veil for its label. A method call crucially does not require the name of the parent object. An object name, on the other hand, is required for access modification and redefinition of methods—thus object names are similar to Redell's local references (or "capabilities"). In the sequel, informally, a *capability* is a reference to an object, and veils are indirect references to its methods. A capability is meant to be shared between the owner and other administrators of an object, and veils are meant to be made available to the users of its methods. Dependencies between object methods often require their redefinitions and access modifications to be simultaneous—therefore the calculus replaces $\mathbf{conc}_\varsigma$'s method update with a more general "administration" primitive. Veils allow a relatively straightforward encoding of the mutex primitives of $\mathbf{conc}_{\varsigma\mathbf{m}}$ (an extension of $\mathbf{conc}_\varsigma$ that facilitates encodings of locks, communication channels, *etc.*), so we do not include those primitives in the syntax.

We show a type system for the resulting language that guarantees safe manipulation of objects with respect to dynamically changing specifications. Informally, we allow object methods to change their exported "type views" at runtime: in other words, resource administrators can not only control resource usage at runtime, but also dynamically specify why they do so (*i.e.*, their higher level intentions). This flexibility is desirable since persistent resources (*e.g.*, file systems, memory) are typically used in several different contexts over time. For example, files are often required to pass through intervals of restricted access; memory locations are dynamically allocated/deallocated to map different data structures over several program executions. By a combination of access control (as provided by the language) and static discipline (provided by the type system) we can show that the intentions of the users and administrators of those resources

are respected through and between such phases of flux. In particular, by labeling types with secrecy groups, we show that well-typedness guarantees secrecy under dynamic access control, even in the presence of possibly untyped, active environments.

Outline of the paper. In the next section we present a concurrent object calculus with veils for dynamic access control (the "veil calculus", for brevity). We accompany the formal syntax and semantics of our language with commentary on the conceptual and technical differences with Gordon and Hankin's original calculus. In Section 3, we present a type system for the language, show examples of well-typed programs, and state our main theorem, *viz.* typing guarantees secrecy under dynamically changing type views and even under untyped environments. We discuss related work in Section 4 and conclude by summarizing our contributions.

2 The untyped veil calculus

In this section we present a variant of the calculus $\mathbf{conc}_{\varsigma}$ [14]. The novel aspects of the language lie in the separation of roles for method definition and invocation; this separation is induced by a fresh treatment of method names via veils.

The syntax is very similar to that of $\mathbf{conc}_{\varsigma}$; as such, it retains most of the simplicity, compactness, and expressivity of the original. Although we make minimal changes to the original calculus (specifically, in the manner of method call and update), the changes have a clear effect on the suitability of the resulting language as a core calculus for studying security properties of concurrent objects. As argued in Section 1, the original calculus cannot separate the ability to call a method (*i.e.*, use a resource) from the ability to redefine it (*i.e.*, do administration on it); moreover, it cannot distinguish between method-access abilities within the same object. Persistent resources characteristically require support for such distinctions for security. The new language improves upon $\mathbf{conc}_{\varsigma}$ in this respect, since veils can enforce those distinctions quite naturally.

$u, v, w ::=$	results
$\quad x$	variable
$\quad m, n, p, q, \theta$	name (capability, veil)
$d ::=$	denotations
$\quad \widetilde{\theta}[\widetilde{\ell} \Mapsto \varsigma(x)\widetilde{(y)b}]$	object
$a, b ::=$	expressions
$\quad u$	result
$\quad p \mapsto d$	denomination
$\quad (\nu n)\, a$	restriction
$\quad a \Rsh b$	fork
$\quad \text{let } x = a \text{ in } b$	evaluation
$\quad \ell(u)$	internal method call
$\quad \ell \Leftarrow (y)b$	internal method update
$\quad \eth\langle u \rangle$	external method call
$\quad u \hookleftarrow d$	external update ("administration")

Names in the veil calculus fall into two conceptual categories (that we do not distinguish syntactically): object names, which we call capabilities, and method alias names, which

we call veils. An object is defined by a map from labels to expressions, and a map from labels to veils. The former map defines the methods of the object. The method bodies abstract on a "self" variable that gets bound to the name of the object at runtime. Unlike $\mathbf{conc}\varsigma$, they also abstract on an "argument" variable—while parameter passing can be encoded even otherwise, having explicit argument abstraction allows better typings. The map from labels to veils defines the external aliases for the methods of the object. As usual, these (finite) maps are written as associated sequences for convenience. We use the notation $\widetilde{\varphi}$ to abbreviate a sequence $\varphi_1, \ldots \varphi_k$, where k is given by $|\widetilde{\varphi}|$. Thus in the object $\widetilde{\theta}[\widetilde{\ell} \Mapsto \varsigma(x)\widetilde{(y)b}]$, the variable x abstracts "self"; for each label $\ell_i \in \widetilde{\ell}$, the name θ_i is a veil for the method identified by that label; the method's body b_i takes the parameter y_i. The maps are sometimes made explicit by writing the object as $\Theta[\varsigma(x)\Delta]$, where $\Theta(\ell_i) = \theta_i$ and $\Delta(\ell_i)(y_i) = b_i$ for each $\ell_i \in \widetilde{\ell}$.

There are separate "internal" and "external" primitives in the syntax for method call and update. The internal primitives $\ell(u)$ and $\ell \Leftarrow\!\!\!| (y)b$ noticeably do not carry any object reference (*cf.* the forms $p.\ell$ and $p.\ell \Leftarrow\!\!\!| b$ in $\mathbf{conc}\varsigma$ [14]). Labels by themselves have no meaning outside objects; hence the use of internal primitives is limited to within objects. The external primitives, on the other hand, can be used in any context. An external method call $\mathfrak{d}\langle u \rangle$ is a message on a veil. Crucially, an object reference is not required for invoking a method (*cf.* [14]). Object references are required for administration. Administration is done via external update $u \leftarrow\!\!\!| d$, which is a generalization of $\mathbf{conc}\varsigma$'s method update: it modifies an object by re-exporting veil bindings and augmenting/overriding several method definitions.

The rest of the syntax is the same as that of $\mathbf{conc}\varsigma$. Informally, the syntactical forms have the following meanings. (The formal semantics is shown later in the section.)

- u is a result (a variable or name) that is returned by an expression.
- $p \mapsto d$ attaches the capability p to an object d.
- $(\nu n)\, a$ creates a new name n that is bound in the expression a, and executes a.
- $a \Rsh b$ is the (non-commutative) parallel composition of the expressions a and b; it returns any result returned by b, while executing a for side-effect. This form, introduced in [14], is largely responsible for the compactness of the syntax, since it provides an uniform way to write expressions that return results, and "processes" that exhibit behaviours. (Of course, expressions that return results can also have side-effects.)
- let $x = a$ in b binds the result of the expression a to the variable x and then executes the expression b; here x is bound in b.
- $\ell(u)$ means a local method call inside an object; see external call.
- $\ell \Leftarrow\!\!\!| (y)b$ means a local method update inside an object; see external update.
- $\mathfrak{d}\langle u \rangle$ means an external call on the veil v, with argument u; in the presence of a denomination $p \mapsto d$ where d exports v for a defined method, the corresponding method expression is exported by substituting veiled calls for internal calls, self updates for internal updates, p for the abstracted self variable, and u for the formal parameter. The details of method export are given below.
- $u \leftarrow\!\!\!| d$ means an external update on the capability u; in the presence of a denomination $u \mapsto d'$, the veils exported by d replace those exported by d', and the methods defined by d augment or override those defined by d'; the capability u is returned.

Example 1. Assume that integers can be encoded in the language, and there is a method handle $\theta_{\texttt{pre}}$ for decrementing a positive integer. Consider the following code. A server creates a new timer object, exports the tick and set methods of the timer on veils $\theta_{\texttt{tick}}$ and $\theta_{\texttt{set}}$, and sets the value of the timer to an integer N by invoking $\theta_{\texttt{set}}$. A client repeatedly ticks the timer by invoking $\theta_{\texttt{tick}}$. At some point, the server creates a new veil $\theta_{\texttt{ntick}}$, and re-exports the tick method of the timer object on this veil. Consequently, since the client does not know this new veil, it can no longer tick the timer. (We elide unnecessary self-bindings $\varsigma(x)$, formal parameters (y), and unit arguments in the code.)

$$
\begin{array}{rll}
\text{System} \stackrel{\text{def}}{=} & (\nu p, \theta_{\texttt{val}}, \theta_{\texttt{set}}, \theta_{\texttt{tick}})\ (\text{Server} \Rsh \text{Client}) & \\
\text{Server} \stackrel{\text{def}}{=} & p \mapsto \theta_{\texttt{val}}\theta_{\texttt{set}}\theta_{\texttt{tick}}[\ \texttt{val} \Mapsto \texttt{val}, & \text{\# timer on capability } p, \text{ with} \\
& \quad \texttt{set}(y) \Mapsto \text{let } \_ = \texttt{val} \Lleftarrow y \text{ in } y, & \text{\# } \texttt{set} \text{ exported on veil } \theta_{\texttt{set}} \\
& \quad \texttt{tick} \Mapsto \text{let } z = \texttt{val} \text{ in} & \text{\# } \texttt{tick} \text{ exported on veil } \theta_{\texttt{tick}} \\
& \qquad\qquad \text{let } z' = \widehat{\theta_{\texttt{pre}}}\langle z \rangle \text{ in } \texttt{set}(z')\ ] \Rsh & \\
& \widehat{\theta_{\texttt{set}}}\langle N \rangle \Rsh \ldots \Rsh & \text{\# timer gets activated...} \\
& (\nu\theta_{\texttt{ntick}})\ p \longleftarrow \theta_{\texttt{val}}\theta_{\texttt{set}}\theta_{\texttt{ntick}}[\ \texttt{val} \Mapsto \texttt{val}\ ] & \text{\# timer gets deactivated} \\
\text{Client} \stackrel{\text{def}}{=} & (\widehat{\theta_{\texttt{tick}}} \Rsh \ldots \Rsh \widehat{\theta_{\texttt{tick}}}) & \text{\# timer ticks}
\end{array}
$$

We show a chemical semantics for the language, much as in [14]. Following the presentation in [13], we employ a grammar of evaluation contexts to tighten the rules.

$\mathcal{E} ::=$	evaluation contexts
$\bullet$	hole
let $x = \mathcal{E}$ in b	evaluation
$\mathcal{E} \Rsh b$	fork side
$a \Rsh \mathcal{E}$	fork main
$(\nu n)\ \mathcal{E}$	restriction

Informally, an evaluation context is an expression container with exactly one hole. By plugging an expression a into the hole of an evaluation context $\mathcal{E}$, we obtain the expression $\mathcal{E}[\![a]\!]$. (In general, plugging may not be capture-free with respect to names or variables.) We define structural congruence of expressions as usual.

Structural congruence $a \equiv b$ # fn (*resp.* bn) collects free (*resp.* bound) names

$$
\frac{n \notin \mathsf{fn}(\mathcal{E}) \cup \mathsf{bn}(\mathcal{E})}{(\nu n)\ \mathcal{E}[\![a]\!] \equiv \mathcal{E}[\![(\nu n)\ a]\!]}\ (\textsc{Struct Res}) \qquad \frac{\mathsf{fn}(a) \cap \mathsf{bn}(\mathcal{E}) = \varnothing}{a \Rsh \mathcal{E}[\![b]\!] \equiv \mathcal{E}[\![a \Rsh b]\!]}\ (\textsc{Struct Par})
$$

(STRUCT EQV) $\equiv$ is an equivalence

Next, we define reduction of expressions. Not surprisingly, perhaps, there are no reduction rules for internal call and update: we restrict the sites of action to the external primitives. The reductions for external call and update, (Red Call) and (Red Upd), have some important differences from the corresponding reductions in $\mathbf{conc}_{\varsigma}$. First, when a method body is exported on call reduction, the free labels in the body are "frozen" via substitutions of veiled calls for internal calls, and self updates for internal updates. The export translation $\downarrow_x^{\Theta}$ is shown below. Second, while object update is a straightforward generalization of method update, such an update also re-exports veil bindings for the methods of the object. In general, the update can block or unblock method calls that are

invoked on past or present veils: thus it serves as an access control mechanism in the language. Update is therefore synonymous with *administration* in this paper.

In the following, we use the notation Δ', Δ to mean the map obtained by augmenting the map Δ with Δ', while overriding bindings for the common labels.

Export $a\downarrow_x^\Theta$ and structural reduction $a \longrightarrow b$

$$\frac{\Theta(\ell) = v}{\ell(u)\downarrow_x^\Theta \stackrel{\text{def}}{=} \eth\langle u\rangle} \qquad (\ell \Leftarrow (y)b)\downarrow_x^\Theta \stackrel{\text{def}}{=} x \hookleftarrow \Theta[\ell \Mapsto \varsigma(x)(y)b\downarrow_x^\Theta]$$

$$(a \Rsh b)\downarrow_x^\Theta \stackrel{\text{def}}{=} a\downarrow_x^\Theta \Rsh b\downarrow_x^\Theta \qquad (\text{let } z = a \text{ in } b)\downarrow_x^\Theta \stackrel{\text{def}}{=} \text{let } z = a\downarrow_x^\Theta \text{ in } b\downarrow_x^\Theta$$

$$((\nu n)\, a)\downarrow_x^\Theta \stackrel{\text{def}}{=} (\nu n)\, a\downarrow_x^\Theta \qquad \frac{a = u,\ p \mapsto d,\ \eth\langle u\rangle, \text{ or } u \hookleftarrow d}{a\downarrow_x^\Theta \stackrel{\text{def}}{=} a}$$

(RED CALL)
$$\frac{d = \Theta[\varsigma(x)\Delta] \quad \Theta(\ell_j) = \theta \quad \Delta(\ell_j)(y) = b}{(p \mapsto d) \Rsh \eth\langle m\rangle \longrightarrow (p \mapsto d) \Rsh b\downarrow_x^\Theta\{p/x, m/y\}}$$

(RED UPD)
$$\frac{d = \_[\varsigma(x)\Delta] \quad d' = \Theta'[\varsigma(x)\Delta'] \quad d'' = \Theta'[\varsigma(x)\Delta', \Delta]}{(p \mapsto d) \Rsh p \hookleftarrow d' \longrightarrow (p \mapsto d'') \Rsh p}$$

(RED EVAL)
$$\text{let } x = n \text{ in } b \longrightarrow b\{n/x\}$$

(RED CONTEXT)
$$\frac{a \longrightarrow b}{\mathcal{E}[\![a]\!] \longrightarrow \mathcal{E}[\![b]\!]}$$

(RED STRUCT)
$$\frac{a \equiv a' \qquad a' \longrightarrow b' \qquad b' \equiv b}{a \longrightarrow b}$$

Freezing labels (by a veil map) in the export translation makes intuitive sense: it assigns a definite meaning to a method expression outside the syntactical scope of its parent object. Freezing labels also facilitates the enforcement of static object invariants (Section 3) amidst runtime administration; indeed labels in isolation cannot provide any runtime access guarantees.

Notice that an update returns the object reference (as in [14]): therefore, say, if an internal update is the rightmost branch of a method definition, a call to the method might return a reference to its parent object. This result is potentially dangerous—a user of the method can obtain administrative abilities on the object. We however do not complicate the semantics to prevent such "errors", partly because they are easy to catch statically. The update in question can of course be localized by a "let" if necessary.

To illustrate the semantics, next we show some sample reductions for parts of the code of Example 1. Here, let $\Theta(\texttt{tick}) = \theta_{\texttt{tick}}$, $\Theta'(\texttt{tick}) = \theta_{\texttt{ntick}}$, $\Delta(\texttt{val}) = \texttt{val}$, $\Delta'(\texttt{val}) = N$, $\Delta''(\texttt{val}) = N - 1$, and let the remaining bindings be as given by the initial denomination of p in the code.

$$\begin{aligned}
p \mapsto \Theta[\varsigma(x)\Delta] \Rsh \grave{\theta}_{\texttt{set}}\langle N\rangle \longrightarrow\ & p \mapsto \Theta[\varsigma(x)\Delta] \Rsh \\
& \quad \text{let } _ = p \hookleftarrow \Theta[\texttt{val} \Mapsto N] \text{ in } N \\
\longrightarrow\ & p \mapsto \Theta[\varsigma(x)\Delta'] \Rsh N && \text{\# activate} \\
p \mapsto \Theta[\varsigma(x)\Delta'] \Rsh \grave{\theta}_{\texttt{tick}} \longrightarrow\ & p \mapsto \Theta[\varsigma(x)\Delta'] \Rsh \\
& \quad \text{let } z = \grave{\theta}_{\texttt{val}} \text{ in let } z' = \grave{\theta}_{\texttt{pre}}\langle z\rangle \text{ in } \grave{\theta}_{\texttt{set}}\langle z'\rangle \\
\longrightarrow^\star\ & p \mapsto \Theta[\varsigma(x)\Delta''] \Rsh N - 1 && \text{\# tick} \\
p \mapsto \Theta[\varsigma(x)\Delta''] \Rsh p \hookleftarrow \Theta'[\texttt{val} \Mapsto \texttt{val}] \longrightarrow\ & p \mapsto \Theta'[\varsigma(x)\Delta] \Rsh p && \text{\# deactivate}
\end{aligned}$$

3 Flux-robust typing

In this section we show a type discipline for systems with concurrent objects that export dynamic "type views". More specifically, we allow methods to change types at runtime: the type of a method corresponds dynamically to the type of the veil it exports. For example, suppose the owner of a file wants to change the type of the content from "public" to "secret". Clearly, the veil for the content field must be changed: while the previous veil could have been public, the new veil has to be secret. If the file additionally has read and write methods that depend on the content field, their types change accordingly: therefore the veils for these methods need to be changed as well.

Changing veils is however not enough for end-to-end secrecy. (This inadequacy is typical of access control mechanisms, as mentioned in Section 1.) A user who can now read the file on its new veil will regard the content as secret (even if it is not). Suppose that the user reads the (previously public) content θ on the new veil, and exports θ as a handle to read a secret key k that he has written to another file: it now becomes possible to publicly read k by invoking θ. Indeed, it is almost always possible to exploit such "type interpretation" errors to leak secrets. (For instance, interpreting secret content as public can be equally bad.) To prevent such errors, the content field must be overridden to reflect its new type. By the same argument, then, it appears that the read and write methods need to be overridden as well—we can however do better. Typically read and write have types that are parametric with respect to the type of the content: informally, whenever the content type is X, the read and write methods have types $(\mathbf{1})X$ and $(X)\mathbf{1}$ (where $\mathbf{1}$ is the unit type). Therefore, those methods reflect their new types as soon as the content field is overridden.

We summarize these insights in the following general principles that govern the type system below. First, an object update is consistent only if the types of the new veils match up with the types of the method definitions. Second, type consistency forces some methods to be overridden; methods whose types are parametric with respect to the types of the overridden methods however need not be overridden themselves. This form of polymorphism is typically exhibited by higher-order (generic) functions, compositionally defined procedures, or (in the degenerate case) methods that have static types, *i.e.*, whose types do not change. We prefer to call these methods "natural" to avoid nomenclatural confusion with any particular brand of polymorphism.

3.1 A type system for secrecy despite flux

The primary goal of the type discipline is *flux robustness*, *i.e.*, type safety despite dynamic changes to type assumptions for methods. Access control is used in an integral way to enforce safety. In the type system, methods are qualified as "flat" or "natural". Flat methods must be overridden whenever veils change. Natural methods may be overridden; if they are, they must remain polymorphically typed, as indicated above.

To specify and verify secrecy, we introduce a system of principals. More specifically, we use indices to identify "owners" of code, and let the type declaration for a name specify the group of indices within which that name is intended to be confined. We then use type safety to verify that each such intention is preserved at runtime.

Secrecy groups as presented are close to those developed in [10]; the basic concepts appear earlier in, *e.g.*, the pi calculus with group creation [9] and the confined lambda calculus [23]. Let ∞ be a countable universe of indices—this is the largest group, also called "public", since a name that belongs to this group may be shared by all principals. Other groups (trusted) are proper subsets of ∞, or group variables (ranged over by $\mathcal{X}$).

$\rho ::=$	qualifiers
$\flat$	flat method
$\natural$	natural method
$\mathcal{H}, \mathcal{I} ::=$	groups
∞	countable universe of indices (public)
$\mathcal{G}$	trusted
$\mathcal{G} ::=$	trusted groups
$\mathcal{X}$	group variable
$\{\ldots\}$	proper subset of ∞
$S, T, U ::=$	types
X	type variable
$\mathsf{Obj}^{\mathcal{G}}[\widetilde{\ell} : \widetilde{(S)T^{\rho}}]$	capability type scheme
$\mathsf{Veil}^{\mathcal{G}}(u.\ell : (S)T)$	veil type
$(\exists x)T$	dependent union type
Null	null type
Un	untrusted type

Typed processes declare types for new names (with $(\nu n : T)\ a$, instead of $(\nu n)\ a$ in Section 2). Informally, the type sorts have the following meanings:

- X ranges over type variables. Group and type variables appear in capability signatures (see below).
- A capability signature $\mathsf{Obj}^{\mathcal{G}}[\widetilde{\ell} : \widetilde{(S)T^{\rho}}]$ is a type scheme that assigns types $(S_i)T_i$ and qualifiers ρ_i to the methods $\ell_i \in \widetilde{\ell}$ of a denoted object. The group $\mathcal{G}$ corresponds to the set of administrators for that object. The scheme binds group and type variables that are shared by the types of the methods in the signature. We interpret a type scheme as an universally quantified type over its bound variables, while leaving the bound variables implicit (*à la* polymorphic types in ML [26]).
- A veil type $\mathsf{Veil}^{\mathcal{G}}(u.\ell : (S)T)$ is dependent on a capability u, and instantiates the type scheme for a method ℓ in the signature of that capability. The veil expects an argument of type S and returns a result of type T. The group $\mathcal{G}$ corresponds to the set of users—the "access-control list"—for the method referenced by the veil. We use dependence in the veil type to prevent the same veil from being exported by different objects. (A similar "no-confusion" property is required, for instance, of datatype constructors [11].)
- Dependent union types $(\exists x)T$ allow capability dependencies to be passed without explicit communication of the capabilities themselves. The type system thus supports the separation of roles of veils and capabilities (as intended) despite enforcing necessary dependencies between them.
- The type Null is given to an expression whose result, if any, is ignored.
- Finally, the type Un is given to any expression whose result, if any, is untrusted.

For example, the signature of a file capability might look like:

$$\mathsf{Obj}^{\{\text{Owner}\}}[\texttt{content} : (\mathbf{1})X^{\flat}, \texttt{read} : (\mathbf{1})X^{\natural}, \texttt{write} : (X)\mathbf{1}^{\natural}]$$

where $\mathbf{1} \stackrel{\text{def}}{=} \mathsf{Obj}[\,]$. If, say, the content is of type T, a veil for `write` may have the type:

$$(\exists z)\mathsf{Veil}^{\{\text{Writer,Owner}\}}(z.\texttt{write} : (T)\mathbf{1})$$

As another example, an authenticated encryption object may be given the signature:

$$\mathsf{Obj}^{\{\text{KeyManager}\}}[\texttt{key} : X^{\flat}, \texttt{authencrypt} : (Y)(\exists z)\mathsf{Veil}^{\mathcal{X}}(z.\texttt{decrypt} : (X)Y)^{\natural}]$$

where the group $\mathcal{X}$ of the decryption handle returned by encryption can be controlled by KeyManager. Let, *e.g.*, $R \stackrel{\text{def}}{=} \{\text{Reader}, \text{KeyManager}\}$, $W \stackrel{\text{def}}{=} \{\text{Writer}, \text{KeyManager}\}$, $RW \stackrel{\text{def}}{=} R \cup W$, key type $T_X \stackrel{\text{def}}{=} \mathsf{Obj}^{R}[\,]$, and content type $T_Y \stackrel{\text{def}}{=} \mathsf{Obj}^{RW}[\,]$. Then `authencrypt` may be exported on a veil with type

$$(\exists z')\mathsf{Veil}^{W}(z'.\texttt{authencrypt} : (T_Y)(\exists z)\mathsf{Veil}^{RW}(z.\mathsf{decrypt} : (T_X)T_Y))$$

The relationship between types and groups is made explicit by a *reach* function, defined below. Informally, the reach of a type is the group within which the inhabitants of that type may be shared (but not without). For example, Un has reach ∞. Group and type variables do not constrain the reach of the type they appear in. Otherwise, the topmost group that appears in a type is taken to be the reach of that type.

Type reach $\|T\|$ with group variables $\mathcal{X}$ equated to ∞

$$\|X\| = \infty \qquad \|(\exists x)T\| = \|T\| \qquad \|\mathsf{Un}\| = \infty \qquad \|\mathsf{Obj}^{\mathcal{G}}[\_ : \_]\| = \mathcal{G} \qquad \|\mathsf{Veil}^{\mathcal{G}}(\_ : \_)\| = \mathcal{G}$$

Let σ range over substitutions of group and type variables, that is, $\sigma : (\mathcal{X} \to \mathcal{H}) + (X \to T)$. We define substitution below; it is mostly standard, except for the substitution of ∞ for a group variable that annotates a type, which "rounds off" that type as untrusted. We say that σ is a *proper* substitution for U if $U\sigma$ is defined.

Group and type substitution $\mathcal{G}\sigma$, $U\sigma$

$$\mathcal{X}\sigma = \sigma(\mathcal{X}) \qquad \{\ldots\}\sigma = \{\ldots\} \qquad X\sigma = \sigma(X) \qquad ((\exists x)T)\sigma = (\exists x)T\sigma \qquad \mathsf{Un}\,\sigma = \mathsf{Un}$$

$$\frac{\mathcal{G}\sigma \neq \infty}{\mathsf{Obj}^{\mathcal{G}}[\widetilde{\ell} : \widetilde{(S)T^{\rho}}]\sigma = \mathsf{Obj}^{\mathcal{G}\sigma}[\widetilde{\ell} : \widetilde{(S\sigma)T\sigma^{\rho}}]} \qquad \frac{\mathcal{G}\sigma = \infty \qquad \widetilde{S}\sigma = \mathsf{Un} \qquad \widetilde{T}\sigma = \mathsf{Un}}{\mathsf{Obj}^{\mathcal{G}}[\widetilde{\ell} : \widetilde{(S)T^{\rho}}]\sigma = \mathsf{Un}}$$

$$\frac{\mathcal{G}\sigma \neq \infty}{\mathsf{Veil}^{\mathcal{G}}(u.\ell : (S)T)\sigma = \mathsf{Veil}^{\mathcal{G}\sigma}(u.\ell : (S\sigma)T\sigma)} \qquad \frac{\mathcal{G}\sigma = \infty \qquad S\sigma = \mathsf{Un} \qquad T\sigma = \mathsf{Un}}{\mathsf{Veil}^{\mathcal{G}}(u.\ell : (S)T)\sigma = \mathsf{Un}}$$

Next, we show typing rules. Let Γ be a sequence of type assumptions $u : T$. The rules judge well-formed assumptions $\Gamma \vdash \diamond$, good types $\Gamma \vdash T$, good subtyping $\Gamma \vdash T <: U$, and well-typed expressions $\Gamma \vdash_{\mathcal{I}} a : T$. In the rules for $\Gamma \vdash T <: U$, we implicitly include the condition $\Gamma \vdash T$ in the antecedent. In the rules for $\Gamma \vdash T$ and $\Gamma \vdash_{\mathcal{I}} a : T$, whenever there are no $\vdash$ judgments in the antecedent, we implicitly include the condition $\Gamma \vdash \diamond$.

In the judgment $\Gamma \vdash_{\mathcal{I}} a : T$, the group $\mathcal{I}$ under $\vdash$ indicates the "trust level" under which the program is to be typed: any result in the program must have a type whose reach intersects $\mathcal{I}$. Informally, principals with indices in $\mathcal{I}$ are allowed to collude—their programs may contain results that have types whose reaches include at least one of the indices in $\mathcal{I}$, but may not contain any result whose type is "out of reach" (*i.e.*, whose reach does not include any index in $\mathcal{I}$). For instance, $\vdash_\infty$ is the most liberal typing relation. In fact $\vdash$ is monotone: $\mathcal{H} \subseteq \mathcal{I}$ implies $\vdash_{\mathcal{H}} \subseteq \vdash_{\mathcal{I}}$.

Let $\mathsf{Veil}^{\mathcal{G}}(u.\ell : (S)T) \in \mathsf{veiltype}(u.\ell : (S)T)$ for all trusted groups $\mathcal{G}$, and $\mathsf{Un} \in \mathsf{veiltype}(u.\ell : (\mathsf{Un})\mathsf{Un})$. Finally, let σ_∞ range over special substitutions that map group variables to ∞ and type variables to Un, that is, $\sigma_\infty : (\mathcal{X} \to \infty) + (X \to \mathsf{Un})$.

Typing rules $\Gamma \vdash \diamond,\ \Gamma \vdash T,\ \Gamma \vdash T <: U,\ \Gamma \vdash_{\mathcal{I}} a : T$

(HYP NONE)
$$\varnothing \vdash \diamond$$

(HYP NEXT)
$$\frac{\Gamma \vdash T \qquad u \notin \mathsf{dom}(\Gamma)}{\Gamma, u : T \vdash \diamond}$$

(TYP VAR)
$$\Gamma \vdash X$$

(TYP EXST)
$$\frac{\Gamma, x : X \vdash T}{\Gamma \vdash (\exists x) T}$$

(TYP OBJ)
$$\frac{\widetilde{\ell}\ \text{distinct} \qquad \forall \{\ell_i \in \widetilde{\ell}\}.\ \Gamma \vdash S_i, T_i}{\Gamma \vdash \mathsf{Obj}^{\mathcal{G}}[\widetilde{\ell} : \widetilde{(S)T^{\rho}}]}$$

(TYP VEIL)
$$\frac{u \in \mathsf{dom}(\Gamma) \qquad \Gamma \vdash S, T}{\Gamma \vdash \mathsf{Veil}^{\mathcal{G}}(u.\ell : (S)T)}$$

(TYP UN)
$$\Gamma \vdash \mathsf{Un}$$

(SUB REFL)
$$\Gamma \vdash T <: T$$

(SUB TRAN)
$$\frac{\Gamma \vdash T <: S \qquad \Gamma \vdash S <: U}{\Gamma \vdash T <: U}$$

(EXP SUB)
$$\frac{\Gamma \vdash_{\mathcal{I}} a : T \qquad \Gamma \vdash T <: U}{\Gamma \vdash_{\mathcal{I}} a : U}$$

(DEP GRNT)
$$\frac{x \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash T\{u/x\} <: (\exists x) T}$$

(DEP ASSM)
$$\frac{x \text{ not free in } U \qquad X \text{ fresh} \qquad \Gamma, x : X, u : T \vdash_{\mathcal{I}} a : U}{\Gamma, u : (\exists x) T \vdash_{\mathcal{I}} a : U}$$

(SUB OBJ)
$$\frac{\widetilde{\ell} : \widetilde{(S)T^{\rho}} \subseteq \widetilde{\ell'} : \widetilde{(S')T'^{\rho'}}}{\Gamma \vdash \mathsf{Obj}^{\mathcal{G}}[\widetilde{\ell'} : \widetilde{(S')T'^{\rho'}}] <: \mathsf{Obj}^{\mathcal{G}}[\widetilde{\ell} : \widetilde{(S)T^{\rho}}]}$$

(NULL EXP)
$$\Gamma \vdash T <: \mathsf{Null}$$

(EXP RES)
$$\frac{\mathcal{I} \cap \|T\| \neq \varnothing \qquad \Gamma(u) = T}{\Gamma \vdash_{\mathcal{I}} u : T}$$

(EXP NEW)
$$\frac{\Gamma, n : T \vdash_{\mathcal{I}} a : U}{\Gamma \vdash_{\mathcal{I}} (\nu n : T)\ a : U}$$

(EXP FORK)
$$\frac{\Gamma \vdash_{\mathcal{I}} a : T \qquad \Gamma \vdash_{\mathcal{I}} b : U}{\Gamma \vdash_{\mathcal{I}} a \Rsh b : U}$$

(EXP EVAL)
$$\frac{\Gamma \vdash_{\mathcal{I}} a : T \qquad \Gamma, x : T \vdash_{\mathcal{I}} b : U}{\Gamma \vdash_{\mathcal{I}} \mathsf{let}\ x = a\ \mathsf{in}\ b : U}$$

(EXP CALL)
$$\frac{\Gamma \vdash_{\mathcal{I}} v : \mathsf{Veil}^{\mathcal{G}}(w.\ell : (S)T) \qquad \Gamma \vdash_{\mathcal{I}} u : S \qquad \mathcal{I} \cap \|T\| \neq \varnothing}{\Gamma \vdash_{\mathcal{I}} \mathfrak{d}\langle u \rangle : T}$$

(EXP CALL UN)
$$\frac{\Gamma \vdash_{\mathcal{I}} v : \mathsf{Un} \qquad \Gamma \vdash_{\mathcal{I}} u : \mathsf{Un}}{\Gamma \vdash_{\mathcal{I}} \mathfrak{d}\langle u \rangle : \mathsf{Un}}$$

(NULL DEN)
$$\frac{\begin{array}{c}\Gamma \vdash_{\mathcal{I}} p : \mathsf{Obj}^{\mathcal{G}}[\widetilde{\ell} : \widetilde{(S)T^{\rho}}] \qquad \mathsf{dom}(\Theta) \cup \mathsf{dom}(\Delta) \subseteq \widetilde{\ell} \qquad \widetilde{(S)T}\sigma = \widetilde{(S')T'} \\ \forall \{\ell_i \in \mathsf{dom}(\Theta)\}.\ \Gamma \vdash_{\mathcal{I}} \Theta(\ell_i) : U_i' \in \mathsf{veiltype}(p.\ell_i : (S_i')T_i') \\ \forall \{\ell_i \in \mathsf{dom}(\Delta)\}.\ \Gamma, y_i : S_i' \vdash_{\mathcal{I}} \Delta(\ell_i)(y_i) \lightning_x^{\Theta} \{p/x\}\sigma : T_i' \\ \forall \{\ell_i \in \mathsf{dom}(\Delta) \mid \rho_i = \natural\}.\ \forall \sigma_\infty. \\ \Gamma, \widetilde{z} : \widetilde{U} \in \mathsf{veiltype}(p.\widetilde{\ell} : \widetilde{(S)T\sigma_\infty}), y_i : S_i\sigma_\infty \vdash_{\mathcal{I}} \Delta(\ell_i)(y_i) \lightning_x^{\widetilde{z}/\widetilde{\ell}} \{p/x\}\sigma_\infty : T_i\sigma_\infty\end{array}}{\Gamma \vdash_{\mathcal{I}} p \mapsto \Theta[\varsigma(x)\Delta] : \mathsf{Null}}$$

(DEN UN)

$$\frac{\Gamma \vdash_{\mathcal{I}} p : \mathsf{Un} \qquad \forall\{\ell_i \in \mathsf{dom}(\Theta)\}.\ \Gamma \vdash_{\mathcal{I}} \Theta(\ell_i) : \mathsf{Un} \qquad \forall\{\ell_i \in \mathsf{dom}(\Delta)\}.\ \Gamma, y_i : \mathsf{Un} \vdash_{\mathcal{I}} \Delta(\ell_i)(y_i) \not\downarrow^{\Theta}_{x} \{p/x\} : \mathsf{Un}}{\Gamma \vdash_{\mathcal{I}} p \mapsto \Theta[\varsigma(x)\Delta] : \mathsf{Un}}$$

(EXP UPD)

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathcal{I}} u : \mathsf{Obj}^{\mathcal{G}}[\widetilde{\ell} : \widetilde{(S)T^{\rho}}] \qquad \mathsf{dom}(\Theta) \cup \mathsf{dom}(\Delta) \subseteq \widetilde{\ell} \qquad \{\ell_i \mid \rho_i = \flat\} \subseteq \mathsf{dom}(\Delta) \\ \widetilde{(S)T}\sigma = \widetilde{(S')T'} \qquad \forall\{\ell_i \in \mathsf{dom}(\Theta)\}.\ \Gamma \vdash_{\mathcal{I}} \Theta(\ell_i) : U_i' \in \mathsf{veiltype}(u.\ell_i : (S_i')T_i') \\ \forall\{\ell_i \in \mathsf{dom}(\Delta)\}.\ \Gamma, y_i : S_i' \vdash_{\mathcal{I}} \Delta(\ell_i)(y_i) \not\downarrow^{\Theta}_{x} \{u/x\}\sigma : T_i' \\ \forall\{\ell_i \in \mathsf{dom}(\Delta) \mid \rho_i = \natural\}.\ \forall\sigma_{\infty}. \\ \Gamma, \widetilde{z} : \widetilde{U} \in \mathsf{veiltype}(u.\widetilde{\ell} : \widetilde{(S)T\sigma_{\infty}}), y_i : S_i\sigma_{\infty} \vdash_{\mathcal{I}} \Delta(\ell_i)(y_i) \not\downarrow^{\widetilde{z}/\widetilde{\ell}}_{x} \{u/x\}\sigma_{\infty} : T_i\sigma_{\infty}\end{array}}{\Gamma \vdash_{\mathcal{I}} u \leftarrowtail \Theta[\varsigma(x)\Delta] : \mathsf{Obj}^{\mathcal{G}}[\widetilde{\ell} : \widetilde{(S)T^{\rho}}]}$$

(EXP UPD UN)

$$\frac{\Gamma \vdash_{\mathcal{I}} u : \mathsf{Un} \qquad \forall\{\ell_i \in \mathsf{dom}(\Theta)\}.\ \Gamma \vdash_{\mathcal{I}} \Theta(\ell_i) : \mathsf{Un} \qquad \forall\{\ell_i \in \mathsf{dom}(\Delta)\}.\ \Gamma, y_i : \mathsf{Un} \vdash_{\mathcal{I}} \Delta(\ell_i)(y_i) \not\downarrow^{\Theta}_{x} \{u/x\} : \mathsf{Un}}{\Gamma \vdash_{\mathcal{I}} u \leftarrowtail \Theta[\varsigma(x)\Delta] : \mathsf{Un}}$$

Notice that Null is not a "good" type—we have Null as a type only because it allows us to give compact rules for well-typed expressions. (Dep Assm) and (Dep Grnt) are standard assume/guarantee rules for propagating dependencies. (Den Un), (Exp Call Un), and (Exp Upd Un) can type arbitrary "untrusted" expressions whose names and type declarations are all public.

(Exp Call) checks that veil invocation is type-safe, *i.e.*, the type of the result matches that suggested by the veil type. (Exp Res) and (Exp Call) check if the typing group intersects the reach of an expected result type. These checks do not constrain irrelevant type assumptions, even if those types are out of reach of the typing group.

(Null Den) and (Exp Upd) are largely similar. There, σ ranges over proper substitutions for group and type variables that are bound by the capability signature. Additionally, σ_∞ ranges over all proper *partial* substitutions that map some of those variables to ∞ and Un. Both rules check if the capability signature is properly instantiated (via σ) by the types of the new veil bindings and the new method bodies. Crucially, every application of (Null Den) and (Exp Upd) can present a different instantiation for the type scheme of the same capability. This allows "dynamic specification" of type assumptions for the methods of an object. For those methods that are qualified natural, (Null Den) and (Exp Upd) also check if the method bodies can be typed polymorphically to match their type schema, with fresh veil bindings and partially instantiated types (via σ_∞). The checks are necessary because we do not require natural methods to be overridden on each update of the object, yet want them to be robust to any changes in type assumptions within the object. (Indeed, (Exp Upd) requires only those methods that are qualified flat to be overridden on update.) There are only finitely many σ_∞ to consider (since there are only finitely many bound group and type variables in the signature). Group variables suffice to account for instantiations with trusted groups; the substitutions σ_∞ account for those instantiations that may map some group variables to ∞, thereby collapsing some types to Un and changing type structure. (Similar subtleties appear in a secrecy type system for asymmetric communication [1] while exploiting polymorphism across trusted types and Un.)

Example 2. Assume that name matching can be encoded in the language. Recall the example with authenticated encryption objects. Assume that p has the shown signature, $\theta_{\mathtt{enc}}$ has the shown veil type, k has type T_X, and $\theta_{\mathtt{key}}$ has type $(\exists z')\mathsf{Veil}^R(z'.\mathtt{key} : T_X)$. Then the following denomination is well-typed under $\vdash_{\{\text{KeyManager}\}}$. A proper σ_∞ that must be considered when typechecking `authencrypt` maps $\mathcal{X}$ to ∞, and X and Y to Un; another such is the empty substitution that does not instantiate any variable.

$$
\begin{array}{l}
p \mapsto \theta_{\mathtt{key}}\theta_{\mathtt{enc}}[\,\mathtt{key} \Mapsto k, \mathtt{authencrypt}(y) \Mapsto \\
\quad (\nu q : \mathsf{Obj}^{\mathcal{X}}[\mathtt{decrypt} : (X)Y^{\natural}])\ (\nu\theta_{\mathtt{dec}} : \mathsf{Veil}^{\mathcal{X}}(q.\mathtt{decrypt} : (X)Y)) \\
\quad\quad \text{let } x' = \mathtt{key} \text{ in } q \mapsto \theta_{\mathtt{dec}}[\,\mathtt{decrypt}(x) \Mapsto \text{if } x \text{ is } x' \text{ then } y\,]^{\Rsh}\ \theta_{\mathtt{dec}}\,]
\end{array}
$$

Next, say we can type Reader's code a_r under $\vdash_{\{\text{Reader}\}}$ and Writer's code a_w under $\vdash_{\{\text{Writer}\}}$. Informally, a_r can obtain the key k by invoking $\theta_{\mathtt{key}}$; a_w can encrypt a term of type T_Y by invoking $\theta_{\mathtt{enc}}$, and share the resulting decryption handle $\theta_{\mathtt{dec}}$ with a_r; and a_r can retrieve the encrypted term by invoking $\theta_{\mathtt{dec}}$ with argument k. However, a_r can never encrypt a term with $\theta_{\mathtt{enc}}$, and a_w can never decrypt a term encrypted with $\theta_{\mathtt{enc}}$.

Example 3. Suppose that Bonnie and Clyde wish to start a session sometime in the future, with a session secret generated by Clyde. Moreover, they wish to use a file p owned by Bonnie to establish that secret when they are ready. We show a safe, well-typed protocol in which the file is used in at least three different ways over time. Bonnie initializes the file content to a new name $\theta_{\mathtt{nw}}$, binding its access to a name $\theta_{\mathtt{c}}$ known to Clyde; the content $\theta_{\mathtt{nw}}$ is a future write handle to the file. Additionally, she programs the file to transition into a publicly usable phase as soon as that content is read off (since she has other tasks for the file). Since Clyde knows $\theta_{\mathtt{c}}$, he can read the file to obtain $\theta_{\mathtt{nw}}$. Later, Bonnie brings the file back into restricted usage, with $\theta_{\mathtt{nw}}$ as its new write handle. She then listens for the secret she expects from Clyde. Accordingly Clyde creates a new secret and writes it to the file by invoking (the earlier obtained) $\theta_{\mathtt{nw}}$. Both Bonnie and Clyde now share the new secret, and can safely start their session.

Let $B \stackrel{\text{def}}{=} \{\text{Bonnie}\}$, $C \stackrel{\text{def}}{=} \{\text{Clyde}\}$, $BC \stackrel{\text{def}}{=} B \cup C$, $\mathbf{1} \stackrel{\text{def}}{=} \mathsf{Un}$, $\mathsf{Sec}^{BC} \stackrel{\text{def}}{=} \mathsf{Obj}^{BC}[\,]$, and $a; b \stackrel{\text{def}}{=} \text{let } x = a \text{ in } b$ for fresh x. Assume $p : \mathsf{Obj}^{B}[\mathtt{content} : (\mathbf{1})X^{\flat}, \mathtt{write} : (X)\mathbf{1}^{\natural}]$, $\theta_{\mathtt{c}} : \mathsf{Veil}^{BC}(p.\mathtt{content} : (\mathbf{1})(\exists z)\mathsf{Veil}^{BC}(z.\mathtt{write} : (\mathsf{Sec}^{BC})\mathbf{1}))$, $\theta_{\mathtt{uc}} : \mathsf{Un}$, and $\theta_{\mathtt{uw}} : \mathsf{Un}$. Then we can type Bonnie's code b under $\vdash_B$ and Clyde's code c under $\vdash_C$.

$$
\begin{array}{rcl}
b & = & (\nu\theta_{\mathtt{nw}} : \mathsf{Veil}^{BC}(p.\mathtt{write} : (\mathsf{Sec}^{BC})\mathbf{1})) \\
& & (\nu\theta_{\mathtt{w}} : \mathsf{Veil}^{B}(p.\mathtt{write} : ((\exists z)\mathsf{Veil}^{BC}(z.\mathtt{write} : (\mathsf{Sec}^{BC})\mathbf{1}))\mathbf{1})) \\
& & \quad p \mapsto \theta_{\mathtt{c}}\theta_{\mathtt{w}}[\,\mathtt{content} \Mapsto \varsigma(x)\ x \leftarrowtail \theta_{\mathtt{uc}}\theta_{\mathtt{uw}}[\,\mathtt{content} \Mapsto \mathtt{content}\,]; \theta_{\mathtt{nw}}, \\
& & \quad\quad\quad \mathtt{write}(y) \Mapsto \mathtt{content} \Lsh y;\,]^{\Rsh} \\
& & \quad \ldots \\
& & \quad (\nu\theta_{\mathtt{nc}} : \mathsf{Veil}^{B}(p.\mathtt{content} : (\mathbf{1})\mathsf{Sec}^{BC})) \\
& & \quad\quad p \leftarrowtail \theta_{\mathtt{nc}}\theta_{\mathtt{nw}}[\,\mathtt{content} \Mapsto \mathtt{content}\,]; \\
& & \quad\quad \text{let } x = \widehat{\theta_{\mathtt{nc}}} \text{ in } \ldots \\
c & = & \text{let } x = \widehat{\theta_{\mathtt{c}}} \text{ in } (\nu k : \mathsf{Sec}^{BC})\ \hat{x}\langle k\rangle; \ldots
\end{array}
$$

3.2 Properties of well-typed code

The main result for the type system of Section 3 is that well-typed code never leaks secrets beyond declared boundaries, even under arbitrary untrusted environments. The

result relies on a standard but non-trivial preservation property: well-typed expressions preserve their types on execution. This property justifies our typing approach.

Proposition 1 (Preservation). *Let* $\Gamma \vdash_{\mathcal{I}} a : T$. *If* $a \longrightarrow b$, *then* $\Gamma \vdash_{\mathcal{I}} b : T$.

Additionally, the type system has two important properties. First, the type given to an expression is not beyond reach, *i.e.*, at least one index in the typing group falls within the reach of the expression type. (Additionally, reaches are preserved by subtyping.)

Proposition 2 (Reach soundness). *Let* $\Gamma \vdash_{\mathcal{I}} a : T \neq \mathsf{Null}$. *Then* $\|T\| \cap \mathcal{I} \neq \varnothing$.

Second, the type system can accommodate arbitrary expressions, as long as they do not contain trusted names. This property is important, since we cannot assume that attackers attempting to learn secrets would politely follow our typing discipline.

Proposition 3 (Typability). *Let* a *be any expression without free labels or variables. Suppose all declared types in* a *are* Un, *and* $\Gamma(n) = \mathsf{Un}$ *for all free names* n *in* a. *Then* $\Gamma \vdash_{\mathcal{I}} a : \mathsf{Un}$ *for all* $\mathcal{I}$.

Finally, we present the main result. Let a be trusted code typed under group $\mathcal{I}$, and b be (perhaps partially) untrusted code typed under the complement group $\infty - \mathcal{I}$. In general, b may be any adversarial code written jointly by an arbitrary attacker in collusion with trusted principals outside $\mathcal{I}$; the trusted part of b may even share trusted names with a. Then if the principals in $\mathcal{I}$ eventually declare an exclusive secret n, this secret can never be learnt by executing b in composition with a.

Theorem 1 (Secrecy). *Let* $\Gamma \vdash_{\mathcal{I}} a : S$ *and* $\Gamma \vdash_{\infty - \mathcal{I}} b : T$. *If* $a \Rsh b \longrightarrow^{\star} (\nu \widetilde{m} : \widetilde{U'})\, (\nu n : U)\, c$ *such that* $\|U\| \subseteq \mathcal{I}$, *then* $c \not\longrightarrow^{\star} \_ \Rsh n$.

The proof is based on a simple argument: if n can be learnt, then T must be the same as U—but the reach of T must contain at least one index in $\infty - \mathcal{I}$, *i.e.*, outside $\mathcal{I}$ (contradiction). A weaker version of the theorem that deals with top-level secrets also holds: for all names m such that $\|\Gamma(m)\| \subseteq \mathcal{I}$, it must be the case that $a \Rsh b \not\longrightarrow^{\star} \_ \Rsh m$.

4 Conclusion

Static analyses have been quite helpful in guaranteeing high-level safety properties of distributed systems: indeed, a significant body of work focuses specifically on safe resource usage [13, 5, 22, 24, 25, 6–8, 29]. Some analyses use access levels, as declared via static type annotations, to guarantee the absence of access violations at runtime [21, 28, 7, 29]. In our previous work [10], we go further by studying the interplay of static secrecy specifications with dynamically acquired permissions, and verify that access checks help respect the specifications at runtime. A similar approach is reflected in hybrid typechecking [12], a type system for secure information flow in a Java-like language [3], and a type system that supports dynamic revocation [19].

An alternative, and perhaps more natural stance is to allow specifications to be inherently dynamic to reflect changing assumptions during execution. Dynamic specifications are often desirable when reasoning about resources in the long run. When additional runtime guarantees can be exploited, dynamic specifications typically also allow

finer analyses than static specifications. Along those lines, one body of work studies the enforcement of policies specified as security automata [31, 18]. Yet another studies systems with declassification, *i.e.*, conservative relaxation of secrecy assumptions at runtime [27]. There is also some recent work on compromised secrets [15, 17] in the context of network protocols. In comparison, our analyses apply more generally to changing assumptions at runtime. Perhaps closest to our work are analyses developed for dynamic access control in languages with locality and migration [20, 16]. Similar ideas also appear in a type system for noninterference that allows the use of dynamic security labels [33].

Our contributions in this paper are two-fold. We develop low-level access control features in an existing object language to make it suitable as a core calculus for studying security properties of concurrent, stateful resources. We then show a typing approach for verifying high-level intentions on resource manipulation in the resulting language. The type system allows dynamic access control specifications, and crucially relies on corresponding low-level guarantees provided by the language runtime to verify those specifications. This combination helps in developing precise security analyses for shared resources that are used under changing assumptions over time.

Acknowledgments. Martín Abadi suggested $\mathbf{conc}\varsigma$ as a possible starting point for the calculus. In addition, he and Cormac Flanagan helped with comments on an earlier draft. This work was partly supported by the National Science Foundation under Grants CCR-0208800 and CCF-0524078, and by Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratory under Contract B554869.

References

1. M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. *Theoretical Computer Science*, 298(3):387–415, 2003.
2. M. Abadi and L. Cardelli. An imperative object calculus. In *TAPSOFT'95: Theory and Practice of Software Development*, pages 471–485. Springer, 1995.
3. A. Banerjee and D. Naumann. Using access control for secure information flow in a Java-like language. In *CSFW'03: Computer Security Foundations Workshop*, pages 155–169. IEEE, 2003.
4. P. D. Blasio and K. Fisher. A calculus for concurrent objects. In *CONCUR'96: Concurrency Theory*, pages 655–670. Springer, 1996.
5. P. D. Blasio, K. Fisher, and C. Talcott. A control-flow analysis for a calculus of concurrent objects. *IEEE Transactions on Software Engineering*, 26(7):617–634, 2000.
6. C. Braghin, D. Gorla, and V. Sassone. A distributed calculus for role-based access control. In *CSFW'04: Computer Security Foundations Workshop*, pages 48–60. IEEE, 2004.
7. M. Bugliesi, G. Castagna, and S. Crafa. Access control for mobile agents: The calculus of boxed ambients. *ACM Transactions on Programming Languages and Systems*, 26(1):57–124, 2004.
8. M. Bugliesi, D. Colazzo, and S. Crafa. Type based discretionary access control. In *CONCUR'04: Concurrency Theory*, pages 225–239. Springer, 2004.
9. L. Cardelli, G. Ghelli, and A. D. Gordon. Secrecy and group creation. *Information and Computation*, 196(2):127–155, 2005.
10. A. Chaudhuri and M. Abadi. Secrecy by typing and file-access control. In *CSFW'06: Computer Security Foundations Workshop*. To appear. IEEE, 2006.

11. T. Coquand. Pattern matching with dependent types. In *Workshop on Types for Proofs and Programs*. Electronic proceedings, 1992.
12. C. Flanagan. Hybrid type checking. In *POPL '06: Principles of programming languages*, pages 245–256. ACM, 2006.
13. C. Flanagan and M. Abadi. Object types against races. In *CONCUR '99: Concurrency Theory*, pages 288–303. Springer, 1999.
14. A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. In *HLCL'98: High-Level Concurrent Languages*, pages 248–264. Elsevier, 1998.
15. A. D. Gordon and A. Jeffrey. Secrecy despite compromise: Types, cryptography, and the pi-calculus. In *CONCUR'05: Concurrency Theory*, pages 186–201. Springer, 2005.
16. D. Gorla and R. Pugliese. Resource access and mobility control with dynamic privileges acquisition. In *ICALP'03: International Colloquium on Automata, Languages, and Programming*, pages 119–132. Springer, 2003.
17. C. Haack and A. Jeffrey. Timed spi-calculus with types for secrecy and authenticity. In *CONCUR'05: Concurrency Theory*, pages 202–216. Springer, 2005.
18. K. W. Hamlen, G. Morrisett, and F. B. Schneider. Certified in-lined reference monitoring on .NET. In *PLAS'06: Programming Languages and Analysis for Security*. To appear. ACM, 2006.
19. C. Hawblitzel and T. von Eicken. Type system support for dynamic revocation. In *Workshop on Compiler Support for System Software*. Electronic proceedings, 1999.
20. M. Hennessy, M. Merro, and J. Rathke. Towards a behavioural theory of access and mobility control in distributed systems. In *FOSSACS'03: Foundations of Software Science and Computational Structures*, pages 282–298. Springer, 2003.
21. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. In *HLCL '98: High-Level Concurrent Languages*, pages 174–188. Elsevier, 1998.
22. M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. *ACM Transactions on Programming Languages and Systems*, 24(5):566–591, 2002.
23. Z. D. Kirli. Confined mobile functions. In *CSFW'01: Computer Security Foundations Workshop*, pages 283–294. IEEE, 2001.
24. J. Kleist and D. Sangiorgi. Imperative objects as mobile processes. *Science of Computer Programming*, 44(3):293–342, 2002.
25. G. Miklau and D. Suciu. Controlling access to published data using cryptography. In *VLDB'03: Very Large Data Bases*, pages 898–909. Springer, 2003.
26. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.
27. A. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *CSFW'04: Computer Security Foundations Workshop*, pages 172–186. IEEE, 2004.
28. R. D. Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for access control. *Theoretical Computer Science*, 240(1):215–254, 2000.
29. F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. *ACM Transactions on Programming Languages and Systems*, 27(2):344–382, 2005.
30. D. D. Redell. Naming and protection in extendible operating systems. Technical Report 140, Project MAC, MIT, 1974.
31. F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.
32. V. T. Vasconcelos. Typed concurrent objects. In *ECOOP'94: European Conference on Object-Oriented Programming*, pages 100–117. Springer, 1994.
33. L. Zheng and A. Myers. Dynamic security labels and noninterference. In *FAST'04: Formal Aspects in Security and Trust*, pages 27–40. Springer, 2004.