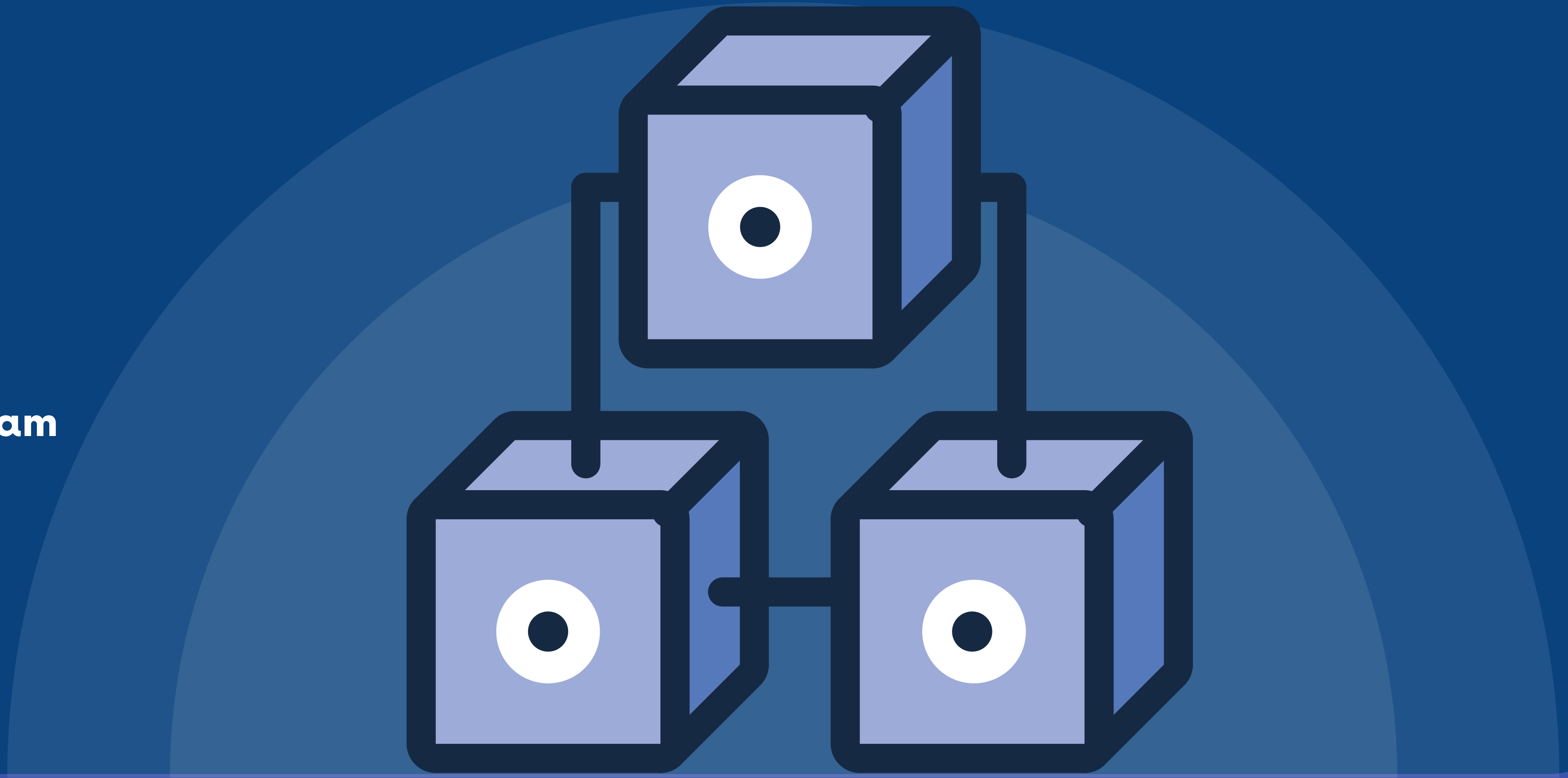
REFAEL YZGEAV

Proof-carrying Smart Contracts

Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, Vikram Saraph,
Eric Koskinen
https://paulgazzillo.com/papers/wtsc18.pdf



Content Outline

Topics for discussion

What is Proof Carrying Code?

Proof-carrying code is a mechanism for ensuring the safe behavior of a program.

In PCC the code consumer (e.g. host) verifies that code provided by an untrusted code producer by executing the saftey policy provided by the code producer.
These rules (**safety policy**), are sufficient guarantees for safe behavior of programs.

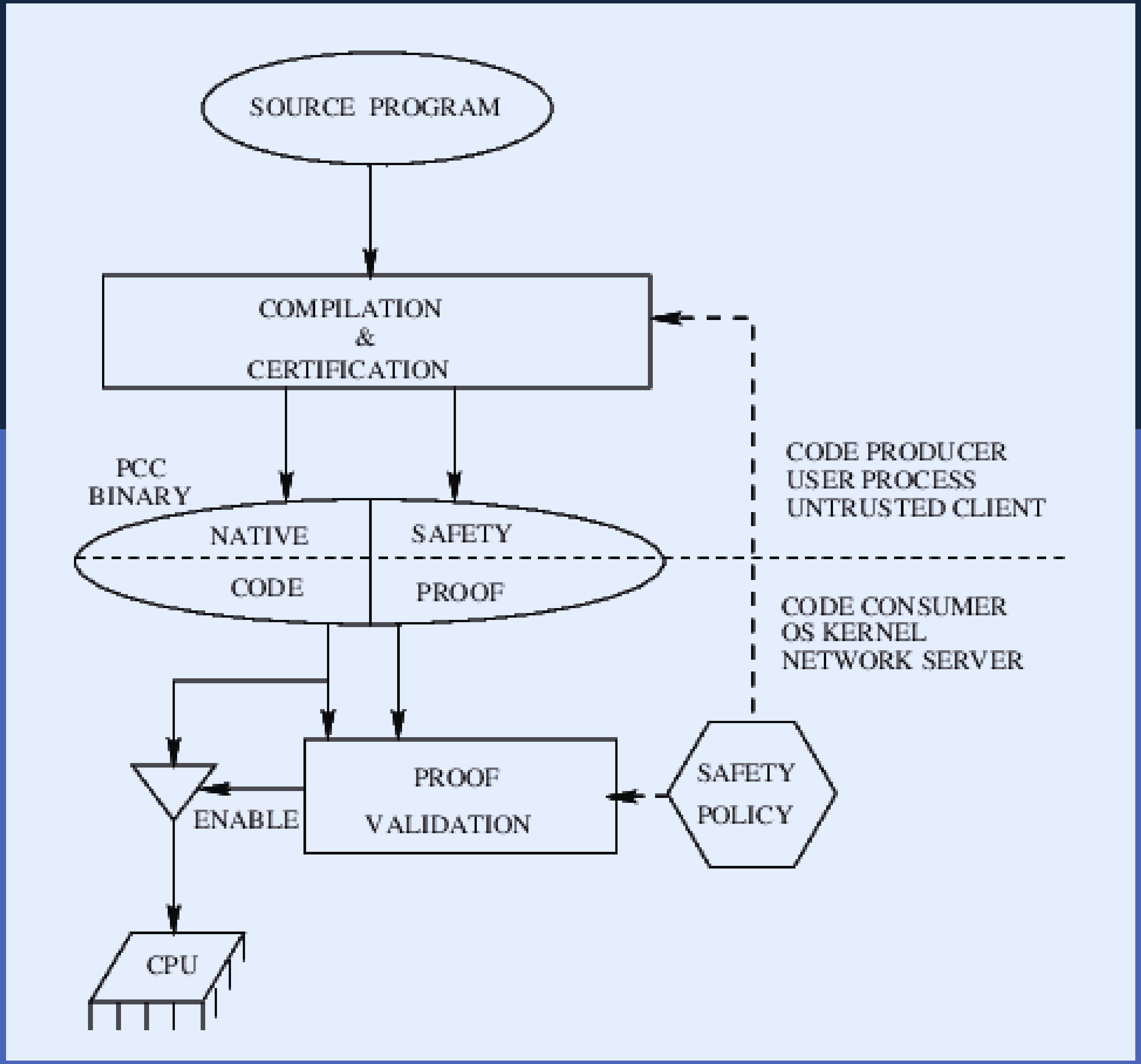
The key idea behind proof-carrying code is that the code producer is required to create a formal safety proof that attests to the fact that the code respects the defined safety policy. Then, the code consumer is able to use a simple and fast proof validator to check, with certainty, that the proof is valid and hence the foreign code is safe to execute.

IMPORTANCE OF THE SAFETY POLICY

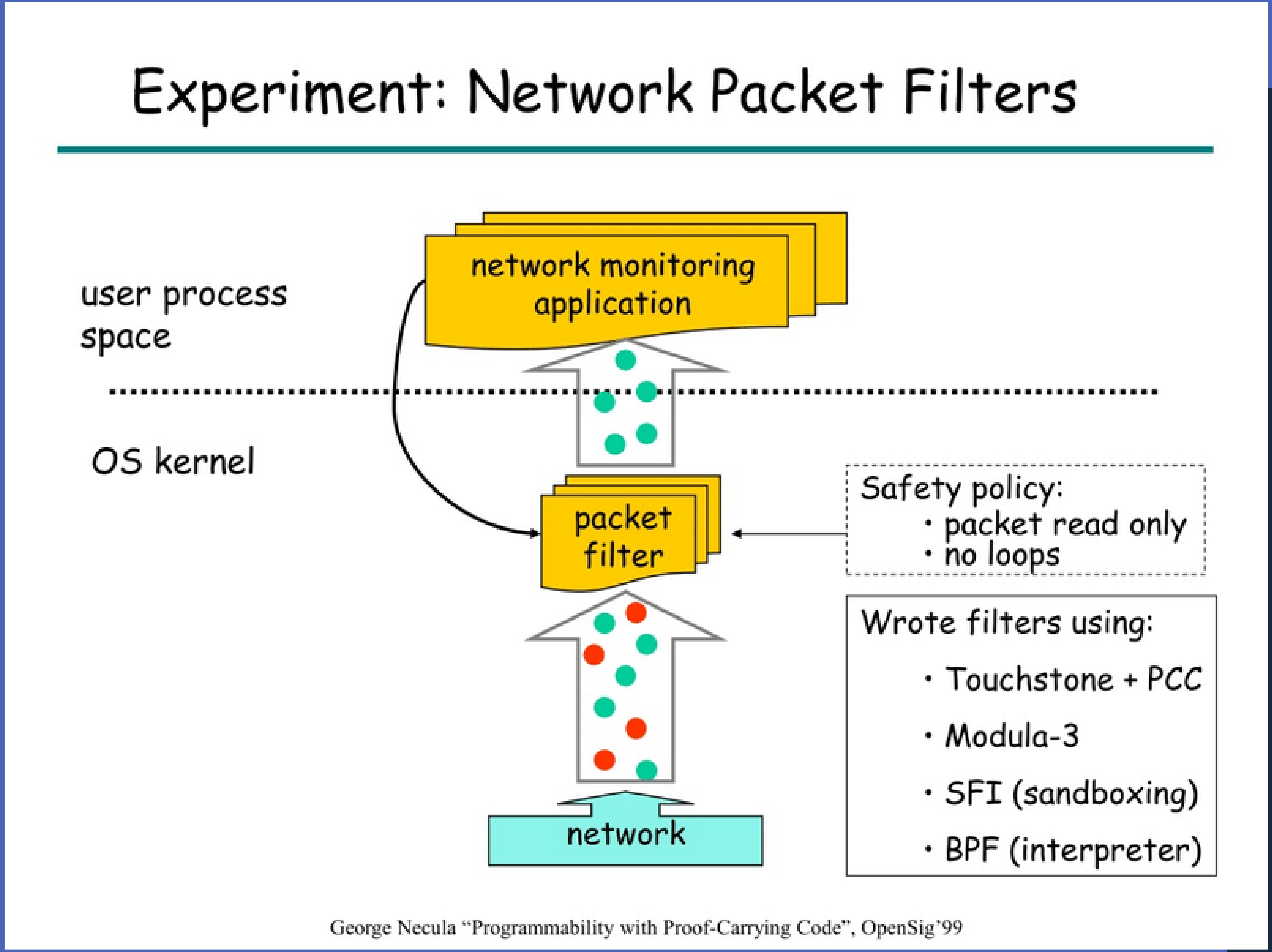
What does the safety policy check?

- Correctness (functional)
- Type correctness
- Memory access & array bound
- Resource-consumption

Example of Proof Carrying code Implementation



Real PCC Example



Experiment: Network Packet Filters

Safety policy:
- packet read only
- no loops

Wrote filters using:
- Touchstone + PCC
- Modula-3
- SFI (sandboxing)
- BPF (interpreter)

George Necula "Programmability with Proof-Carrying Code", OpenSig'99

Packet filter program

This program is written in Machine code, running in kernel mode in order to filter packets.
This program could contain malicious code that writes to the kernel data structure.

The Code producer will create a safety policy attached to his software.
The kernel publishes a security policy specifying properties that any packet filter must obey.
The code consumer will validate the saftey policy to ensure safe usages of the software.

Intro to Smart Contracts

Like a legal contract

Smart Contracts are a set of promises specified in a digital format, following certain rules and data given.
They could operate alone, or using outside resources like the internet.(Oracles)

Another Type of Account

Smart Contracts are another type of Ethereum account, they hold a balance and can send transactions just like an user account.
Opposite to a user account they operate according to the terms defined in the contract.

Implementation

Smart Contracts are written in high level languages such as Solidity, Vyper, Yul and more.
The code is then compiled to EVM bytecode and deployed to the Ethereum blockchain.

Key Benefites of Smart Contracts

Savings

no need for a middleman to take another part of the pie

Speed, efficiency and accuracy

Once a contract is deploy the contract will be exectued immdetily without any beauracy

Transparency

The terms and conditions of these contracts are fully visible and accessible to all relevant parties

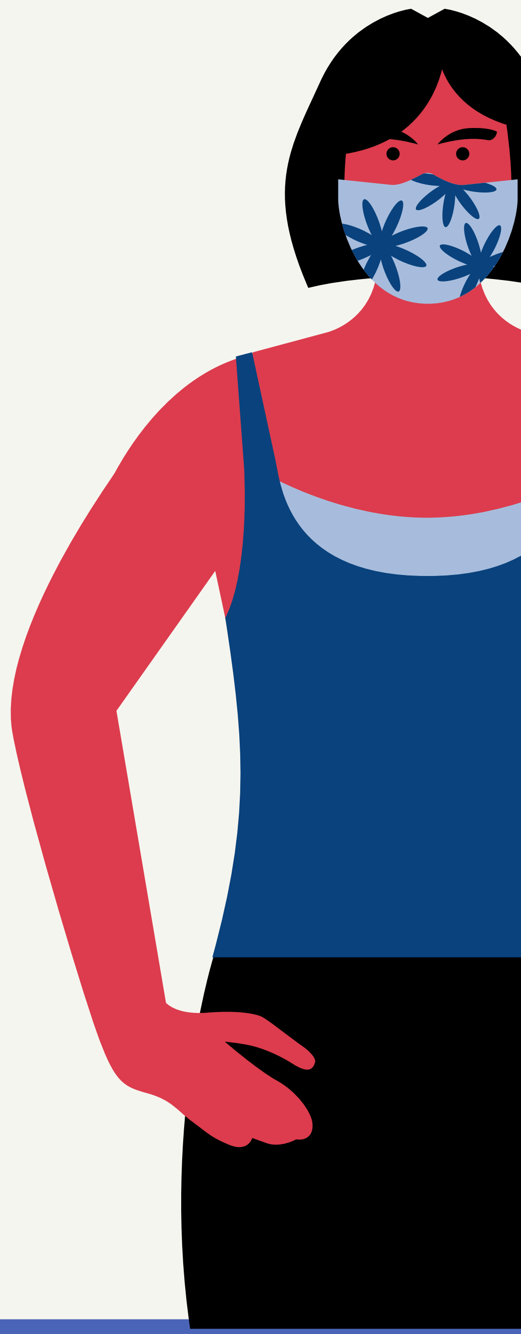
Smart Contracts Security Issues

Smart Contracts are essentially another type of software, therefore they are prone to bugs and vulnerabilities.

In addition, the environment the smart contracts are executed on is the EVM, and it also posses som several vulnerabilities.

Software Related Vulnerabilities

- Incorrect naming
- Incorrect calculation of the output token amount
- Indirect execution of unknown code
- Improper Behavioral Workflow
- Incorrect initialization
- Improper handling of Errors
- Improper Access control



The ERC-20 Token Standard

What is it?

A Token is a representation of **something** in the blockchain. This could be lottery ticket, fiat current(USD), share of a company etc..

Using this representation enables us to make smart contracts intractable with all tokens.
A Token contract is simply another type of Ethereum contract.
The community developed a standard for smart contracts documenting rules a smart contract should follow and how one contract can interoperate with other contracts.

Well known tokens are ERC20, ERC721, ERC777.

The ERC-20 Token standard is a contract standard for fungible token (tokens that are equivalent and interchangeable).
This is the main token standard for etherum.
This token standard consists of a set of function and events.

ERC-20 Token Interface

Functions

- totalSupply()
- balanceOf(account)
- transfer(recipient, amount)
- allowance(owner, spender)
- approve(spender, amount)
- transferFrom(sender, recipient, amount)

Events

- Transfer(from, to, value)
- Approval(owner, spender, value)

ERC-20 Race Condition \ Front-Running

```
function transferFrom(address _from, address
  _to, uint256 _value)    returns (bool success)

function approve(address _spender, uint256
        _value) returns (bool success)
```

Attack

Possible attack vector on standard ERC20 Ethereum Token API was discovered in mid 2019, this attack effects all contracts that implemented the ERC20 standard.

Root Cause

Two main method were responsible for this attack. transferFrom & approve method.

Example Scenario

1. Alice allows Bob to transfer N tokens to Bob
2. Alice regrets and allows to send to Bob only M tokens
3. Bob calls the trasferFrom method with the value N. (Before alices second call was mined)
4. Bob calls again the transferFrom method this time with M value.

Analysis

This attack is possible because approve method overrides the current allowance regardless of weather spender already used it or not.

Implementing PCSC

What is it?

As seen earlier in proof carrying code the code producer gives the software to check that the code is valid.
And the code consumer is the one running the validation.
But in PCSC the code producer is the contract writer and the code consumer are all other people in the network.
In PCSC the code producer is providing the verification that the contract is safe.
This method uses the consensus mechanism of the blockchain in order to ensure validation of the updated smart contract. Any update to the fields needs to be provided by a proof that the update is valid

Our PCSC involves two parts:
- The **parent part** of pcsc includes the smart contract internal state, api methods and formal specification.
- **Child part** of the PCSC includes the source code for all API methods and a proof that the implementation satisfies the spec.

What is the PIMPL Paradigm?

"Point to implementation" is a C++ programming technique that removes the implementation details of a class from its object class by placing the implementation details in a separate class.
It is about hiding the actual class implementation behind the pointer to forward the declared implementation class, so all the implementation's heavy details and dependencies dont pollute the class interface and it's header file.

PCSC Diagram

Parent Contract

General Components:
- Functions specification
- State Management
- Addresses to child contracts

Each function:

```
api_transfer(uint256 value, addr from) : bool
{
child_ptr.transfer()
}
```

Child Contract

Implements the actual functions details and calculation.
Provides a verification proof for the child contract.

```
api_transfer(uint256 value, addr from) : bool
{
 ....code of transfer
}
```

Verification proof of api_transfer

Verification process of PCSC

A multi-step process

Stage 1

The code producer generates a proof that the child contract satisfies the specification.

Stage 2

The producer issues an update operation to the blockchain

Stage 3

The miner validates the proof against the code and produces a new block containing the safe update

Stage 4

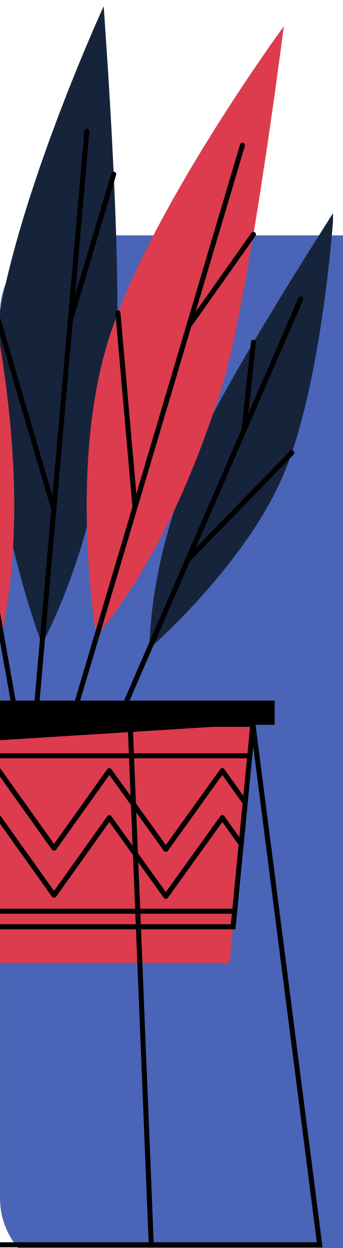
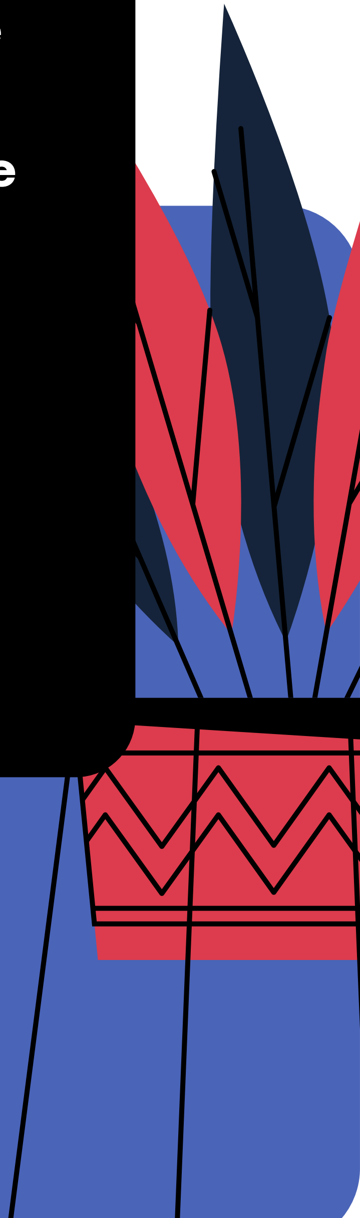
The rest of the blockchain rerun and validate the SAFEUPDATE operation

PCSC In Detail

We assume the parent contract is not valid until provided with an initial child contract (this is because the child contract contains the validation proof). There is nothing preventing the parent contract from having a multiple child contracts, so long as each child address must be modified only with the safe update command to ensure a proof is attached to the contract.

The child contract typically will not have any state, it will instead operate on the state of the parent, if some caching is needed it must be guaranteed to not affect the global state of the parent. The parent contract will use a DELEGATECALL to the child contract to make sure the child contract operates on the parent contract state.

Changing the specification is possible only for making the spec more strict. This means that changes are acceptable as long as the new spec implies the previous one as other child contracts are depend on it.

Updating PCSC

Minor Upgrades

- A new child contract is permitted as long as the safety policy is preserved. It is enforced by the proof verification performed by SAFEUPDATE.
- This enables upgrades to the child contract & lowers obstacles to development.

Major Upgrades

- Updates that alter the safety policy require a consensus among contract participants.
- A new safety policy need to be provided.
- The SAFEUPDATE function will verify the new child contract against the new safety policy, the new child contract will need to generate a proof against the new safety policy.

ERC20 As PCSC - Parent

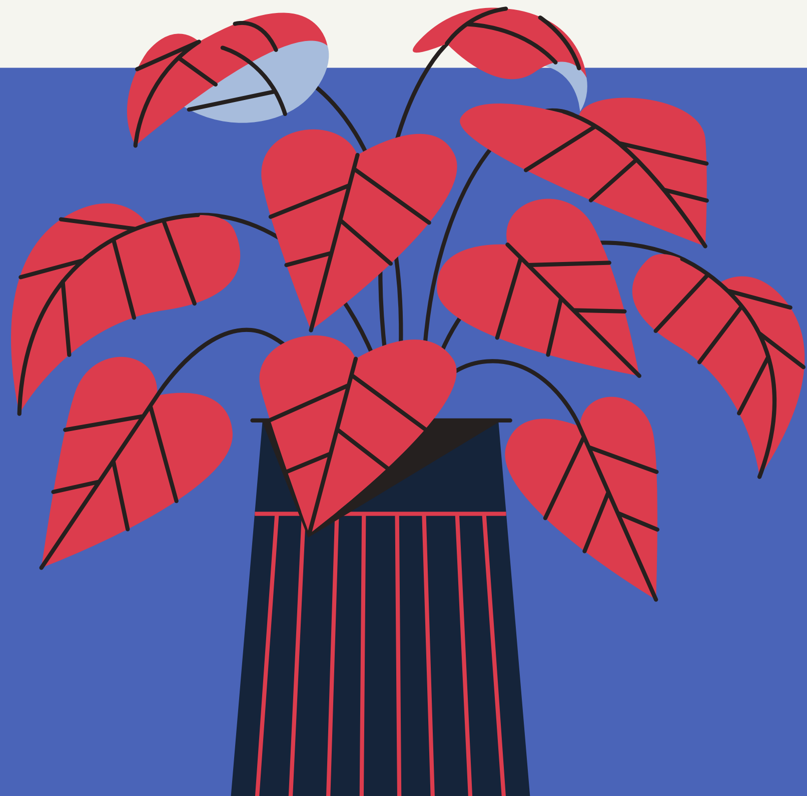
As discussed earlier the parent part will include: API specification and State.
In the ERC20 example the state of the parent will be:

- Balance: Addr -> N

- Allowed: Addr -> (Addr -> N)

- Child_ptr: Addr

- balance is a mapping from addresses to tokens, respresented as natural numbers
- allowed is a mapping from addresses to address-token mappings
- child_ptr is the point to the child part of the PCSC which will contain the implementation



What is the Hoare Logic?

is a formal system with a set of logical rules for reasoning rigorously about the correctness of computer programs. It was proposed in 1969 by the British computer scientist and logician Tony Hoare, and subsequently refined by Hoare and other researchers. The original ideas were seeded by the work of Robert W. Floyd

ERC20 PCSC Parent implementation

api_transfer(uint256 value, addr from) : bool

Preconditions:
I ∧ Σa ‘balance(a) = Σa balance(a)
∧ ‘allowed(from)(me) ≥value
Postconditions:
⇒allowed = ‘allowed[from,me
7→‘allowed(from)(me) −value] ∧rv =
true
∧ ‘allowed(from)(me) ≤value
⇒allowed = ‘allowed ∧rv = false

api_allowance(addr whom) : uint256

Preconditions:
{I ∧ Σa ‘balance(a) = Σa balance(a) ∧
ρme(whom) = allowed(me)(whom)}

api_approve(uint256 value, addr whom) : bool

Preconditions:
I ∧ Σa ‘balance(a) = Σa balance(a)
∧ (‘allowed(me)(whom) =
ρme(whom)
Postconditions:
⇒allowed = ‘allowed[me,whom
7→value] ∧rv = true)
∧ (‘allowed(me)(whom) 6=
ρme(whom) ⇒allowed = ‘allowed ∧rv
= false)

Child Part implementat ion of PCSC

Implementation

The implementation of each API method is houses in the child part of PCSC such that, if the implementation satisfy the spec provided earlier(in the parent part) then the child will be installed and accessed via child_ptr.transfer()

How can we be 100% that the implementation works correctly?

Using the Floyd-Haore style of pre post specifications we can verify that the implementation using the verification conditions.

PCSC allows us from buggy implementation like the original approve from being accepted to the blockchain.
The buggy version of approve in the ERC20 token standard isn't approval in a specification.

What's next?

The authors of the paper intend to make a verification tool for PCSC building on existing work for verification in solidity and EVM.

The authors state that indeed the work is still early in the process and there is a lot to be done.
Future goals include formally modeling proof carrying smart contracts and creating an implementation as an extension of the Etherum blockchain and virtual machine.

In addition they intend to investigate how consensus integrates with these proofs .
For generating and validating proofs, they plan to use off-the-shelf tools, such as Why3

Thank you! For listening