Analysing Object-Capability Security

Toby Murray

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford, OX1 3QD, United Kingdom
toby.murray@comlab.ox.ac.uk

Abstract. Much of the power and utility of modern computing arises in the different forms of cooperation that it enables. However, today this power comes with great risk because those engaged in cooperation are left vulnerable to one another. The Object-Capability (OCap) Model is a promising remedy, because it enables the creation of security-enforcing abstractions, or *patterns*, that can be composed with other code to build systems that enable cooperation whilst minimising vulnerability. Unfortunately, little work has been done to adequately formally analyse standard OCap patterns. In particular, their analysis requires frameworks that are capable of reasoning about their, often novel, security properties whilst taking into account the variation that exists across different OCap systems, particularly in concurrency semantics.
We use the process algebra CSP to examine the implementations of a number of OCap patterns and their security properties in various kinds of OCap system. CSP allows us to not only reason about security properties beyond the reach of previous approaches, such as revocation, but also to consider different models of concurrency that various kinds of OCap system exhibit. We find that while some implementations function correctly when moved from one kind of system to another, others exhibit subtle differences or fail to preserve their security properties altogether.

Key words: Cooperation without vulnerability, Object-Capability security, security-enforcing abstractions, CSP, formal verification.

1 Introduction

The Object-Capability Model Modern computing technology is perhaps the best enabler of cooperation that humankind has yet developed. Indeed, much of the power and utility of modern computing arises in the different forms of cooperation that it enables. However, today this power comes with great risk because those engaged in cooperation are left vulnerable to one another. While desktop machines enable users to run software developed by anyone in the world, they do so at the risk of having their lives ruined by the actions that the software might take on their behalf. While one can write modern software by composing elements from various sources, each of which may have been written by different people at different times, one does so at the risk that one of these components might render the entire product faulty or malicious by its inclusion.

The benefits of enabling such cooperation are obvious and should not be abandoned. Instead, modern security architectures should seek to enable and promote new and better forms of *cooperation without vulnerability*[1].

One architecture that shows significant promise in this area is the *Object-Capability Model* [6]. Object-Capability (OCap) operating systems (OSs), such as EROS and seL4, enable users to run arbitrary code whilst remaining safe from its potential 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[2].

Briefly, an OCap system is a collection of *objects*, connected to each other by *capabilities*. An object is a protected entity comprising state and code that together define its behaviour. An object's state includes both data and the capabilities it possesses. A capability is an unforgeable object reference that allows its holder to send messages to the object it references by *invoking* the capability. An object's code naturally governs how it will react to incoming messages.

The only way to pass capabilities is by sending messages. In practice, object o can pass one of its capabilities, say c, directly to object p only by invoking a capability it possesses to p, including c in the invocation. This implies that capabilities can only be passed between objects that are connected, perhaps via intermediate objects. An object may also create others. In doing so, it must supply any resources required by the newly created object, including its code, as well as any capabilities it is initially to possess. Hence, a newly created object receives its first capabilities solely from its parent.

In OCap OSs like EROS and seL4, each OS process may be thought of as a separate object. In OCap languages like E and Caja, objects are akin to those from object-oriented languages and capabilities are simply object references.

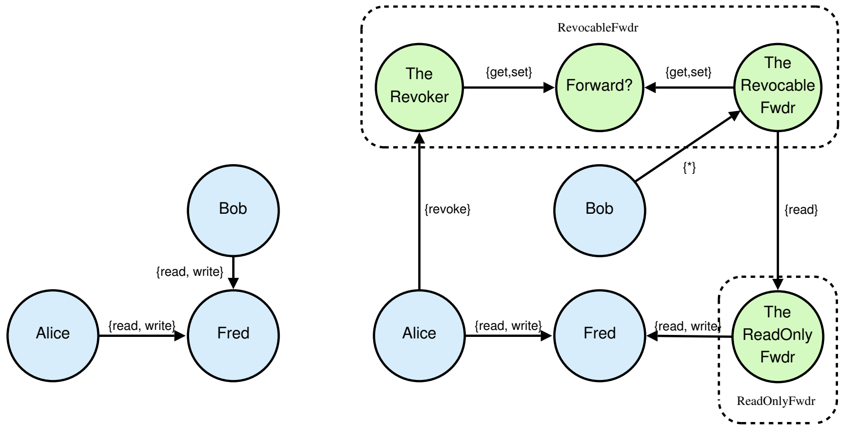
Security-Enforcing Abstractions The OCap model is powerful because it allows programmers to create *security-enforcing abstractions* [6], here called *patterns*, that can be composed with other, perhaps non-security enforcing, code to enable cooperation while minimising vulnerability.

As an example, suppose Alice, an architect, is to cooperate with Bob, a builder. In particular, she needs to be able to send blueprints to Bob via the file Fred which supports standard operations such as read and write. A naive solution, as depicted in Figure 1(a), would give both Alice and Bob direct access to Fred. But now Alice is potentially vulnerable to Bob, since Bob might choose to write malicious contents to Fred. Instead, we can compose two patterns, the ***ReadOnlyForwarder*** and the ***RevocableForwarder***, with Alice, Bob and Fred as depicted in Figure 1(b), in order to ensure they can cooperate without exposing Alice to undue vulnerability. Here, the new objects have been grouped according to the pattern they instantiate.

We instantiate the ***ReadOnlyForwarder*** pattern with Fred to produce an object, TheReadOnlyFwdr, that provides read-only access to Fred. TheReadOn-

[1] This phrase is borrowed directly from Miller *et al.* [7].

[2] Appendix B lists references for each OCap system mentioned in this paper.



(a) Alice is vulnerable to Bob.

(b) By deploying security-enforcing patterns, we minimise Alice's vulnerability to Bob.

Fig. 1: Security-enforcing patterns enable cooperation whilst reducing vulnerability.

lyFwdr responds only to read requests by forwarding the request to its target, here Fred. We then instantiate the ***RevocableForwarder*** pattern with this to produce three objects, only two of which are used directly by Alice or Bob. The first, TheRevocableFwdr, provides indirect access to TheReadOnlyFwdr and is handed to Bob to give him read-only access to Fred. The second, TheRevoker, allows Bob's read-only access to Fred to be revoked and is handed to Alice. When invoked, TheRevocableFwdr forwards whatever message it received to its target, here TheReadOnlyFwdr, so long as the object Forward? holds a boolean value of true. However, when TheRevoker is invoked, it modifies the value held in Forward? to false, thereby preventing TheRevocableFwdr from forwarding any future invocations. Finally, we also give Alice a direct reference to Fred. Now Alice and Bob can cooperate without exposing Alice to undue vulnerability since Bob is prevented from being able to write to Fred. Once they have finished cooperating, Alice can revoke Bob's access to Fred by invoking TheRevoker[3].

The development of security-enforcing OCap patterns has had a long history, of which Miller provides a good overview in [6, Section 9.5]. Over the years, many patterns have been developed in a range of OCap systems. Some patterns in particular have consistently reappeared in a number of different systems, including ***RevocableForwarders*** [8] and ***AttenuatingForwarders***, such as the ***ReadOnlyForwarder*** above, (in *e.g.* E, KeyKOS and Emily), ***Sealer-Unsealers*** [3] (in *e.g.* E, KeyKOS, Emily and Caja) and ***Membranes*** [6] (in *e.g.* E, KeyKOS, DCCS and Emily).

Despite their ubiquity, little work has been done to formally analyse the implementations of these common patterns. The wide variety of OCap systems necessitates the formal analysis of their implementations to determine in which systems they can be applied. In particular, single-threaded OCap languages, like

[3] Note that Bob is still vulnerable to Alice choosing to send him a malicious image. Guarding against this involves being able to determine which images are malicious.

Caja and Joe-E, have vastly different models of concurrency than OCap OSs like EROS and seL4, where each process runs concurrently to all others. Concurrency is known to introduce subtle effects that can enable a system's security properties to be violated. Cryptographic protocols provide the perfect example [10]. Hence, implementations that are originally designed in one kind of system may not function correctly in another. These patterns must, therefore, be analysed in a framework that is capable of expressing various levels of concurrency. This rules out the application of traditional techniques like Take-Grant Models [4] and Knowledge-Behaviour Models [12].

Contribution In this paper, we formally analyse the implementation of a number of common OCap patterns, within the different contexts of OCap languages and operating systems, in order to verify their security properties and determine in which contexts they can be applied. Our formal analysis is conducted using the process algebra CSP [9] and its model-checker FDR [1]. Appendix A provides a brief overview of CSP. We chose CSP because it was explicitly designed for reasoning about concurrency and has a rich and successful history of detecting concurrency-related vulnerabilities, most notably in cryptographic protocols [10].

CSP allows us to express a system's security properties in the form of process *refinements*. In this paper, we restrict our attention to *simple trace-refinements* in which we assert that every sequence of events that can be performed by the system, *System*, can also be performed by some specification process, *Spec*, that is constructed to perform every sequence of events that does not violate the relevant security property. Despite their name, simple trace-refinements can express properties, such as revocation, that are beyond the reach of previous OCap formalisms (*e.g.* [12]). More complex refinements can be used to express subtler properties still [5]. For finite-state *Spec* and *System*, FDR can automatically test whether the refinement holds and return a *counter-example* in the form of a behaviour of *System* that is not a behaviour of *Spec* when it does not.

Using CSP, we model a pattern in the two different contexts of a single-threaded OCap language and a concurrent OCap OS to produce two different processes that represent the pattern in each context. To compare the pattern between the two contexts we simply apply the refinement tests that represent its security properties to each of the processes and compare the results.

This paper is organised as follows. In Section 2, we describe our approach to modelling OCap patterns in CSP, both within the context of OCap languages and OSs, with the aid of the ***Membrane*** pattern as a running example. In Section 3, we then apply this approach to the implementations of some common OCap patterns to reason about their security properties within these two different contexts. We consider the ***RevocableMembrane*** pattern, an extension to the ***Membrane*** pattern that incorporates the logic of the ***RevocableForwarder*** pattern and whose analysis requires reasoning about revocation. We also consider an implementation of the ***Sealer-Unsealer*** pattern and find that its security properties are greatly affected by shifts in concurrency semantics. Finally, we conclude in Section 4 and consider some directions for future research.

2 Modelling OCap Patterns in CSP

In this section, we discuss how to model and reason about the various OCap patterns in CSP within the context of OCap OSs and languages. We do so with the aid of the first pattern that we will analyse, in order to help explain our overall approach. Later, we apply these techniques to other patterns of interest.

The *Membrane* Pattern Figure 2 depicts an instance of the ***Membrane*** pattern [6], which has been applied across many OCap systems, including in OSs such as KeyKOS, languages such as E and distributed systems such as DCCS. This pattern allows one to hand out a capability, c, to an object while ensuring that some behaviour, such as enforcing read-only access to objects, is transitively applied to all capabilities reachable from c. We see here that Alice has been given a capability to a membrane object, TheMembrane. TheMembrane forwards Alice's invocations to Bob, thereby allowing her to invoke Bob. The membrane might implement some behaviour, such as restricting the methods that Alice is allowed to call. In response to an invocation, Bob might return to TheMembrane his capability to Carol. In this case, TheMembrane returns to Alice a capability to a new instance of itself that forwards invocations to Carol rather than Bob. We say here that TheMembrane has *wrapped* the capability to Carol. Alice can now invoke both Bob and Carol, but we ensure that TheMembrane's policy is applied in both cases. TheMembrane also wraps any capabilities that Alice may pass it in her invocations, before forwarding them on. This prevents Bob, for example, from obtaining a direct (not wrapped) capability to Alice, which he could use to violate TheMembrane's policy by sending Alice a direct capability to himself.

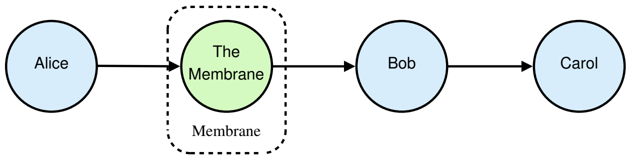


Fig. 2: An instance of the ***Membrane*** pattern.

2.1 System Model

When modelling a pattern, we consider a small system that comprises an instance of the pattern, composed with some other objects. In the case of the ***Membrane*** pattern, the pattern is instantiated by the single object, TheMembrane. It is composed with the other objects, Alice, Bob and Carol. Ideally, we wish to determine whether a pattern preserves its security properties regardless of the objects it is composed with. Hence, when modelling the ***Membrane*** pattern, we would ideally like to specify that Alice, Bob and Carol exhibit maximum possible behaviour. Most patterns inevitably place some obvious constraints on the objects with which they're composed. In this paper, we take these into account as necessary.

Each system comprises a set, *Object*, of objects. For the ***Membrane***, we have $Object = \{\mathsf{Alice}, \mathsf{TheMembrane}, \mathsf{Bob}, \mathsf{Carol}\}$. 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 to object o_2 specifying the operation *op* and containing the argument *arg*. For simplicity, the only operations we distinguish are Call and Return which correspond to an object invocation and return in an OCap language, or an inter-process send and reply in an OCap OS. The set of operations, *Op*, is simply $\{\mathsf{Call}, \mathsf{Return}\}$. The argument, *arg*, may either be a capability or the value null. We avoid modelling data explicitly and instead assume that any message might contain arbitrary data. This gives us the following alphabet for our systems.

$$\{o_1.o_2.op.arg \mid o_1 \in Object, o_2 \in Object, op \in Op, arg \in Object \cup \{\mathsf{null}\}\}$$

Each object, $o \in Object$, has its own alphabet of events in which it can participate, denoted $\alpha(o)$. An object o can be involved in invoking another, o', or in o' invoking it. Hence, we define its alphabet as $\alpha(o) = \{\!|o.o', o'.o \mid o' \in Object - \{o\}|\!\}$. Note that this means that, in our model, messages can only be sent between distinct objects and not from an object to itself.

2.2 Modelling Object Behaviours

We define a CSP process for each type of behaviour within our system. In order to ease both the writing and understanding of these object behaviour definitions, we introduce some convenience notation. The event $\mathsf{send}.other.op.arg$ represents the object sending a message to another object, *other*, specifying operation *op* and argument *arg*. Its dual is $\mathsf{recv}.from.op.arg$ and represents the receipt of a message from the object *from* for operation *op* containing the argument *arg*. In the case of the ***Membrane***, we might specify its behaviour as follows.

$$Membrane(target, self) = Membrane'(\{target\}, self)$$

$$wrap(\mathsf{null}, self) = \mathsf{null}\,; \qquad wrap(other, self) = self$$

$$\begin{aligned}
&Membrane'(targets, self) = \mathsf{recv}?from!\mathsf{Call}?arg \rightarrow \\
&\quad \mathsf{send}?to : targets!\mathsf{Call}!wrap(arg, self) \rightarrow \mathsf{recv}!to!\mathsf{Return}?res \rightarrow \\
&\quad \mathsf{send}!from!\mathsf{Return}!wrap(res, self) \rightarrow \\
&\quad Membrane'(targets \cup (Object \cap \{res, arg\}), self)
\end{aligned}$$

In order to ensure that our systems are finite-state and their analysis is, thus, decidable, we choose not to model object creation explicitly. Instead, we borrow the idea of Spiessens [12] and aggregate the behaviour of any of an object's children into itself. A membrane that is initially forwarding invocations to Bob and who subsequently creates a child membrane to forward invocations to Carol, will act like the aggregation of both itself and its child. A membrane who is initially forwarding requests to *target* and whose identity is *self* is represented by the process $Membrane(target, self)$. TheMembrane's behaviour is captured by the process $Membrane(\mathsf{Bob}, \mathsf{TheMembrane})$.

The function $wrap(arg, self)$ returns the capability that results from the membrane whose identity is *self* wrapping the argument *arg*. When *arg* is null, no wrapping is required. Hence, $\mathit{wrap}(\mathsf{null}, \mathit{self}) = \mathsf{null}$. Otherwise, the argument is a capability and is wrapped by creating a new membrane object that forwards invocations to it. In our case, the new object is aggregated into its parent whose identity is *self*. Hence, a capability to the parent, *self*, is returned to represent the capability to the newly created child that has been aggregated into the parent. Hence, for $\mathit{target} \neq \mathsf{null}$, $\mathit{wrap}(\mathit{target}, \mathit{self}) = \mathit{self}$.

We define the process $\mathit{Membrane}'(\mathit{targets}, \mathit{self})$ to represent a membrane that is the aggregation of the membranes for all of the targets in the set *targets* and whose own identity is *self*. Initially, the membrane is aggregating nothing but itself, hence it is forwarding to its original target only. Once an aggregated membrane has received an invocation, it wraps whatever argument, *arg*, it was passed and offers the choice of forwarding it to any of its targets. Whatever response is received, *res*, it wraps this and returns it to its invoker. It is now aggregating any new membranes that might have been created while wrapping *arg* and *res*. Note that the set *targets* is only ever increased by adding object names from the finite set *Object*. Therefore, *targets* is always bounded and, thus, our membrane model is finite-state.

Given an object, $o \in \mathit{Object}$, whose behaviour is represented by some process, *Behaviour*, we can then represent this object within our system by the process that acts like *Behaviour* except that all $\mathsf{send}.\mathit{other}.\mathit{op}.\mathit{arg}$ events are renamed to $o.\mathit{other}.\mathit{op}.\mathit{arg}$ and likewise all $\mathsf{recv}.\mathit{from}.\mathit{op}.\mathit{arg}$ events are renamed to $\mathit{from}.o.\mathit{op}.\mathit{arg}$. CSP naturally supports this kind of renaming (see Appendix A).

After renaming, the system is then built as the alphabetised parallel composition of these processes, with their corresponding alphabets. Hence, because of CSP's synchronous semantics, the system can perform an event $o_1.o_2.\mathit{op}.\mathit{arg}$ only when the process representing o_1 is willing to perform the event $\mathsf{send}.o_2.\mathit{op}.\mathit{arg}$ while the process representing o_2 is willing to perform $\mathsf{recv}.o_1.\mathit{op}.\mathit{arg}$.

2.3 Modelling Objects Exhibiting Maximum Possible Behaviour

We now discuss how to represent objects, such as Alice, Bob and Carol that are to exhibit maximum possible behaviour as described earlier in Section 2.1. We consider how to represent objects exhibiting maximum possible behaviour within OCap OSs and languages respectively. Their representation differs in each case since each kind of system places different constraints on the possible behaviours an object can exhibit within it.

OCap OSs The case of an object in an OCap OS, such as EROS or seL4, is perhaps the simplest. It is represented by the CSP process $\mathit{Untrusted}_{OS}(\mathit{caps})$ that models an object that initially possesses the capabilities in the set *caps* as follows.

$$\begin{aligned}
&\mathit{Untrusted}_{OS}(\mathit{caps}) = \\
&\quad \mathsf{send}?\mathit{to} : \mathit{caps}?\mathit{op}?\mathit{arg} : \mathit{caps} \cup \{\mathit{null}\} \rightarrow \mathit{Untrusted}_{OS}(\mathit{caps}) \,\square \\
&\quad \mathsf{recv}?\mathit{from}?\mathit{op}?\mathit{arg} \rightarrow \mathit{Untrusted}_{OS}(\mathit{caps} \cup (\{\mathit{arg}, \mathit{from}\} \cap \mathit{Object}))
\end{aligned}$$

This object may invoke any of its capabilities, passing any argument it has access to, at any time. It need not necessarily wait for a response before choosing to invoke another capability. Hence, after doing so, it returns to its initial state. Alternatively, it can choose to block waiting for a message. Once it received a message it acquires the capability it contains as well as a capability to the sender; this allows us to model mechanisms like EROS's "resume" capabilities [11], which are included in inter-process messages in order to allow the invokee to send back a response.

OCap Languages We now consider how to model an object exhibiting maximum possible behaviour in an OCap language, like Caja or Joe-E.

We distinguish between when an object is *active* and when it is *inactive*. An object is active precisely when one of its methods is currently being executed and is inactive otherwise. Naturally, in the sorts of single-threaded OCap languages that we are modelling, only one object is ever active at a time.

We define the process $\mathit{Untrusted}_{\mathit{lang}}(\mathit{caps})$ that represents an initially inactive object in an OCap language that exhibits maximum possible behaviour and initially possesses the capabilities in the set caps.

$$\begin{array}{l}\mathit{Untrusted}_{\mathit{lang}}(\mathit{caps}) = \mathsf{recv}?\mathit{from}?\mathit{op}?\mathit{arg} \rightarrow \\ \quad \mathit{UntrustedActive}_{\mathit{lang}}(\mathit{caps} \cup (\mathit{Object} \cap \{\mathit{arg}, \mathit{from}\}))\end{array}$$

It 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 and exhibits maximum possible behaviour and possesses the capabilities in the set caps is represented by the process $\mathit{UntrustedActive}_{\mathit{lang}}(\mathit{caps})$, which is defined as follows.

$$\begin{array}{l}\mathit{UntrustedActive}_{\mathit{lang}}(\mathit{caps}) = \mathsf{send}?\mathit{to} : \mathit{caps}?\mathit{op}?\mathit{arg} : \mathit{caps} \cup \{\mathit{null}\} \rightarrow \\ \quad \mathit{Untrusted}_{\mathit{lang}}(\mathit{caps})\end{array}$$

It 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 Joe-E and Caja that enforce strict call-return semantics. Patterns that uphold their properties in this model will, therefore, also uphold them in more restrictive settings that disallow these impossible behaviours. Patterns that fail to uphold their properties in our permissive model may do so either because they are broken (*i.e.* fail to uphold their properties in real languages like Joe-E and Caja) or because their security properties are violated by a behaviour that would be impossible in a real language. In cases where a pattern is subverted by a clearly impossible behaviour, we can further restrict the model to remove the impossible behaviour. If the pattern works in the restricted model, we have evidence to believe it will work in real languages like Joe-E and Caja that enforce strict call-return semantics.

2.4 Completing the Analysis

We are now in a position to complete the models of the ***Membrane*** pattern and reason about its security properties. Alice, Bob and Carol are each initially given the capabilities $\{\mathsf{Alice}, \mathsf{TheMembrane}\}$, $\{\mathsf{Bob}, \mathsf{Carol}\}$ and $\{\mathsf{Carol}\}$, respectively. Recall that each are to exhibit maximum possible behaviour in order to reason about the security properties of the pattern as independently of the behaviour of the objects it is composed with as possible. Recall also that TheMembrane is represented by the process $\mathit{Membrane}(\mathsf{Bob}, \mathsf{TheMembrane})$.

The pattern is instantiated in the context of an OCap OS by instantiating each of Alice, Bob and Carol as $\mathit{Untrusted}_{OS}(\mathit{caps})$, where *caps* denotes the appropriate set of capabilities defined above, and coupling these with the representation of TheMembrane to produce a process representing the system denoted $\mathit{MSystem}_{OS}$.

Alternatively, we can instantiate this pattern within the context of an OCap language. First we must decide which object is initially active. Alice appears the only sensible choice. Hence, we represent Alice as $\mathit{UntrustedActive}_{lang}(\mathit{AliceCaps})$, where *AliceCaps* denotes her initial capabilities defined above. Bob and Carol are represented as $\mathit{Untrusted}_{lang}(\mathit{caps})$, where *caps* denotes the appropriate set of capabilities for each. Coupling the representations of Alice, Bob and Carol with that of TheMembrane produces a system represented by the process denoted $\mathit{MSystem}_{lang}$.

We now have two instances of the same ***Membrane*** pattern, modelled in two different kinds of OCap system, to compare. The main property that we wish to reason about is how the pattern affects the propagation of capabilities. This corresponds to a *safety analysis* in conventional access control models, such as [2], and reasons about how a pattern affects which objects can directly interact with each other.

In the case of the ***Membrane***, the obvious property to check is whether Alice can obtain a capability to Bob or Carol. If this occurs, then the system will perform an event from $A = \{|\mathsf{Alice.Bob}, \mathsf{Alice.Carol}|\}$. We can test whether this is the case by checking whether the process that represents the system trace-refines $CHAOS_{\Sigma - A}$, the most general process that never performs any event in A. FDR reveals that both $\mathit{MSystem}_{OS}$ and $\mathit{MSystem}_{lang}$ trace-refine $CHAOS_{\Sigma - A}$, indicating that the pattern does uphold this property in both kinds of system.

3 Analysing Other OCap Patterns

We now consider two other OCap patterns. The first deals with revocation, which is beyond the scope of traditional OCap formalisms to reason about (see *e.g.* [12]). The second highlights the problems of trying to port language-based patterns to OCap operating systems, whose concurrency semantics can break their security properties.

3.1 The *RevocableMembrane* Pattern

The ***RevocableMembrane*** pattern is an extension of the ***Membrane*** pattern that incorporates the logic of the ***RevocableForwarder***, discussed earlier in

Section 1, to allow all capabilities obtained through the membrane to be revoked. The membrane is enhanced to hold a capability to a *bool* object that contains a boolean value. When invoked, the membrane first checks that its bool holds the value true before forwarding the invocation. For each membrane, there is a corresponding *revoker* object that also holds a capability to the membrane's bool. When invoked, the revoker simply alters the contents of the bool to false in order to prevent the membrane from forwarding any further invocations. A membrane's children use the same bool as its parent, thereby allowing all capabilities wrapped by the same membrane to be revoked together.

We represent a bool whose identity is *me* and initial boolean value is *val* as $Bool(me, val)$ which is defined as follows.

$$boolRet(me, \mathsf{true}) = me\ ; \qquad boolRet(me, \mathsf{false}) = \mathsf{null}$$

$$\begin{aligned} &Bool(me, val) = \mathsf{recv}?from!\mathsf{Call}?arg \to \mathsf{send}!from!\mathsf{Return}!boolRet(me, val) \\ &\quad \to (Bool(me, val) \lessdot arg = \mathsf{null} \gtrdot Bool(me, \neg val)) \end{aligned}$$

In response to an invocation, the bool returns its current value. If the argument passed with the invocation is not null, the bool then toggles its value; otherwise it retains its current value. Since the bool can only return values from the set $Object \cup \{\mathsf{null}\}$, a value of false is represented by the bool returning null. Any non-null return represents true but the bool will always return a capability that refers to itself in this case.

We enhance the membrane presented earlier so that it now holds a reference, *bool*, to its bool that it invokes to check its value before deciding whether to forward an invocation. Only if the bool returns non-null does it continue and forward the invocation as before.

$$RMembrane(target, self, bool) = RMembrane'(\{target\}, self, bool)$$

$$\begin{aligned} &RMembrane'(targets, self, bool) = \mathsf{recv}?from!\mathsf{Call}?arg \to \\ &\quad \mathsf{send}.bool.\mathsf{Call}.\mathsf{null} \to \mathsf{recv}.bool.\mathsf{Return}?bVal \to \\ &\quad bVal \neq \mathsf{null}\ \&\ (\ \mathsf{send}!to : targets!\mathsf{Call}!wrap(arg, self) \to \\ &\qquad \mathsf{recv}!to!\mathsf{Return}?res \to \mathsf{send}!from!\mathsf{Return}!wrap(res, self) \to \\ &\qquad RMembrane'(targets \cup (Object \cap \{res, arg\}), self, bool)\ ) \end{aligned}$$

The only kind of behaviour left to define is that for the revoker. A revoker, whose bool is *bool*, is simply defined as follows.

$$\begin{aligned} &Revoker(bool) = \mathsf{recv}?from!\mathsf{Call}!\mathsf{null} \to \mathsf{send}!bool!\mathsf{Call}!bool \to \\ &\quad \mathsf{recv}!bool!\mathsf{Return}?oldVal \to \mathsf{send}!from!\mathsf{Return}!\mathsf{null} \to STOP \end{aligned}$$

The revoker simply waits to be invoked, at which point it Calls its bool, passing it a non-null argument which causes the bool to toggle its value. Once the bool returns, the revoker simply returns null to its caller and then disallows any further invocation.

The instantiation of this pattern in both kinds of system is based on that for the ***Membrane*** pattern presented earlier. We instantiate three objects, TheMembrane, TheBool and TheRevoker to represent this pattern, as $RMembrane(\mathsf{Bob}, \mathsf{TheMembrane}, \mathsf{TheBool})$, $Bool(\mathsf{TheBool}, \mathsf{true})$ and $Revoker(\mathsf{TheBool})$ respectively. Bob and Carol are instantiated as before. Alice is also instantiated as before in each kind of system, except that now she is also initially 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 like to verify that it also enforces revocation. In order to do this, we need to test whether, once TheRevoker has been invoked and this invocation has Returned, that TheMembrane can no longer Call Alice, Bob or Carol. We can test for this using FDR as follows.

We define the most general process, *Spec*, that can never perform an event representing TheMembrane Calling Alice, Bob or Carol once it has performed an event representing TheRevoker Returning from an invocation. The revocation property holds, then, if the system trace-refines *Spec*.

$$\begin{aligned}&\mathit{Spec} = ?e : \Sigma - \{|\mathsf{TheRevoker}.o.\mathsf{Return} \mid o \in \mathit{Object}|\} \to \mathit{Spec}\\ &\quad \Box\ \mathsf{TheRevoker}?o!\mathsf{Return}?\mathit{arg} \to \mathit{CHAOS}_{\Sigma - A},\\ &\quad \text{where } A = \{|\mathsf{TheMembrane}.\mathit{abc}.\mathsf{Call} \mid \mathit{abc} \in \{\mathsf{Alice}, \mathsf{Bob}, \mathsf{Carol}\}|\}\end{aligned}$$

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

$$\begin{aligned}&\langle \mathsf{Alice.TheMembrane.Call.null},\ \mathsf{TheMembrane.TheBool.Call.null},\\ &\ \mathsf{TheBool.TheMembrane.Return.TheBool},\ \mathsf{Alice.TheRevoker.Call.null},\\ &\ \mathsf{TheRevoker.TheBool.Call.TheBool},\ \mathsf{TheBool.TheRevoker.Return.TheBool},\\ &\ \mathsf{TheRevoker.Alice.Return.null},\ \mathsf{TheMembrane.Bob.Call.null}\rangle\end{aligned}$$

Here, 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. We see further evidence of this when we examine the next pattern, whose security properties are completely destroyed when moving it into the OS environment from its original language context.

3.2 The *Sealer-Unsealer* Pattern

The final pattern we consider is the ***Sealer-Unsealer*** pattern [3]. It is 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 capability c'. c' cannot be used for any purpose on its own. However, 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. One of its primary uses is to allow one to transport a sensitive capability, c, via an untrusted intermediary in the form of the innocuous capability c'.

Here, we consider a particular implementation of this pattern, originally developed by Stiegler [13] in E and since ported to Caja, in order to verify its security properties within the context of an OCap language and to determine whether they carry over to OCap OSs.

The implementation works as follows. 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. The purpose of the box is to place a copy of c into the slot when it is invoked. The unsealer works as follows. When invoked with a potential box, it first clears the slot. It then invokes the potential box. If the box was created by the matching sealer, then 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 model of this pattern is depicted in Figure 3. Here, 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.

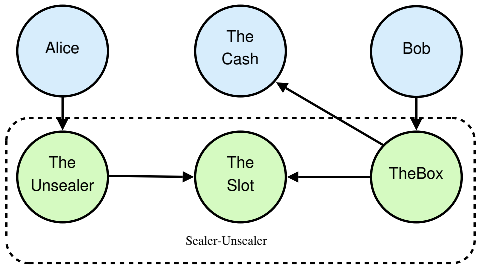


Fig. 3: Modelling the ***Sealer-Unsealer*** pattern.

The behaviour of a slot is approximated by the following process.

$$Slot(val) = \mathsf{recv}?from!\mathsf{Call}?new \rightarrow \mathsf{send}!from!\mathsf{Return}!val \rightarrow Slot(new)$$

Our model cannot distinguish read and write messages. Hence, the slot is implemented as a one-place buffer. When invoked, it stores whatever argument it is passed and returns its old value to the caller.

The behaviour of a box is represented by this process.

$$\begin{aligned}&Box(slot, contents) = \mathsf{recv}?from!\mathsf{Call}?arg \rightarrow\\ &\quad \mathsf{send}.slot.\mathsf{Call}.contents \rightarrow \mathsf{recv}.slot.\mathsf{Return}?val \rightarrow\\ &\quad \mathsf{send}.from.\mathsf{Return}.\mathsf{null} \rightarrow Box(slot, contents)\end{aligned}$$

It receives an invocation, places its contents in the slot by Calling it and receiving its response, then Returns the value null to its invoker.

The behaviour of an unsealer is represented as follows.

$$\begin{aligned}&Unsealer(slot) = \mathsf{recv}?from!\mathsf{Call}?box : Object \rightarrow\\ &\quad \mathsf{send}.slot.\mathsf{Call}.\mathsf{null} \rightarrow \mathsf{recv}.slot.\mathsf{Return}?oldVal \rightarrow\\ &\quad \mathsf{send}.box.\mathsf{Call}.\mathsf{null} \rightarrow \mathsf{recv}.box.\mathsf{Return}.\mathsf{null} \rightarrow\\ &\quad \mathsf{send}.slot.\mathsf{Call}.\mathsf{null} \rightarrow \mathsf{recv}.slot.\mathsf{Return}?contents : Object \rightarrow\\ &\quad \mathsf{send}.from.\mathsf{Return}.contents \rightarrow Unsealer(slot)\end{aligned}$$

It receives an invocation, ensuring that it contains a capability to a supposed box. It then Calls the slot to clear its contents, receiving its old value, and then invokes the box. Once the box Returns, the unsealer Calls its slot to check whether the box placed any contents within it and to clear whatever value it might contain. It accepts only Return messages from the slot that contain a capability. Hence, it will proceed only if the slot contained a capability. It then Returns the slot's contents to its caller.

When modelling this pattern in the context of an OCap language, both Alice and Bob will be initially inactive. Hence, we need to add an extra *driver* object that is initially active that can invoke both Alice and Bob. This object should not allow either to pass capabilities to the other, however. We define its behaviour as follows.

$$Driver(caps) = \mathsf{send}?to : caps!\mathsf{Call}!\mathsf{null} \rightarrow \mathsf{recv}!to!\mathsf{Return}?arg \rightarrow Driver(caps)$$

The object that instantiates this behaviour we call TheDriver.

We instantiate the model of this pattern in the context of each kind of system as follows. In both kinds of system, TheUnsealer, TheSlot and TheBox are instantiated as $Unsealer(\mathsf{TheSlot})$, $Slot(\mathsf{null})$ and $Box(\mathsf{TheSlot}, \mathsf{TheCash})$, respectively. The initial capabilities possessed by Alice, TheCash and Bob in both kinds of system are $\{\mathsf{Alice}, \mathsf{TheUnsealer}\}$, $\{\mathsf{TheCash}\}$ and $\{\mathsf{Bob}, \mathsf{TheBox}\}$, respectively. Each is instantiated in the OS context as $Untrusted_{OS}(caps)$, where $caps$ denotes the appropriate set of initial capabilities, to produce a system represented by the process denoted $SUSystem_{OS}$. Likewise, each of Alice, TheCash and Bob are instantiated in the language context as $Untrusted_{lang}(caps)$ where $caps$ denotes the appropriate set of initial capabilities. Finally, TheDriver is instantiated in the language context as $Driver(\{\mathsf{Alice}, \mathsf{Bob}\})$ to produce a system represented by the process denoted $SUSystem_{lang}$.

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 $A = \{\!|o.\mathsf{TheCash} \mid o \in \{\mathsf{Alice}, \mathsf{Bob}\}|\!\}$.

FDR indicates that this property does not hold for $SUSystem_{lang}$. This is somewhat surprising until one examines the counter-example returned by FDR, which is as follows.

$$\begin{array}{l}
\langle \underline{\mathsf{TheDriver.Alice.Call.null}},\ \underline{\mathsf{Alice.TheUnsealer.Call.Alice}},\\
\mathsf{TheUnsealer.TheSlot.Call.null},\ \mathsf{TheSlot.TheUnsealer.Return.null},\\
\underline{\mathsf{TheUnsealer.Alice.Call.null}},\ \underline{\mathsf{Alice.TheDriver.Return.null}},\\
\mathsf{TheDriver.Bob.Call.null},\ \mathsf{Bob.TheBox.Call.null},\\
\mathsf{TheBox.TheSlot.Call.TheCash},\ \mathsf{TheSlot.TheBox.Return.null},\\
\mathsf{TheBox.Bob.Return.null},\ \mathsf{Bob.TheDriver.Return.TheBox},\\
\mathsf{TheDriver.Alice.Call.null},\ \mathsf{Alice.TheUnsealer.Return.null},\\
\mathsf{TheUnsealer.TheSlot.Call.null},\ \mathsf{TheSlot.TheUnsealer.Return.TheCash},\\
\mathsf{TheUnsealer.Alice.Return.TheCash},\ \mathsf{Alice.TheCash.Call.TheDriver}\rangle
\end{array}$$

Here, the first four events in which Alice partakes have been underlined. Observe that this sequence of events is impossible in languages that enforce strict call-return semantics, such as Joe-E and Caja. In the counter-example, Alice Returns to TheDriver when she ought first return to TheUnsealer. This occurs because, as explained earlier in Section 2.3, our model of an object exhibiting maximum possible behaviour in an OCap language is too permissive.

In order to overcome this weakness, we restrict the system so that Alice can never perform this impossible sequence of events and then repeat the test in the restricted system. The system, $SUSystem_{lang}$, is restricted by placing it in parallel with a process, R, synchronising on all of Alice's events to form the restricted system $SUSystem'_{lang}$ as follows.

$$SUSystem'_{lang} = SUSystem_{lang} \underset{\alpha(\mathsf{Alice})}{\parallel} R$$

$$\begin{aligned}
R &= ?e : \alpha(\mathsf{Alice}) \rightarrow R' \not\lhd e \in \{\!|\mathsf{TheDriver.Alice.Call}|\!\} \not\rhd R \\
R' &= ?e : \alpha(\mathsf{Alice}) \rightarrow R'' \not\lhd e \in \{\mathsf{Alice.TheUnsealer.Call.Alice}\} \not\rhd R \\
R'' &= ?e : \alpha(\mathsf{Alice}) \rightarrow R''' \not\lhd e \in \{\mathsf{TheUnsealer.Alice.Call.null}\} \not\rhd R \\
R''' &= ?e : \alpha(\mathsf{Alice}) - \{\!|\mathsf{Alice.TheDriver.Return}|\!\} \rightarrow R
\end{aligned}$$

.

Repeating the test for $SUSystem'_{lang}$ reveals that, in the restricted system, the pattern does enforce its security property as expected. This indicates that the patter should enforce its security properties in languages like Caja and Joe-E that enforce strict call-return semantics.

While the property does hold in the language context, FDR indicates that in the OS context, represented by the process $SUSystem_{OS}$, the property does *not* hold. The counter-example that FDR returns is the following trace.

$\langle$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$\rangle$

Unlike in the case of the language context, this trace is an obviously valid behaviour. It reveals that 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. In this case, TheCash is still sitting in TheSlot when TheUnsealer checks its contents. Hence, TheUnsealer returns TheCash to Alice, despite the fact that she doesn't possess TheBox.

This means that while the implementation of this pattern has been shown to uphold its security properties in the language context, it cannot be directly applied in OCap OSs.

4 Conclusion

In this paper, we have considered the formal analysis of the implementations of OCap patterns in CSP, in order to compare how they function when moved from OCap languages, like Joe-E and Caja, to OCap OSs, like EROS and seL4. Our analysis has extended beyond the scope of previous OCap formalisms (*e.g.* [12]), in that we have explicitly reasoned about revocation and concurrency.

Some implementations, like that of the ***Membrane***, appear to enforce their security properties in both kinds of system. Others, like that of the ***Revocable-Membrane***, show subtle differences when moved into the context of an OCap OS. Even worse are others, like that of the ***Sealer-Unsealer*** considered in Section 3.2, which, when moved to an OCap OS, fail to uphold their security properties entirely.

It should be noted that these results apply only to the implementations of the patterns considered. For example, they do not show that it is impossible to implement the ***Sealer-Unsealer*** pattern in an OCap OS, but rather that simply applying a successful implementation from an OCap language does not guarantee success.

Despite its utility, our approach does have its shortcomings. We finish by considering these in order to shed light on opportunities for future research. One shortcoming of our approach is the need to ensure that systems remain finite-state in order that they may be checked using FDR. This prevents us from being able to explicitly model object creation. However, it should be noted that this same limitation also applies when modelling cryptographic protocols in CSP and has not prevented success in that area [10]. Hence, while recognising that this limitation does exist, we also have reason to believe that it shouldn't prevent the successful verification of OCap patterns in general. One way to try to overcome this problem is to model object creation explicitly while allocating children from a pre-defined finite pool. For example, in the case of the ***Membrane*** pattern, besides TheMembrane, the system comprises only Alice, Bob and Carol. Bob is the target of the initial membrane, TheMembrane, meaning that only two other child membranes can ever be required: one to wrap Alice and one to wrap Carol. Hence, the pool of children from which to allocate child membranes is finite. Modelling object creation in this way would afford greater precision whilst still keeping the system finite-state.

Another difficulty arises in trying to express systems in single-threaded OCap languages as the composition of a number of processes executing in parallel. Without doing this, we cannot reliably compare the operation of a pattern between the two contexts. However, it does make it difficult to detect vulnerabilities that might arise as a result of recursive invocation, since our security enforcing objects cannot be recursively Called[4]. We are currently investigating various techniques to try to overcome this limitation.

Finally, CSP's natural synchronous semantics make it difficult to reason directly about asynchronous invocation between objects. Common approaches to modelling asynchrony in CSP, such as using buffers, might be applicable here and this is an area we intend to investigate. Doing so would allow us to model asynchronous invocation that exists in many OCap systems, including languages like E and OSs like Coyotos.

[4] See [14] for a real example of this kind of vulnerability in a ***Sealer-Unsealer*** implementation.

Acknowledgments

The author would like to thank Gavin Lowe, who provided insightful review of early drafts of this paper and suggested the model of untrusted objects in OCap languages; Michael Goldsmith, who suggested the idea of explicitly modelling object creation by allocating children from a finite pool; and the anonymous reviewers for their comments.

References

1. Formal Systems (Europe) Limited. *Failures Divergences Refinement: FDR2 User Manual*, 2005. Available at: `http://www.fsel.com/documentation/fdr2/fdr2manual.ps`.
2. M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 1976.
3. J. James H. Morris. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, 1973.
4. R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *Journal of the ACM*, 24(3):455–464, 1977.
5. G. Lowe. On CSP refinement tests that run multiple copies of a process. In *Proceedings of the Seventh International Workshop on Automated Verification of Critical Systems, AVOCS '07*, 2007.
6. M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.
7. M. S. Miller, C. Morningstar, and B. Frantz. Capability-based financial instruments. In *Proceedings of Financial Cryptography 2000*, February 2000. Available at: `http://www.erights.org/elib/capability/ode/index.html`.
8. D. D. Redell. *Naming and Protection in Extendable Operating Systems.* PhD thesis, University of California, Berkeley, 1974. Published as Project MAC Technical Report TR-140, Massachusetts Institute of Technology.
9. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, Upper Saddle River, NJ, USA, 1997.
10. P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *Modelling and Analysis of Security Protocols: the CSP Approach.* Addison Wesley, 2000.
11. J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A fast capability system. In *SOSP '99: Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 170–185, 1999.
12. A. Spiessens. *Patterns of Safe Collaboration.* PhD thesis, Université catholique de Louvain, Louvain-la-Neuve, Belgium, February 2007.
13. M. Stiegler. A picturebook of secure cooperation, 2004. Presentation. Available at: `http://erights.org/talks/efun/SecurityPictureBook.pdf`.
14. D. Wagner. A broken brand?, March 2008. E-mail communication to the `e-lang` mailing list, available at: `http://www.eros-os.org/pipermail/e-lang/2008-March/012508.html`.

A A brief overview of CSP

In this section we give a brief overview of CSP. More details can be obtained elsewhere [9].

CSP is a process algebra for describing and reasoning about concurrent systems. A system modelled in CSP comprises a set of concurrently executing *processes*. Each process usually models some particular component of the system in question. Processes execute by performing *events*. An event represents an atomic communication; this might either be between two processes or between a process and the environment. Processes communicate with each other by synchronising on common events. We write Σ for the set of all visible events.

The process $STOP$ can perform no events. The process $a \rightarrow P$ can perform the event a, and then act like P. The process $?a : A \rightarrow P_a$ offers the set of events A; if a particular event a is performed, the process then acts like P_a. (The prefixing operator "$\rightarrow$" binds tighter than all other operators.)

CSP allows multi-part events where each part is separated by an infix dot ".", such as the event $\mathsf{up}.3$. The process $\mathsf{up}?a : A \rightarrow P_a$ initially offers the set of events $\{\mathsf{up}.a \mid a \in A\}$. The output operator "!" is used to offer specific events to the environment. The process $\mathsf{move}?x{:}X!3 \rightarrow P$ initially offers the set of events $\{\mathsf{move}.x.3 \mid x \in X\}$. The notation $\{\!|\mathsf{move}|\!\}$ denotes the set of events whose first part is move. The first part of a multi-part event is sometimes called a *channel*.

The process $P \;\Box\; Q$ represents an external choice between P and Q; the initial events of both processes are offered to the environment; when an event is performed, that resolves the choice.

The process $P \not< b \not> Q$ acts like P if the boolean condition b is true; otherwise it acts like Q. $b \;\&\; P$ is shorthand for $P \not< b \not> STOP$.

The process $CHAOS_A$ is the most nondeterministic, nondivergent process with alphabet A; it can perform any sequence of events from A, and refuse any events.

If f is a function whose domain includes events of P, then $f[P]$ is the process that can perform $f(x)$ whenever P can perform x, *i.e.* every event, x, of P in the domain of f is renamed to $f(x)$.

$P \underset{A}{\|} Q$ represents the parallel composition of P and Q, synchronising on events from A. For a set of processes, $\{P_1, \ldots, P_n\}$, and a set of alphabets $\{A_1, \ldots, A_n\}$ (each a subset of Σ), $\big\|_{1 \le i \le n}(P_i, A_i)$ represents the *alphabetised* parallel composition of the P_i, where each P_i is allowed to perform events only from the set A_i, and all processes that share a common event must synchronise on it.

We say that one process, Q, *trace-refines* another, P, written $P \sqsubseteq_T Q$, when every sequence of visible events that Q can perform, can also be performed by P. Q trace-refines P if and only if it never performs a sequence of events that P cannot perform.

B References to OCap Systems

This paper refers to a number of different OCap systems. In this appendix, we provide references to further information about each.

B.1 OCap OSs

DCCS J. E. Donnelley. A distributed capability computing system. In *Proceedings of the Third International Conference on Computer Communication*, pages 432–440, 1976.

KeyKOS N. Hardy. The KeyKOS architecture. *Operating Systems Review*, 19(4):8–25, 1985.

EROS J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A fast capability system. In *SOSP '99: Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 170–185, 1999.

seL4 P. Derrin, K. Elphinstone, G. Klein, D. Cock, and M. M. T. Chakravarty. A memory allocation model for an embedded microkernel. In *Proceedings of the 1st International Workshop on Microkernels for Embedded Systems*, January 2007.

Coyotos J. Shapiro, M. S. Doerrie, E. Northup, S. Sridhar, and M. Miller. Towards a verified, general-purpose operating system kernel. In *Proceedings of the 1st NICTA Workshop on Operating System Verification*, October 2004.

B.2 OCap Languages

E M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

Emily M. Stiegler. Emily: A high performance language for enabling secure cooperation. In *Proceedings of the Fifth International Conference on Creating, Connecting and Collaborating through Computing, C5 '07*, pages 163–169, 2007.

Joe-E A. M. Mettler and D. Wagner. The Joe-E language specification (draft). Technical Report UCB/EECS-2006-26, EECS Department, University of California, Berkeley, March 2006.

Caja M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript (draft), 2008. Draft from January 15, 2008, available at: `http://google-caja.googlecode.com/files/caja-spec-2008-01-15.pdf`.