Capability confinement by membranes

Yves Jaradin, Fred Spiessens and Peter Van Roy
Université catholique de Louvain
{yjaradin, fsp, pvr}@info.ucl.ac.be

March 22, 2005

Abstract

In capability systems it can be hard to assert confinement from static analysis only. In this paper we propose *membranes* as an extension for capability secure languages to help ensure confinement at runtime. We apply the extension to a subset of the multi-paradigm language Oz and provide the operational semantics of the extended language.

1 Problem context

Secure systems aim at providing privacy of information and confinement of services. Systems based on Access Control Lists (ACLs) rely on a special module called the "reference monitor", to force an access-control policy. Designation of a resource doesn't imply access to that resource.

Systems based on Capabilities [MSC$^+$01] have no reference monitor : designation is equivalent to access. The language runtime ensures that references are unforgeable. References to resources and services are called capabilities. A process X that has access to a service C can transfer that access to another process Y, only if X has access to Y (via capability). The programmer implements the access policy directly, by carefully refining and distributing capabilities amongst the processes on a need-to-use base. Capability secure languages [MSC$^+$01, SV05, Ree96] are specially designed to assist the programmer in this task.

The major drawback of ACL systems is their inherent vulnerability to the confused deputy attack. This is a simple attack – explained in [Har89, MS03, SV05] – that can effectively counter the reference monitor's attempts to confine the services provided by a server. Consider a server that uses access rights delegated by a client, to perform a service on behalf of that client. Such a server is called a deputy. A rogue client able to designate a resource only the deputy is allowed to use, can lure the deputy into using that resource on its behalf. For instance, a deputy that writes information into a client provided file, can be tricked into overriding its own files.

Because capabilities indivisibly combine the right to use a service with the designation of that service, confused deputies can be avoided. The trouble is that, even in carefully designed capability secure languages, it is not obvious for a programmer to check if his/her program is conform to the access policy he/she is supposed to implement. We see two approaches to solve this problem. One is the introduction of tools for static analysis and model checking. This approach is being investigated by one of the authors. A second one is the introduction of a new language construct, compatible with the capability approach, that dynamically guards confinement strategies. This paper concentrates on the second approach, and introduces a language construct called "membrane".

Inevitably, such constructs re-separate designation from the ability to use the designated resource or service. We therefor only recommend membranes as an additional fall-back, not as a replacement for careful design and implementation. Being aware of the risk of re-introducing confused deputies , we are confident that we will be able to identify safe abstractions and patterns for using membranes, that avoid this vulnerability.

We investigated the addition of language constructs for confinement by extending the Mozart[Con] implementation of the multi-paradigm language Oz. The name for our construct was inspired by the interception mechanism of the Kell calculus[BS03].

2 Design guidelines

In this section we elaborate on the security principles described by Saltzer and Schroeder in [SS75] that need specific attention in the context of confinement.

- Simplicity and ease of use. Programmers should not require special skills to use membranes. The confining effects of membranes should be easy to infer. Therefor the semantic model will be kept as simple as possible, and seamlessly integrate with the semantics of the language.
- Fail-safety. Careful distribution of capabilities on a need-to-use base is the preferred way to confine resources. Membranes are a mechanism to *enhance* the fail-safety of capability systems rather than to *replace* capability-based confinement.
- Fine granularity. To provide the finest possible granularity, membranes will confine individual unforgeable references.
- Dynamic control. The confinement boundary of membranes can be adjusted at runtime.

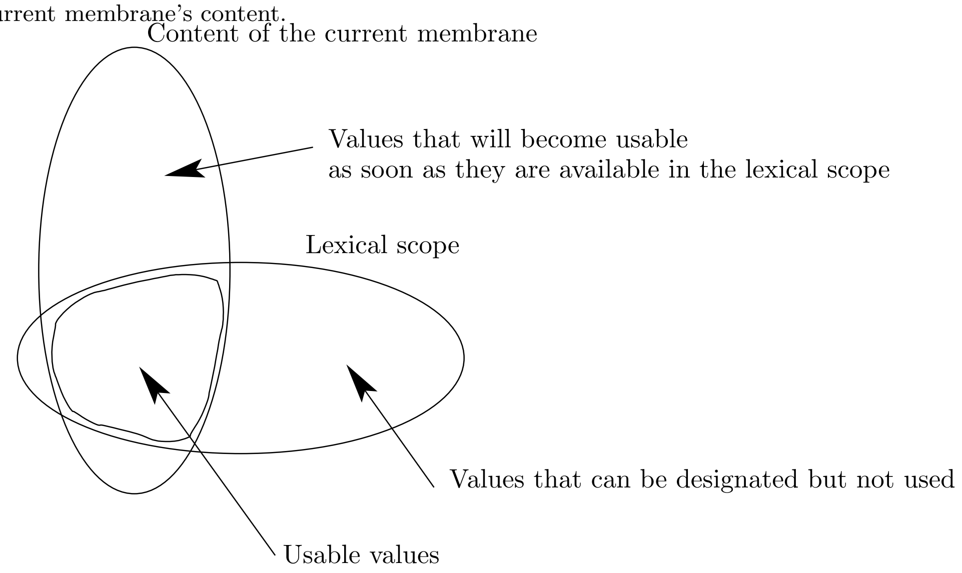
3 Membranes

A membrane is an execution context associated with a set of token values, called the membrane's content. Token values are unforgeable and therefore unique values. An instruction executed in the context of a membrane can only use references to values in the set associated with that membrane.

When a token value is *created*, it is automatically placed in (the content of) the membrane of the creating instruction. Values that were not created inside the membrane context have to be explicitly *exported* to (the content of) the membrane to become available.

Every instruction is executed in the context of exactly one membrane, by default the current one. To run an instruction in another membrane, it has to be explicitly *executed* in the context of that membrane.

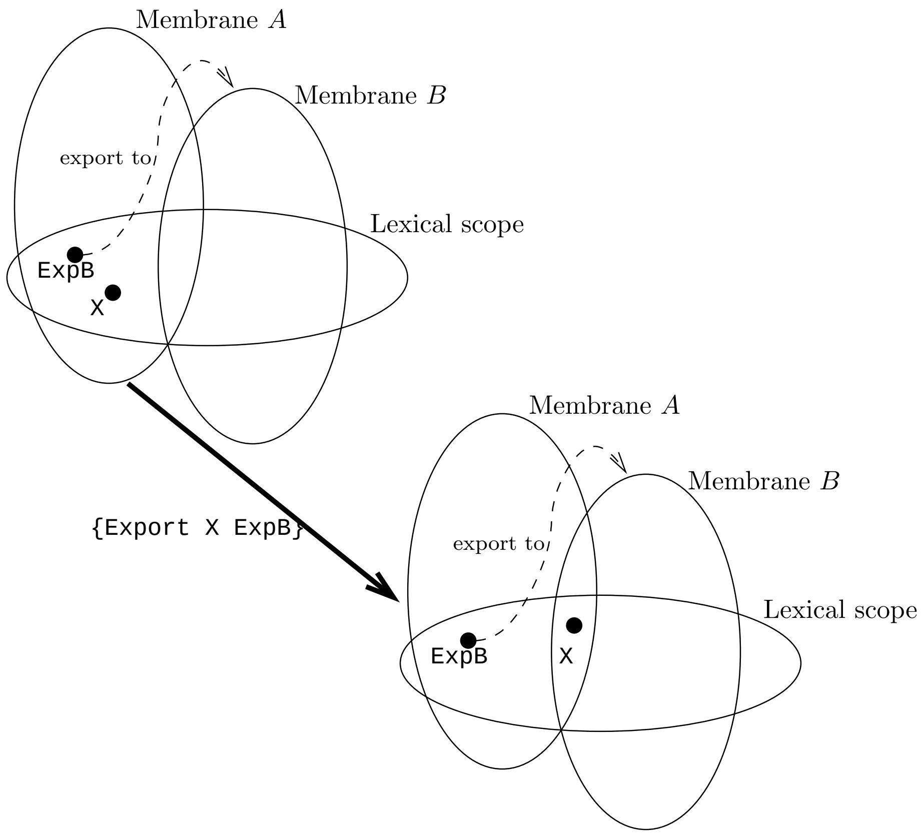
Only the values in the static scope can be designated and only the values in the membrane context can be accessed. An instruction can therefore only use values in the intersection of the static scope and the current membrane's content.


Content of the current membrane
Values that will become usable
as soon as they are available in the lexical scope
Lexical scope
Values that can be designated but not used
Usable values

There are three new language primitives directly involving membranes: membrane creation, exporting a token value to a membrane, and invoking execution inside a membrane.

`NewMembrane` This primitive creates a new membrane and returns its identification token (`Id`), its export token (`Exp`) and its execution token (`Exe`).

`Export` This primitive takes an export token and a token value. If both are present in the current membrane, then the value is made also available in the membrane designated by the export token.



`Exec` This primitive takes an execution token and a zero-argument procedure (a token value). If both are present in the current membrane then the procedure is executed in the context of the membrane designated by the execution token.

3.1 Formal semantics

In this section, we present the semantic of a simplified version of Oz featuring membranes. Extension to full Oz is straightforward. This section is adapted from chapter 13 of [VH04].

The general computation model consists of a multiset of threads interacting with one shared store. A thread is a list of membrane statements. A membrane statement is a pair membrane, statement. (In Oz, a thread is a list of statements.)

The store consists of two parts : a single-assignment store and a predicate store. The single-assignment store contains the bindings between variables and between variables and values. The single-assignment store is strictly monotonic : bindings can only be added.

The predicate store contains relations over variables, values and closures. The following predicates are used:

(`´:´`(ξ **`proc`**`{$ ...} S` **`end`**)) binds the procedure name ξ to the closure **`proc`**`{$ ...} S` **`end`**

($exp(\pi\ \mu)$) binds the export token π to it's membrane μ

($exe(\epsilon\ \mu)$) binds the execution token ϵ to it's membrane μ

($in(\alpha\ \mu)$) tells that the token-value α is in the content of membrane μ.

3.1.1 Notations

We will use the infix notation $x : y$ for the `´:´`$(x\ \ y)$ predicate. We will use the infix notation $x = y$ for the bindings in the single-assignment store. We will denote the store as a conjunction of predicates and bindings.

An empty thread is denoted $\langle\rangle$. A non-empty thread T is denoted $\langle\mu(S)\quad T'\rangle$ where $\mu(S)$ is the first membrane statement and T' is the rest of the thread. Multisets of threads are denoted with curly font ($\mathcal{T}$) without braces or union symbols. A roman font (T) is used for threads. A membrane statement is denoted $\mu(S)$ where μ is a membrane and S is a statement.

3.1.2 Abstract syntax

S is a statement, `X` is a variable identifier, α is a structural value (integer, atom or record).

S	::=	**`skip`**	empty statement
	|	$S_1\ S_2$	sequential composition
	|	**`thread`** `S` **`end`**	thread introduction
	|	`{NewMembrane` $\mathtt{X}_1\ \mathtt{X}_2\ \mathtt{X}_3$`}`	membrane introduction
	|	`{Export` $\mathtt{X}_1\ \mathtt{X}_2$`}`	membrane exportation
	|	`{Exec` $\mathtt{X}_1\ \mathtt{X}_2$`}`	membrane execution
	|	**`local`** `X` **`in`** S **`end`**	variable introduction
	|	$\mathtt{X}_1$`=`$\mathtt{X}_2$	variable–variable binding
	|	`X=`α	variable–value binding
	|	**`if`** `X` **`then`** S_1 **`else`** S_2 **`end`**	conditional statement
	|	`{NewName X}`	name introduction
	|	**`proc`**`{X` $\mathtt{X}_1\ \ldots\ \mathtt{X}_n$`}` S **`end`**	procedural value introduction
	|	`{X` $\mathtt{X}_1\ \ldots\ \mathtt{X}_n$`}`	procedure call

3.1.3 Configurations and reduction rules

A configuration is a couple $(\mathcal{T}, \sigma)$, where $\mathcal{T}$ is a multiset of threads and σ is a store.

Reduction rules are denoted:

$$\frac{\mathcal{T}}{\sigma}\ \Bigg\|\ \frac{\mathcal{T}'}{\sigma'}\ \text{if } C$$

The rule transforms a configuration that matches the pattern $(\mathcal{T}, \sigma)$ into a configuration that matches the pattern $(\mathcal{T}', \sigma')$ if condition C holds.

Reduction rules that depend on a single membrane statement are denoted by:

$$\frac{\mu(S)}{\sigma}\ \Bigg\|\ \frac{\mu(S')}{\sigma'}\ \text{if } C$$

rather than by:

$$\begin{array}{c||c} \langle \mu(S)\quad T\rangle & \langle \mu(S')\quad T\rangle \\ \hline \sigma & \sigma' \end{array} \text{ if } C$$

3.1.4 Sequential and concurrent execution

Concurrency

$$\begin{array}{c||c} \mathcal{T}\mathcal{U} & \mathcal{T}'\mathcal{U} \\ \hline \sigma & \sigma' \end{array} \text{ if } \begin{array}{c||c} \mathcal{T} & \mathcal{T}' \\ \hline \sigma & \sigma' \end{array}$$

Equivalence

$\begin{array}{c||c} \mathcal{T} & \mathcal{T}' \\ \hline \sigma & \sigma' \end{array}$ if the configuration $(\mathcal{T}, \sigma)$ is equivalent to the configuration $(\mathcal{T}', \sigma')$ by α-renaming, name bijection and equivalence of bindings.

Elimination of terminated threads

$$\begin{array}{c||c} \langle\rangle & \\ \hline \sigma & \sigma \end{array}$$

Empty statement

$$\begin{array}{c||c} \langle \mu(\texttt{skip})\quad T\rangle & T \\ \hline \sigma & \sigma \end{array}$$

Sequential composition

$$\begin{array}{c||c} \langle \mu(S_1\, S_2)\quad T\rangle & \langle \mu(S_1)\quad \langle \mu(S_2)\quad T\rangle\rangle \\ \hline \sigma & \sigma \end{array}$$

Thread introduction

$$\begin{array}{c||c} \langle \mu(\texttt{thread}\ S\ \texttt{end})\quad T\rangle & \langle \mu(\texttt{skip})\quad T\rangle\ \ \langle \mu(S)\quad \langle\rangle\rangle \\ \hline \sigma & \sigma \end{array}$$

3.1.5 Membrane specific statements

Membrane introduction

$$\begin{array}{c||c} \mu(\{\texttt{NewMembrane}\ x_{exp}\ \ x_{exe}\ \ x_{id}\}) & \mu(x_{exp} = \pi\ x_{exe} = \epsilon\ x_{id} = \mu_2) \\ \hline & \sigma \wedge in(\pi, \mu) \wedge exp(\pi, \mu_2) \\ \sigma & \wedge in(\epsilon, \mu) \wedge exe(\epsilon, \mu_2) \\ & \wedge in(\mu_2, \mu) \end{array}$$

if π, ϵ and μ_2 are fresh names.

This statement creates a new membrane and binds it's export token π to x_{exp}, it's execution token ϵ to x_{exe} and it's identification token μ_2 to x_{id}. `NewMembrane` never fails.

Exporting a value from the current membrane to another membrane

$\begin{array}{c||c} \mu(\{\texttt{Export}\ x_{tok}\ \ x_{exp}\}) & \mu(\texttt{skip}) \\ \hline \sigma & \sigma \wedge in(\alpha, \mu_2) \end{array}$ if $\exists\sigma' : \sigma \equiv \sigma' \wedge x_{tok} = \alpha \wedge in(\alpha, \mu) \wedge x_{exp} = \pi \wedge in(\pi, \mu) \wedge exp(\pi, \mu_2)$ where α is a token-value.

For this rule to reduce x_{tok} has to be bound to a token value that is in the current membrane's content and x_{exp} has to be bound to an export token that is in the current membrane's content. When reduced, the value bound to x_{tok} is added to the content of the membrane with export token π.

Executing a statement in a membrane

$$\frac{\mu(\texttt{\{Exec}\ x_{proc}\ x_{exe}\texttt{\}})}{\sigma} \;\Bigg\|\; \frac{\mu_2(S)}{\sigma}$$

if $\exists\sigma' : \sigma \equiv \sigma' \wedge x_{proc} = \xi \wedge in(\xi, \mu) \wedge x_{exec} = \epsilon \wedge in(\epsilon, \mu) \wedge exe(\epsilon, \mu_2) \wedge \xi : \textbf{\texttt{proc\{\$\}}}\ S\ \textbf{\texttt{end}}$

For this rule to reduce x_{proc} has to be bound to a zero-argument procedure value that is in the current membrane and x_{exe} has to be bound to an execution token that is in the current membrane. This rule reduces to a membrane statement, the membrane being μ_2 and the statement corresponding to the application of the procedure designated by ξ (see next section).

3.1.6 Other statements

Variable introduction

$$\frac{\mu(\textbf{\texttt{local}}\ X\ \textbf{\texttt{in}}\ S\ \textbf{\texttt{end}})}{\sigma} \;\Bigg\|\; \frac{\mu(S\{X \to x\})}{\sigma}$$

if x is a fresh variable and $S\{X \to x\}$ is S with all free occurrences of X substituted by x.

Bindings

$$\frac{\mu(x = y)}{\sigma} \;\Bigg\|\; \frac{\mu(\textbf{\texttt{skip}})}{\sigma \wedge x = y}$$

if $\sigma \wedge x = y$ is consistent.

$$\frac{\mu(x = \alpha)}{\sigma} \;\Bigg\|\; \frac{\mu(\textbf{\texttt{skip}})}{\sigma \wedge x = \alpha}$$

if $\sigma \wedge x = \alpha$ is consistent.

Conditional execution

$$\frac{\mu(\textbf{\texttt{if}}\ x\ \textbf{\texttt{then}}\ S_1\ \textbf{\texttt{else}}\ S_2\ \textbf{\texttt{end}})}{\sigma \wedge x = \textbf{\texttt{true}}} \;\Bigg\|\; \frac{\mu(S_1)}{\sigma \wedge x = \textbf{\texttt{true}}}$$

$$\frac{\mu(\textbf{\texttt{if}}\ x\ \textbf{\texttt{then}}\ S_1\ \textbf{\texttt{else}}\ S_2\ \textbf{\texttt{end}})}{\sigma \wedge x = \textbf{\texttt{false}}} \;\Bigg\|\; \frac{\mu(S_2)}{\sigma \wedge x = \textbf{\texttt{true}}}$$

Name introduction

$$\frac{\mu(\texttt{\{NewName}\ x\texttt{\}})}{\sigma} \;\Bigg\|\; \frac{\mu(x = \xi)}{\sigma \wedge in(\xi, \mu)}$$

if ξ is a fresh name.

Procedural value introduction

$$\frac{\mu(\textbf{\texttt{proc\{}}x\ x_1\ \ldots\ x_n\texttt{\}}\ \texttt{S}\ \textbf{\texttt{end}})}{\sigma} \;\Bigg\|\; \frac{\mu(x = \xi)}{\sigma \wedge in(\xi, \mu) \wedge \xi : \textbf{\texttt{proc\{\$}}\ \texttt{X}_1\ \ldots\ \texttt{X}_n\texttt{\}}\ \texttt{S}\ \textbf{\texttt{end}}}$$

if ξ is a fresh name.

Procedure call

$$\frac{\mu(\texttt{\{}x\ x_1\ \ldots\ x_n\texttt{\}})}{\begin{array}{c}\sigma \wedge x = \xi \wedge in(\xi, \mu) \\ \wedge \xi : \textbf{\texttt{proc\{\$}}\ \texttt{X}_1\ \ldots\ \texttt{X}_n\texttt{\}}\ \texttt{S}\ \textbf{\texttt{end}}\end{array}} \;\Bigg\|\; \frac{\mu(S\{\texttt{X}_1 \to x_1, \ldots, \texttt{X}_n \to x_n\})}{\begin{array}{c}\sigma \wedge x = \xi \wedge in(\xi, \mu) \\ \wedge \xi : \textbf{\texttt{proc\{\$}}\ \texttt{X}_1\ \ldots\ \texttt{X}_n\texttt{\}}\ \texttt{S}\ \textbf{\texttt{end}}\end{array}}$$

3.2 Using membranes

We are still investigating different ways of using the expressive power of membranes to program confinement. The following example of a versatile sandbox illustrates how membranes can be used to build abstractions for confinement.

3.2.1 Example of a versatile sandbox

Problem

In a capability system we have an untrusted procedure that we want to run. We want the procedure to have access to certain capabilities (e.g. to display information on the screen) and we also want to confine other capabilities (e.g. no access to the file system) from the procedure. The usual solution consists of an iterative process of reachability analysis followed by improving confinement. This can become impractical if the application is big. With membranes, usability analysis can be significantly simpler than reachability analysis.

Solution with membranes

We first create a membrane (C) that will confine the export and exec tokens of the sandbox membrane (S). We make C's export and exec tokens available in C for the sole purpose of being able to export these tokens to the sandbox membrane (S). For the same reason, we export `MkModule` (the sandbox utility library) and the confined procedure `P` to C. We make the policy functions `InCtrl` and `OutCtrl` available in C or the sole purpose of being able to export these tokens to the test membrane (T). We instantiate the sandbox library that will allow to export tokens to the sandbox mediated by the `InCtrl` policy.

In the context of membrane C, we create the sandbox membrane (S) and export the necessary values to S. In the context of membrane C, we also create the test membrane (T) and export the policy functions to T. In the context of membrane S,We instantiate the sandbox library that will allow to export tokens from the sandbox mediated by the `OutCtrl` policy and we apply the confined procedure to this library.

The sandbox library is created by `MkModule` from a boolean policy function `Ctrl`. It consists of:

- The `tryExport` function. This function exports a token-value `Tok` to a membrane with export token designated by `Exp`, only if `Tok` is not an export token and the export is allowed by the policy function `Ctrl`. `Exp` has to be in C's content but not necessarily in the current membrane's content. The policy function will be applied in the context of the test membrane T.
- The `register` procedure. This procedure takes an export token (present in the current membrane's content) for a membrane μ and makes the library usable from μ's context. It exports it's argument to membrane C so that the argument can receive values exported by `tryExport`.
- The `exportToTest` procedure. This is a utility to make values usable by the policy functions.

- The `exportToken` value. This is the export token of the sandbox membrane (S).

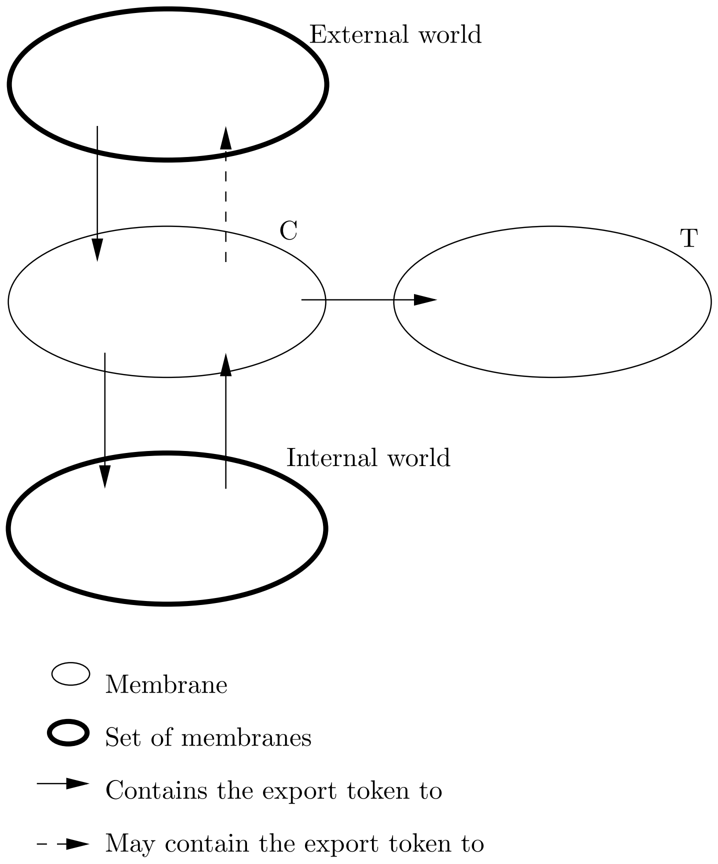
A membrane is part of the internal world (what is inside the sandbox) if it is S or it has been created by code running in the context of an internal world membrane.

A membrane is part of the external world (what is outside the sandbox) if it is not in the internal world, not C and not T.

The abstraction guards the following invariant : If a value is present in an internal and an external membrane, then it is present in C and T and either `InCtrl` or `OutCtrl` approved the export. The only exceptions to this rule are the export and exec tokens for C and the procedure to confine. These values are normally present in the internal and external world without being in T or being approved by either `InCtrl` or `OutCtrl`.

The following diagram gives an overview of the membrane configuration. Plain arrows point to the membrane of which the export token is available when the sandbox is created. Dashed arrows point to the membrane of which the export token can become available later. The test `IsExportToken` guarantees that no other arrows exist.

The proof that the invariant is respected is based on inspection of the code running in C's context. This inspection is straightforward because of the lexical confinement of C's exec token.



Here is an implementation using the actual Oz syntax, extended with membranes.

```
proc{VersatileSandBox InCtrl OutCtrl P ?M}
   ExpC ExeC  ExpT ExeT  ExpS ExeS
   fun{MkModule Ctrl}
      module(tryExport:fun{$ Tok Exp}
                          if {IsExportToken Tok} then false
                          else Res in
                             {Export Tok ExpC}
                             {Exec proc{$}
                                      {Export Tok ExpT}
                                      {Exec proc{$}
                                               Res={Ctrl Tok}
                                            end ExeT}
                                      if Res then
                                          {Export Tok Exp}
                                      end
                                   end ExeC}
                             Res
                          end
                       end
             register:proc{$ Exp}
                         {Export [ExeC ExpC] Exp}
                         {Export Exp ExpC}
                      end
             exportToTest:proc{$ Tok}
                             if {Not {IsExportToken Tok}} then
                                {Export Tok ExpC}
                                {Exec proc{$}
                                         {Export Tok ExpT}
                                      end ExeC}
                             end
                          end
             exportToken:ExpS
            )
   end
in
   {NewMembrane ExpC ExeC _}
   {Export [ExpC ExeC MkModule P InCtrl OutCtrl]
    ExpC}
   M={MkModule InCtrl}
   {Exec proc{$}
            {NewMembrane ExpS ExeS _}
            {Export [ExpC ExeC ExpS MkModule P]
             ExpS}
            {NewMembrane ExpT ExeT _}
            {Export [OutCtrl InCtrl]
             ExpT}
            {Exec proc{$}
                     {P {MkModule OutCtrl}}
                  end ExeS}
         end ExeC}
end
```

To keep the example short and readable we used `IsExportToken` as if it was a primitive. We used a version of `Export` that allows its first argument to be a list of token values rather than a single value.

3.2.2 Discussion

Safe defaults

A newly created membrane is completely confined from the rest of the world. Only export can change this situation. Confinement proofs are simplified because usability depends only on a single language construct.

Dynamic control of confinement

Confinement by membranes can be refined at runtime, values can be exported depending on runtime conditions, membranes can be created, etc. This enables the creation of a vast diversity of abstractions.

Interaction with capability-based security

Confinement depends on liveness properties as well as on safety properties. The caretaker (revocable forwarder) pattern[MS03] is a good example in which revocation of one functionality relies on the availability of another functionality (the revoker itself).

An attacker can try to use membranes to defy revocation by confining the revoker.

Such an attack can be repelled in the following way : create a new name which is only exported to the membranes which are known to provide all the needed values and check for it's presence before doing anything which requires the use of some value to guarantee security.

Just like in pure capability systems, the designation is unforgeable.

Conditions on execution

Upon `Exec`, procedures are checked for presence in the current membrane. We considered also the stricter behavior of checking them for presence in the destination membrane. In general, this was too restrictive as it would force the programmer to export the procedure before executing it, requiring control of the export token. The programmer can always make an abstraction on top of `Exec` to control the use of the exec token.

3.3 Guidelines for extended semantics in Oz

The semantics presented before is sufficient to have a Turing-complete language, but lacks many important concepts such as state or exceptions. Other concepts can be added but some care must be taken to ensure that membranes keep their good properties.

The need for a value extends to the need for the presence of that value in the current membrane. Oz operations that block when a variable is unbound should also block when the variable is bound to a token value that is absent from the current membrane.

Values are created in the current membrane Oz operations that create a token value should add the value to the current membrane but not to another membrane. This ensures that every value can be used in the context of the membrane it was created in, and is automatically confined to the current membrane.

Consequent confinement An operation should only require the presence of a value in the current membrane if the language can guarantee that the effects of the operation can only be reached via operations that have the same requirements.

4 Related work

4.1 Confinement at the OS level

Operating systems traditionally provide several mechanisms for confinement. Apart from the memory isolation between processes (implemented by special hardware in the MMU), there is also file confinement between users, nearly always implemented with Access Control Lists (ACLs) or derivatives (e.g. Unix permissions). This provides for provable confinement of files.

4.2 Early approaches

Some earlier systems augmented the OS with primitives intended to make intra-application confinement possible. KeyKOS[Fra88] added factories and Multics[Sal74] added rings. These primitives allow a form of confinement that is very different from the pure OS confinement we mentioned above. These systems did not survive but the principles are still useful.

4.3 Language design work

Lexical scoping can help confinement but the intricacies of nested scoping can complicate reachability analysis. Encapsulation simplifies the scoping mechanism and therefore also the reachability analysis. Membranes are an effort to improve confinement in Oz, and therefore are related to the work of securing the Oz language [SV05, Con] (the Oz-E project) The concept can then be compared to similar constructs in other capability systems such as E [MSC$^+$01].

4.4 Consequent Interposition

The e-lang community has been discussing an alternative approach also called ”membranes” [Mil03]. The original rationale for this abstraction was generalized revocability, rather than confinement. While capability systems cannot revoke access, they can effectively revoke authority (the potential effects of having access) if the accessed entity can be instructed to stop providing its service to its clients. Any capability can be made ”revocable” by wrapping it in a proxy that can be deactivated this way.

The general principle of inter-positioning an allied proxy is useful for confinement too.

Generalized revocability provides for subjects in one group to offer only *revocable* authority to subjects in another group (both groups are assumed to be disjunct, and subjects in the second group have no direct access to subjects in the first group). A "membrane" between these groups will automatically wrap all capabilities in a suitable wrapper to avoid direct access between any to subjects in the subgroup. The generated wrappers will act exactly as the membrane, so that interposition is assured consequently. The general principle use is still inter-positioning and thus useful for helping to ensure generalized confinement. At the moment of writing, no detailed design or implementation of this mechanism is yet available.

Consequent Interposition has the advantage of never introducing confused deputies. The designation of an inter-positioned (possibly revoked) proxy is different from the designation of the original capability, and should not confuse a deputy who has access to the original capability.

4.5 Process Calculi

This research started from the M-calculus [SS03] and Kell-calculus [BS03] but diverted rapidly from these formalisms. We kept the notion of an ambient as a context of execution. We add no use for the mobility concept but we consider adopting the hierarchical structure of ambients in future work.

5 Future plans

We are continuing this work in three directions:

- Resolving confused deputies. By separating designation from authority, membranes introduce confused deputies. The fundamental reason is that authorization has to be decided not only based on the subject and object but also on the relation between the two. The advantage we have over traditional ACL systems is that our designations are unforgeable. We will consider recombination of designation and authority as an additional primitive in future work.
- Distributing membranes. Confinement is extremely important for distributed applications. The membranes described here would be a great tool to prove distributed confinement. We plan to develop cryptographic protocols to implement them on untrusted networks. We also have to add support for the inherently hierarchical structure of the trust on networks. Because we don't want false security (as in point 3 of the guidelines), membranes should not allow to express confinement of a resource on a node that we don't trust. Hierarchical membranes are possible, and the ambient calculi shows the way.
- Abstraction building. Using the primitives presented here is awkward because of the extremely fine level of confinement that they provide. We need abstractions to take care of most of the export

work. We can build abstractions as demonstrated by the versatile sandbox (Section 3.2.1). Other abstractions like the powerbox[SM02] and its variant will become available. We intend to build a complete framework to facilitate programming with membranes.

6 Conclusion

How do the membranes fulfill their goals? The definitions are simple and reasoning about them is simple but their use isn't as simple as we would like, mainly because of the numerous values which have to be exported before anything useful can be executed in a different membrane. The provided defaults are indeed very safe as a newly created membrane is a perfect sandbox. The granularity is as fine as possible : it is at the level of the individual values. The new primitives provide for dynamic control of membranes but are not part of the actual confinement mechanism. Confinement itself is enforced by the membrane sensitive conditions of the reduction rules.

7 Acknowledgements

This research was partially funded by the MILOS project (Walloon Region of Belgium, Convention 114856), the PEPITO project (European Union Fifth Framework Programme, Global Computing IST-2001-33234), and the EVERGROW project (European Union Sixth Framework Programme, Project number 001935).

References

[BS03] Philippe Bidinger and Jean-Bernard Stefani. The kell calculus: operational semantics and type system. In *Proceedings 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS 03)*, Paris, France, November 2003.

[Con] The Mozart Consortium. Mozart/Oz website. *http://www.mozart-oz.org*.

[Fra88] Bill Frantz. KeyKOS - A Secure, High-Performance Environment for S/370. In *Proceedings of SHARE 70 I (SHARE Inc, Chicago).*, pages 465–471, February 1988.

[Har89] Norm Hardy. The confused deputy. *ACM SIGOPS Oper. Syst. Rev*, 22(4):36–38, 1989. `http://www.cap-lore.com/CapTheory/ConfusedDeputy.html`.

[Mil03] Mark S. Miller. [e-lang] Revoking Capabilities, January 2003. Mail posted at e-lang mailing list, available at `http://www.eros-os.org/pipermail/e-lang/2003-January/008434.html`.

[MS03] Mark S. Miller and Jonathan Shapiro. Paradigm regained: Abstraction mechanisms for access control. In *8th Asian Computing Science Conference (ASIAN03)*, pages 224–242, December 2003.

[MSC$^+$01] Mark Miller, Marc Stiegler, Tyler Close, Bill Frantz, Ka-Ping Yee, Chip Morningstar, Jonathan Shapiro, Norm Hardy, E. Dean Tribble, Doug Barnes, Dan Bornstien, Bryce Wilcox-O'Hearn, Terry Stanley, Kevin Reid, and Darius Bacon. E: Open source distributed capabilities, 2001. Available at `http://www.erights.org`.

[Ree96] Jonathan A. Rees. A security kernel based on the lambda-calculus. Technical report, MIT, 1996.

[Sal74] Jerome H. Saltzer. Protection and the control of information sharing in multics. *Commun. ACM*, 17(7):388–402, 1974.

[SM02] Marc Stiegler and Mark S. Miller. A capability based client: The darpabrowser. Technical Report Focused Research Topic 5 / BAA-00-06-SNK, Combex, Inc., June 2002. Available at `http://www.combex.com/papers/darpa-report/index.html`.

[SS75] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63:9, pages 1278–1308, September 1975.

[SS03] Alan Schmitt and Jean-Bernard Stefani. The m-calculus: A higher-order distributed process calculus. In *In Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages (POPL'03)*, New Orleans, LA, USA, January 15-17 2003.

[SV05] Fred Spiessens and Peter Van Roy. The Oz-E project: Design guidelines for a secure multiparadigm programming language. In *Multiparadigm Programming in Mozart/Oz: Extended Proceedings of the Second International Conference MOZ 2004*, volume 3389 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2005.

[VH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, March 2004.