Analysing Object-Capability Security

Toby Murray
Oxford University Computing Laboratory

Cooperation and Vulnerability

Much of the power and utility of modern computing arises in the different forms of cooperation it enables.

Currently, this power comes with great risk because those engaged in cooperation are left vulnerable to one another.

- Desktop PCs can run arbitrary software.
  - But running arbitrary code can ruin your life.
- Software can be built by composing independent components.
  - But any component can render the entire product faulty or malicious by its inclusion.

Modern software architectures must, therefore, enable new and better forms of cooperation without vulnerability.

A Promising Remedy

The Object-Capability (OCap) Model is an architecture that shows significant promise:

- OCap Operating Systems (OSs), such as EROS and seL4, enable users to run arbitrary code whilst remaining safe from its misbehaviour.
- OCap languages, such as E and Caja, enable code to be composed from arbitrary sources whilst ensuring that malicious code cannot harm the user or the rest of the application.

An Object-Capability (OCap) System

Comprises a collection of objects connected by capabilities.

Object An encapsulated, protected entity comprising code and mutable state. State includes both data and capabilities.

Capability An unforgeable object reference. Allows the holder to send a message to the referenced object by invoking it.

Message-sending is the only means by which objects can interact.

Object o can pass a capability, c, directly to object p only by sending a message to p that contains c.

Capabilities can only be passed between already connected objects.

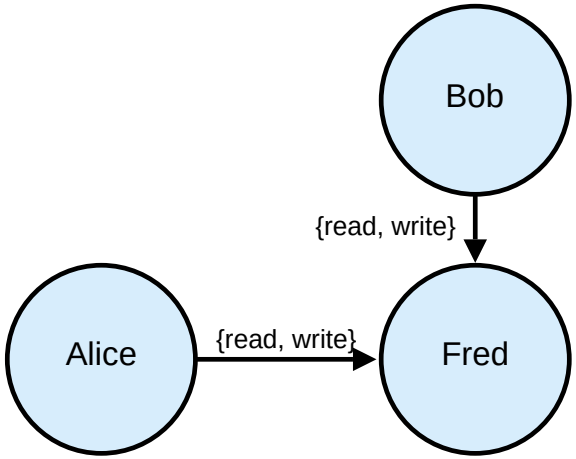
Objects may create new ones. Parent must supply the code of its children and any capabilities they are to initially possess.

Security-Enforcing Abstractions

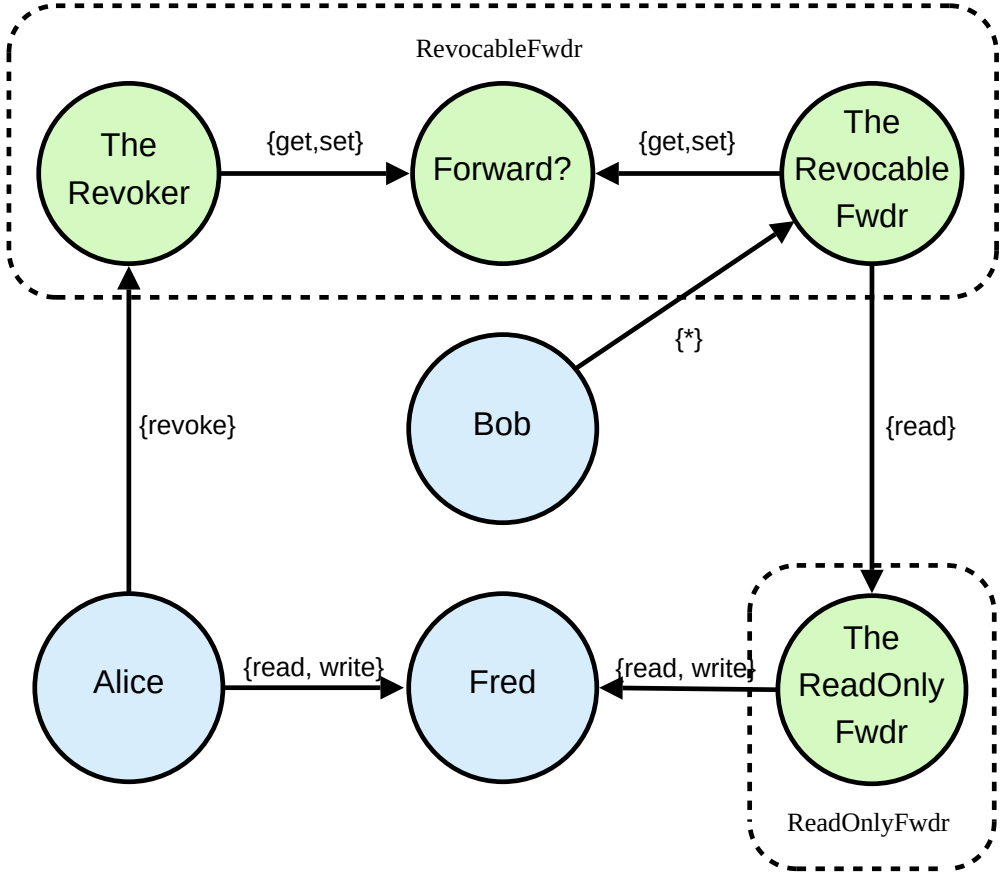
The OCap model is powerful because it allows programmers to create security-enforcing abstractions, or patterns, that can be composed with other code to enable cooperation whilst minimising vulnerability.

Example Alice, an architect, needs to cooperate with Bob, a builder. She needs to send Bob blueprints via the file Fred.

A naive solution gives both Alice and Bob read-write access to Fred.



A better solution employs two patterns: the ***ReadOnlyForwarder*** and the ***RevocableForwarder***.



Common Patterns

Some patterns have consistently reappeared in different systems:

- ***RevocableForwarders*** and ***AttenuatingForwarders***, such as the ***ReadOnlyForwarder*** above, (in *e.g.* E, KeyKOS and Emily),
- ***Sealer-Unsealers*** (in *e.g.* E, KeyKOS, Emily and Caja) and
- ***Membranes*** (in *e.g.* E, KeyKOS, DCCS and Emily).

Their wide use necessitates formal verification.

OCap systems differ widely, particularly in terms of concurrency:

- OCap languages like Caja are single-threaded.
- In OCap OSs like seL4, all processes execute concurrently.

Patterns that work correctly in one context can be faulty in the other.

Modelling OCap Patterns in CSP

We build a system for each context that we want to consider the pattern in.

We then represent the pattern's security properties as CSP refinement checks. Here we consider only properties that can be expressed as simple trace refinements of the form $Spec \sqsubseteq_T System$.

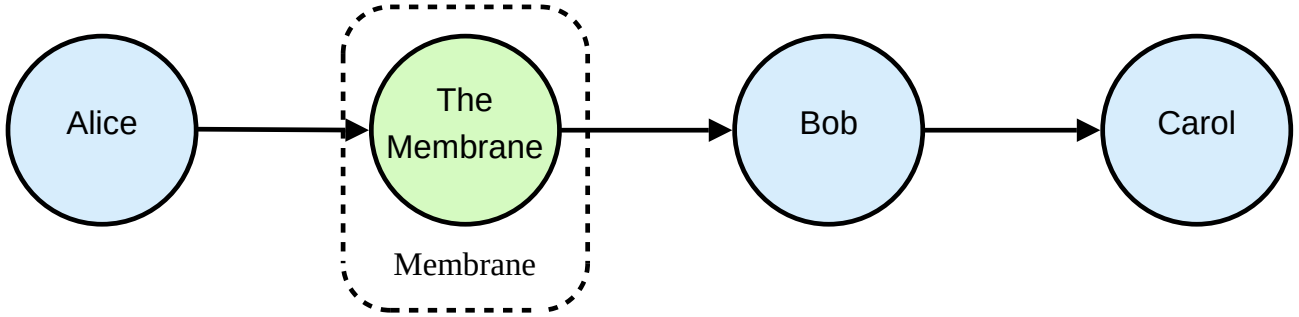
Despite their simplicity, these enable us to reason effectively about properties, such as revocation, that are beyond the reach of previous formalisms.

So long as the system is finite-state, these checks can be automatically carried out using the CSP refinement checker, FDR.

We can compare a pattern's behaviour in different contexts by comparing the results of applying the same refinement tests to each of the systems.

A Modelling Example

Example The ***Membrane*** pattern. Allows a policy to be applied to all capabilities reachable from a particular capability.

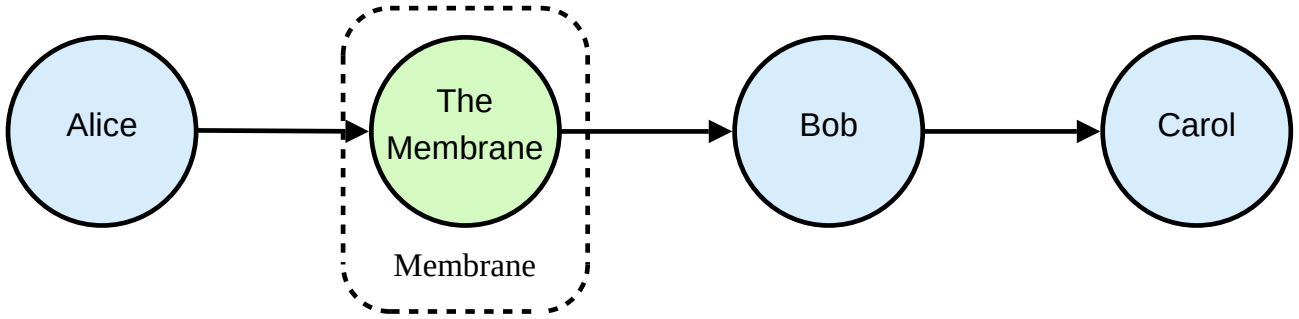


TheMembrane wraps Alice's capability to Bob and may enforce some policy, such as restricting the methods she is allowed to call.

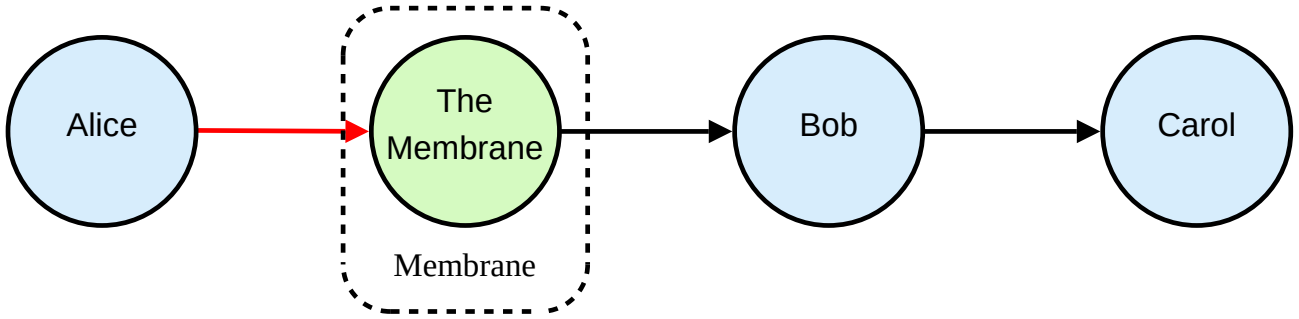
TheMembrane acts as a proxy between Alice and Bob, wrapping all capabilities passed in either direction.

Wrapping a capability, c, involves creating a child membrane object that acts as a proxy for c.

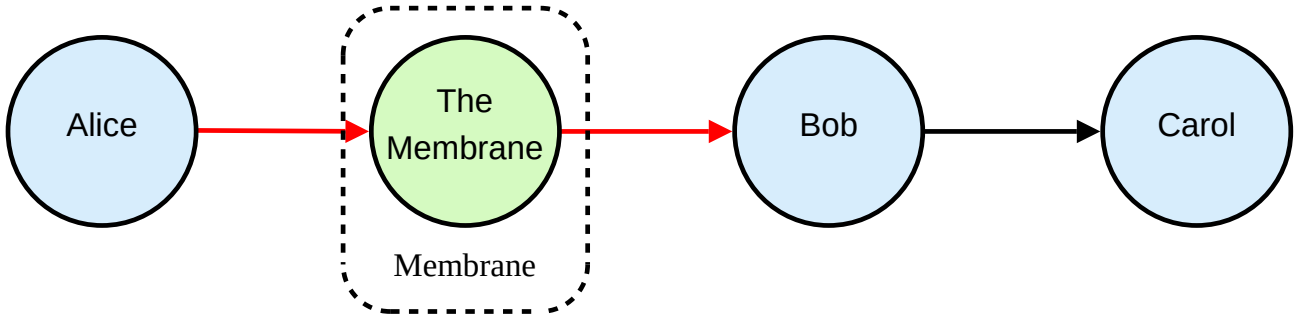
Membranes in Action



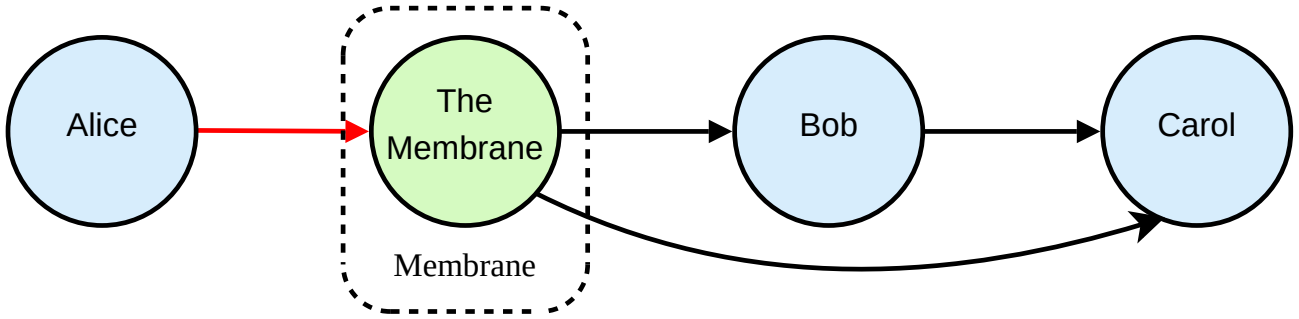
Membranes in Action



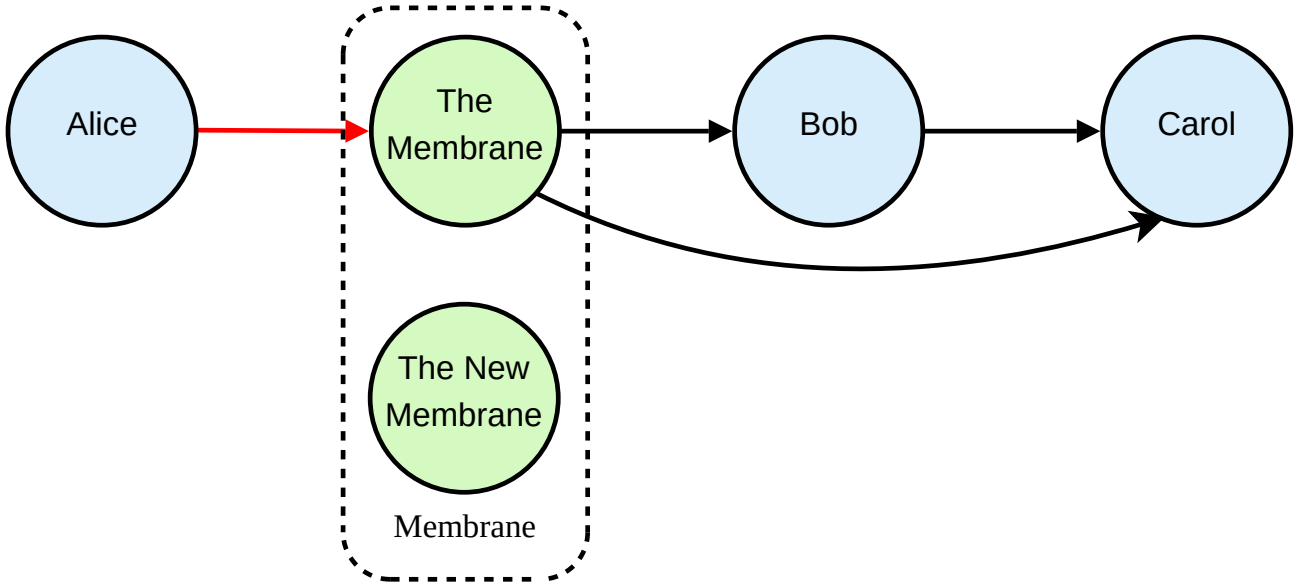
Membranes in Action



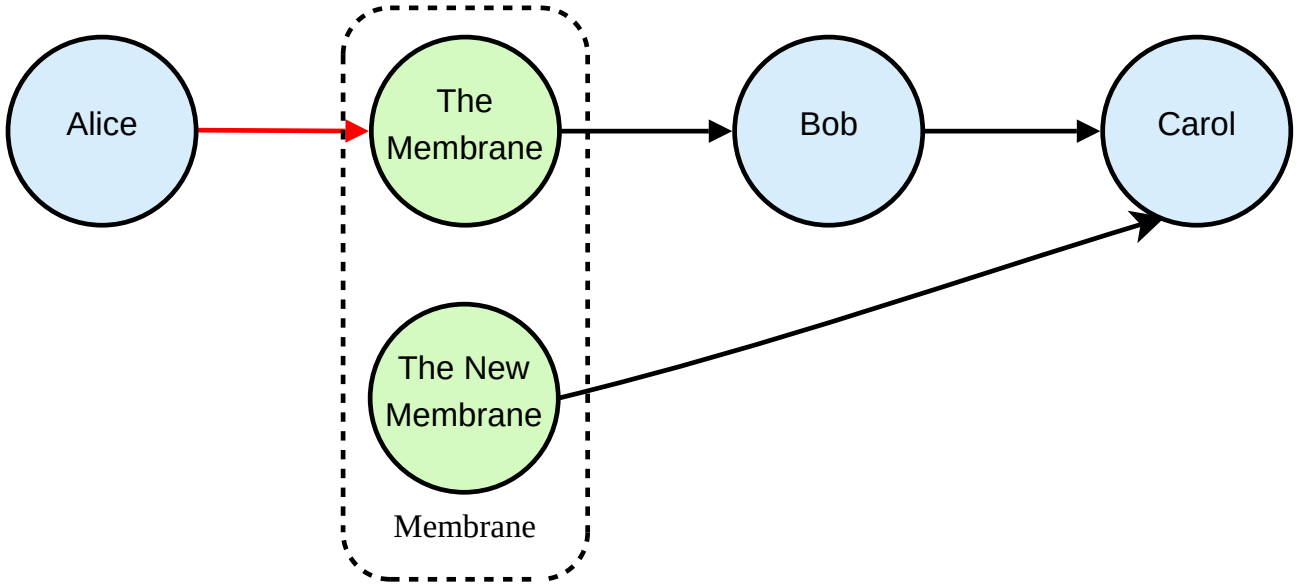
Membranes in Action



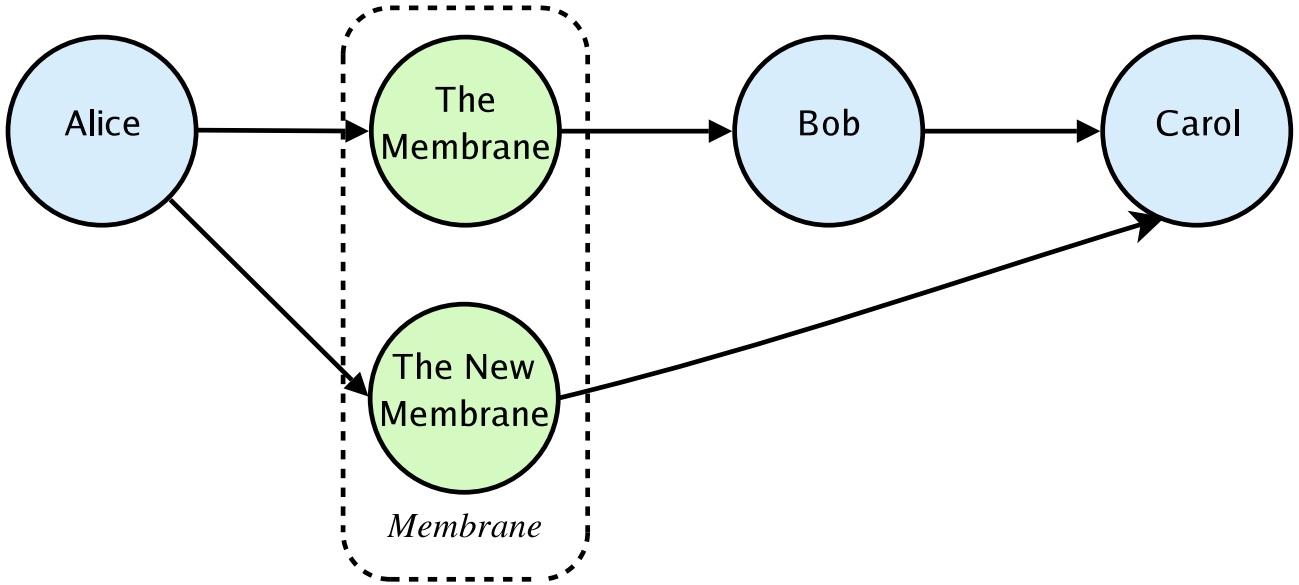
Membranes in Action



Membranes in Action



Membranes in Action



Modelling Patterns in CSP

We consider a small system that comprises an instance of the pattern composed with some other objects that exhibit maximum possible behaviour.

$Object$ is the set of objects in the system.

Example $Object = \{\mathsf{Alice}, \mathsf{TheMembrane}, \mathsf{Bob}, \mathsf{Carol}\}$

We use events of the form $o_1.o_2.op.arg$, to represent the sending and receipt of a message from object $o_1 \in Object$ to object $o_2 \in Object$ specifying the operation $op \in Op$ and containing the argument $arg \in Object \cup \{\mathsf{null}\}$.

For simplicity, $Op = \{\mathsf{Call}, \mathsf{Return}\}$. These represent an object invocation and return in an OCap language or an inter-process send and reply in an OCap OS.

The alphabet of each object, o: $\alpha(o) = \{|o.o', o'.o \mid o' \in Object - \{o\}|\}$.

Modelling Object Behaviours

We define a process for each object in the system.

The system is then formed as the alphabetised parallel composition of these processes, with their corresponding alphabets.

Modelling a Membrane

Example To keep the system finite-state, a membrane aggregates its own behaviour as well as that of all of its children.

When wrapping a capability, c, the membrane returns a capability to itself, rather than to a new child membrane object.

Next time it is invoked, it offers the choice of forwarding the invocation to c, as well as to those objects it was wrapping previously.

Modelling Objects Exhibiting Maximum Possible Behaviour

Example Objects such as Alice, Bob and Carol from the ***Membrane*** example.

Representation differs between OCap OSs and languages, since each kind of system places different constraints on the allowed behaviour of objects within it.

Maximum Behaviour in an OCap OS

An untrusted object in an OCap OS may invoke any of its capabilities, passing any argument it has access to at any time. It need not wait for a response before invoking another capability.

May also choose to block waiting for an invocation, at which point it receives the capability contained in the invocation as well as a capability to the sender which can be used later to send back a reply.

Maximum Behaviour in an OCap Language

Active An object in an OCap language is active precisely when one of its methods is currently being executed and is inactive otherwise.

Only one object is ever active at a time.

An object that is inactive waits to be invoked, at which point it obtains whatever capability may have been passed with the invocation, as well as a capability to the invoker that it can use later to respond to the invocation.

Once invoked, the object becomes active.

An object that is active can choose to invoke any of its capabilities and pass any argument it has access to. After doing so, it becomes inactive.

Note that this model is very permissive and allows behaviours that would be impossible in languages like Caja that enforce strict call-return semantics.

We choose not to model strict call-return semantics as this would involve modelling the call-stack. This is not only tedious but also prevents our systems from being finite-state without imposing artificially low limits on the maximum size of the stack.

If a pattern is "broken" by an impossible behaviour, we can further restrict the model to disallow the impossible behaviour and repeat the test.

Analysing the *Membrane* pattern

We instantiate two systems, $MSystem_{OS}$ and $MSystem_{lang}$, to represent this pattern in the contexts of an OCap OS and an OCap language, respectively.

In both cases, Alice, Bob and Carol are each initially given the capabilities {Alice, TheMembrane}, {Bob, Carol} and {Carol}, respectively.

TheMembrane is initially given a capabilities to itself and to Bob, and Bob is set as its initial target.

Alice is the object that is initially active $MSystem_{lang}$.

One obvious property to test is whether Alice can obtain a capability to Bob or Carol.

If this occurs, then the system will perform an event from $A = \{|\mathsf{Alice.Bob}, \mathsf{Alice.Carol}|\}$.

A system, $System$, performs no such event if

$$CHAOS_{\Sigma - A} \sqsubseteq_T System.$$

FDR indicates that this property holds for both $MSystem_{OS}$ and $MSystem_{lang}$.

The *RevocableMembrane* pattern

Extends the ***Membrane*** pattern by incorporating the logic of the ***RevocableForwarder*** pattern.

Allows all capabilities obtained through the membrane to be revoked. Enforces transitive revocable access.

The membrane is enhanced to hold a capability to a bool object that contains a boolean value. Membrane forwards requests only if its bool contains a true value.

A corresponding revoker object holds a reference to the same bool. When invoked, it sets the value of the bool to false.

A membrane's children use the same bool as their parent. Hence, invoking the revoker causes all capabilities wrapped by the membrane to be revoked.

Analysing the *RevocableMembrane* pattern

We base the instantiation of this pattern on those for the ***Membrane*** pattern.

We instantiate three objects, TheMembrane, TheBool and TheRevoker to represent this pattern.

Bob and Carol are instantiated as before. Alice is instantiated as before in each kind of system, except now she is also given a capability to TheRevoker.

The same properties hold for this pattern in each kind of system as do for the ***Membrane*** presented earlier.

We would also like to verify that it enforces revocation.

Verifying Revocation

Revocation holds if once TheRevoker has been invoked and this invocation has Returned, the TheMembrane can no longer Call Alice, Bob or Carol.

We can test this using a simple trace refinement (see the paper).

FDR reveals that this holds for the language-based system but not for the system modelled in the OS context. It gives the following trace as a counter-example.

⟨Alice.TheMembrane.Call.null, TheMembrane.TheBool.Call.null,
TheBool.TheMembrane.Return.TheBool, Alice.TheRevoker.Call.null,
TheRevoker.TheBool.Call.TheBool, TheBool.TheRevoker.Return.TheBool,
TheRevoker.Alice.Return.null, TheMembrane.Bob.Call.null⟩

TheMembrane checks TheBool, then TheRevoker alters its value and returns, after which TheMembrane Calls Bob since TheBool was true when TheMembrane checked it.

This is an example of a race condition that exhibits itself as a time-of-check-time-of-use (TOCTOU) vulnerability.

It shows that patterns can exhibit subtle differences in behaviour when moved from the language environment into OCap OSs, due to the greater level of concurrency they allow.

The *Sealer-Unsealer* pattern

Used to create two corresponding objects, a sealer and an unsealer.

The sealer can be invoked with a message containing a capability, c, at which point it returns a new "inert" capability c'.

An object that possesses the unsealer can invoke it, passing c', at which point it will return the original capability, c.

Many implementations of this pattern have appeared in both OCap languages, like E and Caja, and OCap OSs like KeyKOS.

Allows one to transport a sensitive capability, c, via an untrusted intermediary in the form of the innocuous capability c'.

We consider a particular implementation developed for OCap languages to verify its properties and see whether they hold when it is moved to the context of OCap OSs.

The *Sealer-Unsealer* implementation

Each sealer and unsealer have access to a common object, called a slot, that can store a single capability.

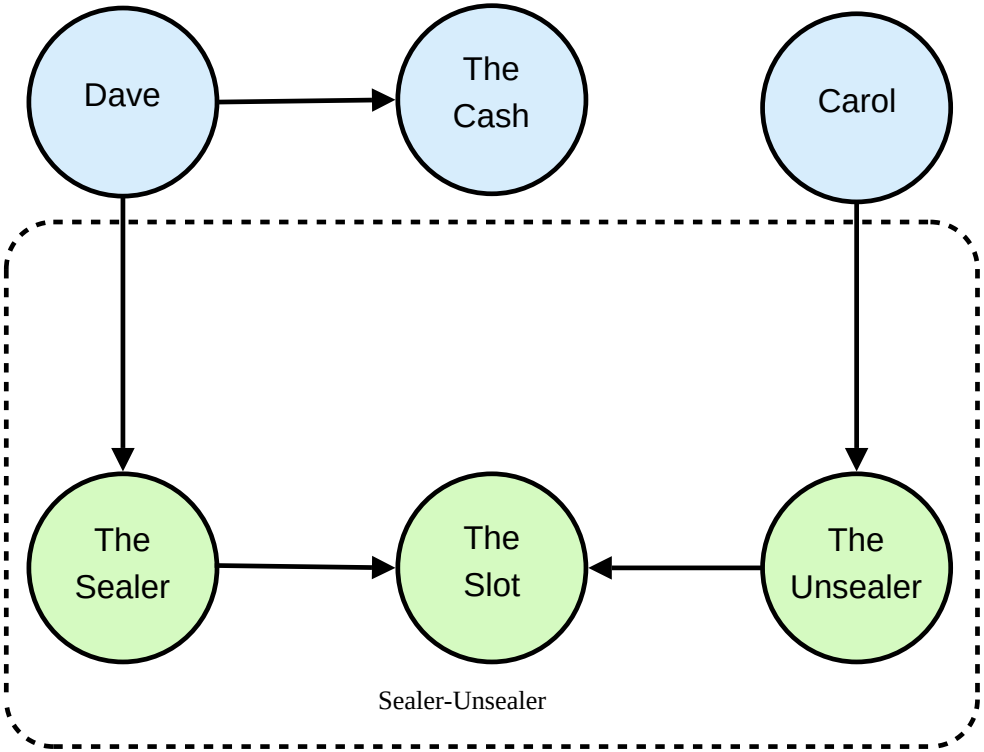
When the sealer is invoked with a capability, c, it creates a new object called a box. It gives the box c and a capability to the slot that is shared with the unsealer.

When invoked, the box places a copy of c into the slot.

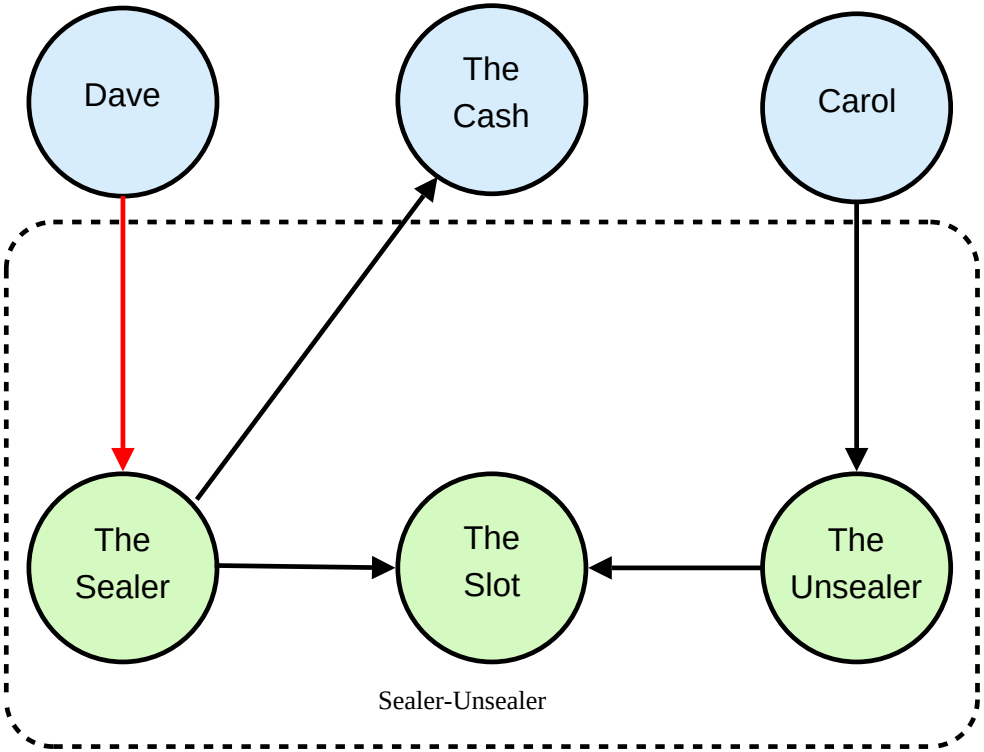
When invoked with a potential box, the unsealer first clears the slot. It then invokes the potential box. If the box was created by the matching sealer, it will have access to the same slot as the unsealer.

Hence, once the invocation of the box returns, the slot should contain c if the box was sealed by the corresponding sealer. If the slot contains a capability, then the unsealer takes a copy of it and clears the slot before returning the capability.

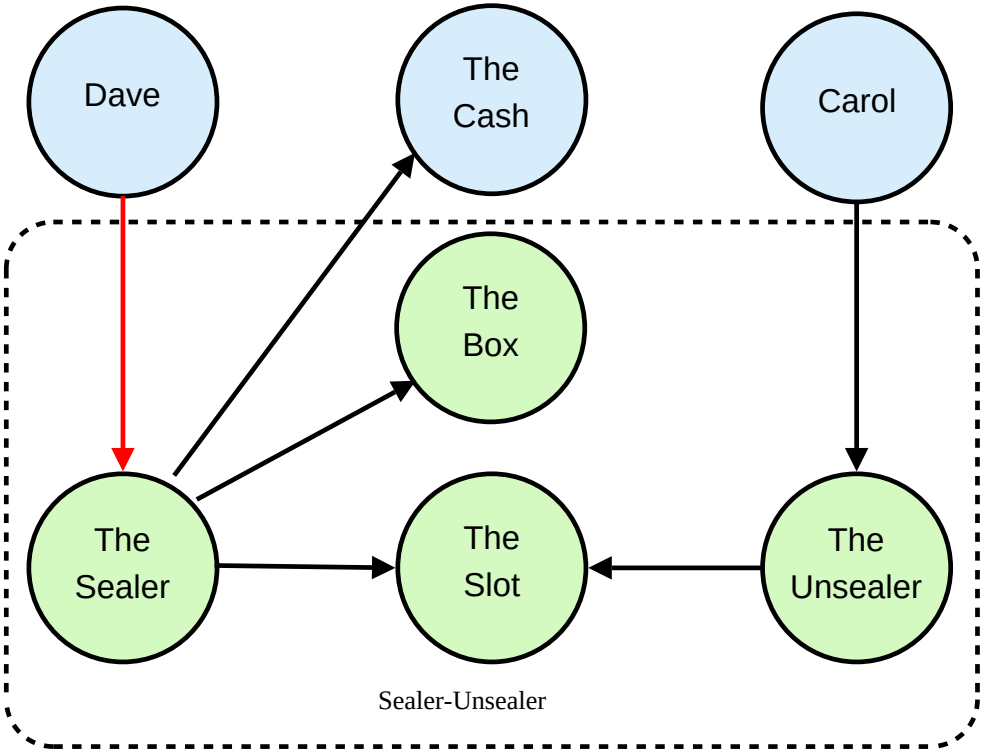
A Sealer-Unsealer in Action



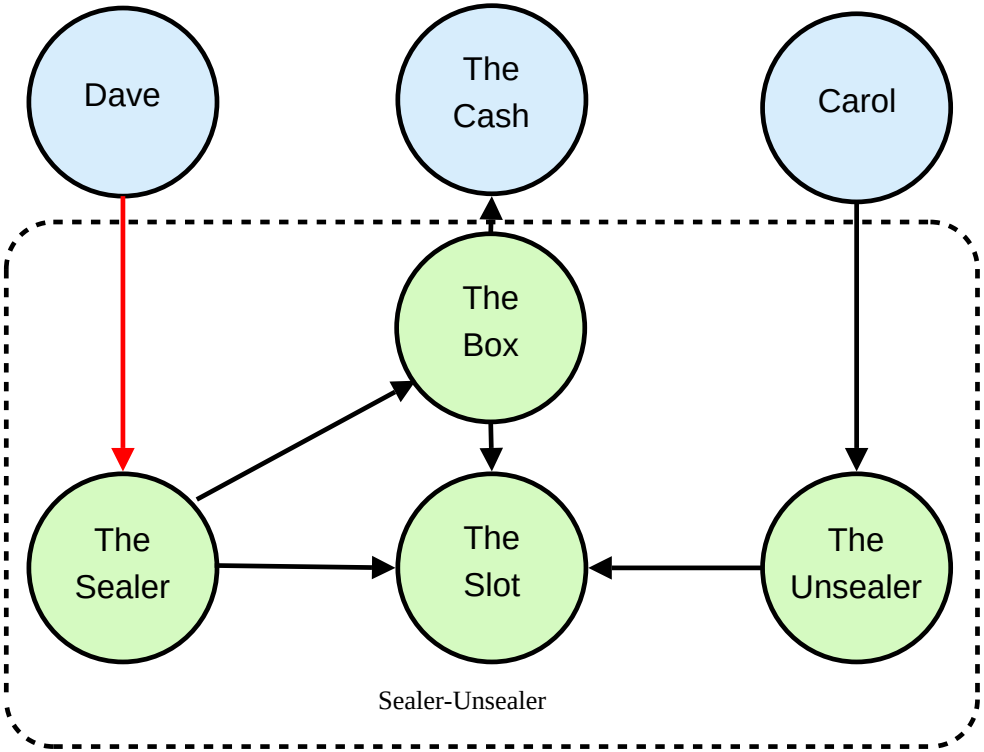
A Sealer-Unsealer in Action



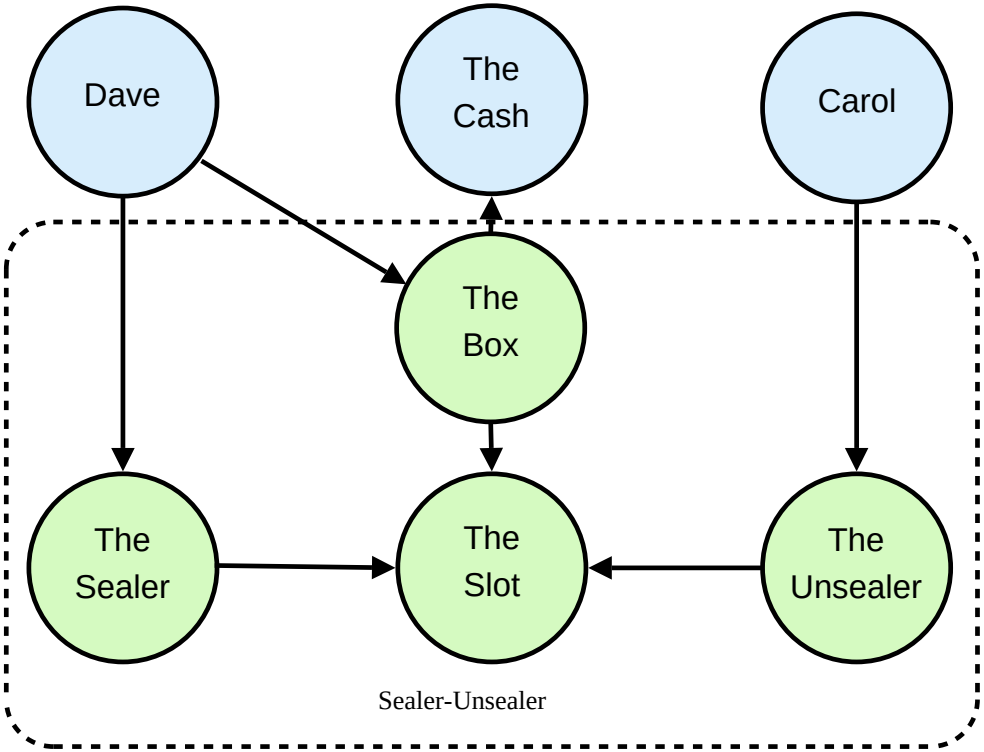
A Sealer-Unsealer in Action



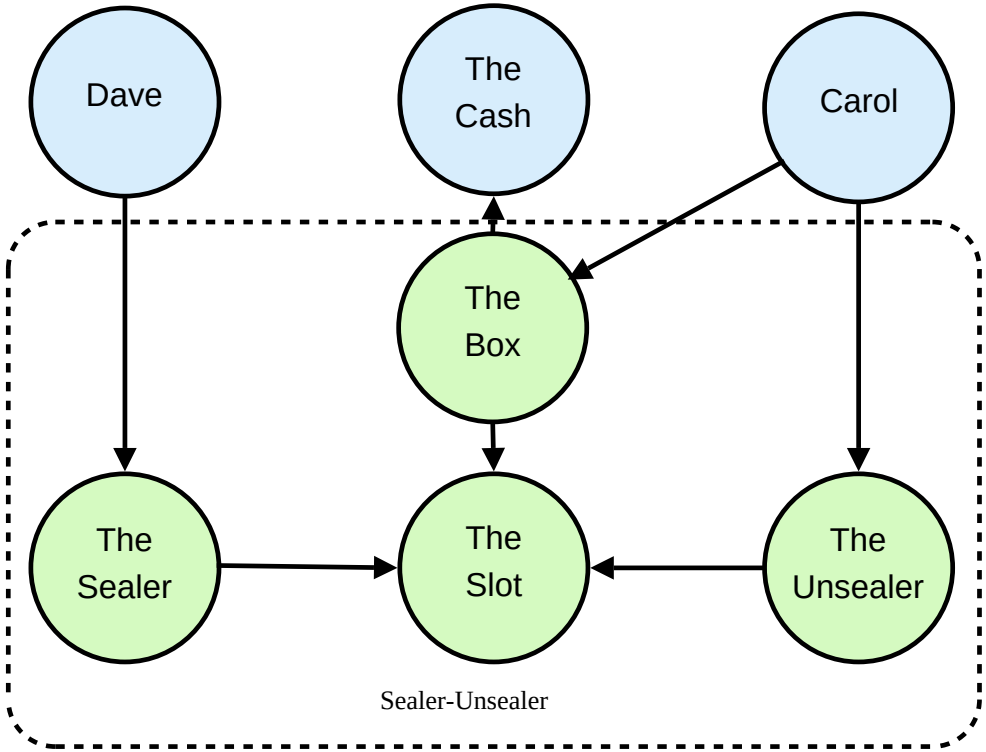
A Sealer-Unsealer in Action



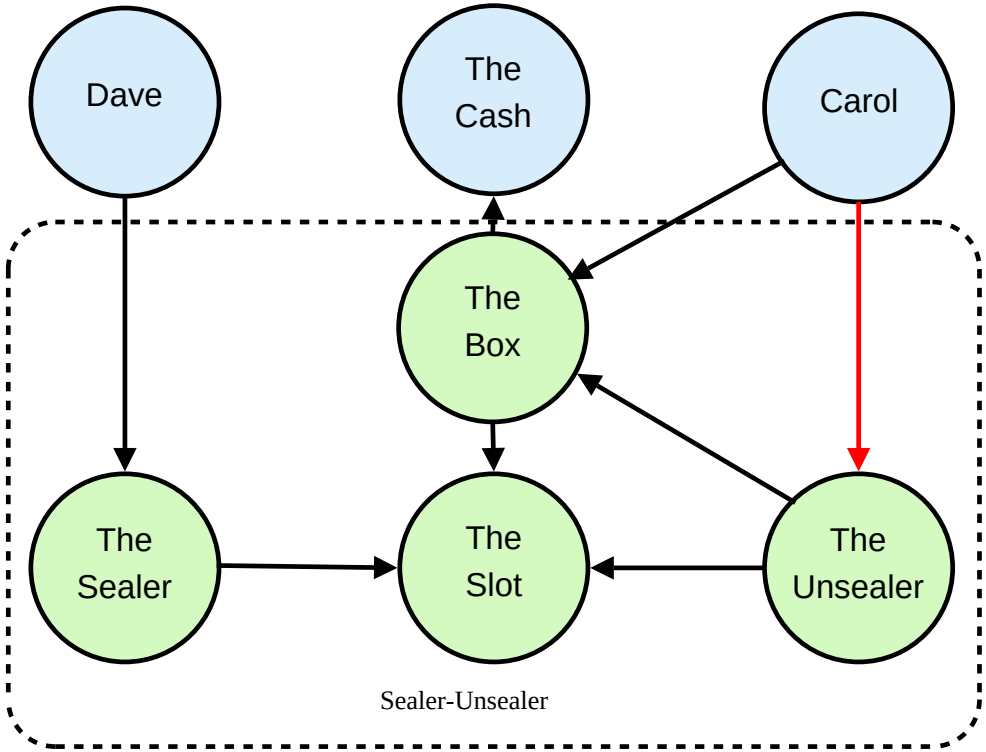
A Sealer-Unsealer in Action



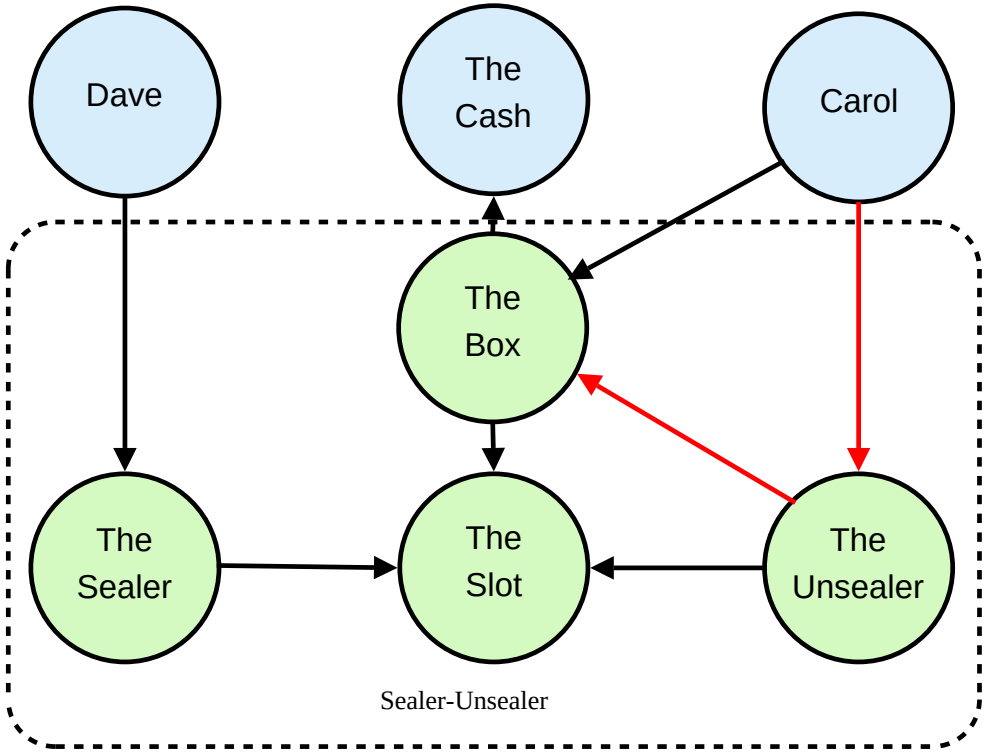
A Sealer-Unsealer in Action



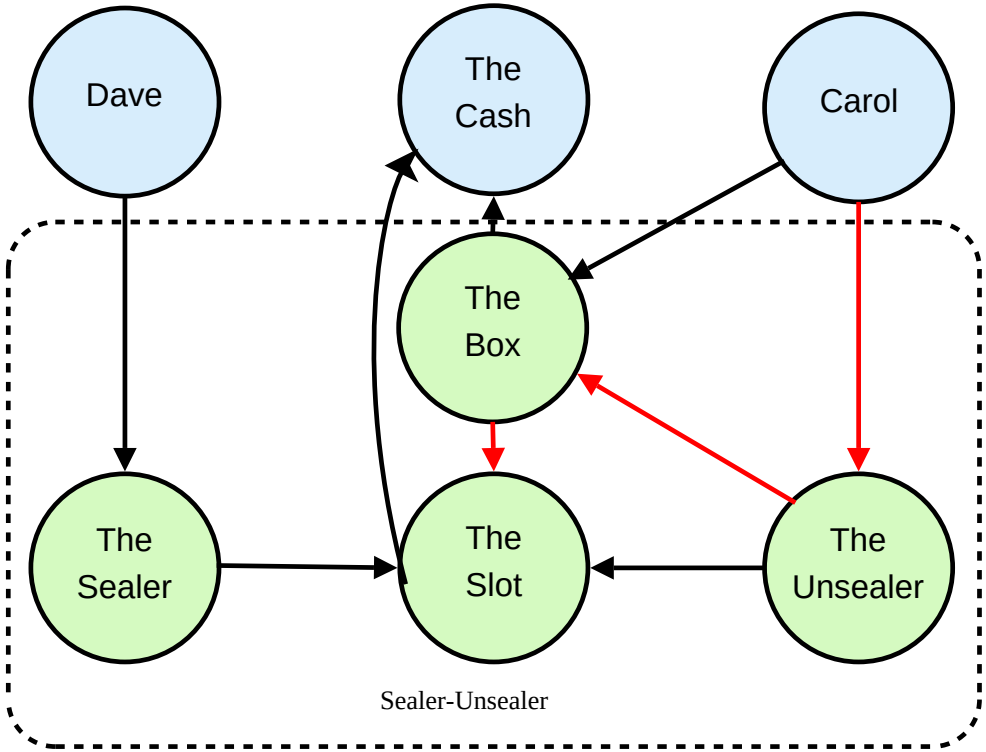
A Sealer-Unsealer in Action



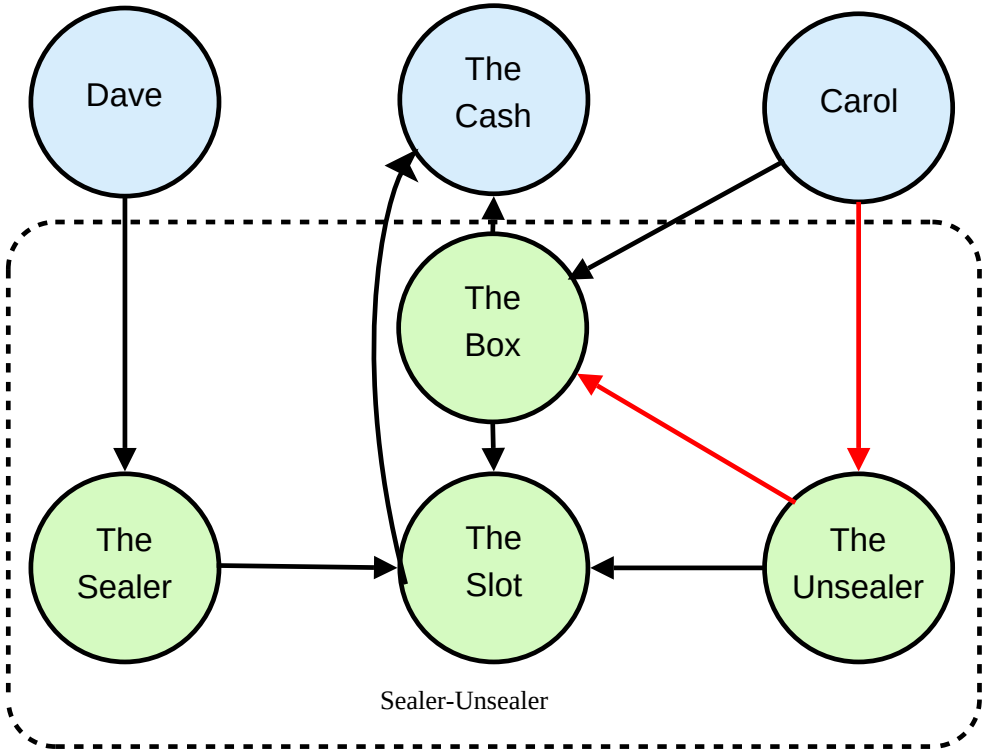
A Sealer-Unsealer in Action



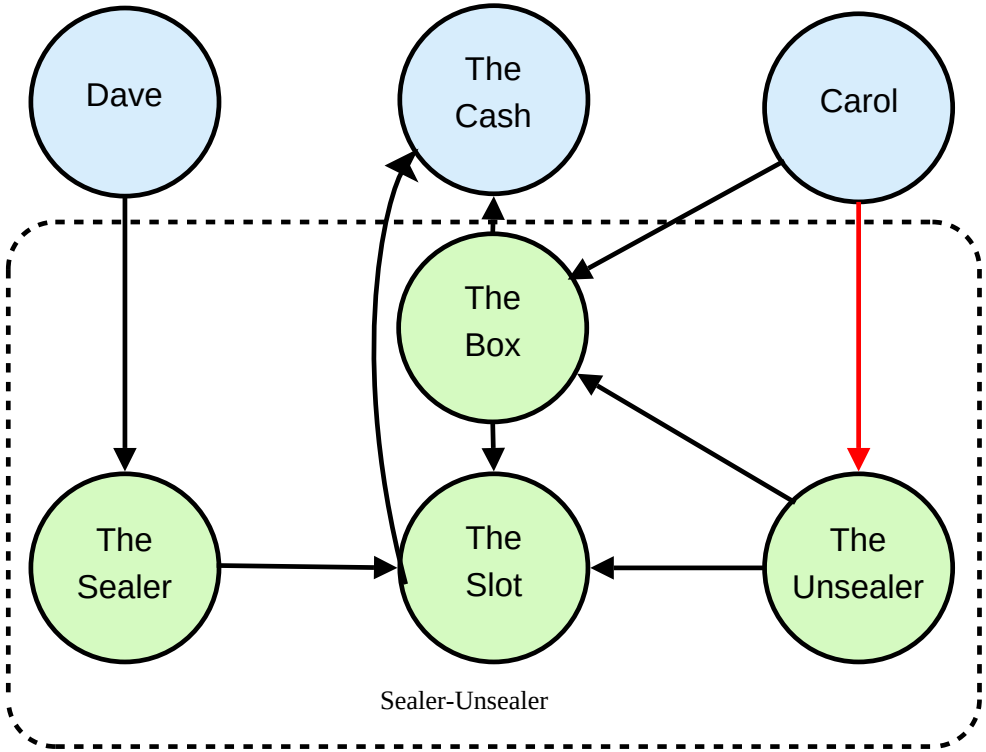
A Sealer-Unsealer in Action



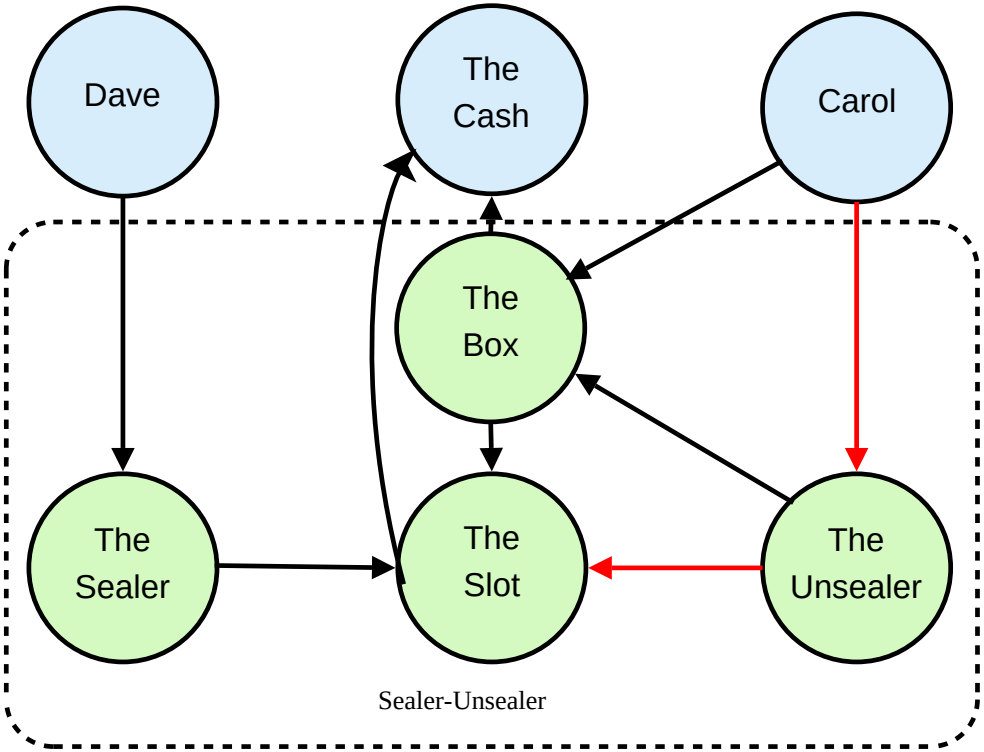
A Sealer-Unsealer in Action



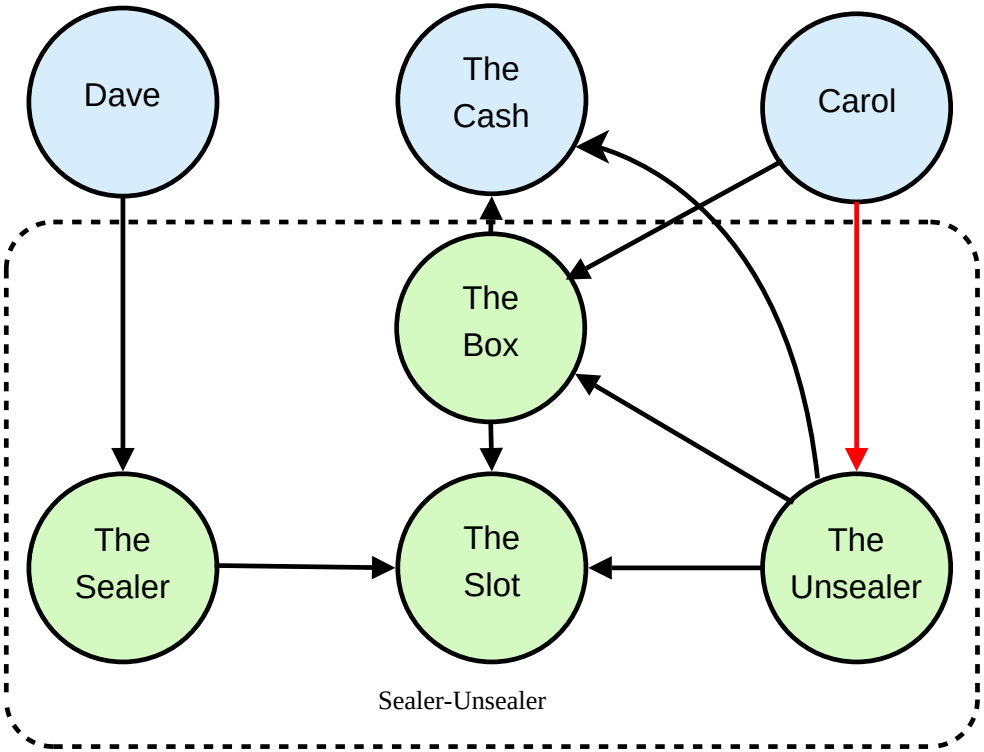
A Sealer-Unsealer in Action



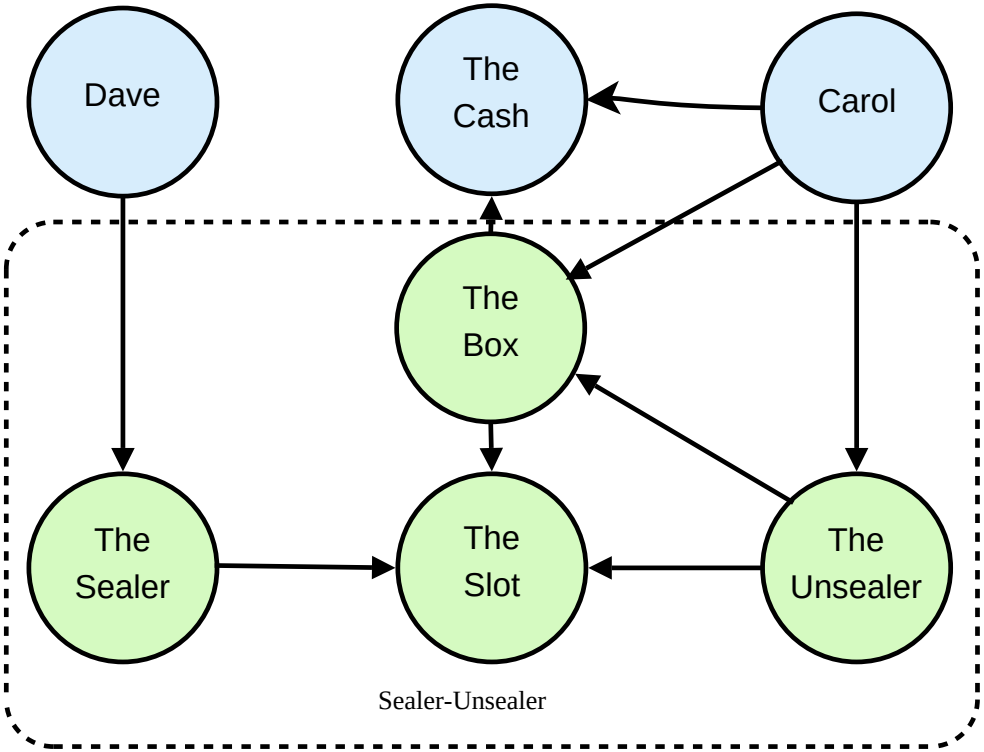
A Sealer-Unsealer in Action



A Sealer-Unsealer in Action

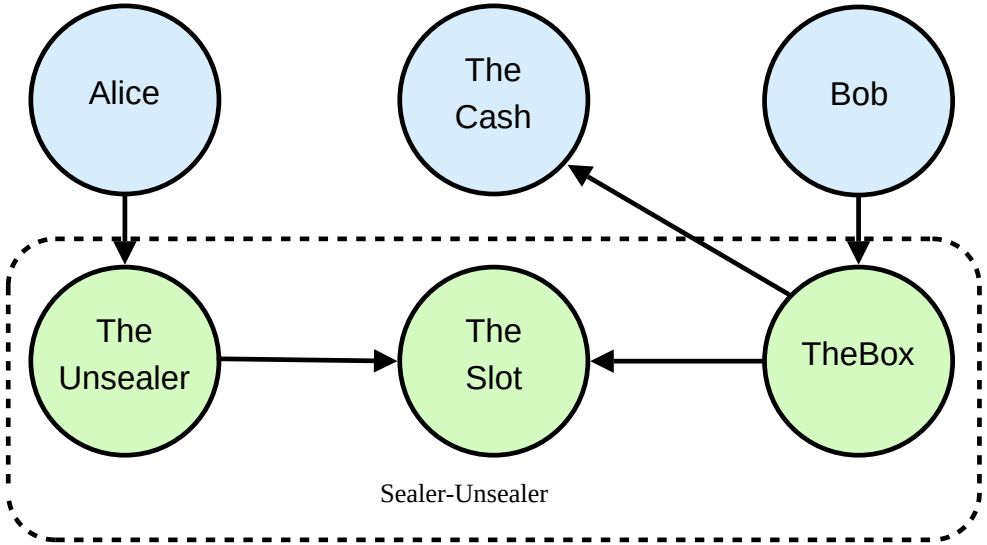


A Sealer-Unsealer in Action



Modelling the *Sealer-Unsealer* implementation

A sealer (not depicted) has been used to seal a capability to a valuable object, TheCash, to produce an innocuous box, TheBox, which has been handed to Bob. Alice has been given a capability to the corresponding unsealer, TheUnsealer. The pattern should prevent both Alice and Bob from obtaining TheCash.



Analysing the *Sealer-Unsealer* pattern

TheUnsealer, TheSlot and TheBox are instantiated as depicted.

The initial capabilities possessed by Alice, TheCash and Bob in both kinds of system are {Alice, TheUnsealer}, {TheCash} and {Bob, TheBox}, respectively.

Each of Alice, TheCash and Bob are instantiated in the language context as inactive. We add an extra driver object that is initially active, that invokes both Alice and Bob without passing capabilities between them.

To test that this pattern enforces its security properties, we simply need to test whether Alice or Bob can obtain TheCash, *i.e.* whether the system can ever perform an event from the set $A = \{|o.\mathsf{TheCash} \mid o \in \{\mathsf{Alice}, \mathsf{Bob}\}|\}$.

Perhaps surprisingly, FDR indicates that this property does not hold for the language case. However, the counter-example returned is an impossible behaviour.

$\langle$TheDriver.Alice.Call.null, Alice.TheUnsealer.Call.Alice,
TheUnsealer.TheSlot.Call.null, TheSlot.TheUnsealer.Return.null,
TheUnsealer.Alice.Call.null, Alice.TheDriver.Return.null, . . .$\rangle$

We restrict the system by placing it in parallel with a process, R, synchronising on all events in Alice's alphabet. R allows all sequences of events from Alice's alphabet except those that contain that above. Repeating the test for the restricted system reveals that the pattern does enforce its security property as expected.

FDR indicates that in the OS context the property does not hold, giving the following counter-example.

⟨Alice.TheUnsealer.Call.Alice, TheUnsealer.TheSlot.Call.null,
TheSlot.TheUnsealer.Return.null, Bob.TheBox.Call.null,
TheBox.TheSlot.Call.TheCash, TheUnsealer.Alice.Call.null,
TheSlot.TheBox.Return.null, Alice.TheUnsealer.Return.null,
TheUnsealer.TheSlot.Call.null, TheSlot.TheUnsealer.Return.TheCash,
TheUnsealer.Alice.Return.TheCash, Alice.TheCash.Return.TheCash⟩

This trace is an obviously valid behaviour.

Alice can obtain TheCash if she invokes TheUnsealer, passing herself, and Bob subsequently chooses to invoke TheBox between when the TheUnsealer clears and checks TheSlot.

Hence, the implementation of this pattern cannot be directly applied in OCap OSs, despite its utility in OCap languages.

Limitations and Future Work

Systems must remain finite-state.

- We cannot explicitly model object creation.

Expressing single-threaded OCap language systems as the composition of concurrently executing processes is clumsy but necessary to enable the proper comparison of patterns between these two contexts.

- Makes it difficult to detect vulnerabilities that arise from recursive invocation since our security-enforcing objects cannot be recursively Called.

Reasoning about asynchronous systems requires further work (*e.g.* introducing buffers etc.).