# Code-Carrying Authorization

Sergio Maffeis[2,3], Martín Abadi[1,2], Cédric Fournet[1], and Andrew D. Gordon[1]

[1] Microsoft Research
[2] University of California, Santa Cruz
[3] Imperial College London

**Abstract.** In authorization, there is often a wish to shift the burden of proof to those making requests, since they may have more resources and more specific knowledge to construct the required proofs. We introduce an extreme instance of this approach, which we call Code-Carrying Authorization (CCA). With CCA, access-control decisions can partly be delegated to untrusted code obtained at run-time. The dynamic verification of this code ensures the safety of authorization decisions. We define and study this approach in the setting of a higher-order spi calculus. The type system of this calculus provides the needed support for static and dynamic verification.

## 1 Introduction

The generation, transmission, and checking of evidence plays a central role in authorization. The evidence may include, for instance, certificates of memberships in groups, delegation assertions, and bindings of keys to principals. Typically, the checking is done dynamically, that is, at run-time, in reference monitors. When a reference monitor considers a request from a principal, it evaluates the evidence supplied by the principal in the context of a local policy and other information. It is also possible—and indeed attractive—to perform some of the checking statically, at the time of definition of a system. This static checking may rely on logical reasoning or on type systems, and may guarantee that enforcement of a policy is done thoroughly and correctly.

A growing body of research explores the idea that the evidence may include or may be organized as a logical proof [17,4,15,9,20]. For instance, in the special case of proof-carrying code (PCC), the proofs guarantee code safety, and the requests are typically for running a piece of code [17]. In another example, the clients of a web server may present proofs that their requests should be granted [5]. This idea provides a principled approach to authorization. It also provides an approach to auditing in which the proofs that motivate access-control decisions can be logged and analyzed [20]. While the burden of proof generation shifts to the principal that makes a request, the proof need not be trusted, so the reference monitor still needs to verify the proof. Dynamic proof verification may fail; accordingly, any static checking needs to accommodate this possibility.

Thus arises the question of how to reconcile static checking with proof-carrying and dynamic verification. As an interesting specific instance of this question, one may wonder how to incorporate dynamic verification in the existing typed spi calculus for authorization of Fournet et al. [12]. In that calculus, a static type system guarantees the safe enforcement of an authorization policy. It does not include proofs as first-class objects, nor the possibility of dynamic verification. One might think about adding proofs and proof-checking as primitives to this calculus, in order to support dynamic verification and authorization. While that idea may seem "natural", to our surprise we discovered that a more general idea is both technically cleaner and more powerful in supporting interesting authorization scenarios. With "Proof-Carrying Authorization" (PCA) [4] in mind, we call this idea "Code-Carrying Authorization" (CCA).

CCA consists in passing not proofs but pieces of code that perform run-time verification. These pieces of code are essentially fragments of a reference monitor. They are themselves checked dynamically, since in general they are not trusted. Analogously, the Open Verifier project [8] has started to explore a generalization of PCC in which mobile code is accompanied by untrusted verifiers.

Following the Curry-Howard isomorphism, one may view proofs as programs. Still, with PCA [4], those programs are only checked, not executed. With CCA, programs are executed as well, though in a controlled way. No additional language for proofs is needed; we can use arbitrary code, subject to dynamic typing. Thus, in comparison with PCA, CCA allows a more open-ended, flexible notion of evidence without requiring the introduction of special syntax.

In the present paper, we explore dynamic verification and authorization in the context of a typed spi calculus. Technically, this calculus is a higher-order spi calculus [3] with dynamic typing. Both the higher-order features and the dynamic typing rely on fairly standard constructs [19,2], though with some new technical complications and new applications. In particular, the dynamic typing can require theorem proving. The calculus includes only shared-key cryptography; further cryptographic primitives might be added as in later work by Fournet et al. [13]. Optionally, the calculus also includes first-class proof hints, which can alleviate or eliminate the theorem-proving task at the reference monitor. We prove results that establish the safety of authorization decisions with respect to policies. (The full version of this paper [16] contains detailed proofs.)

We exploit this calculus in a range of small but challenging examples. These examples illustrate some of the advantages of dynamic verification and of CCA in particular. For instance, in some of the examples, a server can enforce a rich authorization policy while having only simple, generic code; clients provide more detailed code for run-time access control. Such examples are beyond the scope of previous systems.

In addition to the research on PCA and on types for authorization cited above, our work is related to a broad range of applications of process calculi to security. These include, for instance, distributed pi calculi with trust relations and mobile code [18,14]. Interestingly, some of these calculi support remote attestation and dynamic subtyping checks (however, with rather different goals and type structures, and no typecase) [10].

## 2 A Spi Calculus with Dynamic Verification

In this section we review the calculus for authorization on which we build [12], and discuss our extensions for dynamic verification.

*Authorization Logics.* Our approach is parametric in the choice of an authorization logic used as a policy language. The only constraint on the logic is that it be monotonic and closed under substitution (see [16]). For example, Datalog [7], Binder [11], and CDD [1] are valid authorization logics. In the rest of the paper, we use Datalog as an authorization logic, and write $S \models C$ when policy $S$ entails the clause $C$. Informally, entailment means that access requests that depend on $C$ should be granted according to $S$.

Our running example is based on an electronic conference reviewing system. The conference server contains a policy that controls the access to the database of paper reviews. This policy expresses authorization facts such as $\mathit{PCMember}(\mathit{alice})$, which means "Alice has been appointed as a member of the program committee of the conference", or authorization rules such as

$$\mathit{Review}(U,\mathit{ID},R) :- \mathit{PCMember}(U),\mathit{Opinion}(U,\mathit{ID},R)$$

which means "if a committee member holds a certain opinion on any paper, she can submit a review for that paper". Capitalized variables such as $U$, $\mathit{ID}$, and $R$ are bound logical variables. Lower-case identifiers (such as *alice* above), together with any other values of the process language, are uninterpreted logical atoms.

*Process Syntax and Semantics.* The core language consists of an asynchronous spi calculus where parallel processes can send messages to each other on named channels. For example, we may write:

$$\mathsf{out}\ a(M) \mid \mathsf{in}\ a(x); P \rightarrow P\{M/x\}$$

The symbol $\rightarrow$ represents a computation step. On the left of $\rightarrow$, we have a parallel composition of a process that sends a message (actually $M$) on the channel $a$ and a process that receives a message (represented by the formal parameter $x$) on $a$ and then executes $P$; on the right is the result, in which the formal parameter is replaced with the actual message.

Messages include channel names, cryptographic keys, pairs, and encryptions. We assume that encryption preserves the integrity of the payload. There are operations for decomposing and matching pairs and for decrypting messages. For example,

$$\mathsf{decrypt}\ \{M\}k\ \mathsf{as}\ \{y\}k; Q \rightarrow Q\{M/y\}$$

represents the only way to "open" the encryption $\{M\}k$ to retrieve $M$.

Two special constructs have no effects on the semantics of programs, but are annotations that connect the authorization policy to the protocol code: statements and expectations. A *statement*, such as $\mathit{SentOn}(a,b)$, should be manually inserted in the code in order to record that, at a particular execution point, the clause $\mathit{SentOn}(a,b)$ is regarded as true. An *expectation*, such as $\mathsf{expect}\ \mathit{GoodParam}(x)$, should label program points where the clause $\mathit{GoodParam}(x)$ must hold for the run-time value of $x$. For example, the following code is safe with respect to the policy $\mathit{GoodParam}(X) :- \mathit{SentOn}(a,X)$:

$$(\mathsf{out}\ a(b) \mid \mathit{SentOn}(a,b)) \mid \mathsf{in}\ a(x); (\mathsf{expect}\ \mathit{GoodParam}(x) \mid \mathsf{out}\ c(x))$$

To this core language, we add a new kind of message $(x{:}T)P$ that represents the process $P$ parametrized by $x$ of type $T$, and operations to spawn such processes and to check the type of messages dynamically. The formal syntax of messages and processes is as follows:

**Syntax for Messages and Processes:**

| | |
|---|---|
| $a,b,c,k,x,y,z$ | name |
| $M,N ::=$ | message |
| $\quad x$ | $\quad$ name |
| $\quad \{M\}N$ | $\quad$ authenticated encryption of $M$ with key $N$ |
| $\quad (M,N)$ | $\quad$ message pair |
| $\quad (x{:}T)P$ | $\quad$ code $P$ parametric in $x$ |
| $\quad \mathsf{ok}$ | $\quad$ token conveying logical effects (see Section 3) |
| $P,Q,R ::=$ | process |
| $\quad \mathsf{out}\ M(N)$ | $\quad$ asynchronous output of $N$ to channel $M$ |
| $\quad \mathsf{in}\ M(x{:}T);P$ | $\quad$ input of $x$ from channel $M$ ($x$ has scope $P$) |
| $\quad !\mathsf{in}\ M(x{:}T);P$ | $\quad$ replicated input |
| $\quad \mathsf{new}\ x{:}T;P$ | $\quad$ fresh generation of name $x$ ($x$ has scope $P$) |
| $\quad P \mid Q$ | $\quad$ parallel composition of $P$ and $Q$ |
| $\quad \mathbf{0}$ | $\quad$ null process |
| $\quad \mathsf{decrypt}\ M\ \mathsf{as}\ \{y{:}T\}N;P$ | $\quad$ bind $y$ to decryption of $M$ with key $N$ ($y$ has scope $P$) |
| $\quad \mathsf{split}\ M\ \mathsf{as}\ (x{:}T,y{:}U);P$ | $\quad$ solve $(x,y) = M$ ($x$ has scope $U$ and $P$; $y$ has scope $P$) |
| $\quad \mathsf{match}\ M\ \mathsf{as}\ (N,y{:}U);P$ | $\quad$ solve $(N,y) = M$ ($y$ has scope $P$) |
| $\quad \mathsf{spawn}\ M\ \mathsf{with}\ N$ | $\quad$ spawn $M$ instantiated with $N$ |
| $\quad \mathsf{typecase}\ M\ \mathsf{of}\ x{:}T;P$ | $\quad$ typecheck $M$ at type $T$ ($x$ has scope $P$) |
| $\quad C$ | $\quad$ statement of clause $C$ |
| $\quad \mathsf{expect}\ C$ | $\quad$ expectation that clause $C$ is derivable |

Notations: $(\widetilde{x}{:}\widetilde{T}) \triangleq (x_1{:}T_1,\ldots,x_n{:}T_n)$ and $\mathsf{new}\ \widetilde{x}{:}\widetilde{T};P \triangleq \mathsf{new}\ x_1{:}T_1;\ldots\mathsf{new}\ x_n{:}T_n;P$
Let $S = \{C_1,\ldots,C_n\}$. We write $S \mid P$ for $C_1 \mid \ldots \mid C_n \mid P$.

For notational convenience, we may omit type annotations, especially for Un types.

Both spawn and typecase are standard constructs. However, in combination they turn out to be very useful for our purposes. For example, a verifier process can accept untrusted messages from the network, check that they are well-typed as processes with input of type $T$, and then send the code out to the network once again on an untrusted channel, wrapped in an encryption meant to signify that the contents are now guaranteed to be type-safe:

$$\mathsf{in}\ \mathit{unCode}(x);\mathsf{typecase}\ x\ \mathsf{of}\ y{:}\mathsf{Pr}(T);\mathsf{out}\ \mathit{tsCode}(\{y\}k)$$

A code user can accept such encrypted code packages, and run the code passing it a parameter $M$ of the correct type $T$ without further checking:

$$\mathsf{in}\ \mathit{tsCode}(x);\mathsf{decrypt}\ x\ \mathsf{as}\ \{y\}k;\mathsf{spawn}\ y\ \mathsf{with}\ M$$

As usual in the pi calculus, we define the formal semantics of the calculus by a set of structural congruence rules (see [16]) that describe what terms should be considered syntactically equivalent, and a set of reduction rules (displayed below) that describe how processes evolve. Most of these reduction axioms are standard. Rule (Red Typecase) requires some typing environment $E$ in which the check $E \vdash M : T$ can be performed. In order to define such environments, we parametrize the reduction relation by an initial environment (which can also be chosen as $\varnothing$ if necessary). Rule (Red Res) dynamically adds the names defined by restriction contexts to the current typing environment, and

**Rules for Reduction:** $P \rightarrow_E P'$

| | |
|---|---|
| $\mathsf{out}\ a(M) \mid \mathsf{in}\ a(x{:}T);P \rightarrow_E P\{M/x\}$ | (Red Comm) |
| $\mathsf{out}\ a(M) \mid !\mathsf{in}\ a(x{:}T);P \rightarrow_E P\{M/x\} \mid !\mathsf{in}\ a(x{:}T);P$ | (Red !Comm) |
| $\mathsf{decrypt}\ \{M\}k\ \mathsf{as}\ \{y{:}T\}k;P \rightarrow_E P\{M/y\}$ | (Red Decrypt) |
| $\mathsf{split}\ (M,N)\ \mathsf{as}\ (x{:}T,y{:}U);P \rightarrow_E P\{M,N/x,y\}$ | (Red Split) |
| $\mathsf{match}\ (M,N)\ \mathsf{as}\ (M,y{:}U);P \rightarrow_E P\{N/y\}$ | (Red Match) |
| $\mathsf{spawn}\ (x)P\ \mathsf{with}\ M \rightarrow_E P\{M/x\}$ | (Red Spawn) |
| $E \vdash M : T \Rightarrow \mathsf{typecase}\ M\ \mathsf{of}\ y{:}T;P \rightarrow_E P\{M/y\}$ | (Red Typecase) |
| $P \rightarrow_{E,env(Q)^{\widetilde{x}}} P' \Rightarrow P \mid Q \rightarrow_E P' \mid Q \quad$ (where $\{\widetilde{x}\} \cap fn(P,Q) = \varnothing$) | (Red Par) |
| $P \rightarrow_{E,x:T} P' \Rightarrow \mathsf{new}\ x{:}T;P \rightarrow_E \mathsf{new}\ x{:}T;P'$ | (Red Res) |
| $P \equiv Q, Q \rightarrow_E Q', Q' \equiv P' \Rightarrow P \rightarrow_E P'$ | (Red Struct) |

Notation: $P \rightarrow_E^{*\equiv} P'$ is $P \equiv P'$ or $P \rightarrow_E^{*} P'$.

rule (Red Par) adds the new clauses and names ($env(Q)^{\widetilde{x}}$) defined by parallel contexts. The technical reasons for these definitions, which should become apparent in Section 3, are illustrated in the following small example. Consider the reduction step:

$$\mathsf{new}\ a{:}T;(\mathsf{typecase}\ a\ \mathsf{of}\ y{:}T;P) \rightarrow_\varnothing \mathsf{new}\ a{:}T;P\{a/y\}$$

By (Red Res), this reduction takes place if $\mathsf{typecase}\ a\ \mathsf{of}\ y{:}T;P \rightarrow_{a:T} P\{a/y\}$, and this is a valid instance of (Red Typecase) since the typing environment is now $a{:}T$, and $a{:}T \vdash a : T$ is clearly a valid typing judgment.

These rules allow a typecase process $\mathsf{typecase}\ M\ \mathsf{of}\ y{:}T;P$ to reduce provided the message $M$ can be typechecked in an environment $E$ that collects clauses and names defined in any parallel context. In an implementation, it may be impractical to collect the full environment, because, for example, $E$ takes the form $E', E''$ where the clauses and names of $E'$ are local, while those in $E''$ are distributed across remote machines. Still, it is fine for an implementation to typecheck the message in the local environment $E'$, because, by a standard weakening lemma, if $E' \vdash M : T$ then also $E', E'' \vdash M : T$. Such an implementation would not admit reduction steps that depend on implicit knowledge of remote clauses and names. This is not a problem in our theory, as we are concerned with safety properties; in practice, we can convey knowledge of remote clauses and names by explicit use of cryptography, as in the examples in later sections.

For brevity, we use derived notations for tuples and pattern-matching, and omit type annotations when they are not necessary. The tuple $(M_1, M_2, \ldots, M_n)$ abbreviates the nested pairs $(M_1,(M_2, \ldots, M_n))$. We write $\mathsf{tuple}\ M\ \mathsf{as}\ (\underline{N}_1, \ldots, \underline{N}_n);P$ to pattern-match a tuple, where $M$ is a tuple, and each $\underline{N}_i$ is an atomic pattern (either a variable pattern $x$, or a constant pattern $=M$, where $M$ is a message to be matched). For each variable, we introduce a split, and for each constant a match. For example, for a fresh $z$ we have

$$\begin{aligned}&\mathsf{tuple}\ (a,b,c)\ \mathsf{as}\ (x,{=}b,y);P \triangleq \\ &\qquad \mathsf{split}\ (a,(b,c))\ \mathsf{as}\ (x,z);\mathsf{match}\ z\ \mathsf{as}\ (b,z);\mathsf{split}\ (z,z)\ \mathsf{as}\ (y,z);P\end{aligned}$$

We also allow pattern-matching in conjunction with input and decryption processes.

*Safety.* Relying on the operational semantics, we give a formal definition of safety (much as in [12]). This notion makes precise the intuitive relation between assumptions, expectations, and program execution. The idea is that a process is safe if whenever during an execution the statement $\mathsf{expect}\ C$ is reached (i.e., it appears at the top level, possibly inside some nested name restrictions) the environment has accumulated enough rules and facts to entail $C$.

It is also important to know when a process is safe even if it is executed in parallel with a malicious opponent. Following a common approach, we model the opponent as an arbitrary untyped process, with no statements or expectations.

**Safety, Opponents and Robust Safety:**

A process $P$ is *safe for $E$* if and only if whenever $P \rightarrow_E^{*\equiv} \mathsf{new}\ \tilde{x}{:}\tilde{T};(\mathsf{expect}\ C \mid P')$, we have $P' \equiv \mathsf{new}\ \tilde{y}{:}\tilde{U};(S \mid P'')$ and $S \cup \mathit{clauses}(E) \models C$ with $(\{\widetilde{y}\} \cap \mathit{fn}(C)) = \varnothing = (\{\widetilde{x},\widetilde{y}\} \cap \mathit{dom}(E))$.
A process $O$ is an *opponent* if and only if it contains no statement or expectation, and every type annotation is $\mathsf{Un}$.
A process $P$ is *robustly safe for $E$* if and only if for any opponent $O$, $P \mid O$ is safe for $E,\widetilde{x}{:}\widetilde{\mathsf{Un}}$, where $\widetilde{x}$ are the free names of $O$ not in the domain of $E$.

For example, the process $P = \mathsf{out}\ b(a) \mid \mathsf{in}\ b(x);\mathsf{expect}\ A(x)$ is safe for $A(a)$, but not robustly safe, as an opponent that replaces $a$ with $c$ can lead to an unsatisfied expectation: $\mathsf{in}\ b(x);\mathsf{out}\ b(c) \mid P \rightarrow_{A(a)}^{*} \mathsf{expect}\ A(c)$.

## 3 A Type System for Robust Safety

We present a dependent type system that statically guarantees safety and robust safety. We extend the system of [12] with a type constructor $\mathsf{Pr}(T)$ for process code parametric in $T$, and rules for the $\mathsf{spawn}$ and $\mathsf{typecase}$ constructs. Most of the rules in this section (including those for new constructs) are largely standard rules adapted to the present context. We are pleased by how much advantageous reuse has been possible.

We prove that typability with respect to an environment $E$ entails safety for $E$ and, if all the types in $E$ are $\mathsf{Un}$ ("untrusted"), also robust safety.

*Types and Environments.* Type $\mathsf{Un}$ is inhabited by any message that may come or go to the opponent, like for example a ciphertext that can be considered untrusted until it is decrypted. Upon decryption, one may reason that the contents were created by a principal that knows the encryption key. Types $\mathsf{Ch}(T)$ and $\mathsf{Key}(T)$ are inhabited by secure channels or secret keys for communicating or encrypting messages of type $T$. A dependent type $(x{:}T,U)$ is inhabited by the pairs $(M,N)$ where $M$ has type $T$, and $N$ has type $U\{M/x\}$. Type $\mathsf{Ok}(S)$ is inhabited only by the token $\mathsf{ok}$, and is used to attach effects to the payload of channels and keys. When a variable in the environment has type $\mathsf{Ok}(S)$, it is safe to assume that $S$ holds.

**Syntax for Types:**

$T,U ::= \mathsf{Un} \mid \mathsf{Ch}(T) \mid \mathsf{Key}(T) \mid (x{:}T,U) \mid \mathsf{Pr}(T) \mid \mathsf{Ok}(S)$

$T$ is *generative* iff $T$ is of the form $\mathsf{Un}$, $\mathsf{Ch}(U)$, or $\mathsf{Key}(U)$, for some $U$.
Notation: $(x_1{:}T_1,\ldots,x_n{:}T_n,T_{n+1}) \triangleq (x_1{:}T_1,\ldots,(x_n{:}T_n,T_{n+1}))$

For example, the type declaration $kra : \mathsf{Key}(id{:}\mathsf{Un}, r{:}\mathsf{Un}, \mathsf{Ok}(Opinion(alice, id, r)))$ says that $kra$ is a key for encrypting a tuple like $(paper, text, \mathsf{ok})$ where $paper$ and $text$ are untrusted values and the $\mathsf{ok}$ token indicates that the key conveys the logical effect $Opinion(alice, paper, text)$.

Typing environments are lists of name bindings and clauses. We write $dom(E)$ for the set of names defined (i.e., appearing to the left of a binding ":") in environment $E$. We write $env(P)$ for the top-level clauses of process $P$, with suitable name bindings for any top-level restrictions, and $clauses(E)$ for the clauses contained at the top level and inside the top-level $\mathsf{Ok}$ types of $E$. We use a standard notion $E \vdash \diamond$ of well-formedness for environments (see [16]).

**Syntax for Environments, and Functions:** $env(P)$, $clauses(E)$

$E ::= \varnothing \mid E, x{:}T \mid E, C$ Notation: $E(x) = T$ if $E = E', x{:}T, E''$

$clauses(\varnothing) = \varnothing \quad clauses(E, x{:}T) = clauses(E) \quad$ (if $T \neq \mathsf{Ok}(S)$)
$clauses(E, C) = clauses(E) \cup \{C\} \quad clauses(E, x{:}\mathsf{Ok}(S)) = clauses(E) \cup S$

$env(P \mid Q)^{\widetilde{x},\widetilde{y}} = env(P)^{\widetilde{x}}, env(Q)^{\widetilde{y}}$ (where $\{\widetilde{x}, \widetilde{y}\} \cap fn(P \mid Q) = \varnothing$)
$env(\mathsf{new}\ x{:}T; P)^{x,\widetilde{x}} = x{:}T, env(P)^{\widetilde{x}}$ (where $\{\widetilde{x}\} \cap fn(P) = \varnothing$)
$env(C)^{\varnothing} = C \qquad env(P)^{\varnothing} = \varnothing$ (otherwise)
Convention: $env(P) \triangleq env(P)^{\widetilde{x}}$ for some distinct $\widetilde{x}$ such that $env(P)^{\widetilde{x}}$ is defined.

*Typing Rules.* For each message constructor there are two typing rules, one to give it an informative type, and one to give it type $\mathsf{Un}$. Rules of the second kind are useful to show that any opponent process can be typed.

Rule (Msg Encrypt) shows that an encryption under a trusted key does not need to be trusted, in the sense that it can be sent to an opponent. Rules (Msg Proc) and (Msg Proc Un) invoke the typing relation for processes in an environment that assumes respectively type $T$ or type $\mathsf{Un}$ for the process parameter $x$. Rule (Msg Ok) is typical of this typed approach to verification: in order for an $\mathsf{ok}$ token to convey the effects $S$, it must be the case that the clauses contained in the environment (which include the policy and all the facts consequently accumulated by $\mathsf{Ok}$ types) entail each of the clauses in $S$.

**Rules for Messages:** $E \vdash M : T$

(Msg $x$)
$$\frac{E \vdash \diamond \quad x \in dom(E)}{E \vdash x : E(x)}$$

(Msg Encrypt)
$$\frac{E \vdash M : T \quad E \vdash N : \mathsf{Key}(T)}{E \vdash \{M\}N : \mathsf{Un}}$$

(Msg Encrypt Un)
$$\frac{E \vdash M : \mathsf{Un} \quad E \vdash N : \mathsf{Un}}{E \vdash \{M\}N : \mathsf{Un}}$$

(Msg Pair)
$$\frac{E \vdash M : T \quad E \vdash N : U\{M/x\}}{E \vdash (M,N) : (x{:}T, U)}$$

(Msg Pair Un)
$$\frac{E \vdash M : \mathsf{Un} \quad E \vdash N : \mathsf{Un}}{E \vdash (M,N) : \mathsf{Un}}$$

(Msg Ok Un)
$$\frac{E \vdash \diamond}{E \vdash \mathsf{ok} : \mathsf{Un}}$$

(Msg Proc)
$$\frac{E, x{:}T \vdash P}{E \vdash (x{:}T)P : \mathsf{Pr}(T)}$$

(Msg Proc Un)
$$\frac{E, x{:}\mathsf{Un} \vdash P}{E \vdash (x{:}\mathsf{Un})P : \mathsf{Un}}$$

(Msg Ok)
$$\frac{E, S \vdash \diamond \quad clauses(E) \models C \quad \forall C \in S}{E \vdash \mathsf{ok} : \mathsf{Ok}(S)}$$

Rule (Proc Res) requires to type $P$ in an environment with the additional binding $x{:}T$. Correspondingly, the reduction rule (Red Res) assumes the binding in the run-time environment of its premise. Rule (Proc Par) collects the effects of process $Q$ to typecheck $P$, and vice versa. Similarly, the premise of (Red Par) assumes $env(Q)$ in the environment of its premise. Rule (Proc Expect) requires an expected clause to be entailed by the environment, much in the same way as (Msg Ok). Rule (Proc Typecase) is somewhat subtle. It corresponds to an $\mathsf{Un}$ rule if we pick $U$ and $T$ to be $\mathsf{Un}$. Moreover, the type $U$ is not related *a priori* to the type $T$. In typical examples, the rule allows us to check a message $M$ received at type $\mathsf{Un}$ and bind a variable $y$ of some more useful type $T$ to this message if the check succeeds. The remaining rules come in pairs, with one rule that assumes informative types and one that assumes $\mathsf{Un}$ types. Most of them are straightforward. For example, (Proc Output) says that a message of type $T$ can be sent on a channel of type $\mathsf{Ch}(T)$, and (Proc Decrypt) says that the variable $y$ that represents the payload of a ciphertext of type $\mathsf{Un}$ decrypted with a key of type $\mathsf{Key}(T)$ can be assumed to have type $T$ in the continuation process. The rules for split and match are similar.

**Rules for Processes:** $E \vdash P$

(Proc Nil)
$$\frac{E \vdash \diamond}{E \vdash \mathbf{0}}$$

(Proc Res)
$$\frac{E, x{:}T \vdash P \quad T \text{ generative}}{E \vdash \mathsf{new}\ x{:}T; P}$$

(Proc Fact)
$$\frac{E, C \vdash \diamond}{E \vdash C}$$

(Proc Expect)
$$\frac{E, C \vdash \diamond \quad \mathit{clauses}(E) \models C}{E \vdash \mathsf{expect}\ C}$$

(Proc Par)
$$\frac{E, env(Q) \vdash P \qquad E, env(P) \vdash Q \qquad fn(P \mid Q) \subseteq dom(E)}{E \vdash P \mid Q}$$

(Proc Typecase)
$$\frac{E \vdash M : U \qquad E, x : T \vdash P}{E \vdash \mathsf{typecase}\ M\ \mathsf{of}\ x{:}T; P}$$

(Proc Spawn)
$$\frac{E \vdash M : \mathsf{Pr}(T) \quad E \vdash N : T}{E \vdash \mathsf{spawn}\ M\ \mathsf{with}\ N}$$

(Proc Spawn Un)
$$\frac{E \vdash M : \mathsf{Un} \quad E \vdash N : \mathsf{Un}}{E \vdash \mathsf{spawn}\ M\ \mathsf{with}\ N}$$

(Proc Input)
$$\frac{E \vdash M : \mathsf{Ch}(T) \qquad E, x{:}T \vdash P}{E \vdash [!]\mathsf{in}\ M(x{:}T); P}$$

(Proc Input Un)
$$\frac{E \vdash M : \mathsf{Un} \qquad E, x{:}\mathsf{Un} \vdash P}{E \vdash [!]\mathsf{in}\ M(x{:}\mathsf{Un}); P}$$

(Proc Output)
$$\frac{E \vdash M : \mathsf{Ch}(T) \quad E \vdash N : T}{E \vdash \mathsf{out}\ M(N)}$$

(Proc Output Un)
$$\frac{E \vdash M : \mathsf{Un} \quad E \vdash N : \mathsf{Un}}{E \vdash \mathsf{out}\ M(N)}$$

(Proc Decrypt)
$$\frac{E \vdash M : \mathsf{Un} \quad E \vdash N : \mathsf{Key}(T) \quad E, y{:}T \vdash P}{E \vdash \mathsf{decrypt}\ M\ \mathsf{as}\ \{y{:}T\}N; P}$$

(Proc Decrypt Un)
$$\frac{E \vdash M : \mathsf{Un} \quad E \vdash N : \mathsf{Un} \quad E, y{:}\mathsf{Un} \vdash P}{E \vdash \mathsf{decrypt}\ M\ \mathsf{as}\ \{y{:}\mathsf{Un}\}N; P}$$

Notation: brackets denote optional constructs.

As a simple example, we can show that for $E = Bar{:}-Foo, b{:}\mathsf{Ch}(\mathsf{Ok}(Bar))$, the typing judgment $E \vdash Foo \mid \mathsf{out}\ b(\mathsf{ok})$ is valid. The judgment follows by an instance of (Proc Par), from $E \vdash Foo$ and $E, Foo \vdash \mathsf{out}\ b(\mathsf{ok})$. The latter in turn follows by (Proc Output) and (Msg Ok), where the second rule uses the logical inference $\mathit{clauses}(E, Foo) \models Bar$. Section 4 includes a longer, detailed example of how the interplay between static and dynamic typechecking makes this type system expressive.

*Results.* We obtain a type preservation result and a safety theorem that guarantees that typability implies safety.

**Lemma 1 (Type Preservation).** *If $E \vdash P$ and $P \rightarrow_E^{*\equiv} P'$ then $E \vdash P'$.*

**Theorem 1 (Safety).** *If $E \vdash P$ then $P$ is safe for $E$.*

The safety theorem makes explicit the connection between the environment used for typing (existentially quantified in related work), and the run-time environment.

In order to show that our notion of opponent is not restrictive in a typed setting, we prove that any opponent can be typed in an environment that does not make trust assumptions. Finally, we prove that if a process $P$ is safe for a security policy $S$ and an untrusted environment, then it is robustly safe.

**Lemma 2 (Opponent Typability).** *For opponent $O$, $\widetilde{x}{:}\widetilde{\mathsf{Un}} \vdash O$, where $fn(O) \subseteq \{\widetilde{x}\}$.*

**Theorem 2 (Robust Safety).** *If $\widetilde{x}{:}\widetilde{\mathsf{Un}}, S \vdash P$ then $P$ is robustly safe for $\widetilde{x}{:}\widetilde{\mathsf{Un}}, S$.*

For example, let us consider process $Q = \mathsf{out}\ b(a,\mathsf{ok}) \mid \mathsf{in}\ b(x,y); \mathsf{expect}\ A(x)$. It is easy to see that given $E = a{:}\mathsf{Un}, b{:}\mathsf{Ch}(x{:}\mathsf{Un}, A(x)), A(a)$ we have $E \vdash Q$, so $Q$ is safe for $E$. On the other hand it is not possible to derive $a{:}\mathsf{Un}, b{:}\mathsf{Un}, A(a) \vdash Q$, so we cannot prove robust safety (which does not hold).

*Dynamic Verification.* We define a derived construct to verify that a piece of code $M$, when passed a parameter $N$ of type $T$ enforces property $S$. The idea is to typecheck dynamically $M$, against the parameter type $T$ and an implicit parameter $c$ that is a channel used to return the result of verification, namely an $\mathsf{ok}$ token carrying the effects $S$. The continuation process $P$ will execute only if verification succeeds, that is $M$ sends an $\mathsf{ok}$ on channel $c$.

$$\begin{array}{lll}\mathsf{verify}\ M\langle[\widetilde{z{:}\mathsf{Un}}, N{:}T]\rangle{:}S;P & \triangleq & \mathsf{new}\ c{:}\mathsf{Ch}(\mathsf{Ok}(S)); \\ & & \big(\mathsf{typecase}\ M\ \mathsf{of}\ y{:}\mathsf{Pr}([\widetilde{z{:}\mathsf{Un}}, T,]\mathsf{Ch}(\mathsf{Ok}(S))); \\ & & \mathsf{spawn}\ y\ \mathsf{with}\ ([\widetilde{z}, N,]c) \mid \mathsf{in}\ c(x{:}\mathsf{Ok}(S)); P\big) \\ & & (\text{where } \{c,y,x\} \cap fn(P,M,[N,]S) = \varnothing, \text{ and } \{\widetilde{z}\} \subseteq fn(S))\end{array}$$

One may wonder whether it is prudent to run the code of an untrusted verifier that is guaranteed to enforce a certain policy. Although additional precautions may be appropriate, this guarantee is substantial. By lexical scoping, the code of the verifier cannot contain capabilities that are not already known by its generator; other capabilities can only be passed explicitly as parameters. Moreover, the verifier must be well-typed in the run-time typing environment, which can be restricted conveniently to further limit potential side effects. On the other hand, this guarantee does not cover other kinds of attacks (such as information leaks or denial-of-service attacks), which may be addressed independently.

## 4 Examples: a Conference Program Committee

As a benchmark for the effectiveness of CCA, we revisit the conference program committee example of [12]. We first review the idealized electronic conference system, then present two examples that illustrate the benefits of CCA.

*Review: an Electronic Conference Reviewing System.* There are three kinds of principals: the program committee chair (pc-chair), identified with the server, the program committee members (pc-members), and potential reviewers. The last two are clients of the server. We model only the portion of the conference reviewing system for delegating and filing reviews. The authorization policy $S$, from the subjective viewpoint of the pc-chair, is:

$$
\begin{aligned}
S = \mathit{Review}(U,ID,R) &:- \mathit{Reviewer}(U,ID), \mathit{Opinion}(U,ID,R)\\
\mathit{Review}(U,ID,R) &:- \mathit{PCMember}(U), \mathit{Opinion}(U,ID,R)\\
\mathit{Reviewer}(V,ID) &:- \mathit{Reviewer}(U,ID), \mathit{Delegate}(U,V,ID)\\
\mathit{Delegate}(U,W,ID) &:- \mathit{Delegate}(U,V,ID), \mathit{Delegate}(V,W,ID)\\
\mathit{Delegate}(U,U,ID) &:- \mathit{Opinion}(U,ID,R)
\end{aligned}
$$

The predicate $\mathit{Opinion}(u,id,r)$ states that principal $u$ holds opinion $r$ on paper $id$, and is under the control of $u$ itself (that is, the code identified with $u$ can freely assert that predicate). The predicate $\mathit{Delegate}(u,v,id)$ states that principal $u$ delegates its capability to review paper $id$ to principal $v$, and is also under the control of $u$. All the other predicates are controlled by the pc-chair, and should be asserted only within server code.

Cryptographic keys can be associated with each of these predicates to convey authorization facts through untrusted messages. Thus, the pc-chair may appoint *alice* as a pc-member by sending her a token $\{alice\}kp$ encrypted under a key that carries the effect $\mathit{PCMember}(alice)$, and similarly for the other predicates. We define the type of the keys that correspond to each effect, and the type of a channel that implements a database where the pc-chair stores the keys of all potential users:

$$
\begin{aligned}
KA &= \mathsf{Key}(u{:}\mathsf{Un}, id{:}\mathsf{Un}, \mathsf{Ok}(\mathit{Reviewer}(u,id)))\\
KP &= \mathsf{Key}(u{:}\mathsf{Un}, \mathsf{Ok}(\mathit{PCMember}(u)))\\
KD &= \mathsf{Key}(z{:}\mathsf{Un}, id{:}\mathsf{Un}, \mathsf{Ok}(\mathit{Delegate}(v,z,id)))\\
KR &= \mathsf{Key}(id{:}\mathsf{Un}, r{:}\mathsf{Un}, \mathsf{Ok}(\mathit{Opinion}(v,id,r)))\\
T &= \mathsf{Ch}(v{:}\mathsf{Un}, (KD,KR))
\end{aligned}
$$

Keys of type $KA$ or $KP$ are used by the pc-chair only, to assign a paper to a reviewer or to appoint a pc-member respectively. Keys of type $KD$ or $KR$ (parametric in $v$) can be used by principal $v$ to convey either an opinion or a delegation effect. Type $T$ is the type of a channel used to retrieve the keys of each registered user. Note that it is a dependent type that binds the free parameter $v$ of types $KD$ and $KR$.

*Off-line Delegation.* Our first example presents a system that lets reviewers appoint subreviewers without involving the pc-chair in the process. A typical solution that does not use CCA is to have a reviewer present to the server a request that contains her opinion, together with some evidence that represents a chain of delegation. The server then runs an algorithm to traverse the chain and check corresponding permissions, and grants access if the evidence is satisfactory. This solution commits the server to a specific verification algorithm (or a fixed number thereof). Using CCA instead, the server code can be simpler and parametric. For example, the server is defined by the same code whether or not the delegation chain is ordered, has limited length, or delegation is permitted at all. Along with each request to file a review, the server receives the code of a verifier and some evidence. It verifies that the code enforces the desired authorization policy, and grants access without further checks. The relevant portion of the server code is:

$$
\begin{aligned}
&\mathit{Server}(pwdb{:}T, ka{:}KA, kp{:}KP) =\\
&S \mid !\mathsf{in}\ \mathit{filereview}(v,id,r,p,e);\\
&\quad \mathsf{verify}\ p\langle (v,r,e,(pwdb,ka,kp)) {:} (v{:}\mathsf{Un}, r{:}\mathsf{Un}, \mathsf{Un}, (T,KA,KP)) \rangle {:} \mathit{Review}(v,id,r);\ [...]
\end{aligned}
$$

It contains the assertion of policy $S$, and a process always ready to accept messages on the public channel *filereview*. Parameters $v$, $id$, and $r$ are interpreted as a request from principal $v$ to file review $r$ on paper $id$. Parameter $p$ is the code of a verifier that must be run to grant authorization (i.e., prove *Review*(*v*,*id*,*r*)) on data including the evidence received as the last parameter *e*, and local credentials provided by the server. The parameters passed by the server to the verifier *p* are the name *v* of the principal issuing the request, the report *r*, the evidence *e*, and a triple $(pwdb, ka, kp)$. Channel *pwdb* can be used to retrieve user credentials. Keys *ka* and *kp* are the secret keys used by the pc-chair to appoint reviewers and pc-members. If verification succeeds, authorization is granted, and *r* is a valid review for *id*.

A delegate *v* receives from a reviewer a request to review paper *id*, with additional parameters *p* (the verifier code to be passed on to the server), and *dc* (the evidence that represents a chain of delegation). The delegate may appoint another sub-reviewer, adding a delegation step to the chain $(v, \{u, id, ok\}kdv, dc)$, or file a review, adding evidence of its opinion to the top of the chain:

```
Delegate(v:Un,krv:KR,kdv:KD) =
!in reviewrequest(=v,id,p,dc);
(in accept(r); Opinion(v,id,r) | out filereview(v,id,r,p,({id,r,ok}krv,dc)) |
(in delegate(u); Delegate(v,u,id) | out reviewrequest(u,id,p,(v,{u,id,ok}kdv,dc)))
```

The pc-member can embed its logical effects directly in the verification code. For that reason, it transmits as evidence ok tokens with empty logical effects. The verifier *fver*, used to file a review ignores the principal name and the evidence, states that *v* holds opinion *r* on *id*, parses the server credentials to get the key to appoint pc-members, proves that *v* is a pc-member, by decrypting the appointment token (passed by the server earlier on), and finally signals success.

```
PCMember(v:Un,pctoken:Un,idtoken:Un) =
!in paperassign(=v,id,idtoken);
(in review(r); out filereview(v,id,r,fver,ok) |
(in delegate(u); out reviewrequest(u,id,dver,ok))
fver = (_,_,keys,return) (Opinion(v,id,r) | tuple keys as (_,_,kp);
        decrypt pctoken as {=v,_}kp; out return(ok))
```

The verifier code *dver* involves a loop to gather and verify all the elements of the delegation chain. Because of space constraints, we relegate it to the full version [16].

This code, and a few additional code fragments not shown here, can be assembled into a program that represents the entire conference reviewing system. This program typechecks in an environment of the form $\widetilde{x}:\widetilde{\mathsf{Un}}$ (according to the rules of Section 3). Therefore, Theorem 2 applies, and guarantees robust safety. In this particular case, this theorem implies that expectations in the server code, such as *Review*(*v*,*id*,*r*), are always satisfied at run-time when they occur, even in an untrusted environment.

*Server-Side Proxy.* Our second example illustrates the use of verifiers as server-side proxies installed by clients. It illustrates the flexibility of using typecase and spawn independently from the derived verify construct.

We modify our previous example so that the pc-member sends the delegation verifier *dver* directly to the server, which can use it to authorize requests from delegated reviewers. We show the code for dealing with delegated reviews, which is the most interesting. The server registers proxies for each pc-member, and accepts requests on each proxy. A message on the public channel *newproxy* causes the server to typecheck the code *dver* and install it as a handler and verifier for requests coming from reviewers delegated by pc-member *u*:

```
Server(pwdb:T,ka:KA,kp:KP) =
S | new protectedfilereview:V;
    (!in newproxy(dver); typecase dver is y:Pr(U);
        spawn y with ((pwdb,ka,kp),protectedfilereview)
    |!in protectedfilereview(v,id,r,_); expect Review(v,id,r); [...])
U = ((T,KA,KP),V)
V = Ch(v:Un,id:Un,r:Un,Ok(Review(v,id,r)))
```

Once appointed, a pc-member installs its delegation proxy on the server. The proxy receives requests from delegates on a dedicated channel and authorizes them. Upon delegation, the pc-member needs to send to the delegate a request that contains the name of the dedicated channel and evidence of delegation. The evidence consists of a delegation chain that contains a delegation step $\{u,id,\mathsf{ok}\}kdv$ (the name of the delegate and the paper id encrypted under the delegation key of the pc-member, and an ok token) and the list terminator (another ok token):

```
PCMember(v:Un,pctoken:Un) =
!in paperassign(=v,id,idtoken);
new filesubreview:Un;
    out newproxy(dver) |
    (in delegate(u); out reviewrequest(u,id,filesubreview,({u,id,ok}kdv,ok)))
```

The verifier *dver* now installs a process ready to listen to delegate requests on channel *filesubreview*, and then verifies requests similarly to the code shown above for off-line delegation. The main differences are that, in this case, the result returned by the verification process needs to contain the parameters $v, id, r$ of the effect $Review(v,id,r)$ to be enforced, and the code (given in the full version [16]) does not contain the implicit delegation effect $Delegate(v,u,id)$.

The code for the delegate is little changed. It files reviews on the dedicated channels, or delegates further:

```
Delegate(v:Un,krv:KR,kdv:KD) =
!in reviewrequest(=v,id,filereview,dc);
(in accept(r); Opinion(v,id,r) | out filereview(v,id,r,({id,r,ok}krv,dc)) |
(in delegate(u); Delegate(v,u,id) | out reviewrequest(u,id,filereview,(v,{u,id,ok}kdv,dc)))
```

*Best-Effort Evidence.* Our third example presents a system that supports the possibility for reviewers to appoint sub-reviewers, without needing immediate access to their delegation credentials. In a completely static type system, a typical delegation protocol such as the one presented in the previous section needs to record in a delegation chain the causal relation between delegation steps. Hence, a reviewer that momentarily does not have access to its delegation key cannot appoint a sub-reviewer.

We present a protocol that is well-typed, hence guarantees that, each time authorization to file a review is granted, the requesting principal is provably a reviewer. Yet, the protocol is “best-effort”, in that authorization can be denied at run-time if the server has not yet received all the delegation messages necessary to reconstruct a valid delegation chain.

To simplify the presentation, and to illustrate another advantage of CCA, we present code that does not use cryptography. Suppose that the machine of the reviewer is down, so she picks up the phone and asks a sub-reviewer to review a paper and to send his opinion (in the form of a simple verifier) to the server, trusting that the review will be accepted. The sub-reviewer can do so, or delegate further by issuing another informal request and by separately contacting the server to communicate his delegation decision:

```
Delegate(v:Un) =
!in phonereviewrequest(=v,id);
  (in accept(r); out filereview(v,id,r,fver))
|(in delegate(u); out phonereviewrequest(u,id) | out latedelegation(v,u,id,dver))
fver = (return)(Opinion(v,id,r)|out return(ok))
dver = (return)(Delegate(v,u,id)|out return(ok))
```

The server independently accepts requests for filing reviews and messages that state delegation decisions. In the first case, the server simply verifies that the review can be filed; in the second case it verifies that it is safe to assert a delegation step. At run-time the server authorizes the request to file a review from a delegate only if it has already verified enough delegation evidence to form a chain that originates from an appointed reviewer:

```
Server() =
S | PCMember(alice) | Reviewer(bob,42)
   | (!in filedreview(v,id,r,fver); verify fver⟨⟩:Review(v,id,r); [...])
   | (!in latedelegation(v,u,id,dver); verify dver⟨⟩:Delegate(v,u,id);Delegate(v,u,id))
```

In previous static systems, this sort of best-effort code was not possible. The code had to be written so that the expectation *Review*($v$,*id*,$r$) could occur only after code that would check the necessary delegation facts.

## 5 From Theorem Proving to Proof Checking

We have shown how to pass and dynamically check the code of a verifier process. The dynamic check may involve invoking a theorem prover, potentially a costly operation. On the other hand, passing proofs only requires the receiving side to have a proof checker, reducing both the trusted computing base and the performance cost of verification. For this reason, we extend our framework with the capability to pass also *hints*, that can help the receiver of a reference monitor with the logical proofs involved in dynamic typechecking. Hints could be proofs, in the formal sense of the word, or any other kind of information which may (or may not) be helpful. In particular, hints could be incomplete proofs, that simplify rather than eliminate theorem proving.

*From* ok*s to Hints.* The ok token can already be interpreted as an empty hint, that leaves to the typechecker the burden of finding a proof. We parametrize ok tokens by a generic language of (possibly empty) proof hints $H$. Hints may contain variables, so that they can be combined at run-time to form larger hints. Expectations now mention a term that can be used as a hint to prove $C$.

**Syntax for Hints**

| | |
|---|---|
| $M,N ::= \mathsf{ok}\,H \mid \dots$ | proof hint $H$ replaces $\mathsf{ok}$ |
| $P,Q,R ::= \mathsf{expect}\ C\ \mathsf{by}\ M \mid \dots$ | expectation that clause $C$ is derivable by $M$ replaces $\mathsf{expect}\ C$ |

The notion of type-safety does not change (just replace $\mathsf{expect}\ C$ by $\mathsf{expect}\ C\ \mathsf{by}\ M$), since the final result that we desire is still that any expectation is justified by logical entailment. It is the verification process that can be made simpler by adopting a verification relation, which naturally should imply entailment.

**Verification Relation:** $\mathscr{V}(M,C,S)$

Given an authorization logic $(\mathscr{C}, fn, \models)$, we assume an abstract verification predicate $\mathscr{V}$ that holds only if a message $M$ is a proof of clause $C$ starting from policy $S$, and such that $\mathscr{V}(M,C,S) \Rightarrow S \models C$.

We use hints and the verification relation in the typing rules that involve logical effects. In particular, we only need to replace (Msg Ok), (Msg Ok Un), and (Proc Expect) by the corresponding typing rules given below.

**Typing Rules for Hints**

(Msg Hint)

$$\frac{E,S \vdash \diamond \quad fn(H) \subseteq dom(E) \quad \mathscr{V}(H,C,clauses(E)) \quad \forall C \in S}{E \vdash \mathsf{ok}\,H : \mathsf{Ok}(S)}$$

(Msg Hint Un)

$$\frac{E \vdash \diamond \quad fn(H) \subseteq dom(E)}{E \vdash \mathsf{ok}\,H : \mathsf{Un}}$$

(Proc Expect Hint)

$$\frac{E,C \vdash \diamond \quad E \vdash M : \mathsf{Ok}(S) \quad C \in S}{E \vdash \mathsf{expect}\ C\ \mathsf{by}\ M}$$

The rules for hints are the obvious adaptations of the corresponding rules for $\mathsf{ok}$. Note that verification can assume as lemmas the effects of hints that are just variables, because they are included by $clauses(E)$ in the premise of (Msg Hint). Rule (Proc Expect Hint) no longer involves verification directly. It is the premise needed to give $M$ the $\mathsf{Ok}(S)$ type that may involve proof-checking.

This type system conservatively extends the one without hints. In fact, the type system presented in Section 3 correspond exactly to the instance of the current type system where $H$ is empty, each expectation is of the form $\mathsf{expect}\ C\ \mathsf{by}\ \mathsf{ok}$, and $\mathscr{V}(M,C,S)$ is defined as $S \models C$.

**Theorem 3 (Safety with Hints).** *(i) If $E \vdash P$ then $P$ is safe for $E$. (ii) If $\widetilde{x}{:}\widetilde{\mathsf{Un}}, S \vdash P$ then $P$ is robustly safe for $\widetilde{x}{:}\widetilde{\mathsf{Un}}, S$.*

The syntactic sugar from Section 4 can be adapted easily to hints by making explicit the variable $x$ that is bound to the hint that results from the verification process, so that it can be used in subsequent expectations, or to build more complex hints.

*Verification in Datalog.* For the examples, we use the simple hint language and logical verification relation for Datalog defined below, where $S \models_1 C$ is the single-step entailment relation.

For example, considering $S = D{:}-C, C{:}-B, B{:}-A, A$ and $S_1 = D{:}-C, C{:}-B, B$ and $S_2 = C, D{:}-C$, we have that $\mathscr{V}(\mathsf{ok}\,(S_1,S_2),D,S)$ follows by an instance of (Verify Pair) with premises $\mathscr{V}(\mathsf{ok}\,S_1,C,S)$, $\mathscr{V}(\mathsf{ok}\,S_1,D{:}-C,S)$, and $\mathscr{V}(\mathsf{ok}\,S_2,D,S_1)$.

**Hints and Verification**

$H ::= S \mid M$ proof hint: clauses $S$ or message $M$

(Verify S)

$$\frac{S \models_1 C' \quad \forall C' \in S' \qquad S' \models_1 C}{\mathscr{V}(\mathsf{ok}\, S', C, S)}$$

(Verify Pair)

$$\frac{\mathscr{V}(\mathsf{ok}\, M_1, C', S) \quad \forall C' \in \overline{M_2} \qquad \mathscr{V}(\mathsf{ok}\, M_2, C, \overline{M_1})}{\mathscr{V}(\mathsf{ok}\,(M_1, M_2), C, S)}$$

$\overline{\mathsf{ok}\, S} = S \qquad \overline{(M_1, M_2)} = \overline{M_1} \cup \overline{M_2} \qquad \overline{M} = \varnothing$ otherwise

*Example: Best-Effort Evidence Revisited.* We revisit the example of Section 4. In the system without automatic theorem prover, it is not enough to perform the operational checks that grant authorization. It is also necessary to provide the logical engine with hints on how to derive the right authorization facts.

For example, a reviewer $v$ for paper $id$ that decides to appoint a sub-reviewer $u$, needs to tell the server how to derive from the policy the fact $Reviewer(u,id)$, based on the facts that may be available by the time the request is submitted. In particular, the hint $H$ in the verifier code $dver$ contains the facts $Delegate(v,u,id)$, stated by $v$ itself, $Reviewer(v,id)$ which $v$ cannot state, but that it can assume to be asserted by the time the delegation request is filed, and the rule needed to conclude $Reviewer(u,id)$. The (simpler) case for filing reviews is given in the full version [16].

$H = Reviewer(U,ID) :- Reviewer(V,ID), Delegate(V,U,ID);\ Reviewer(v,id); Delegate(v,u,id)$
$dver = (return)(Delegate(v,u,id) \mid \mathsf{out}\ return(\mathsf{ok}(H))$

The server code needs to change the effects obtained by verifying a delegation request, essentially stating a lemma useful to prove further authorization.

$S \mid PCMember(alice) \mid Reviewer(bob,id) \mid \ldots$
$\quad \mid (!\mathsf{in}\ latedelegation(v,u,id,dver);\ \mathsf{verify}\ dver\langle\rangle{:}Reviewer(u,id); Reviewer(u,id))$

## 6 Conclusions

In this paper, we introduce "Code-Carrying Authorization" as a discipline for passing fragments of a reference monitor rather than proofs in order to perform run-time authorization. These fragments are themselves checked dynamically, since in general they are not trusted. We present a typing discipline that statically enforces safety with respect to authorization logics, and explore the notion of passing (proof) hints as a way to alleviate the dynamic verification process. The recent literature contains other type systems for authorization policies. While we base our work on that of Fournet et al. [12], because of its simplicity, the ideas that we explore should carry over to more elaborate languages. In particular, these variants would address the problem of partial trust [13]. They may also enable us to instantiate CCA in a general-purpose programming language such as F# [6] (a dialect of ML). Going beyond the present exploration (in which we emphasize concepts and theory over practice), such extensions are important for the further study of CCA and its applications.

**Acknowledgments.** We thank Gordon Plotkin for useful comments and suggestions. Sergio Maffeis is supported by EPSRC grant EP/E044956/1. This work was done while Maffeis was visiting Microsoft Research, Silicon Valley, whose hospitality is gratefully acknowledged.

## References

1. M. Abadi. Access control in a core calculus of dependency. In *Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin*, pages 5–31. Elsevier, 2007. Volume 172 of ENTCS.
2. M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *POPL'89: Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 213–227. ACM, 1989.
3. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Inf. and Comp.*, 148:1–70, 1999.
4. A. W. Appel and E. W. Felten. Proof-carrying authentication. In *CCS'99: Proceedings of the 6th ACM Conference on Computer and Communications Security*, pages 52–62, 1999.
5. L. Bauer, M. A. Schneider, and E. W. Felten. A general and flexible access-control system for the Web. In *Proceedings of the 11th USENIX Security Symposium*, pages 93–108, 2002.
6. J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 17–32. IEEE, June 2008.
7. S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Trans. Knowl. Data Eng.*, 1(1):146–166, 1989.
8. B.-Y. E. Chang, A. J. Chlipala, G. C. Necula, and R. R. Schneck. The Open Verifier framework for foundational verifiers. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2005)*, pages 1–12. ACM, 2005.
9. A. Cirillo, R. Jagadeesan, C. Pitcher, and J. Riely. Do As I SaY! Programmatic access control with explicit identities. In *CSF'07: 20th IEEE Computer Security Foundation Symposium*, pages 16–30. IEEE, 2007.
10. A. Cirillo and J. Riely. Access control based on code identity for open distributed systems. In *TGC'07: Trustworthy Global Computing*, volume 4912 of *Lecture Notes in Computer Science*, pages 169–185. Springer, 2007.
11. J. DeTreville. Binder, a logic-based security language. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 105–113. IEEE, 2002.
12. C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies. *ACM Trans. Program. Lang. Syst.*, 29(5):25, 2007.
13. C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies in distributed systems. In *CSF'07: 20th IEEE Computer Security Foundation Symposium*, pages 31–45. IEEE, 2007.
14. M. Hennessy, J. Rathke, and N. Yoshida. safeDpi: a language for controlling mobile code. *Acta Inf.*, 42(4-5):227–290, 2005.
15. C. Lesniewski-Laas, B. Ford, J. Strauss, R. Morris, and M. F. Kaashoek. Alpaca: extensible authorization for distributed services. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 432–444. ACM, 2007.
16. S. Maffeis, M. Abadi, C. Fournet, and A. D. Gordon. Code-carrying authorization. Long version: `http://www.doc.ic.ac.uk/~maffeis/cca.pdf`, 2008.
17. G. C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119. ACM, 1997.
18. J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. *J. Autom. Reas.*, 31(3-4):335–370, 2003.
19. D. Sangiorgi. From pi-calculus to higher-order pi-calculus - and back. In *TAPSOFT'93: Theory and Practice of Software Development*, pages 151–166, 1993.
20. J. A. Vaughan, L. Jia, K. Mazurak, , and S. Zdancewic. Evidence-based audit. In *21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 163–176. IEEE, June 2008.