KU LEUVEN

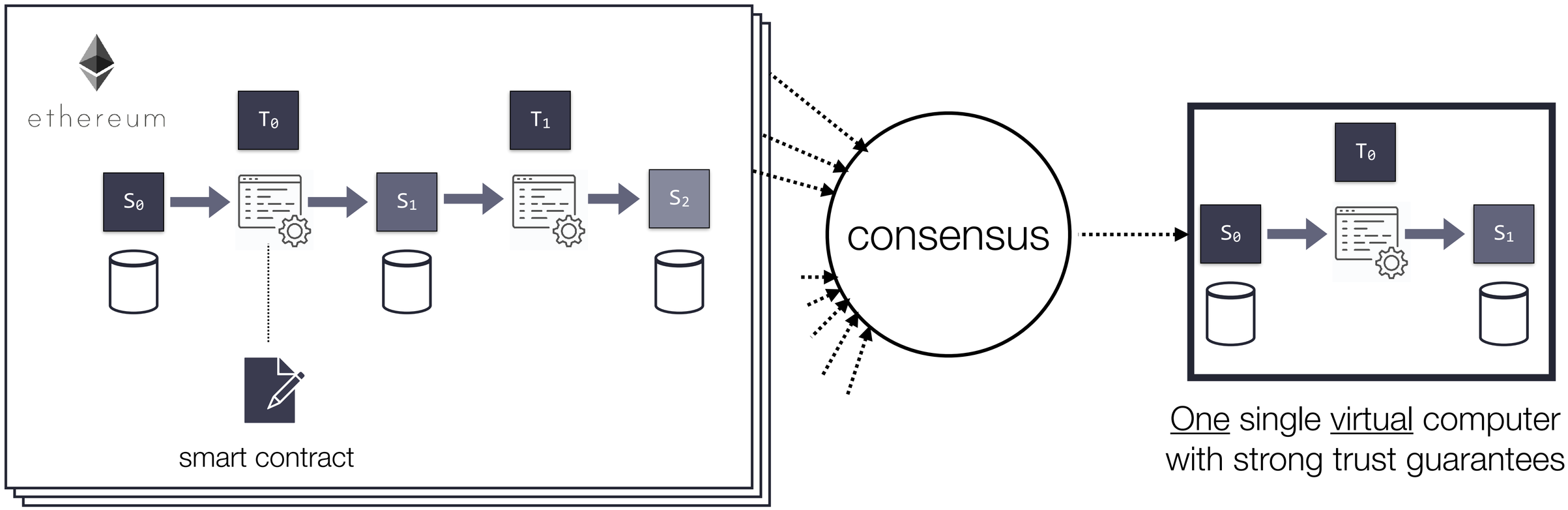
DistriNet

Capability-based financial instruments or: object-capabilities meet smart contracts

Tom Van Cutsem
DistriNet, KU Leuven
IFIP WG2.16 meeting, March 2024

tvcutsem.github.io be.linkedin.com/in/tomvc github.com/tvcutsem twitter.com/tvcutsem @tvcutsem@techhub.social

Blockchains as computers that can make “credible commitments”



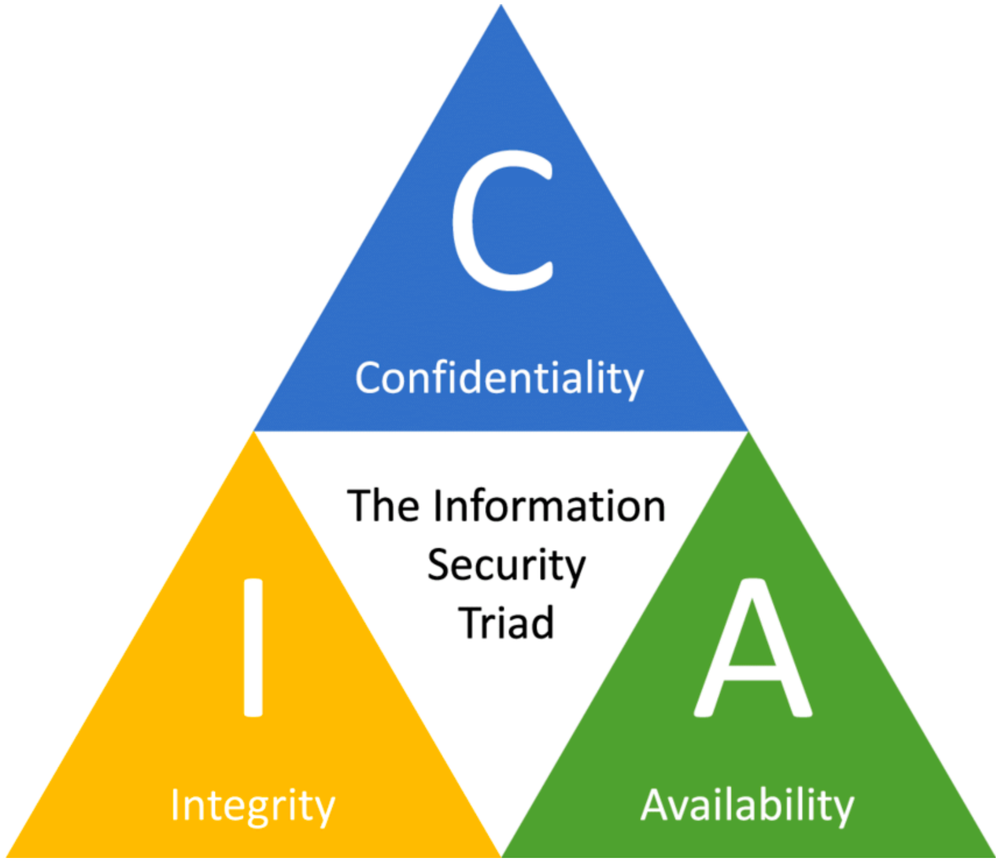
Today's talk

- Access control of "electronic rights" (aka digital assets) in smart contracts
- Object-capabilities: an access control system hidden within the lambda calculus
- How to represent electronic rights using object-capabilities? The Electronic Rights Transfer Protocol (ERTP)
- Documenting common code patterns in ERTP-based contracts
- An ocap-safe subset of JavaScript: "Hardened JavaScript"

The CIA triad from an application security perspective

- **Confidentiality** (a.k.a. Secrecy): No one can infer information they are not supposed to know. Confidentiality usually rests on **cryptography** to keep information secret.
  - Example violation: "*Bob learns how much money Alice has in her bank account*"
  - Example threat: side channel attack.
- **Integrity** (a.k.a. Safety): No "bad" things happen. Integrity usually rests on **access control** determining what agents can cause what effects.
  - Example violation: "*Bob steals Alice's money*"
  - Example threat: confused deputy attack.
- **Availability** (a.k.a. Liveness): "Good" things continue to happen.
  - Example violation: "*Bob prevents Alice from spending her money as she wants*"
  - Example threat: a denial of service attack.

(Source: Miller, "A Taxonomy of Security Issues", 2021, https://agoric.com/blog/technology/a-taxonomy-of-security-issues)



(Image source: Nikander, Jussi & Manninen, Onni & Laajalahti, Mikko. (2020). Requirements for cybersecurity in agricultural communication networks. Computers and Electronics in Agriculture.)

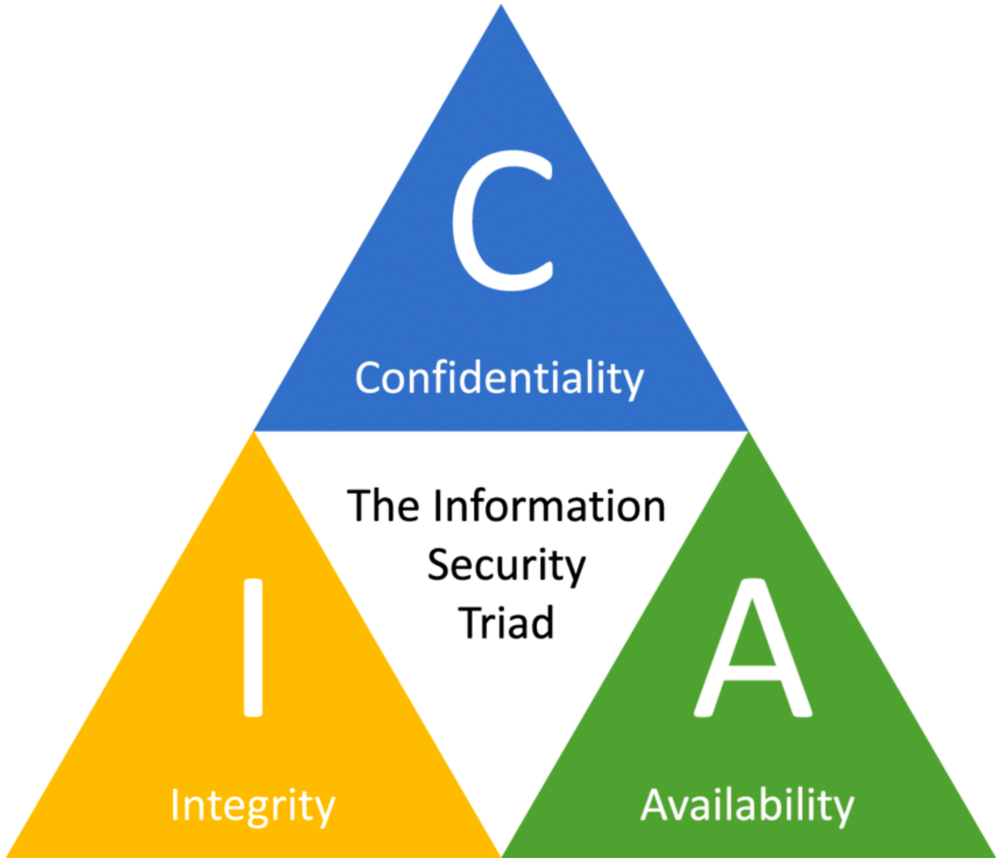
The CIA triad from an application security perspective

- **Confidentiality** (a.k.a. Secrecy): No one can infer information they are not supposed to know. Confidentiality usually rests on **cryptography** to keep information secret.
  - Example violation: "*Bob learns how much money Alice has in her bank account*"
  - Example threat: side channel attack.

Our focus

- **Integrity** (a.k.a. Safety): No "bad" things happen. Integrity usually rests on **access control** determining what agents can cause what effects.
  - Example violation: "*Bob steals Alice's money*"
  - Example threat: confused deputy attack.
- **Availability** (a.k.a. Liveness): "Good" things continue to happen.
  - Example violation: "*Bob prevents Alice from spending her money as she wants*"
  - Example threat: a denial of service attack.

(Source: Miller, "A Taxonomy of Security Issues", 2021, https://agoric.com/blog/technology/a-taxonomy-of-security-issues)

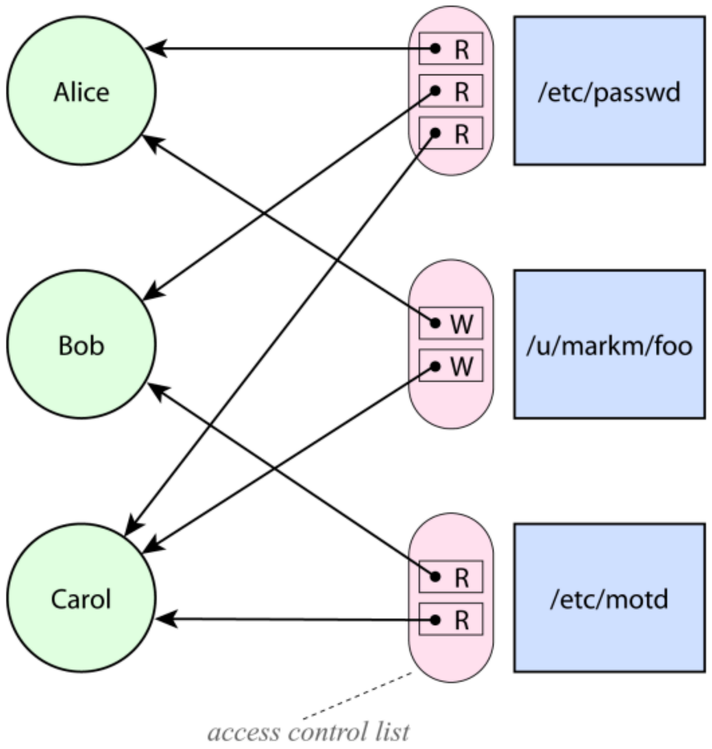


(Image source: Nikander, Jussi & Manninen, Onni & Laajalahti, Mikko. (2020). Requirements for cybersecurity in agricultural communication networks. Computers and Electronics in Agriculture.)

Access control: two alternative views

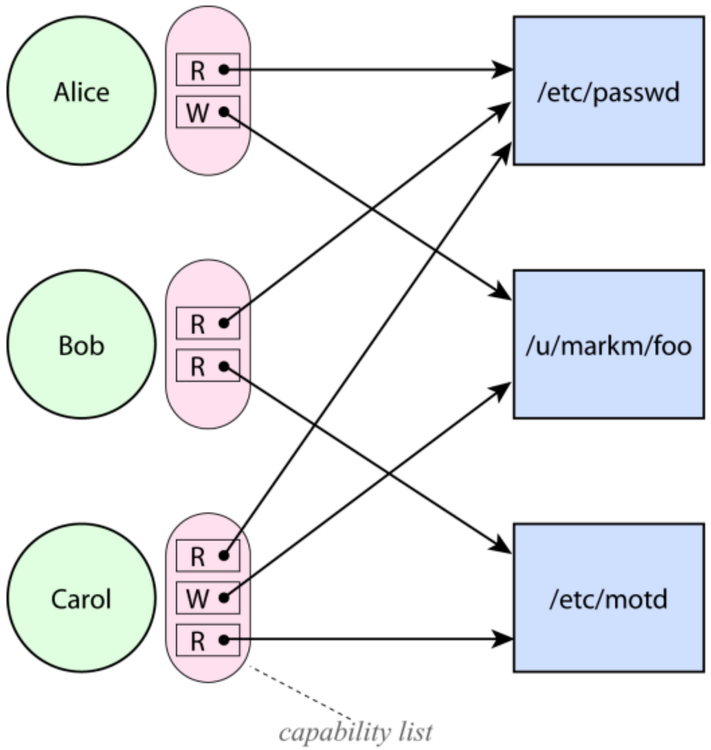
Access control lists (ACLs)

Access control organised around **identity.**



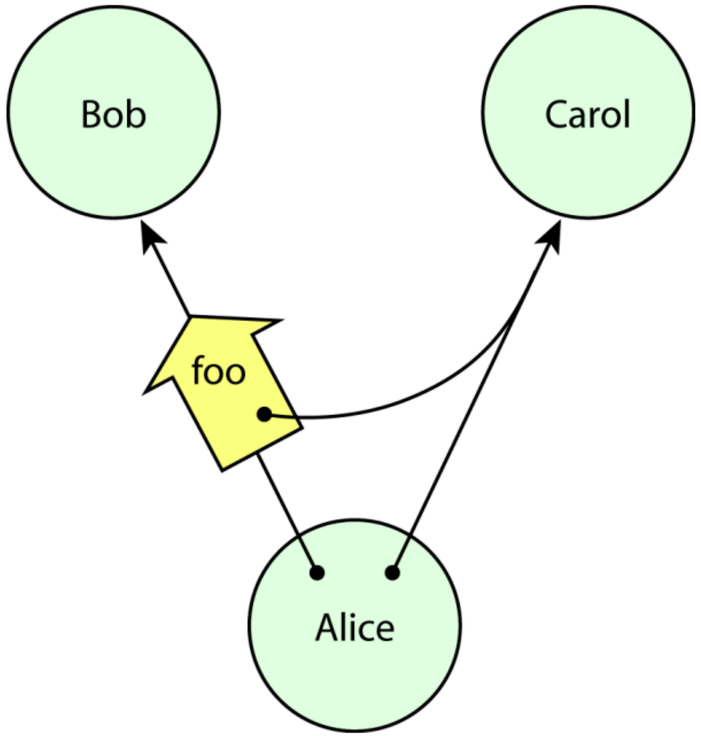
Capabilities (caps)

Access control organised around specific acts of **authorization.**



(source: Miller *et al.* "Capability myths demolished", 2003)

Capability systems excel at delegating authority



Granovetter Diagram

A capability both designates a resource *and* authorises some kind of access to it.

The two are inseparable.

Naming and authority are bundled.

(source: Miller *et al.* “Capability myths demolished”, 2003)

“Only connectivity begets connectivity”

Three simple rules that describe how authority is acquired in an object-capability system:

Creation: e.g. alice creates carol herself

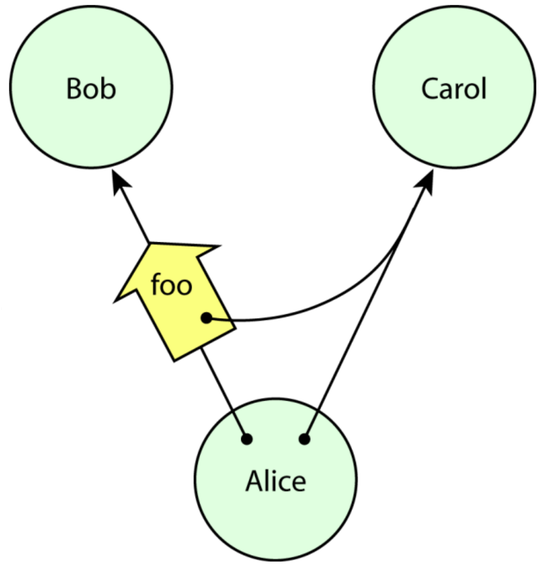
```
// alice executes:
let carol = makeCarol()
```

Endowment: e.g. at creation, alice is endowed with authority to access carol

```
// alice’s constructor:
function makeAlice(carol) {…}
```

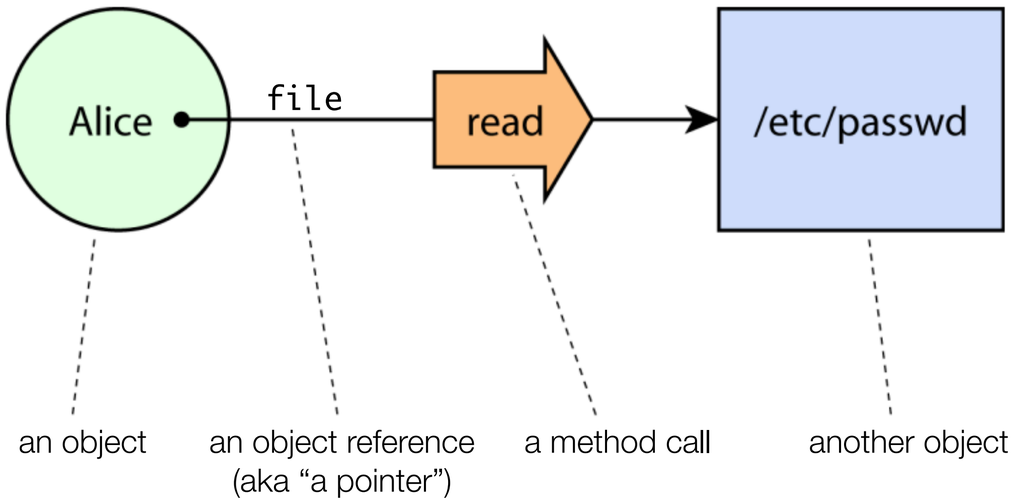
Transfer: e.g. alice transfers carol to bob

```
// alice executes:
bob.foo(carol)
```



What are **object**-capabilities?

- In a memory-safe programming language, an object-capability is simply **an unforgeable reference (a pointer)** to an object (or a function)
  - The designated resource = the object being pointed to
  - Exercising authority = invoking one of the designated object's public methods




```
// alice executes:
file.read()
```


(source: Miller *et al.* "Capability myths demolished", 2003)

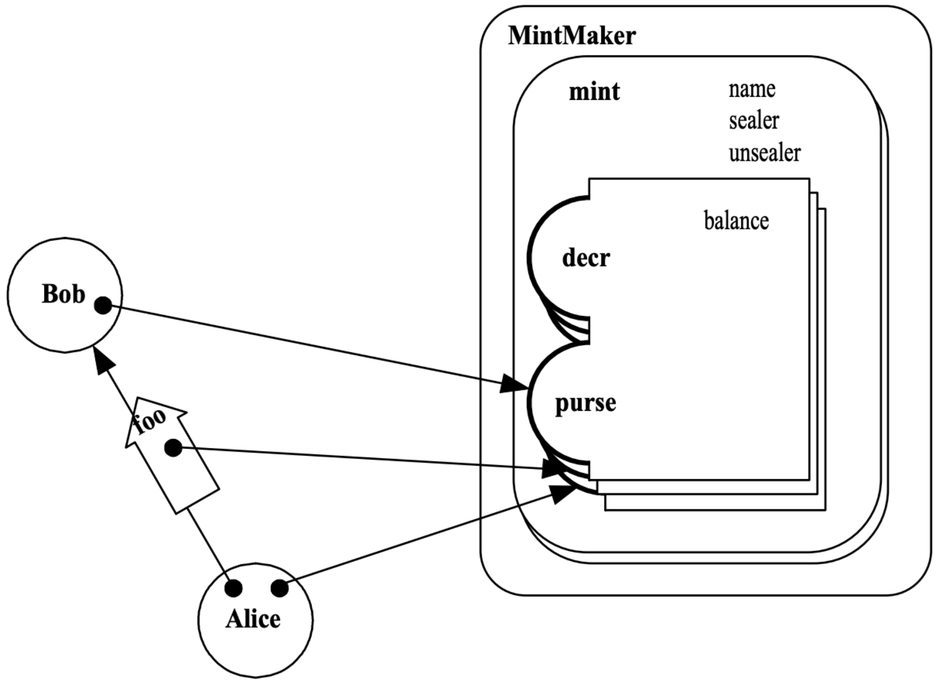
When is a language an **object-capability language**?

1. The language must be **memory-safe**: object pointers are unforgeable
   - Cannot typecast an int to a pointer, cannot randomly access heap memory, …
2. The language must offer strong **encapsulation**
   - Objects need a way to privately store pointers to other objects
3. The language must **not** provide access to **undeniable** (ambient) **authority**
   - Examples of undeniable authority: the ability to import arbitrary modules, the ability to access or update mutable global variables
4. The only way to **delegate authority** is by sharing a pointer to an object
   - “Only connectivity begets connectivity”

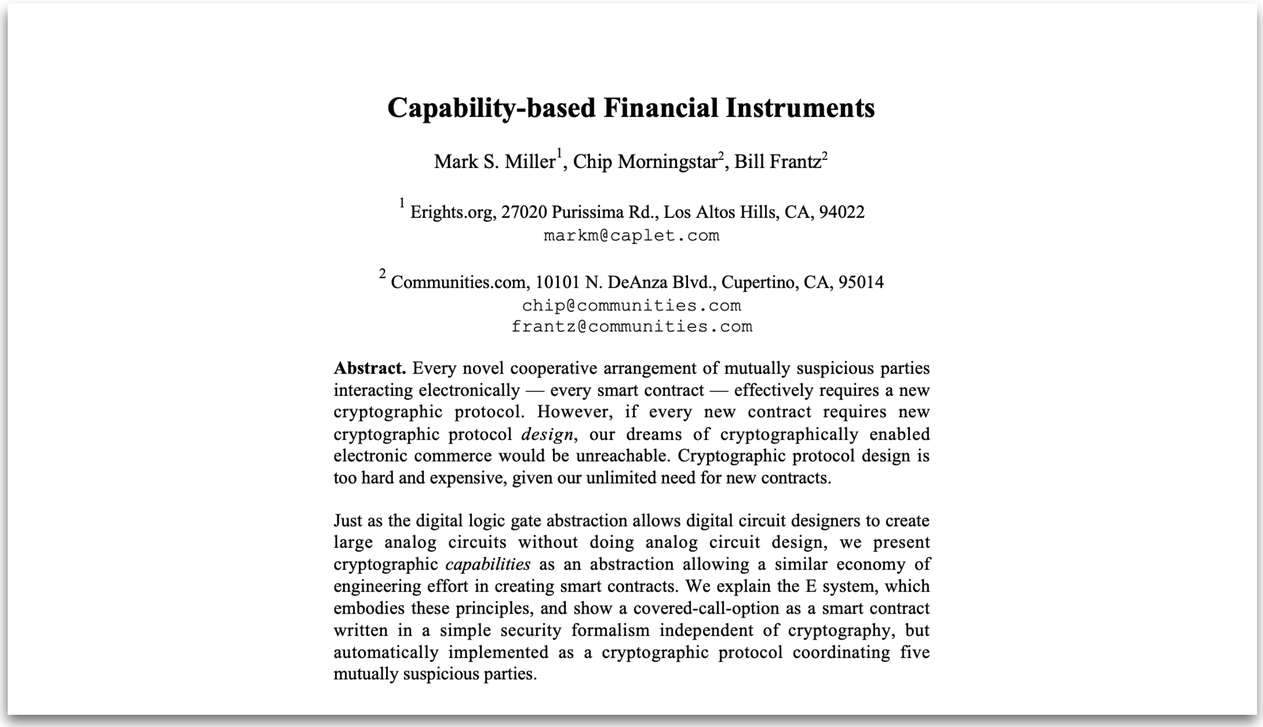
From object-capabilities to “electronic rights”

- An object-capability is a kind of “right”:
  - A bearer instrument: whoever holds it can use it
  - Properties: Non-exclusive, non-fungible, exercisable, non-assayable
- Starting from only object-capabilities, can we build other kinds of “rights”?
  - Example: e-money.
  - Properties: Exclusive, fungible, non-exercisable, assayable

2000s: Capability-based “financial instruments”



Simple e-money protocol
in 25 lines of *E* code



Capability-based Financial Instruments

Mark S. Miller[1], Chip Morningstar[2], Bill Frantz[2]

[1] Erights.org, 27020 Purissima Rd., Los Altos Hills, CA, 94022
markm@caplet.com

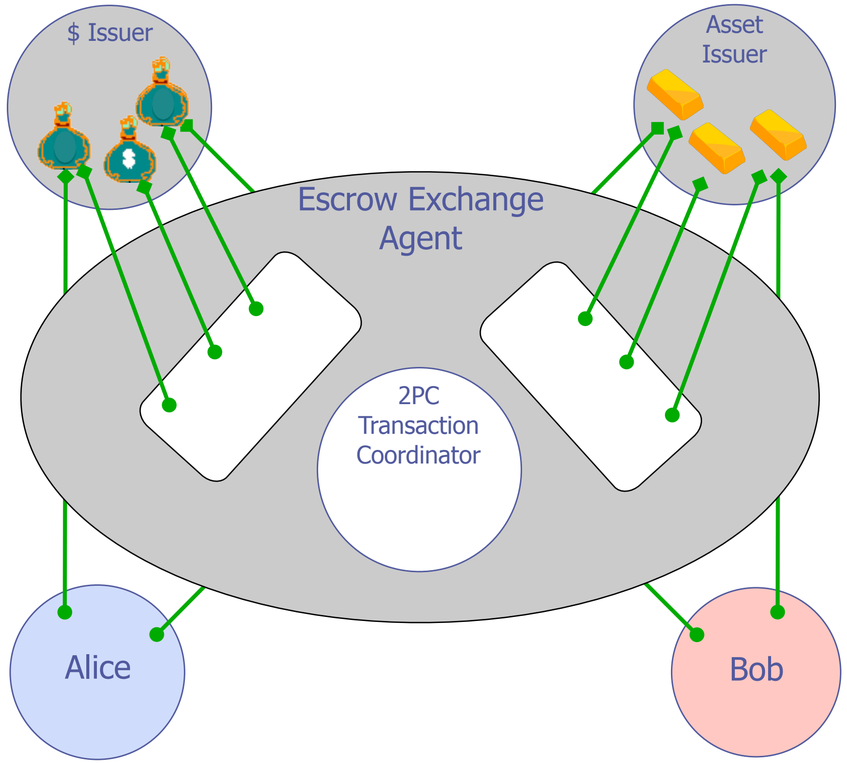
[2] Communities.com, 10101 N. DeAnza Blvd., Cupertino, CA, 95014
chip@communities.com
frantz@communities.com

Abstract. Every novel cooperative arrangement of mutually suspicious parties interacting electronically — every smart contract — effectively requires a new cryptographic protocol. However, if every new contract requires new cryptographic protocol *design*, our dreams of cryptographically enabled electronic commerce would be unreachable. Cryptographic protocol design is too hard and expensive, given our unlimited need for new contracts.

Just as the digital logic gate abstraction allows digital circuit designers to create large analog circuits without doing analog circuit design, we present cryptographic *capabilities* as an abstraction allowing a similar economy of engineering effort in creating smart contracts. We explain the E system, which embodies these principles, and show a covered-call-option as a smart contract written in a simple security formalism independent of cryptography, but automatically implemented as a cryptographic protocol coordinating five mutually suspicious parties.

(Miller *et* al., Financial Cryptography 2000)

2010s: Distributed Resilient Secure ECMAScript (Dr. SES)



Secure escrow exchange protocol in 42 lines of JavaScript

Distributed Electronic Rights in JavaScript

Mark S. Miller[1], Tom Van Cutsem[2], and Bill Tulloh

[1] Google, Inc.
[2] Vrije Universiteit Brussel

Abstract. Contracts enable mutually suspicious parties to cooperate safely through the exchange of rights. Smart contracts are programs whose behavior enforces the terms of the contract. This paper shows how such contracts can be specified elegantly and executed safely, given an appropriate distributed, secure, persistent, and ubiquitous computational fabric. JavaScript provides the ubiquity but must be significantly extended to deal with the other aspects. The first part of this paper is a progress report on our efforts to turn JavaScript into this fabric. To demonstrate the suitability of this design, we describe an escrow exchange contract implemented in 42 lines of JavaScript code.

Keywords: security, distributed objects, object-capabilities, smart contracts

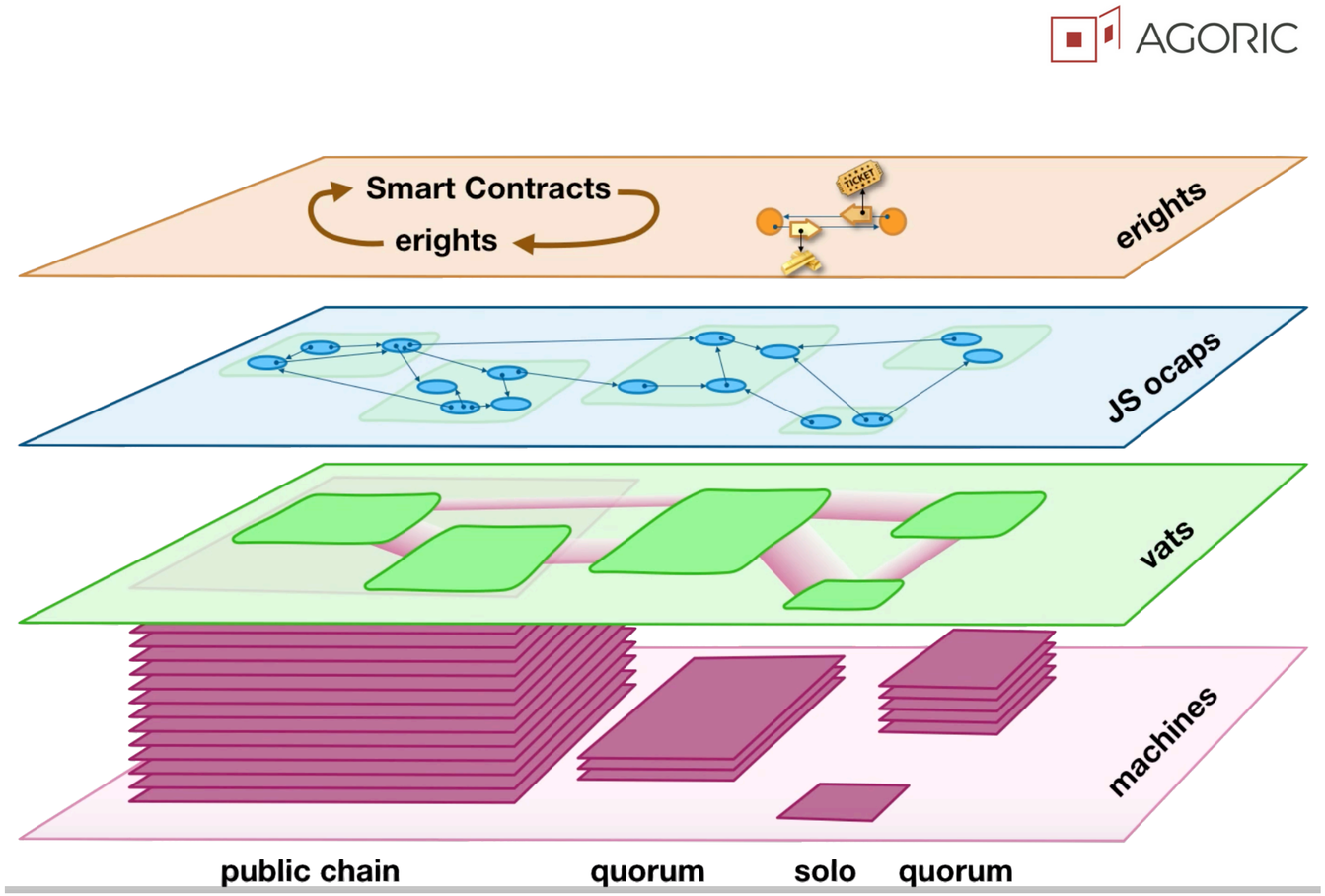
(Miller *et al.*, European Symposium on Programming, 2013)

2020s: The Agoric stack: writing smart contracts in JavaScript

Digital assets or “erights”
managed by
Zoe framework

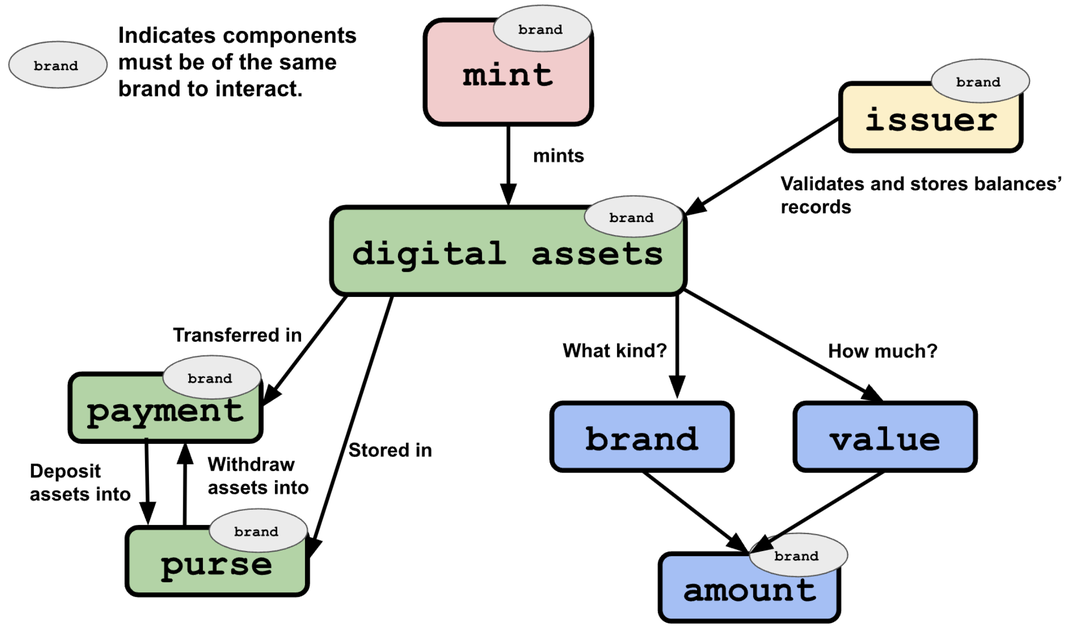
written in
Hardened JavaScript

executing on
a blockchain
(Tendermint / Cosmos)

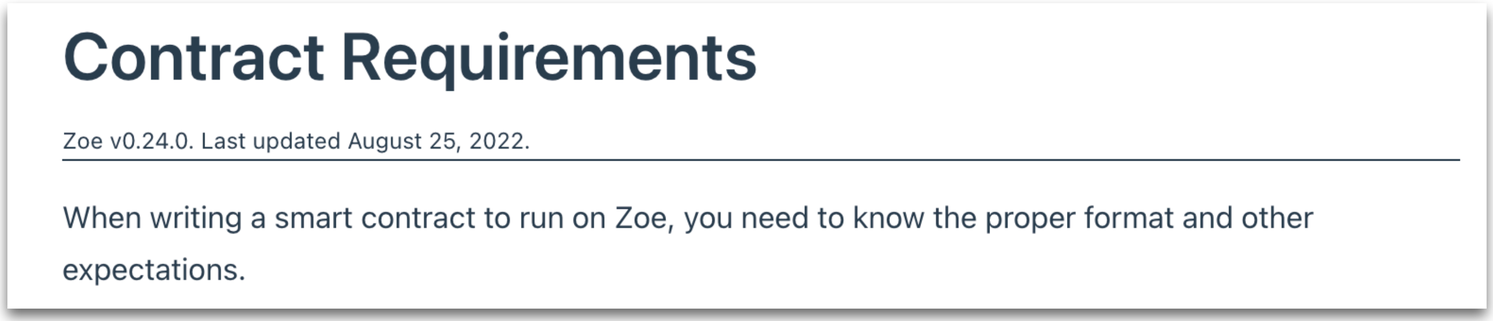


ERTP: the Electronic Rights Transfer Protocol

- In ERTP, digital assets or “rights” are represented as object-capabilities
- Access to a **`mint`** object → authority to mint new assets of a given **`brand`**
- Access to a **`payment`** object *and* a **`purse`** object of the *same* **`brand`** → authority to spend (transfer) the asset
- Access to an **`invitation`** object → authority to participate in a contract



Zoe and ERTP: patterns



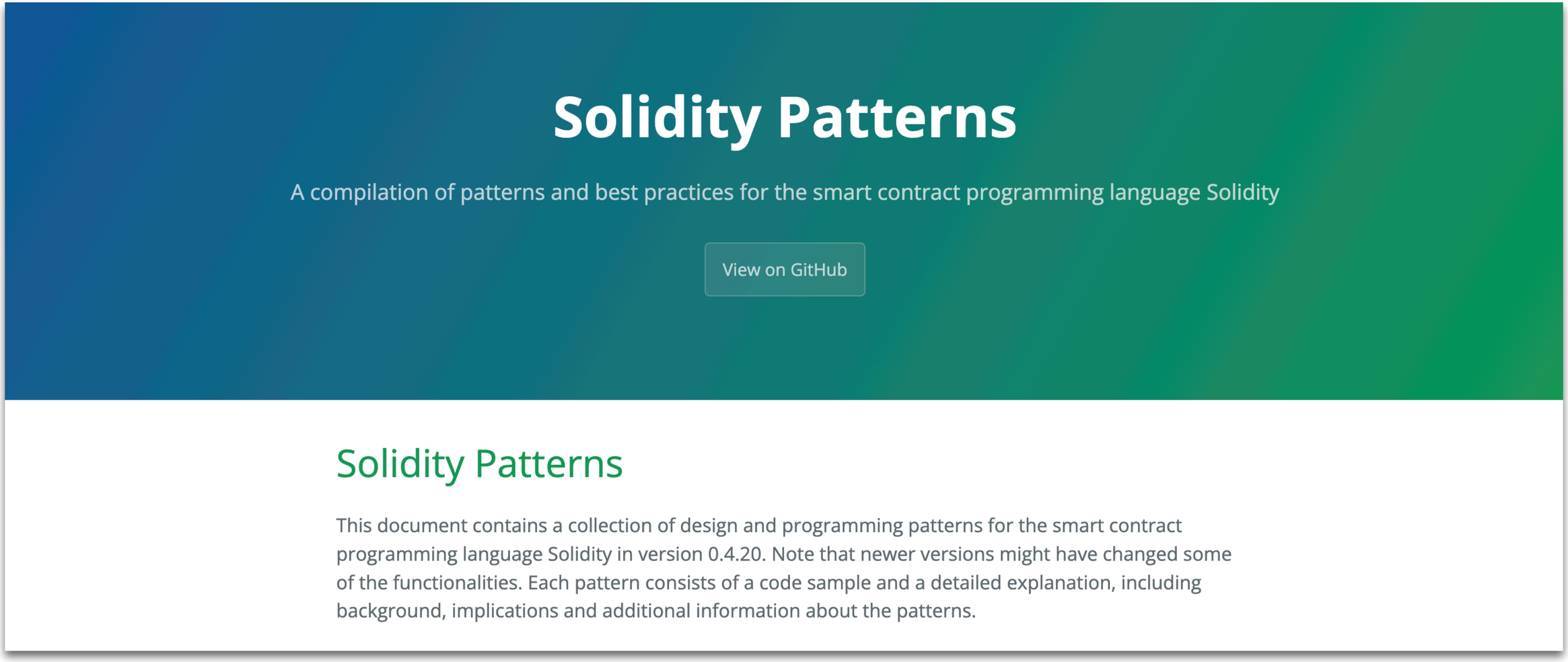
Contract Requirements

Zoe v0.24.0. Last updated August 25, 2022.

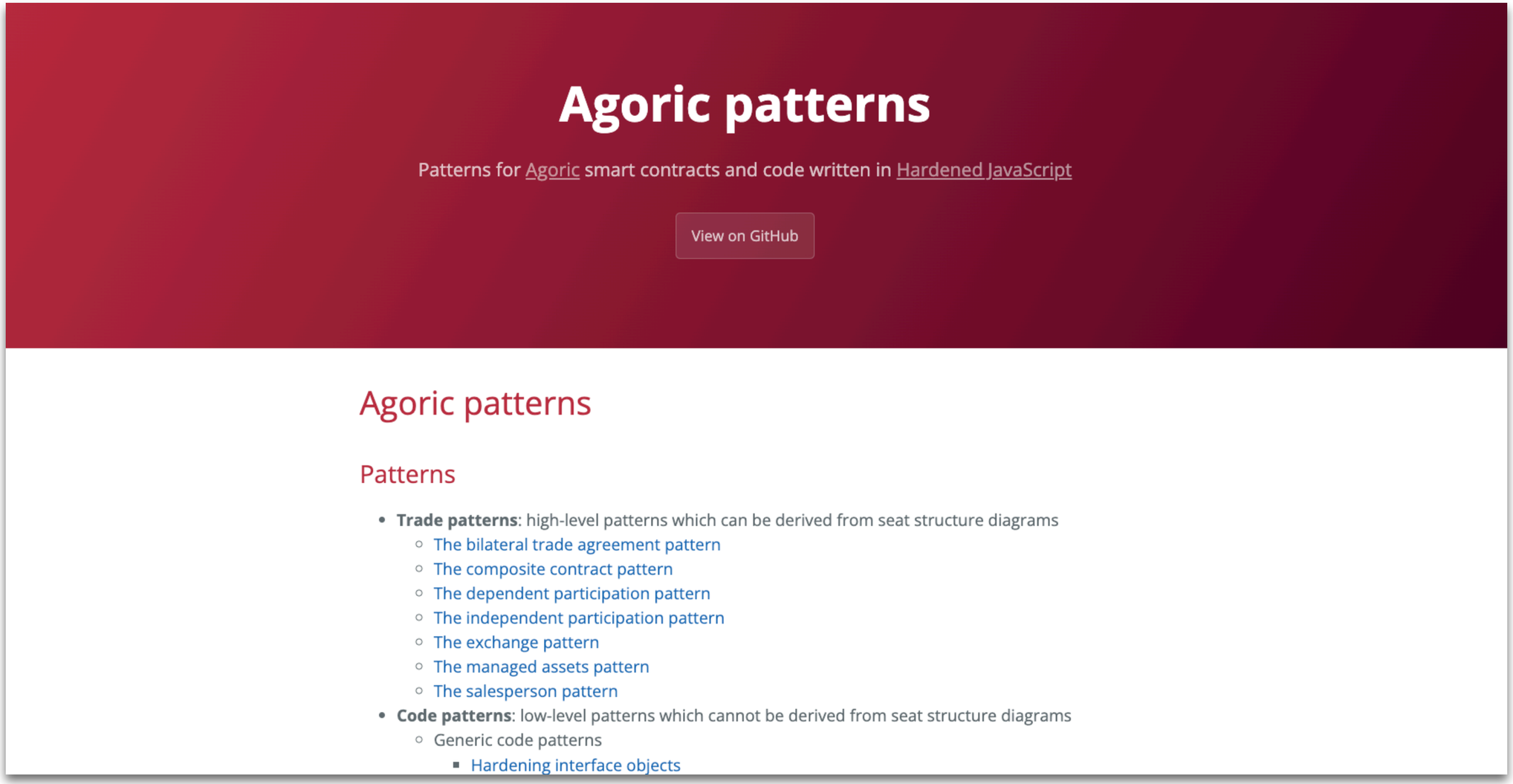
When writing a smart contract to run on Zoe, you need to know the proper format and other expectations.

(source: Agoric)

- Security of assets now hinges on the reachability of object-capabilities
- Must carefully reason about (transitive) reachability
- Patterns help facilitate this reasoning (elevate the level of abstraction)



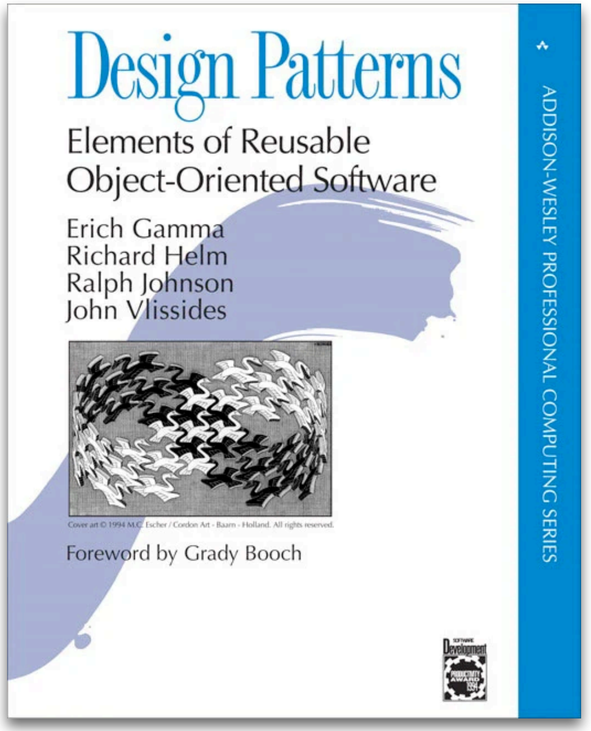
https://fravoll.github.io/solidity-patterns/



https://ilyasmercan.github.io/AgoricPatterns/

Mining patterns

- **Trade** patterns: patterns of arranging *invitations* and *seats* within a Zoe contract
- **Code** patterns: patterns of arranging interactions among *objects* in Hardened JavaScript
- Why? Establish a pattern language. Elevate levels of abstraction. Document best-practices.
- Provide insight into how to manage authority over digital assets using object-capabilities
- Pattern repository made available on GitHub (PRs welcome)



Mined corpus: 10 idiomatic Agoric smart contracts

Name	Description
Atomic Swap	A basic trade of digital assets between two parties.
Automatic Refund	A trivial contract that gives the user back what they put in.
Barter Exchange	An exchange with an order book letting all kinds of goods to be offered for explicit barter swaps.
Covered Call	Creates a call option, which is the right to buy an underlying asset.
Mint and Sell NFTs	A contract that mints NFTs and sells them through a separate sales contract.
Mint Payments	An example of minting fungible tokens.
Oracle	A low-level oracle contract for querying Chainlink (opens new window)or other oracles.
OTC Desk	A contract for giving quotes that can be exercised. The quotes are guaranteed to be exercisable because they are actually options with escrowed underlying assets.
Sell Items	A generic sales contract, mostly used for selling NFTs for money.
Simple Exchange	A basic exchange with an order book for one asset, priced in a second asset.

Contracts x Code patterns

	Hardening interface objects	Enforcing eventual interac-tions with untrusted parties	Verifying untrusted invitations	Validate all contract parame-ters	The delayed initializa-tion pattern	The revocable contract pattern	Design least authority interfaces	The secure shutdown patterns	Partition authority using facets
Atomic Swap	x	x	x	x			x		x
Automatic Refund	x	x	x				x		x
Barter Exchange	x	x	x				x		x
Covered Call	x	x	x	x			x		x
Mint and Sell NFTs	x	x	x				x		x
Mint Payments	x	x	x				x		x
Oracle	x	x	x		x	x	x	x	x
OTC Desk	x	x	x	x			x		x
Sell Items	x	x	x	x			x		x
Simple Exchange	x	x	x	x			x		x

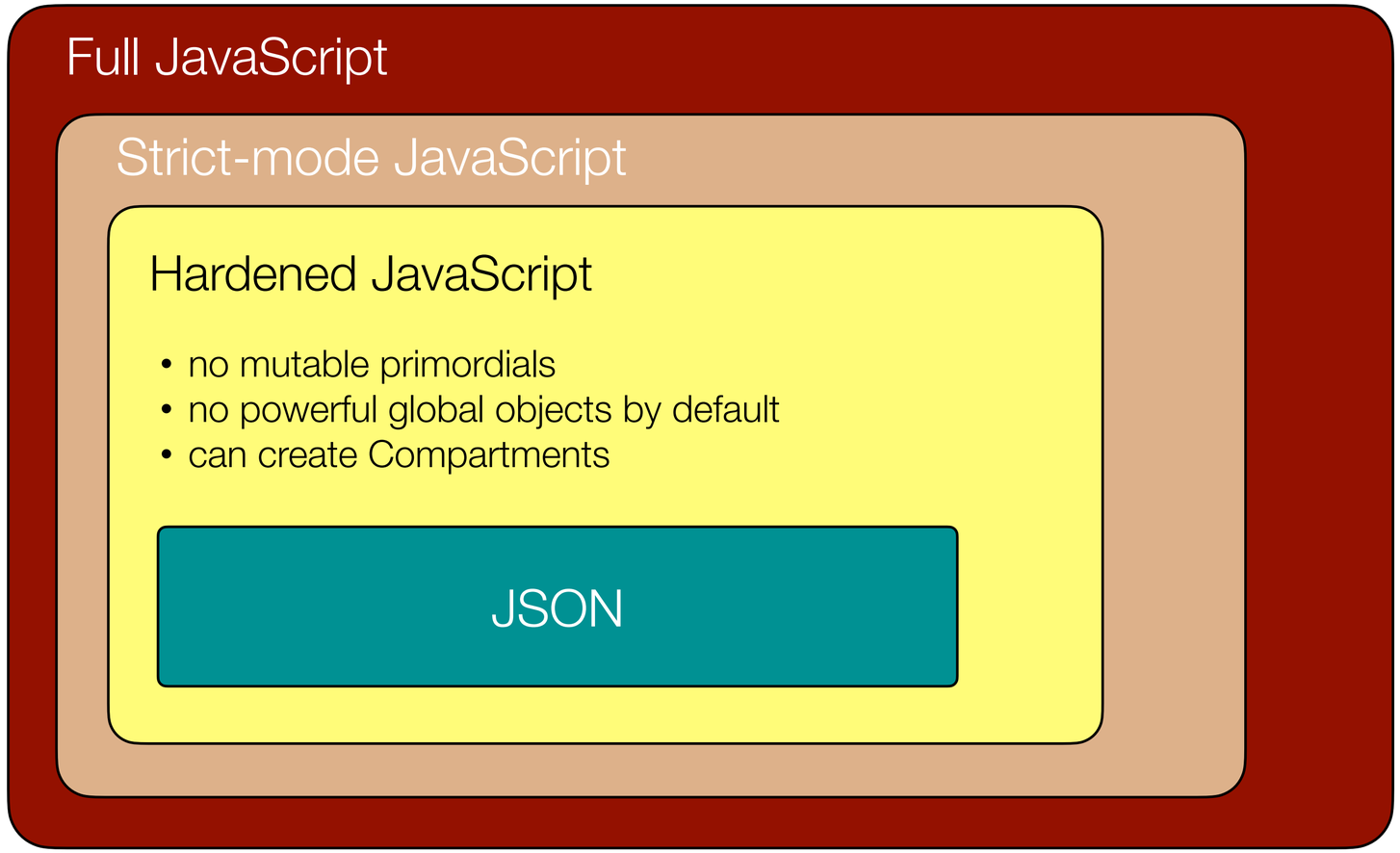
Table B.2: Summarizing table of studied smart contracts and the code patterns found in these smart contracts.

Identified code patterns

- Make modules powerless by default → requires **Compartments**
- Make all *interface* objects transitively *immutable* → requires **hardening**
- Make all *interactions* with untrusted parties *asynchronous* (similar to the *checks-effects-**interaction*** pattern in Solidity) → requires **eventual send**
- *Verify* the authenticity of received objects: establish the authenticity via a separate path to a trusted issuer (“*brands check*”) → requires **sealer/unsealer pairs**
- Design *least authority* interfaces → partition authority using **facets**

Turning JavaScript into an ocap-safe language: “Hardened JavaScript”

Hardened JavaScript is a secure subset of standard JavaScript

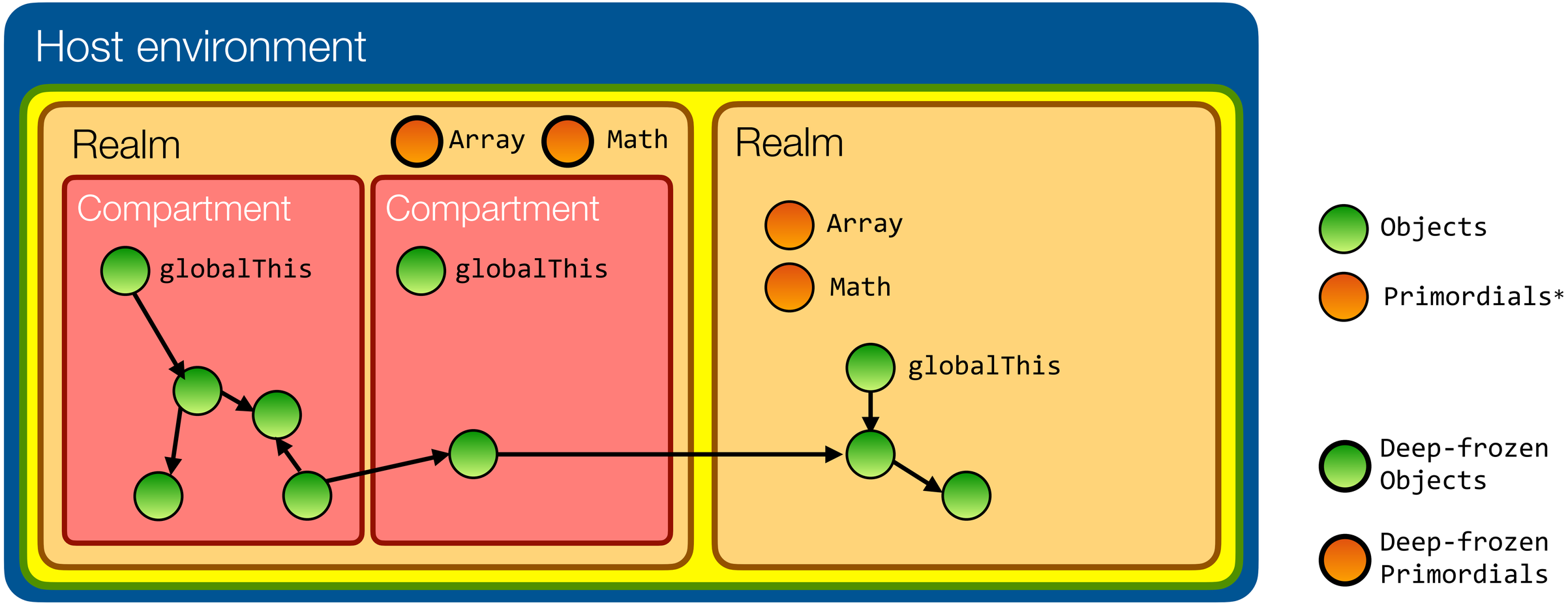


(inspired by the diagram at https://github.com/Agoric/Jessie)

Key idea: code running in hardened JS can only affect the outside world through objects (capabilities) explicitly granted to it from outside.

Isolating modules using Compartments

Each Compartment has its own global object but shared (immutable) primordials.



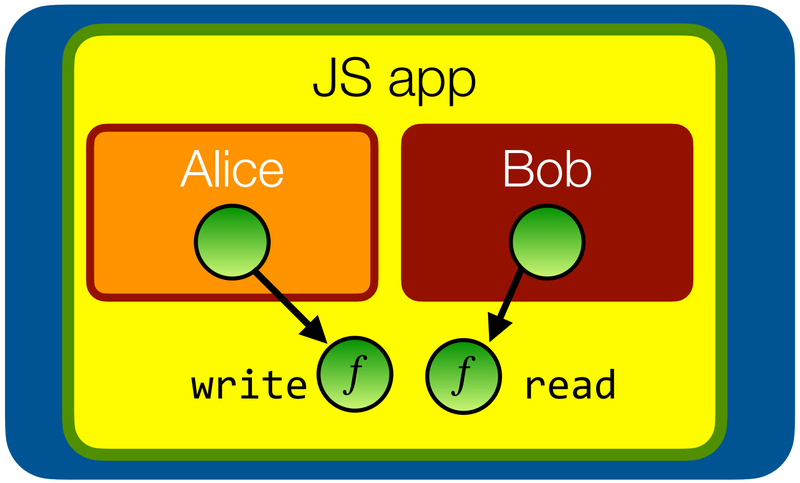
* Primordials: built-in objects like `Object`, `Object.prototype`, `Array`, `Function`, `Math`, `JSON`, etc.

Example: apply POLA to a basic shared log

We would like Alice to only **write** to the log, and Bob to only **read** from the log.

Assume each module is loaded in a separate Compartment (see: LavaMoat)

```
import * as alice from "alice.js";
import * as bob from “bob.js";

function makeLog() {
  const messages = [];
  function write(msg) { messages.push(msg); }
  function read() { return [...messages]; }
  return Object.freeze({read, write});
}


let log = makeLog();
alice(log.write);
bob(log.read);
```

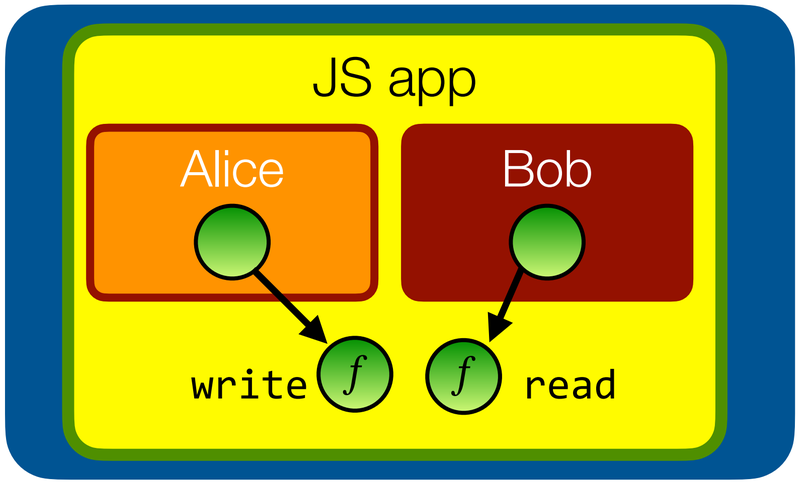


Example: apply POLA to a basic shared log

We would like Alice to only **write** to the log, and Bob to only **read** from the log.

Assume each module is loaded in a separate Compartment (see: LavaMoat)

```
import * as alice from "alice.js";
import * as bob from “bob.js";

function makeLog() {
  const messages = [];
  function write(msg) { messages.push(msg); }
  function read() { return [...messages]; }
  return Object.freeze({read, write});
}


let log = makeLog();
alice(log.write);
bob(log.read);
```

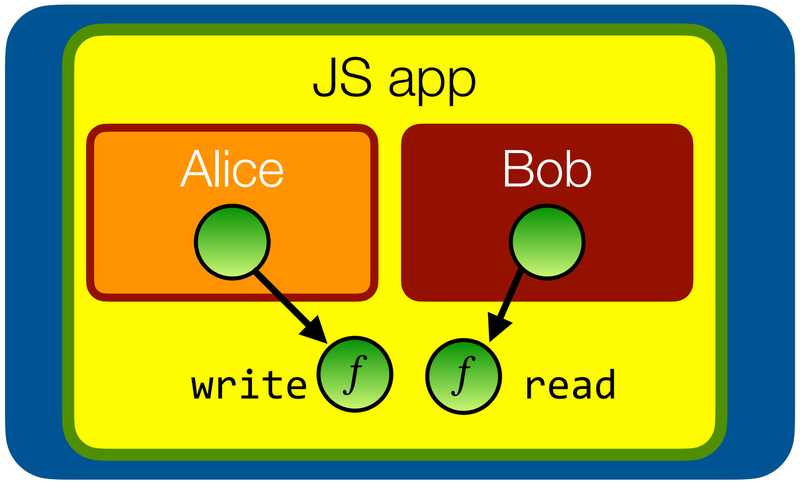



```
// Bob can still modify the function object itself
read.apply = function() { “gotcha” };
```

Example: apply POLA to a basic shared log

Hardened JavaScript provides a `harden` function that “deep-freezes” an object (recurse through properties *and* the prototype chain)

```
import * as alice from "alice.js";
import * as bob from “bob.js";

function makeLog() {
  const messages = [];
  function write(msg) { messages.push(msg); }
  function read() { return [...messages]; }
  return harden({read: read, write: write});
}


let log = makeLog();
alice(log.write);
bob(log.read);
```




```
// Bob can still modify the function object itself
read.apply = function() { “gotcha” };
```

Hardening is critical to achieve “defensive consistency”

- **“Defensive consistency”** (a.k.a. **“Robust Safety”** [Swasey *et al.*, 2017]): a service should retain its functional correctness (i.e. internal invariants and post-conditions) towards well-behaved clients in the presence of adversarial clients (who may exhibit arbitrary behaviour)

- The harden function visits every value in the transitive closure over all own properties *and* over the prototype chain and ensures the value is either frozen or “powerless” (e.g. a primitive value)

- Clients are no longer able to tamper with the *API surface* of the object *graph* rooted in the hardened object. Clients can only *read* properties and *call* functions.

- A hardened service **does not imply immutability or purity!** Hardened functions may close over mutable state (this includes the get/set functions of accessor properties).

- Hardening a service is a **necessary but not always a sufficient** step to create defensively consistent services in JavaScript

Is JavaScript an **object-capability language**?

1. The language must be **memory-safe**: object pointers are unforgeable

   ✓ Yes, all correct JavaScript VM implementations provide this guarantee

2. The language must offer strong **encapsulation**

   ✓ Yes, by hiding variables through lexical scope (also, private fields)

3. The language must **not** provide access to **undeniable** (ambient) **authority**

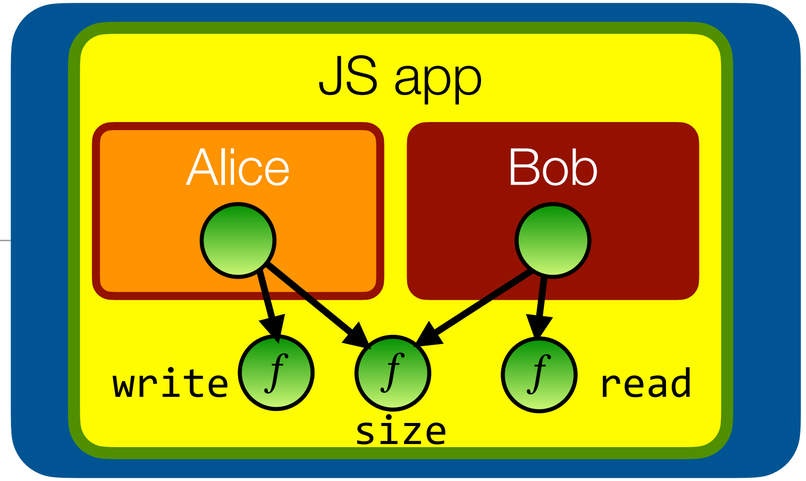
   ✓ Yes, when using Hardened JavaScript to load each module in its own Compartment

4. The only way to **delegate authority** is by sharing a pointer to an object

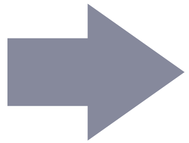
   ✓ Yes, if modules export *only hardened objects* (no mutable global variables)

What if Alice and Bob need more authority?

If over time we want to expose more functionality to Alice and Bob, we need to refactor all of our code.



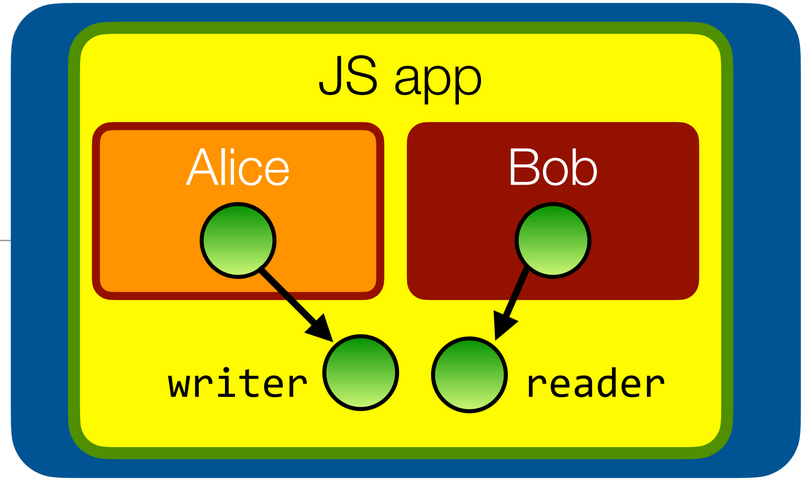

```
import * as alice from "alice.js";
import * as bob from “bob.js";

function makeLog() {
  const messages = [];
  function write(msg) { messages.push(msg); }
  function read() { return [...messages]; }
  return harden({read, write});
}


let log = makeLog();
alice(log.write);
bob(log.read);
```



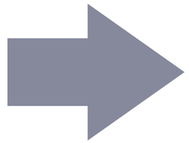
```
import * as alice from "alice.js";
import * as bob from “bob.js";

function makeLog() {
  const messages = [];
  function write(msg) { messages.push(msg); }
  function read() { return [...messages]; }
  function size() { return messages.length(); }
  return harden({read, write, size});
}

let log = makeLog();
alice(log.write, log.size);
bob(log.read, log.size);
```

Expose distinct authorities through **facets**

Separate powerful authority across several distinct interfaces through nested objects called *facets*




```
import * as alice from "alice.js";
import * as bob from “bob.js";

function makeLog() {
  const messages = [];
  function write(msg) { messages.push(msg); }
  function read() { return [...messages]; }
  function size() { return messages.length(); }
  return harden({read, write, size});
}

let log = makeLog();
alice(log.write, log.size);
bob(log.read, log.size);
```

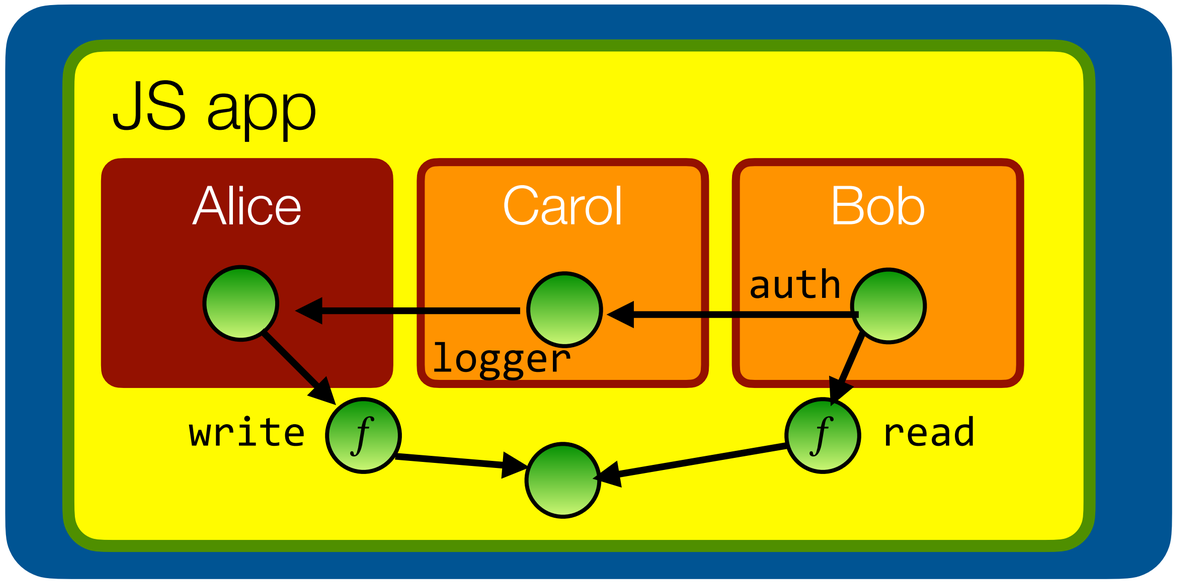


```
import * as alice from "alice.js";
import * as bob from “bob.js";

function makeLog() {
  const messages = [];
  function write(msg) { messages.push(msg); }
  function read() { return [...messages]; }
  function size() { return messages.length(); }
  return harden({
    reader: {read, size},
    writer: {write, size}
  });
}

let log = makeLog();
alice(log.writer);
bob(log.reader);
```

Another exercise in POLA

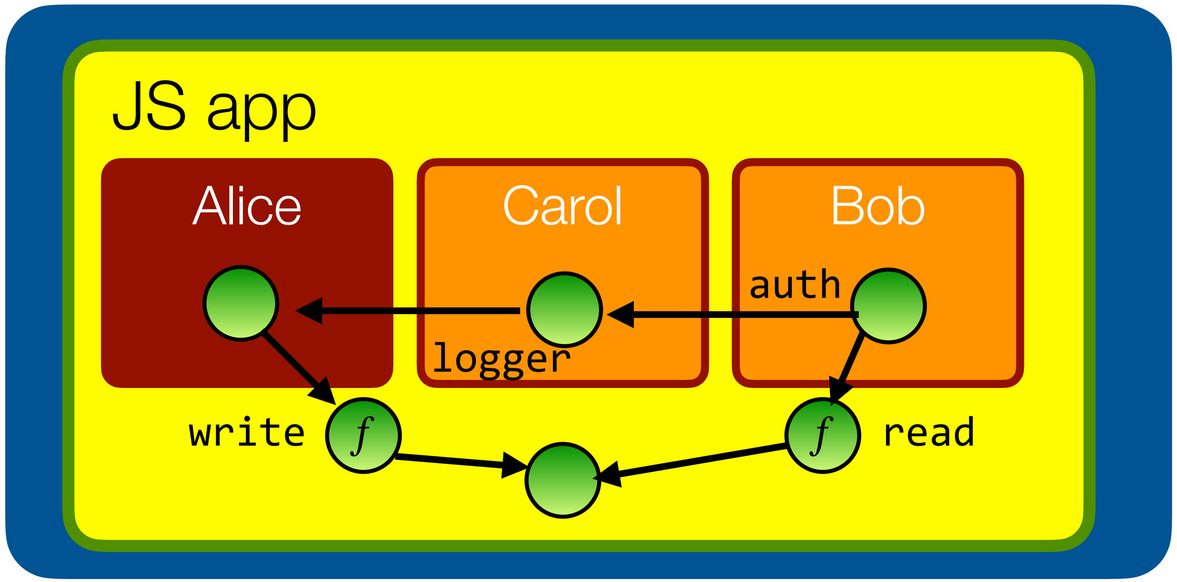
- Bob wants to know if the messages Alice has logged were previously authorized by Carol
- Bob trusts Carol (to authorize messages), but he does not trust Alice

```
import * as alice from "alice.js";
import * as bob from “bob.js";
import * as carol from “carol.js";

function makeLog() {
  const messages = [];
  function write(msg) { messages.push(msg); }
  function read() { return [...messages]; }
  return harden({read, write});
}

let log = makeLog();

alice(log.write);
bob(log.read, carol.auth);
carol(alice);
```



Another exercise in POLA

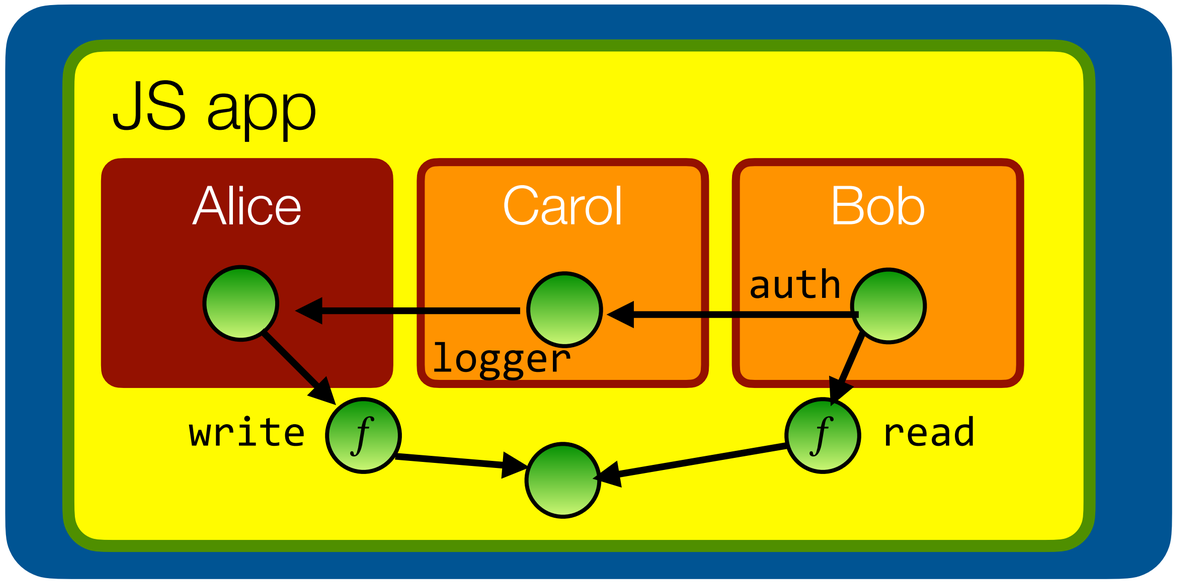
```
// in carol.js
function makeCarol(logger) {
  let [seal, unseal] = makeSealerUnsealerPair();
  let stamped = new WeakMap();
  function authorizeAndLog(msg) {
    let stamp = seal(msg);
    stamped.set(stamp, msg);
    logger.write({…msg, stamp});
  }
  function auth(msg) {
    return stamped.get(unseal(msg.stamp)) == msg;
  }
  return harden({authorizeAndLog, auth});
}

// in bob.js
function makeBob(read, auth) {
  …
  let msgs = read(); // untrusted
  let authorized = msgs.filter(msg => auth(msg));
  …
}
```



This is called “rights amplification”. It’s a useful POLA building block.

- Only code that has access to both the unseal function and the mapping can authorize

```
// in carol.js
function makeCarol(logger) {
  let [seal, unseal] = makeSealerUnsealerPair();
  let stamped = new WeakMap();
  function authorizeAndLog(msg) {
    let stamp = seal(msg);
    stamped.set(stamp, msg);
    logger.write({…msg, stamp});
  }
  function auth(msg) {
    return stamped.get(unseal(msg.stamp)) == msg;
  }
  return harden({authorizeAndLog, auth});
}

// in bob.js
function makeBob(read, auth) {
  …
  let msgs = read(); // untrusted
  let authorized = msgs.filter(msg => auth(msg));
  …
}
```



Code patterns in HardenedJS smart contracts: recap

- Make modules powerless by default → requires **Compartments**
- Make all *interface* objects transitively *immutable* → requires **hardening**
- Make all *interactions* with untrusted parties *asynchronous* (similar to the *checks-effects-**interaction*** pattern in Solidity) → requires **eventual sending**
- *Verify* the authenticity of received objects: establish the authenticity via a separate path to a trusted issuer (“*brands check*”) → requires **sealer/unsealer pairs**
- Design *least authority* interfaces → partition authority using **facets**

KU LEUVEN

DistriNet

Capability-based financial instruments or: object-capabilities meet smart contracts

Tom Van Cutsem
DistriNet, KU Leuven
March 2024

Questions?

tvcutsem.github.io be.linkedin.com/in/tomvc github.com/tvcutsem twitter.com/tvcutsem @tvcutsem@techhub.social