IEEE SecDev | 2020

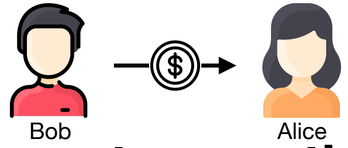
IEEE COMPUTER SOCIETY

Analysis of Blockchain Smart Contracts: Techniques and Insights

Shinhae Kim and Sukyoung Ryu

Basic Concept of a Blockchain

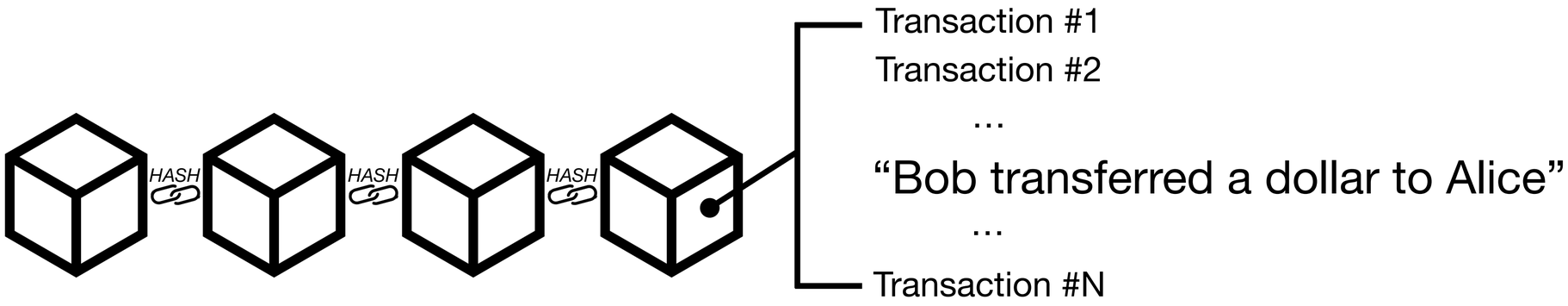
> A blockchain is a chain of **"blocks"**



a data structure that **stores transactions** among parties

> Each block has the **hash of previous block data**

- Modifying the block data is **computationally infeasible**

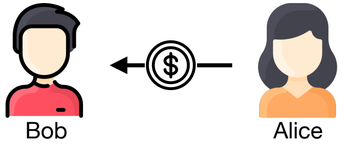


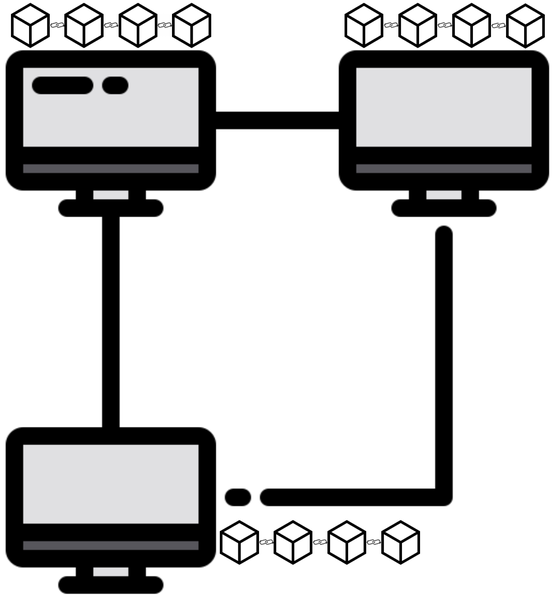
Basic Concept of a Blockchain

> A blockchain is built on top of a **P2P network**

- Each node has a **copy of the chain**

> When a **new transaction** is made..

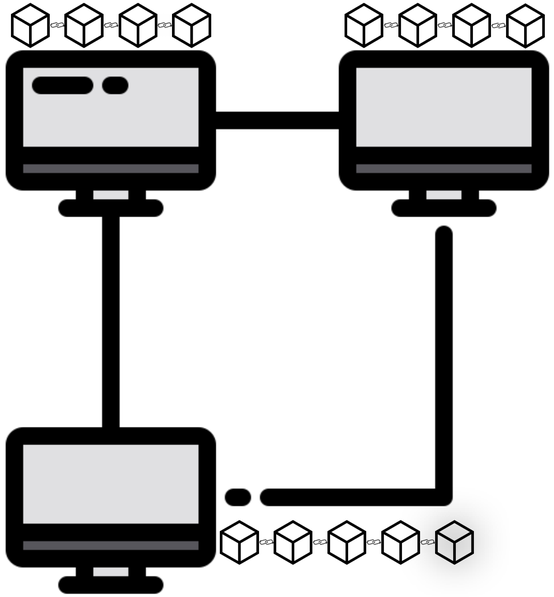




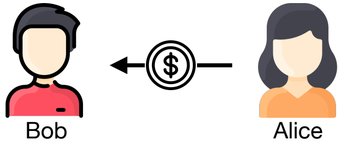
Basic Concept of a Blockchain

> A blockchain is built on top of a **P2P network**

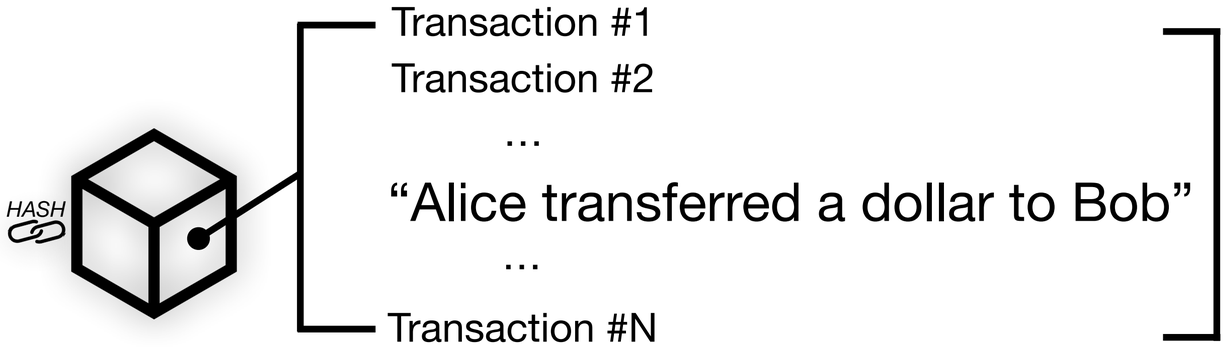
- Each node has a **copy of the chain**



> When a **new transaction** is made..



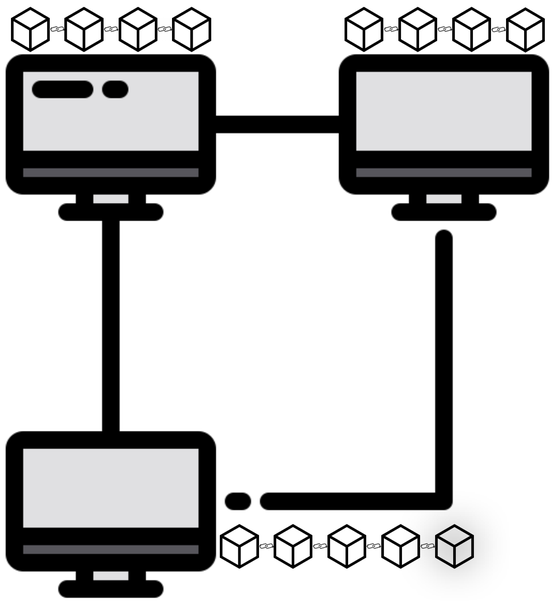
1. A node forms a block with the transaction



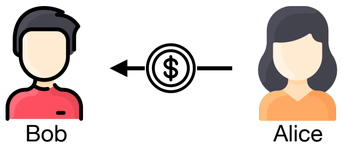
Basic Concept of a Blockchain

> A blockchain is built on top of a **P2P network**

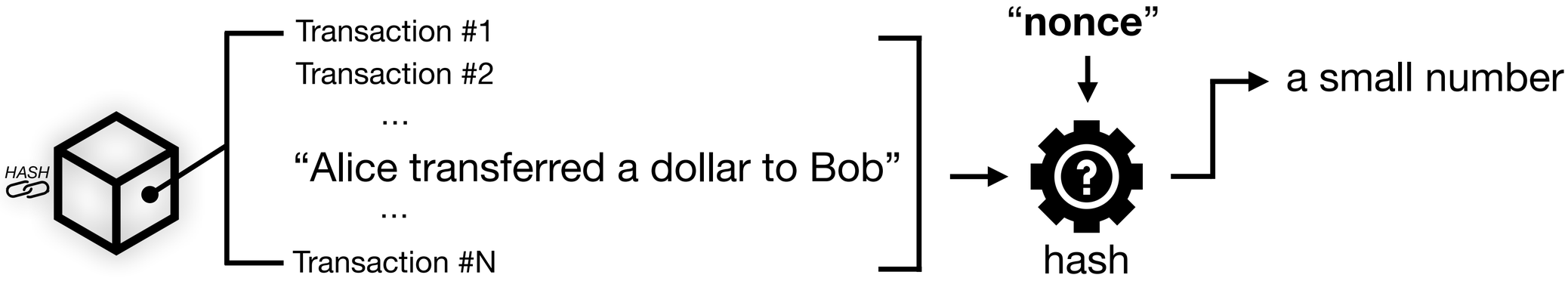
- Each node has a **copy of the chain**



> When a **new transaction** is made..



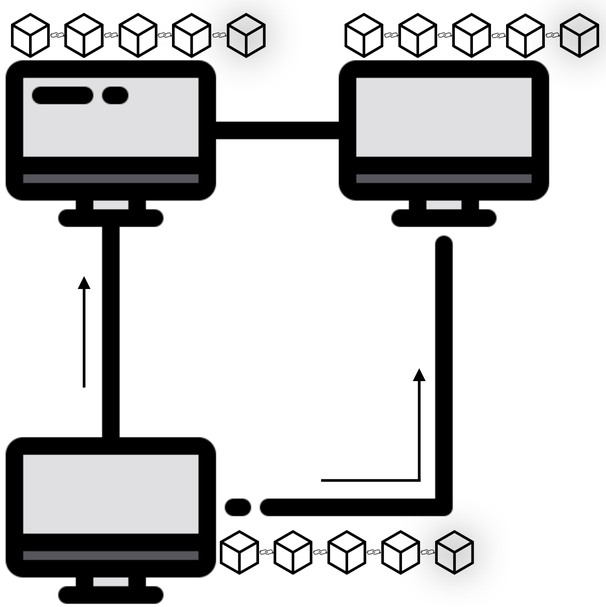
1. A node forms a block with the transaction



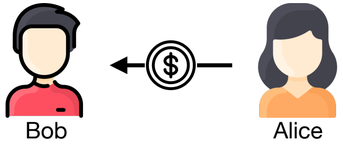
Basic Concept of a Blockchain

> A blockchain is built on top of a **P2P network**

- Each node has a **copy of the chain**



> When a **new transaction** is made..

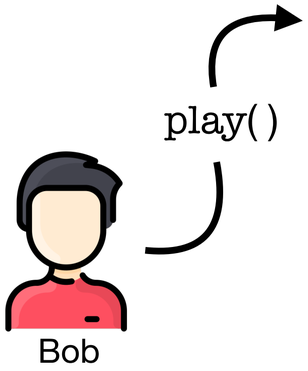


1. A node forms a block with the transaction
2. The node broadcasts the block
3. Other nodes append the block as the tail of the chain

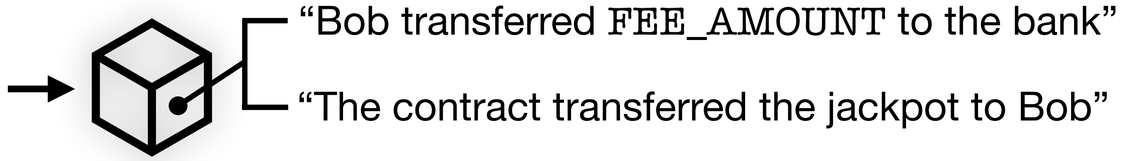
Basic Concept of a Smart Contract

> A smart contract is a **program deployed** on the blockchain

- When invoked, it **automatically makes transactions**

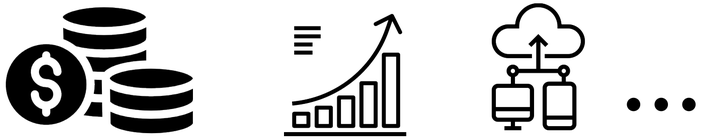



```
function play() payable {
    assert(msg.value == TICKET_AMOUNT);
    pot += msg.value;

    var random = uint(block.blockhash(block.number))
        + block.timestamp + block.difficulty
            + block.number;

    if (mulmod(random, 1, 2) == 0) {
        bank.transfer(FEE_AMOUNT);
        msg.sender.transfer(pot - FEE_AMOUNT);
        pot = 0;
    }
}
```



Smart Contract Vulnerabilities

> Smart contracts enabled blockchain **applicable to various domains**



> However, smart contracts have suffered from **vulnerabilities**

Vulnerability	Description
Arbitrary Writes to Storage	Any users can overwrite values in storage slots [31].
Block Variable Dependency	Contracts use block hash or timestamp to generate random values [18, 19].
Careless Handling of `tx.origin`	`tx.origin` returns the address of a transaction-initiated account [27].
`DELEGATECALL`	`DELEGATECALL` passes the caller context to its callee [19].
Inability to Transfer Ether	Contracts can receive Ether but do not provide a way for users to withdraw [23].
DoS: External Call Failures	A callee may throw an exception. For example, `send` assigns 2300 units of gas only [18].

Vulnerability	Description
Unexpected Fallback Function	Fallback executions take place in case of `send` or no matching function signature [18].
Integer Division	Dividing integers rounds down the quotient, always resulting in an integer [41].
Integer Overflow/Underflow	Arithmetic operations can result in overflow/underflow [113].
Private Information Leakage	Private fields do not guarantee confidentiality as transaction records are public [18].
Reentrancy	An external malicious callee can re-enter the body of its caller [18, 19, 20].
Usage of `call.value`	`call.value` forwards all remaining gas [41].
Return Value Bypassing	In case of exceptions, the built-in functions `send` and `call` return `false` [18, 19].

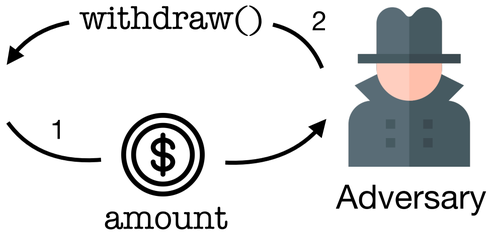
Vulnerability	Description
Transaction Order Dependency	Miners can control the order of deploying transactions in a block [18, 19, 20].
Transfer to an Orphan Address	Accounts lose Ether if they send it to a non-existing address [26].
Unauthorized Ether Withdrawal	Invoking Ether transfers is possible by any users [31].
Unauthorized `SELFDESTRUCT`	Any users can destroy a contract if no authorization exists for `SELFDESTRUCT` [31].
Function without Visibility	Functions without visibility declarations are public by default [19].
Unbounded Mass Operations	Iterating a loop can cause an out-of-gas exception if its iteration number is too high [32].

a list of “exploitable-vulnerabilities”

Smart Contract Vulnerabilities

Reentrancy	An external malicious callee can re-enter the body of its caller [18, 19, 20].
DELEGATECALL	DELEGATECALL passes the caller context to its callee [19].

```
mapping (address => uint) private balances;

function withdraw() public {
    uint amount = balances[msg.sender];
    (bool success, ) = msg.sender.call.value(amount)("");
    require(success);
    balances[msg.sender] = 0;
}
```



- The adversary stole **$50 million** from the DAO contract (June, 2016)
- Ethereum blockchain was forked into Ethereum and Ethereum Classic

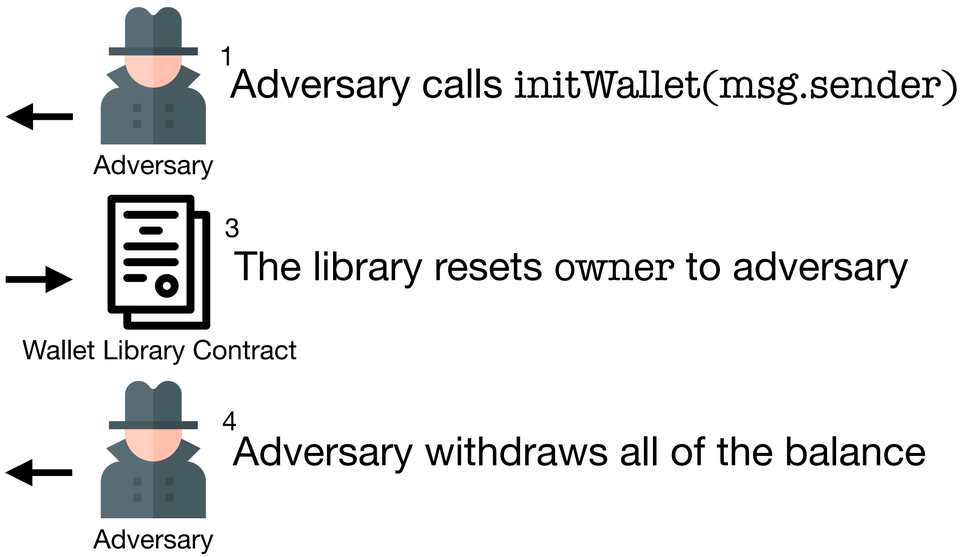
Smart Contract Vulnerabilities

Reentrancy	An external malicious callee can re-enter the body of its caller [18, 19, 20].

DELEGATECALL	DELEGATECALL passes the caller context to its callee [19].

```
contract Wallet {
    address _walletLibrary;
    address owner;

    function Wallet(address _owner) {
        _walletLibrary = ${hardcoded address};
        _walletLibrary.delegatecall(bytes4(sha3("initWallet(address)")), _owner);
    }
    function () payable {
        if (msg.value > 0)
            Deposit(msg.sender, msg.value);
        else if (msg.data.length > 0)
            _walletLibrary.delegatecall(msg.data);
    }
```

2 The wallet delegates the call to the library



- The adversary stole **$31 million** from the wallet contract (July, 2017)

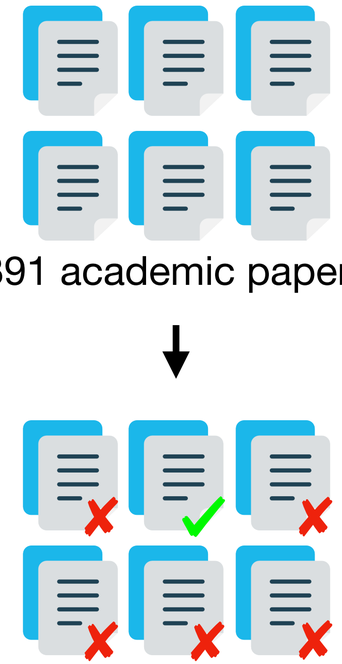
Contribution of this Research

> Such vulnerabilities led to massive research on **smart contract analysis**

And this paper is …

the first **comprehensive survey** of research on **smart contract analysis**

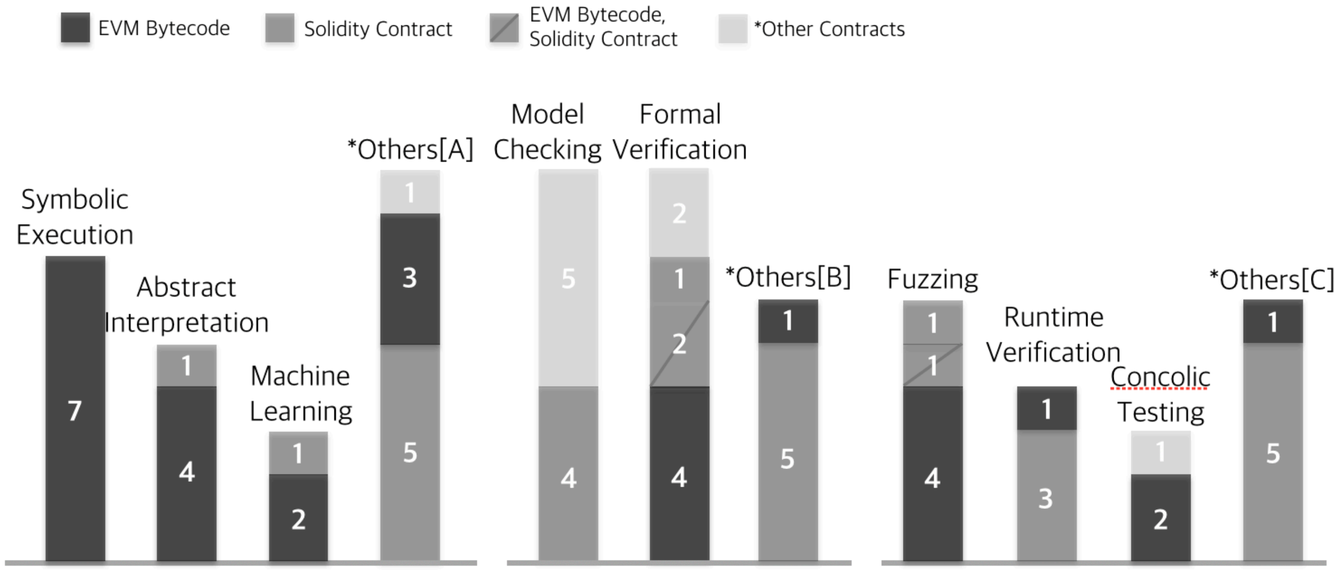
Overview of this Research



Overview of this Research

391 academic papers

67 “analysis-related” papers



(Sec. III-A) Static Analysis for Vulnerability Detection

(Sec. III-B) Static Analysis for Program Correctness

(Sec. III-C) Dynamic Analysis

Others[A]

XPath Query, Refinement Algorithm, AST Transformation, IR Transformation, Code Embedding, Model Checking, Program Synthesis, Taint Analysis, Simulation

Others[B]

Program Synthesis, Inductive Synthesis, Resource Analysis

Others[C]

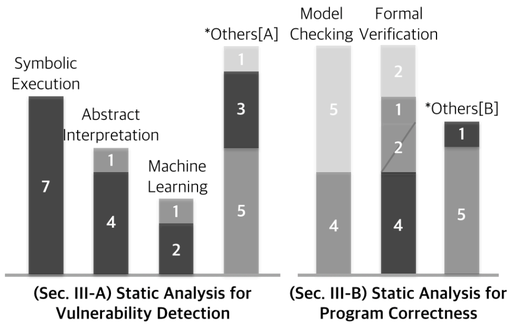
Mutation Testing, N-Version Programming, Specification Mining, Replay Testing, Emulation

Other Contracts

WASM Contract, BitML Contract, Bitcoin Contract, Sol Contract, Pseudocode, Oak Contract, Michelson Contract, Lolisa Contract

Research Trends of Smart Contract Analysis

Static analysis has been **more dominant** than dynamic analysis

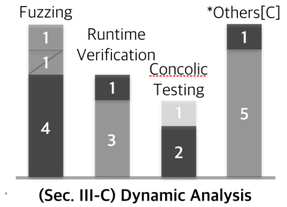


Static Analysis

> Smart contracts need to be audited **before deployment**

- Smart contracts **cannot be modified** once deployed

> Static analysis can reason over **all possible paths**



Dynamic Analysis

> Smart contracts are often **opaque** (i.e., no source code available)

- Bytecode **loses some information** such as variable types

> Dynamic analysis has no **false positives**

Research Trends of Smart Contract Analysis

The **most dominant techniques** are the following:

> **Symbolic execution**

- Smart contracts often have **small code sizes**

> **Model checking**

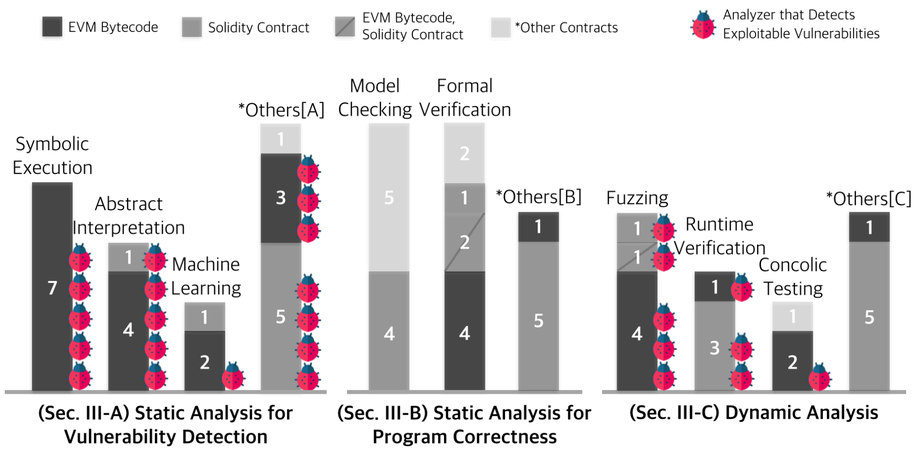
- Smart contract **execution environments** need to be considered as well
  (nuXmv, Corral, Uppaal, MCK model checkers)

> **Formal verification**

- The **highest level of correctness** can be guaranteed
  (Isabelle/HOL, F*, Coq, K framework)

Research Trends of Smart Contract Analysis

- A large portion of work is to **detect vulnerabilities**
  - > Vulnerabilities can directly result in **financial losses**



Vulnerability	Description	[22] Symbolic Execution	[23] Symbolic Execution	[24] Symbolic Execution	[25] Symbolic Execution	[26] Symbolic Execution	[31] Abstract Interpretation	[32] Abstract Interpretation	[33] Abstract Interpretation	[35] Abstract Interpretation	[37] Machine Learning	[38] Machine Learning	[41] Machine Learning	[43] Others[A]	[44] Others[A]	[45] Others[A]	[48] Others[A]	[49] Others[A]	[50] Others[A]	[87] Fuzzing	[88] Fuzzing	[89] Fuzzing	[91] Fuzzing	[92] Fuzzing	[94] Runtime Verification	[95] Runtime Verification	[98] Concolic Testing	[100] Concolic Testing
Arbitrary Writes to Storage	Any users can overwrite values in storage slots [31].	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	●
Block Variable Dependency	Contracts use block hash or timestamp to generate random values [18, 19].	●	○	○	○	○	○	○	○	○	●	○	●	○	○	○	○	●	○	●	○	○	○	●	○	○	○	○
Careless Handling of `tx.origin`	`tx.origin` returns the address of a transaction-initiated account [27].	○	○	○	○	○	○	○	●	○	●	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
`DELEGATECALL`	`DELEGATECALL` passes the caller context to its callee [19].	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	●	○	○	●	○	○	○	○	○	○	○	○
Inability to Transfer Ether	Contracts can receive Ether but do not provide a way for users to withdraw [23].	○	●	○	○	●	●	○	○	○	○	●	●	○	○	●	○	○	○	●	○	○	○	●	○	○	○	○
DoS: External Call Failures	A callee may throw an exception. For example, `send` assigns 2300 units of gas only [18].	○	○	○	○	○	○	●	○	○	●	○	●	○	○	○	○	○	○	●	○	○	○	●	○	●	○	○
Unexpected Fallback Function	Fallback executions take place in case of `send` or no matching function signature [18].	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○
Integer Division	Dividing integers rounds down the quotient, always resulting in an integer [41].	○	○	○	○	○	○	○	○	○	○	○	●	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○
Integer Overflow/Underflow	Arithmetic operations can result in overflow/underflow [113].	○	○	○	○	○	○	●	○	○	●	○	●	○	●	○	○	●	●	○	○	○	○	●	○	●	○	○
Private Information Leakage	Private fields do not guarantee confidentiality as transaction records are public [18].	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○
Reentrancy	An external malicious callee can re-enter the body of its caller [18, 19, 20].	●	○	○	○	○	●	○	●	●	●	○	●	○	○	●	●	●	○	●	○	○	●	●	●	●	●	○
Usage of `call.value`	`call.value` forwards all remaining gas [41].	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Return Value Bypassing	In case of exceptions, the built-in functions `send` and `call` return `false` [18, 19].	●	○	○	○	○	●	○	●	○	●	○	●	○	○	○	○	●	○	●	○	○	○	●	○	●	○	○
Transaction Order Dependency	Miners can control the order of deploying transactions in a block [18, 19, 20].	●	○	○	○	○	●	○	○	○	●	○	○	●	○	○	○	○	○	○	●	○	○	○	○	●	○	○
Transfer to an Orphan Address	Accounts lose Ether if they send it to a non-existing address [26].	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Unauthorized Ether Withdrawal	Invoking Ether transfers is possible by any users [31].	○	●	○	●	○	●	○	●	○	○	●	○	○	○	●	●	●	○	○	○	○	○	○	○	○	○	○
Unauthorized `SELFDESTRUCT`	Any users can destroy a contract if no authorization exists for `SELFDESTRUCT` [31].	○	●	○	●	●	○	○	●	○	○	●	○	○	○	●	●	●	○	○	○	○	○	○	○	○	○	○
Function without Visibility	Functions without visibility declarations are public by default [19].	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●
Unbounded Mass Operations	Iterating a loop can cause an out-of-gas exception if its iteration number is too high [32].	○	○	●	○	○	○	●	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○

Insights from the Survey

Open Challenges

> Ambiguity in **Contract Behaviors**

> Unstable **Language Semantics**

> Lack of Clear **Property Definition**

Research Directions

> New **Language Design**

> **Type-based Approaches**

> **Machine Learning** Techniques

> Rising **Platforms** and **Languages**

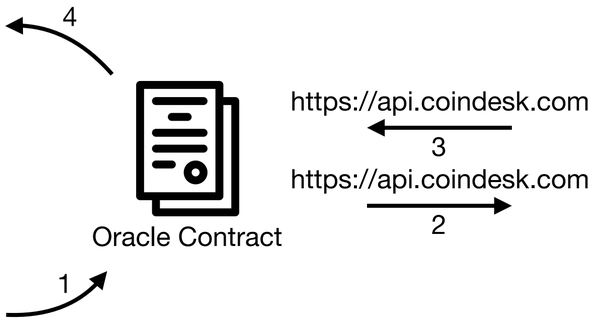
Ambiguity in Contract Behaviors

> Miners can control the **order of deploying transactions**

Vulnerability	Description
Transaction Order Dependency	Miners can control the order of deploying transactions in a block [18, 19, 20].

> Smart contracts often rely on **off-chain sources** through oracles

```
function callback(bytes32 key, string callbackData) external payable only_dAppBridge {
    usd_btc_rate = callbackData;
    bytes32 newkey = setURLTimeout("callback", 240,
                                   "https://api.coindesk.com/v1/bpi/currentprice.json",
                                   "", "bpi.USD.rate");
}
function startTesting() public {
    if(msg.sender == owner)
        bytes32 newkey = setURLTimeout("callback", 0,
                                       "https://api.coindesk.com/v1/bpi/currentprice.json",
                                       "", "bpi.USD.rate");
}
```



> Most contracts do not have their **source code available**

Unstable Language Semantics

> Solidity frequently releases **new versions** and undergoes **semantic changes**

> Other languages often do not have **language specifications**

```
pragma solidity ^0.4.11;

contract ReactExample {
    address private owner;

    function ReactExample() public {
        owner = msg.sender;
    }
    function kill() public {
        require(msg.sender == owner);
        selfdestruct(owner);
    }
    function () public payable {
        revert();
    }
}
```

Solidity before ver. 0.5.0

→

```
pragma solidity ^0.5.0;

contract ReactExample {
    address payable private owner;

    constructor () public {
        owner = msg.sender;
    }
    function kill() public {
        require(msg.sender == owner);
        selfdestruct(owner);
    }
    function () external payable {
        revert();
    }
}
```

Solidity after ver. 0.5.0

Lack of Clear Property Definition

> Existing analyzers do not have **clear definitions** of vulnerabilities

- The detected "**patterns**" do not necessarily imply vulnerabilities

```
function enter() {
    uint amount = msg.value;
    if ((amount < contribution)...) {
        msg.sender.send(msg.value);
        return;
    }
    addParticipant(msg.sender, inviter);
    address next = inviter;
    uint rest = amount;
    uint level = 1;
    while ((next != top) && (level < 7)) {
        uint toSend = rest/2;
        next.send(toSend);
        /.../
    }
    next.send(rest);
    Tree[next].totalPayout += rest;
}
```

→

Two paths that have **different Ether flows** exist?

YES! ↓

Transaction Order Dependency vulnerability

Lack of Clear Property Definition

> Existing analyzers do not have **clear definitions** of vulnerabilities

- The detected "**patterns**" do not necessarily imply vulnerabilities

```
function enter() {
    uint amount = msg.value;
    if ((amount < contribution)...) {
        msg.sender.send(msg.value);
        return;
    }
    addParticipant(msg.sender, inviter);
    address next = inviter;
    uint rest = amount;
    uint level = 1;
    while ((next != top) && (level < 7)) {
        uint toSend = rest/2;
        next.send(toSend);
        /.../
    }
    next.send(rest);
    Tree[next].totalPayout += rest;
}
```

Two paths that have **different Ether flows** exist?

YES!

Transaction Order Dependency vulnerability

False Positive!

Promising Research Directions

> New **Language Design**

- Languages with **fully specified semantics** are amenable to analysis

> **Type-based Approaches**

- Most statically-typed languages do not support **strong type checking**
- Enriching existing type systems can **discover vulnerabilities**

```
contract Bob {
    function bar(Alice alice){
        alice.foo(42);
    }
}
```

```
contract Alice {
    function foo(uint x) returns (uint) {
        return x;
    }
}
```

Compiler before ver. 0.5.0 does not check whether `alice` is an address of `Alice` contract

Promising Research Directions

> **Machine Learning** Techniques

- Machine learning may improve the **scalability and accuracy** of smart contract analysis

> Rising **Platforms** and **Languages**

- Blockchain ecosystem is still **evolving very fast**
- Research on rising platforms and languages is **at an early stage**
  - e.g. Hyperledger Fabric, EOS (platform)
    e.g. Serpent, Vyper (language)

Thanks for your Listening!