A Certified Implementation of a Distributed Security Logic

Nathan Whitehead
University of California, Santa Cruz
nwhitehe@cs.ucsc.edu

based on joint work with

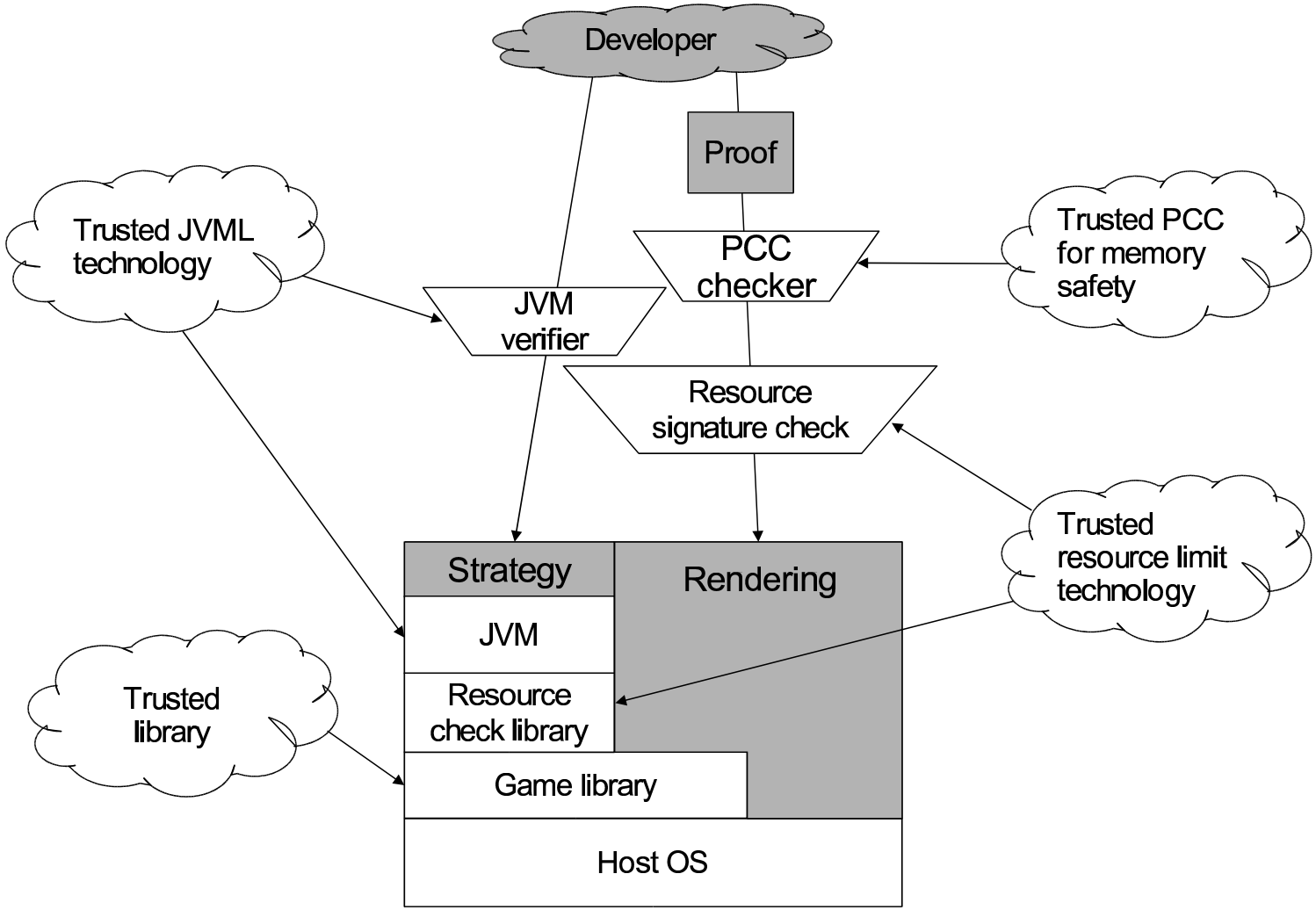
Martín Abadi
University of California, Santa Cruz
abadi@cs.ucsc.edu

George Necula
University of California, Berkeley
necula@cs.berkeley.edu

Access Control

- Modern access control systems must work in a *distributed* manner.
- They need to decide *when untrusted code is safe to execute*.
- It is possible to combine Binder and the calculus of constructions in a general purpose distributed security logic.
- This can express policies relying on
  trust through assertions from authorities
  and *trust through checking safety proofs*.

Game Cell Phone Example

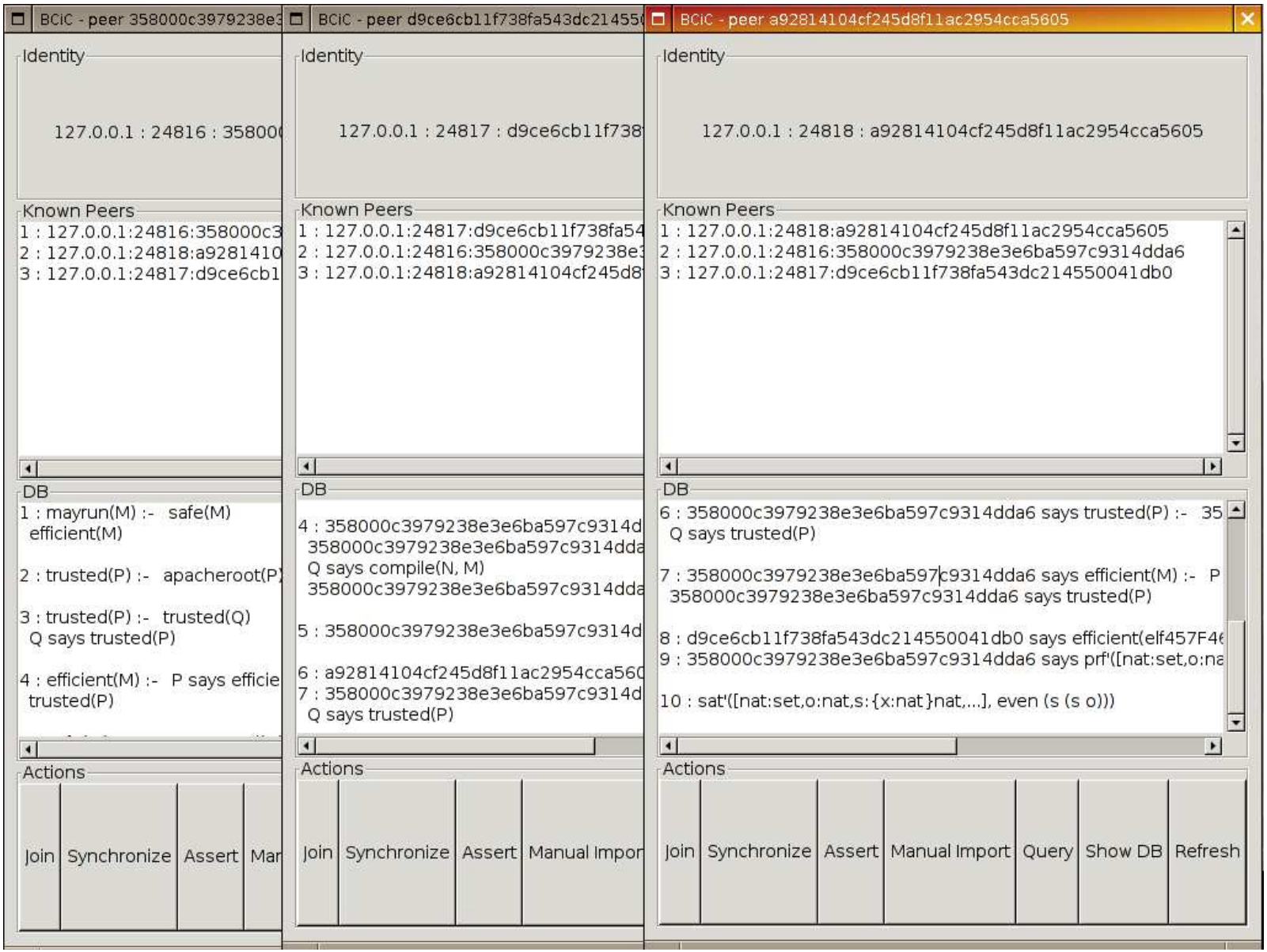


Example Policy Excerpt

```
use r_TAL in
  forall P:prg
    mayrun(P) :- believe(safe P),
                 believe(economical P).
end

use r_TAL in
  forall L:prg
    believe(safe L) :-
      lib_signer says trusted(L).
    believe(economical L) :-
      lib_signer says trusted(L).
end
```

BCC

- BCC combines *Binder* and the *calculus of constructions*.
- Equivalently, start with Datalog as the base for an access control system, then add in features as necessary.
- We add to Datalog the `says` operator, function symbols, and special predicates `sat` and `believe` connected to the calculus of constructions.
- `says` allows distributed reasoning.
- `sat(P)` means `P` is true by some proof, `believe(P)` means `P` is believed to be true.


BCiC - peer 358000c3979238e3
Identity
127.0.0.1 : 24816 : 358000
Known Peers
1 : 127.0.0.1:24816:358000c3
2 : 127.0.0.1:24818:a9281410
3 : 127.0.0.1:24817:d9ce6cb1
DB
1 : mayrun(M) :- safe(M)
efficient(M)
2 : trusted(P) :- apacheroot(P)
3 : trusted(P) :- trusted(Q)
Q says trusted(P)
4 : efficient(M) :- P says efficie
trusted(P)
Actions
Join
Synchronize
Assert
Mar
BCiC - peer d9ce6cb11f738fa543dc21455
Identity
127.0.0.1 : 24817 : d9ce6cb11f738
Known Peers
1 : 127.0.0.1:24817:d9ce6cb11f738fa54
2 : 127.0.0.1:24816:358000c3979238e3
3 : 127.0.0.1:24818:a92814104cf245d8
DB
4 : 358000c3979238e3e6ba597c9314d
358000c3979238e3e6ba597c9314dda
Q says compile(N, M)
358000c3979238e3e6ba597c9314dda
5 : 358000c3979238e3e6ba597c9314d
6 : a92814104cf245d8f11ac2954cca560
7 : 358000c3979238e3e6ba597c9314d
Q says trusted(P)
Actions
Join
Synchronize
Assert
Manual Impor
BCiC - peer a92814104cf245d8f11ac2954cca5605
Identity
127.0.0.1 : 24818 : a92814104cf245d8f11ac2954cca5605
Known Peers
1 : 127.0.0.1:24818:a92814104cf245d8f11ac2954cca5605
2 : 127.0.0.1:24816:358000c3979238e3e6ba597c9314dda6
3 : 127.0.0.1:24817:d9ce6cb11f738fa543dc214550041db0
DB
6 : 358000c3979238e3e6ba597c9314dda6 says trusted(P) :- 35
Q says trusted(P)
7 : 358000c3979238e3e6ba597c9314dda6 says efficient(M) :- P
358000c3979238e3e6ba597c9314dda6 says trusted(P)
8 : d9ce6cb11f738fa543dc214550041db0 says efficient(elf457F4
9 : 358000c3979238e3e6ba597c9314dda6 says prf'([nat:set,o:na
10 : sat'([nat:set,o:nat,s:{x:nat}nat,...], even (s (s o)))
Actions
Join
Synchronize
Assert
Manual Import
Query
Show DB
Refresh

Reference Monitor Requirements (Anderson)

- Must *always be invoked* for every access control decision and cannot be bypassed.
- Must be *tamper-proof*.
- Must be "small enough to be subject to analysis and tests, the completeness of which can be assured" (i.e. *correct*).

Since reference monitors are critical to security, it is a good place to focus our energy on proving correctness.

Motivation

- Combining different logics could lead to subtle *inconsistencies*.
- Our ad hoc implementation surely contains bugs.
- We turn to Coq in order to *express our logic formally* and *prove theorems about it* to gain assurance that it works.
- We then extract a *certified implementation* for the logic.

Related Work

- There are many existing formal models of access control.
- Some previous reference monitors have been certified by hand.
- DHARMA is a certified implementation of a *distributed delegation logic* encoded into PVS and extracted into Lisp.

  DHARMA is based on *access control lists* and *delegation*, while BCC is based on *predicate logic* and *proof checking*.
- *Proof-carrying authentication* has been done for other undecidable security logics.

Contributions

We encode a series of logics leading up to BCC.

Datalog — encoding, proof of decidability, decision procedure

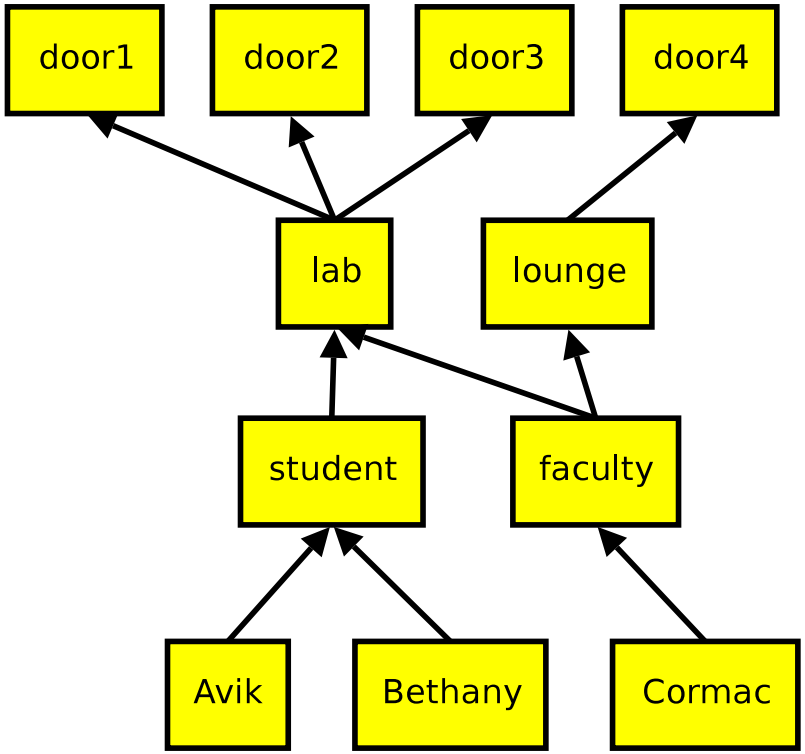
Binder — encoding, proof of decidability, decision procedure

Horn logic — encoding, sound proof checker, incomplete prover

BCC — encoding, sound proof checker, incomplete prover

Datalog for Access Control

```
student(avik).
student(bethany).
faculty(cormac).
lab(X) :- student(X).
lab(X) :- faculty(X).
lounge(X) :- faculty(X).
mayopen(X, door1) :- lab(X).
mayopen(X, door2) :- lab(X).
mayopen(X, door3) :- lab(X).
mayopen(X, door4) :- lounge(X).
```



Encoding Datalog in Coq

```
Inductive term : Set :=
| ident : nat -> term
| var : nat -> term.
Inductive atomic : Set :=
| atom : nat -> list term -> atomic.
Inductive form : Set :=
| clause : atomic -> list atomic -> form.

Inductive derive : list form -> atomic -> Prop :=
| derive_step :
    forall (LF : list form)(F : form)(S : substitution),
      In F LF ->
      forallelts atomic
        (fun x => derive LF (subs_atomic S x))
        (body F) ->
      derive LF (subs_atomic S (head F)).
```

Proving Decidability

- Decision procedure works by *bottom-up evaluation*.
- Most important part of algorithm is *matching* clauses against database.

Soundness If a match is found, applying the substitution to the body of the clause yields atomic formulas in the database.

Completeness If no match is found, there is no such substitution.

Termination Extending the database eventually reaches a fix point.

Program Extraction - Translating

```
Definition
 match_term (T1 T2 : term) : option substitution :=
  match T1 with
  | ident i =>
     match T2 with
     | ident j =>
        if eq_nat_dec i j then
         Some nil
        else None
     | var j => None
     end
  | var i =>
     match T2 with
     | ident j => Some ((i,j)::nil)
     | var j => None
     end
  end.
```

Program Extraction - Translating

```
let match_term t1 t2 =
 match t1 with
 | Ident i ->
   (match t2 with
    | Ident j ->
      (match eq_nat_dec i j with
       | Left -> Some Nil
       | Right -> None)
    | Var j -> None)
 | Var i ->
   (match t2 with
    | Ident j ->
        Some ((i, j) :: Nil)
    | Var j -> None)
```

Program Extraction - Simplifying

```
Lemma forall_dec :
  forall (A:Set)(P:A->Prop)(L:list A),
    (forall (x:A), {P x} + {~P x}) ->
    {forallelts A P L} + {~forallelts A P L}.
Proof.
  intros A P L H.
  induction L.
  left. unfold forallelts.
  intros x H2.
  simpl in H2; contradiction.
  elim IHL; intro IHL2; clear IHL.
  assert ({P a}+{~ P a}).
  ...
  simpl. right; assumption.
Qed.
```

Program Extraction - Simplifying

```
let rec forall_dec l h =
  match l with
  | Nil -> Left
  | Cons (a, l0) ->
    (match forall_dec l0 h with
     | Left -> h a
     | Right -> Right)
```

Program Extraction - Generating

```
Definition eq_term_dec (A1 A2 : term) : {A1 = A2} + {A1 <> A2}.
  intros A1 A2.
  decide equality A1 A2;
    apply eq_nat_dec.
Defined.
```

Program Extraction - Generating

```
let eq_term_dec a1 a2 =
  match a1 with
  | Ident x ->
    (match a2 with
     | Ident n0 -> eq_nat_dec x n0
     | Var n0 -> Right)
  | Var x ->
    (match a2 with
     | Ident n0 -> Right
     | Var n0 -> eq_nat_dec x n0)
```

Binder (DeTreville)

- Binder adds a `says` operator and the notion of *importing/exporting* clauses from one context to another.
- There are several other choices for extending Datalog, Binder is convenient because it is simple and practical.
- Encoding in Coq:

```
Inductive atomic : Set :=
| bare : nat -> list term -> atomic
| says : term -> nat -> list term -> atomic.
```

- Redoing proofs is actually easy, just need some additional checks for equality between principals.

Binder - Distributed User Authorization

```
authority(V) :-
    root says authority(V).

authority(V) :-
    authority(U),
    U says authority(V).

valid-user(V) :-
    authority(U),
    U says valid-user(V).
```

Horn Logic

- Function symbols allow *structured data*, not just identifiers.

  Can model access control lists and capabilities.

  Works on tree data structures (e.g. file systems, XML).

- Encoding in Coq:

```
Inductive term : Set :=
| var : nat -> term
| func : nat -> list term -> term.
```

- The downside is that *we lose general completeness*.

Induction Scheme for Terms

Induction over terms gets complicated by the inner `list term`. Luckily the `Scheme` command automatically generates the correct induction principle.

```
term_rec :
 forall (P : term -> Set) (Q : list term -> Set),
  (forall (n : nat) (l : list term), Q l -> P (func n l)) ->
  (forall n : nat, P (var n)) ->
  Q nil ->
  (forall t : term, P t ->
     forall l : list term, Q l -> Q (t :: l)) ->
  forall t : term, P t.
```

Certified Proof Checker

- We could proceed as before and prove a specific algorithm is sound but not complete.
- Instead we construct a *certified proof checker*.
- This lets any proof generator be used.
- Since its result is checked, this guarantees soundness even if the algorithm is not encoded in Coq.

Encoding Horn Logic Proofs

```
Inductive derive : list form -> atomic -> Prop :=
| derive_step :
    forall (LF : list form)(F : form)(S : substitution),
      In F LF ->
      forallelts atomic
        (fun x => derive LF (subs_atomic S x))
        (body F) ->
      derive LF (subs_atomic S (head F)).

Inductive prf : Set :=
| prf_step : nat -> substitution -> list prf -> prf.
```

Proof Validity

```
Inductive valid_proof : list form -> atom * prf -> Prop :=
| valid_proof_step :
    forall (i:nat)(Prg:list form)(Rule:form)(Body:list atom)
      (Subs:list (nat*term))(Subprfs:list prf)(G:atom),
      i < length Prg ->
      Rule = nth i Prg (clause (atm 0 nil) nil) ->
      Body = map (fun x => subs_atomic x Subs)
             (body Rule) ->
      length Body = length Subprfs ->
      (forall (x:atom*prf),
        In x (zip atom prf Body Subprfs) ->
        valid_proof Prg x
      ) ->
      G = subs_atomic (head Rule) Subs ->
      valid_proof Prg (G, (prf_step i Subs Subprfs)).
```

Properties

- First show `valid_proof` is decidable. From this proof we extract a proof checker function.
- Then show `valid_proof` is sound with respect to `derive`.
- Some subtlety here; we need to make sure extracted code does not make extra assumptions about its data.

  This is true of `valid_proof` because proof data objects are entirely in `Set`.

Encoding BCC

Encoding for calculus of constructions from Bruno Barras' CoC.

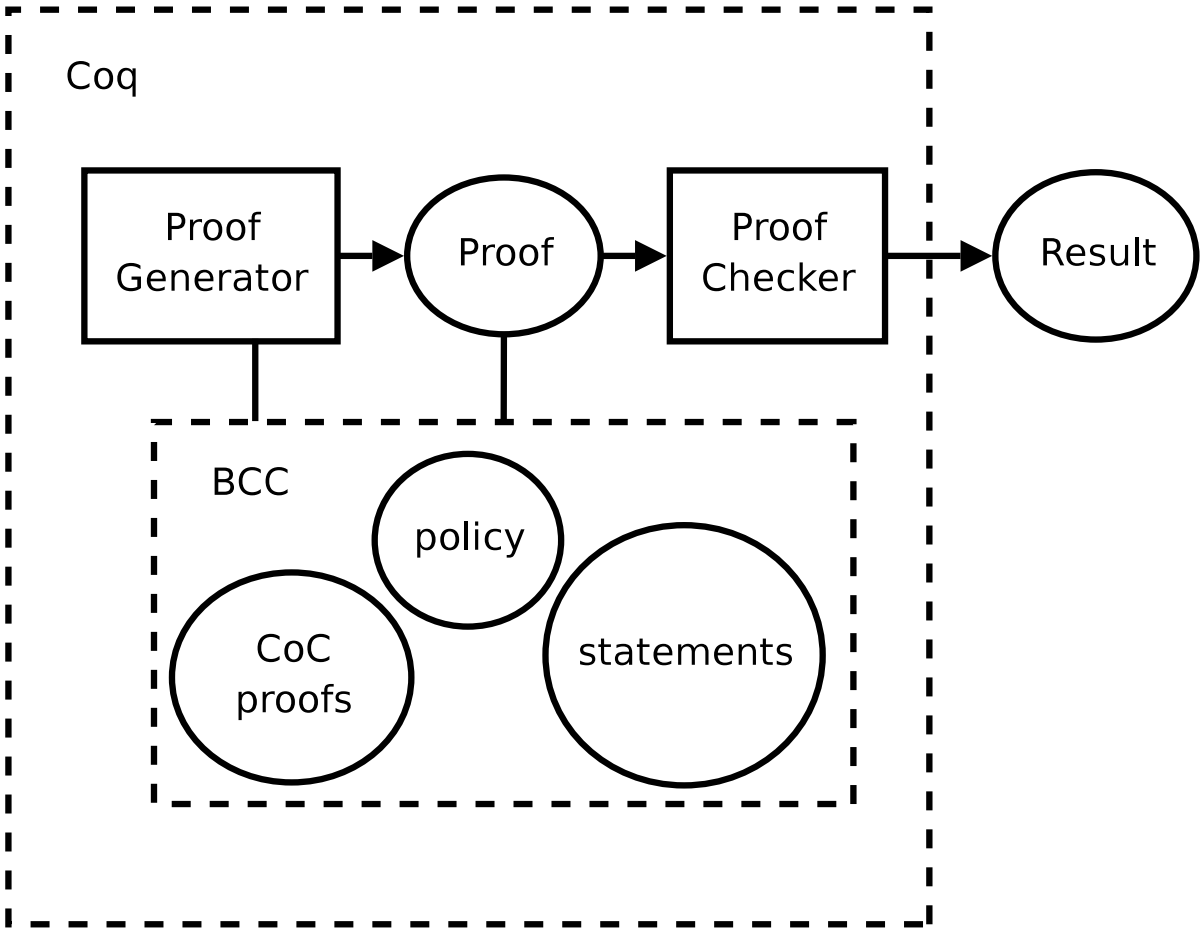
```
Inductive sort : Set :=
| kind : sort
| set : sort
| prop : sort.

Inductive ccterm : Set :=
| Srt : sort -> ccterm
| Ref : nat -> ccterm
| Abs : ccterm -> ccterm -> ccterm
| App : ccterm -> ccterm -> ccterm
| Prod : ccterm -> ccterm -> ccterm.

Inductive term : Set :=
| var : nat -> term
| func : nat -> list term -> term
| cctrm : ccterm -> term.
```

```
Inductive patomic : Set :=
| pred : nat -> list term -> patomic
| sat : list ccterm -> ccterm -> patomic
| believe : list ccterm -> ccterm -> patomic.

Inductive atomic : Set :=
| bare : patomic -> atomic
| says : term -> patomic -> atomic.

Inductive form : Set :=
| clause : atomic -> list atomic -> form
| use_sig : list ccterm -> form -> form
| forallb : nat -> form -> form
| forallcc : ccterm -> form -> form.

Inductive wf : list ccterm -> Prop :=
| wf_nil : wf nil
| wf_var : forall e T s, typ e T (Srt s) -> wf (T :: e)
with typ : list ccterm -> ccterm -> ccterm -> Prop :=
| type_prop : forall e, wf e -> typ e (Srt prop) (Srt kind)
| type_set : forall e, wf e -> typ e (Srt set) (Srt kind)
```

```
| type_var :
    forall e,
    wf e -> forall (v : nat) t, lift t e v -> typ e (Ref v) t
| type_abs :
    forall e T s1, typ e T (Srt s1) ->
    forall M (U : term) s2, typ (T :: e) U (Srt s2) ->
    typ (T :: e) M U -> typ e (Abs T M) (Prod T U)
| type_app :
    forall e v (V : term), typ e v V ->
    forall u (Ur : term), typ e u (Prod V Ur) -> typ e (App u v) (subst v Ur)
| type_prod :
    forall e T s1, typ e T (Srt s1) ->
    forall (U : term) s2, typ (T :: e) U (Srt s2) -> typ e (Prod T U) (Srt s2)
| type_conv :
    forall e t (U V : term),
    typ e t U -> conv U V -> forall s, typ e V (Srt s) -> typ e t V.
```

```
Inductive form : Set :=
| clause : atomic -> list atomic -> form
| use_sig : list ccterm -> form -> form
| forallcc : ccterm -> form -> form.

Inductive derive : list form -> atomic -> Prop :=
| derive_sat :
    forall (LF : list form)(e : list ccterm)(o t : ccterm),
      typ e o t -> derive LF (bare (sat e t))
| derive_believe :
    forall (LF : list form)(e : list ccterm)(o t : ccterm),
      typ e o t -> derive LF (bare (believe e t))
| derive_step :
    forall (LF : list form)(F : form)(S : substitution),
      In F LF ->
      no_free_cc_vars F ->
      forallelts atomic
        (fun x => derive LF (subs_atomic S x))
        (body F) ->
      derive LF (subs_atomic S (head F)).
...
```

BCC - The Big Picture



Size Statistics

Datalog Encoding and proofs are 2500 lines of Coq, extracted to 300 lines of OCaml, of which 50 are redundant definitions of bool, option, etc..

Binder 2600 lines of Coq for encoding and all proofs.

Horn logic Encoding and proofs are 1000 lines of Coq (without completeness), generic prover is 200 lines of Prolog.

BCC Encoding and proofs are 2100 lines of Coq, prover is 500 lines of Prolog.

Future Work

- Characterize completeness conditions for BCC and restriction on calculus of construction terms.
- Prove more meta-theory about BCC to make sure semantic description is correct, for example forms of conservativity.
- Perhaps automate proof checker generation from description of Coq predicate, or alter extraction to do this automatically.
- Develop big realistic examples.

Conclusions

- In the end we have certified implementations of a series of security logics suitable for a variety of access control applications.
- Coq worked well for this application, there were no deal-breakers.

  The hardest things were managing the explosion of lemmas and keeping track of where we were in the proof.
- If you have suggestions for improvement or if I did everything wrong, please let me know!