Exploring the design space of smart contract languages

Tom Van Cutsem
March 2023
IFIP WG 2.16 Delft Meeting

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

DistriNet in a Nutshell (incl. *capabilities in applied research*)

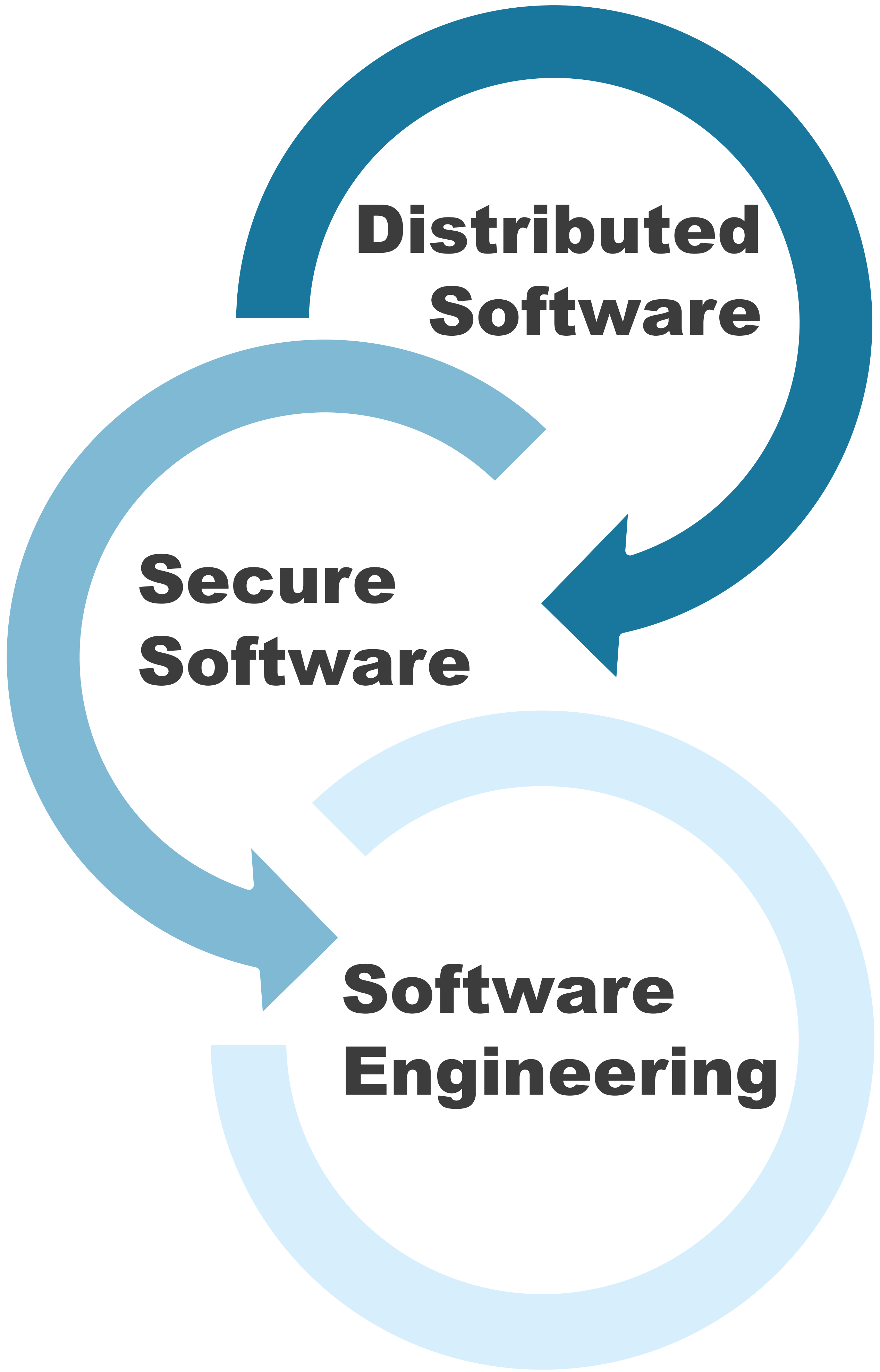
100+ FTEs

10+ professors
10+ research managers
4 business office
10+ postdocs
55+ PhD students

1984

30+ years track record

6 spin off companies



20+ ongoing projects

Fundamental research → Strategic Basic research → Collaborative research → Ready-to-market

100 + industry collaborations

DistriNet

Research agenda

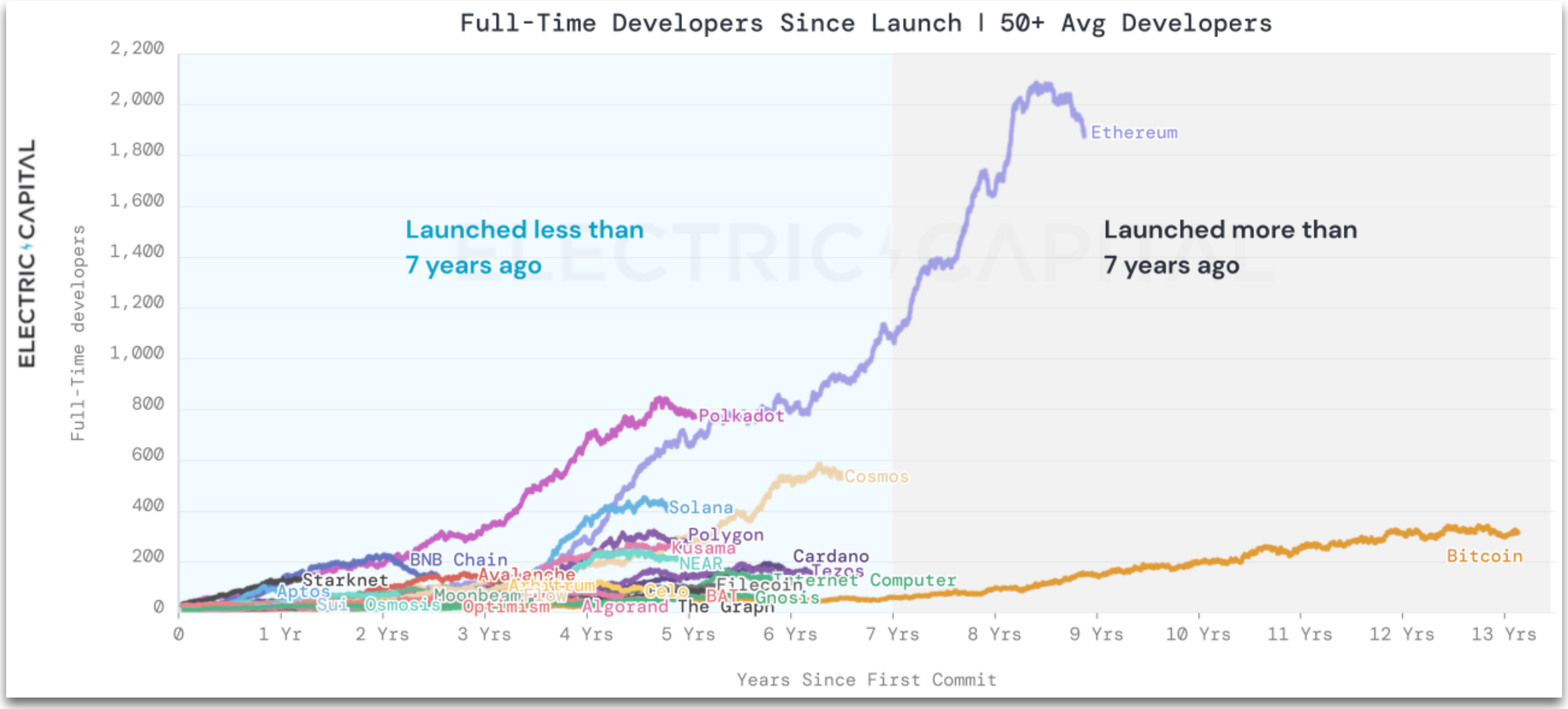
- Web3 and decentralized computing technologies
- Security, privacy & scalability of **blockchain** systems
- Programmable blockchains (**smart contracts**)
- Finding better ways to bridge “Web2” and “Web3”

CYBERSECURITY FLANDERS
BUILDING YOUR DIGITAL FUTURE

Vlaamse overheid

https://cybersecurity-research.be

Web3: a growing developer ecosystem



(Source: Electric Capital, blockchain developer report, January 2023)

Application-specific

General-purpose

VS

ethereum

What is a smart contract?

A software program that automatically moves digital assets according to arbitrary pre-specified rules

(Vitalik Buterin, Ethereum White Paper, 2014)

What is a smart contract?

A software program that can receive, store & send “money”

Essentially, a program with its own “bank account”

Smart contracts: basic principle

- A vending machine is an **automaton** that can trade **physical** assets

1. insert coins

2. dispense drink

Smart contracts: basic principle

- A smart contract is an **automaton** that can trade **digital** assets

code

1. insert digital coins (tokens)

2. dispense other digital assets or electronic rights

But who should we trust to faithfully execute the automaton’s code?

- A smart contract is an **automaton** that can trade **digital** assets

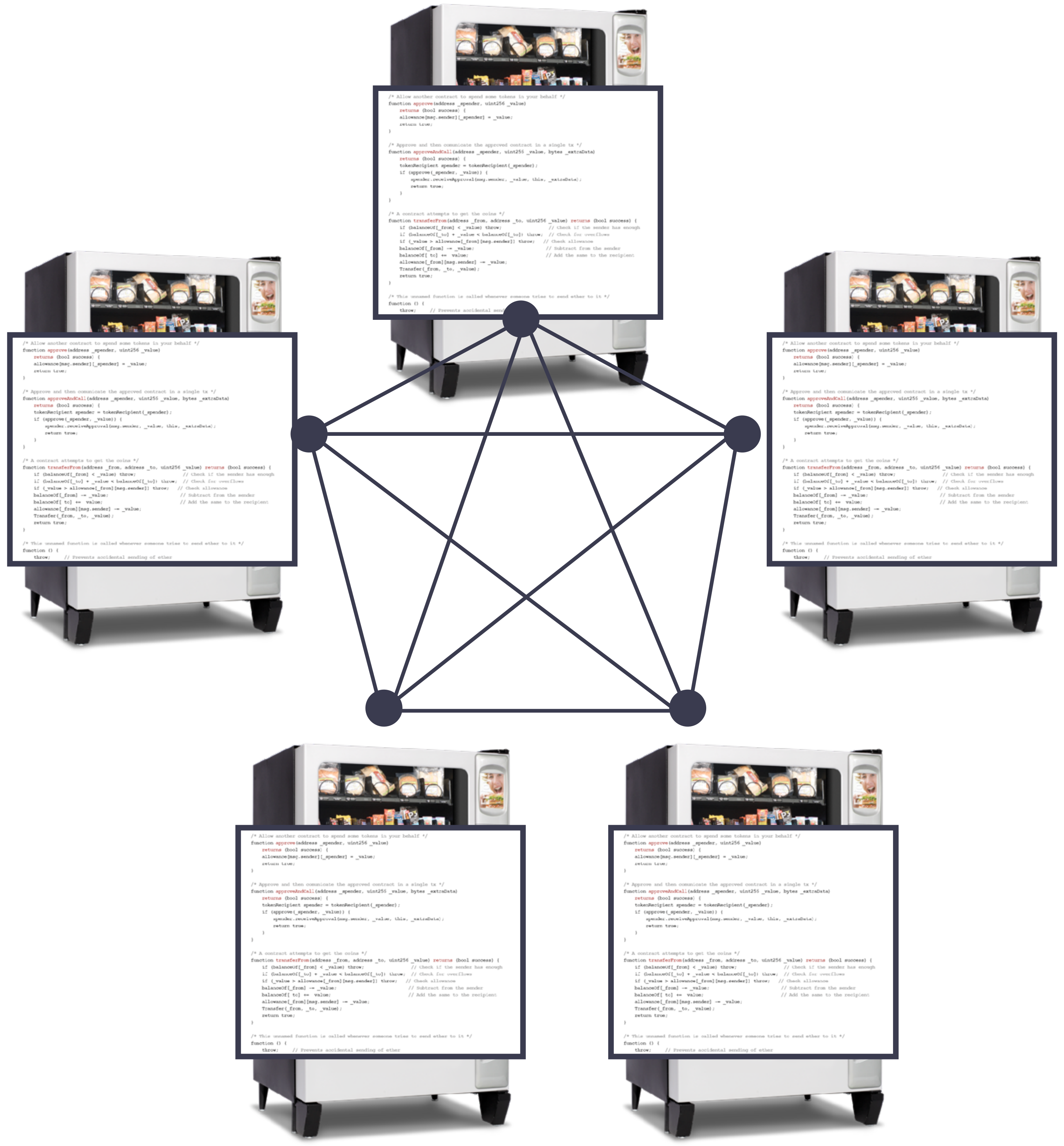
code

1. insert digital coins (tokens)

2. dispense other digital assets or electronic rights

Delegate trust to a decentralised network

- A smart contract is a **replicated automaton** that can trade **digital** assets

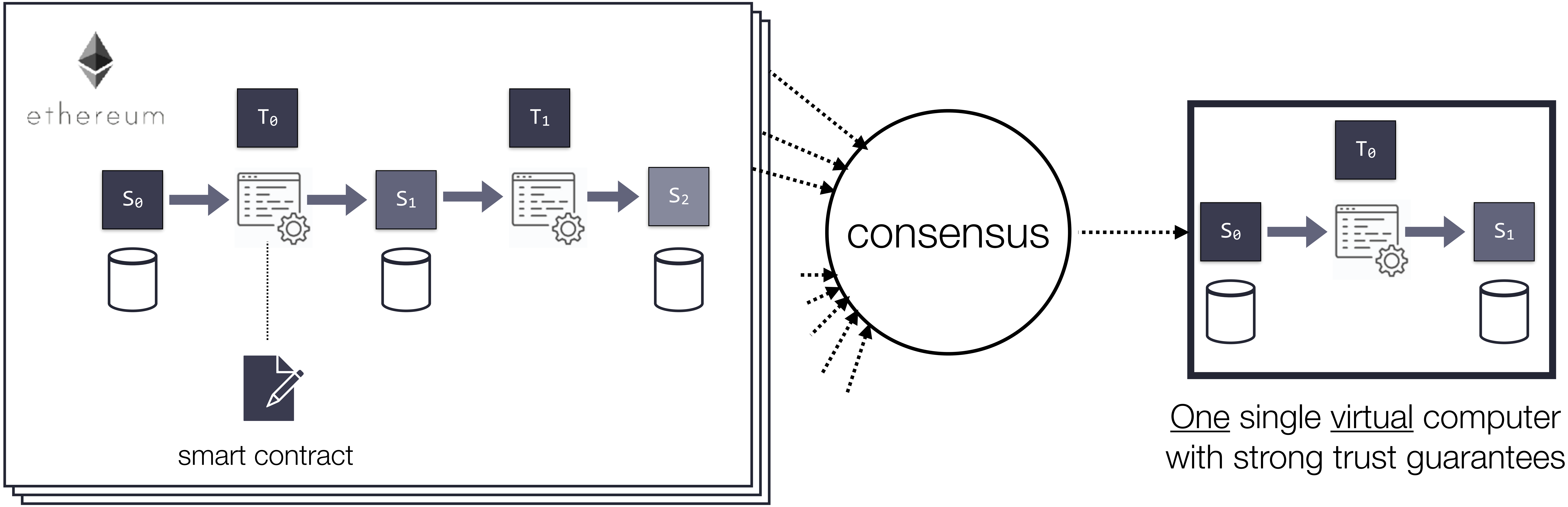


1. insert digital coins (tokens)

2. dispense other digital assets or electronic rights

replicated code

Blockchains as computers that can make "credible commitments"



Many (1000s) untrustworthy physical computers

One single virtual computer with strong trust guarantees

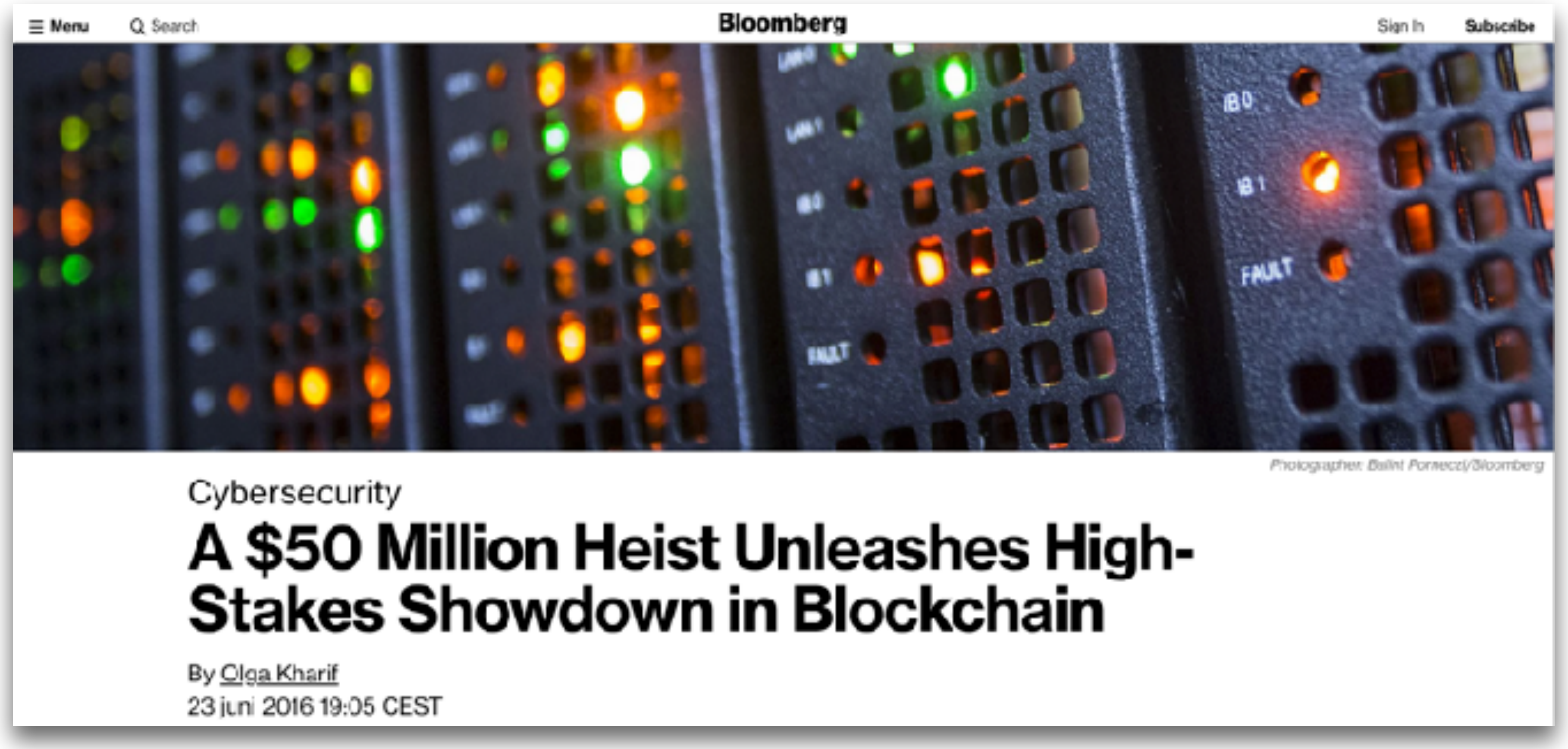
But… one must still trust the contract code

- Once deployed, a smart contract is immutable

- Small bugs may have big consequences

- Re-entrancy hazard → 2016 DAO attack

- Incorrect code initialization → 2017 Parity wallet bug



Bloomberg

Cybersecurity

A $50 Million Heist Unleashes High-Stakes Showdown in Blockchain

By Olga Kharif

23 juni 2016 19:05 CEST

CNBC

MARKETS BUSINESS INVESTING TECH POLITICS CNBC TV WATCHLIST PRO

THE FINTECH EFFECT

THE FINTECH EFFECT

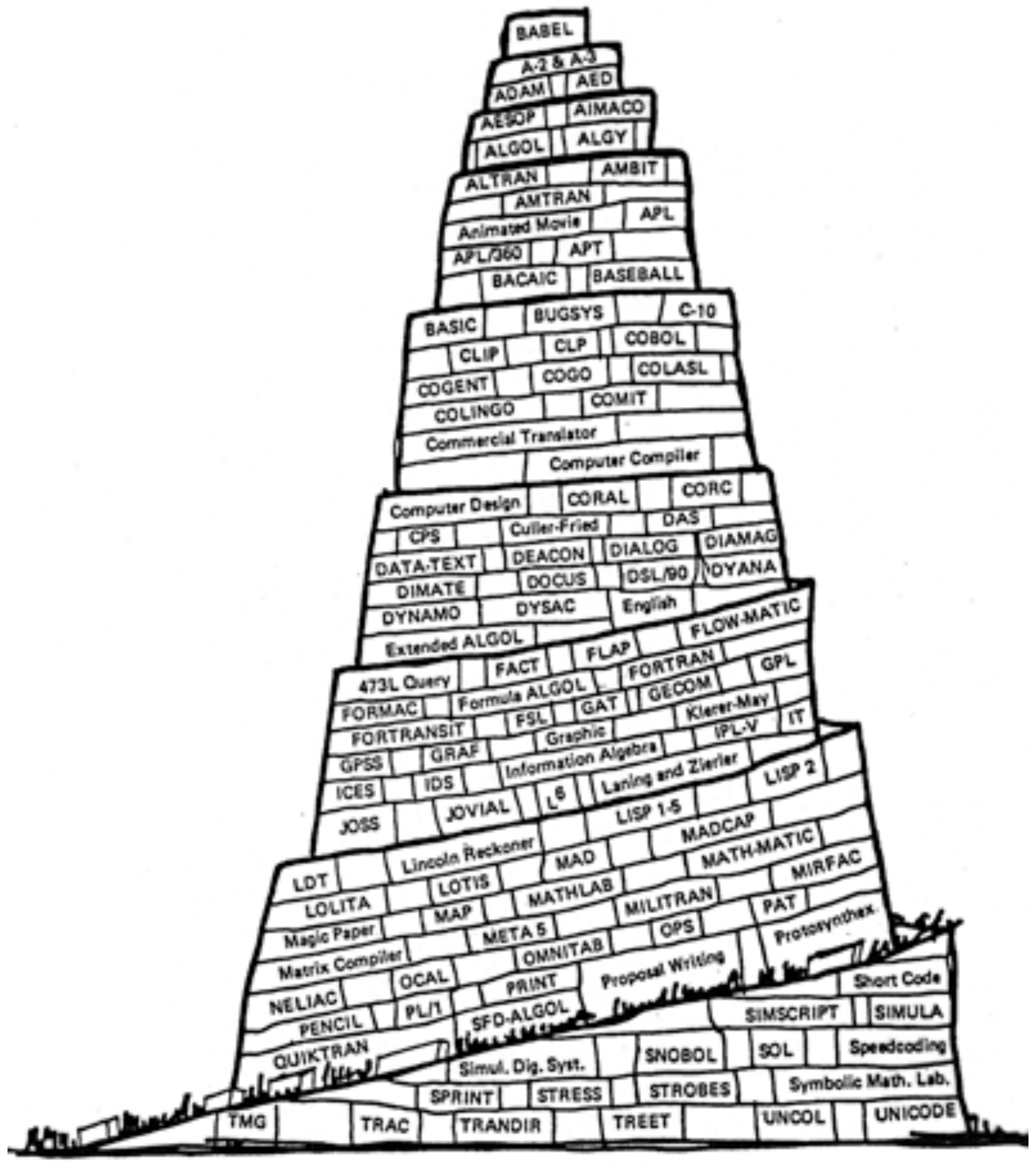
'Accidental' bug may have frozen $280 million worth of digital coin ether in a cryptocurrency wallet

PUBLISHED WED, NOV 8 2017-6:42 AM EST | UPDATED WED, NOV 8 2017-1:20 PM EST

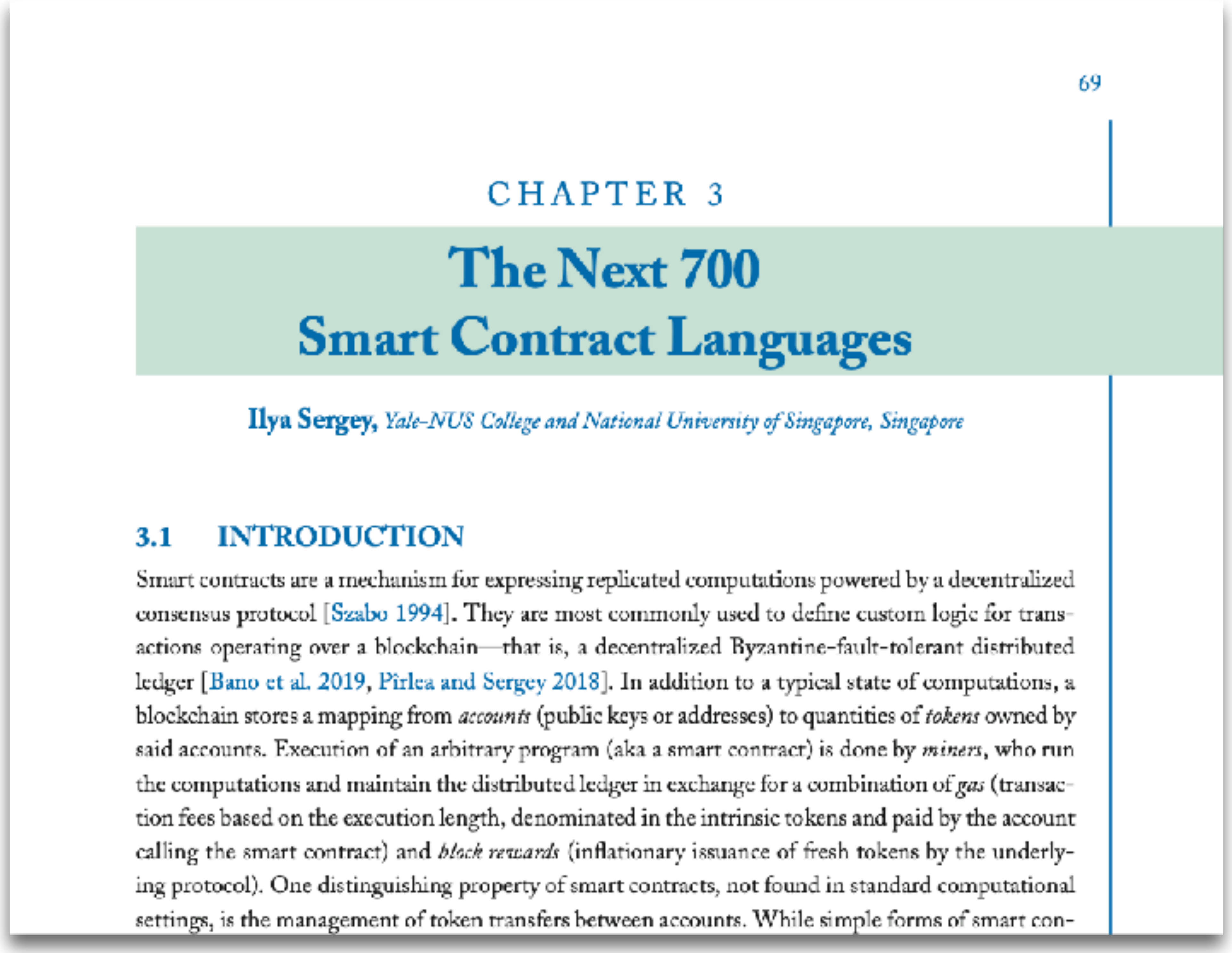
Need better/safer contract languages

- Cambrian explosion of new smart contract languages in the last 5 years

- Solidity, Scilla, Flint, Obsidian, Move, Vyper, Matoko, Plutus, Zoe, Michelson, Clarity, Rholang, …



PL Design & Smart Contracts


69

CHAPTER 3

The Next 700
Smart Contract Languages

Ilya Sergey, *Yale-NUS College and National University of Singapore, Singapore*

3.1 INTRODUCTION

Smart contracts are a mechanism for expressing replicated computations powered by a decentralized consensus protocol [Szabo 1994]. They are most commonly used to define custom logic for transactions operating over a blockchain—that is, a decentralized Byzantine-fault-tolerant distributed ledger [Bano et al. 2019, Pirlea and Sergey 2018]. In addition to a typical state of computations, a blockchain stores a mapping from *accounts* (public keys or addresses) to quantities of *tokens* owned by said accounts. Execution of an arbitrary program (aka a smart contract) is done by *miners*, who run the computations and maintain the distributed ledger in exchange for a combination of *gas* (transaction fees based on the execution length, denominated in the intrinsic tokens and paid by the account calling the smart contract) and *block rewards* (inflationary issuance of fresh tokens by the underlying protocol). One distinguishing property of smart contracts, not found in standard computational settings, is the management of token transfers between accounts. While simple forms of smart con-

Ilya Sergey, “The Next 700 Smart Contract Languages”
in *Principles of Blockchain Systems*, 2021

“[...] we must systematise [language] design so that a new language is a point chosen from a well-mapped space, rather than a laboriously devised construction”

- Peter Landin, “The Next 700 Programming Languages”, 1966

PL Design & Smart Contracts

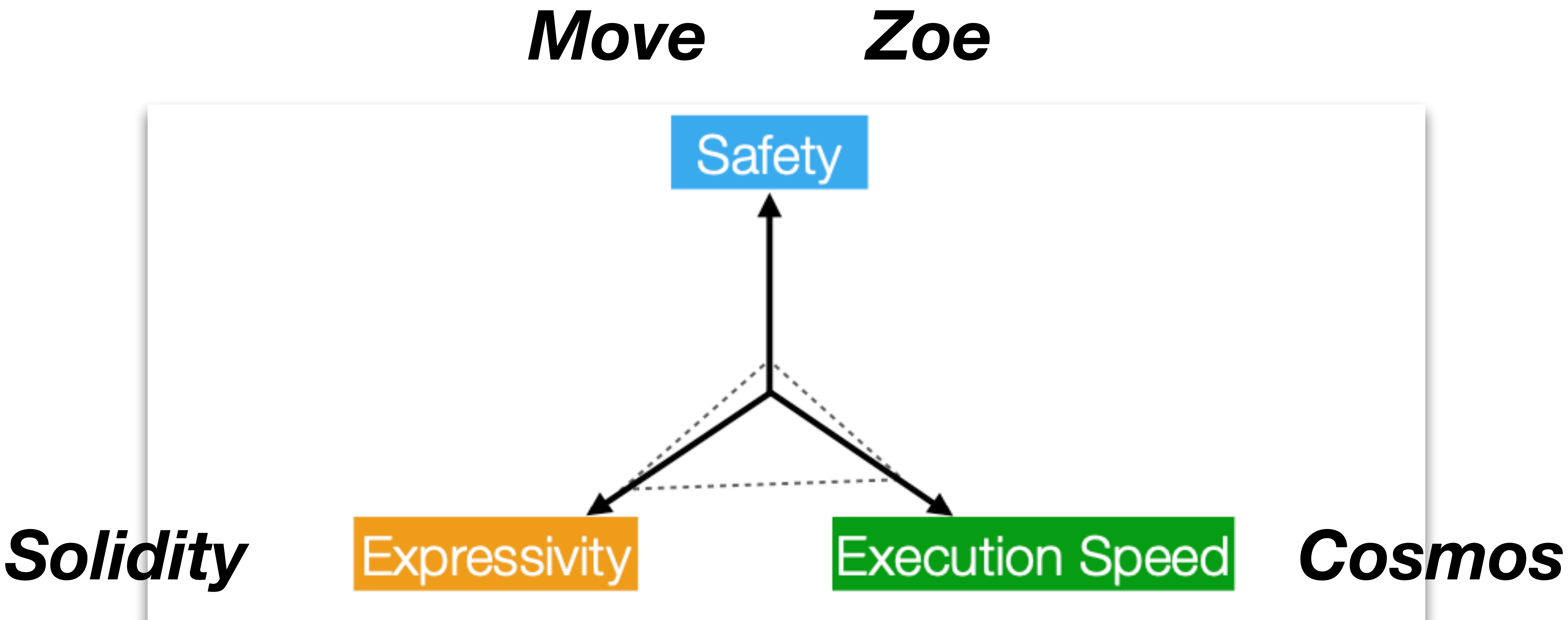
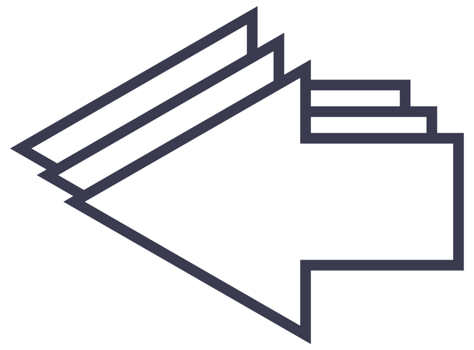


Figure 1.1: Language Design Trade-off. The dashed line shows EVM's design choices.

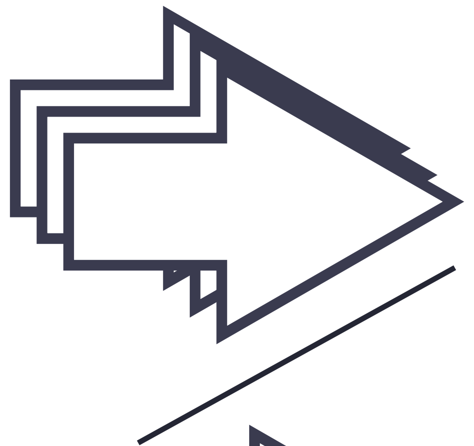
Running example: a Kickstarter-style crowdfunding contract

crowdfunding contract

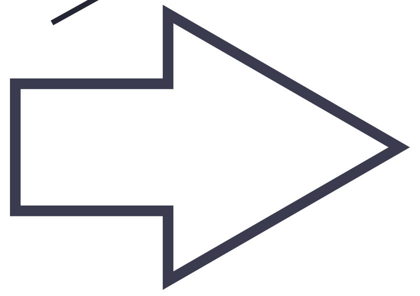


1. Backers deposit tokens (pledge support)

2. Wait until deadline to see if the goal was met

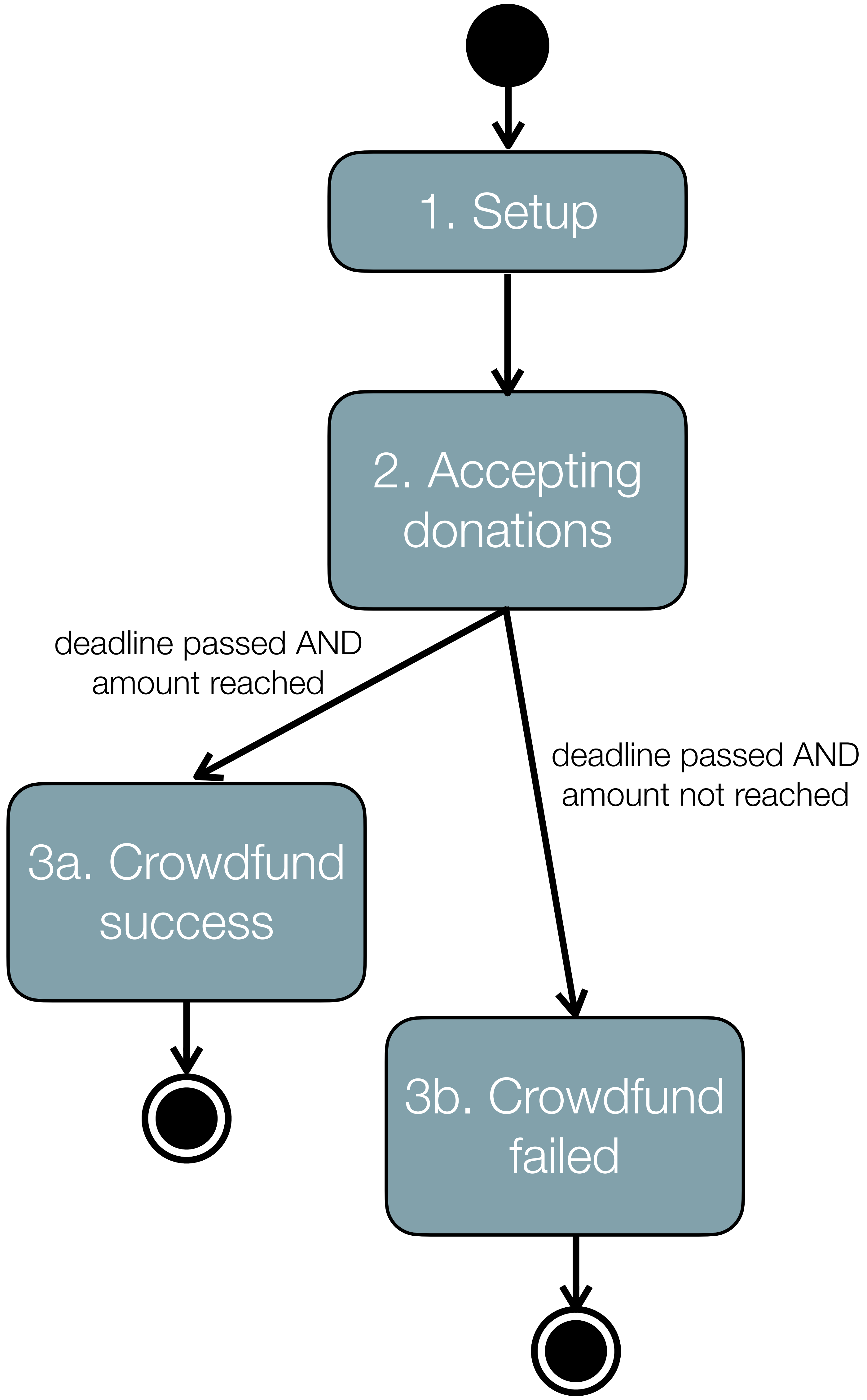


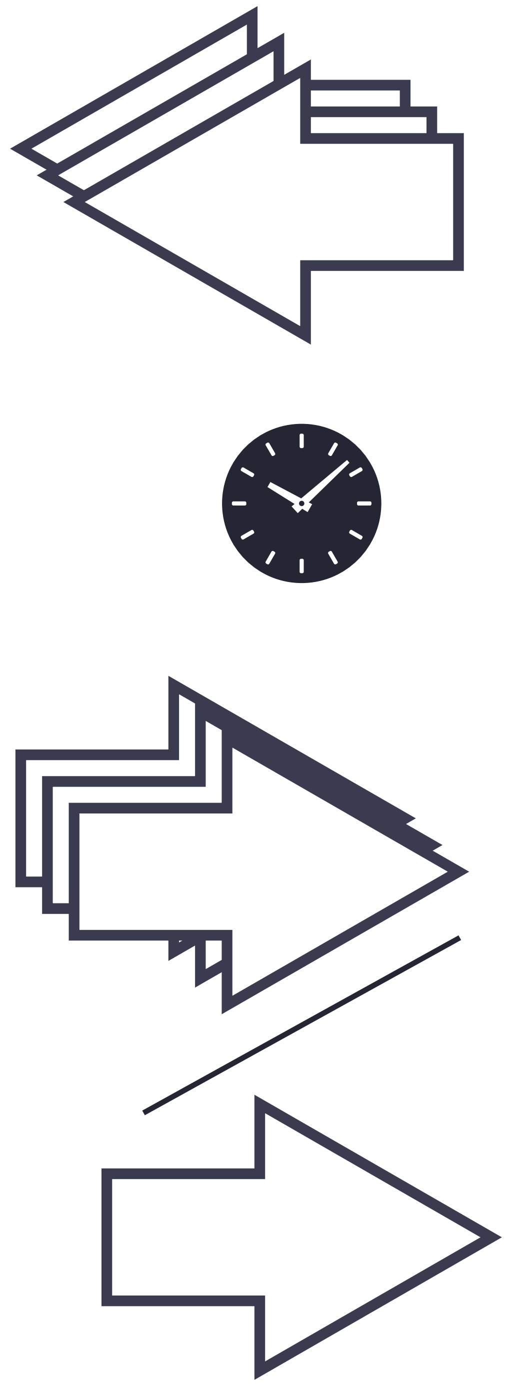
3a. Either the backers withdraw their share…



3b. or the beneficiary withdraws the full deposit

Running example: a Kickstarter-style crowdfunding contract





1. Backers deposit tokens (pledge support)

2. Wait until deadline to see if the goal was met

3a. Either the backers withdraw their share…

3b. or the beneficiary withdraws the full deposit

Solidity on Ethereum

Solidity

- Designed by Gavin Wood (~2013-2014)
- Native language of the Ethereum ecosystem
- Contracts = state + functions
- JavaScript-like syntax
- Compiles to EVM bytecode
- By far the most popular “Web3” language

The crowdfunding contract in Solidity

```
contract Crowdfunding {

    address public owner;    // the beneficiary address
    uint256 public deadline; // campaign deadline in number of days
    uint256 public goal;     // funding goal in ether
    mapping (address => uint256) public backers; // the share of each backer

    constructor(uint256 numberOfDays, uint256 _goal) {
        owner = msg.sender;
        deadline = block.timestamp + (numberOfDays * 1 days);
        goal = _goal;
    }
    function donate() public payable {
        require(block.timestamp < deadline); // before the fundraising deadline
        backers[msg.sender] += msg.value;
    }

    function claimFunds() public {
        require(address(this).balance >= goal); // funding goal met
        require(block.timestamp >= deadline); // after the withdrawal period
        require(msg.sender == owner);
        payable(msg.sender).transfer(address(this).balance);
    }
    function getRefund() public {
        require(address(this).balance < goal); // campaign failed: goal not met
        require(block.timestamp >= deadline); // in the withdrawal period
        uint256 donation = backers[msg.sender];
        backers[msg.sender] = 0;
        payable(msg.sender).transfer(donation);
    }
}
```

(Based on: Ilya Sergey, "The next 700 smart contract languages", Principles of Blockchain Systems 2021)

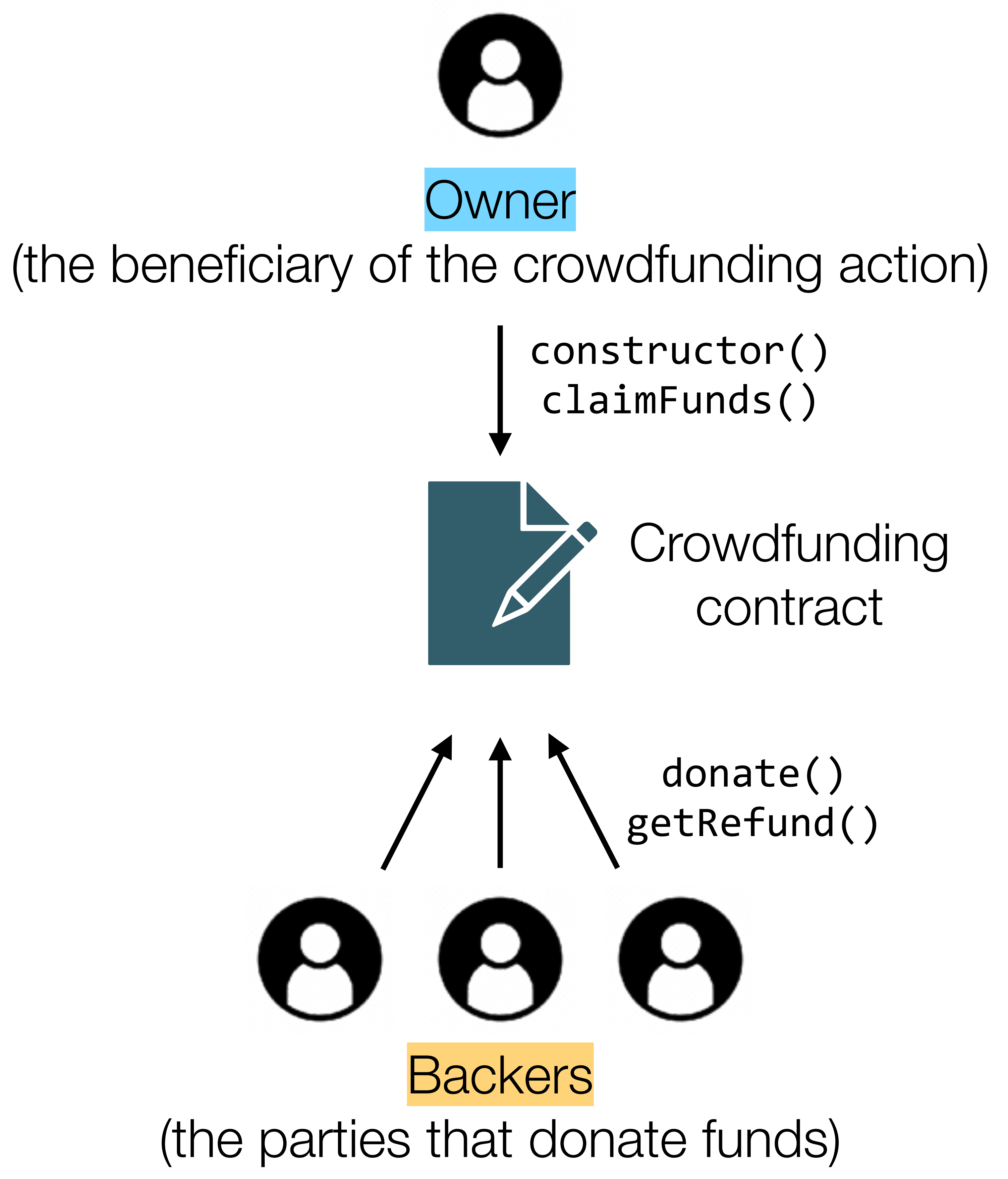
The crowdfunding contract in Solidity

```
contract Crowdfunding {

    address public owner;     // the beneficiary address
    uint256 public deadline; // campaign deadline in number of days
    uint256 public goal;     // funding goal in ether
    mapping (address => uint256) public backers; // the share of each backer

    constructor(uint256 numberOfDays, uint256 _goal) {
        owner = msg.sender;
        deadline = block.timestamp + (numberOfDays * 1 days);
        goal = _goal;
    }
    function donate() public payable {
        require(block.timestamp < deadline); // before the fundraising deadline
        backers[msg.sender] += msg.value;
    }

    function claimFunds() public {
        require(address(this).balance >= goal); // funding goal met
        require(block.timestamp >= deadline); // after the withdrawal period
        require(msg.sender == owner);
        payable(msg.sender).transfer(address(this).balance);
    }
    function getRefund() public {
        require(address(this).balance < goal); // campaign failed: goal not met
        require(block.timestamp >= deadline); // in the withdrawal period
        uint256 donation = backers[msg.sender];
        backers[msg.sender] = 0;
        payable(msg.sender).transfer(donation);
    }
}
```



The crowdfunding contract in Solidity

```
contract Crowdfunding {
    address public owner;    // the beneficiary address
    uint256 public deadline; // campaign deadline in number of days
    uint256 public goal;     // funding goal in ether
    mapping (address => uint256) public backers; // the share of each backer
    constructor(uint256 numberOfDays, uint256 _goal) {
        owner = msg.sender;
        deadline = block.timestamp + (numberOfDays * 1 days);
        goal = _goal;
    }
    function donate() public payable {
        require(block.timestamp < deadline); // before the fundraising deadline
        backers[msg.sender] += msg.value;
    }
    function claimFunds() public {
        require(address(this).balance >= goal); // funding goal met
        require(block.timestamp >= deadline); // after the withdrawal period
        require(msg.sender == owner);
        payable(msg.sender).transfer(address(this).balance);
    }
    function getRefund() public {
        require(address(this).balance < goal); // campaign failed: goal not met
        require(block.timestamp >= deadline); // in the withdrawal period
        uint256 donation = backers[msg.sender];
        backers[msg.sender] = 0;
        payable(msg.sender).transfer(donation);
    }
}
```

Instructions to deposit and withdraw money (ether)

The dangers of imperative code: a **faulty** crowdfunding contract

Faulty

```
contract Crowdfunding {
    address public owner;
    uint256 public deadline;
    uint256 public goal;
    mapping (address => uint256) public backers;

    constructor(uint256 numberOfDays, uint256 _goal) public {
        owner = msg.sender;
        deadline = block.timestamp + (numberOfDays * 1 days);
        goal = _goal;
    }
    function donate() public payable {
        require(block.timestamp < deadline);
        backers[msg.sender] = msg.value;
    }

    function claimFunds() public {
        require(address(this).balance >= goal);
        require(block.timestamp >= deadline);
        require(msg.sender == owner);
        payable(msg.sender).transfer(address(this).balance);
    }
    function getRefund() public {
        require(address(this).balance < goal); // goal not met
        require(now >= deadline); // in the withdrawal period
        uint256 donation = backers[msg.sender];
        payable(msg.sender).transfer(donation);
        backers[msg.sender] = 0;
    }
}
```

Original

```
contract Crowdfunding {
    address public owner;
    uint256 public deadline;
    uint256 public goal;
    mapping (address => uint256) public backers;

    constructor(uint256 numberOfDays, uint256 _goal) {
        owner = msg.sender;
        deadline = block.timestamp + (numberOfDays * 1 days);
        goal = _goal;
    }
    function donate() public payable {
        require(block.timestamp < deadline);
        backers[msg.sender] += msg.value;
    }

    function claimFunds() public {
        require(address(this).balance >= goal);
        require(block.timestamp >= deadline);
        require(msg.sender == owner);
        payable(msg.sender).transfer(address(this).balance);
    }
    function getRefund() public {
        require(address(this).balance < goal);
        require(block.timestamp >= deadline);
        uint256 donation = backers[msg.sender];
        backers[msg.sender] = 0;
        payable(msg.sender).transfer(donation);
    }
}
```

A long list of vulnerabilities in Solidity contracts

SWC Registry

Smart Contract Weakness Classification and Test Cases

The following table contains an overview of the SWC registry. Each row consists of an SWC identifier (ID), weakness title, CWE parent and list of related code samples. The links in the ID and Test Cases columns link to the respective SWC definition. Links in the Relationships column link to the CWE Base or Class type.

ID	Title	Relationships	Test cases
SWC-136	Unencrypted Private Data On-Chain	CWE-767: Access to Critical Private Variable via Public Method	• odd_even.sol • odd_even_fixed.sol
SWC-135	Code With No Effects	CWE-1164: Irrelevant Code	• deposit_box.sol • deposit_box_fixed.sol • wallet.sol • wallet_fixed.sol
SWC-134	Message call with hardcoded gas amount	CWE-655: Improper Initialization	• hardcoded_gas_limits.sol
SWC-133	Hash Collisions With Multiple Variable Length Arguments	CWE-294: Authentication Bypass by Capture-replay	• access_control.sol • access_control_fixed_1.sol • access_control_fixed_2.sol

Move on Aptos

Move

- Origins in Facebook's *Diem* (neé *Libra*) project
- Green field language design for smart contracts
- Rust-like, with custom virtual machine
- *Resource types*: linear types to track objects with monetary value (avoid accidental copies or drops)
- Now used as part of Aptos and Sui blockchains

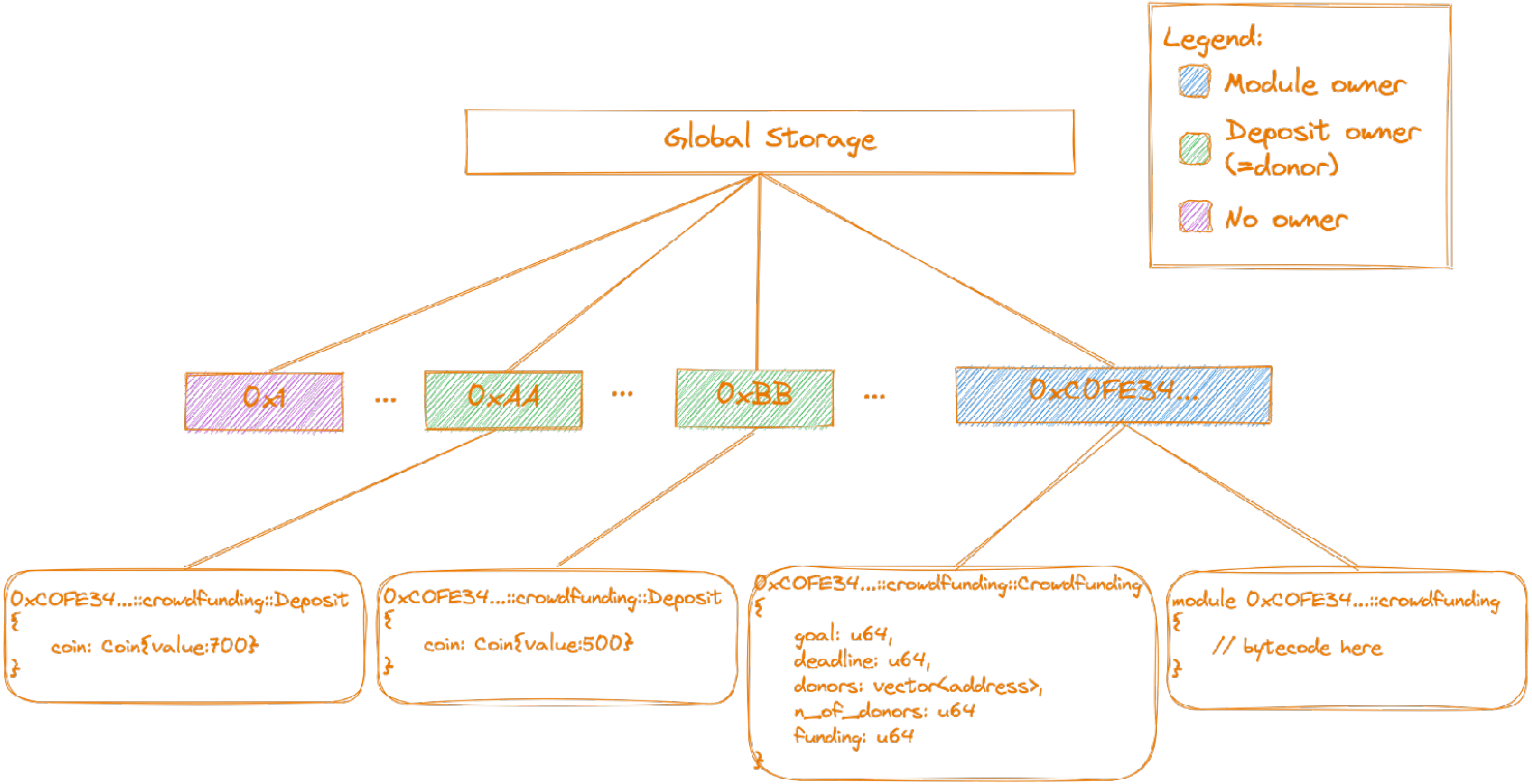
Move: A Language With Programmable Resources

Sam Blackshear, Evan Cheng, David L. Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Rain, Dario Russi, Stephane Sezer, Tim Zakian, Runtian Zhou *

Note to readers: This report was published before the Association released White Paper v2.0, which includes a number of key updates to the Libra payment system. Outdated links have been removed, but otherwise, this report has not been modified to incorporate the updates and should be read in that context.

Abstract. We present *Move*, a safe and flexible programming language for the Libra Blockchain [1][2]. Move is an executable bytecode language used to implement custom transactions and smart contracts. The key feature of Move is the ability to define custom *resource types* with semantics inspired by linear logic [3]: a resource can never be copied or implicitly discarded, only moved between program storage locations. These safety guarantees are enforced statically by Move's type system. Despite these special protections, resources are ordinary program values — they can be stored in data structures, passed as arguments to procedures, and so on. First-class resources are a very general concept that programmers can use not only to implement safe digital assets but also to write correct business logic for wrapping assets and enforcing access control policies. The safety and expressivity of Move have enabled us to implement significant parts of the Libra protocol in Move, including Libra coin, transaction processing, and validator management.

Move’s Global Storage account model



Crowdfunding contract in Move

```
module crowdfunding {

  struct Deposit<phantom CoinType> has key {
    coin: Coin<CoinType>,
  }

  struct CrowdFunding<phantom CoinType> has key {
    goal: u64,
    deadline: u64,
    backers: vector<address>,
    funding: u64,
  }

  public entry fun initialise_crowdfunding<CoinType>(account: &signer, goal: u64, minutes: u64) {
    let addr = signer::address_of(account);
    assert!(addr == @owner, EONLY_DEPLOYER_CAN_INITIALISE);
    let now = timestamp::now_seconds() / MINUTE_CONVERSION_FACTOR;
    move_to(account, CrowdFunding<CoinType> {
      goal: goal,
      deadline: now + minutes,
      backers: vector::empty<address>(),
      funding: 0,
    });
  }
```

The crowdfunder stores a 'CrowdFunding' resource to track campaign state

Crowdfunding contract in Move

```
public entry fun donate<CoinType>(account: &signer, fund_addr: address, amount: u64) acquires Deposit, CrowdFunding {
    assertCrowdfundingInitialised<CoinType>(fund_addr);
    assertDeadlinePassed<CoinType>(fund_addr, false);

    let addr = signer::address_of(account);
    assert!(coin::balance<CoinType>(addr) >= amount, ENO_SUFFICIENT_FUND);
    let coin_to_deposit = coin::withdraw<CoinType>(account, amount);
    let val = coin::value<CoinType>(&coin_to_deposit);
    let cf = borrow_global_mut<CrowdFunding<CoinType>>(fund_addr);

    if (!exists<Deposit<CoinType>>(addr)) {
        let to_deposit = Deposit<CoinType> {coin: coin_to_deposit};
        move_to(account, to_deposit);
        let backers = &mut cf.backers;
        vector::push_back<address>(backers, addr);
    } else {
       let deposit = borrow_global_mut<Deposit<CoinType>>(addr);
       coin::merge<CoinType>(&mut deposit.coin, coin_to_deposit);
    }
    cf.funding = cf.funding + val;
}
```

Each backer receives a ‘Deposit’ resource to track their donation

Crowdfunding contract in Move

```
public entry fun claimFunds<CoinType>(account: &signer, fund_addr: address) acquires Deposit, CrowdFunding {
    assertCrowdfundingInitialised<CoinType>(fund_addr);
    assertGoalReached<CoinType>(fund_addr, true);
    assertDeadlinePassed<CoinType>(fund_addr, true);
    let addr = signer::address_of(account);
    assert!(addr == fund_addr, EONLY_CROWDFUNDING_OWNER);
    let backers = &mut borrow_global_mut<CrowdFunding<CoinType>>(fund_addr).backers;
    withdrawCoinsFromDeposits<CoinType>(addr, backers);
    destroyCrowdfunding<CoinType>(fund_addr);
}


fun withdrawCoinsFromDeposits<CoinType>(fund_addr: address, backers: &mut vector<address>) acquires Deposit {
    while (!vector::is_empty<address>(backers)) {
        let backer_addr = vector::pop_back<address>(backers);
        let Deposit<CoinType>{ coin: coins } = move_from<Deposit<CoinType>>(backer_addr);
        coin::deposit(fund_addr, coins);
    }
}
```

‘Deposit’ resource is destroyed and coins are added to crowdfunding balance

How does Move address Solidity's most common vulnerabilities?

Not so smart contracts ···

# ID	Aa Name	Vulnerability type	Addressed by Move	Solved in Solidity 0.8+
1	Integer Overflow	Overflow/ Underflow	Yes	Yes
2	Forced Ether Reception	Access Control	Yes	No
3	Unprotected Function	Access Control	No	No
4	Wrong Constructor Name	Access Control Constructor naming	Yes	Yes
5	Reentrancy	Logic	Yes	No
6	Unchecked External Call	Logic	Yes	No
7	Variable Shadowing	Logic	No	
8	Incorrect Interface	Wrong Interface	Yes	No
9	Bad Randomness	Blockchain Infrastructure	No	No
10	Denial of Service	Blockchain Infrastructure	No	No
11	Race Condition	Blockchain Infrastructure	No	No

Crytic, (2018). Not so smart contracts. https://github.com/crytic/not-so-smart-contracts

Secbit, (2018). Awesome buggy erc20 tokens. https://github.com/sec-bit/awesome-buggy-erc20-tokens

Certik's "immovables"

- Bad smells in Move code

Blogs Tech & Dev

Moving the Immovables: Lessons Learned From Our Aptos Smart Contract Audit

14-11-2022

Move is a programming language specifically designed for building secure and formally verified smart contracts. Move's language features provide a strong set of security protections through strict type enforcement and load-time verifications. Developers who master Move's built-in resources and programming patterns can produce more secure projects than those developed in conventional languages that lack these features.

Resurrecting unsafe types (e.g. unsigned int with overflow)

- Example: reintroducing unsafe integer arithmetic (with underflow/overflow) when porting code from Solidity…

```
struct I128 has copy, drop, store {
  bits: u128
}

/// u128 with the first bit set. An `I128` is negative if this bit is set.
const U128_WITH_FIRST_BIT_SET: u128 = 1 << 127;

public fun is_neg(x: &I128): bool {
  x.bits > U128_WITH_FIRST_BIT_SET
}

public fun add(a: &I128, b: &I128): I128 {
  if (a.bits >> 127 == 0) { // A is positive
    if (b.bits >> 127 == 0) { // B is positive
      return I128 { bits: a.bits + b.bits }
    } else {    // B is negative
      if (b.bits - (1 << 127) <= a.bits)
        return I128 { bits: a.bits - (b.bits - (1 << 127)) };  // Return positive
      return I128 { bits: b.bits - a.bits } // Return negative
    }
  } else { // A is negative
    if (b.bits >> 127 == 0) { // B is positive
      if (a.bits - (1 << 127) <= b.bits)
        return I128 { bits: b.bits - (a.bits - (1 << 127)) }; // Return positive
      return I128 { bits: a.bits - b.bits } // Return negative
    } else { // B is negative
      return I128 { bits: a.bits + (b.bits - (1 << 127)) }
    }
  }
}
```

Misunderstood reference safety

- Example: marking important structs with copy, drop capabilities thus bypassing the borrows checker

```
struct Config has key {
  stores: vector<CoinStore>,
}

struct CoinStore has copy, store, drop {
  coint_type: String,
  fees: u8,
  // many other fields are omitted
}

const ERROR_COIN_TYPE_NOT_FOUND: u64 = 3;

fun borrow_mut(account: &signer, coin_type: &String): CoinStore acquires Config
{ let address = signer::address_of(account);
  assert!(address == @contract_owner, ERROR_PERMISSION_DENIED);
  let config = borrow_global_mut<Config>(address);

  let (e, i) = contains(&config.stores, coin_type);
  if (e) {
    *vector::borrow_mut(&mut config.stores, i)
  } else {
    abort ERROR_COIN_TYPE_NOT_FOUND
  }
}

public entry fun increase_fees<C>(account: &signer) acquires Config {
  let coin_type = type_name<C>();
  let store = borrow_mut(account, &coin_type);
  store.fees = store.fees + 1;
}
```

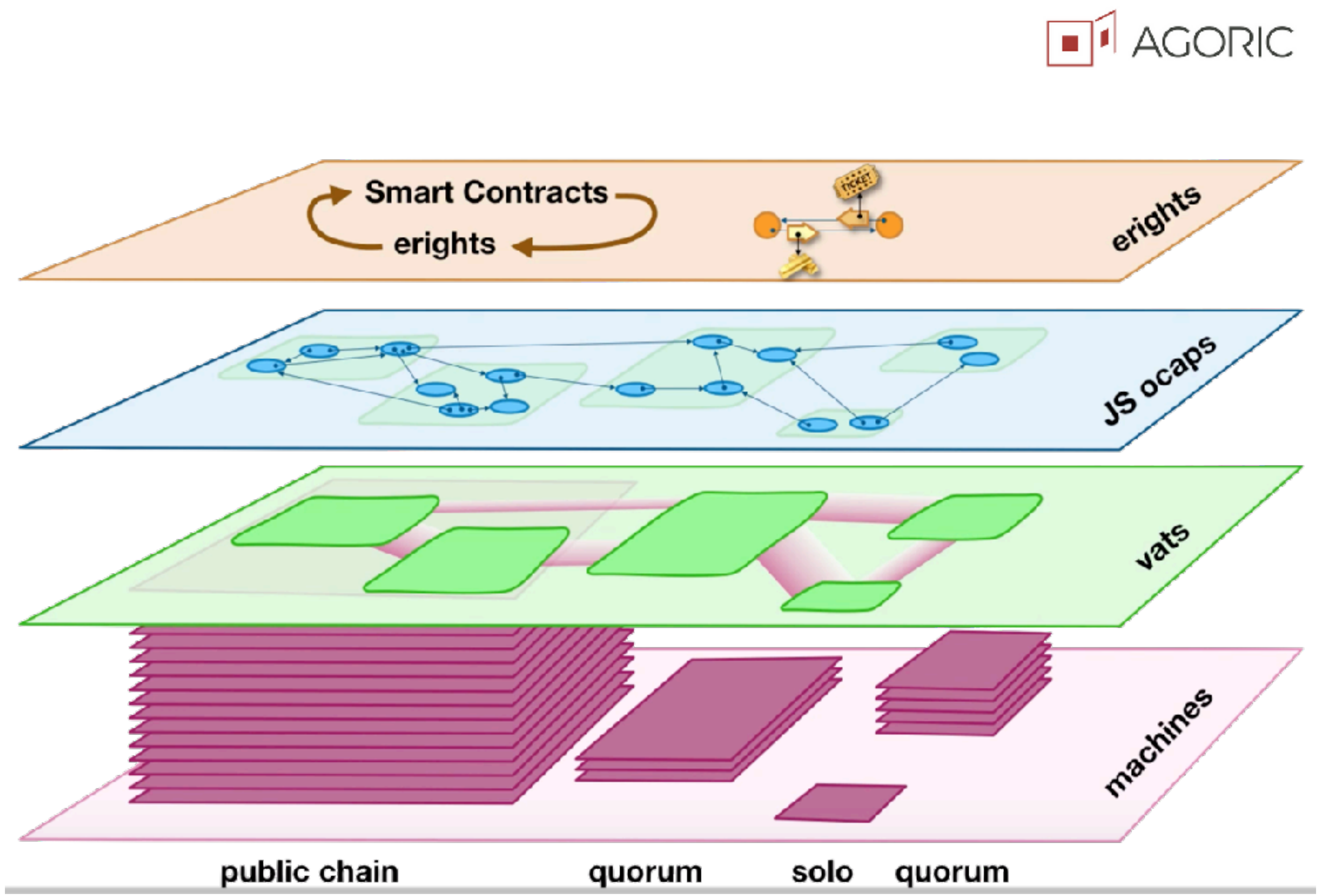
Zoe on Agoric

Agoric: use JavaScript to write secure smart contracts for Web3

Digital assets
managed by
Zoe framework

written in
Hardened JavaScript

executing on
A public blockchain
(Tendermint / Cosmos)



Object-capability security

"(Language-based) Security as extreme Modularity"

\- Mark S. Miller

Modularity: avoid needless dependencies (to prevent bugs)

Security: avoid needless authority (to prevent exploits)

"Only connectivity begets connectivity"

Three simple rules that describe how authority can be acquired in a capability-secure system:

Creation: e.g. alice creates carol herself

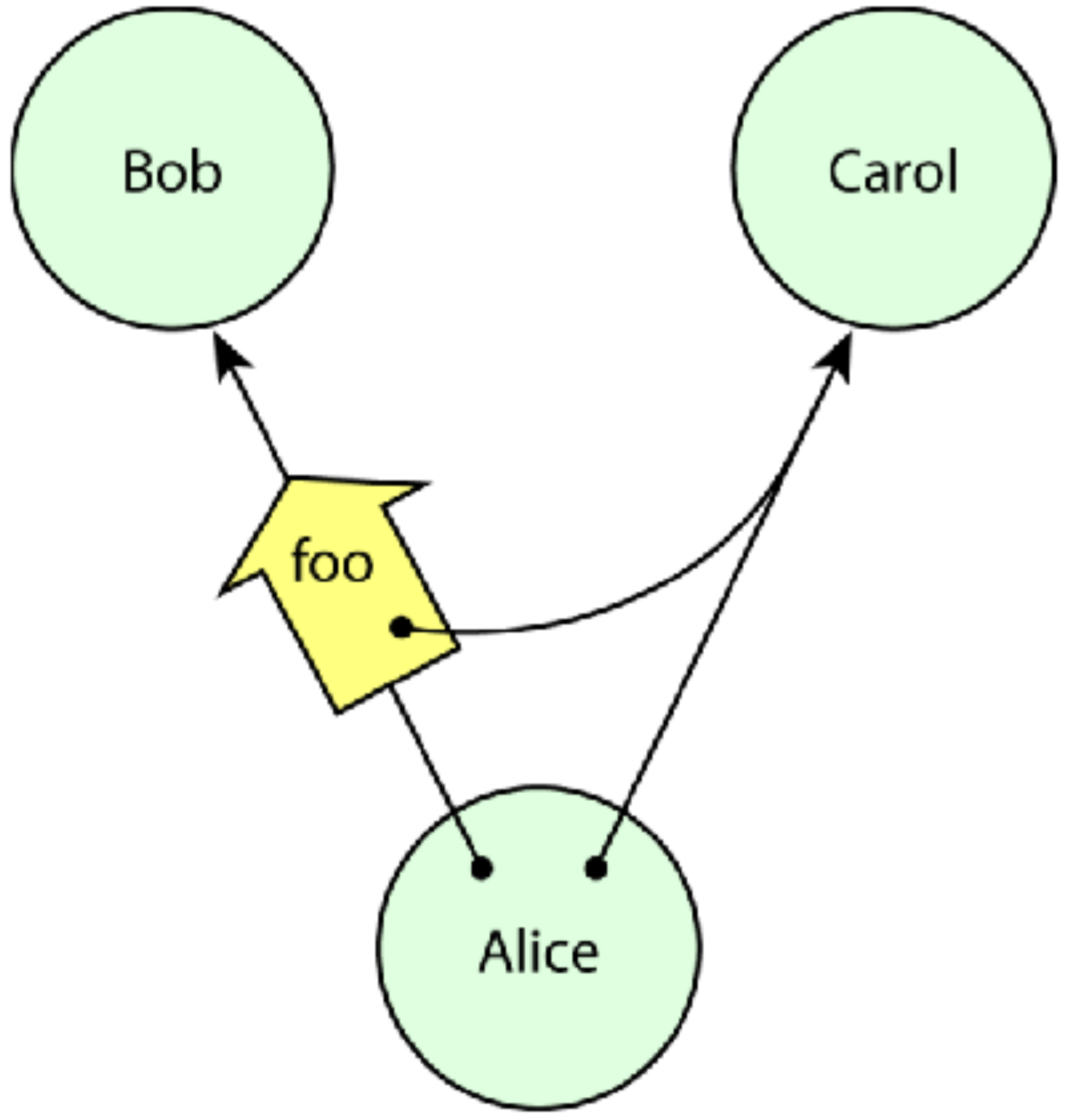
```
// alice executes:
let carol = makeCarol()
```

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

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

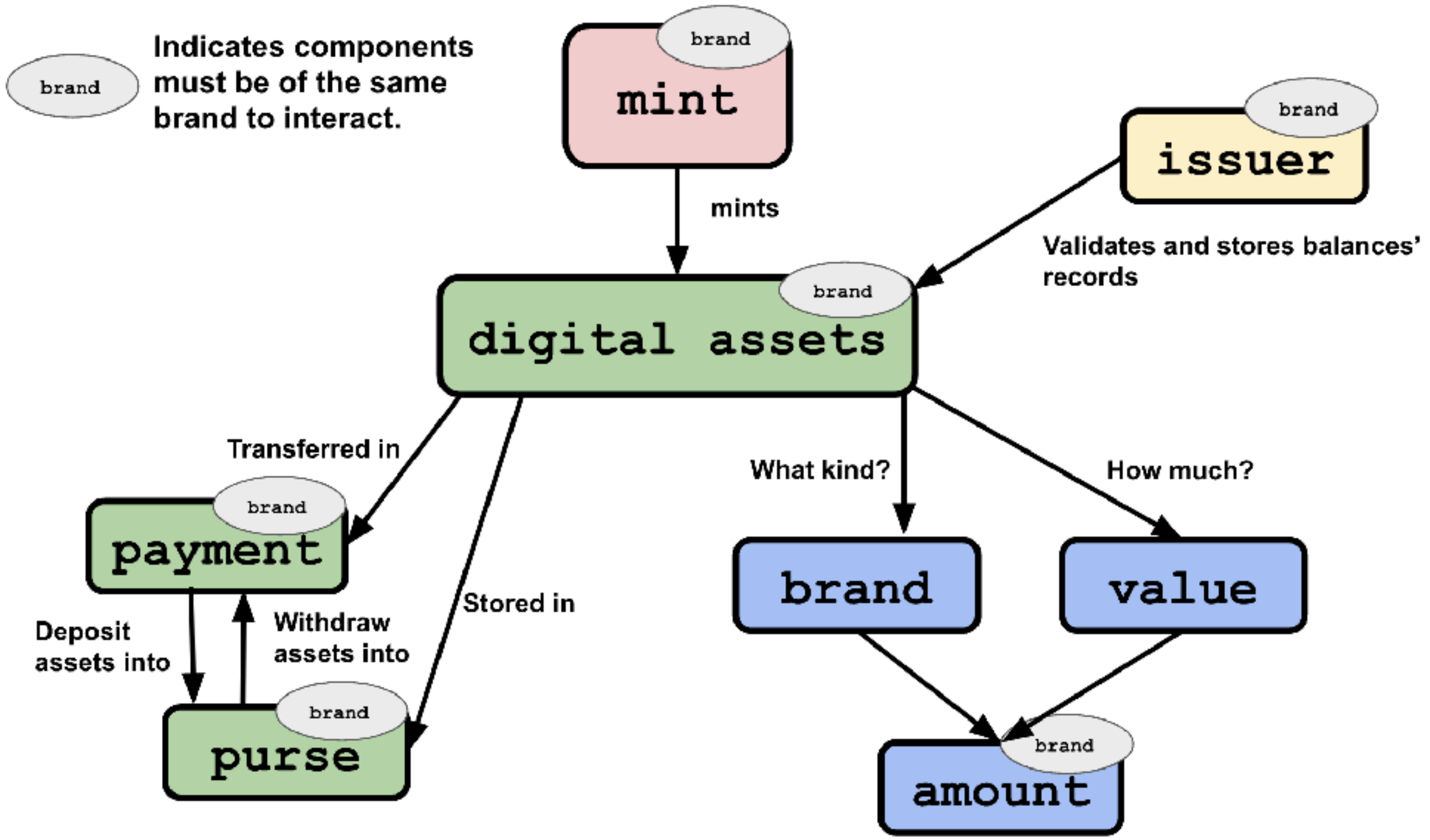
Transfer: e.g. alice transfers carol to bob

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



Zoe ERTP: Electronic Rights Transfer Protocol

- In Zoe, digital assets are represented as Objects
- Access to a Payment object => authority to spend the asset



```
const start = zcf => {

  assertIssuerKeywords(zcf, harden(['Donation']));

  const { coinBrand, deadline, goal } = zcf.getTerms();

  const target = AmountMath.make(coinBrand, goal);
  const backerseats = [];

  const claimOfferHandler = seat => { … };

  const donateOfferHandler = seat => { … };

  const reclaimOfferHandler = seat => { … };

  const creatorFacet = Far('creatorFacet', {
      makeClaimInvitation: () => zcf.makeInvitation(claimOfferHandler, 'claim'),
  });

  const donorFacet = Far('donorFacet', {
      makeDonateInvitation: () => zcf.makeInvitation(donateOfferHandler, ‘donate'),
      makeReclaimFundsInv: () => zcf.makeInvitation(reclaimOfferHandler, 'reclaim'),
  });

  return harden({ creatorFacet, donorFacet });
};

export { start };
```

Interface to Zoe

Separate interfaces per contract party

Zoe: Offer Safety

```
const alphaCoin = makeIssuerKit("AlphaCoin");
const alphaCoinPurseBob = await E(alphaCoin.issuer).makeEmptyPurse();
…

// Bob donates 30 AlphaCoins
const bobDonateInvitation = await E(donorFacet).makeDonateInvitation();
const bobProposal = harden({
  give: { Donation: AmountMath.make(alphaCoin.brand, 30n) },
  exit: { waived: null },
});

//Bob takes 30 AlphaCoins out of his purse, creating a payment
const bobPayment = alphaCoinPurseBob.withdraw(AmountMath.make(alphaCoin.brand, 30n));

//Bob offers the invitation, the proposal and the payment to zoe and gets a seat in return
const bobSeat = await E(zoe).offer(bobDonateInvitation, bobProposal, harden({ Donation: bobPayment }));

//Bob should get an offer result which states that his donation has been made
await E(bobSeat).getOfferResult();

// if the campaign fails then Bob gets a refund
await E(bobSeat).getPayout('Donation').then(payment => alphaCoinPurseBob.deposit(payment));
```

Users explicitly specify what assets a contract can access. Zoe keeps these assets in escrow while the contract executes.

```
const claimOfferHandler = seat => {

  let totalAmount = AmountMath.make(coinBrand, 0n);
  donors.forEach(donorSeat => {
      totalAmount = AmountMath.add(totalAmount,
                                   donatorSeat.getAmountAllocated('Donation', coinBrand), coinBrand);
  });

  // if crowdfunding succeeded…
  if (deadlinePassed() && AmountMath.isGTE(totalAmount, target)) {
      donors.forEach(donorSeat => {
          seat.incrementBy(donorSeat.decrementBy(harden(donorSeat.getCurrentAllocation())));
          zcf.reallocate(donorSeat, seat);
          donorSeat.exit();
      });
      seat.exit();
      return ‘Donations claimed';
  }

  //if there is still time left, notify that the deadline hasn't expired
  seat.exit();
  return 'The deadline has not yet passed';
};
```

```
const claimOfferHandler = seat => {

  let totalAmount = AmountMath.make(coinBrand, 0n);
  donors.forEach(donorSeat => {
      totalAmount = AmountMath.add(totalAmount,
                                   donatorSeat.getAmountAllocated('Donation', coinBrand), coinBrand);
  });

  // if crowdfunding succeeded…
  if (deadlinePassed() && AmountMath.isGTE(totalAmount, target)) {
      donors.forEach(donorSeat => {
          seat.incrementBy(donorSeat.decrementBy(harden(donorSeat.getCurrentAllocation())));
          zcf.reallocate(donorSeat, seat);
          donorSeat.exit();
      });
      seat.exit();
      return ‘Donations claimed';
  }

  //if there is still time left, notify that the deadline hasn't expired
  seat.exit();
  return 'The deadline has not yet passed';
};
```

APIs for safe math with currency amounts

```
const claimOfferHandler = seat => {

  let totalAmount = AmountMath.make(coinBrand, 0n);
  donors.forEach(donorSeat => {
      totalAmount = AmountMath.add(totalAmount,
                                   donatorSeat.getAmountAllocated('Donation', coinBrand), coinBrand);
  });

  // if crowdfunding succeeded…
  if (deadlinePassed() && AmountMath.isGTE(totalAmount, target)) {
      donors.forEach(donorSeat => {
          seat.incrementBy(donorSeat.decrementBy(harden(donorSeat.getCurrentAllocation())));
          zcf.reallocate(donorSeat, seat);
          donorSeat.exit();
      });
      seat.exit();
      return 'Donations claimed';
  }

  //if there is still time left, notify that the deadline hasn't expired
  seat.exit();
  return 'The deadline has not yet passed';
};
```

APIs to access and modify allocation of assets between parties

Zoe: programming patterns

Contract Requirements

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

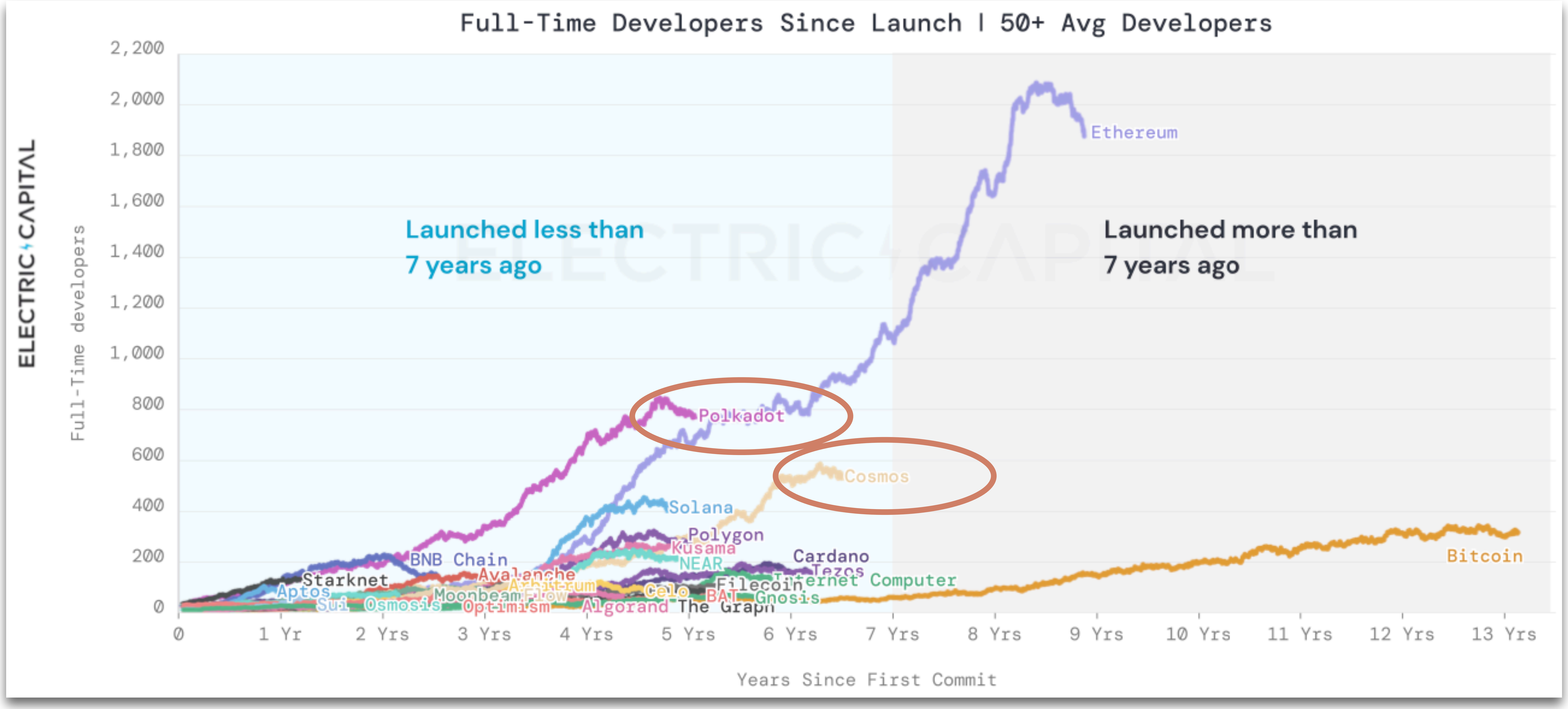
When writing a smart contract to run on Zoe, you need to know the proper format and other expectations.

(source: Agoric)

- Interface objects must be explicitly made immutable (“hardened”)
- Remote objects: must use eventual send API to send async messages
- Objects received from counterparty must first be verified with a trusted issuer
- No static types: manual user input validation
- Must carefully reason about what objects to keep private

General-purpose languages on the blockchain

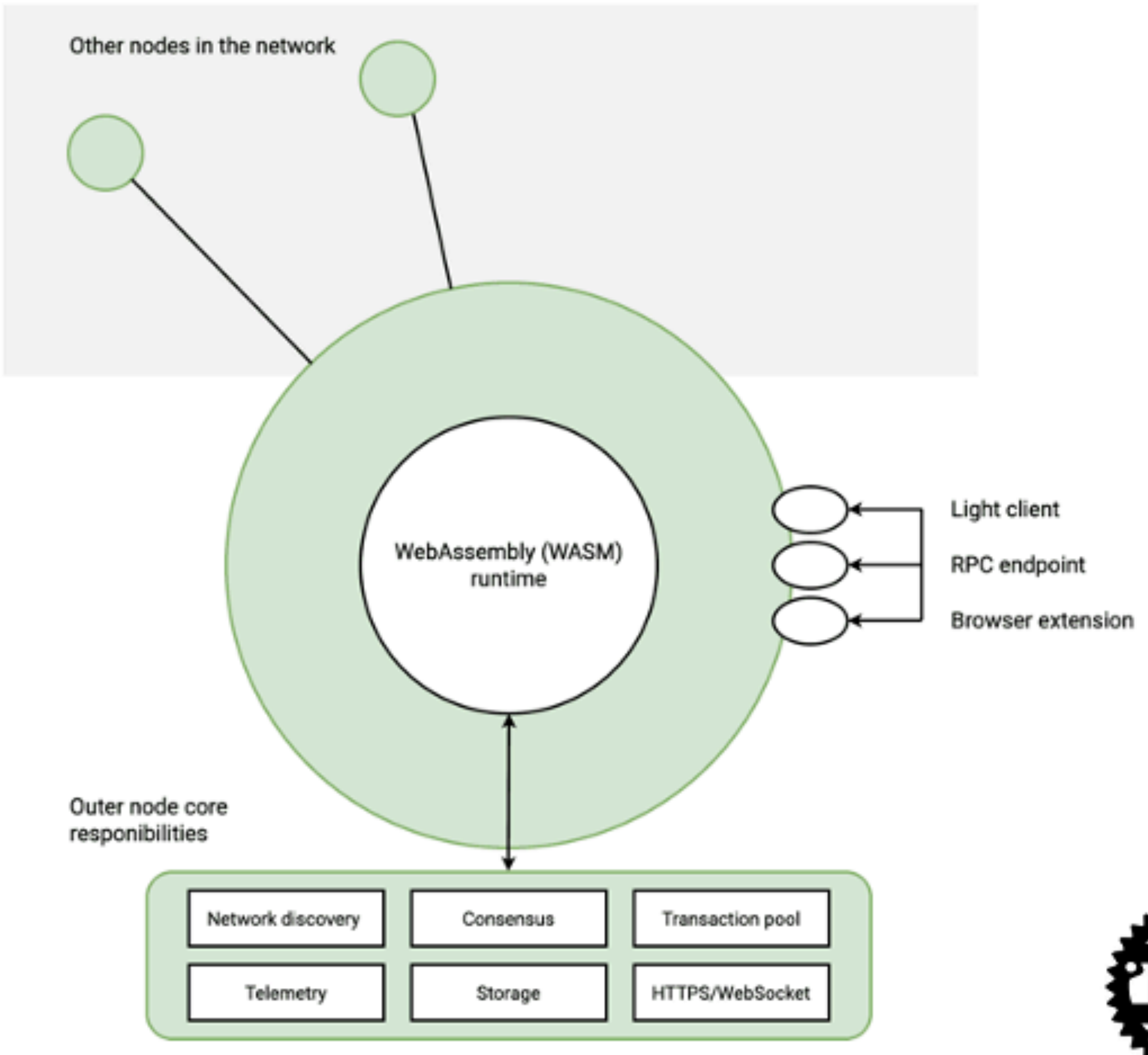
Web3: a growing developer ecosystem



(Source: Electric Capital, blockchain developer report, January 2023)

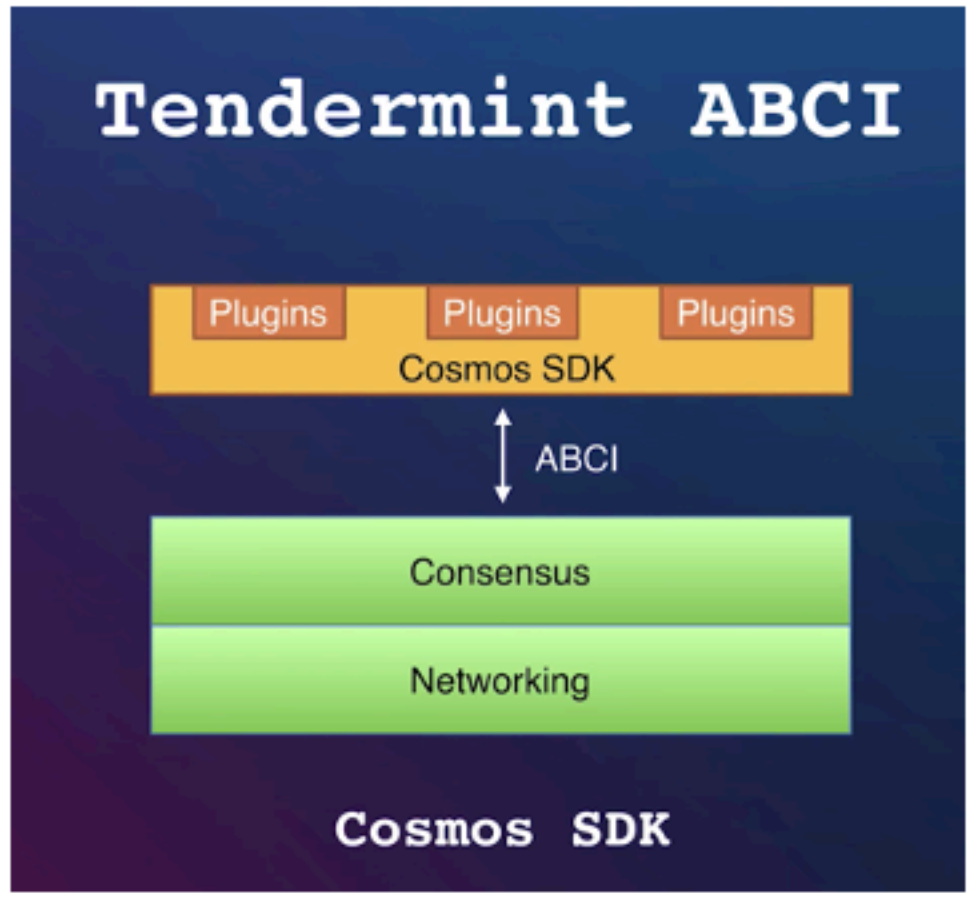
Emerging Appchain SDK frameworks

Polkadot “Substrate” (Rust)



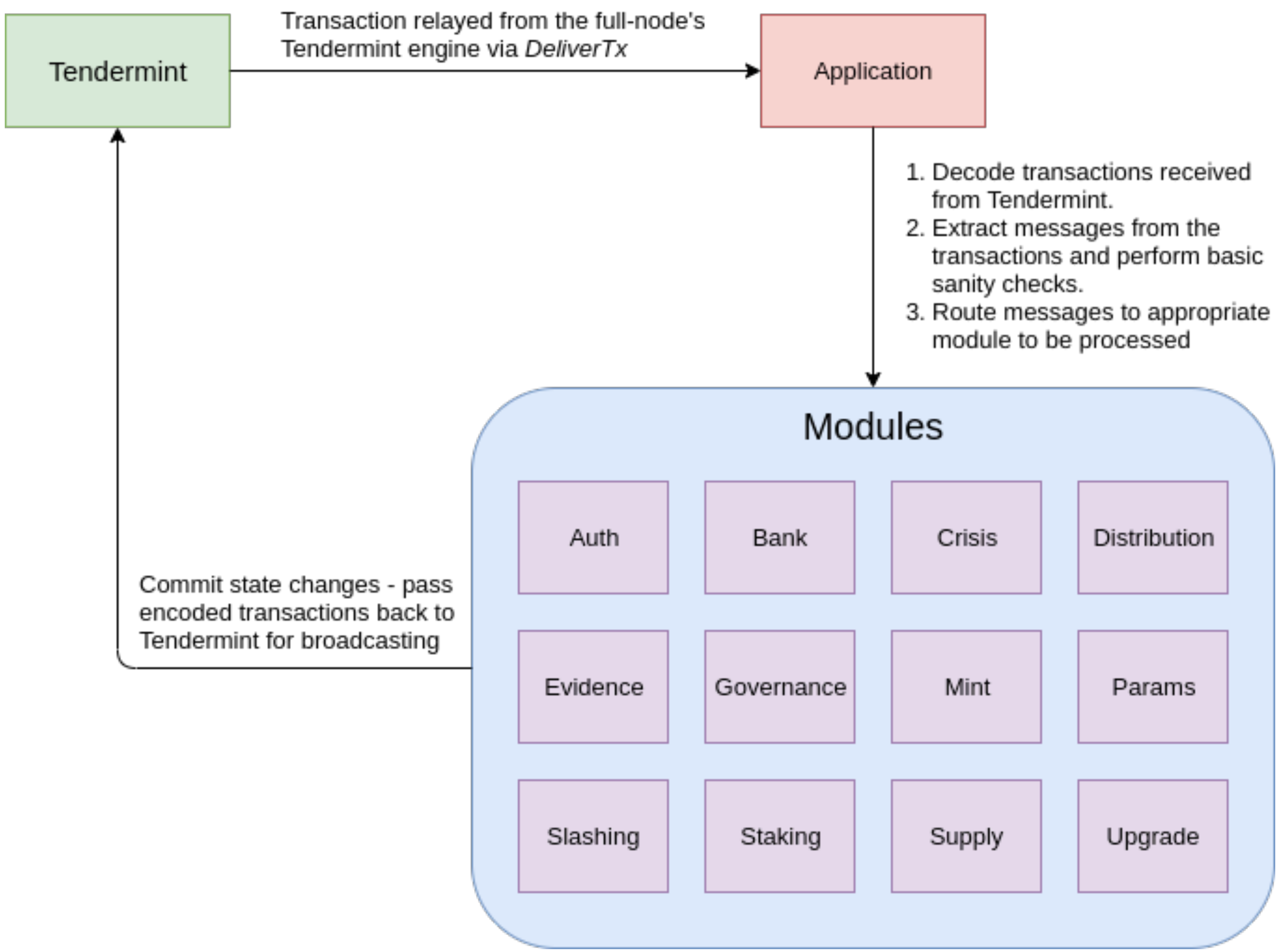
(source)

Cosmos SDK (Go)



(source)

Cosmos SDK: modules and keepers



Cosmos SDK: modules and keepers

```
x/{module_name}
├── client
│   ├── cli
│   │   ├── query.go
│   │   └── tx.go
│   └── testutil
│       ├── cli_test.go
│       └── suite.go
├── exported
│   └── exported.go
├── keeper
│   ├── genesis.go
│   ├── grpc_query.go
│   ├── hooks.go
│   ├── invariants.go
│   ├── keeper.go
│   ├── keys.go
│   ├── msg_server.go
│   └── querier.go
├── module
│   └── module.go
│   └── abci.go
├── simulation
│   ├── decoder.go
│   ├── genesis.go
│   ├── operations.go
│   └── params.go
├── {module_name}.pb.go
├── autocli.go
├── codec.go
├── errors.go
├── events.go
├── events.pb.go
├── expected_keepers.go
├── genesis.go
├── genesis.pb.go
├── keys.go
├── msgs.go
├── params.go
├── query.pb.go
├── tx.pb.go
└── README.md
```

```
├── keeper
│   ├── genesis.go
│   ├── grpc_query.go
│   ├── hooks.go
│   ├── invariants.go
│   ├── keeper.go
│   ├── keys.go
│   ├── msg_server.go
│   └── querier.go
```

```go
func (k Keeper) AppendStep(ctx sdk.Context, step types.Step) uint64 {
    count := k.GetStepCount(ctx)
    step.StepId = count

    store := prefix.NewStore(ctx.KVStore(k.storeKey), types.KeyPrefix(types.StepKey))
    appendedValue := k.cdc.MustMarshal(&step)
    store.Set(GetStepIDBytes(step.StepId), appendedValue)
    k.SetStepCount(ctx, count+1)
    return count
}

func (k Keeper) GetStep(ctx sdk.Context, id uint64) (val types.Step, found bool) {
    store := prefix.NewStore(ctx.KVStore(k.storeKey), types.KeyPrefix(types.StepKey))
    b := store.Get(GetStepIDBytes(id))
    if b == nil {
        return val, false
    }
    k.cdc.MustUnmarshal(b, &val)
    return val, true
}
```

Explicit read/write from/to the blockchain (key-value store)

Vulnerabilities in Cosmos code

Vulnerabilities

Not So Smart Contract	Description
Incorrect signers	Broken access controls due to incorrect signers validation
Non-determinism	Consensus failure because of non-determinism
Not prioritized messages	Risks arising from usage of not prioritized message types
Slow ABCI methods	Consensus failure because of slow ABCI methods
ABCI methods panic	Chain halt due to panics in ABCI methods
Broken bookkeeping	Exploit mismatch between different modules' views on balances
Rounding errors	Bugs related to imprecision of finite precision arithmetic
Unregistered message handler	Broken functionality because of unregistered msg handler
Missing error handler	Missing error handling leads to successful execution of a transaction that should have failed

(Source: Crytic)

Vulnerabilities in Cosmos code: non-determinism

Non-determinism

Non-determinism in conensus-relevant code will cause the blockchain to halt. There are quite a few sources of non-determinism, some of which are specific to the Go language:

- `range` iterations over an unordered map or other operations involving unordered structures
- Implementation (platform) dependent types like `int` or `filepath.Ext`
- goroutines and `select` statement
- Memory addresses
- Floating point arithmetic operations
- Randomness (may be problematic even with a constant seed)
- Local time and timezones
- Packages like `unsafe`, `reflect`, and `runtime`

Smart contract languages: summary

	Assets represented as…	Strengths	Issues
Solidity	Integers: mapping (address => uint)	Straightforward imperative code. Accessible, familiar.	Base language too error-prone. Use libraries (OpenZeppelin). Reentrancy.
Move	Resource types: struct Coin has key {…}	Reference safety, resource types.	Global storage model requires different way of structuring code. “Immovables”.
Zoe	Payment and Purse objects: mint.mintPayment(amount)	Reuse subset of a general-purpose language (JavaScript).	Must carefully follow coding idioms, no language support for asset management. Complex framework.
Cosmos	Coin objects managed by a dedicated Bank module	Reuse general-purpose language (Go).	Explicit save/restore of blockchain state. Avoid non-determinism. Complex framework.

PL Design & Smart Contracts

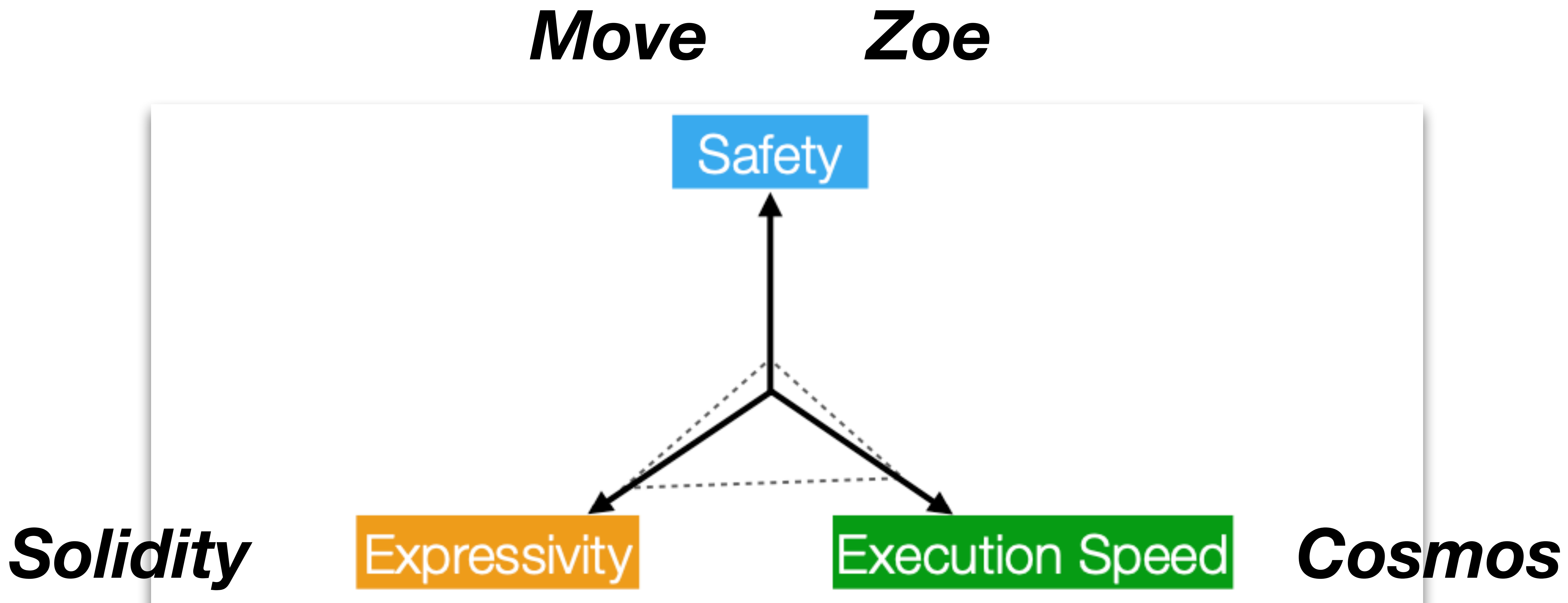
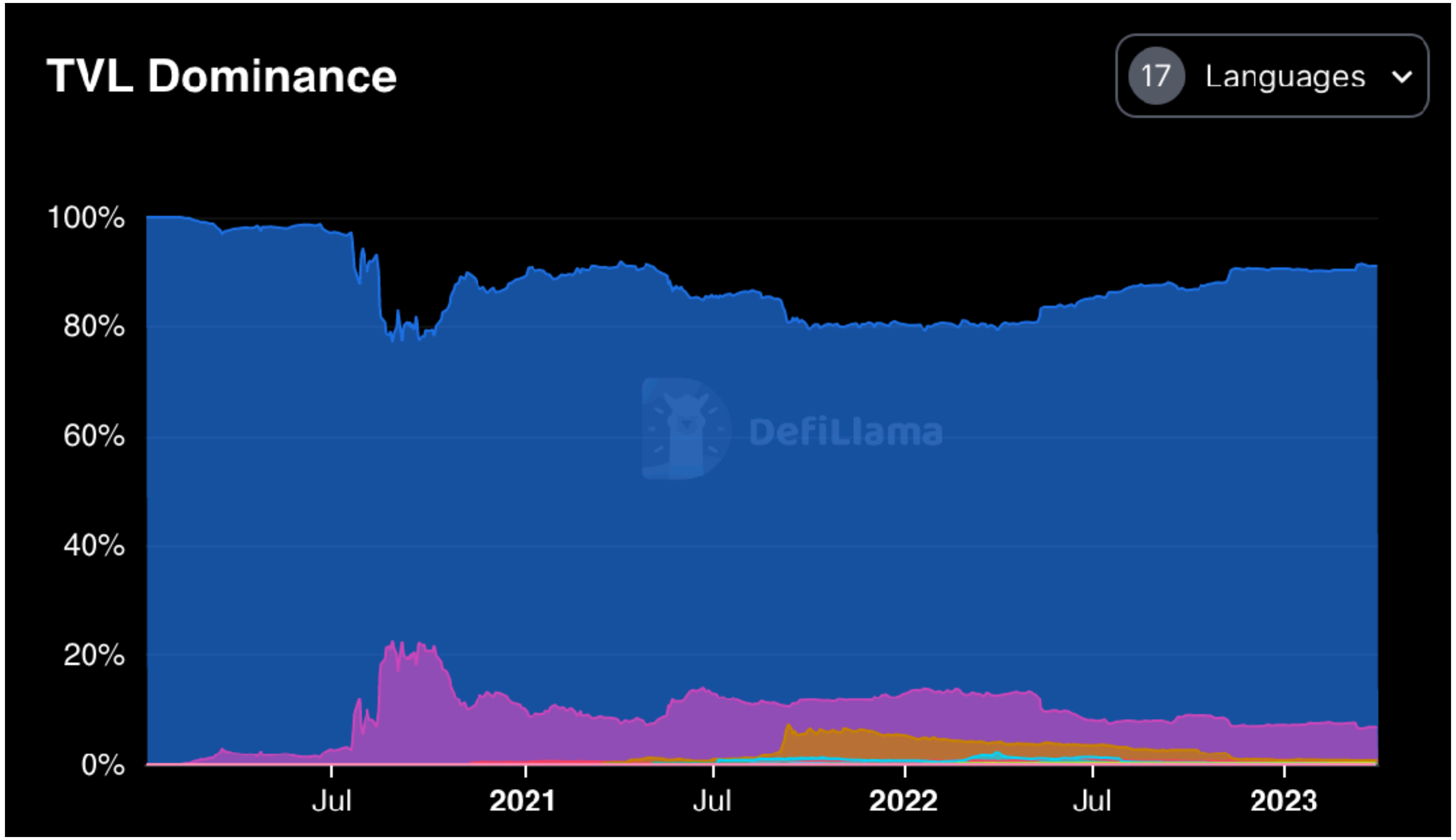


Figure 1.1: Language Design Trade-off. The dashed line shows EVM's design choices.

Reality check: what actually gets used

TVL = Total value locked in smart contract programs



31 mrt 2023
- Solidity 91.05%
- Vyper 6.91%
- Rust 0.7%
- Cairo 0.49%
- Bitcoin Script 0.24%
- Haskell 0.19%
- Ride 0.11%
- C++ 0.11%
- C# 0.1%
- Java 0.04%
- Others 0.07%

(Source: Defillama, april 2023)

KU LEUVEN

DistriNet

Exploring the design space of smart contract languages

Tom Van Cutsem

Thanks for listening!

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