# Analysing Object-Capability Patterns with Mur$\phi$

Deian Stefan

Stanford University

# OCap Model and Motivation

- **Object-Capability (OCap) model is a promising approach to secure programming**
  - Access control is *not* a separate concern
- **Abstractions, or *patterns*, are used to compose systems**
  - Systems built using the *Principle of Least Authority* (POLA)
  - E.g., solitaire should not have the authority to write to any file on your disk
- **Motivation**
  - OCap model and it's patterns of cooperation are increasingly being applied
  - Formal analysis of OCap model is limited

# What is the OCap model?

- **Object**
  - Instance: Code & State (mutable references to other objects)
  - Data
- **Capability: reference to non-data, i.e., instance**
- **Reference graph $\equiv$ access graph**
  - Interaction *only* by sending messages
- **Graph dynamics and connectivity**
  - Initial conditions
  - Parenthood: if $A$ creates $B \rightarrow$ only connection to $B$ is through $A$
  - Endowment: when $A$ creates $B$ it can give it capabilities it owns
  - Introduction: $A$ has capability to $B$ and $C$, but $B$ and $C$ are not directly connected. $A$ can give $B$ capability to $C$

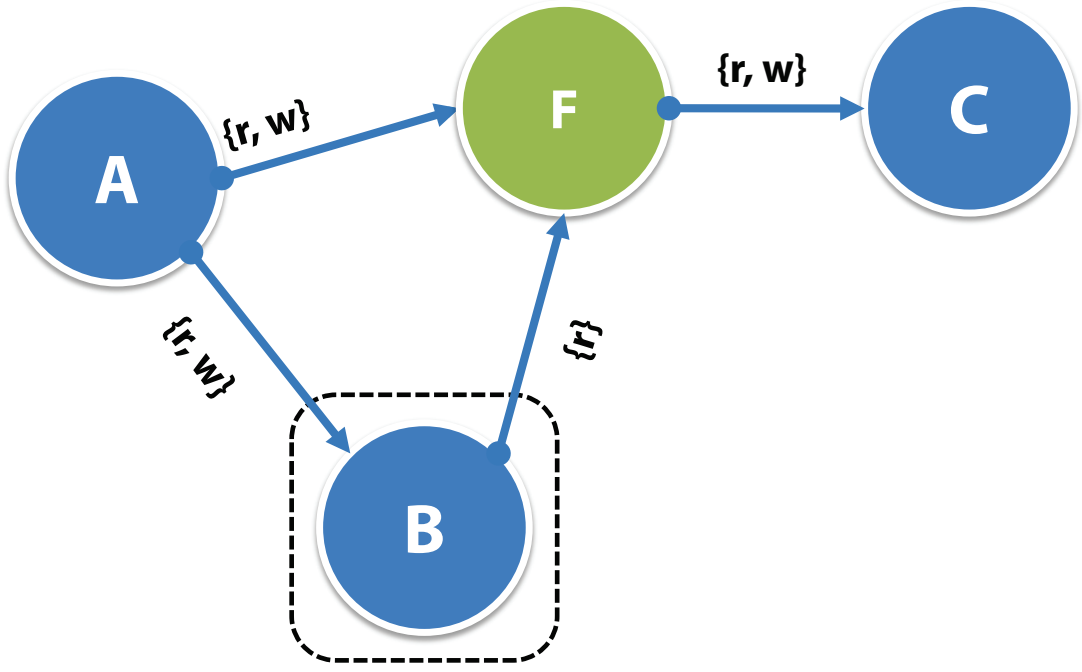
# Patterns & their security properties

- **Membrane & membrane forwarder pattern**
  - Object on transitive path acquires unwrapped capability
- **Revocable forwarder**
  - Object still capable of sending message after Revoker revokes access
- **Revocable membrane forwarder**
  - Any object on the transitive path still able to send message after revocation
- **Sealer and unsealer**
  - Object gets direct access to “sealed” value

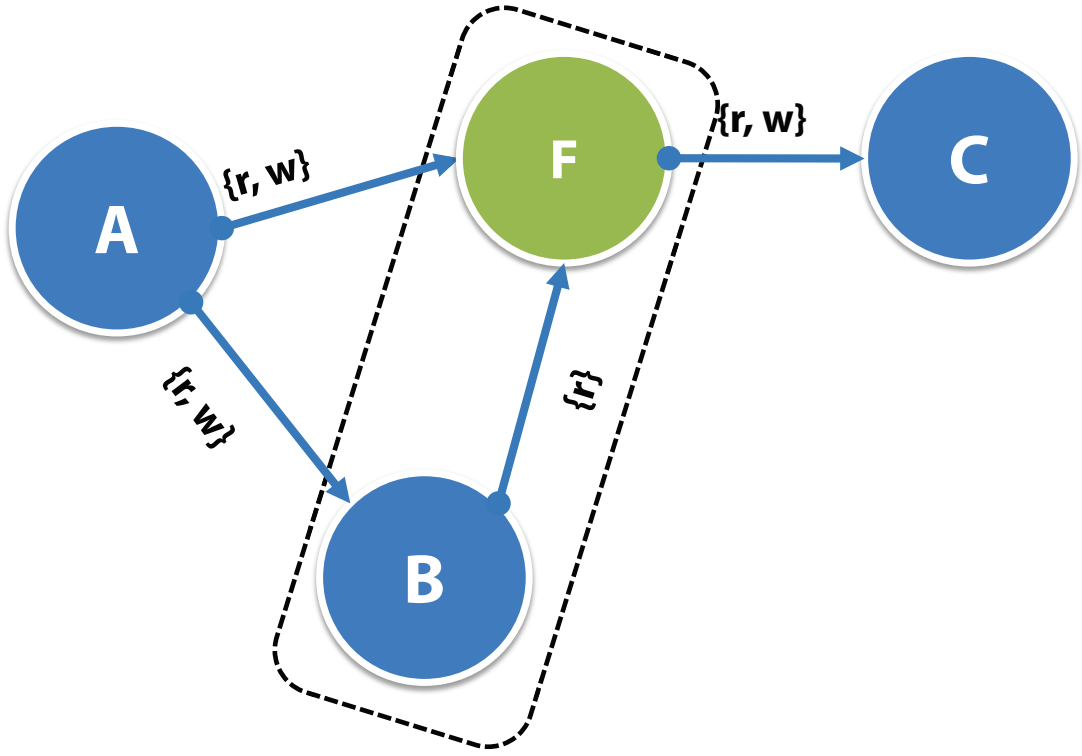
# Modeling approach

- **Call-return framework**
  - Objects in idle state may receive messages
  - Objects in blocking state await `Return` message
  - Once an object sends a message, it blocks
- **Programming Languages: single-message simulates sequential call-stack-like approach**
- **Operating/Distributed Systems: multiple-message simulates concurrency**

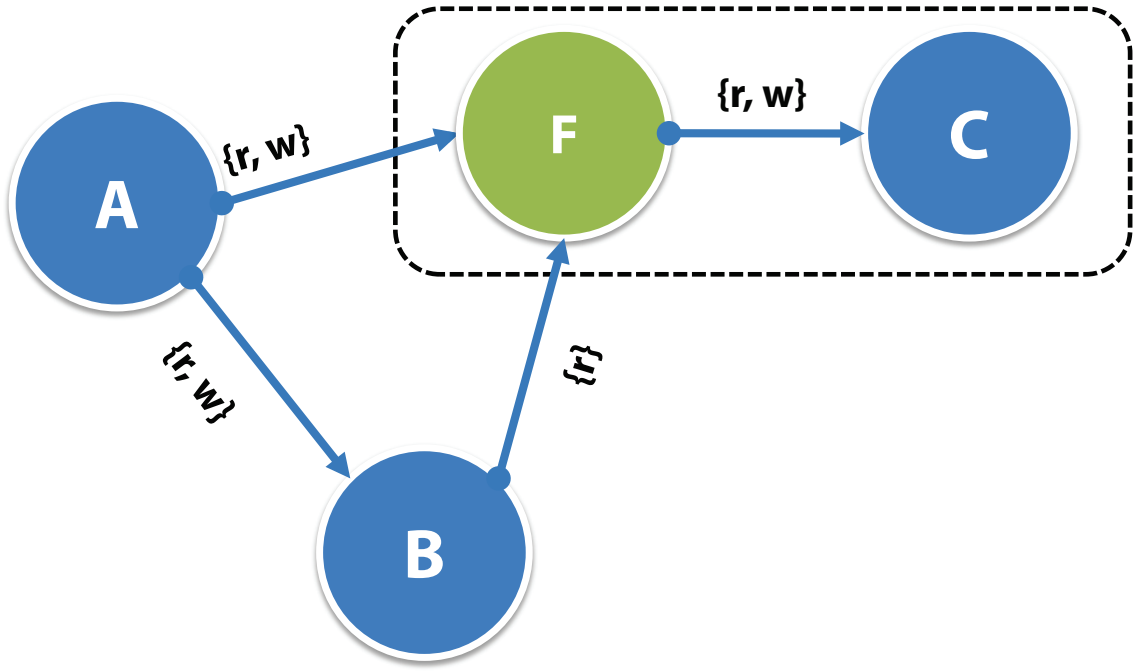
# Membrane Pattern



# Membrane Pattern



# Membrane Pattern



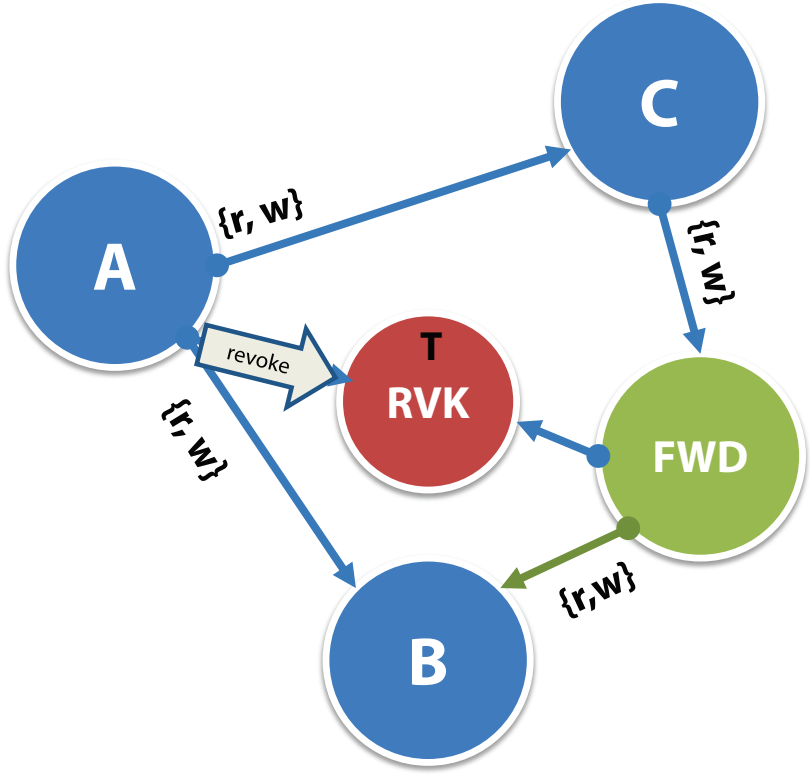
# Results: Membrane Pattern

- **PL/OS setting: no attacks**
- **Should this pattern be used whenever you want transitive read-only?**
  - OCap system: yes
  - Other capability system: *maybe*
- **Attack on TahoeLAFS "transitive-read-only"**

```
tahoe add-alias fun-dir URI:DIR2:lrxr3kzsb...rvxvclweezjuhzq
tahoe add-alias fun-dir-read-only URI:DIR2-RO:4fudupw...mianvfwsrvxvclweezjuhzq
echo hello tahoe | tahoe put - fun-dir:alo #200 OK
tahoe get fun-dir-read-only:alo #hello tahoe
echo hello again | tahoe put - fun-dir-read-only:alo # 500 NotWriteableError
tahoe manifest fun-dir: | head -n1  | tahoe put - fun-dir:silly
tahoe add-alias fun-dir-read-only-ha ‘tahoe get fun-dir-read-only:silly‘
echo hello again | tahoe put - fun-dir-read-only-ha:alo #200 OK
tahoe get fun-dir:alo #hello again
```

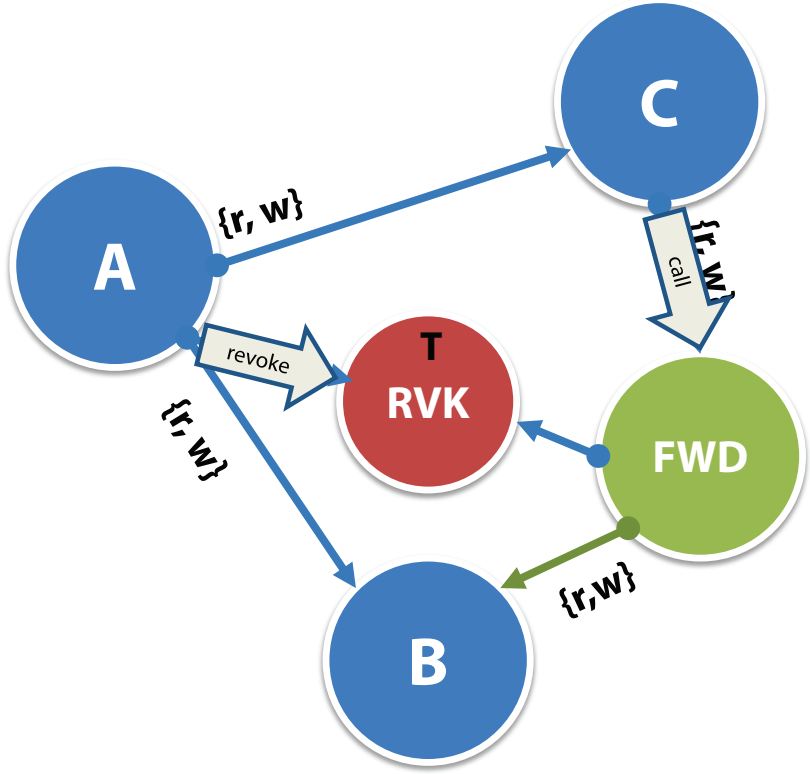
# Results: Revocable Memb. Forwarder

'Vulnerable' to scheduling/network:



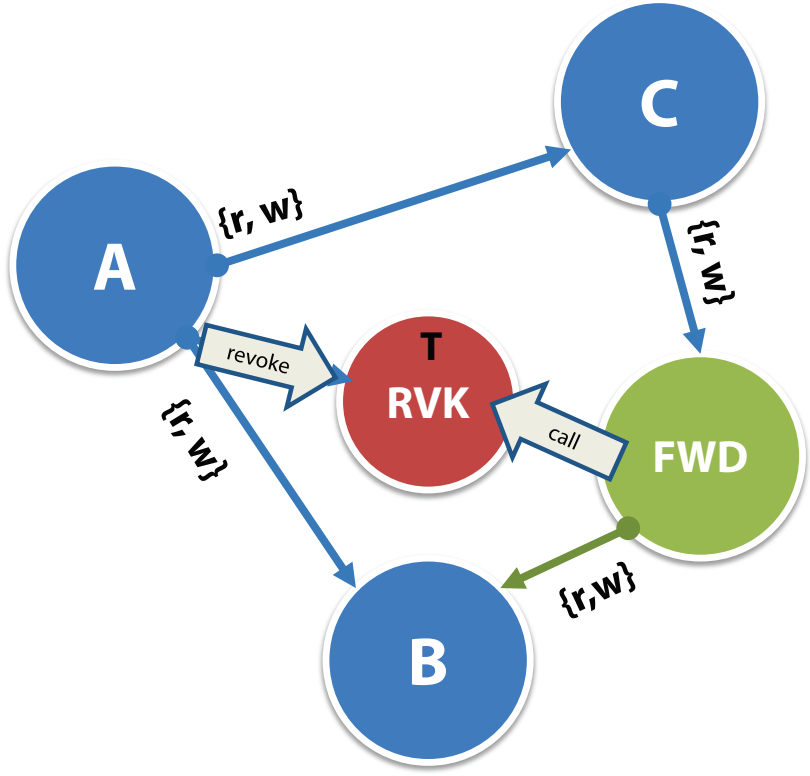
# Results: Revocable Memb. Forwarder

'Vulnerable' to scheduling/network:



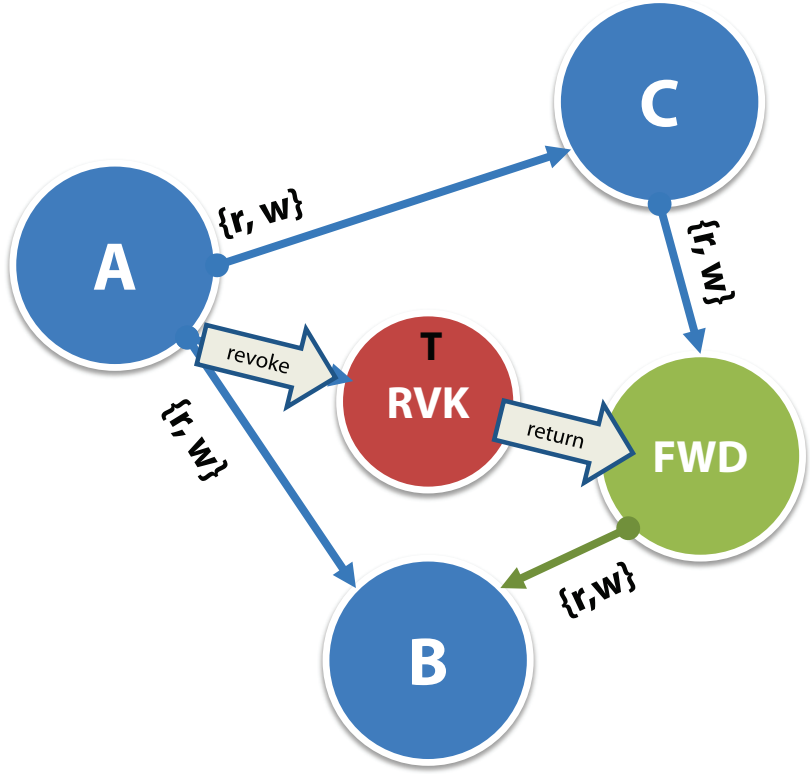
# Results: Revocable Memb. Forwarder

'Vulnerable' to scheduling/network:



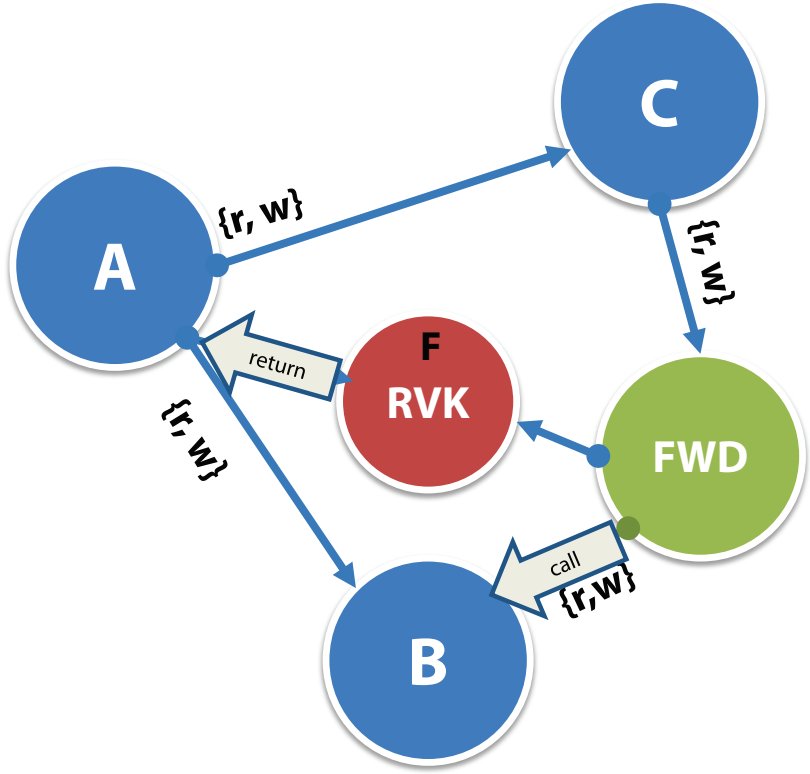
# Results: Revocable Memb. Forwarder

'Vulnerable' to scheduling/network:



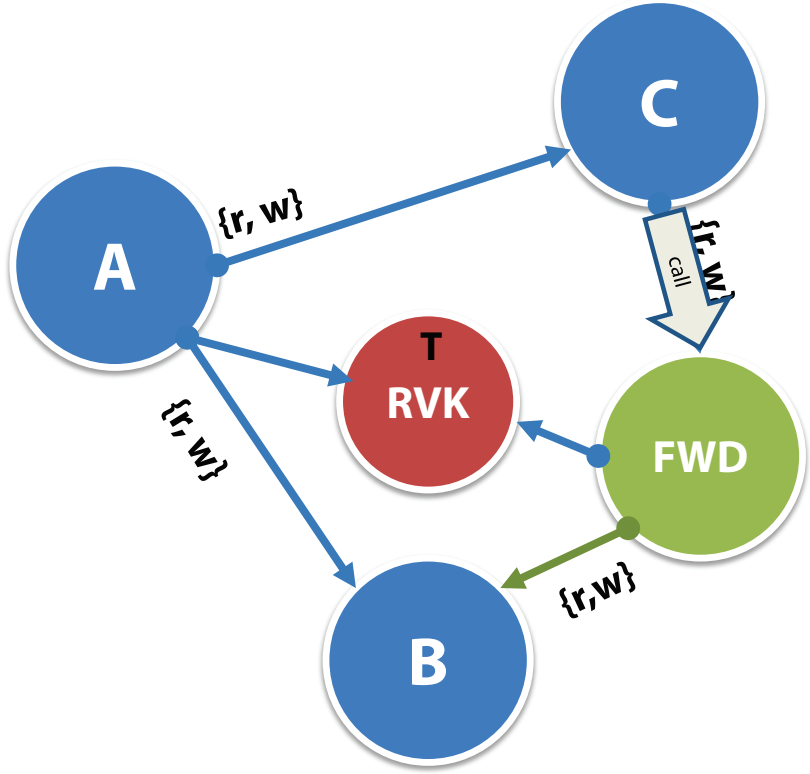
# Results: Revocable Memb. Forwarder

'Vulnerable' to scheduling/network:



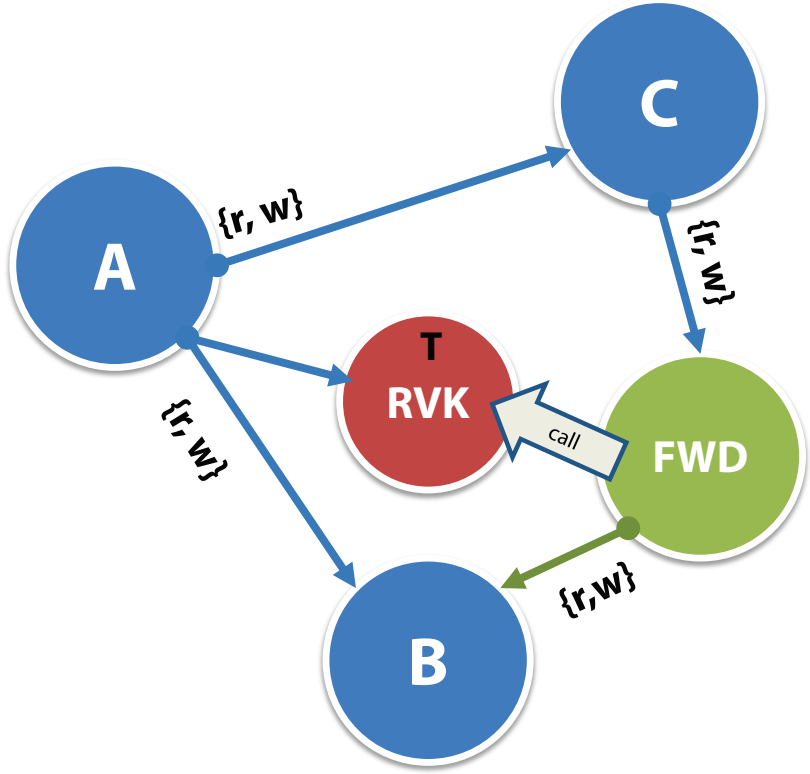
# Results: Revocable Memb. Forwarder

Time of check/time of use vulnerability:



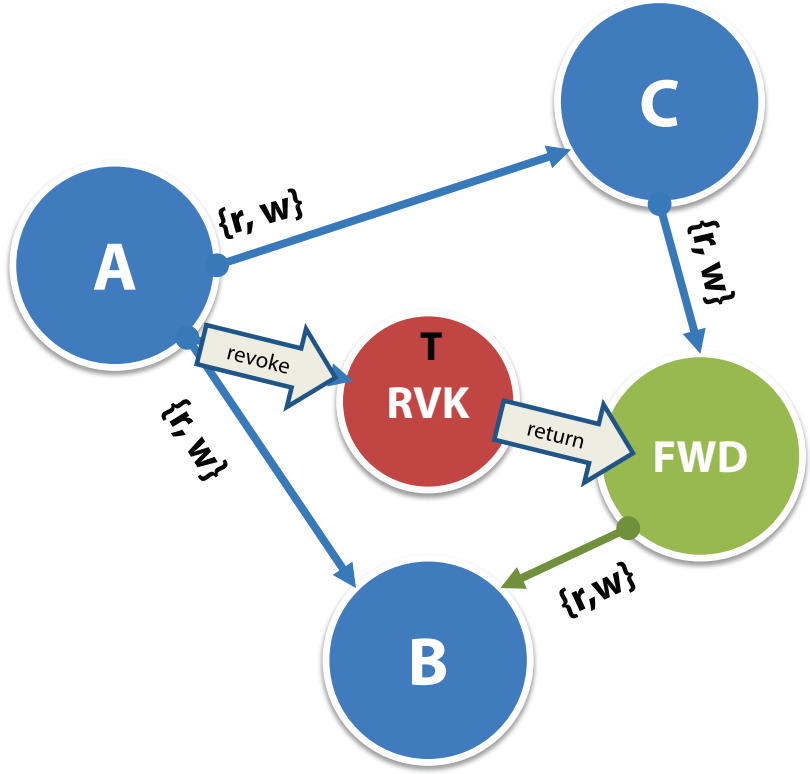
# Results: Revocable Memb. Forwarder

Time of check/time of use vulnerability:



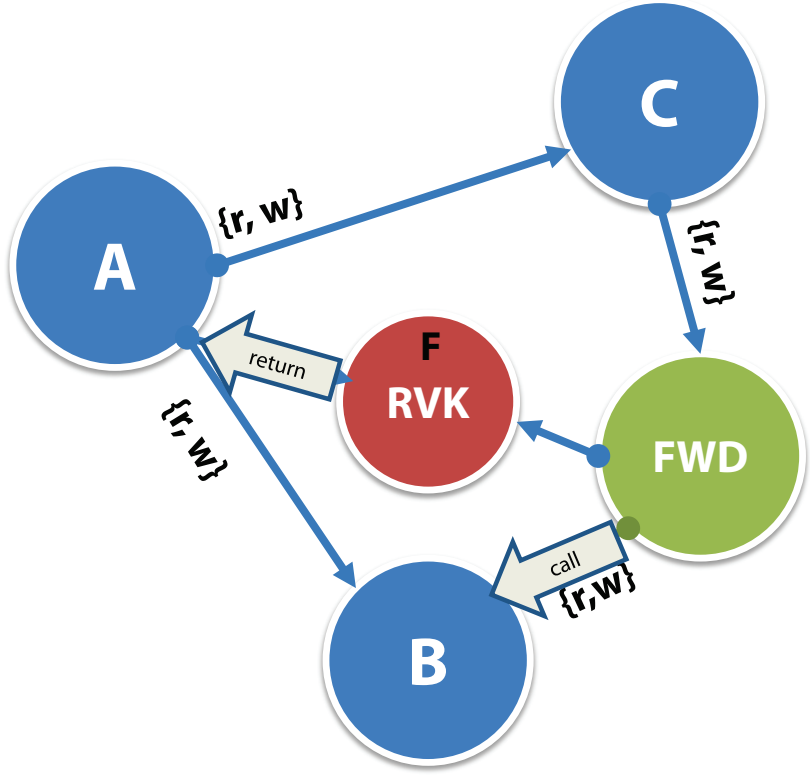
# Results: Revocable Memb. Forwarder

Time of check/time of use vulnerability:



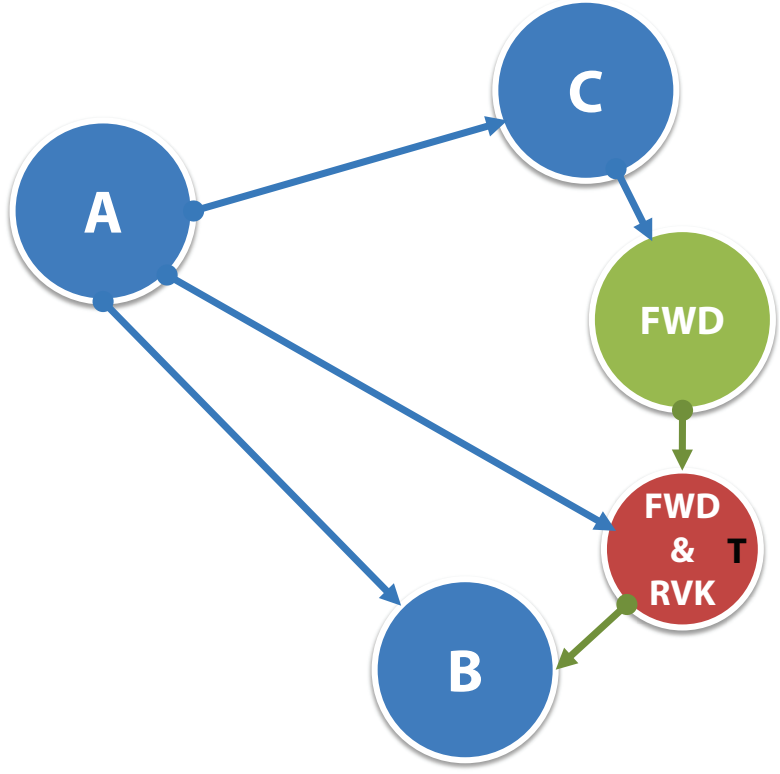
# Results: Revocable Memb. Forwarder

Time of check/time of use vulnerability:



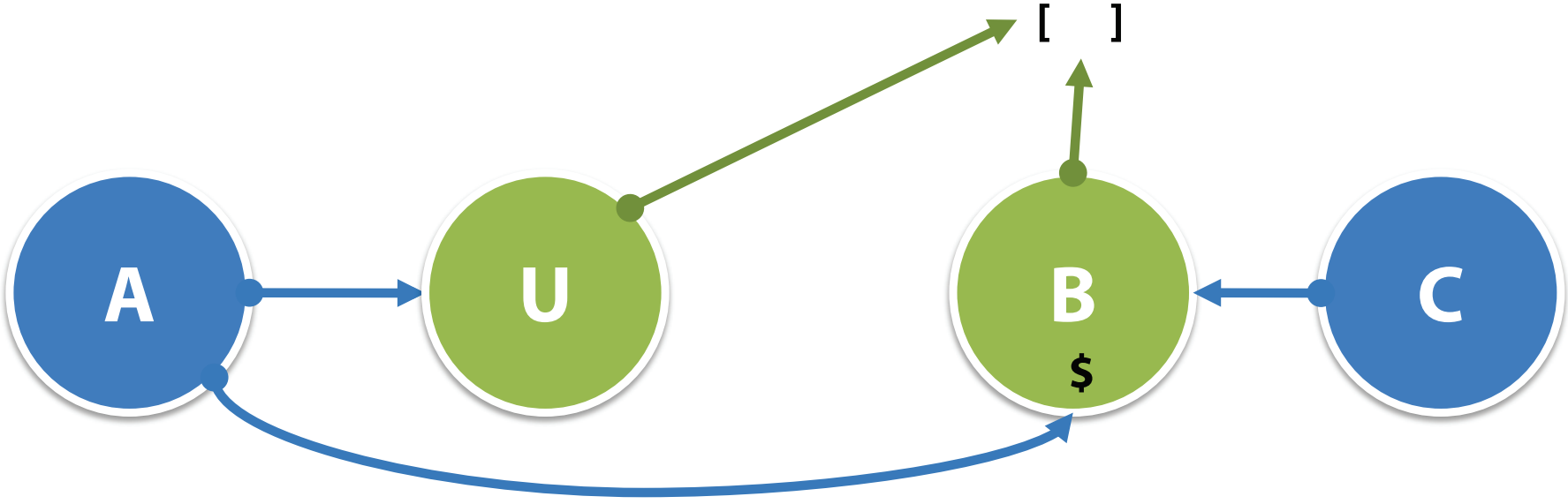
# Results: Revocable Memb. Forwarder

Fixing the time of check/time of use vulnerability:

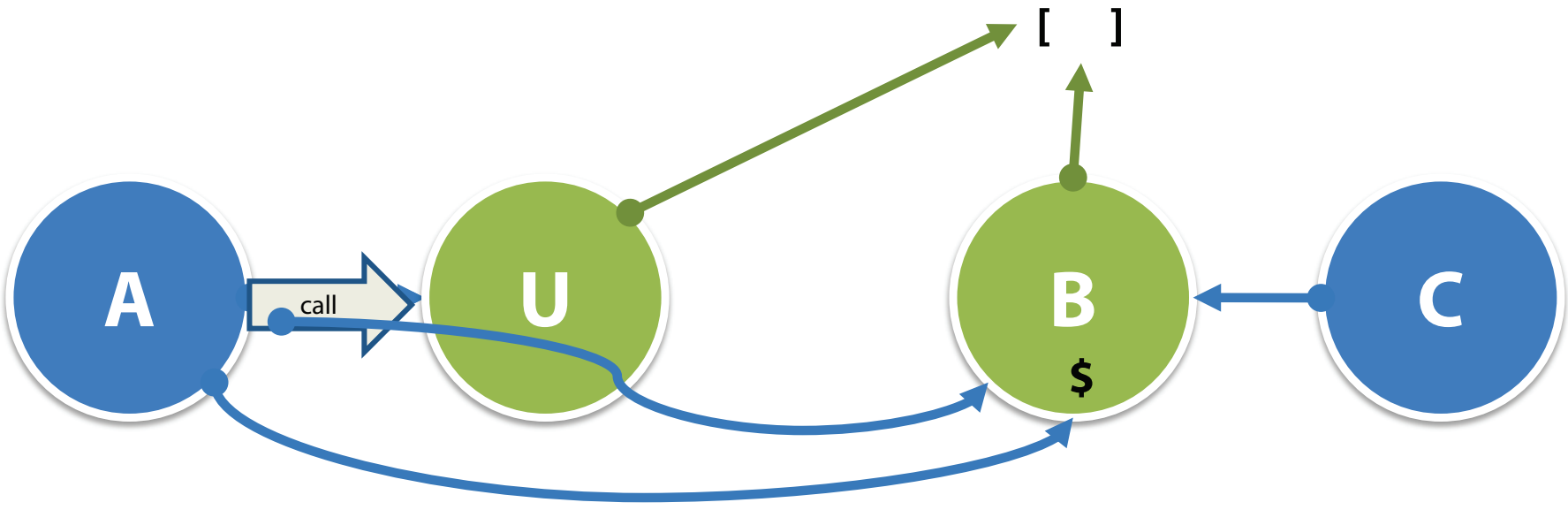



- Forwarder forwards all messages but revoke
- Revoker forwards all messages, handles revoke
- Can wrap messages to allow for forwarding of revoke

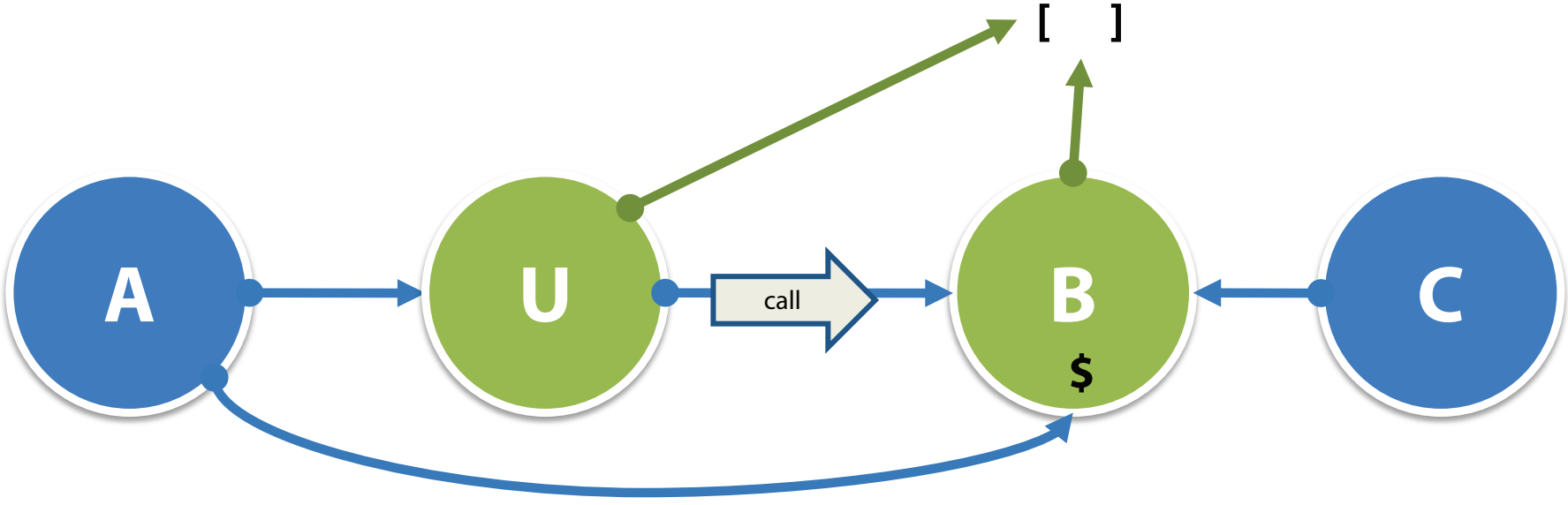
# Sealer/Unsealer pattern



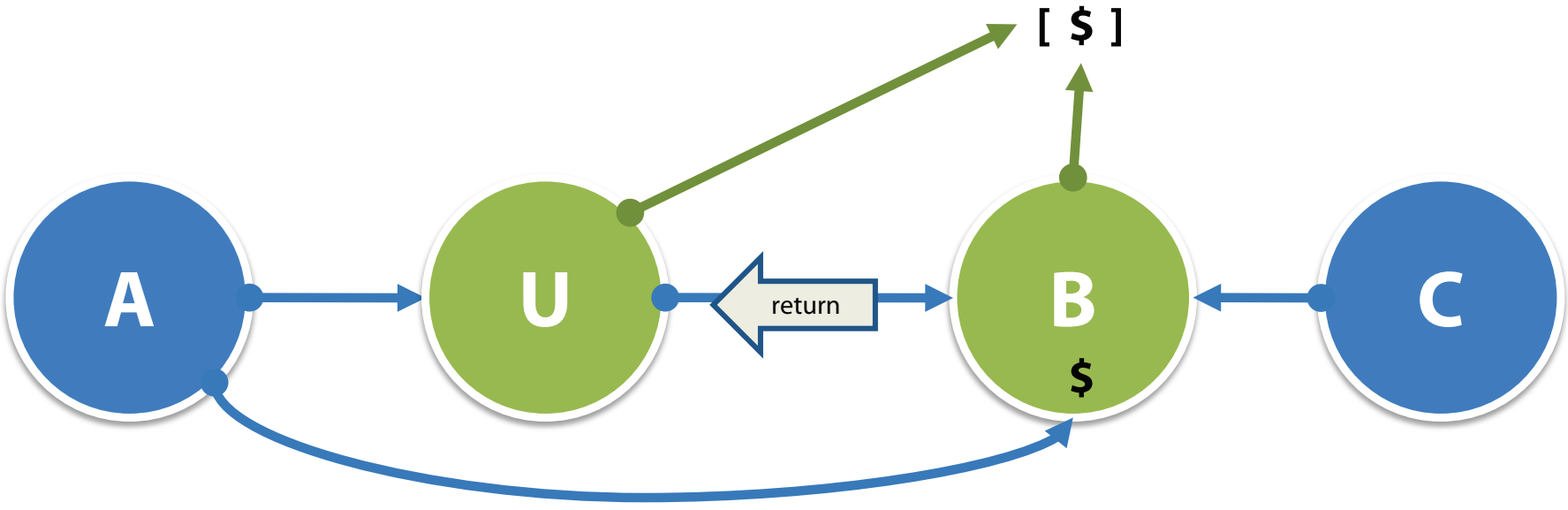
# Sealer/Unsealer pattern



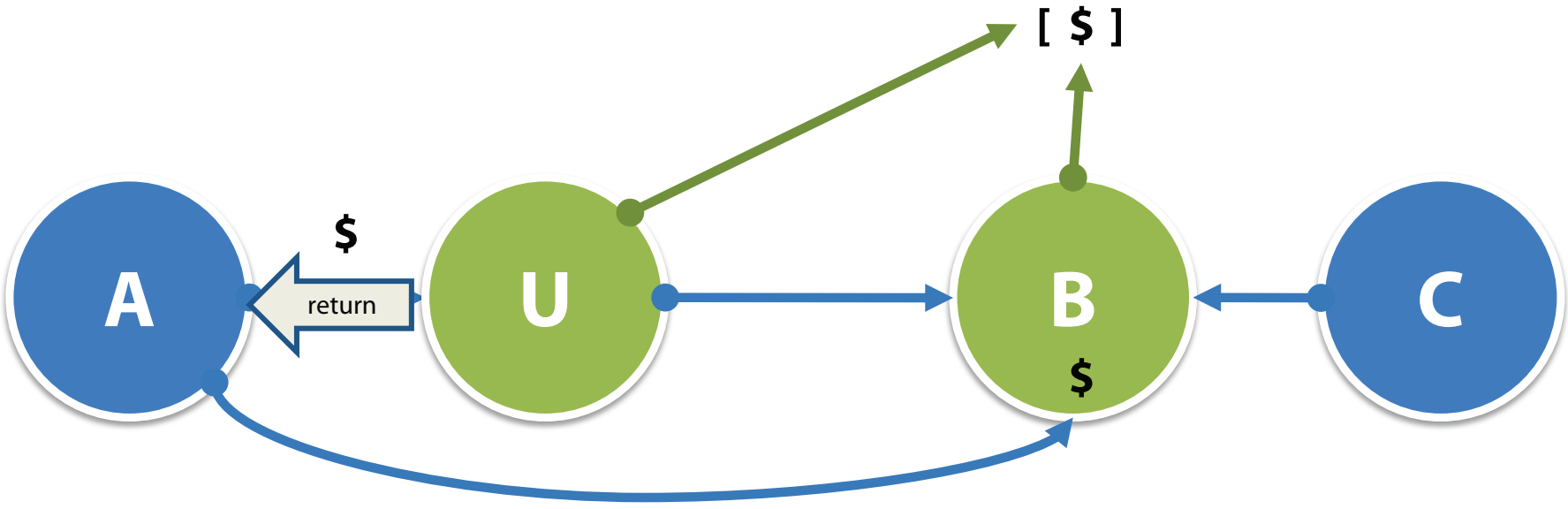
# Sealer/Unsealer pattern



# Sealer/Unsealer pattern

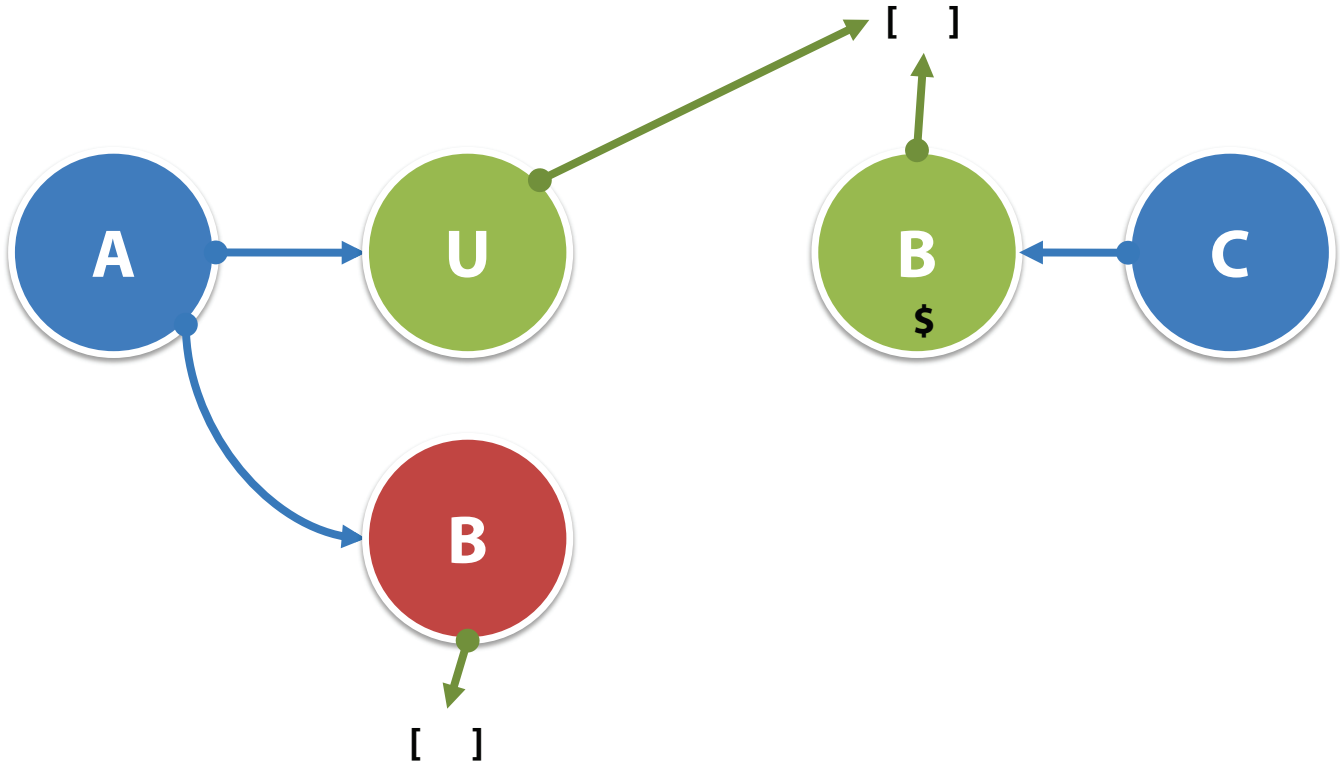


# Sealer/Unsealer pattern



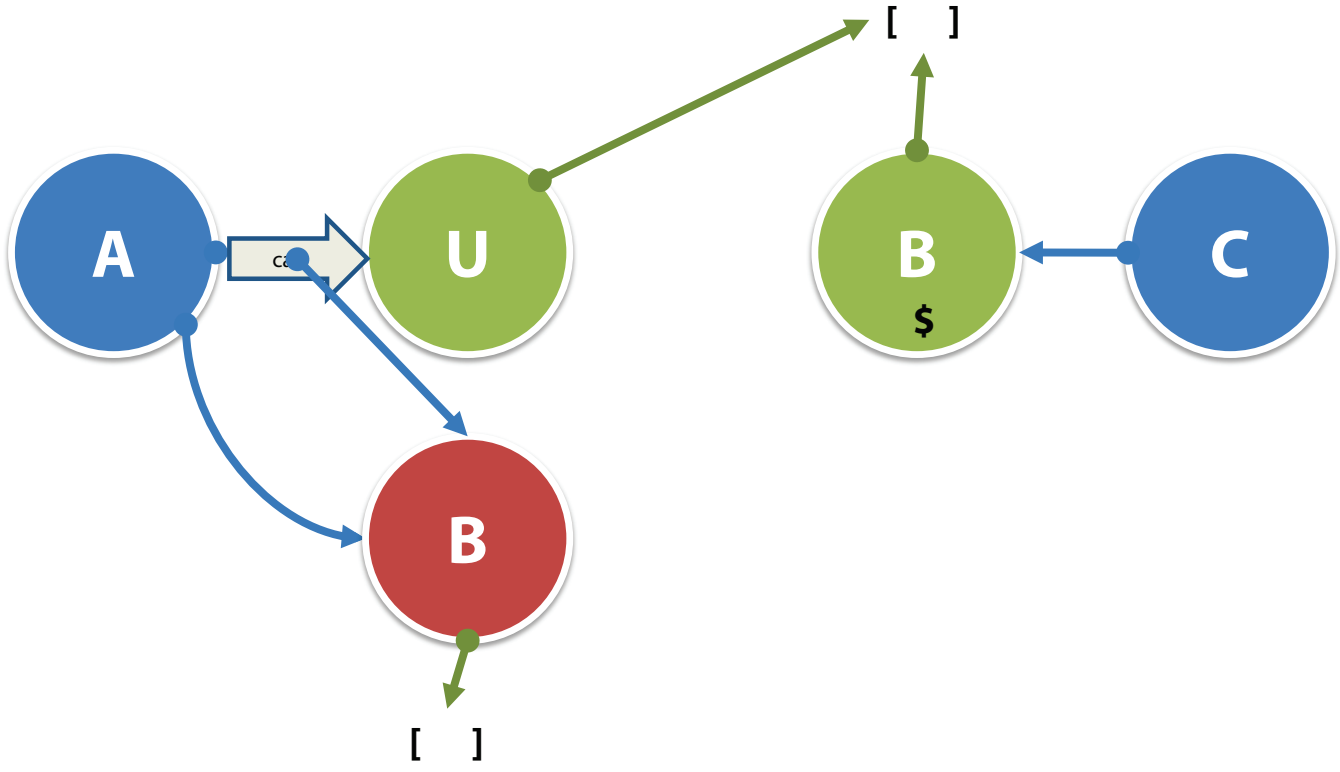
# Results: Sealer/Unsealer pattern

Attack:



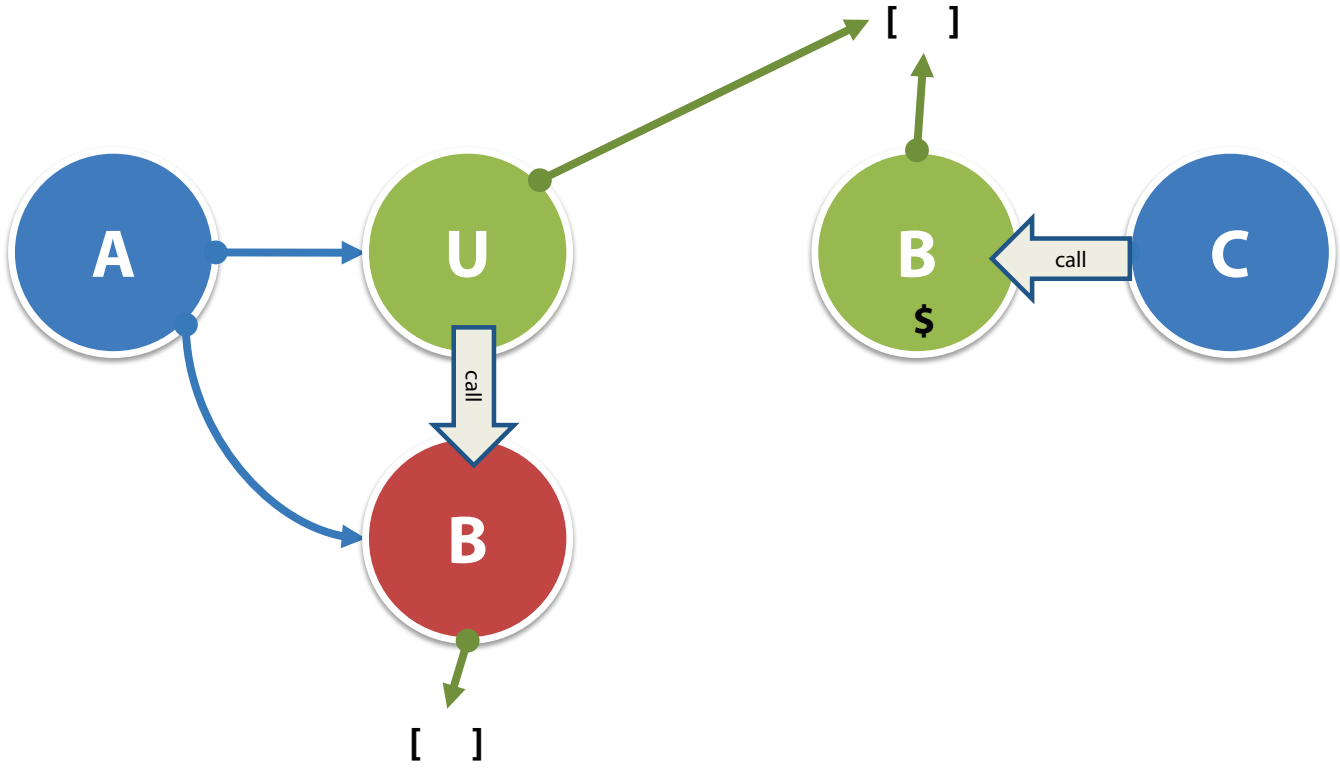
# Results: Sealer/Unsealer pattern

Attack:



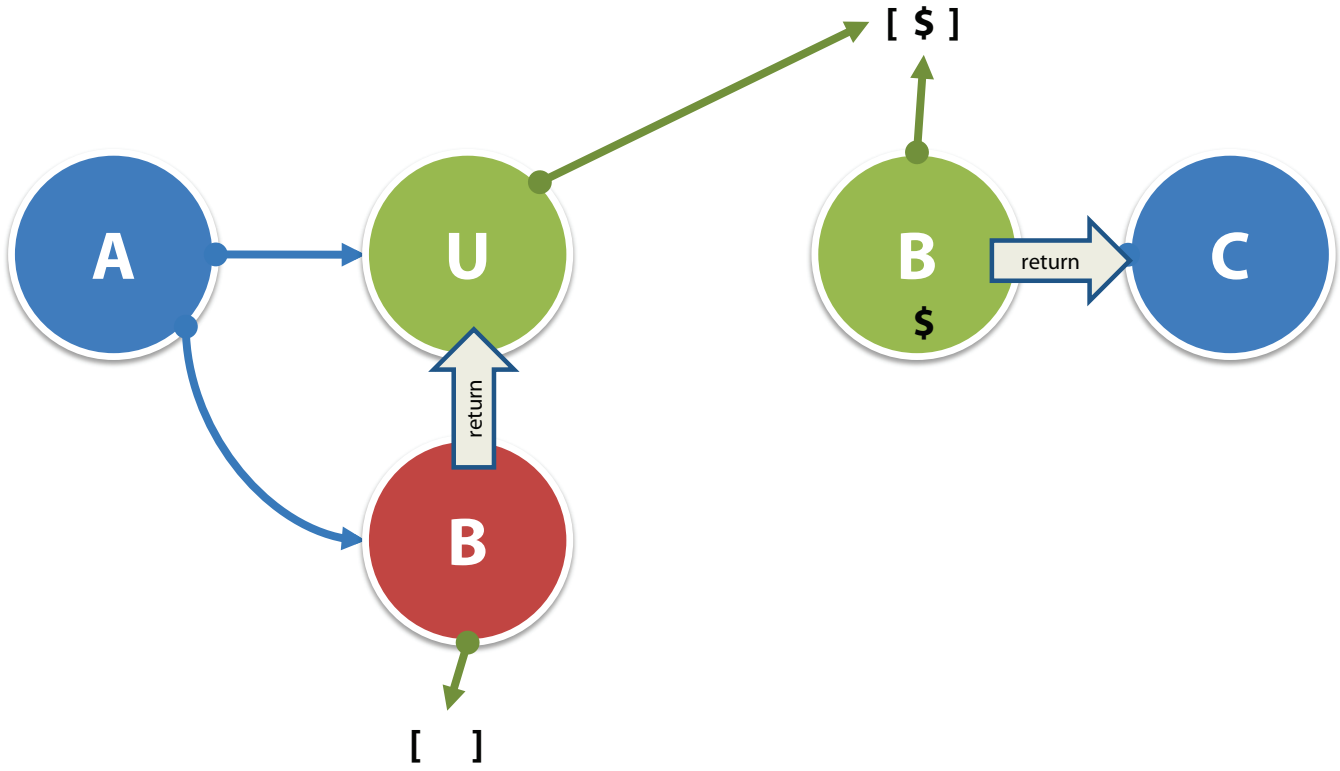
# Results: Sealer/Unsealer pattern

Attack:



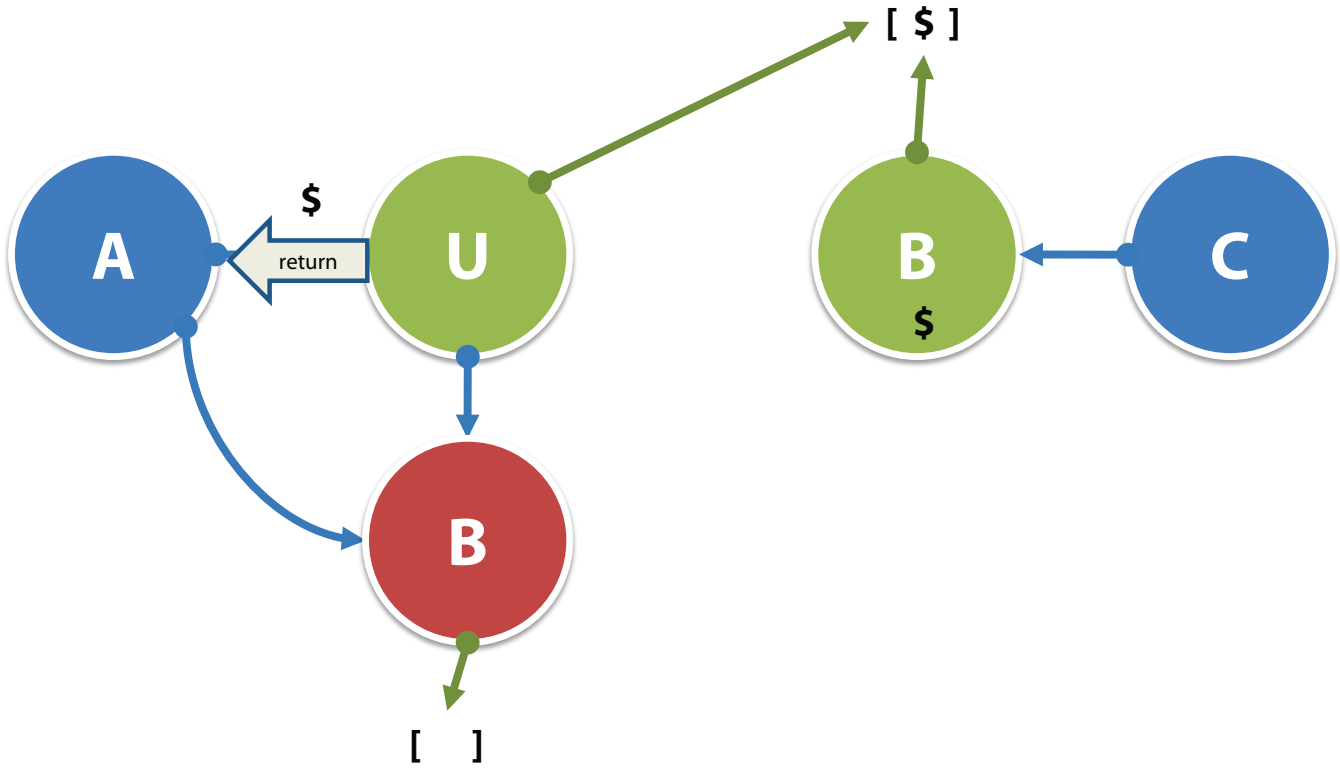
# Results: Sealer/Unsealer pattern

Attack:



# Results: Sealer/Unsealer pattern

Attack:



# Results: Sealer/Unsealer pattern

Fixing the vulnerability. Unsealer should:

- Clear slot
- Call box (that also writes its capability in slot)
- Check the calling-box is slot-modifier box
- Return value only if above is true

# Results: Joe-E Sealer/Unsealer pattern

```
public class SealerUnsealer {
private Object shared;
private Object modifier;

public SealerUnsealer() {}
public Box Seal(Object o){ return new Box(o); }

public Object Unseal(Box box) {
shared=null; box.share();
//FIX: if(!modifier.equals(box)) return null;
return shared;
}
public class Box {
private final Object box_shared;
public Box(Object o) { box_shared = o; }
public void share() {
 shared = box_shared;
 modifier = this;
}
}
```