Analysing Object-Capability Patterns With Murφ

Deian Stefan John C. Mitchell
Stanford University
{deian,mitchell}@cs.stanford.edu

Abstract

Object Capability (OCap) patterns can be used to enforce the principle of least authority. However, despite the popularity and promise of OCap patterns, these patterns have not been sufficiently analyzed or verified. We analyze several OCap patterns using the Murφ verification tool and confirm the utility of our model by finding previously known vulnerabilities. Because these vulnerabilities were not repaired in previous studies, we provide new patterns and verify their correctness, within the accuracy of our model. Considering both distributed system and language-based settings that are designed to support the OCap model, we show that some patterns that are secure in a language context are vulnerable in a distributed systems context. We further demonstrate implementable attacks on the revocable forwarder and sealer/unsealer patterns, using a (distributed) subset of E and provide code illustrating and implementing repairs.

Keywords Language–based security, Object Capability, Model-checking, E

1. Introduction

Although considerable effort is devoted to producing secure software systems, a significant number of system vulnerabilities and exploits are discovered each year [18]. One continuing problem is that many abstractions we are accustomed to using are inherently flawed. Consider, for example, the widely used POSIX API. A simple POSIX-based logging application, `aLogger`, when running on behalf of user `alice`, though only requiring write access to `/var/log/alice.log`, is given the full authority of user `alice`. This *ambient authority* is assumed by many POSIX system calls; for example, the system call

```
open(const char *pathname, int flags);
```

can be used by an application to open any file the invoking user may access. Even if `aLogger` is *honest* in only opening the log file, an exploit modifying the path name can result in `aLogger` overwriting sensitive files. This is possible because `aLogger` is executed with ambient authority and can overwrite whatever file `alice` has write-permission to.

The widely accepted *Principle of Least Authority* (POLA) dictates that system components should be given only the minimal authority needed to achieve their intended purpose, and no more. This principle can be applied to run `aLogger` by first opening `/var/log/alice.log` and only providing `aLogger` the file descriptor. Although the modified `aLogger` will, in actuality, be running with ambient authority, if all `open` system calls are disallowed, a POLA-like setting can be created and `aLogger` will not have the authority write to any file other than the that corresponding to the provided file descriptor. Of course, many applications using POSIX (and many other APIs) rely on ambient authority, a side effect of insecure APIs and widely accepted but security-unconscious programming patterns.

One general approach for developing secure software uses the Object Capability (OCap) model [10, 13, 14]. The OCap model refines the conventional object model by disallowing ambient authority and implicit transfers of authority, with example OCap programming languages such as E [28], JoeE [8], Emily [27], and W7 [23] often obtained by prohibiting features such as mutable static state, forged pointers and the ability of an object to access another's private state [13]. While the OCap model appears to be gaining popularity, comparatively little effort beyond the work of a few authors [7, 19–21, 26] has been devoted to analyzing OCap programming patterns that are designed to achieve specific security goals. In an interesting line of work, Murray [19] has analyzed several OCap patterns and discovered vulnerabilities in them. Since Murray did not develop repairs for some critical problems, or relate his model to implemented systems based on OCap concepts, we therefore his study by developing another model using Murφ [3, 15], confirm the adequacy of our model by comparison with his, present new patterns that address the limitations of the analyzed patterns, and show how the vulnerabilities and repairs can be achieved in implemented systems.

The main contributions of this work are:

[Copyright notice will appear here once 'preprint' option is removed.]

- *Murφ OCap model.* While our model is similar to the one developed by Murray [19], there are some differences. For example, our model forces strict call-return semantics, our treatment of sealer/unseal distinguishes between slot reads and writes (and the slot does not return the value to the writer), and for the membrane pattern we distinguish between wrapped and unwrapped messages instead of aggregating objects.
- *Confirm previous vulnerabilities found by Murray.* The correspondence between our model-checking results and the set of vulnerabilities found in the previous study [19] confirms the repeatability and robustness of these results. Further, this gives us more confidence when we check our repairs and do not find additional vulnerabilities, since other studies do not find more vulnerabilities than ours.
- *Develop and verify repairs.* We propose modified patters that repair the vulnerabilities found by model-checking, and use model-checking to confirm the security of the improved patterns. (This is new, relative to previous studies.)
- *Study vulnerabilities and repairs in OCap languages and distributed systems.* We explore the correspondence between the OCap model, as we understand it and as formulated in Murφ, and contemporary programming languages and distributed/operating systems where developers may attempt to use OCap patterns for security. In particular, we demonstrate by executable implementation vulnerabilities and repairs of OCap patterns in a subset of E and explain ways that TahoeLAFS falls short of enforcing OCap properties needed to support its claimed security guarantees.

This paper is organized as follows. Section 2 reviews object capabilities and related concepts. Section 3 describes our Murφ model of Ocap features and patterns, while section 4 presents our model-checking results. Section 5 describes applications of our analysis to languages implementing this model (such as Joule, Erlang, and E), provides attacks and repairs on E-implemented OCap patterns, and demonstrates an attack on TahoeLAFS. Section 6 concludes.

2. Background

In this section we present some of the background on which the present work builds. We first detail the OCap model and OCap patterns used for safe collaboration, followed by a short introduction to Murφ model-checker.

2.1 The OCap model

In the OCap model there is no distinction between a subject (e.g. Alice) and an object (e.g. a file, or class instance). Rather, both are considered objects and all communication between objects is accomplished by sending messages on references. Following [13], an object can be a *primitive*, such as the literal 2, or an *instance* that is a combination of code

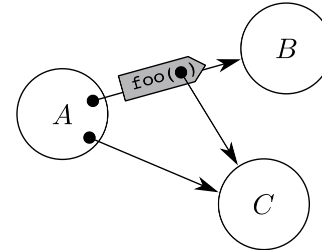

Figure 1. Connectivity by introduction: A sends B message `foo` with capability to C.

and state; we consider a process an instance of a program, in the same way that an object may be an instance of a class.

An object's state may include references to primitive objects or other instances. A reference to an instance is called a *capability*. The state of a system is often visualized using a *reference graph* whose nodes are objects and whose directed edges show references from one object to another. Because all communication in an OCap system is accomplished by sending messages on references, the only way for an object to send a message to another is if there is an edge in the reference graph. This limitation on communication is the fundamental source of isolation, confinement, and security in OCap systems. Because messages may include capabilities, the reference graph may change as capabilities are acquired and dropped.

In an OCap system an object may come to posses a capability only through the following methods:

- *Initial conditions*.
- *Parenthood*. When object A creates another object B, it is the only object in the system with the capability to B.
- *Endowment*. If object A has a capability to object C, then it can create another object B such that B is already 'endowed' with a capability to C.
- *Introduction*. If object A has capability to objects B and C, then A can give B (resp C) capability to C (resp B) by sending it a message containing the capability.

A direct consequence of these rules is that "only connectivity begets connectivity" – no message may occur between disjoint subgraphs of the reference graph. This is particularly important in building secure systems because at each 'snapshot' of the dynamic graph, it is directly clear who has access to what and the connectivity rules state how the graph may change.

Consider, for example, the graph where A has capability to B and C. The latter, C, has a read/write capability to file `/etc/passwd`. The only way B can gain *any* access to `/etc/passwd` is through A introducing C (or some forwarder) to B, as shown in Figure 1.

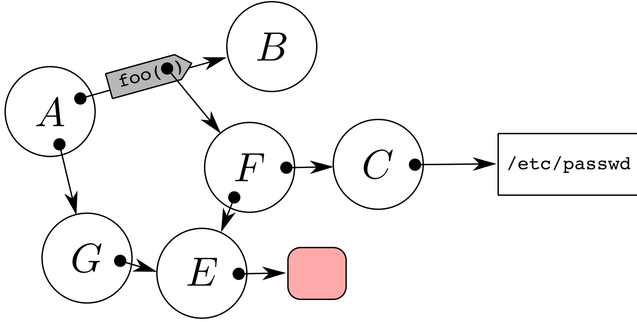

Figure 2. Revocable forwarder: A sends B message `foo` with capability to F, and, indirectly, revocable access to C. We use circular rectangles to denote mutable slots.

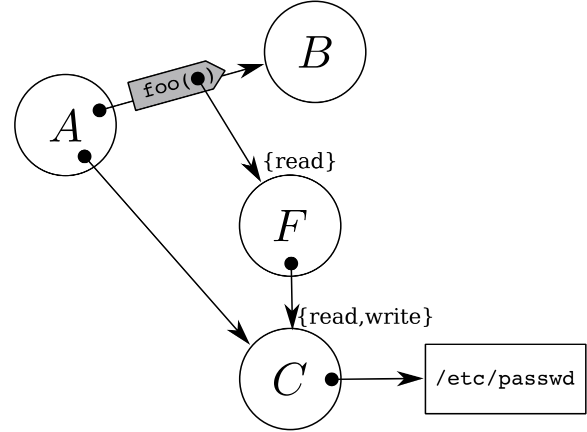

Figure 3. Attenuated forwarder: A sends B message `foo` with capability to F, and, indirectly, read-only access to C.

2.2 OCap patterns

OCap system designers and programmers have developed software patterns to solve standard recurring problems. We summarize three in this subsection.

2.2.1 Revocable forwarder pattern

Because a capability is simply a reference to an object, it is not clear how to revoke a capability, once granted. While a capability cannot be simply revoked, a revocation pattern allows an object to revoke the access granted by a capability. Though other patterns are conceivable, a common revocation pattern is Redell's caretaker pattern [13, 22]. Using the caretaker pattern A can give B revocable access to C as follows. First, A creates three objects:

- An enable slot object E with a mutable boolean slot.
- A gate G, with capability to E. Upon receiving message `toggle`, G toggles E's slot value.
- A *forwarder* F, with capability to E and C. Upon receiving message m, F checks the value of E's slot and only if it is true, it proceeds to forward m to C.

Now, instead of giving B capability to C, A gives B capability to F, as shown in Figure 2. While E's slot remains true B can send any messages to C, as if F is not in the reference path. However, A can revoke B's access to C at any time, simply by sending G the message `toggle`.

2.2.2 Attenuated and membrane forwarders

Suppose A wants to give B a capability to C, while attenuating (restricting) B's privilege to a certain message, e.g., `read`. As in the caretaker pattern, A creates a forwarder, F. In this case, F only accepts the message `read`, as shown in Figure 3. Of course, this read-only forwarder pattern can be combined with the caretaker pattern to grant a revocable, read-only capability.

Note that this pattern only restricts B's privilege to C through F. If C responds to B's `read` message with a capability to itself then B's authority will entail write access to `/etc/passwd` as well. Thus, if C cannot be trusted to respect the read-only capability, A must use a membrane forwarder, instead of a read-only forwarder. A membrane forwarder [13], wraps every capability in either direction and thus if C returns B a raw capability to itself, the membrane will wrap it, effectively making it read-only. As in the case of the read-only forwarder, the membrane pattern can be combined with the caretaker pattern to grant a revocable, *transitive* read-only capability. This is pattern is commonly called the *revocable membrane forwarder pattern*.

2.2.3 Right amplification pattern

The final pattern we consider provides right amplification using sealer/unsealer pairs [10, 29]. The basic premise of sealer/unsealer pairs is similar to public-/private-key pairs: A may use a sealer to "seal" another object \$ which C can unseal only if it has the corresponding unsealer. Though some OCap systems, such as E, support sealer/unsealer pairs as a primitive, others implement it using a pattern. Following [29], the sealer/unsealer pattern consists of

- A slot object S with a mutable value.
- A sealer with capability to S, that, upon receiving the message `seal` with argument \$, creates and returns a box B (with endowed capability to S). Box B, when invoked, writes \$ into S's slot.
- An unsealer U with capability to S, that, upon receiving the message `unseal` with argument B', clears S's slot, and invokes B'. After invoking B' it reads and returns the slot contents, if any.

Note that for the unsealer to return \$, it must invoke box B' such that $B = B'$. As an example use, consider Figure 4. In this case object A can send C a capability to \$ through the curious object D.

2.3 Introduction to Murφ

The Murφ model checker provides an input language that is used to define a nondeterministic finite-state system. The

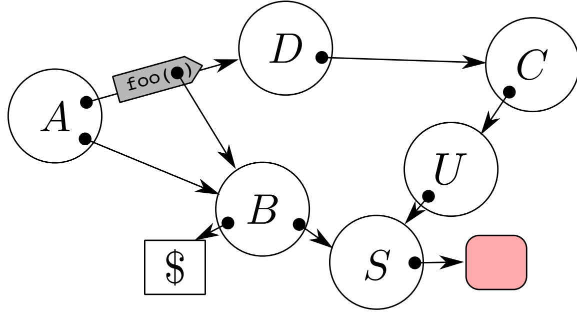

Figure 4. Sealer/unsealer pattern: A sends C a capability to \$ through D. Using a sealer, A sends D a capability to box B. Upon receipt of B, C can use unsealer U and box B (which writes \$ to S's slot) to retrieve \$
.

tool then enumerates all reachable states of the nondeterministic system and checks whether any specified invariants fail in any state. If a desired property fails, then Murφ produces the system *trace,* or sequence of actions from the initial state, leading to the error state. If no errors are found, then Murφ will explore all reachable states of the finite system and verify that none of them are erroneous. The main limit on the size of the checkable system arises from the space needed to store a hash table of all to the system states that have already been reached. If this table exceeds the capacity of main memory, then the state enumeration procedure slows substantially so that it becomes infeasible to complete the process. However, all of the models we formulated in this study were well within the memory bounds of the computers we used.

Finite-state verification tools such as Murφ have proven useful in the analysis of security protocols. For example, Murφ was successfully used in [16] to verify small protocols such as the Needham-Schroeder public key protocol, the Kerberos protocol, and the TMN cellular telephone protocol. In [17] Murφ was also successfully applied to the analysis of the SSL 3.0 handshake protocol using a "rational reconstruction" methodology which was adopted in [5] to analyze the 802.11i 4-Way Handshake. A more recent analysis [2] used Murφ to analyze DNSSEC.

3. Modeling OCap patterns in Murφ

We model several aspects of the OCap model using the Murφ verification tool. As in [19], we are interested in verifying OCap pattern properties in the context of sequential programming languages (PL) and distributed/operating systems (D/OS). As a number of capability systems in these domains already exist [4, 6, 8, 12, 25, 27, 28], we believe that our analysis is useful towards understanding the security properties of OCap patterns when used in complex systems. We note that our D/OS model reflects access in concurrent shared memory systems and therefore also appears applicable to programming languages based on the (fine-grained) actor model, including Joule [30], Erlang [1], and (a subset of) E [28].

3.1 Objects

We model objects in Murφ using a type `Object`, which is a record consisting of:

- ▷ *object state*: indicates if the object is *idle* or *blocking*.
- ▷ *c-list*: represents capabilities to objects in the graph. The c-list index designates the object, while the value indicates if the capability is *raw*, *wrapped*, i.e. attenuated, or *null*, i.e. no capability. Without loss of generality, in using the term *c-list* we refer to the non-null elements of the array.
- ▷ *u-list,* μ: boolean array of all system objects indicating possible unattenuated communication path with the current object. Communication from a membrane object should always be attenuated, hence we can use this list in verifying the pattern.
- ▷ *behavior description*: indicates how the object should behave, i.e. if is a forwarder, a gate, a membrane, an unsealer, etc.
- ▷ *behavior-related state*: state corresponding to the object's behavior, e.g. if the object is a revocable forwarder it must differentiate between the slot object and object it forwards to.

Our object model parallels the definition of an instance, which consists of state, including a list of capabilities (c-list), and code (behavior).

In modeling patterns and object communication, multiple objects are necessary. Hence, all the objects in our model are placed in a global array. This simplifies our invariant descriptions and moreover closely resembles a finite-length heap. We additionally note the object model can easily be extended, e.g. to support forwarding gates, see Section 4.2, or even object allocation/creation.

3.2 Object connectivity

In our model, object communicate using a shared network buffer. In the case of PL, only a single message can be placed on the network. Conversely, in the D/OS setting, multiple messages can be exchanged, in parallel; in our experiments, we varied the network size to buffers of length 2–4.

Objects exchange inter-object-communication messages (IOCMs). An IOCM is a 'network-layer' message, composed of several fields:

- ▷ *source,* $\mathcal{S}(m)$: object initiating the communication.
- ▷ *destination,* $\mathcal{D}(m)$: object intended to receive the message. Note that objects can only send messages to object's in their c-list.
- ▷ *message type,* m: indicates the transmitted object-layer message. Specifically, `call, return, grant, revoke` or `unseal`.

- ▷ *message arguments, a*: certain message types, e.g. `grant`, require arguments. Though extensible, in our model, a message argument can be a capability or boolean value (used in the revocable forwarder pattern when sending forwarder status).
- ▷ *wrapped flag*: boolean flag used to distinguish between raw and wrapped messages. A message is wrapped if it is sent from a membrane object, or from an object, e.g. forwarder, reacting to a received, wrapped message.
- ▷ *time stamp, τ_m*: integral value indicating the time the message was sent.
- ▷ *path, π_m*: boolean mask indicating the intermediate objects the message 'passed through'.

We use standard call-return semantics in modeling message exchanges. Specifically, an object in an idle state may invoke a capability from its c-list, i.e. send an IOCM (`call`, `grant`, etc.) to the corresponding object. Thereafter the object transitions into a blocking state, awaiting a `return` message.

3.3 OCap patterns

Given an overview of our object and communication model we now detail the specific patterns me modeled, their security properties, and our approach to verifying the invariance of these properties. We use $\mathcal{S}(m) \xrightarrow{m(a)} \mathcal{D}(m)$ to denote a (object-layer) message m with argument a being sent from source object $\mathcal{S}(m)$ to destination object $\mathcal{D}(m)$. We use $\xrightarrow{m(a)}{}^{+}$ to indicate a series of message being sent, the last of which is m, i.e. transitive (but not reflexive) relation.

3.3.1 Membrane forwarder pattern

The property we expect the membrane forwarder pattern to uphold is *transitive attenuation*. In other words, any message whose path contains a membrane forwarder should be wrapped.

To describe the invariant, consider an object O_i sending a message m over path π_m to object O_k. Recall that each IOCM contains the full message path π_m and a flag indicating if the message is wrapped, while every object O_k contains a u-list, μ_k. If O_k receives a raw message, we say the path from O_j to O_k for $j \in \pi_m$ is *unattenuated*, and update the O_k's u-list by setting the jth element, i.e. $\mu_k[j] := \texttt{true}$. Letting $\mathcal{M}(O_i)$ hold true if O_i is a membrane object and false otherwise. We express the property of the membrane forwarder pattern as invariant:

$$\forall O_i, O_k.\mathcal{M}(O_i) \land \neg\mu_k[i].$$

That is, for every membrane object O_i, the path from the membrane object to every other object O_k is attenuated (not unattenuated).

3.3.2 Revocable (membrane) forwarder pattern

Our model is used to verify both the revocable forwarder pattern, and the revocable membrane forwarder pattern. However, since verifying the latter simply requires the additional verification of the membrane forwarder pattern we focus on the revocable forwarder pattern.

We expect the revocable forwarder pattern to have the property of *no-forwarding post revocation*. Specifically, we expect the forwarder to not send any messages if the corresponding enable flag is false. To verify this property we, however, require a notion global time. Compared to the implicit time of [19], we explicitly model time as a simple shared counter that is incremented whenever a message is placed on the network. Because our model is relatively small, this does not result in any overflows or other modeling anomalies.

Recall that each IOCM contains a time stamp τ_m corresponding to the (time) counter value when message m was sent. Additionally, each enable slot E has a time stamp τ_E, stored in the object's behavior-related state. Specifically, τ_E is set to the time counter value at the slot modification time, i.e. when the flag is toggled.

Generalizing Figure 2, a revocable forwarder pattern contains a tuple $\langle G, E, F\rangle$ consisting of a gate, an enable slot, and a forwarder. Letting $\nu(E)$ correspond to the slot value, we can express the property of the revocable forwarder pattern in terms of the invariant:

$$\forall\langle G, E, F\rangle.\ \nexists m.\mathcal{S}(m) = F \land \neg\nu(E) \land \tau_E < \tau_m.$$

In other words, for every tuple with the slot value unset, there is no message placed on the network, by the forwarder, after the slot value was modified.

3.3.3 Right amplification pattern

For the sealer/unsealer pair pattern we expect *the unsealer to not be able to gain access to the slot contents, unless it has the (correct) box returned by the corresponding sealer*.

Following Figure 4, the modeled right amplification pattern contains a tuple $\langle B, U\rangle$, consisting of a sealer-returned box and a corresponding unsealer. Letting $\nu(B)$ correspond to the box slot value, we can express the right amplification pattern using invariant:

$$\forall\langle B, U\rangle.\ \nexists O, B'.B' \neq B \land O \xrightarrow{\texttt{unseal}(B')} U \xrightarrow{\texttt{return}(\nu(B))}{}^{+} O$$

The invariant simply states that for every tuple $\langle B, U\rangle$ there is no object O, and fake box B', such that O's invocation of the unsealer with the 'fake' box returns the value store in the actual box slot B. Although the property can be further generalized to multiple sealer-returned boxes, we limit our specification for clarity and without loss of generality.

3.4 Model limitations

As in [19], our model is limited in several ways. First, we do not model object creation. This has the implication that

the dynamics of the access graph only changes based on the rule of introduction. As noted in [19, 26], this is not a severe limitation since an object's behavior can be defined to aggregate the behavior of the objects it creates. Nevertheless, we designed our model to allow for the addition of object creation. Specifically, since every object in our model resides in a global array (akin to the program heap), the `Object` type can be extended to add a flag, `allocated`, that is used to indicate whether the object has already been created. To model object creation, then, an object takes possession of a free object (one whose `allocated` flag is not set) in the array, sets the `allocated` flag and any other flags and capabilities (in its c-list) to define its behavior. This approach directly extends to creation of multiple objects that can be used in patterns such as revocation.

Second, for each pattern we verify, we initialize our system with a small access graph consisting of some already-interconnected objects. Specifically, the objects required to realize the pattern, e.g. gates and forwarders used in the selective revocation pattern, are part of the initial access graph having 'acquired' their capabilities by initial conditions. For example, in modeling the selective revocable pattern our initial system consists of the access graph of Figure 2, in addition to A and B having capabilities to other arbitrary objects. This limitation is a direct consequence of the previous, no object creation, limitation. Extending the model to support object creation further removes this limitation.

4. Verification results

We used the Murφ verification tool to check the OCap model and patterns detailed in Section 3. Additionally, addressing the found vulnerabilities, detailed in this section, we model and verify several new patterns (that address found vulnerabilities) in the same manner. For every pattern, we ran the verifier in the PL and D/OS setup, modifying the network size, and varying initial access graphs.

4.1 Original OCap patterns

Our Murφ verification results confirm the previously found vulnerabilities of [19], in addition to revealing a denial-of-service (DOS) vulnerability in the revocable forwarder pattern. Our model checking results are detailed below.

4.1.1 Membrane forwarder pattern

As expected, Murφ did not find any trace that violated the membrane forwarder pattern in either the PL or D/OS setting. Although we found no vulnerabilities for this pattern, we stress that this pattern holds (up to our model) for OCap systems; implementing membrane-like patterns in other capability system does not guarantee transitive attenuation. In Section 5.2 we show a violation of a transitive read-only property for the file system TahoeLAFS [31].

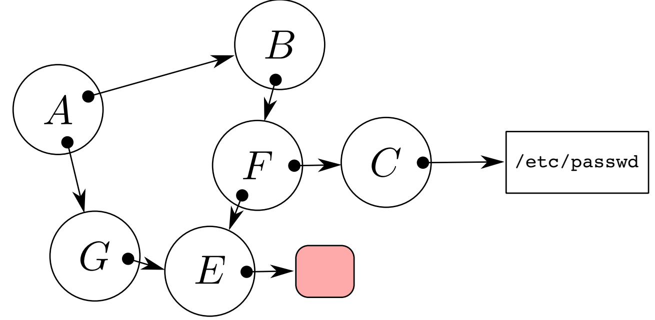

Figure 5. Revocable (membrane) forwarder: F is a (membrane) forwarder to C, G is a gate, and E is the enable slot object used by the two.

4.1.2 Revocable (membrane) forwarder pattern

In the context of sequential programming languages (PL), Murφ did not produce any traces that violate the revocable forwarder pattern, or revocable membrane forwarder pattern. However, in the context of concurrent shared memory systems (D/OS) we found that the revocable membrane pattern has a time-of-check-to-time-of-use (TOCTTOU) vulnerability.

Murφ found a violating trace for the access graph[1] of Figure 5. We use the message type `isEnabled` to denote a `call` message sent by forwarder F to the enable slot object E, to read its value. Using this simplification, and starting with the enable-slot's value set to true, the sequence of message calls violating the invariant, as found by Murφ, are:

1.$B \xrightarrow{\texttt{call}(\cdot)} F$

2.$F \xrightarrow{\texttt{isEnabled}()} E$

3.$E \xrightarrow{\texttt{return}(true)} F$

4. $A \xrightarrow{\texttt{toggle}()} G \rightarrow E$

5. $E \rightarrow G \xrightarrow{\texttt{return}()} A$

6.$F \xrightarrow{\texttt{call}(\cdot)} C$

In this trace, forwarder F received the enabled-status in step 3, but did not use the value (returned by E) until step 6. At the time of use, the F's copy of the value was stale, since the slot object was toggled in the steps following the check. This TOCTTOU vulnerability is equivalent to the vulnerability found in [19] using FDR.

We also modified the invariant of the revocable (membrane) forwarder pattern, as given in Section 3.3.2, to re-

[1] We note that in the initial access graph of our actual model, B did not have a capability to F. However, since the first steps of the violating trace consisted of A granting B capability to F, we omit this step for simplicity and focus on the subsequent access graph graph (in which B already has a capability to F).

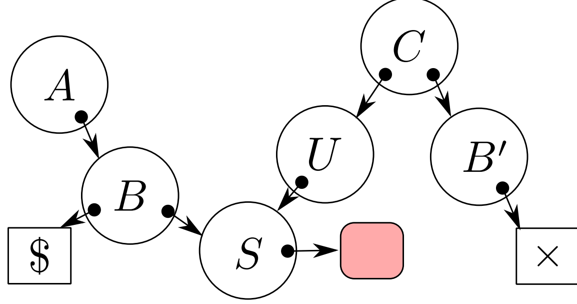

Figure 6. Sealer/unsealer pattern initial dynamic graph. A has a capability to box B which has capabity to \$ and slot object S, that is common to the unsealer U. C has capability to box B'.

move the notion of time. This alternative invariant

$$\forall \langle G, E, F\rangle.\ \nexists m.\mathcal{S}(m) = F \wedge \neg\nu(E)$$

has no notion of message send time. Hence, even if the forwarder sends a message before the enable slot is modified, if the message is received after the slot has been toggled the invariant is violated. The following Murφ trace

$$\begin{array}{lll}
1. A \xrightarrow{\texttt{toggle()}} G \rightarrow E & & \\
2. & & B \xrightarrow{\texttt{call}(\cdot)} F \\
3. & & F \xrightarrow{\texttt{isEnabled()}} E \\
4. & & E \xrightarrow{\texttt{return}(true)} F \\
5. E \rightarrow G \xrightarrow{\texttt{return()}} A & & F \xrightarrow{\texttt{call}(\cdot)} C
\end{array}$$

highlights the revocable (membrane) forwarder pattern's susceptibility to DOS attacks. In this case, the `toggle` message is sent by A in the first step, but not handled until the last step. Assuming the slot handles messages in a first-in-first-out fashion, this vulnerability is realizable if B is able to delay the delivery of A's messages, e.g. by flooding the network in a distributed system, or meddling with the operating system scheduler. This vulnerability was not found in [19].

4.1.3 Right amplification pattern

As for the revocable (membrane) forwarder pattern, Murφ found a violating trace only in the D/OS context for the sealer/unsealer pair pattern. Specifically, in a D/OS setting an unsealer is able to retrieve the slot contents without using, but colluding with, the corresponding box.

Murφ found a violating trace for dynamic access graph shown in Figure 6, where B' is a box object that C creates by sealing an arbitrary object; note that B and B' are created by different sealers. The violating trace, which is equivalent to the attack found in [19], is:

$$\begin{array}{lll}
1. C \xrightarrow{\texttt{unseal}(B')} U & & \\
2. U \xrightarrow{\texttt{clear()}} S & & \\
3. S \xrightarrow{\texttt{return()}} U & & \\
4. U \xrightarrow{\texttt{call()}} B' & & \\
5. B' \xrightarrow{\texttt{return()}} U & & \\
6. & & A \xrightarrow{\texttt{call()}} B \\
7. & & B \xrightarrow{\texttt{return()}} A \\
8. U \xrightarrow{\texttt{read()}} S & & \\
9. S \xrightarrow{\texttt{return(\$)}} U & & \\
10. U \xrightarrow{\texttt{return(\$)}} C & &
\end{array}$$

We observe that the premise for the found vulnerability is that C can invoke the unsealer with any box, e.g. B', and after the unsealer has cleared the slot, if the correct box B is invoked, the slot is filled with \$. The unsealer then proceeds to read the box contents and return it to the original caller (having assumed it was B' that filled the slot). We note that our model does not explicitly model the interaction with the slot as we shown; specifically, `clear` and `read` are inline calls modifying a shared variable, and not messages sent to an object. Nevertheless, this is only an implementation simplification that does not alter the modeling semantics.

4.2 Proposed OCap patterns

Having found previously detailed vulnerabilities, we believe that our model checking is sound and suitable for verifying other patterns. Hence, we address the vulnerabilities detailed in Section 4.1 by proposing and verifying alternative, but behaviorally equivalent, patterns.

4.2.1 Revocable (membrane) forwarder pattern

Addressing the TOCTTOU vulnerability of the revocable (membrane) forwarder pattern (see Section 4.1.2) we propose the slightly modified pattern shown in Figure 7. This pattern consists of a 'special' forwarder F, which takes any message m with argument a and sends the forwarding gate F_G a `forward` message with $m(a)$ as the argument. A *forwarding gate*, in this case F_G, reacts to two messages: `toggle`, which is used to revoke access by toggling a local enable flag, and `forward` which is used to conditionally forward the received arguments. Continuing with Figure 7, F_G forwards $m(a)$ to object C, if the local enable flag is set. Exploring 13,720,000 states of the proposed pattern, Murφ found no trace violating the pattern invariant.

Note, however, that the proposed pattern, like the original, is susceptible to DOS attacks. To make the pattern DOS-

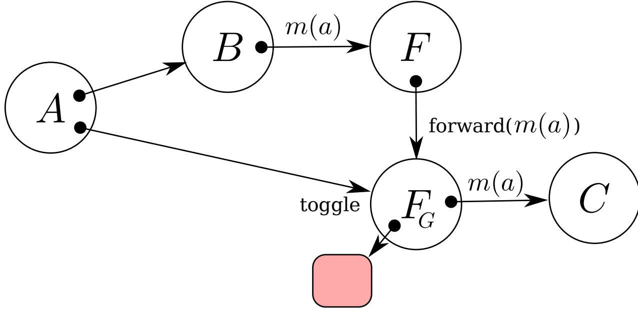

Figure 7. Proposed revocable forwarder: F_G is a forwarder gate to C, with a local enable slot. F is a special forwarder that 'lifts' messages into the `forward` function.

invulnerable, our proposed pattern need only be modified to combine A's behavior with that of the forwarding gate F_G into a single object.

4.2.2 Right amplification pattern

As in the case of the revocable forwarder, we propose a modification to the sealer/unsealer pair pattern. Following Figure 4, the modified sealer/unsealer pattern consists of

- A slot object S.
- A sealer with capability to S, that, upon receiving the message `seal` with argument $, creates and returns a box B (with endowed capability to S). Box B, when invoked, writes $ and capability to itself into S's slot.
- An unsealer U with capability to S, that, upon receiving the message `unseal` with argument B', clears S, and invokes B'. After invoking B', if the slot is not empty, the unsealer verifies that the capability read from the slot is that of the invoked box, i.e. $B = B'$. If the check succeeds, the unsealer returns the slot contents, otherwise fails, e.g by throwing an exception.

As for the original pattern, we used the Murφ model checker to verify the pattern; we found no trace violating invariant.

5. Model application

In this section, we relate our Murφ model to implemented systems; specifically, we detail attacks on OCap patterns implemented using the asynchronous subset of E, and an attack on TahoeLAFS's transitive read-only system property. For the former, we also present E code implementing of our alternative robust patterns.

Although concurrency model of many OCap languages and systems, e.g. Joe-E and Waterken [8], do not correspond to the verified concurrent shared memory model (D/OS), the vulnerabilities found in [19] (and confirmed by our Murφ model) are applicable to systems following the (fine-grained) actor model. As previously mentioned, languages implementing this model includes Joule, Erlang, and a subset of E. Our attacks do not extend to the full E.

5.1 E

E is a dynamically typed object capability language. E's semantics closely resemble those of Scheme, with the added notion of message sending and method dispatch commonly found in object oriented languages, such as Smalltalk. Unlike other languages, using a cryptographic capability protocol [9, 10], E can extend the object reference graph beyond a single machine, securely and transparently. We describe E using examples that explore the concepts necessary to understand our implementations of the OCap patterns; the interested reader is referred to [28] for additional details.

For simplicity, consider the implementation of an up/-down counter. Such a simple counter can be defined in E as follow:

```
def counterMaker(initialValue) {
  var counterVal := initialValue
  def counter {
    to inc() { counterVal := counterVal + 1 }
    to dec() { counterVal := counterVal - 1 }
    to getVal() { return counterVal }
  }
  return counter
}
```

Both counterMaker and counter define objects, the former returning a closure containing a new counter with an initial value. The counter's methods increment, decrement and return the value of the mutable variable, counterVal, when the counter object receives the inc, dec, and getVal messages, respectively. For example, the following E program creates a counter object with an initial value of 3, increments the object twice and prints its value:

```
def ctr := counterMaker(3)
ctr.inc()
ctr.inc()
def curVal := ctr.getVal()
println('Counter is $curVal')
```

Similar to Java's Proxy [24], E allows objects to receive messages that do not match the object's static API using a **match** clauses. For example, the above counter can alternatively be defined as follows:

```
def counterMaker(initialValue) {
  var counterVal := initialValue
  def counter {
    match [verb, args] {
      if(verb == "inc") {
        counterVal := counterVal + 1
      } else if(verb == "dec") {
        counterVal := counterVal - 1
      } else if(verb == "getVal") {
        counterVal
      }
    }
  }
  return counter
}
```

Note that in the **match** clause, the verb corresponds to the matched message name, while args contains the message arguments. Given an object the **E.call** method can be used to send messages matched by the **match** clause. In this example, we further highlight that **return** is often implicit

(**return** is a form of Ejector, the details of which can be found in [28]).

Finally, to use E in a distributed setting it is necessary to extend local, live, references to network-meaningful (and reliable) references. For example, to make the previous ctr reference usable by other E processes, we convert it to a "sturdy reference":

```
def ctrSRef := makeSturdyRef.temp(ctr)
```

Sturdy references can then be converted to URI's that are shared among E processes. Similarly, URI's can be converted to sturdy references which can be further converted back to live references:

```
def ctrLive := ctrSRef.getRcvr()
```

Sending messages to remote objects is, however, implemented differently. Specifically, E implements a promise-pipeline for sending asynchronous messages to remote objects. Hence, to increment the remote counter corresponding to ctrLive it is necessary to use the 'eventually' operator ← as follows:

```
ctrLive ← inc()
```

Conversely, to get the value of the remote counter we wrap the message call in a **when**−**catch** clause:

```
def newVal := ctrLive ← getVal()
when(newVal) → {
  println('Counter is $curVal')
} catch err {
  println('Failed with $err')
}
```

As newVal is only a promise (that the value will eventually arrive) we cannot use it until it is resolved, or alternatively is broken. Similar to a **try**−**catch** blocks, a **when**−**catch** block is used to handle the success/failure of an event (in the latter case, promise resolution). As we are interested in applying our model to concurrent systems, we focus only on the asynchronous and distributed subset of E; in the remaining sections "E" refers to this particular subset.

5.1.1 Revocable forwarder pattern

We implemented the revocable forwarder pattern using E, based on the original revocation implementations of [9, 13]. The implementation is shown in Listing 1. In this listing, EnableSlotObj returns an enable slot object with which the remaining code interacts as if the slot is a remote object. The makeCaretaker creates a revocable forwarder to target and a gate that can be used to change the slot value. Note that both the forwarder and gate use the eventually operator ← when sending messages (asynchronously) to the slot. Additionally, the forwarder only forwards (using **E.call**) the received message (verg with arguments args) to target when the promise (of the slot enable value) is resolved and is **true**.

An example using the pattern, similar to the scenario of Figure 1, is shown in Listing 2. In the example bob is given a capability to a clarice-forwarder, after which

```
def EnableSlotObj() {
  var enabled := true
  def makeSlot {
    to toggle () { enabled := !enabled }
    to isEnabled () { return enabled }
  }
  return makeSturdyRef.temp(makeSlot).getRcvr()
}

def makeCaretaker(target) {
  def slot := EnableSlotObj()
  def forwarder {
    match [verb, args] {
      def doFwd := slot ← isEnabled()
      when(doFwd) → {
        if (doFwd) {
          E.call(target, verb, args)
        } else {
          throw("Forwarding disabled")
        }
      } catch e {
          println('Could not get slot value: $e')
      }
    }
  }
  def gate {
    to toggle() { slot ← toggle() }
  }
  return [forwarder, gate]
}
```

Listing 1. E implementation of the revocable forwarder pattern.

```
def clariceCreator(x) {
  var cntr := x
  def clarice {
    to printMe() {
      cntr := cntr+1
      println('Clarice counter: $cntr')
    }
  }
  return clarice
}

def bobCreator() {
  var clarice := null
  def bob {
    to foo(a) { clarice := a }
    to bar() { clarice.printMe() }
  }
  return bob
}

def bob := bobCreator()
def clarice := clariceCreator(41)
def [f,g] :=makeCaretaker(clarice)

bob.foo(f)
bob.bar()
g.toggle()
bob.bar()
```

Listing 2. An example use the revocable forwarder pattern. When executed the program prints "Clarice counter: 42" and then throw an exception.

```
def forwardingGate(target) {
  var enabled := true
  def fgate {
    to toggle() { enabled := !enabled }
    to forward(verb, args) {
        if (enabled) {
          E.call(target, verb, args)
        } else {
          throw("Forwarding disabled")
        }
      }
  }
  return makeSturdyRef.temp(fgate).getRcvr()
}

def makeCaretaker(target) {
  def fgate := forwardingGate(target)
  def forwarder {
    match [verb, args] {
      fgate ← forward(verb, args)
    }
  }
  return [forwarder, fgate]
}
```

Listing 3. E implementation of the proposed revocable forwarder pattern.

the object invokes the capability (by sending the printMe message); once the capability is revoked (g.toggle()), the following attempt to invoke the capability fails. Similar to this example, but using three distributed processes for the target object, gate, and forwarder we were able to reproduce the Murφ violation trace (see Section 4) in which the slot is disabled, but the forwarder still sends messages. Listing 3 shows our alternative pattern, corresponding to the Murφ model in Figure 7 which addresses the original vulnerability. Implementing this pattern in full E can be accomplished by simply modifying the forwarding gate to return fgate directly, and modifying the forwarder to use the local method invocation operator, instead of the eventually operator.

5.1.2 Right amplification pattern

Listing 4 shows the implementation of the sealer/unsealer pair pattern described in Section 2. We base our implementation on the full E implementation of [29]. We note that although the implementation seems quite complex, the nested when-catch blocks are only used to sequence the slot clearing, box invocation, and slot read. Additionally, in the unsealer implementation, we use **Ref**.**promise**() to create a promise value which the unsealer resolves if the slot is read, or ‘smashes’ in case of failure.

An example using the right amplification pattern is shown in Listing 5. This example also includes an attempt to circumvent the pattern property by invoking unseal with a box created by a sealer that does not directly correspond to the unsealer. Using two processes we were able to reproduce the attack of Section 4, as found by Murφ. In this attack, a process is used to create two sealer/unsealer pairs, after which the sealers are used to seal distinct objects, effectively creating a ‘real’ and ‘fake’ box (with respect to one of the pairs).

```
def SlotObj(defaultVal) {
  var content := defaultVal
  def makeSlot {
    to clearSlot() { content := defaultVal }
    to readSlot() { return content }
    to writeSlot(cont) { content := cont }
  }
  return makeSturdyRef.temp(makeSlot).getRcvr()
}

def makeBrandPair() {
    def noObject{}
    def slot := SlotObj(noObject)
    def makeSealedBox(obj) {
        def box {
            to shareContent() {
              slot ← writeSlot(obj)
            }
        }
        return box
    }
    def sealer {
        to seal(obj) {
          return makeSealedBox(obj)
        }
    }
    def unsealer {
      to unseal(box) {
         def [pVal, resolvVal] := Ref.promise()
         when(slot ← clearSlot()) → {
           when(box ← shareContent()) → {
             def val := slot ← readSlot()
             when(val) → {
               if(val == noObject) {
                 throw("Invalid box")
               }
               resolvVal.resolve(val)
             } catch rErr {
               resolvVal.smash('Did not read: $rErr')
             }
           } catch sErr {
             resolvVal.smash('Did not share: $sErr')
           }
         } catch cErr {
           resolvVal.smash('Did not clear: $cErr')
         }
         return pVal
      }
    }
    return [sealer, unsealer]
}
```

Listing 4. E implementation of the right amplification pattern.

A separate process is then used to remotely invoke the real box, while the unsealer is remotely invoked with the fake box. In negligible time, the unsealer returned the slot contents.

As in the revocable forwarder pattern case, we implemented our proposed pattern, shown in Listing 6. The new pattern differs from the original only minimally: we write to the slot a Pair object that holds the box capability and original box content, and add the additional check to unseal, as discussed in Section 4.

5.2 TahoeLAFS

The Tahoe Least Authority File System (TahoeLAFS) [31] is a file system that uses capabilities for access control. Specif-

```
def [s,u] := makeBrandPair()
def box := s.seal("secret")

def val := u.unseal(box)
when(val) → {
  println(`First unsealed to: $val`)
} catch e {
  println("First failed to unseal")
}

def [s2,u2] := makeBrandPair()
def fakeBox := s2.seal("fake")

def val2 := u.unseal(fakeBox)
when(val2) → {
  println(`Second unsealed to: $val2`)
} catch e {
  println("Second failed to unseal")
}
```

Listing 5. An example use the right amplification pattern. When executed the program prints "First unsealed to: secret", followed by "Second failed to unseal".

ically, TahoeLAFS uses a URI consisting of random (cryptographically generated) string as a capability that uniquely identifies and designates a file/directory; following the capability model, a capability does not separate authority and designation, nor is it forgeable. However, the TahoeLAFS model is not an OCap model. Hence, properties such as the membrane forwarder that we verified to hold for OCap systems (up to our model) do not necessarily hold for other capability models; an observation also made in [11].

Furthermore, TahoeLAFS claims to have

> ... the property of transitive read-only – users who have read-write access to the directory can get a read-write-cap to a child, but users who have read-only access to the directory can get only a read-only-cap to a child. It is [their] intuition that this property would be a good primitive for users to build on, and patterns like this are common in the capabilities community ... [31]

The membrane pattern can be used to enforce the property of transitive read-only, however (non-membrane) attenuated forwarders cannot. Consider the simple setup in which A has a read-write capability to directory D, and B has a read-only capability to the directory. If the system property holds, B *should* only have transitive read-only access for files stored in D. However, an attack is directly apparent given that TahoeLAFS does not have a method of distinguishing between data and capabilities (and thus cannot implement the membrane pattern). Specifically, A can create a file F in directory D consisting of the read-write capability to D (or any subdirectory). B, invoking the read-only capability can read file F at which point it acquires a read-write capability to D—highlighting a system-property violation, which we confirmed on TahoeLAFS public test grid.

```
def Pair(al,bl) {
  var a := al
  var b := bl
  def mkPair {
    to fst() { return a }
    to snd() { return b }
  }
  return mkPair
}
def SlotObj(defaultVal) {
  var content := defaultVal
  def makeSlot {
    to clearSlot() { content := defaultVal }
    to readSlot() { return content }
    to writeSlot(cont) { content := cont }
  }
  return makeSturdyRef.temp(makeSlot).getRcvr()
}

def makeBrandPair() {
    def noObject{}
    def slot := SlotObj(noObject)
    def makeSealedBox(obj) {
        def box {
            to shareContent() {
              slot <- writeSlot(Pair(obj,box))
            }
        }
        return box
    }
    def sealer {
        to seal(obj) {
          return makeSealedBox(obj)
        }
    }
    def unsealer {
      to unseal(box) {
         def [pVal,resolvVal] := Ref.promise()
         when(slot <- clearSlot()) -> {
           when(box <- shareContent()) -> {
             def val := slot <- readSlot()
             when(val) -> {
               if((val==noObject) ||
                  (box != val.snd())) {
                 throw("Invalid box")
                }
               resolvVal.resolve(val.fst())
             } catch rErr {
               resolvVal.smash(`Did not read: $rErr`)
             }
           } catch sErr {
             resolvVal.smash(`Did not share: $sErr`)
           }
         } catch cErr {
           resolvVal.smash(`Did not clear: $cErr`)
         }
         return pVal
      }
    }
    return [sealer, unsealer]
}
```

Listing 6. E implementation of the proposed right amplification pattern.

6. Conclusions

We modeled several Object Capability patterns using the Murφ verification tool. Our results confirm Murray's previous work in finding a vulnerability in the revocable forwarder pattern and sealer/unsealer pattern when used in a concurrent shared memory systems. We further implement attacks exploiting the vulnerabilities in E using fine grained distributed objects. Addressing the patterns' vulnerabilities, we propose alternative patterns for which we did not find any system-property violations.

References

[1] J. Armstrong. *Concurrent programming in ERLANG*. Prentice Hall PTR, 1996. ISBN 013508301X.

[2] J. Bau and J. C. Mitchell. A security evaluation of dnssec with nsec3. In *Network and Distributed Systems Securith (NDSS)*, 2010.

[3] D. Dill. The Murphi verification system. In *Proceedings of the 8th International Conference on Computer Aided Verification*, pages 390–393. Springer-Verlag, 1996. ISBN 3540614745.

[4] N. Hardy. KeyKOS architecture. *ACM SIGOPS Operating Systems Review*, 19(4):8–25, 1985. ISSN 0163-5980.

[5] C. He and J. C. Mitchell. Analysis of the 802.11i 4-Way Handshake. In *WiSe '04: Proceedings of the 3rd ACM Workshop on Wireless Security*, pages 43–50, New York, NY, USA, 2004. ACM. ISBN 1-58113-925-X. doi: http://doi.acm.org/10.1145/1023646.1023655.

[6] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[7] S. Maffeis, J. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *2010 IEEE Symposium on Security and Privacy*, pages 125–140. IEEE, 2010.

[8] A. Mettler, D. Wagner, and T. Close. Joe-E: A security-oriented subset of Java. In *17th Network & Distributed System Security Symposium*, 2010.

[9] M. Miller and J. Shapiro. Paradigm regained: Abstraction mechanisms for access control. *Advances in Computing Science–ASIAN 2003*, pages 224–242, 2003.

[10] M. Miller, C. Morningstar, and B. Frantz. Capability-based financial instruments. In *Financial Cryptography*, pages 349–378. Springer, 2001.

[11] M. Miller, K. Yee, J. Shapiro, et al. Capability myths demolished. Technical report, Johns Hopkins University, Tech. Rep, 2003.

[12] M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. `http://google-caja.googlecode.com/files/caja-spec-2008-01-15.pdf`, 2008.

[13] 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.

[14] M. S. Miller, K.-P. Yee, and J. Shapiro. Capability myths demolished. Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003.

[15] J. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Mur/spl phi. *sp*, page 0141, 1997. ISSN 1540-7993.

[16] J. C. Mitchell, M. Mitchell, and U. Stern. Automated Analysis of Cryptographic Protocols Using Murϕ. In *IEEE Symposium on Security and Privacy*, pages 141–151, 1997. URL `citeseer.ist.psu.edu/mitchell97automated.html`.

[17] J. C. Mitchell, V. Shmatikov, and U. Stern. Finite-State Analysis of SSL 3.0. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium, 1998*, pages 16–16, Berkeley, CA, USA, 1998. USENIX Association.

[18] Mitre. Common vulnerabilities and exposures. `http://cve.mitre.org/cve/cve.html`, 2011.

[19] T. Murray. Analysing object-capability security. In *Proceedings of the Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (FCS-ARSPA-WITS'08)*, 2008.

[20] T. Murray. *Analysing the security properties of object-capability patterns*. PhD thesis, University of Oxford, 2010.

[21] T. Murray and G. Lowe. Analysing the information flow properties of object-capability patterns. *Formal Aspects in Security and Trust*, pages 81–95, 2010.

[22] D. Redell and D. Redell. Naming and protection in extendable operating systems. 1974.

[23] J. A. Rees. A security kernel based on the lambda-calculus. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1996.

[24] K. Renaud and H. Evans. Engineering Java Proxy Objects using Reflection. In *Proceedings of the NET. OBJECTDAYS*. Citeseer, 2000.

[25] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Kiawa Island, SC, December 1999. ACM.

[26] A. Spiessens. *Patterns of safe collaboration*. PhD thesis, Université catholique de Louvain, February 2007.

[27] M. Stiegler. Emily: A high performance language for enabling secure cooperation. In *Creating, Connecting and Collaborating through Computing, 2007. C5'07. The Fifth International Conference on*, pages 163–169. IEEE. ISBN 0769528066.

[28] M. Stiegler. The E language in a walnut. `http://www.skyhunter.com/marcs/ewalnut.html`, 2000.

[29] M. Stiegler. A picturebook of secure cooperation. Presentation, 2004. `http://erights.org/talks/efun/SecurityPictureBook.pdf`.

[30] E. Tribble, M. Miller, N. Hardy, and D. Krieger. Joule: Distributed application foundations. Technical report, Agorics Inc., Los Altos, 1995.

[31] Z. Wilcox-O'Hearn and B. Warner. Tahoe: the least-authority filesystem. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, pages 21–26. ACM, 2008.