How to Break the Bank: Semantics of Capability Policies

Sophia Drossopoulou[1] and James Noble[2]

[1] Imperial College London s.drossopoulou@imperial.ac.uk
[2] Victoria University of Wellington kjx@ecs.vuw.ac.nz

Abstract. The object capability model is a de-facto industry standard widely adopted for the implementation of secure software. We call *capability policies* the policies enforced by programs using object capabilities. Such policies tend to restrict the objects and the circumstances which may access services. In this paper we argue that capability policies should be made explicit and written separately from the code implementing them. We also argue that the specification of capability policies requires concepts that go beyond the features of current specification languages. Moreover, we argue that we need methodologies with which to prove that programs adhere to their capability policies as specified.
To give precise semantics to capability policy specifications, we propose *execution observations*, which talk about various properties of a program's execution. We use execution observations to write the formal specification of five out of the six informal policies in the mint example, famous in the object capability literature. In these specifications, the conclusions but also the premises may relate to the state before as well as after execution, the code may be existentially or universally quantified, and interpretation quantifies over all modules extending the current module. In statically typed languages, adherence of code to the capability policies relies heavily on the guarantees provided by type system features such as final and private.

1 Introduction

Capabilities — unforgeable authentication tokens — have been used to provide security and task separation on multi-user machines since the 60s [5], *e.g.* PDP-1, operating systems *e.g.* CAL-TSS [13], and the CAP computer and operating system [38]. In capability-based security, resources can only be accessed via capabilities: possessing a capability gives the right to access the resource represented by that capability.

Object capabilities [22] apply capabilities to object-oriented programming. In an object capability system, an object is a capability for the services the object provides: any part of a program that has a reference to an object can always use all the services of that object. To restrict authority over an object, programmers must create an intermediate proxy object which offers only restricted services, delegating them back to the original object.

Object capabilities afford simpler and more fine-grained protection than privilege levels (as in Unix), static types, ad-hoc dynamic security managers (as in Java or JSand [1]), or state-machine-based event monitoring [2]. Object capabilities have been adopted in several programming languages [24, 19, 36] and are increasingly used for the provision of security in web programming in industry [25, 37, 32].

Thus, object capabilities are employed to enforce policies which restrict access to services: which objects, and under what conditions, may activate these services. We call such policies, which regulate access to services, *capability policies.* Capability policies are *program centred*, *fine grained*, *open* in the sense that they specify aspects of the behaviour of all possible extensions of a program, and have *necessary* as well as *sufficient* parts; the latter require that certain effects may only take place if the originating code or the runtime context satisfy some conditions, [7].

The key problem with object capability programming as practiced today is that — because capabilities are just objects — code manipulating capabilities is tangled together with code supporting the functional behaviour of the program. The actual capability policies enforced by a program are *implicit* and scattered throughout the program's code, and the functionality concerns are tangled with those of the capability policy.

We argue that capability policies should be specified separately from the program implementing them. We also argue that the specification of capability policies requires features that go beyond what is available in current specification languages.

We propose that capability policies can be specified through *execution observations*, which are, essentially observations relating to program execution, accessibility, reachability and tracing. For example, execution observations can say things like "execution of a given code snippet in a given runtime context will access a certain field", or "it is possible to reach certain code through execution of some initial code".

We follow the Mint example [24] to illustrate our ideas; using execution observations we give precise meaning to five out of the six policies proposed informally in that paper. In these policies, the conclusions *but also the premises* may relate to the state before as well as after execution, the code may be existentially or universally quantified, and interpretation quantifies over all modules extending the current module. In the process of developing the mint specifications, we were surprised by the many different, and plausible interpretations we found for the policies.

The paper is organised as follows: Section 2 presents the Mint [19] as an example of object capability programming, implemented in Joe-E/Java. Based on that example, Section 3 distills the characteristics of capability policies. Section 4 then outlines executions observations, while section 5 uses them to express those policies, and discusses alternative interpretations. Section 6 discusses alternative meanings of these policies. Section 7 surveys related work, and Section 8 concludes.

2 The Mint: an Object Capability Example

We use as running example a system for electronic money proposed in [24]. This example allows for mints with electronic money, purses held within mints, and transfers of funds between purses. The *currency of a mint* is the sum of the balances of all purses created by that mint. The standard presentation of the mint example defines six capability policies, which we repeat here, as they were described in [24]:

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.

```
//  module M_Mint
public final class Mint {       }


//  module M_Purse
public final class Purse {
    private final Mint mint;
    private long balance;
    // INV: balance ≥ 0


    // PRE: balance ≥ 0
    // POST: result.mint=mint ∧ result.balance = 0
    public Purse(Mint mint, long balance) {
       if (balance<0){ throw new IllegalArgtException();  };
       this.mint = mint;
       this.balance = balance;
    }

    // PRE: true
    // POST: result.mint=prs.mint ∧ result.balance = 0
    public Purse(Purse prs ) {
       this.mint = prs.mint;
       this.balance = 0;
    }

    // PRE: this.mint=prs.mint ∧ amt≤prs.balance ∧
    //      amt+this.balance≥0
    // POST: this.balance=this.balance$_{pre}$+amnt ∧
    //       prs.balance=prs.balance$_{pre}$- amnt
    public void deposit(Purse prs, long amnt) {
       if ( mint!=prs.mint
            || amnt>prs.balance || amnt+balance<0 )
                      { throw new IllegalArgtException(); };
       prs.balance -= amnt;
       balance += amnt;   }
    }
}
```

Fig. 1. The Mint example, code taken from [19], specifications added by us

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.

An immediate consequence of these policies is that the mint capability gives its holder the ability to subvert the currency system by "printing money", and that "printing money" is only possible, if one holds the mint. This means that while purse capabilities may safely be passed around the system, the mint capability must be carefully protected. This also means that protecting the mint suffices in order to protect the currency.

Several different implementations have been proposed for the mint. Fig.1 contains an implementation in Joe-E [19], a capability-oriented subset of Java, which restricts static variables and reflection.

In Fig.1, the policies are adhered to through the interplay of appropriate actions in the method bodies (*e.g.* the check in line 15), with the use of Java's *restrictive* language features (private members are visible to the same class only; final fields cannot be changed after initialisation; and final classes cannot be extended). The code concerned with the functional behaviour is tangled with the code implementing the policy (*e.g.* in deposit, lines 35-36 are concerned with the functionality, while lines 32-34 are concerned with **Pol_2**). The implementation of *one* policy is scattered throughout the code, and may use explicit runtime tests, as well as restrictive elements (*e.g.* **Pol_2** is implemented through a checks in line 32, the private and final annotations, and the initialisation in line 24). Note that an apparently innocuous change to this code — such as a public getMint accessor that returned a purse's mint — would be enough to leak the mint to untrusted code, destroying the security of the whole system, and thus break the bank!

An alternative implementation of the mint example appears in figure 2. Here, the Purse objects are used as indices into a map, which is held and administered by the Mint objects. A similar scattering and tangling of the policies and the functionality may be observed in this implementation. Policy **Pol_2** is implemented through the runtime tests in lines 17 and 19, though the transfer of moneys in lines 21 and 22, and through the use of the private modifier on the database field.

Comparison with Hoare-logic style specification In the code from Fig. 1, we have given an specification in terms of PRE- and POST-conditions, in a style like that of JML. Note that except for **Pol_5**, this specification does not imply the capability policies.

No garbage collection of Purses In all above, there is an implicit assumption that no purses are destroyed. This assumption is necessary because destruction of a purse would decrease the currency of a mint, in opposition to **Pol_3**.

```
//  module M_alt_Purse
public final class Purse {       }

//  module M_alt_Mint
 public final class Mint {
     private final HashMap<Purse,long> database
                                 = new HashMap<>();

     public Purse makePurse(long balance) {
          Purse p = new Purse();
          database.put(p,balance);
          return p;
     }

    public void deposit(Purse from, Purse into, long amnt) {
           if ( (amount < 0)
                || (!database.contains(from))
                || (database.get(from) < amnt)
                || (!database.contains(into)) )
                         { throw new IllegalArgtException(); };
           database.put(from, database.get(from) - amnt);
           database.put(into, database.get(into) + amnt);
      }
 }
```

Fig. 2. An alternative Mint implementation using a map as a database.

3 Capability Policies

We use the term *capability policy* to describe policies which restrict the circumstances under which objects may have access to services. A range of such capability policies are discernible from the literature [22, 24, 23].

In the case of the Mint example, and apart from **Pol_6**, which requires separate studies [27], the policies from the previous section are concerned with the following services:

S_1 Transfer of money between purses.
S_2 Affecting the balance of a purse.
S_3 Affecting the currency of a mint.

Policy **Pol_1** expresses conditions for **S_1**: the service is protected by the respective purses. Policies **Pol_4** and **Pol_5** express conditions for **S_2**: the service is protectedby the purse, and the balance is always positive. Finally, **Pol_2** and **Pol_3** express conditions for **S_3**: the service is protected by the mint, and the currency can only increase.

Capability policies generally have the following characteristics:

- They are *program centered*: they talk about properties of programs rather than properties of specifications or protocols.
- They are *fine-grained*: they can talk about *individual objects*, while *coarse-grained* policies only talk about large components such as file servers or the DOM.
- They are *open*. *Open* requirements must be satisfied for any use of the code *extended in any possible manner* — e.g. through dynamic loading, inheritance, subclassing, mashups, mixins, reflection, intercession, or any other extension mechanism supported by the programming language. This is in contrast to *closed* specifications that need only be satisfied for the actual code snippet itself.
- They have *necessary* as well as *sufficient* elements. Sufficient elements essentially promise that execution of a code snippet in a state satisfying a given pre-condition will reach another state which satisfies some post-condition [10]. Necessary elements promise that if execution of a code snippet reaches a certain state, or changes state in a certain way, or accesses some program entity, then the code snippet must satisfy some given properties.

4 Execution Observations for the Semantics of Capability Policies

In this section we introduce *execution observations*, the concepts necessary to give precise meaning to policies. In our yet unpublished report [8] we describe their manifestation in a "capability-safe" Java-subset and give precise definitions. We believe that such execution observations can easily be defined for a range of different programming languages and paradigms. In this paper, we bring out the most salient issues of execution observations.

Modules and Linking To model the open nature of capability policies, we need to describe both the program we are checking, and potential extensions of that program (through subclasses, mashups, imports etc). For this we use *modules*, M, to denote programs, and $*$ to describe the combination of two programs into one larger program.

Adherence to policies often relies on the correct use of restrictive features. We support the method and class annotations private and final. The type rules of Java forbid access to private fields or methods outside their classes, forbid extensions of final classes, forbid redefinitions of final methods in subclasses, and forbid assignment to final fields outside their constructor [8].

The $*$ operator links modules together into new modules. Thus, $\mathsf{M}_{\mathsf{Purse}} * \mathsf{M}_{\mathsf{Mint}}$ is a module. Linking performs some compatibility checks, and therefore $*$ is only partially defined. For example, because the field balance is private, $\mathsf{M}_{\mathsf{Purse}} * \mathsf{M}'$ would be undefined, if M' contained the expression newPurse.balance. The operation $*$ is only defined if it gives rise to a well-formed module.

Code Modules are not directly executable, but are necessary for the execution of *code snippets*. We use the variables code, code$'$ to range over code snippets.

Runtime Configurations and Code Execution Execution takes place in the context of runtime configurations $\kappa \in RTConf$. A configuration is a stack frame and heap. A stack frame is a tuple consisting of the following four components: the address of the receiver, a mapping giving values to the formal parameters, the class identifier, and the method identifier of the method being executed. A heap is a mapping from object addresses to objects.

Execution of a code snippet code for a module M takes a configuration κ and returns a value v and a new configuration κ'. We describe this through a large step semantics, of the shape $\mathsf{M},\ \kappa,\ \mathsf{code} \rightsquigarrow \kappa',\ \mathsf{v}'$.

Reached and Arising Snapshots When verifying adherence to policies, it is essential to consider only those snapshots (*i.e.,* configuration and code pairs) which may arise through the execution of the given modules. For example, if we considered *any* well-formed snapshots (well-formed in the sense of the type system), then we would be unable to show that **Pol_5** is obeyed by the mint example. Namely, **Pol_5** guarantees that balances are always positive: configurations where the balance is negative are well-formed, but will never actually arise in the execution of the program.

$\mathcal{R}each(\mathsf{M}, \kappa, \mathsf{code})$ is the set of snapshots corresponding to the start of the execution of the body of any constructor or method called in the process of executing code in the context of M and κ. For example, $(\kappa_2, \mathsf{this.mint := prs.mint;\ this.balance := 0}) \in \mathcal{R}each(\mathsf{M}, \kappa_1, \mathsf{p1.deposit(p2);\ p3 = newPurse(p2)})$. Note that $\mathcal{R}each(\mathsf{M}, \kappa, \mathsf{code})$, corresponds to the *complete* body of a method; for example, $(\kappa_3, \mathsf{this.balance := 0}) \notin \mathcal{R}each(\mathsf{M}, \kappa_4, \mathsf{p3 = newPurse(p2)})$ for any κ_3 and κ_4. $\mathcal{R}each(\mathsf{M}, \kappa, \mathsf{code})$ is always defined, even though it may be infinite if execution of $\mathsf{M},\ \kappa,\ \mathsf{code}$ does not terminate.

$\mathcal{A}rising(\mathsf{M})$ is the set of snapshots which may be reached during execution of some initial snapshot, κ_0, code_0. Similarly to $\mathcal{R}each(\mathsf{M}, \kappa, \mathsf{code})$, the function $\mathcal{A}rising(\mathsf{M})$ is always defined.

Accessible and Used Objects As we have already discussed, objects protect services, *i.e.* some policies may require that services are only accessible through certain objects. Therefore, availability of a service is predicated on accessibility of the corresponding object. Therefore we need to model accessibility of objects.

We distinguish between $\mathcal{A}ccAll(\mathsf{M}, \kappa)$ — the set of all objects which are accessible from the frame in κ through *any* path — and $\mathcal{A}ccPub(\mathsf{M}, \kappa)$ — the set of all objects accessible through paths which include only public fields, and private fields of objects of the same class as this.

The notation $\mathsf{z} :_\kappa \mathsf{c}$ indicates that z is the name of an object which exists in the heap of κ and belongs to class c — with no requirement that there should be a path from the frame to this object.

The notation $\kappa \in \mathsf{c}$ expresses that the currently executing method in κ comes from class c, while $\kappa \in \mathsf{M}$ expresses that the class of the currently executing method is defined in module M.

$\mathcal{U}sed(\mathsf{M}, \kappa, \mathsf{code})$ is the set of all addresses used during execution of code in the configuration κ.

Paths, Pure Expressions, and Predicates Capability policies are program-centered, therefore in order to express their semantics we need to be able to talk about all program entities, such as paths. For example, mint, prs.mint are paths.
Paths are interpreted in the context of runtime configurations,
$\lceil \cdot \rfloor \,:\, Path \longrightarrow\, RTConf \,\longrightarrow\, Value$
so that $\lceil \mathsf{p} \rfloor_\kappa = v$ if p is a path and $\emptyset, \kappa, \mathsf{p} \rightsquigarrow \kappa, v$.
Functions and predicates are interpreted in the expected manner
$\lceil \cdot \rfloor \,:\, Func\_Id \times Var\_Id^* \longrightarrow\, RTConf \,\longrightarrow\, Value$
$\lceil \cdot \rfloor \,:\, Pred\_Id \,\longrightarrow\, \mathcal{P}\,(Value^*)$
involving any necessary unfoldings of the definitions. Therefore,
$\lceil \mathsf{f}(\mathsf{p1}, ...\mathsf{pn}) \rfloor_\kappa = \lceil fFbody[\mathsf{p1}/x1, ...\mathsf{pn}/xn] \rfloor_\kappa$, where $fBody$ is the function definition of f, with free variables $x1$,... xn. Finally, $\lceil P(p1, ...pn) \rfloor_\kappa = \lceil P \rfloor (\lceil p1 \rfloor_\kappa, ...\lceil pn \rfloor_\kappa)$.

Defining the currency In the Mint example, and using the code from Fig. 1, the function Currency is defined as follows

$$\mathsf{Currency}(\mathsf{mnt}) = \textstyle\sum_{\mathsf{p} \in Ps(\mathsf{mnt})} \mathsf{p.balance}$$
$$\text{where } Ps(\mathsf{mnt}) = \{\mathsf{p} \mid \mathsf{p} : \mathsf{Purse} \,\wedge \mathsf{p.mint} = \mathsf{mnt}\}$$

On the other hand, using the code from Fig. 2, the Currency is defined as follows

$$\mathsf{Currency}(\mathsf{mnt}) = \textstyle\sum_{\{\ \mathsf{p} \ \mid\ \mathsf{mnt.database.contains(p)}\ \}} \mathsf{mnt.database.get(p)}$$

5 Semantics of the Mint policies

We now turn our attention to the precise meaning of the first five policies from the Mint example. (We do not address the sixth policy as our formalisation does not yet incorporate trust). We discuss the policies in order of increasing complexity of their specification, rather than in numerical order.

Note that in the following we are specifying the policies regardless of whether the code is taken from Fig. 1, or from Fig. 2. We have already given definitions for $\mathsf{Currency}$ as defined in Fig. 1 and in Fig. 2. Wrt to the term $\mathsf{prs.balance}$, when taking the Fig. 2 version, the term $\lceil \mathsf{prs.balance} \rfloor_\kappa$ is a shorthand for $\lceil \mathsf{mnt.database.get(prs)} \rfloor_\kappa$ where mnt is such that $\mathsf{mnt.database.contains(prs)}$.

The fifth policy **Pol_5**, "Balances are always non-negative integers", is akin to a class invariant [20, 28, 33]. We can express the policy directly by requiring that a module M satisfies **Pol_5**, if for all M' legal extensions of M, and snapshots $(\kappa, \_)$ arising through execution of the augmented program $\mathsf{M} * \mathsf{M}'$, the balance is positive in κ.

$$\boxed{\begin{array}{c} \mathsf{M} \models \textbf{Pol\_5} \\ \text{iff} \\ \forall\, \mathsf{M}'.\ \forall(\kappa, \mathsf{code}) \in \mathcal{A}rising(\mathsf{M} * \mathsf{M}').\ \forall \mathsf{prs} :_\kappa \mathsf{Purse}. \\ \lceil \mathsf{prs.balance} \rfloor_\kappa \geq 0 \end{array}}$$

Note that the arising snapshots are considered in the context of the *extended* module $\mathsf{M} * \mathsf{M}'$, where M' is universally quantified. This reflects the open nature of capability policies, and allows calling methods and accessing fields defined in M but also in M' before reaching the snapshot (κ, code).

Note also that $\mathcal{A}rising(\mathsf{M} * \mathsf{M}')$ catches snapshots at the *beginning* of a method execution. Therefore, if a method were to temporarily set balance to a negative value, but restored it to a positive value before returning, would *not* violate **Pol_5**.

The third policy **Pol_3**, stating "The mint can only inflate its own currency", could mean that the currency of a mint never decreases, or that the mint cannot affect the currency of a different mint. As we shall see later on, the second interpretation is a corollary of **Pol_2**; here we analyse the first interpretation:

$$\boxed{\begin{array}{c} \mathsf{M} \models \textbf{Pol\_3} \\ \text{iff} \\ \forall\, \mathsf{M}'.\ \forall(\kappa, \mathsf{code}) \in \mathcal{A}rising(\mathsf{M} * \mathsf{M}').\ \forall \mathsf{mnt} :_\kappa \mathsf{Mint}. \\ \mathsf{M} * \mathsf{M}', \kappa, \mathsf{code} \leadsto \kappa', \mathsf{v} \\ \Longrightarrow \\ \lceil \mathsf{Currency(mnt)} \rfloor_\kappa \leq \lceil \mathsf{Currency(mnt)} \rfloor_{\kappa'} \end{array}}$$

Namely, we require that for any arising snapshot (κ, code), and any execution originating from (κ, code) and leading to a new configuration κ', the $\mathsf{Currency}$ at the old configuration is less than or equal to the currency at the new configuration. Therefore, in the conclusion we talk about the values of functions in the old configuration (*i.e.*

$\lceil \mathsf{Currency(mnt)} \rfloor_\kappa$) as well as those in the new configuration (*i.e.* $\lceil \mathsf{Currency(mnt)} \rfloor_{\kappa'}$). Conclusions which are in terms of the old as well as the new state are common in standard approaches to program specification. **Pol_3** describes a monotonic property, and is therefore related to history invariants [14].

The first policy. **Pol_1** states "With two purses of the same mint, one can transfer money between them". We can understand **Pol_1** to mean that if p1 and p2 are purses of the same mint, then the method call p1.deposit(p2, m) will transfer the money. In section 6, we shall present two other possible meanings for this policy. We write this first interpretation of **Pol_1** as:

$$\begin{array}{c}
\mathsf{M} \models \mathbf{Pol\_1A} \\
\text{iff} \\
\forall\, \mathsf{M}'.\forall\, (\kappa, \mathsf{p1.deposit(p2,m)}) \in \mathcal{A}rising(\mathsf{M} * \mathsf{M}'). \\
\forall \mathsf{p1}, \mathsf{p2} :_\kappa \ \mathsf{Purse}.\ \lceil \mathsf{p1.mint} \rfloor_\kappa = \lceil \mathsf{p2.mint} \rfloor_\kappa \quad \wedge \quad \lceil \mathsf{p2.balance} \rfloor_\kappa \geq \mathsf{m} \\
\wedge \quad \mathsf{M} * \mathsf{M}', \kappa, \mathsf{p1.deposit(p2,m)} \leadsto \kappa', \mathsf{v} \\
\Longrightarrow \\
\lceil \mathsf{p1.balance} \rfloor_{\kappa'} = \lceil \mathsf{p1.balance} \rfloor_\kappa + \mathsf{m} \quad \wedge \quad \lceil \mathsf{p2.balance} \rfloor_{\kappa'} = \lceil \mathsf{p2.balance} \rfloor_\kappa - \mathsf{m}.
\end{array}$$

The specification **Pol_1A** again ranges over all module extensions, M′. This policy is stated as a sufficient condition, and is related to a Hoare triples,[3]. The quantification over modules M′ requires that the code M′ can do nothing to break the behaviour of the deposit method from M, thus either requiring the use of restrictive features (*e.g.* forcing the method deposit to be final, or the class Purse to be final, or package confined), or the use of contracts, where subclasses are implicitly expected to satisfy the superclass's contract,

The fourth policy **Pol_4**, "No one can affect the balance of a purse they don't have", says that if some runtime configuration affects the balance of some purse prs, then the original runtime configuration must have had access to the prs itself.

$$\begin{array}{c}
\mathsf{M} \models \mathbf{Pol\_4} \\
\text{iff} \\
\forall\, \mathsf{M}',\ (\kappa, \mathsf{code}) \in \mathcal{A}rising(\mathsf{M} * \mathsf{M}').\ \ \forall \mathsf{prs} :_\kappa \mathsf{Purse}. \\
\mathsf{M} * \mathsf{M}', \kappa \leadsto \kappa', \mathsf{v}
\end{array}$$

[3] The corresponding Hoare triple would be

$$\begin{array}{c}
\{\ \mathsf{p1.mint} = \mathsf{p2.mint}\ \wedge \mathsf{p1.balance} = \mathsf{k1}\ \wedge\ \mathsf{p2.balance} = \mathsf{k2} + \mathsf{m}\ \} \\
\mathsf{p1.deposit(p2,m)} \\
\{\ \ \mathsf{p1.balance} = \mathsf{k1} + \mathsf{m}\ \wedge\ \mathsf{p2.balance} = \mathsf{k2}\ \}
\end{array}$$

$$\begin{array}{c} \wedge \quad \lceil \mathsf{prs.balance} \rfloor_{\kappa} \neq \lceil \mathsf{prs.balance} \rfloor_{\kappa'} \\ \Longrightarrow \\ \lceil \mathsf{prs} \rfloor_{\kappa} \in \mathcal{U}sed(\mathsf{M} * \mathsf{M}', \kappa, \mathsf{code}) \end{array}$$

Note that in contrast to the previous policies, and in contrast to the standard approach to program specification, the premise of the policy is in terms of both the old configuration (here $\lceil \mathsf{prs.balance} \rfloor_{\kappa}$ and the new configuration (here $\lceil \mathsf{prs.balance} \rfloor_{\kappa'}$).

The second policy. **Pol_2**, stating "Only someone with the mint of a given currency can violate conservation of that currency.", is similar to **Pol_4**, in that it mandates that a change (here a change in the currency) may only happen if the originating configuration had access to an entity (here access to the mint).

$$\begin{array}{c} \mathsf{M} \models \mathbf{Pol\_2} \\ \text{iff} \\ \forall\, \mathsf{M}', (\kappa, \mathsf{code}) \in \mathcal{A}rising(\mathsf{M} * \mathsf{M}').\ \forall \mathsf{mnt} :_{\kappa} \mathsf{Mint}. \\ \mathsf{M} * \mathsf{M}', \kappa, \mathsf{code} \leadsto \kappa', \mathsf{v} \\ \wedge\ \lceil \mathsf{Currency(mnt)} \rfloor_{\kappa} \neq \lceil \mathsf{Currency(mnt)} \rfloor_{\kappa'} \\ \Longrightarrow \\ \lceil \mathsf{mnt} \rfloor_{\kappa} \in \mathcal{U}sed(\mathsf{M} * \mathsf{M}', \kappa, \mathsf{code}) \end{array}$$

Policy Characteristics The meanings of policies given in the previous section vary, but they share common characteristics:

- They refer to a fixed module M, and all its legal extensions M'.
- They specify that execution of some code, under some conditions, guarantees some conclusions.
- Both conditions *and* conclusions may refer to properties of the state *before* as well as *after* execution.
- The code may be universally or existentially quantified, or explicitly given.

6 Alternative interpretations of the Mint Policies

Because our policy descriptions have precise semantics — unlike the informal English policies from the original Mint example — a single English policy can have a number of plausible interpretations in our notation. We explore some of these alternatives here; we were surprised how many different interpretations we uncovered while analysing this example.

The first policy revisited. **Pol_1** states "With two purses of the same mint, one can transfer money between them". In section 4.2 we proposed as possible meaning that the call $\mathsf{p1.deposit(p2)}$ will transfer the money. This is perhaps an over-specification, as it prescribes *how* the transfer is to take place — by calling the $\mathsf{p1.deposit(p2)}$ method. Alternatively, we may want to require only that it is *possible* for the transfer to take place, without constraining the program design. We can define a second, more general version of the policy, which only requires the existence of a code snippet that performs the transaction, provided that purses $\mathsf{p1}$ and $\mathsf{p2}$ share the same mint, that $\mathsf{p2}$ has sufficient funds, and that they are both accessible in κ without reading private fields ($\mathcal{A}ccPub(\mathsf{M},\kappa)$).

Module M satisfies policy **Pol_1B**
iff

$$\begin{gathered}\forall(\kappa,\_)\in\mathcal{A}rising(\mathsf{M}).\ \forall \mathsf{p1},\mathsf{p2}\ :_\kappa \mathsf{Purse}.\\ \lceil \mathsf{p1.mint}\rfloor_\kappa = \lceil \mathsf{p2.mint}\rfloor_\kappa \ \wedge\ \lceil \mathsf{p2.balance}\rfloor_\kappa \geq \mathsf{m}\\ \wedge\ \lceil \mathsf{p1}\rfloor_\kappa, \lceil \mathsf{p2}\rfloor_\kappa \in \mathcal{A}ccPub(\mathsf{M},\kappa)\\ \Longrightarrow\\ \exists\ \mathsf{code}.\ \forall \mathsf{M}'.\\ \mathsf{M} * \mathsf{M}', \kappa, \mathsf{code} \ \rightsquigarrow\ \kappa', \mathsf{v}\\ \wedge\ \ \lceil \mathsf{p1.balance}\rfloor_{\kappa'} = \lceil \mathsf{p1.balance}\rfloor_\kappa + \mathsf{m}\\ \wedge\ \ \lceil \mathsf{p2.balance}\rfloor_{\kappa'} = \lceil \mathsf{p2.balance}\rfloor_\kappa - \mathsf{m}.\end{gathered}$$

Note that this policy requires that execution of the code has the required properties for *all* extending modules M'.

A third possible meaning of **Pol_1** is that $\mathsf{deposit}$ can be called successfully only if the two purses belonged to the same mint:

Module M satisfies policy **Pol_1C**
iff

$$\begin{gathered}\forall \mathsf{M}'.\ \forall(\kappa, \mathsf{p1.deposit(p2)})\in\mathcal{A}rising(\mathsf{M} * \mathsf{M}').\\ \mathsf{M} * \mathsf{M}', \kappa, \mathsf{p1.deposit(p2, m)}\ \rightsquigarrow\ \kappa', \mathsf{v}\\ \Longrightarrow\\ \lceil \mathsf{p1.mint}\rfloor_\kappa = \lceil \mathsf{p2.mint}\rfloor_\kappa\end{gathered}$$

The requirement $\mathsf{M} * \mathsf{M}', \kappa, \mathsf{p1.deposit(p2, m)}\ \rightsquigarrow\ \kappa', \mathsf{v}$ is crucial in the premise, in that it ensures that execution does not lead to an error (our current definition of the language $\mathcal{C}_\mathsf{j}$ does not support exceptions). Note, also, that in this specification the conclusion is only concerned with properties observable in the original configuration, κ, while the premise is concerned with properties observable in κ as well as κ'. This reflects the deny nature of the policy.

Finally, a fourth, and more straightforward meaning of **Pol_1** would mandate that the balance of a purse p1 may change only if deposit was executed on p1 or with p1 as an argument. This can be expressed as follows:

$$
\begin{array}{c}
\text{Module } \mathsf{M} \text{ satisfies policy } \textbf{Pol\_1D} \\
\text{iff} \\
\forall \mathsf{M}'. \forall (\kappa, \mathsf{code}) \in \mathcal{A}rising(\mathsf{M} * \mathsf{M}').\ \forall \mathsf{p1} :_{\kappa} \mathsf{Purse}. \\
\mathsf{M} * \mathsf{M}', \kappa, \mathsf{code} \rightsquigarrow \kappa', \mathsf{v} \\
\wedge\ \lceil \mathsf{p1.balance} \rfloor_{\kappa} \neq \lceil \mathsf{p1.balance} \rfloor_{\kappa'} \\
\Longrightarrow \\
\exists \kappa',\ \text{s.t.} \\
(\kappa', \_) \in \mathcal{R}each(\mathsf{M} * \mathsf{M}', \kappa, \mathsf{code}) \\
\wedge \quad \kappa' = (\_, \_, \mathsf{Purse}, \mathsf{deposit}) \\
\wedge \quad (\lceil \mathsf{this} \rfloor_{\kappa'} = \lceil \mathsf{p1} \rfloor_{\kappa} \vee \lceil \mathsf{prs} \rfloor_{\kappa'} = \lceil \mathsf{p1} \rfloor_{\kappa})
\end{array}
$$

The assertion $(\kappa', \_) \in \mathcal{R}each(\mathsf{M}*\mathsf{M}', \kappa, \mathsf{code}) \wedge \kappa' = (\_, \_, \mathsf{Purse}, \mathsf{deposit})$ guarantees that execution of the snapshot (κ, code) will reach a point where it calls the method deposit from Purse. The assertion $(\lceil \mathsf{this} \rfloor_{\kappa'} = \lceil \mathsf{p1} \rfloor_{\kappa} \vee \lceil \mathsf{prs} \rfloor_{\kappa'} = \lceil \mathsf{p1} \rfloor_{\kappa})$ guarantees that the receiver or the first argument of that method call will be $\lceil \mathsf{p1} \rfloor_{\kappa}$.

The second policy revisited. **Pol_2**, "Only someone with the mint of a given currency can violate conservation of that currency." mandates that a change in the currency may only happen if the originating configuration had access to the mint. In section 4.2 we took "access to" to mean that the code executed eventually would read the mint object (i.e. that the mint was in the set $\mathcal{U}sed$). We see three alternative interpretations for the meaning of *having access to*:

1. $\lceil \mathsf{mnt} \rfloor_{\kappa} \in \mathcal{U}sed(\mathsf{M}, \kappa, \mathsf{code})$, i.e. that execution of code in the context of κ will at some point use the object mnt.
2. $\lceil \mathsf{mnt} \rfloor_{\kappa} \in \mathcal{A}cc\mathcal{A}ll(\mathsf{M}, \kappa)$, i.e. that κ has a path from the stack frame to mnt which involves any fields.
3. $\lceil \mathsf{mnt} \rfloor_{\kappa} \in \mathcal{A}cc\mathcal{P}ub(\kappa, \mathsf{code})$, i.e. that κ has a path from the stack frame to mnt which involves only public fields, or private fields from the same class as the current receiver.

This means there are two further ways in which **Pol_2** may be understood:

$$
\begin{array}{c}
\mathsf{M} \models \textbf{Pol\_2B} \\
\text{iff} \\
\forall\ \mathsf{M}'. \forall (\kappa, \mathsf{code}) \in \mathcal{A}rising(\mathsf{M} * \mathsf{M}').\ \forall \mathsf{mnt} :_{\kappa} \mathsf{Mint}. \\
\mathsf{M} * \mathsf{M}', \kappa, \mathsf{code} \rightsquigarrow \kappa', \mathsf{v}
\end{array}
$$

$$\begin{array}{c} \wedge \ \lceil \mathsf{Currency}(\mathsf{mnt}) \rfloor_{\kappa} \neq \lceil \mathsf{Currency}(\mathsf{mnt}) \rfloor_{\kappa'} \\ \Longrightarrow \\ \lceil \mathsf{mnt} \rfloor_{\kappa} \in \mathcal{AccAll}(\mathsf{M} * \mathsf{M}', \kappa) \end{array}$$

$$\begin{array}{c} \mathsf{M} \models \textbf{Pol\_2C} \\ \text{iff} \\ \forall\, \mathsf{M}'.\ \forall(\kappa, \mathsf{code}) \in \mathcal{Arising}(\mathsf{M} * \mathsf{M}').\ \ \forall \mathsf{mnt} :_{\kappa} \mathsf{Mint}. \\ \mathsf{M} * \mathsf{M}', \kappa, \mathsf{code} \rightsquigarrow \kappa', \mathsf{v} \\ \wedge \ \lceil \mathsf{Currency}(\mathsf{mnt}) \rfloor_{\kappa} \neq \lceil \mathsf{Currency}_{\kappa'}(\mathsf{mnt}) \rfloor_{\kappa} \\ \Longrightarrow \\ \lceil \mathsf{mnt} \rfloor_{\kappa} \in \mathcal{AccPub}(\mathsf{M} * \mathsf{M}', \kappa) \end{array}$$

Our interpretation of **Pol_2** in section 4.2 uses the first choice. In [8], we prove that $\mathsf{M}_{\mathsf{Mint}} * \mathsf{M}_{\mathsf{Purse}} \models$ **Pol_2A**. Moreover, we prove lemmas which guarantee that $\mathsf{mnt} :_{\kappa} \mathsf{Mint}$ and $\lceil mnt \rfloor_{\kappa} \in \mathcal{Used}(\mathsf{M}, \kappa, \mathsf{code})$ imply that $\lceil mnt \rfloor_{\kappa} \in \mathcal{AccAll}(\mathsf{M}, \kappa)$. Ttherefore any code which satisfies **Pol_2A** also adheres to **Pol_2B**. This gives that $\mathsf{M}_{\mathsf{Mint}} * \mathsf{M}_{\mathsf{Purse}} \models$ **Pol_2A**.

What about **Pol_2C**? It gives a stronger guarantee than **Pol_2B**, and therefore is to be preferred over **Pol_2B**, however, $M_{\mathsf{Purse}} * M_{\mathsf{Mint}}$ does *not* satisfy **Pol_2C**. More importantly, without the concept of package and package-local classes, or some concept of ownership, it is impossible to write an implementation for Purse so that it satisfies **Pol_2C**. The following example shows why:

```
class CentralBank {
    private final Mint myMint = new Mint();

    public void inflate() {
          Purse tmpPurse = new Purse(myMint,1000000000)
     }
 }
```

A `CentralBank` has a mint. The `inflate` method creates a new temporary purse containing a billion dollars from thin air — perhaps this method should have been called `quantitativeEasing`. Now consider a client of a `CentralBank` object — the finance minister say. The finance minister does *not* have a public access to the mint (because the central bank is supposed to be independent!) so by **Pol_2C** she should not be able to inflate the currency. If, however, the finance minister calls `myCentralBank .inflate` then the currency will be inflated all the same.

Discussion We leave the question as to the "correct" meaning of the policies open. Our contribution is the provision of the tools with which to give precise meaning to policies, and the clarification of the differences. However, the "correct" meaning is determined by the use of the policies in the wider setting, for example, in the application of the policies to prove properties of the use of the mint/purse system, e.g. in the escrow example [21].

7 Related Work

Object capabilities were first introduced [22] seven years ago, and many recent studies manage or verify safety or correctness of object capability programs.

Google's Caja [25] applies sandboxes, proxies, and wrappers to limit components' access to *ambient* capabilities. Sandboxing has been validated formally: Maffeis et al. [16] develop a model of JavaScript, demonstrate that it obeys two principles of object capability systems and show how untrusted applications can be prevented from interfering with the rest of the system. Alternatively, Taly et al. [35] model JavaScript APIs in Datalog, and then carry out a Datalog search for an "attacker" from the set of all valid API calls. This search is similar to the quantification over potential code snippets in our model. Murray and Lowe [26] model object capability programs in CSP, and use a model checker to ensure program executions do not leak information.

Karim et al. apply static analysis on Mozilla's JavaScript Jetpack extension framework [12], including pointer analyses. Bhargavan et al. [3] extend language-based sandboxing techniques to support "defensive" components that can execute successfully in otherwise untrusted environments. Meredith et al. [18] encode policies as types in higher order reflective π-calculus.. Politz et al. [29] use a JavaScript typechecker to check properties such as *"multiple widgets on the same page cannot communicate."* — somewhat similar in spirit to our **Pol_4**. Lerner et al. extend this system to ensure browser extensions observe *"private mode"* browsing conventions, such as that *"no private browsing history retained"* [15]. Dimoulas et al. [6] generalise the language and typechecker based approach to enforce explicit policies, that describe which components may access, or may influence the use of, particular capabilities.

The WebSand [4, 17] and Jeeves [40] projects use dynamic techniques to monitor safe execution of information flow policies. Richards et al. [31] extended this approach by incorporating explicit dynamic ownership of objects (and thus of capabilities) and policies that may examine the history of objects' computations. While these dynamic techniques can restrict or terminate the execution of a component that breaches its security policies, they cannot guarantee in advance that such violations can never happen. While information flow policies are concerned with the flow of objects (and thus also capabilities) across the program code, our work is more concerned with the identification of the objects which protect the services.

A few formal verification frameworks address JavaScript's highly dynamic, prototype-based semantics. Gardner et al. [9] developed a formalisation of JavaScript based on separation logic and verified examples. Xiong and Qin et al. [39, 30] worked on similar lines. Swamy et al. [34] recently developed a mechanised verification technique for JavaScript based on the Dijkstra Monad in the F* programming language. Finally, Jang et al. [11] developed a machine-checked proof of five important properties of a web browser — again similar to our simple deny policies — such as *"cookies may not be shared across domains"* by writing the minimal kernel of the browser in Coq.

8 Conclusions and Future Work

In this paper, we have advocated that capability policies are necessary for reasoning about programs using object capability security. We have argued that capability policies are program centred, fine grained, open, and contain necessary as well as sufficient conditions.

These novel features of the policies require novel features in specifications. We have proposed execution observations, and developed a capability specification style, which incorporates universal and existential quantification over program code, explicit naming of snapshots before, after and during execution, and their use in premises and in conclusions. We have used our approach to specify most of the Mint example.

We have shown how efforts at specifying policies precisely can uncover ambiguities in policies' interpretations, and can help find additional implicit policies that can be made explicit. We have proposed another five policies for the Mint, and formulated then in our language.

In further work, we want to refine the execution observations, to develop a programmer-friendly notation for specifications, to consider the specification of the further policies we uncovered as well as other policies from the literature, and to extend our toy language to encompass further salient programming language features. We also want to develop a formal logic to support reasoning about code's adherence to capability policies. We need to model trust (or the lack of it) between components, so we can model systems composed both trusted and untrusted code. Finally, inspired by the original Mint work, we want to consider the specification and verification of capabilities in other programming languages, in particular, languages without static types.

Acknowledgments

This work is partly supported by the Royal Society of New Zealand Marsden Fund and by the EU FP7 project Upscale.

References

1. Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In *ACSAC*, 2012.
2. Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with Polymer. In *PLDI*, 2005.
3. Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Language-based defenses against untrusted browser origins. In *USENIX Security*, 2013.
4. Arnar Birgisson, Alejandro Russo, and Andrei Sabelfeld. Capabilities for information flow. In *Programming Languages and Analysis for Security (PLAS)*, 2011.
5. Jack B. Dennis and Earl C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Comm. ACM*, 9(3), 1966.

6. Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. Declarative policies for capability control. In *Computer Security Foundations Symposium (CSF)*, 2014.
7. Sophia Drossopoulou and James Noble. The need for capability policies. In *(FTfJP)*, 2013.
8. Sophia Drossopoulou and James Noble. Towards Capability Policy Specification and Verification. Technical Report ECSTR-14-05, School of Engineering and Computer Science, Victoria University of Wellington, 2014.
9. Philippa Gardner, Sergio Maffeis, and Gareth David Smith. Towards a program logic for JavaScript. In *POPL*, 2012.
10. C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.
11. Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing browser security guarantees through formal shim verification. In *USENIX Security*, 2012.
12. Rezwana Karim, Mohan Dhawan, Vinod Ganapathy, and Chung-Chieh Shan. An Analysis of the Mozilla Jetpack Extension FrameworK. In *ECOOP*, Springer, 2012.
13. Butler W. Lampson and Howard E. Sturgis. Reflection on an Operating System Design. *Communications of the ACM*, 19(5), 1976.
14. K. Rustan M. Leino and Wolfram Schulte. Using history invariants to verify observers. In *ESOP*, 2007.
15. Benjamin S. Lerner, Liam Elberty, Neal Poole, and Shriram Krishnamurthi. Verifying web browser extensions' compliance with private-browsing mode. In *European Symposium on Research in Computer Security (ESORICS)*, September 2013.
16. S. Maffeis, J.C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *Proc of IEEE Security and Privacy*, 2010.
17. Jonas Magazinius, Alejandro Russo, and Andrei Sabelfeld. On-the-fly inlining of dynamic security monitors. *Computers & Security*, 31(7):827–843, 2012.
18. L.G. Meredith, Mike Stay, and Sophia Drossopoulou. Policy as types. arXiv:1307.7766 [cs.CR], July 2013.
19. Adrian Mettler, David Wagner, and Tyler Close. Joe-E a Security-Oriented Subset of Java. In *NDSS*, 2010.
20. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.
21. Mark S. Miller, Tom Van Cutsem, and Bill Tulloh. Distributed electronic rights in JavaScript. In *ESOP*, 2013.
22. Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Baltimore, Maryland, 2006.
23. Mark Samuel Miller. Secure Distributed Programming with Object-capabilities in JavaScript. Talk at Vrije Universiteit Brussel, mobicrant-talks.eventbrite.com, October 2011.
24. Mark Samuel Miller, Chip Morningstar, and Bill Frantz. Capability-based Financial Instruments: From Object to Capabilities. In *Financial Cryptography*. Springer, 2000.
25. Mark Samuel Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Safe active content in sanitized JavaScript, 2008. code.google.com/p/google-caja/.
26. Toby Murray and Gavin Lowe. Analysing the information flow properties of object-capability patterns. In *FAST*, LNCS, 2010.
27. James Noble and Sophia Drossopoulou. A rational reconstruction of the escrow example. In *(FTfJP)*, 2014.
28. Matthew Parkinson. Class invariants: the end of the road? In *IWACO*, 2007.
29. Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. ADsafety: Type-based verification of JavaScript sandboxing. In *USENIX Security*, 2011.
30. Shengchao Qin, Aziem Chawdhary, Wei Xiong, Malcolm Munro, Zongyan Qiu, and Huibiao Zhu. Towards an axiomatic verification system for JavaScript. In *TASE*, pages 133–141, 2011.

31. Gregor Richards, Christian Hammer, Francesco Zappa Nardelli, Suresh Jagannathan, and Jan Vitek. Flexible access control for JavaScript. In *OOPSLA*, pages 305–322, 2013.
32. Marc Stiegler. The lazy programmer's guide to security. HP Labs, www.object-oriented-security.org.
33. Alexander J. Summers, Sophia Drossopoulou, and Peter Müller. The need for Flexible Object Invariants. In *IWACO*, ACM DL, July 2009.
34. Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the dijkstra monad. In *PLDI*, pages 387–398, 2013.
35. Ankur Taly, Ulfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. Automated Analysis of Security-Critical JavaScript APIs. In *IEEE Symposium on Security and Privacy (SP)*, 2011.
36. The Newspeak Team. Several Newspeak Documents. `newspeaklanguage.org/`, September 2012.
37. Tom van Cutsem. Membranes in JavaScript, 2012. `prog.vub.ac.be/˜tvcutsem/-invokedynamic/js-membranes`.
38. M. V. Wilkes and R. M. Needham. The Cambridge CAP computer and its operating system, 1979.
39. Wei Xiong. *Verification and Validation of JavaScript*. PhD thesis, 2013, Durham University.
40. Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *POPL*, 2012.