The need for Capability Policies

- position paper -

Sophia Drossopoulou & James Noble
WG2.16, 13 May 2013

A very powerful program

- Stolen shamelessly from David Wagner, http://www.cs.berkeley.edu/~daw/talks/PLAS06.ps

Object Capabilities

- *Unforgeable* capabilities
  - Possession implies Right
  - No other access control checking
- Principle of Least Authority
  - No Ambient Authority
- Capabilities + Pure Object-Orientation = Object-Capabilities

Object Capabilities

Bob
Carol
foo
Alice

- Metric space: *d(x,y)* = *Dijkstra(x,y)*

Object Capabilities

- Metric space: *d(x,y)* = *Dijkstra(x,y)*

Object Capabilities and JavaScript

- Object Capabilities make secure JavaScript etc *possible*
- But:
  - Code is low level,
  - Code does not explicitly express the capability policy,
  - Security concerns are tangled with functionality concerns,
  - Code more about mechanism (how) than the policy (what).

Capability Policies – Our Position

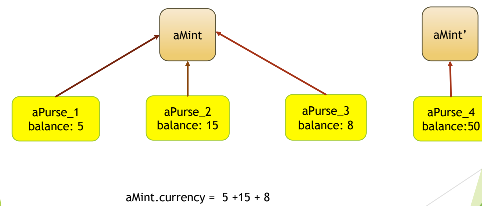
- What are Capability Policies?
- Are Capability Policies Novel?
- Can we reason about Capability Policies?

What are Capability Policies? The mint example

We consider the electronic money as proposed in [MillerEtAl,FinCrypto'00]

- Mints with electronic money,
- Purses held within mints,
- Transfers of funds between purses,
- The *currency* of a mint is the sum of balances of its purses.

Mint example – the objects



aMint.currency = 5 +15 + 8

What are Capability Policies? The mint example

We consider the electronic money as proposed in [MillerEtAl,FinCrypto'00]

- Mints with electronic money,
- Purses held within mints,
- Transfers of funds between purses,
- The *currency* of a mint is the sum of balances of its purses.

Purses trust the mint to which they belong. Programs using the money system trust their purses (and thus the mint). Users trust the money system, but do *not* trust each other.

There is also an implicit assumption that no purses are destroyed.

Mint example – Java code

```
public final class Mint {        }
public final class Purse {
  private final Mint mint;  private long balance;
  public Purse(Mint mint, long balance) {
       if (balance<0) { throw …  };
       this.mint = mint; this.balance = balance; }
  public Purse(Purse prs) {
       mint = prs.mint;  balance = 0; }
  public void deposit(Purse prs, long amnt) {
       if ( mint!=prs.mint || amnt>prs.balance
                           || amnt+balance<0 )
            { throw …   };
       prs.balance -= amnt;  balance += amnt; }
}
```

Capability Policies – the mint example

The policy, as in [MillerEtAl,FinCrypto'00]

- Pol_1 With two purses of the same mint, one can transfer money between them.
- Pol_2 Only someone with the mint of a given currency can violate conservation of that currency.
- Pol_3 The mint can only inflate its own currency.
- Pol_4 No one can affect the balance of a purse they don't have.
- Pol_5 Balances are always non-negative integers.
- Pol_6 A reported successful deposit can be trusted as much as one trusts the purse one is depositing into.

14

The code: policy scattered and tangled

```
public final class Mint {      }
public final class Purse {
   private final Mint mint;  private long balance;
   public Purse(Mint mint, long balance) {
         if (balance<0) { throw … };
         this.mint = mint; this.balance = balance; }
   public Purse(Purse prs) {
         mint = prs.mint;  balance = 0; }
   public void deposit(Purse prs, long amnt) {
         if ( mint!=prs.mint || amnt>prs.balance || amnt+balance<0 )
                   { throw …   };
         prs.balance -= amnt;  balance += amnt; }
}
```

- Pol_2 Only someone with the mint of a given currency can violate conservation of that currency.

We believe that such policies should be explicitly and formally stated, and adherence of the code should be formally verified.

16

Capability Policies – Our Position

- What are Capability Policies?
- Are Capability Policies Novel?
  Formal specification of capability policies poses new questions for specification languages.
- Can we reason about Capability Policies?

17

Capability Policies

- Program centered They talk about properties of programs rather than protocols.
- Fine-grained They talk about individual objects, rather than modules/groups of objects.
- Open They must be satisfied by any use of the code extended in any possible manner (*closed* requirements need only be satisfied by the code itself).
- Rely elements Execution in a state satisfying some condition will lead to new state satisfying new condition.
- Deny elements If we reach a certain state/modify some property, then some other event will happen/will have happened.

18

Capability Policies are Program Centered

They talk about individual objects, rather than modules/groups of objects.

- Pol_1 With two purses of the same mint, one can transfer money between them.
- Pol_2 Only someone with the mint of a given currency can violate conservation of that currency.
- Pol_3 The mint can only inflate its own currency.
- Pol_4 No one can affect the balance of a purse they don't have.
- Pol_5 Balances are always non-negative integers.
- Pol_6 A reported successful deposit can be trusted as much as one trusts the purse one is depositing into.

As opposed to coarse-grained security concerns which restrict control/information flow between components, eg the mint cannot affect the inventory.

19

Capability Policies are Fine-Grained

They talk about individual objects, rather than modules/groups of objects.

- Pol_1 With two purses of the same mint, one can transfer money between them.
- Pol_2 Only someone with the mint of a given currency can violate conservation of that currency.
- Pol_3 The mint can only inflate its own currency.
- Pol_4 No one can affect the balance of a purse they don't have.
- Pol_5 Balances are always non-negative integers.
- Pol_6 A reported successful deposit can be trusted as much as one trusts the purse one is depositing into.

Note that a mint's currency is an *indirect* property of program state, and may depend on several objects, not necessarily reachable from the mint object.
Protocols typically talk about calls to the API, but not about indirect properties. 20

Capability Policies are Open

- Must be satisfied by any extensions of the code extended (*closed* requirements need only be satisfied by the code itself)
  - Subclassing,
  - Mashups,
  - Dynamic loading etc,
- Program verification is usually closed, while web security is open.

21

Capability Policies have Rely & Deny Elements

Rely elements Execution in a state satisfying some condition will lead to new state satisfying new condition.

Deny elements If we reach a certain state/modify some property, then some other event will happen/will have happened.

- Pol_1 With two purses of the same mint, one can transfer money between them.
  Rely: can transfer; Deny: … of same mint
- Pol_2 Only someone with the mint of a given currency can violate conservation of that currency.
  Deny: … only someone of same mint
- Pol_3 The mint can only inflate its own currency.
  Deny: … only inflate

22

Capability Policies have Rely & Deny Elements

Rely elements Execution in a state satisfying some condition will lead to new state satisfying new condition.

Deny elements If we reach a certain state/modify some property, then some other event will happen/will have happened.

- Pol_4 No one can affect the balance of a purse they don't have.
  Deny: … if affect, they must have the purse
- Pol_5 Balances are always non-negative integers.
  Deny: … balance never negative – like 2 state invariant
- Pol_6 A reported successful deposit can be trusted as much as one trusts the purse one is depositing into.
  ???

23

Rely vs Deny

Rely Execution in a state satisfying some condition will lead to new state satisfying new condition.
describe *sufficient* conditions.

Deny If we reach a certain state/modify some property, then some other event will happen/will have happened.
describe *necessary* conditions.

Deny specifications related to, but different from, deny in deny guarantee, correspondence assertions and refinement types.

24

Capability Policies – Our Goals

- What are Capability Policies?
  Capability policies express the security concerns of a program (what).
- Are Capability Policies Novel?
  Formal specification of capability policies poses new questions for specification languages.
- Can we reason about Capability Policies?
  Reasoning that code adheres to capability policies needs to make use of programming languages' "restrictive" features (type, privacy, ownership etc).

25

Rely Elements Reasoning not surprising.

Pol_1 With two purses of the same mint, one can transfer money between them.

Requires a proof that

```
prs1.deposit(prs2,amt)
```

transfers `amt` from `prs1` to `prs2` (Hoare Logic)

26

Deny Elements Reasoning combines disciplines.

Pol_2 Only someone with the mint of a given currency can violate conservation of that currency.

Reasoning about this property combines Hoare-Logic style reasoning, with footprint analysis, and reliance on restrictive features:

- Analysis of the "footprint" of currency, i.e. which objects' state may affect the currency of a mint,
- Use of privacy/finality annotations to deduce which methods may affect the footprint (restrictive language features),
- Analysis of the effect of these methods (Hoare Logic).

27

Deny Elements Reasoning combines disciplines.

```
public final class Mint {      }

public final class Purse {
   private final Mint mint;  private long balance;
    public Purse(Mint mint, long balance) {
          if (balance<0) { throw …  };
          this.mint = mint; this.balance = balance; }
    public Purse(Purse prs) {
          mint = prs.mint;  balance = 0; }
    public void deposit(Purse prs, long amnt) {
          if ( mint!=prs.mint || amnt>prs.balance || amnt+balance<0 ){throw …};
          prs.balance -= amnt;  balance += amnt; }
}
```

- *Footprint*(`aMint.currency()`) `={ p:Purse | p.mint==aMint }.balance`

30

Deny Elements Reasoning combines disciplines.

```
public final class Mint {      }

public final class Purse {
   private final Mint mint;  private long balance;
    public Purse(Mint mint, long balance) {
          if (balance<0) { throw …  };
          this.mint = mint; this.balance = balance; }
    public Purse(Purse prs) {
          mint = prs.mint;  balance = 0; }
    public void deposit(Purse prs, long amnt) {
          if ( mint!=prs.mint || amnt>prs.balance || amnt+balance<0 ){throw …};
          prs.balance -= amnt;  balance += amnt; }
}
```

- *Footprint*(`aMint.currency()`) `={ p:Purse | p.mint==aMint }.balance`
- privacy/finality annotations:
  - Class `Purse` is final, and field `mint` is final.
  - Therefore `mint` can only be set through constructors `Purse(Purse)` and `Purse(Mint,long)`.
  - Therefore `{ p:Purse | p.mint==aMint }` is only affected by these constructors.
  - Field `balance` is private.
  - Therefore *Footprint*(`aMint.currency()`) only affected by methods `deposit(Purse,long)` and constructors `Purse(Purse)` and `Purse(Mint,long)`.

31

Deny Elements Reasoning combines disciplines.

```
public final class Mint {      }

public final class Purse {
   private final Mint mint;  private long balance;
    public Purse(Mint mint, long balance) {
          if (balance<0) { throw …  };
          this.mint = mint; this.balance = balance; }
    public Purse(Purse prs) {
          mint = prs.mint;  balance = 0; }
    public void deposit(Purse prs, long amnt) {
          if ( mint!=prs.mint || amnt>prs.balance || amnt+balance<0 ){throw …};
          prs.balance -= amnt;  balance += amnt; }
}
```

- *Footprint*(`aMint.currency()`) only affected by method `deposit(Purse,long)` and constructors `Purse(Purse)` and `Purse(Mint,long)`.
- Analysis of the effect of these methods/constructors (Hoare Logic):
  - the constructor `Purse(Purse,long)` does not affect the currency.
  - the constructor `Purse(Mint,long)` affects the currency.
  - the method `deposit(Purse,long)` does not affect the currency.
- Therefore, affect `aMint.currency()` only though calling `Purse(aMint,amt)`, and thus only when holding `aMint`.

32

Open Policies need more language features

```
public final class Mint {      }

public       class Purse {
   private final Mint mint;  private long balance;
    public Purse(Mint mint, long balance) {
          if (balance<0) { throw …  };
          this.mint = mint; this.balance = balance; }
    public Purse(Purse prs) {
          mint = prs.mint;  balance = 0; }
    public void deposit(Purse prs, long amnt) {
          if ( mint!=prs.mint || amnt>prs.balance || amnt+balance<0 ){throw …};
          prs.balance -= amnt;  balance += amnt; }
}
```

- Assume that class `Purse` is *not* a final.
- Then, to satisfy **Pol_2** in an open setting, you need to ensure that subclasses will not give access to fields `mint` or `balance`.
- This means that fields `mint` and `balance` should be owned.
- Similar patterns arise in other setting, cf. membranes.

33

Deny Elements can also be achieved through "hand-coded" restrictions

```
def makeMint(name) : any {
   def [ sealer, unsealer ] := makeBrandPair(name)
   def mint {
      to makePurse( var balance:( int >= 0) ) : any {
         def decr( amount:(0..balance) ) : void
                  { balance -= amount  }
         def purse {
            to sprout( ) : any { return mint.makePurse (0) }
            to getDecr( ) : any { return sealer.seal ( decr ) }
            to deposit(amount : int, src) :void {
               unsealer.unseal(src.getDecr()) ( amount )
               balance += amount
            }
         }
      return purse
      }
   }
   return mint
}
```

34

Deny Elements can also be achieved through better language features

```
class Mint.new(name : String) {

   class Purse.new (balance' : Number) is owned {
     var balance: Number is confidential := balance'
   } // owners as readers

   method newPurse(amount : Number) -> Purse {
     return Purse.new(amount)}

   method deposit(from : Purse, to : Purse, amount : Natural) -> Done {
     if ((amount > 0) && {(from.balance - amount) >= 0)}
        then {
          from.balance := from.balance - amount
          to.balance := to.balance + amount
        } else { Exception.raise("Fraud detected") }
   }
}
```

35

Deny Elements can also be achieved through better language features

```
class Mint.new(name : String) {

  method newPurse(amount : Number) ownedby(o) -> Purse {
     return object is owned(self &&& o) { var balance := amount }
  } // owners as readers or owners as modifiers

  method balance(p : Purse) { p.balance }

  method deposit(from : Purse, to : Purse, amount : Natural) -> Done {
     if ((amount > 0) && {(from.balance - amount) >= 0)}
        then {
          from.balance := from.balance - amount
          to.balance := to.balance + amount
        } else { Exception.raise("Fraud detected") }
   }
}
```

36

Deny Elements can also be achieved through better language features

```
class Mint.new(name : String) {
   def ledger = WeakMap.new<Purse,Number> owned(self)

   method newPurse(amount : Number) -> Purse {
     def p = object { var _ }
     ledger[p] = amount
     return p
   }

   type Purse = Object

   method deposit(from : Purse, to : Purse, amount : Natural) -> Done {
     if ((amount > 0) && ((ledger[from] - amount) >= 0)
           && (ledger.contains(to)))
        then {
          ledger[from] := ledger[from] - amount
          ledger[to] := ledger[to] + amount
        }
   }
```

37

Further Work

- Design a specification language for Capability Policies – temporal logic?
- Investigate what trust means (Pol_6).
- Investigate "Restrictive Programming Language Features" to support Capability Policies.
- Develop Mixed Logics to Reason about Programs' adherence to Capability Policies.

38