Towards Formal Verification of Solidity Smart Contracts Using PAT

Christopher Skorka, Lee Goymer, Hadrien Bride, Zhé Hóu, and
Jin Song Dong
Institute for Integrated and Intelligent Systems (IIIS),
Griffith University

Background

A smart contract is a compute protocol intended to facilitate, verify, or enforce the negotiation or performance of a contract.

Popular platform: Ethereum.

Popular language: Solidity.

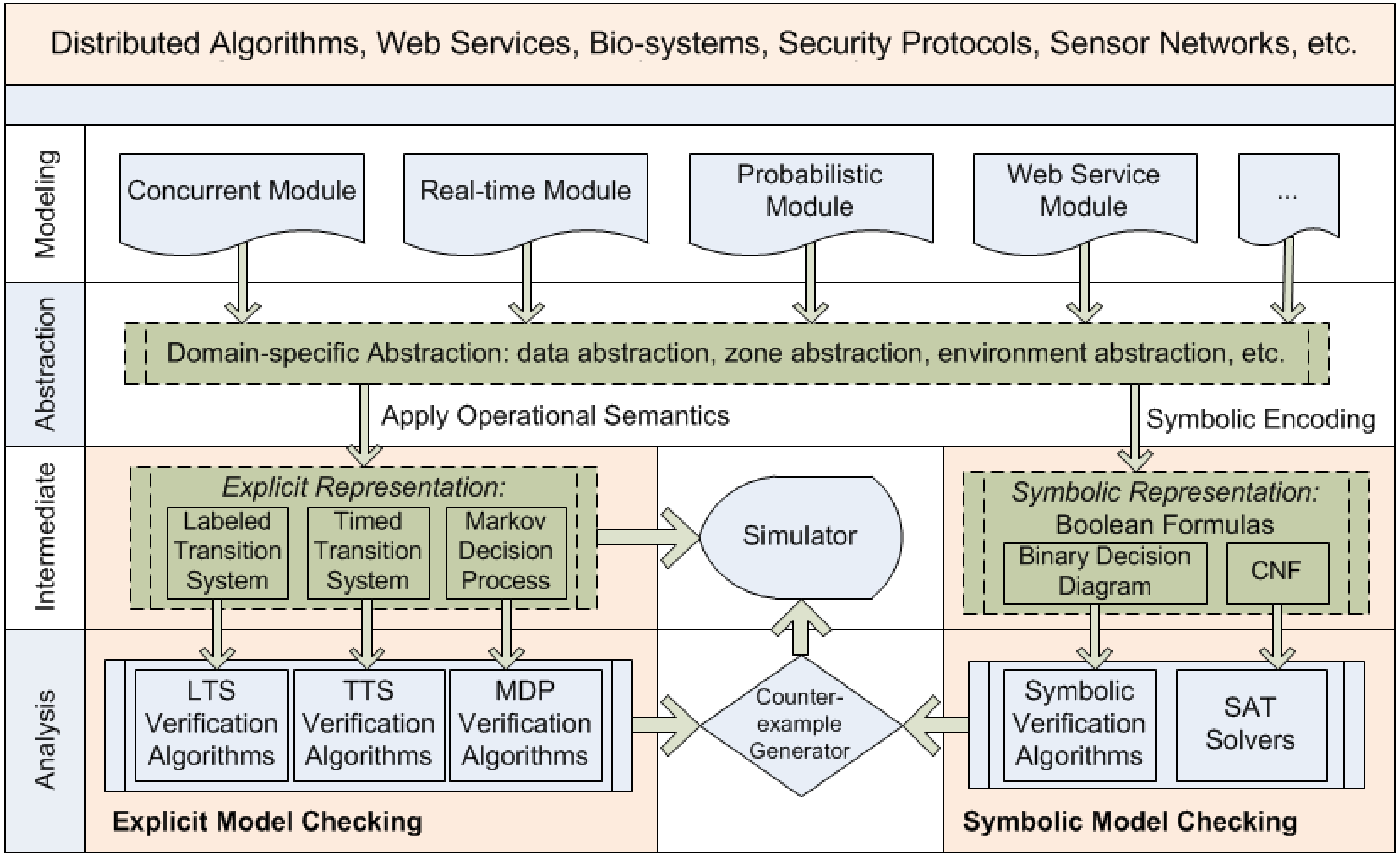
Based on the blockchain technology.

Used with the cryptocurrency token ether.

Features of Smart Contracts

- Self-executing and self-enforcing
- Decentralised control
- Reduced costs associated with contracting
- Blockchain technology solves the "double-spending" problem
- Open networks
  - Everyone can join
  - Easy to attract criminals
- Immutable
  - Once a smart contract is in places, it cannot be tampered
  - If a smart contract has vulnerabilities, it's hard to fix
  - 50 million USD gone in the DAO attack
  - Ethereum sometimes has to perform a "hard fork" to reappropriate the stolen funds
- Possible to mix trusted code with untrusted code

Improving Security: The PAT Approach



Improving Security: The PAT Approach

01
Find core features in Solidity

02
Model Solidity contracts in PAT

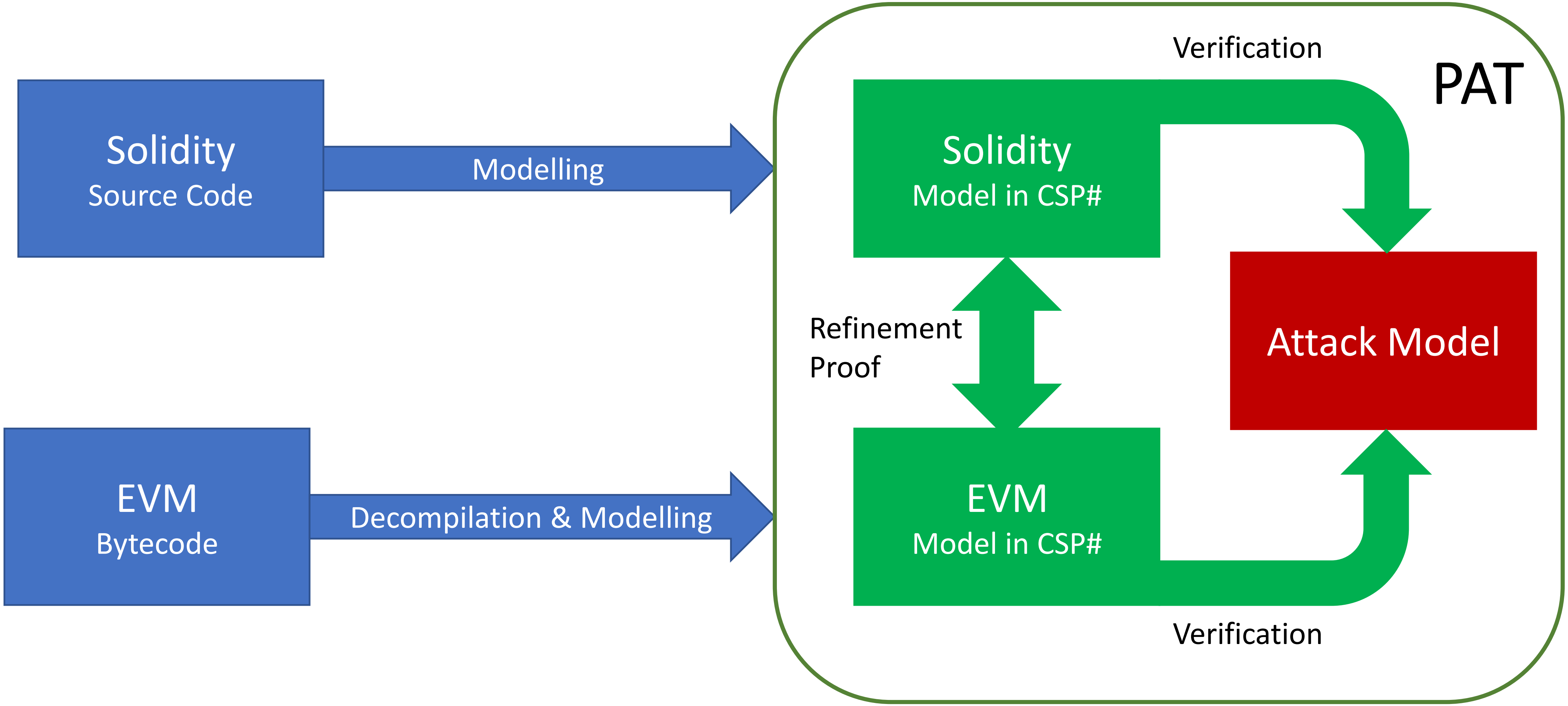
03
Survey security problems

04
Build attack models

05
Verify Solidity contracts against attack models

Improving Security: The PAT Approach

Many smart contracts don’t reveal source code



Solidity Vulnerabilities

- Timestamp can be modified at runtime
- Transaction values can be changed last minute
- Re-entrance
- Stack overflow
- Integer overflow
- Exception handling
- Delegate calls can be used to execute new unknown code
- Sending money from contracts automatically is prone to vulnerabilities

Exceptions in Solidity

- Out of stack (> 1024)
- Out of Gas
- Out of (array) index
- Code not found (call to undefined external function)
- Called function throws exceptions
- New contract not finished properly during creation
- /0 (Div by zero), %0 (Modulo by zero)
- Ether paid to a function without ‘payable’
- Received ether via a public accessor function
- ‘Throw’ for any custom reason
- Shift by a negative amount
- Convert negative or too large values into enum types
- External function call to a contract with no code
- .transfer() fail

Solidity Properties in Verification

- Different versions
- Stack based – 1024 levels (top 16 accessible) – stack overflow exception
- Logical evaluations apply short circuits
- ContractAddress.send() causes the contracts fall back functions to
- Call, callcode and calldelegate break type-safety and should be avoided
- Literal division used to be truncated to integers but now isn’t
- Function calls to other contracts cannot return anything but whether they finished or crashed
- ‘var’ variable declaration will use the simplest type possible for the given expression
- Accessing the hash of a block more than 256 blocks before will return 0
- Exceptions don’t bubble through call(), send(), callcode() and calldelegate() but instead return false
- Allows inline assembly code which has different behaviours altogether

From Solidity to CSP#

Solidity Code

```
function confirmPurchase()
inState(State.Created)
require(msg.value == 2 * value)
payable
{
   purchaseConfirmed();
   buyer = msg.sender;
   state = State.Locked;
}
```

CSP# Code (PAT)

```
ConfirmPurchase(msgsender, msgvalue) =
if(state == Created && msgvalue == 2 * value){
   confirmPurchase{
      purchaseConfirmed();
      buyer = msgsender;
      state = Locked;
   } -> StandBy()
};
```

Verification by Model Checking

We can then verify various properties of the Solidity code in PAT

- Deadlock free
- Functional correctness
  - Program reaches desired states
  - Program doesn’t reach “bad” states
- Stack/integer overlow?
- Object-oriented features?

Next steps:

- Automate the translation
- Cover more properties/vulnerabilities