



**VIRGINIA TECH™**

# Vulnerabilities in Smart Contracts



Raghav Agrawal  
James Hinton  
Sydney May  
Rohit Sathye

# OUTLINE



---

# WHAT TO EXPECT

1. Problem Statement
2. Motivation
3. Limitations
4. System, Network, Threat and Security Models
5. Our Work
6. Results
7. Challenges
8. Conclusion



**PROBLEM STATEMENT  
MOTIVATION  
LIMITATION**



---

# ELEPHANT IN THE ROOM

What if Smart Contracts are not SMART!

Immutable on the Blockchain?

Vulnerabilities live forever.



# ELEPHANT IN THE ROOM

What if Smart Contracts are not SMART!

Immutable on the Blockchain?

Vulnerabilities live forever.





---

# ELEPHANT IN THE ROOM

What if Smart Contracts are not SMART!

Immutable on the Blockchain?

Vulnerabilities live forever.

**What are the most common security vulnerabilities in Smart Contracts? How can they be identified? How can they be fixed?**





---

# WHY WE CHOSE THIS

Smart Contracts : hot topic in Blockchain

Team majoring in Security

Project = Smart Contracts + Security





# LIMITATIONS

---

Vulnerability Scanners for smart contracts are **EVERYWHERE...**

...But **MULTIPLE** are needed to **FULLY** check for vulnerabilities in smart contracts

Having to use multiple scanners to check the security of a smart contract puts the **burden of security on the user**

This can lead to many users **AVOIDING VULNERABILITY SCANNING** their smart contracts at all as scanning is seen as **TOO LARGE A BURDEN**



# MODELS

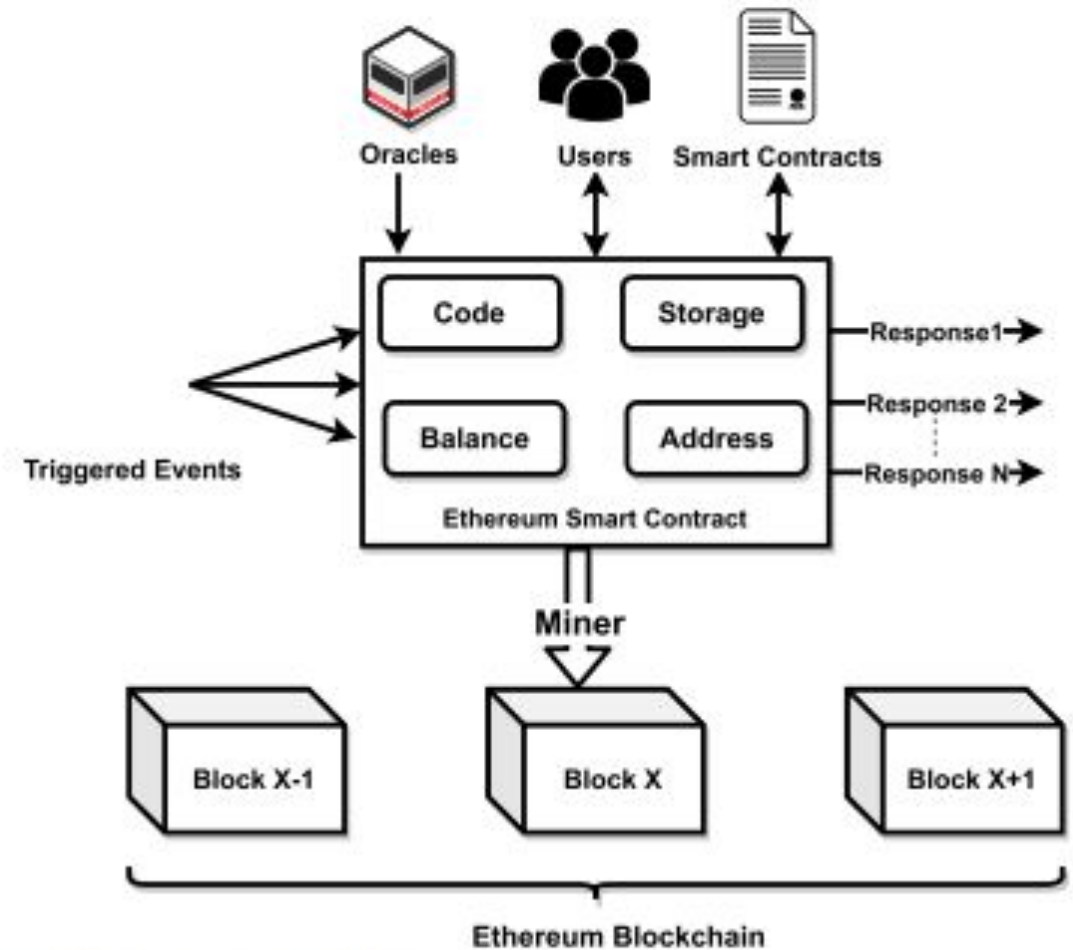
# SYSTEM

## Components:

- Code
- Storage
- Balance
- Address

## Actors:

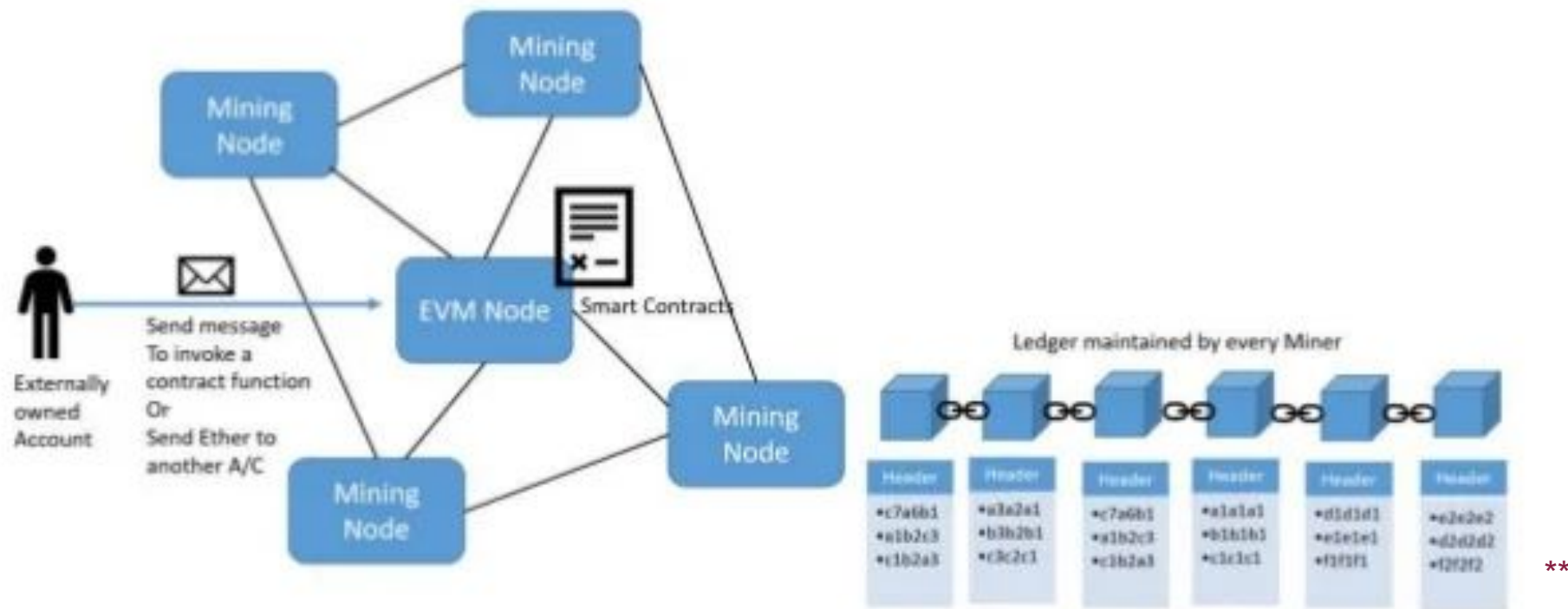
- Miners
- Oracles
- Users
- Other Smart Contracts



**FIGURE 5.** Structure of Ethereum smart contract. \*\*

# NETWORK

## Ethereum Blockchain framework



# THREAT

1. Code vulnerabilities
2. Malicious code
3. Social engineering
4. Supply chain attacks
5. Governance issues



# SECURITY

1. Implement Robusts Access Control
  - a. Ownable Pattern
  - b. Role Based Pattern
2. Use safe code functionality
  - a. `assert()`
  - b. `revert()`
  - c. `require()`
3. Test Contracts and Verify Code Correctness
4. Independent Code Reviews
5. Implement Disaster Recovery Plans
6. Event Monitoring
7. Reduce Code Complexity
8. Defend Against Common Code Vulnerabilities



# WHAT WE DID

# DAMN VULNERABLE SMART CONTRACT





# DAMN VULNERABLE SMART CONTRACT

- Smart Contract like DVWA
- 7 vulnerabilities packed into one contract
  - Reentrancy
  - Arithmetic Issues
  - Unchecked return values
  - Access control
  - Denial of Service
  - Bad Randomness
  - Front-Running

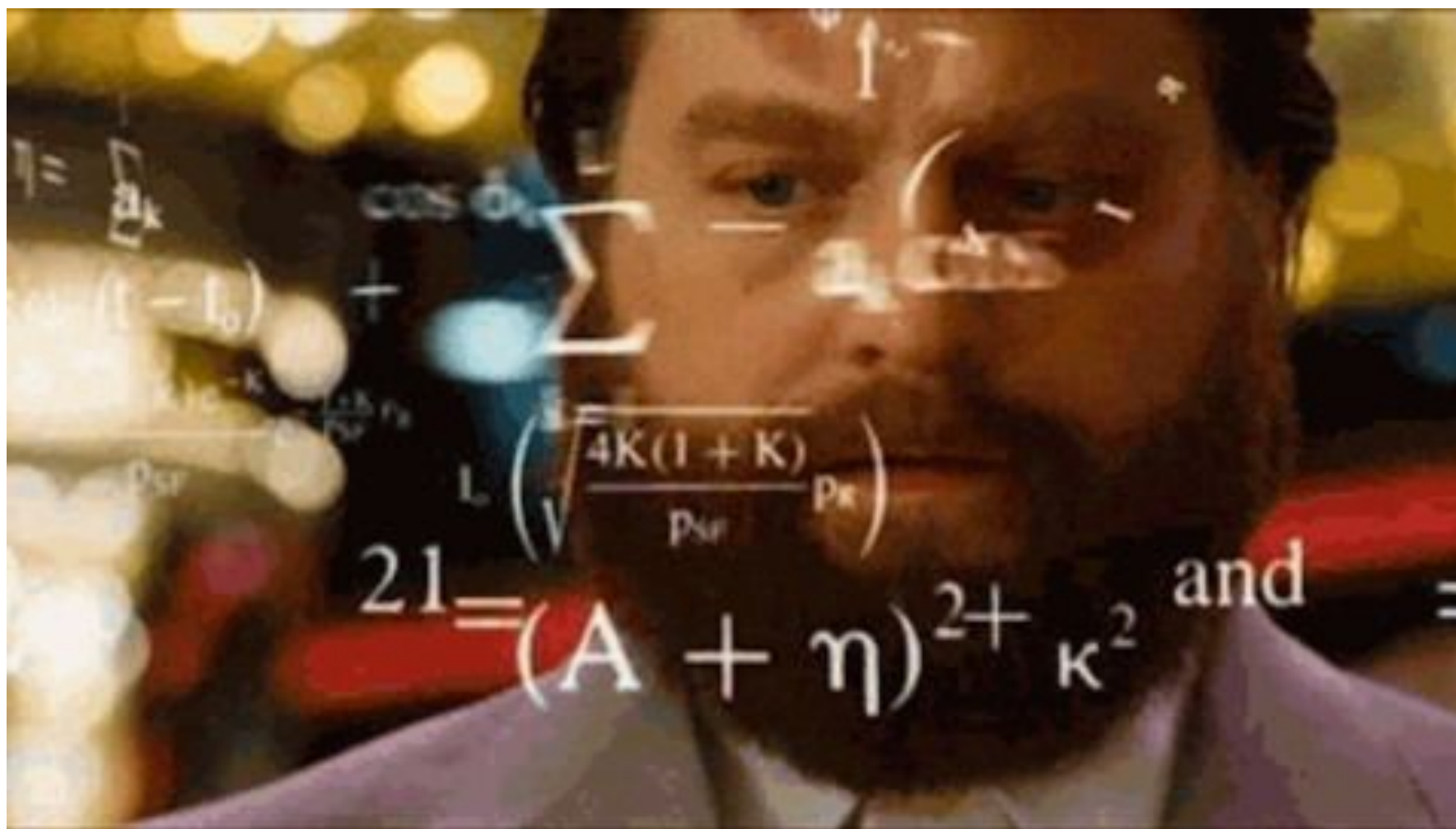


# Reentrancy

```
function withdraw_reentrancy(uint _amount) public {  
    require(balance >= _amount );  
  
    //Re-entrancy  
    (bool ret,) = msg.sender.call{value: _amount}("");  
    require(ret, "Send Failed");  
    balance -= _amount;  
}
```

- FALLBACK function of the “sender”
- Evil sender calls vulnerable Smart Contract again
- Recursive call
- Empty the Smart Contract
- HARD FORK

# Arithmetic Issues



# Arithmetic Issues

- Overflow
- Underflow

```
// Arithmetic issues
function withdraw_underflowflow(uint _amount) public {
    require(msg.sender == owner, "Only owner can widthdar");
    uint diff = balance - _amount;
    require(diff >= 0);
    balance -= _amount;
    (bool ret,) = msg.sender.call{value: _amount}("");
    require(ret, "Send Failed");
}
```

# Arithmetic Issues

- **VAR** keyword (deprecated)

```
uint public answer;

function loop_arithmetic() public {
    uint value = 1000;
    for (var i = 0; i < value; i++) {
        answer = i;
    }

    // When i goes past 255, it wraps back to 0;
    // The loop continues for ever and runs out of gas.
}
```

# Access Control



# Access Control

- Who is allowed to call the function?

```
// Access control
function init() public {
    owner = msg.sender;
}

constructor() {
    init();
}
```

- Parity multi-sig Wallet



---

# Unchecked Return Values

- Low level functions like CALL
- What happens when they fail?
- FALSE return value
- Need to catch to maintain proper state



# Unchecked Return Values

```
34  ✓ function withdraw_noRetCheck(uint _amount) public {
35      require(msg.sender == owner, "Only owner can widthdar");
36      require(balance >= _amount );
37      //balances[msg.sender] -= _amount;
38      balance -= _amount;
39
40      // Not catching return values
41      msg.sender.call{value: _amount}("");
42  }
43
44  ✓ function withdraw(uint _amount) public {
45      require(balance >= _amount );
46      balance -= _amount;
47      (bool ret,) = msg.sender.call{value: _amount}("");
48      require(ret, "Send Failed");
49  }
```

# Denial Of Service

- Logic Vulnerability
- Can render the contract useless
- Probably caused by other vulnerability

Here:

Funds get transferred to admin, so users can not cancel their bid. Even if admin fixes, attacker can relaunch the attack.

```
function bid(uint amount) public {
    total_balance += amount;
    balances[msg.sender] += amount;
}

function cancel() public {
    total_balance -= balances[msg.sender];
    (bool ret,) = msg.sender.call{value: balances[msg.sender]}("");
    require(ret, "Send Failed");
    balances[msg.sender] = 0;
}

function withdraw() public{
    (bool ret,) = owner.call{value: total_balance}("");
    require(ret, "Send Failed");
}
```

# Bad Randomness

- Using a non-random value for randomness
- Ex: using the hash of a block as random number is not safe
- Block is recent so attacker can recompute rand num easily

```
contract DVSC3 {  
  
    // solution  
    uint mySecretSol = 75  
  
    function pay() public payable {  
        require(msg.value >= 1 ether);  
        // using hash of block num as rand num  
        if (block.blockhash(blockNumber) % 2 == 0) {  
            msg.sender.transfer(this.balance);  
        }  
    }  
}
```

# Front-Running

- No private data should NEVER be stored in a smart contract
- Storing solution = attackers steal and copy tx with higher fee
- Result: attacker tx chosen by miner and attacker wins prize

```
contract DVSC3 {  
    // solution  
    uint mySecretSol = 75  
  
    function pay() public payable {  
        require(msg.value >= 1 ether);  
        // using hash of block num as rand num  
        if (block.blockhash(blockNumber) % 2 == 0) {  
            msg.sender.transfer(this.balance);  
        }  
    }  
}
```

# VULNERABILITY SCANNING

Process of systematically identifying vulnerabilities in software by using different techniques.

The purpose of this is to detect potential security weaknesses that can be exploited by attackers.

Some popular techniques to detect vulnerabilities:

1. Code Review
2. Static Analysis
3. Dynamic Analysis
  - a. Fuzzing
4. Penetration Testing



# 1. Code Review

Manually checking syntax errors, logical flaws, and other vulnerabilities

Which code below do you believe prevents a vulnerability from occurring?

Q1: Which of the following code snippets **PREVENTS** a vulnerability?

```
String updateServer = request.getParameter("updateServer");
if(updateServer.indexOf(";")==-1 && updateServer.indexOf("&")==-1){
    String [] commandArgs = {
        Util.isWindows() ? "cmd" : "/bin/sh",
        "-c", "ping", updateServer
    }
    Process p = Runtime.getRuntime().exec(commandArgs);
}
```

```
String updateServer = request.getParameter("updateServer");
if(ValidationUtils.isAlphanumericOrAllowed(updateServer, '-', '_', '.')){
    String [] commandArgs = {
        Util.isWindows() ? "cmd" : "/bin/sh",
        "-c", "ping", updateServer
    }
    Process p = Runtime.getRuntime().exec(commandArgs);
}
```

# 1. Code Review

Q1: Which of the following code snippets **PREVENTS** a vulnerability?

```
String updateServer = request.getParameter("updateServer");
if(updateServer.indexOf(";")==-1 && updateServer.indexOf("&")==-1){
    String [] commandArgs = {
        Util.isWindows() ? "cmd" : "/bin/sh",
        "-c", "ping", updateServer
    }
    Process p = Runtime.getRuntime().exec(commandArgs);
}
```

```
String updateServer = request.getParameter("updateServer");
if(ValidationUtils.isAlphanumericOrAllowed(updateServer, '-', '_', '.')){
    String [] commandArgs = {
        Util.isWindows() ? "cmd" : "/bin/sh",
        "-c", "ping", updateServer
    }
    Process p = Runtime.getRuntime().exec(commandArgs);
}
```

Incorrect! While there is some validation, it is based on block listing and will still allow command injection (ex. `rm -rf /`).

Only checks if input does not contain 'i' or '&'

- Would allow an attacker to execute arbitrary commands on host operating system

# 1. Code Review

Q1: Which of the following code snippets **PREVENTS** a vulnerability?

```
String updateServer = request.getParameter("updateServer");
if(updateServer.indexOf(";")==-1 && updateServer.indexOf("&")==-1){
    String [] commandArgs = {
        Util.isWindows() ? "cmd" : "/bin/sh",
        "-c", "ping", updateServer
    }
    Process p = Runtime.getRuntime().exec(commandArgs);
}
```

```
String updateServer = request.getParameter("updateServer");
if(ValidationUtils.isAlphanumericOrAllowed(updateServer, '-', '_', '.')){
    String [] commandArgs = {
        Util.isWindows() ? "cmd" : "/bin/sh",
        "-c", "ping", updateServer
    }
    Process p = Runtime.getRuntime().exec(commandArgs);
}
```

You're right! The code sample is preventing a vulnerability by using input allow listing.

Checks if input only has valid characters

## 2. Static Analysis

Takes an input (source code of a smart contract) and returns if it satisfies a property or not

Important to note that it does not execute the input

It reasons all the possible paths the input could take during the execution

- **Example:** examining the structure of the source code to determine what it would mean for the contracts operation at runtime

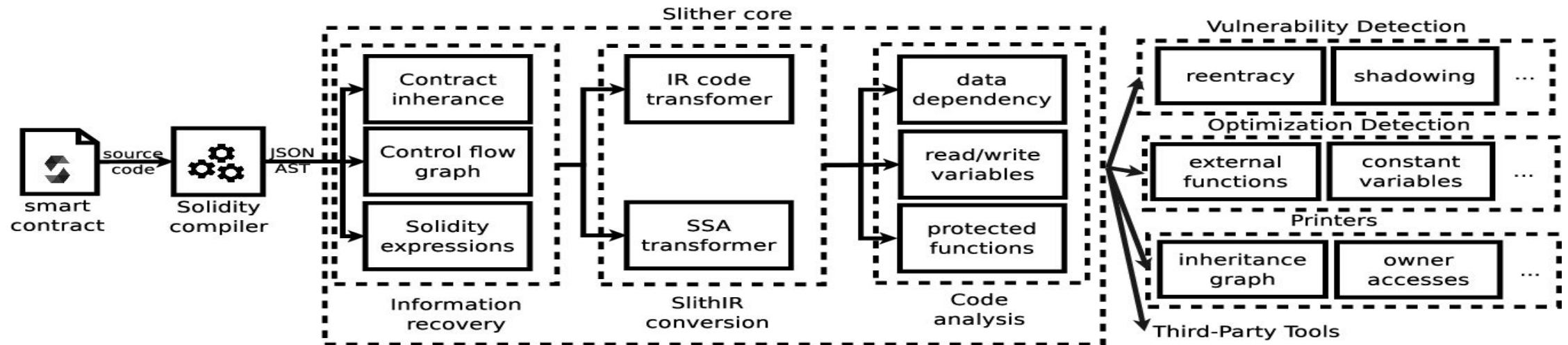


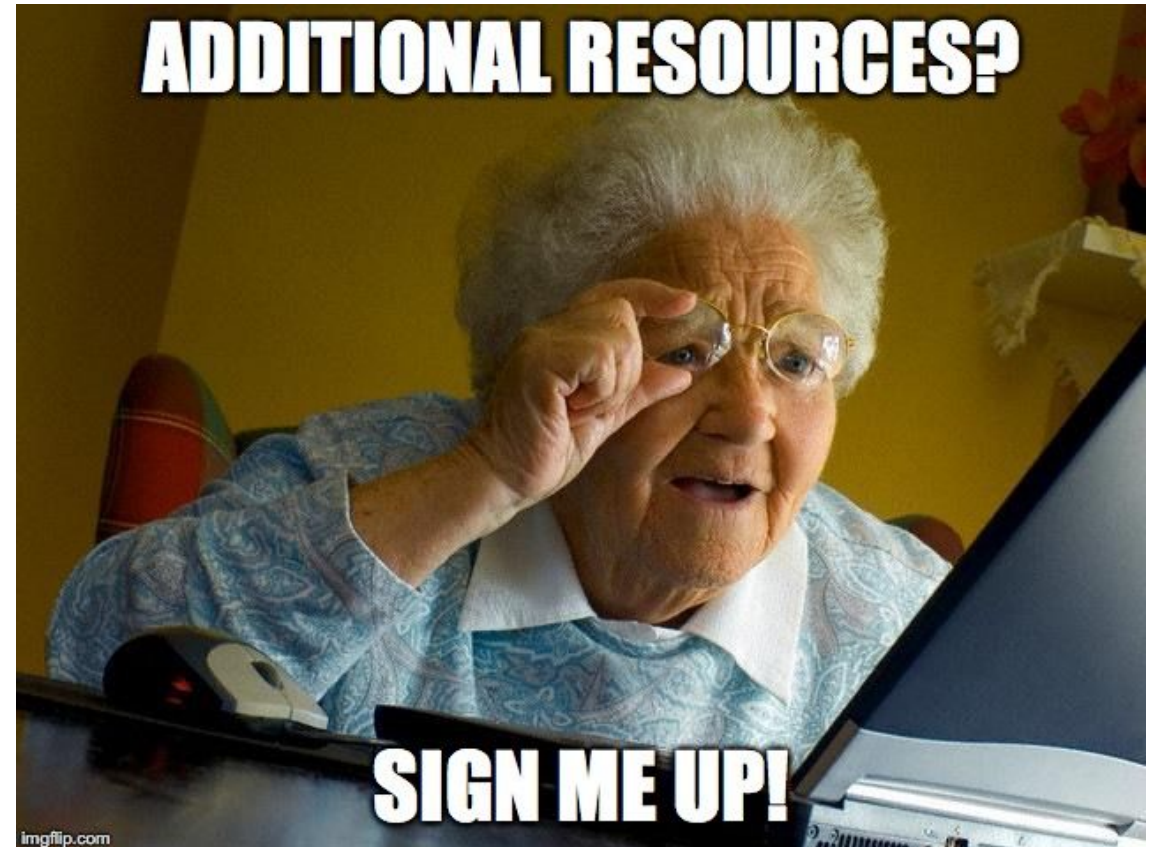
Fig. 1: Slither overview

### 3. Dynamic Analysis

Dynamic analysis generates symbolic inputs or concrete inputs to a smart contracts functions to see if any execution trace(s) violates specific properties

The tradeoff with dynamic analysis is that:

- Can identify vulnerabilities that only appear during execution
- Can be more time consuming and require more resources to perform



## 3a. Fuzzing

Fuzzing is an example of dynamic analysis technique for verifying arbitrary properties in smart contracts

Main idea:

- Send randomized input data to an application
  - Usually corrupted, unexpected data types/formats, or larger than normal amount of data
- See if any weaknesses in the application's input validation or error handling processes appear
  - Usually leads to buffer overflows or code injection

The main advantage of fuzzing is that it can simulate real-world scenarios

The main disadvantage of fuzzing is that it may require significant computing resources and expertise

- Also important to note this does not find all vulnerabilities

## 4. Penetration Testing

Penetration testing involves simulating real-world attacks on a smart contract to identify vulnerabilities

Usually involves a team of security professionals who identify vulnerabilities, then attempt to exploit them

The main difference between fuzzing and penetration testing:

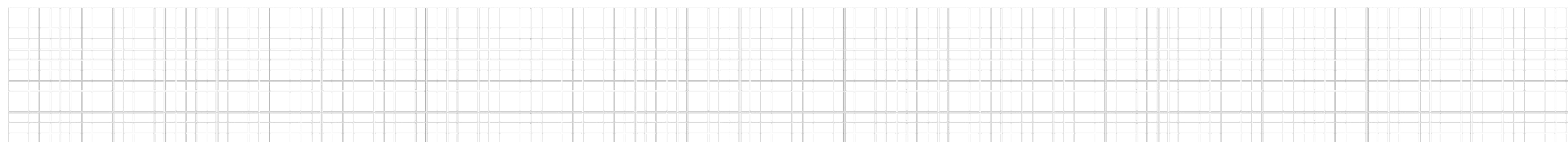
- Fuzzing focuses on input validation or error handling processes
- Penetration testing focus vulnerabilities through real-world attacks and assess the security posture of the smart contract



# Contract Analysis Tools

Echidna: A Fast Smart Contract Fuzzer 

---



# RESULTS

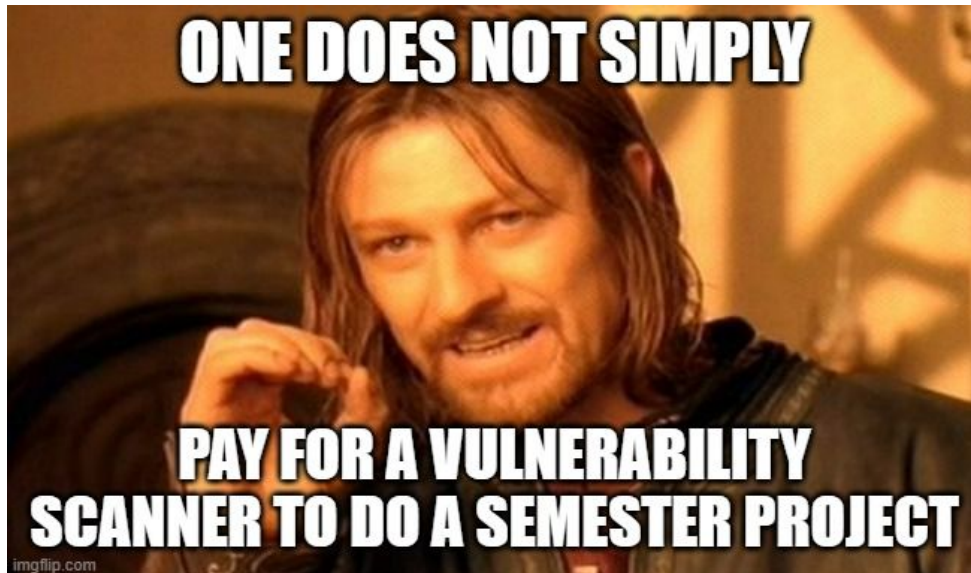
Contract	Vulnerability	Echidna	Slither	Securify	Remix
dvsc.sol	Reentrancy	YES	YES	YES	YES
dvsc.sol / dvsc2.sol	Unchecked Math Underflow / Overflow	YES	NO	NO	NO
dvsc2.sol	Incorrect Declarations (var in dvsc2.sol)	NO	NO	YES	NO
dvsc3.sol	Visibility/ Exposed Secrets	NO	YES	YES	NO
dvsc3.sol	Bad Randomness	YES	YES	NO	NO
dvsc.sol	Unchecked Return values	NO	NO	YES	YES
dvsc.sol	Access Control	NO	YES	YES	NO
	7	3	4	5	2

# CHALLENGES

Some Vulnerabilities Difficult to Implement: Time Manipulation, Short Address

Some Vulnerabilities are Unknown or Not Yet Exploited

Not all Vulnerability Scanners for Smart Contracts are Open-Sourced



Attackers be like:



# CONCLUSIONS



# LEARNINGS

Smart Contracts can be very vulnerable

Creating a vulnerable smart contract = easy

Creating a secure smart contract = difficult





---

# CONCLUSIONS

Write secure smart contracts

Implement access control mechanism and minimize the attack surface

Thoroughly test your smart contract

Stay current with the latest security threats and vulnerabilities in the smart contract industry





# REFERENCES

Kushwaha, S. S., Joshi, S., Singh, D., Kaur, M., & Lee, H. N. (2022). Systematic review of security vulnerabilities in ethereum blockchain smart contract. *IEEE Access*, 10, 6605-6621.

Li, W., Bu, J., Li, X., & Chen, X. (2022, August). Security analysis of DeFi: Vulnerabilities, attacks and advances. In *2022 IEEE International Conference on Blockchain (Blockchain)* (pp. 488-493). IEEE.

Mense, A., & Flatscher, M. (2018, November). Security vulnerabilities in ethereum smart contracts. In *Proceedings of the 20th international conference on information integration and web-based applications & services* (pp. 375-380).

Ren, M., Ma, F., Yin, Z., Fu, Y., Li, H., Chang, W., & Jiang, Y. (2021, August). Making smart contract development more secure and easier. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 1360-1370).

Sayeed, S., Marco-Gisbert, H., & Caira, T. (2020). Smart contract: Attacks and protections. *IEEE Access*, 8, 24416-24427.



QUESTIONS



**VIRGINIA TECH™**