Secure Cooperation of Untrusted Components

Cutting Edge Research

Winter term 2022/23

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: October 26, 2022

Outline

- Motivation
- Access Control
- The Object Capability Paradigm
- A Capability Type System
- Conclusion and Future Work

1 Motivation

A sorting library in Java

- You just found the "*best list sorting class ever*" in the WWW
- Interface of the class:

```
class Sorter {
  ...
  public void sort(List<? extends Comparable> list) {
    ...
  }
}
```

- Your code:

```
List<Contact> contacts = ...;
Sorter sorter = new Sorter();
sorter.sort(contacts);
```

- Your belief: `sort()` only uses `Contact.compareTo()`

1 Motivation

A sorting library in Java

- You just found the “*best list sorting class ever*” in the WWW
- Interface of the class:

```
class Sorter {
   ...
   public void sort(List<? extends Comparable> list) {
      ...
   }
}
```

```
Socket sock = new Socket(...);
PrintStream stream = new PrintStream(...);
Contact c = (Contact)list.get(i);
stream.println(c.getEMail());
```

- Your code:

```
List<Contact> contacts = ...;
Sorter sorter = new Sorter();
sorter.sort(contacts);
```

- Your belief: `sort()` only uses `Contact.compareTo()` **???**

Principle of least authority (POLA)

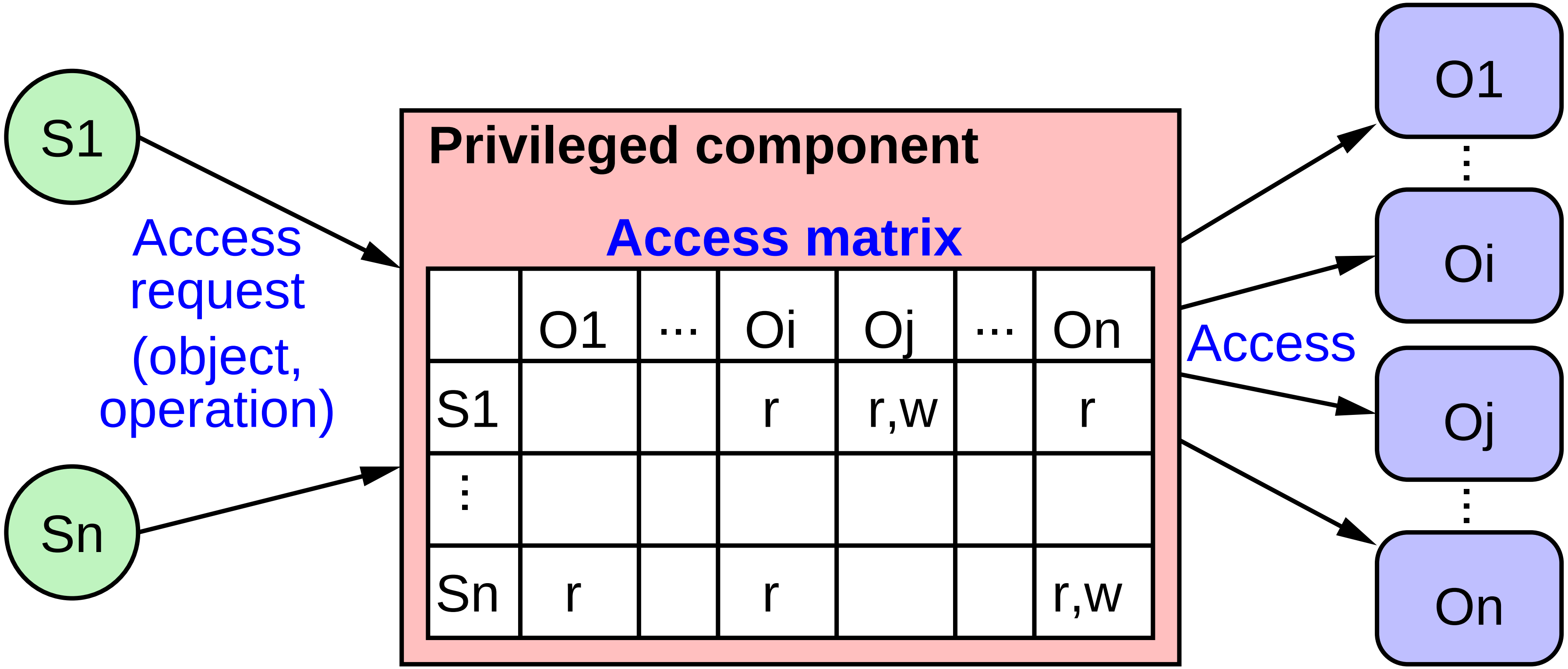
> *A software component should receive just the authority required to fulfill its intended purpose* [1]

- Difference between *authority* and *permission* [2][3]
  - *authority* also includes indirect effects
  - e.g., component may make another component perform an action, which is not directly permitted
  - e.g., action may be permitted but not available
- Basis: access control mechanisms
  - access matrix
  - access control lists (ACLs), capabilities

2 Access Control ...

Classical implementation of access control

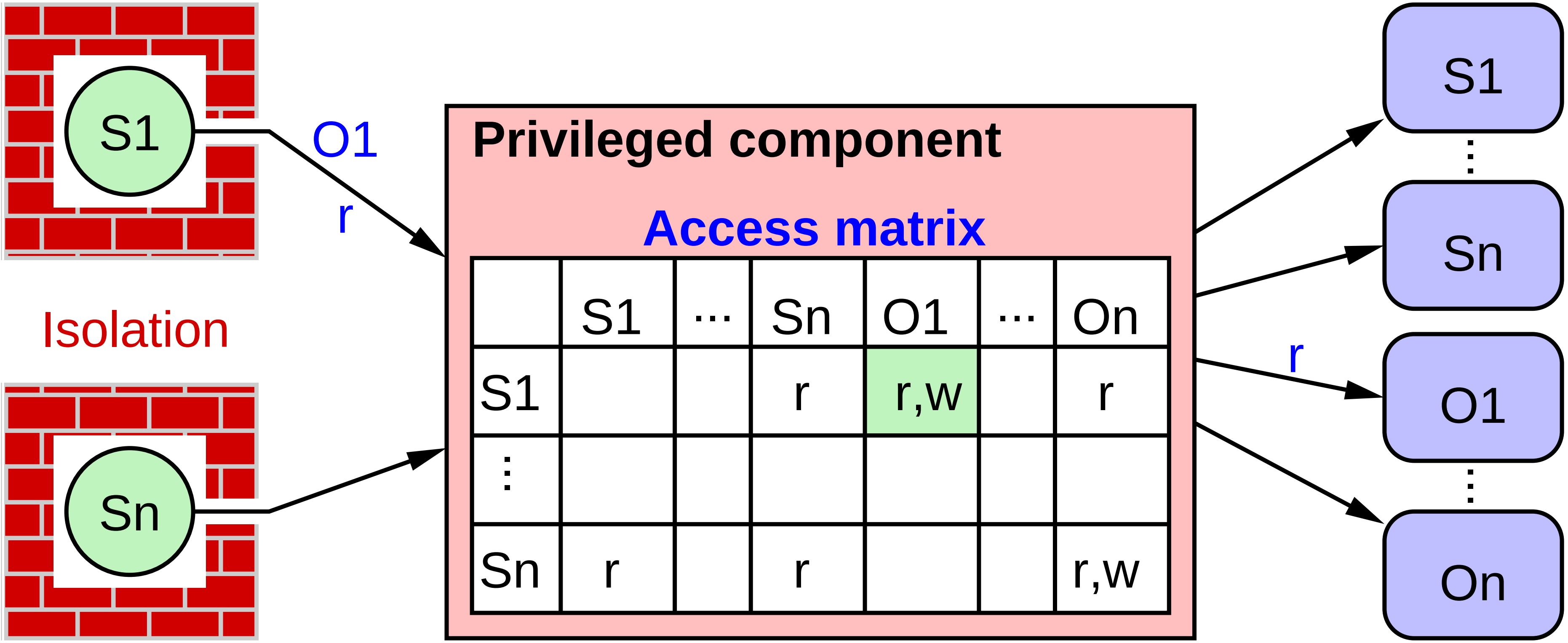
- Textbook figure:
  - subjects act upon objects
  - accesses are mediated via access matrix



	O1	...	Oi	Oj	...	On
S1			r	r,w		r
⋮						
Sn	r		r			r,w

Classical implementation of access control

- More realistic:
  - subjects are objects that may actively perform operations
  - subjects have direct access only to a privileged component



Privileged component

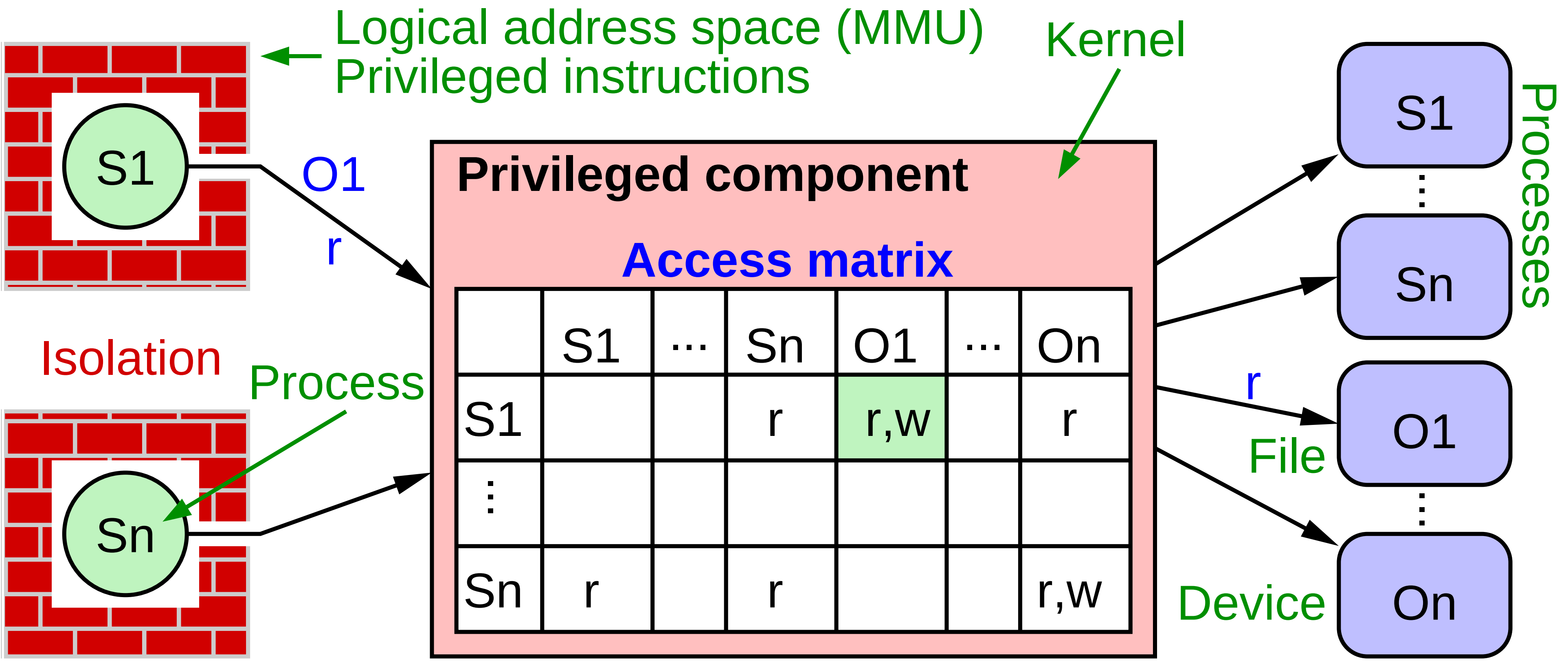
Access matrix

	S1	···	Sn	O1	···	On
S1			r	r,w		r
⋮						
Sn	r		r			r,w

2 Access Control ...

Classical implementation of access control

- Example: Linux
  - subjects = processes, isolated via hardware
  - all accesses mediated by the kernel

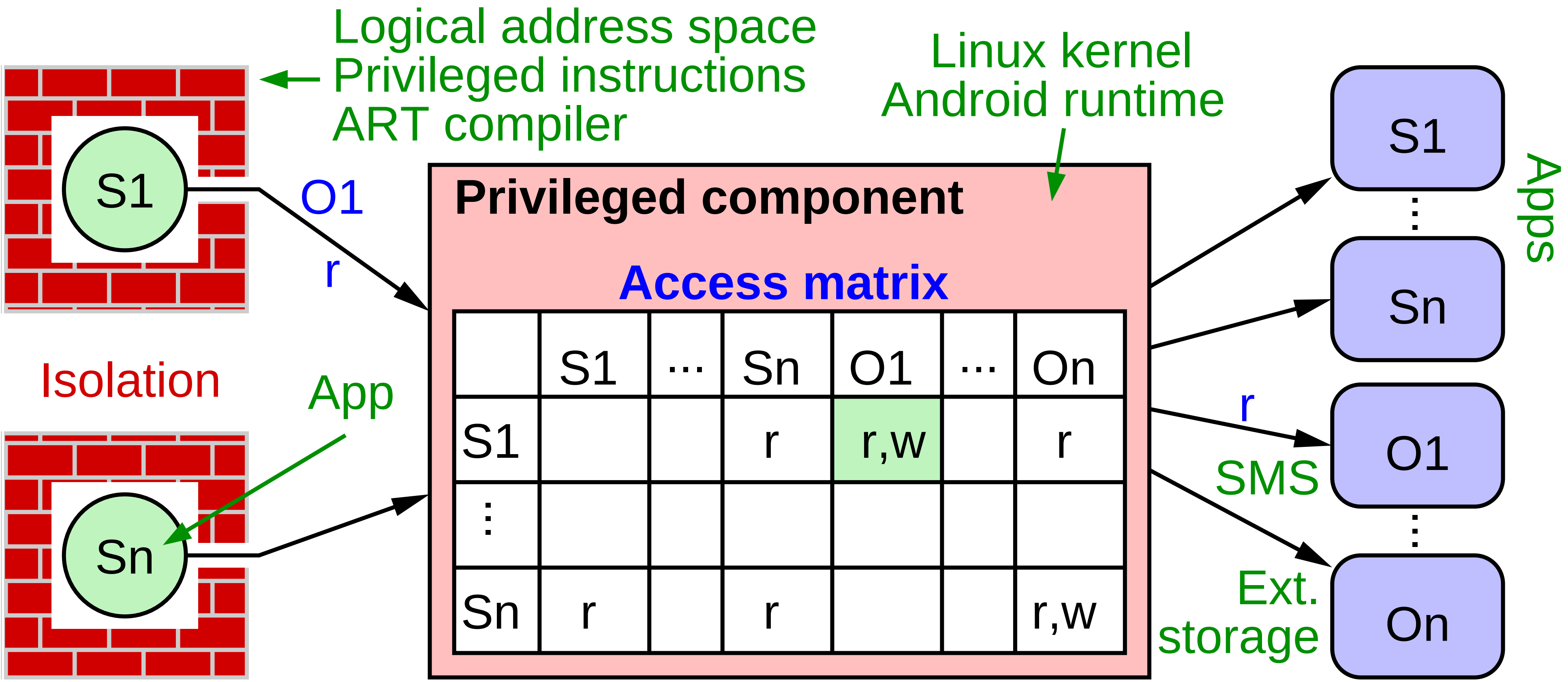


Access matrix

	S1	···	Sn	O1	···	On
S1			r	r,w		r
⋮						
Sn	r		r			r,w

Classical implementation of access control

- Example: Android
  - subjects = apps, objects = subsystems
  - accesses mediated by kernel and runtime

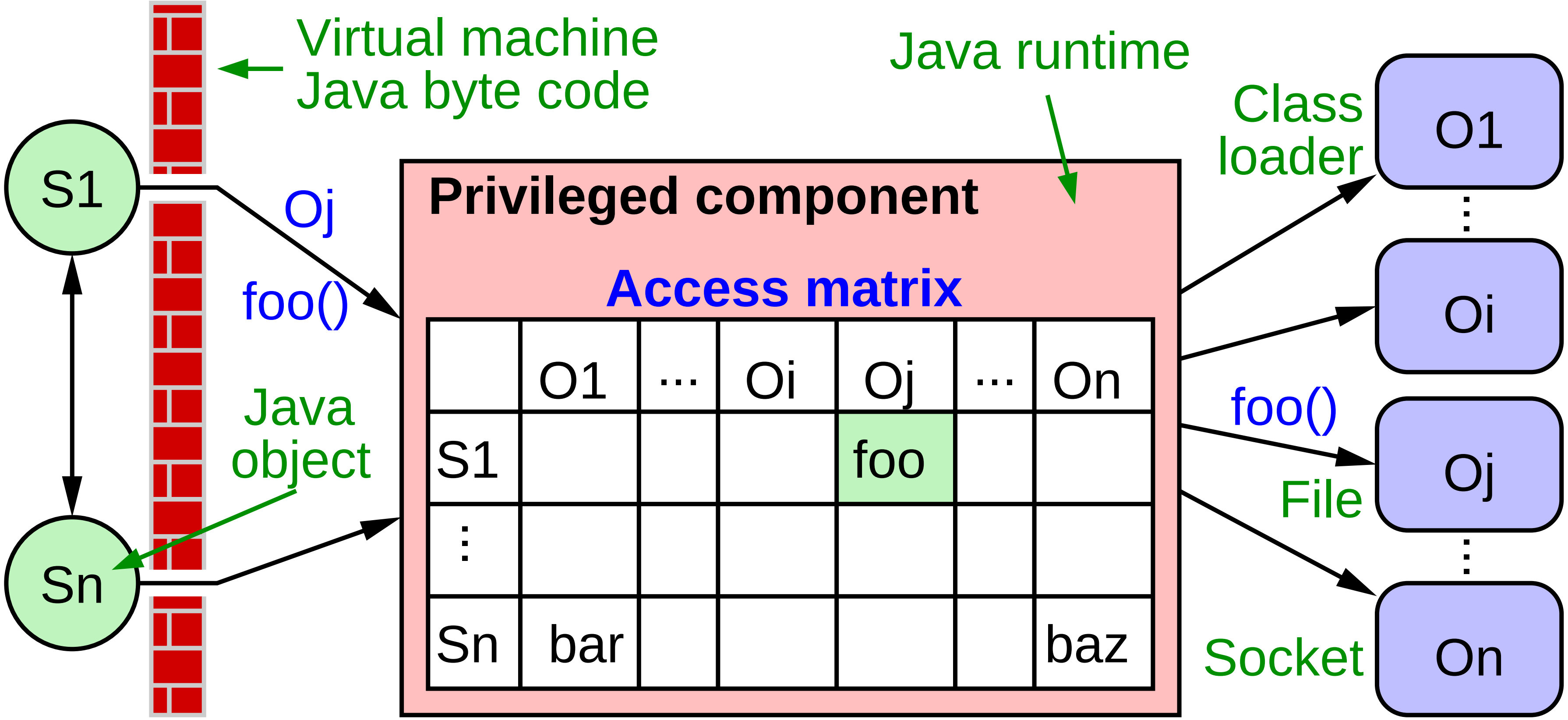


	S1	···	Sn	O1	···	On
S1			r	r,w		r
⋮						
Sn	r		r			r,w

2 Access Control ...

Classical implementation of access control

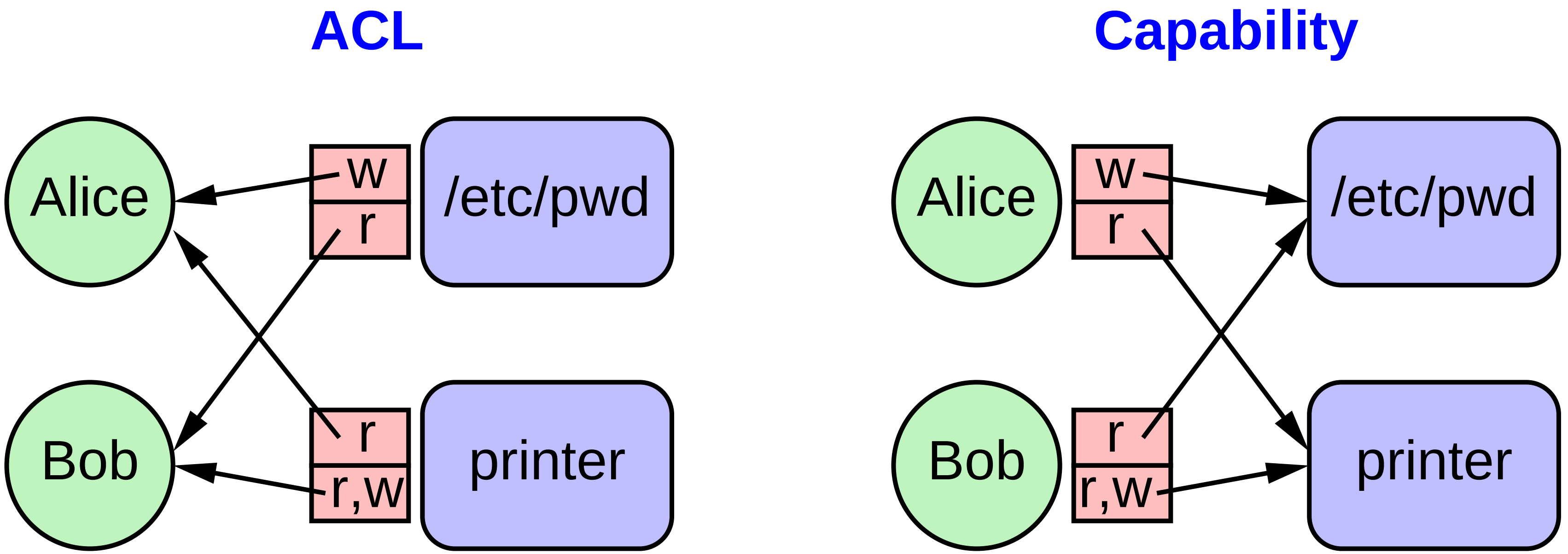
- Example: Java security manager
  - subjects = Java objects, not fully isolated
  - Java runtine mediates method calls on ‘critical‘ objects



	O1	···	Oi	Oj	···	On
S1				foo		
⋮						
Sn	bar					baz

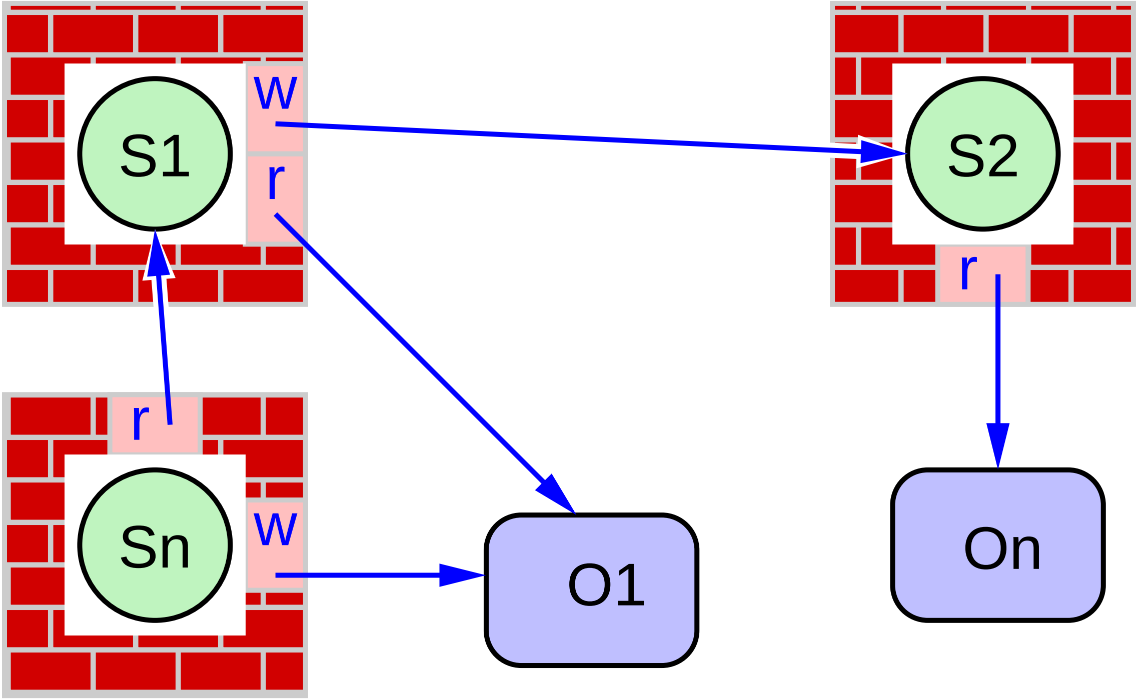
Access control using capabilities

- Capability: unforgeable information **given to** a subject, enabling it to perform operations on an object
  - inseparably combines designation with authority [4]
- Comparison:



Access control using capabilities

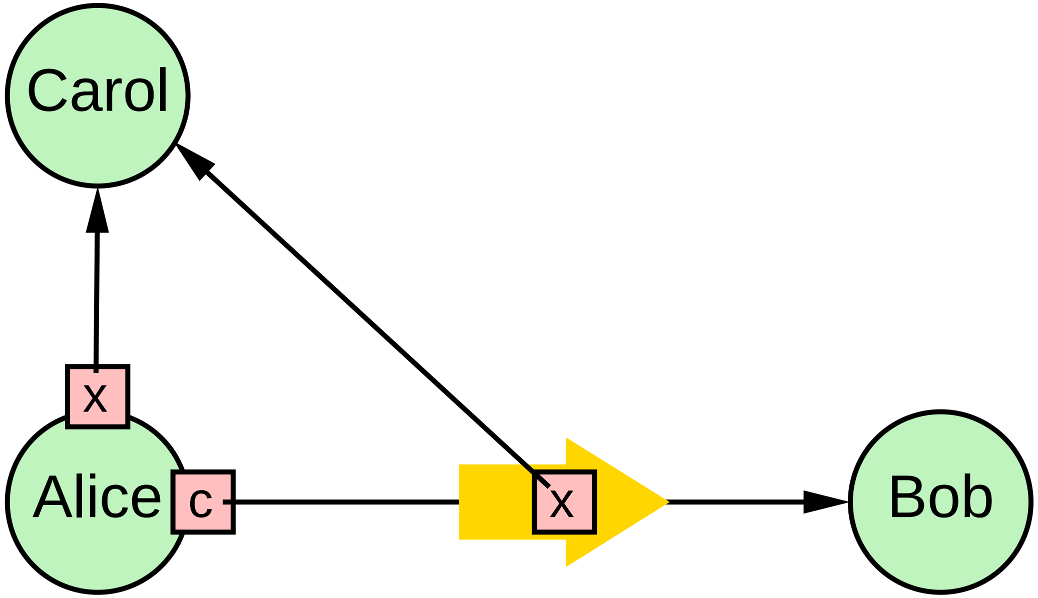
- Capability: unforgeable information **given to** a subject, enabling it to perform operations on an object
  - inseparably combines designation with authority [4]
- Results in decentralized access control



Dynamics of access permissions

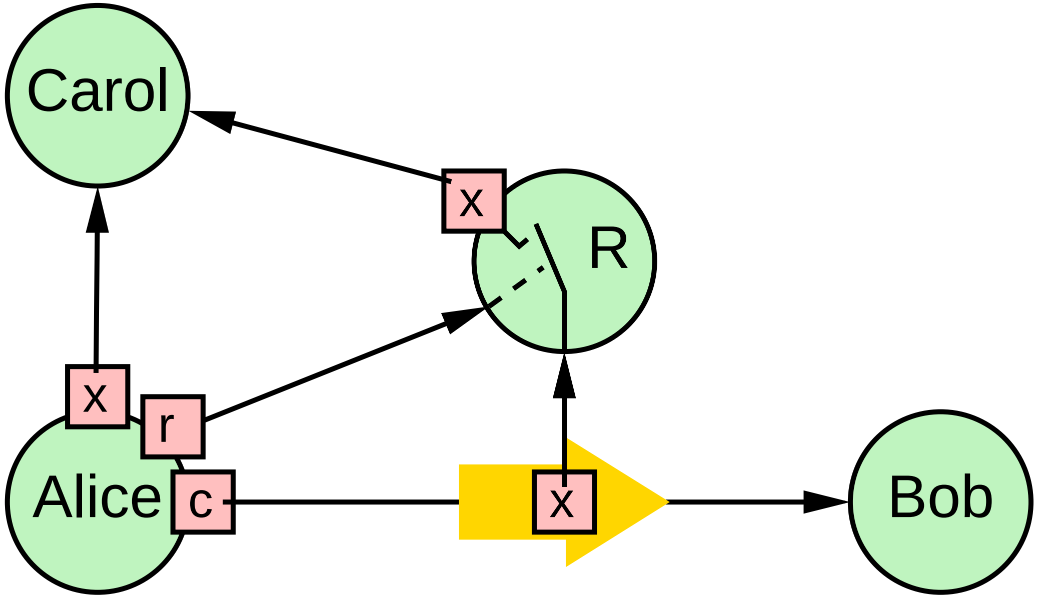
- How can the acess matrix be modified at runtime?
  - changing the access matrix must require proper authority!
- ACLs
  - typically: objects have a unique *owner*
  - owner is allowed to change ACL arbitrarily
- Capabilities
  - capabilities may be passed between subjects
  - but not arbitrarily: passing a capability requires a capability! [4]
  - capabilities may be weakened (attenuated), but not amplified
  - capabilities also support revocation [4]
    - by using the caretaker pattern [5]

Dynamics of access permissions



- Capabilities
  - capabilities may be passed between subjects
  - but not arbitrarily: passing a capability requires a capability! [4]
  - capabilities may be weakened (attenuated), but not amplified
  - capabilities also support revocation [4]
    - by using the caretaker pattern [5]

Dynamics of access permissions



- Capabilities
  - capabilities may be passed between subjects
  - but not arbitrarily: passing a capability requires a capability! [4]
  - capabilities may be weakened (attenuated), but not amplified
  - capabilities also support revocation [4]
    - by using the caretaker pattern [5]

Discussion

- Classical implementation
  - granularity of subjects is often restricted
  - permissions must be checked for each access
  - centralized mediator can be a bottleneck
  - privileged component can lead to security problems
  - restricted dynamics (e.g., no delegation)
- Capabilities
  - allow fine grained subjects
  - allow delegation of authority
  - access restrictions can be enforced by construction
    - i.e., no (or less) checks at runtime

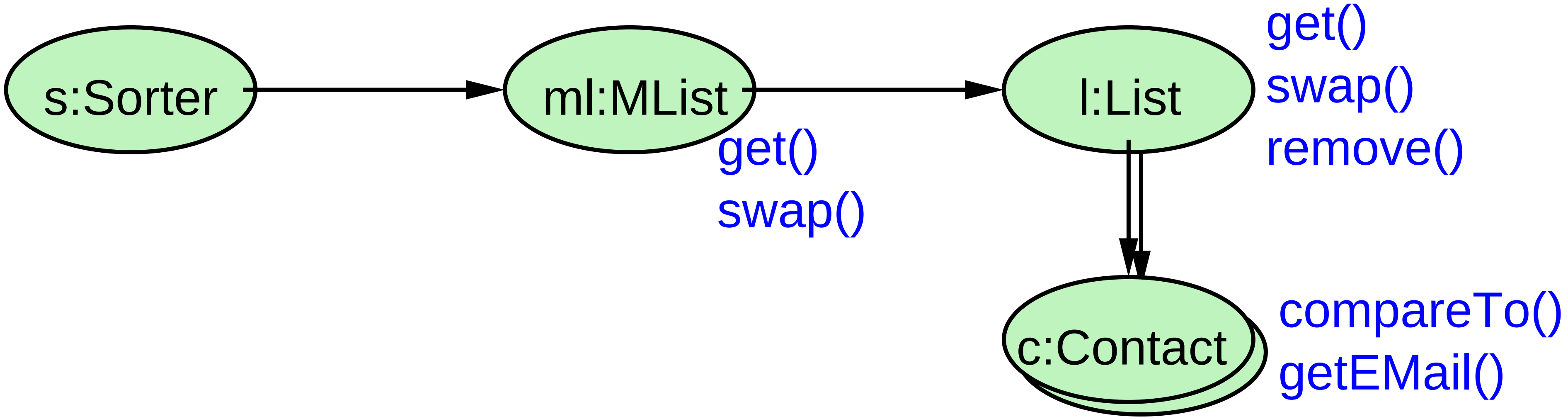
3 The Object Capability Paradigm [5][6]

- Basis: pure object oriented programming
  - everything is an object (even the subjects)
  - access to attributes only via method calls
- An object reference is a capability to access the object
  - note: no distinction is made between different operations
  - i.e. the capability allows to call all available methods
- How can an object A receive a capability to B? [6]
  - if A creates B, A has a reference (capability) to B
  - A can receive the reference to B from another object C
    - as an argument of A's constructor
    - as an argument of a method call (when C calls A)
    - as a result of a method call (when A calls C)

The Object Capability Paradigm ...

Attenuation of authority

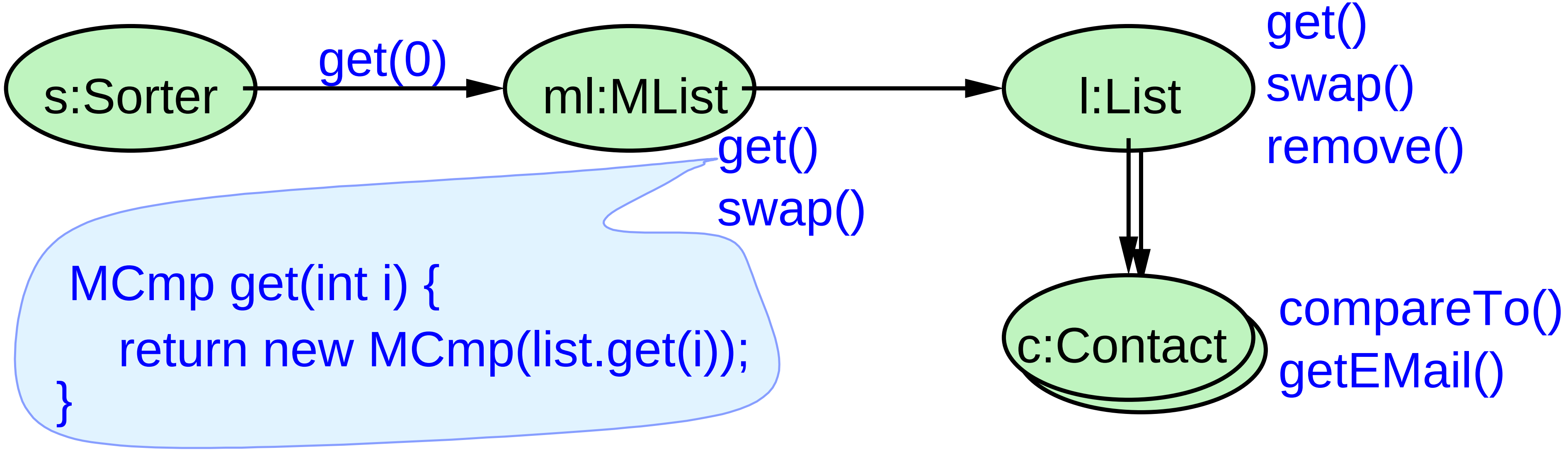
- How can we minimize the authority granted by a reference?
- Answer: membrane pattern [3][5]
  - wrap the object into a membrane that provides less methods and/or restricted methods (that may return membranes)
  - i.e., membrane acts as fine-grained capability
- In the `Sorter` example:



The Object Capability Paradigm ...

Attenuation of authority

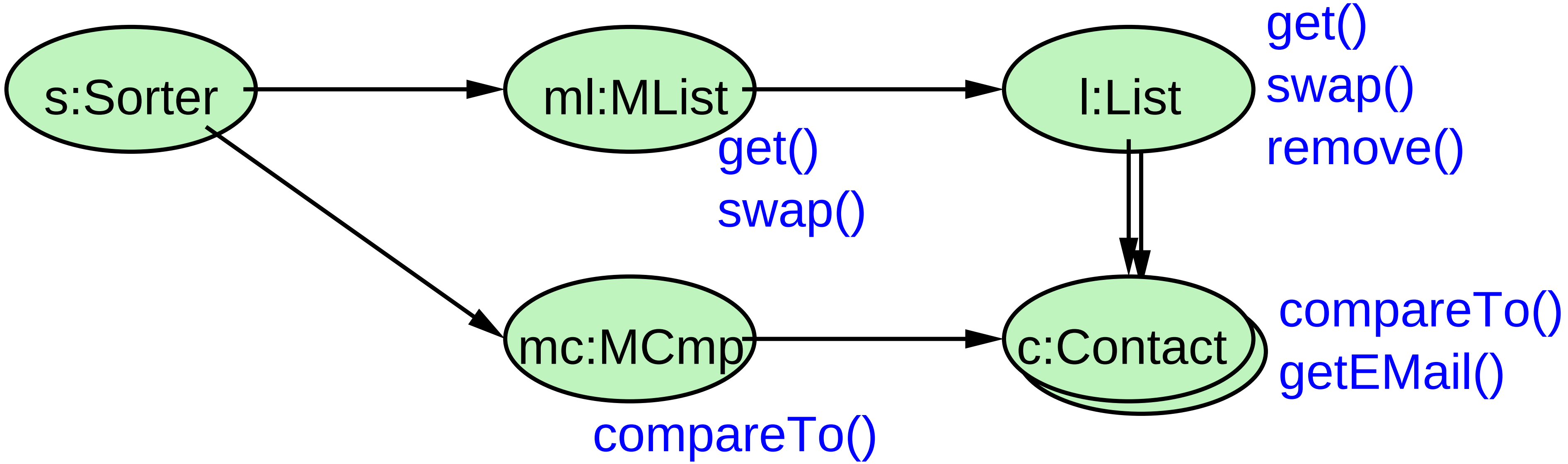
- How can we minimize the authority granted by a reference?
- Answer: membrane pattern [3][5]
  - wrap the object into a membrane that provides less methods and/or restricted methods (that may return membranes)
  - i.e., membrane acts as fine-grained capability
- In the `Sorter` example:



```
MCmp get(int i) {
   return new MCmp(list.get(i));
}
```

Attenuation of authority

- How can we minimize the authority granted by a reference?
- Answer: membrane pattern [3][5]
  - wrap the object into a membrane that provides less methods and/or restricted methods (that may return membranes)
  - i.e., membrane acts as fine-grained capability
- In the `Sorter` example:



The Object Capability Paradigm ...

Secure programming languages [3][7]

- Based on the object capability paradigm and security patterns
- Foundations of security: [8]
  - memory safety: references cannot be forged
  - object encapsulation: no data access without reference
    - implies: no static methods / attributes
- Remaining shortcomings:
  - system can be attacked ‘from below’ [9]

    ⇒ must only permit code written in the secure language
    ⇒ use a secure *intermediate* language (byte code)
  - how can we know the minimal required access rights?
  - run time overhead induced by (cascaded) membranes

4 A Capability Type System

- Most programming languges are typed
- A reference type specifies requirements on the referenced object
  - e.g. `Comparable` requires that the object provides a method `compareTo()`
- A reference type also restricts the use of the referenced object
  - `Comparable` itself does not allow to invoke `getEMail()`
- Thus, types can be used to specify required / granted rights
- Idea: split capability into two parts
  - reference controls whether object can be accessed or not
  - type of reference variable controls the permitted methods
- Additional security requirements:
  - a method can be called only if both type and object permit it
  - type casts must not allow to amplify authority

4.1 Types [10]

- **Type**: a specification of properties of data objects
  - or: a collection of objects with specified properties
- **Type system**: set of rules assigning a type to language constructs, such as variables, expressions, objects, ...
- **Type checking**: verifying and enforcing the constraints of types
- For ease of presentation: we just consider interface types
- An **interface type** defines all available / usable methods, together with their argument and result types
  - for simplicity: we just consider one argument and one result
- Important relation: **subtype** relation
  - S is subtype of T, if each object of type S also has type T
  - usually written as $S <: T$, **here**: $T \leq S$

Types ...

Formal representation of types

- Type: $\mathcal{T} = \mathcal{MS}^{\mathbb{A}^*}$
  - a type defines a state for each method
    - i.e., it maps a string to the corresponding method state
  - $\mathbb{A}^*$ = the set of all strings
- Method state: $\mathcal{MS} = \mathcal{A} \times (\mathcal{M} \cup \{\perp\})$
  - a method state consists of an assertion (permission) and an optional method signature
- Assertions: $\mathcal{A} = \{\mathit{denied}, \mathit{avail}\}$
  - *denied*: type does not allow to call this method
  - *available*: type provides the method with the given signature
- Method signature: $\mathcal{M} = \mathcal{T} \times \mathcal{T}$

Types ...

Subtype relation

- S is subtype of $T \Rightarrow$ object of type S can be used where an object of type T is required
  - i.e., $o : S$ can be assigned to $v : T$ (without any further action)
- Structural typing: for $T \leq S$, S must provide a compatible method for each method provided by T
- Thus, we define:

denied < *avail*

$$\frac{T, S \in \mathcal{T} \qquad \forall a \in \mathbb{A}^* : T(a) \leq S(a)}{T \leq S}$$

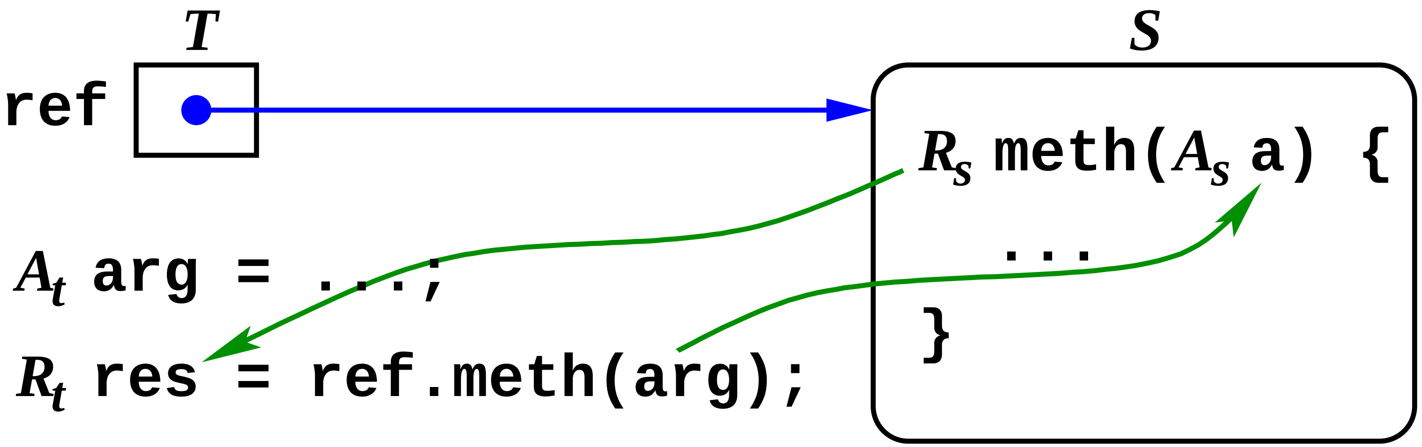
$$\frac{\begin{array}{c} t = (\pi_t, \sigma_t) \in \mathcal{MS} \\ s = (\pi_s, \sigma_s) \in \mathcal{MS} \\ \pi_t \leq \pi_s \\ \sigma_t \neq \bot \wedge \sigma_s \neq \bot \;\Rightarrow\; \sigma_t \leq \sigma_s \end{array}}{t \leq s}$$

$$\frac{\begin{array}{c} t = (A_t, R_t) \in \mathcal{M} \\ s = (A_s, R_s) \in \mathcal{M} \\ A_s \leq A_t \\ R_t \leq R_s \end{array}}{t \leq s}$$

Types ...

Covariance and contravariance

- Example:
  - `interface` T { R_t `meth(`A_t`);` }
  - `interface` S { R_s `meth(`A_s`);` } with $T \leq S$
- Situation when calling `meth`:



R_s **meth(**A_s **a) {**
...
}

A_t **arg = ...;**
R_t **res = ref.meth(arg);**

  - passing the argument requires $A_s \leq A_t$
  - passing the result requires $R_t \leq R_s$

4.2 Types as Capabilities: The COSMA Type System [11]

Security property

- If $o : T_0$ is assigned to $v : T_n$ via a sequence of casts to types $T_1, ..., T_n$, v allows to call a method m only if **all** T_i allow that
- I.e., no amplification of authority
- Property holds recursively:

```
class T0 {
   R0 m() {
      return new R0();
   }
}
class R0 {
   void m1() { ... }
   void m2() { ... }
}
```

```
interface T1 {
   R1 m();
}
interface R1 {
   void m1();
}
```

```
T0 v0 = new T0();
T1 v1 = v0;
v1.m().m1(); // OK
v1.m().m2(); // Err
T0 v2 = v1;  // Err
```

Types as Capabilities: The COSMA Type System ...

Optional methods

- Type system is still too restrictive (no downcast at all)
- We want to allow a **limited** downcast
  - i.e. only if the source type permits it
- Additional assertion: *optional* $\in \mathcal{A}$
  - *optional* means that the method may or may not be available
    - calling the method is permitted,
    - but there is no guarantee that the method is available
  - order: *denied* $<$ *optional* $<$ *avail*
- We need a new “legal cast” relation: $\prec$
  - $T \prec S \;\; \Leftrightarrow$ the *static* type check will allow a cast from S to T (although it may fail at runtime)

Types as Capabilities: The COSMA Type System ...

Legal cast relation

- We allow a (down)cast from S to T, even if some method m is
  - *available* in T and *optional* in S, or
  - *optional* in T and *denied* in S

$$\frac{\begin{array}{c} T, S \in \mathcal{T} \\ \forall m \in \mathbb{A}^* : T(m) \prec S(m) \end{array}}{T \prec S}$$

$$\frac{\begin{array}{c} t = (\pi_t, \sigma_t) \in \mathcal{MS} \\ s = (\pi_s, \sigma_s) \in \mathcal{MS} \\ \neg(\pi_t = \mathit{avail} \wedge \pi_s = \mathit{denied}) \\ \sigma_t \neq \bot \wedge \sigma_s \neq \bot \;\Rightarrow\; \sigma_t \prec \sigma_s \end{array}}{t \prec s}$$

$$\frac{\begin{array}{c} t = (A_t, R_t) \in \mathcal{M} \\ s = (A_s, R_s) \in \mathcal{M} \\ A_s \prec A_t \\ R_t \prec R_s \end{array}}{t \prec s}$$

Types as Capabilities: The COSMA Type System ...

Runtime actions

- If we have $T \prec S$, but $T \not\leq S$, we need to perform some actions at runtime
- $\exists m : m$ is *available* in T and *optional* in S:
  - we need a type check to ensure that m is actually available
- $\exists m : m$ is *optional* in T and *denied* in S:
  - we need a membrane to ensure that m cannot be called via T
  - let M be the type of this membrane
  - requirement: $T \leq M$, M doesn't grant more authority than S
  - problem: all $x \in \mathcal{A}$ with *optional* $\leq x$ permit calling m
  - solution: new element *unavailable* with *optional* $<$ *unavail*
    - asserts that the object does **not** provide the method

Types as Capabilities: The COSMA Type System ...

Creating membranes

- We first extend the $\prec$ relation properly:

$$\frac{\begin{array}{c} t = (\pi_t, \sigma_t) \in \mathcal{MS} \\ s = (\pi_s, \sigma_s) \in \mathcal{MS} \\ \neg(\pi_t = \mathit{avail} \wedge (\pi_s = \mathit{denied} \vee \pi_s = \mathit{unavail})) \\ \sigma_t \neq \bot \wedge \sigma_s \neq \bot \;\Rightarrow\; \sigma_t \prec \sigma_s \end{array}}{t \prec s}$$

- Next, we need a rule to determine the membrane type
  - let $T \cap_r S$ be the smallest subtype of T that does not grant more rights than S
  - for contravariance: $T \cap^r S$ is the largest supertype of S that does not grant more rights than T

Types as Capabilities: The COSMA Type System ...

Restricting method permissions

- Partial order: *denied* $<$ *optional* $<$ *avail*
  unavail

- $t \cap_r s$:

$t \setminus s$	*denied*	*optional*	*avail*	*unavail*
denied	*denied*	*denied*	*denied*	*denied*
optional	*unavail*	*optional*	*optional*	*unavail*
avail	- - -	*avail*	*avail*	- - -
unavail	*unavail*	*unavail*	*unavail*	*unavail*

- $t \cap^r s$:

$t \setminus s$	*denied*	*optional*	*avail*	*unavail*
denied	*denied*	*denied*	*denied*	*unavail*
optional	*denied*	*optional*	*avail*	*unavail*
avail	*denied*	*optional*	*avail*	*unavail*
unavail	*denied*	*denied*	*denied*	*unavail*

Types as Capabilities: The COSMA Type System ...

Restricted subtype

$$\frac{T, S \in \mathcal{T}}{T \cap_r S = \lambda m.(T(m) \cap_r S(m))}$$

$$\frac{\begin{array}{c} t = (\pi_t, \sigma_t) \in \mathcal{MS} \\ s = (\pi_s, \sigma_s) \in \mathcal{MS} \end{array}}{t \cap_r s = (\pi_t \cap_r \pi_s, \sigma_t \cap_r \sigma_s)}$$

$$\frac{s \in \mathcal{M} \cup \{\perp\}}{\perp \cap_r s = \perp} \qquad \frac{t \in \mathcal{M} \cup \{\perp\}}{t \cap_r \perp = \perp}$$

$$\frac{\begin{array}{c} t = (A_t, R_t) \in \mathcal{M} \\ s = (A_s, R_s) \in \mathcal{M} \end{array}}{t \cap_r s = (A_s \cap^r A_t, R_t \cap_r R_s)}$$

Types as Capabilities: The COSMA Type System ...

Restricted supertype

$$\frac{T, S \in \mathcal{T}}{T \cap^r S = \lambda m.(T(m) \cap^r S(m))}$$

$$\frac{\begin{array}{c} t = (\pi_t, \sigma_t) \in \mathcal{MS} \\ s = (\pi_s, \sigma_s) \in \mathcal{MS} \end{array}}{t \cap^r s = (\pi_t \cap^r \pi_s, \sigma_t \cap^r \sigma_s)}$$

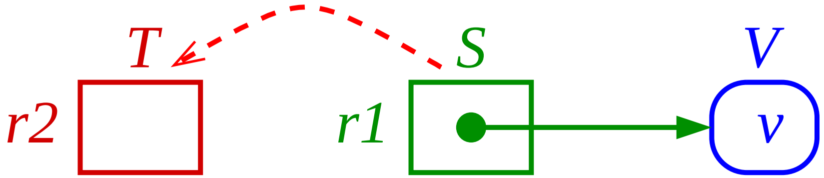
$$\frac{s \in \mathcal{M} \cup \{\perp\}}{\perp \cap^r s = \perp} \qquad \frac{t \in \mathcal{M} \cup \{\perp\}}{t \cap^r \perp = \perp}$$

$$\frac{\begin{array}{c} t = (A_t, R_t) \in \mathcal{M} \\ s = (A_s, R_s) \in \mathcal{M} \end{array}}{t \cap^r s = (A_s \cap_r A_t, R_t \cap^r R_s)}$$

Types as Capabilities: The COSMA Type System ...

Runtime cast actions

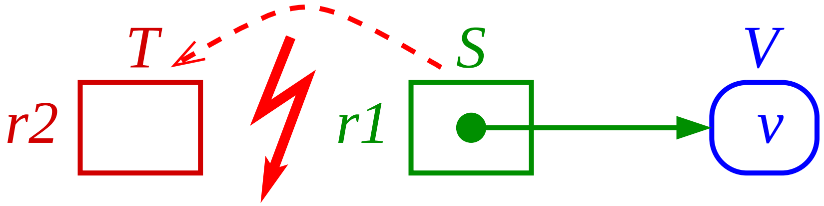
- Situation:



Types as Capabilities: The COSMA Type System ...

Runtime cast actions

- Situation:

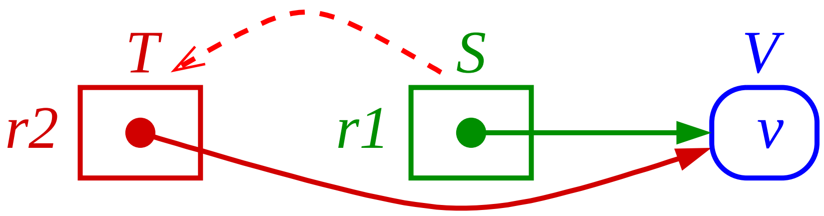


- $T \not\preceq S$: static type error!
    - is already determined when loading a component

Types as Capabilities: The COSMA Type System ...

Runtime cast actions

- Situation:

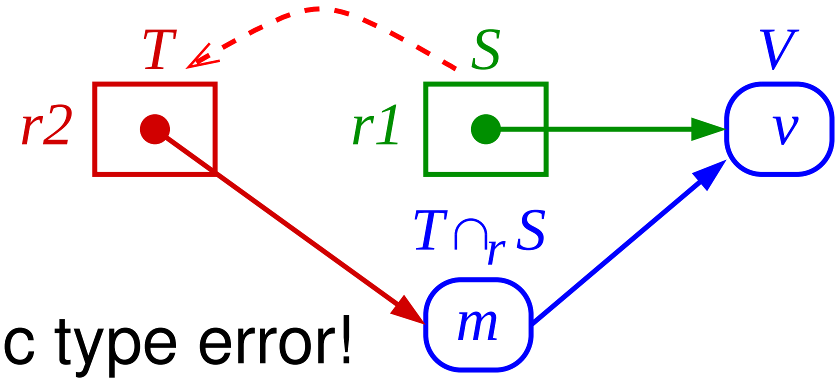


- $T \not\leq S$: static type error!
  - is already determined when loading a component
- $T \leq S$: assign reference as is
  - access restrictions of S are also enforced by T

Types as Capabilities: The COSMA Type System ...

Runtime cast actions

- Situation:

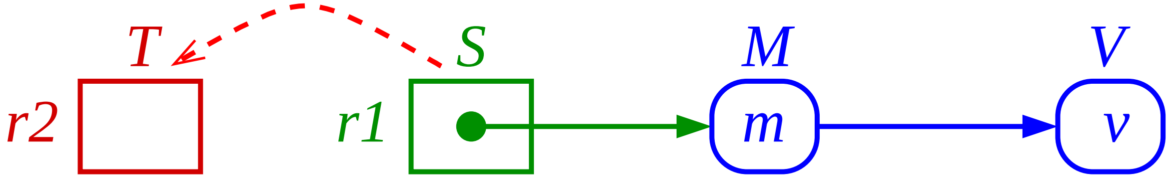


- $T \not\leq S$: static type error!
  - is already determined when loading a component
- $T \leq S$: assign reference as is
  - access restrictions of S are also enforced by T
- Otherwise: create a membrane with type $T \cap_r S$
  - access restrictions of S are enforced by membrane and T

Types as Capabilities: The COSMA Type System ...

Cascading membranes

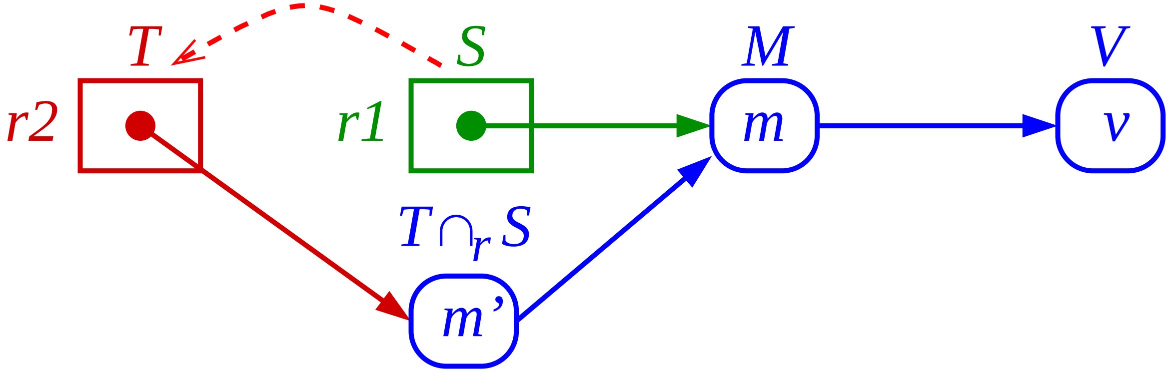
- What happens if the assigned object already is a membrane?
- Situation:



Types as Capabilities: The COSMA Type System ...

Cascading membranes

- What happens if the assigned object already is a membrane?
- Situation:

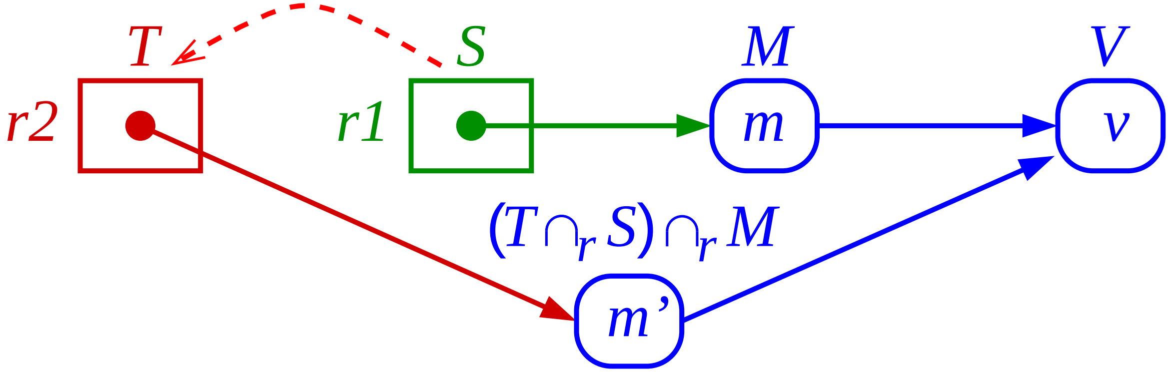


- Cascading membranes can lead to severe inefficiency
  - method calls are forwarded multiple times

Types as Capabilities: The COSMA Type System ...

Cascading membranes

- What happens if the assigned object already is a membrane?
- Situation:



- Cascading membranes can lead to severe inefficiency
  - method calls are forwarded multiple times
- Solution: new membrane includes restrictions of M
  - can forward calls directly to the real object

4.3 Extensions

- Remove security restrictions inside a single component
  - introduce security contexts and a generic permission "*local*"
  - a reference that assures that the object is in the local context can be downcasted without limitation
- Add classes to the type system
  - direct access to attributes is allowed via a *local* reference
- Add array types
  - array modeled as class with `read()` and `write()` method
- Allow unsafe casts, i.e. unsafe covariant types
  - i.e. if S is subtype of T, allow S[] being used as T[]
  - problem: T[] has `write(`T `e)`, while S[] has `write(`S `e)`
    - S[] is not a subtype of T[], since S is not a supertype of T
  - may result in a runtime type error when `write()` is called

4.3 Extensions ...

- Additional generic permissions, e.g. "*transferable*"
  - (only) a *transferable* reference can be passed to a different context
  - allows implementation of *confined* types [12]
  - e.g., objects of a class declared as *non-transferrable* can never be accessed from another context
- Unifying structural and nominal typing [13][14]
  - advantage of structural typing: no need to explicitly declare subtype relationship ("*implements*")
  - problem of structural typing: cannot express semantic restrictions
  - solution: type system allows to specify a semantic category for each method

5 Conclusion and Future Work

- Software systems should obey the POLA
- Capabilities combine designation with authority
- Object capability systems use references as capabilities
  - fine grained access control requires the use of membranes
- Types can serve as a specification of fine grained access rights
  - type system must not allow amplification of rights
  - (restricted) downcast is possible by introducing membranes
  - often, access rights need not be checked at runtime

- Future work:
  - extension of type system (e.g., revocation)
  - full implementation of a virtual machine using the type system
    - including modular operating system

References

[1] M. S. Miller and J. S. Shapiro, "Paradigm Regained: Abstraction Mechanisms for Access Control," in Advances in Computing Science - ASIAN 2003. Progamming Languages and Distributed Computation, ser. LNCS, vol. 2896. Springer, 2003, pp. 224–242.

[2] "Permissions vs. Authority" `https://everything2.com/title/Permission+versus+Authority`

[3] M. S. Miller, "Robust composition: Towards a unified approach to access control and concurrency control," Ph.D. Thesis, Johns Hopkins University, Baltimore, Maryland, May 2006.

[4] M. S. Miller, K. P. Yee, and J. Shapiro, "Capability Myths Demolished," Systems Research Laboratory, Johns Hopkins University, Technical Report SRL2003-02, 2003.

[5] T. Murray, "Analysing object-capability security," in Proc. of the Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security, Pittsburgh, PA, USA, Jun. 2008, pp. 177–194.

[6] M. S. Miller, B. Tulloh, and J. S. Shapiro, "The Structure of Authority: Why Security Is not a Separable Concern," in Proc. 2nd Intl. Conf. on Multiparadigm Programming in Mozart/Oz. Charleroi, Belgium: Springer, 2004, pp. 2–20.

[7] A. Mettler, D. Wagner, and T. Close, "Joe-E: A Security-Oriented Subset of Java," in Network and Distributed Systems Symposium. Internet Society, Jan. 2010, pp. 357–374.

[8] "Walnut/Secure Distributed Computing." `http://wiki.erights.org/wiki/Walnut/Secure_Distributed_Computing#Capabilities`

[9] M. Stiegler, "The E Language in a Walnut." `http://www.skyhunter.com/marcs/ewalnut.html`

[10] B. C. Pierce, Types and programming languages. MIT Press, 2002.

[11] R. Wismüller and D. Ludwig, "Secure Cooperation of Untrusted Components Using a Strongly Typed Virtual Machine." International Journal on Advances in Security, 12(1&2):53-68, June 2019.

[12] C. Grothoff, J. Palsberg and J. Vitek, "Encapsulating Objects with Confined Types." ACM SIGPLAN Notices 36(11), Aug. 2001, pp. 32–44.

[13] D. Malayeri and J. Aldrich,"Integrating Nominal and Structural Subtyping," in Proceedings of the European Conference on Object-Oriented Programming (ECOOP '08), July 2008, pp. 260–284.

[14] B. H. Liskov, "A Behavioral Notion of Subtyping," ACM Transactions on Programming Languages and Systems, Volume 16, Issue 6, Nov. 1994, pp 1811–1841.