a place of mind
THE UNIVERSITY OF BRITISH COLUMBIA

eTainter: Detecting Gas-Related Vulnerabilities in Smart Contracts

Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman

Artifacts Evaluated
acm
Reusable

ISSTA 2022

Ethereum smart contracts

Increasing adoption

- Finance (DeFi), gaming (NFTs), etc
- Hold nearly 23% of Ethereum supply (~$30B) [1]

Attack incidents

- DAO: $60M theft
- Parity: $300M lost

[1] https://etherscan.io/stat/supply

Smart contracts

```
contract PIPOT {
  struct order {
    address player;
    uint betPrice;
   }
  mapping (uint => order[]) orders ;

  function buyTicket (uint betPrice) public payable {
    orders[game].push(order(msg.sender, betPrice));
     //some code
    }

  function pickTheWinner(uint winPrice) public{
     //some code
     for(uint i=0; i< orders[game].length; i++){
        if (orders[game][i].betPrice == winPrice){
           orders[game][i].player.transfer(toPlayer);
          }
       }
    }
```

Entry points

Compiled

```
PUSH1 0x64
SWAP1
CALLVALUE
MUL
PUSH1 0x02
SLOAD
PUSH1 0x00
SWAP1
DUP2
MSTORE
PUSH1 0x08
PUSH1 0x20
MSTORE
```

EVM bytecode opcodes

Smart contracts: Gas concept

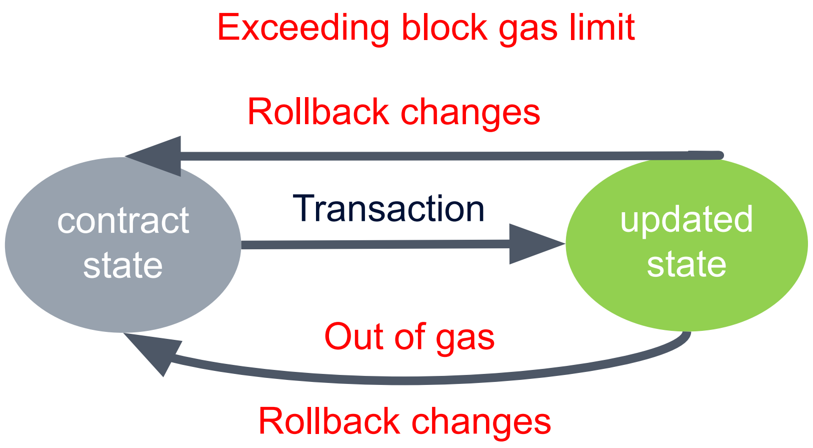
- Executing contract costs gas
- Gas cost for every EVM low-level instruction (opcode)
- Contract's users pay the gas cost

User specifies

- Function to call
- Max gas user wants to spend on executing the function

EVM bytecode opcodes	Gas cost
PUSH1 0x64	3
SWAP1	3
CALLVALUE	2
MUL	5
PUSH1 0x02	3
SLOAD	100-2100
PUSH1	3
SWAP1	3
DUP2	3
MSTORE	X
PUSH1 0x08	3
PUSH1 0x20	3
MSTORE	X

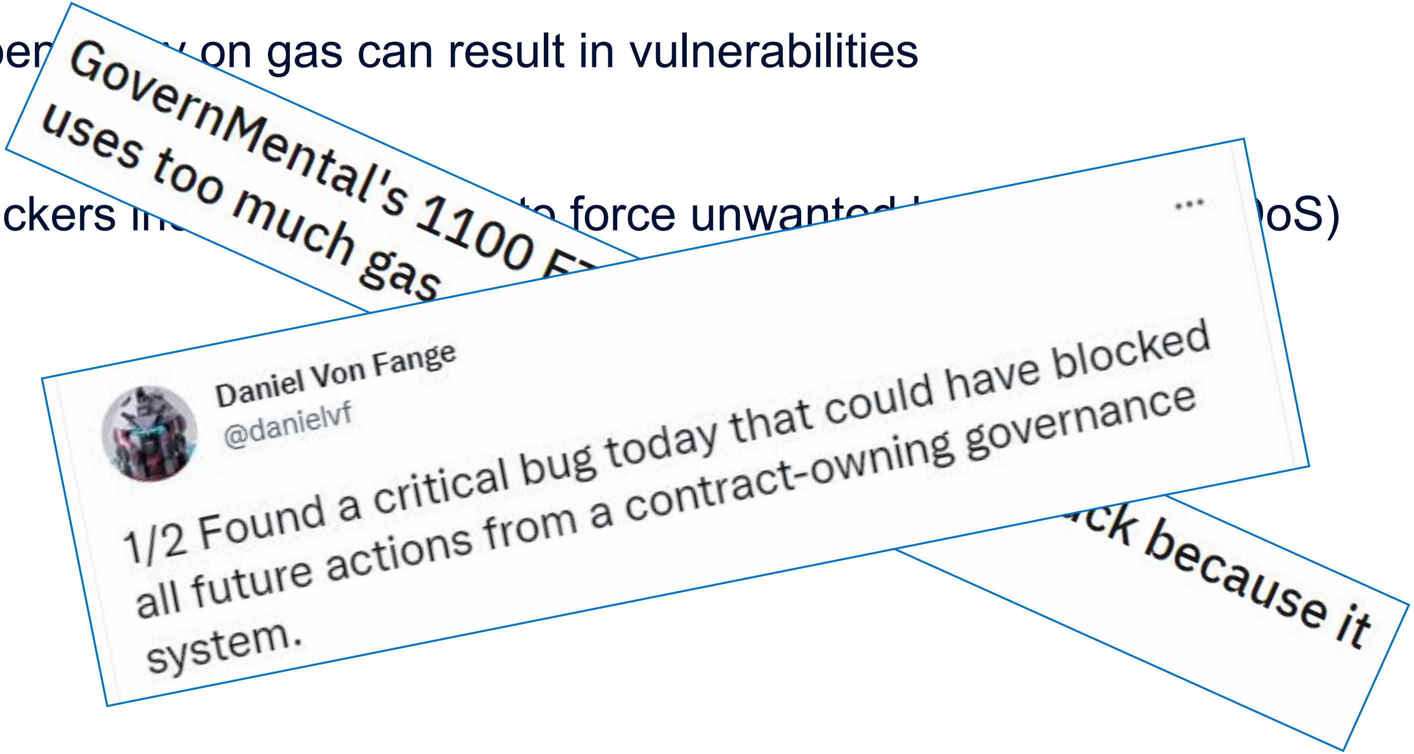
Smart contracts: Gas concept



- Ethereum enforces block gas limit
  - To ensure that blocks can’t be arbitrarily large
  - Transactions get reverted as well when exceeding the limit

Gas-related attacks and consequences

- Depen… on gas can result in vulnerabilities
- Attackers in… to force unwanted … oS)



Gas-related vulnerabilities

1. Unbounded Loop

```
contract PIPOT {
  struct order {
    address player;
    uint betPrice;
   }
  mapping (uint => order[]) orders ;

  function buyTicket (uint betPrice) public payable {
    orders[game].push(order(msg.sender, betPrice));
     //some code
    }
  function pickTheWinner(uint winPrice) public {
     // arbitrary length iteration
     for(uint i=0; i< orders[game].length; i++){
         if (orders[game][i].betPrice == winPrice){

             orders[game][i].player.transfer(toPlayer);
           }
        }
  }
```

Bounded by dynamic array

Gas-related vulnerabilities

2. Dos with Failed Call

```
contract PIPOT {
  struct order {
    address player;
    uint betPrice;
   }
  mapping (uint => order[]) orders ;

  function buyTicket (uint betPrice) public payable {
    orders[game].push(order(msg.sender, betPrice));
     //some code
    }
  function pickTheWinner(uint winPrice) public {
     // arbitrary length iteration
     for(uint i=0; i< orders[game].length; i++){
         if (orders[game][i].betPrice == winPrice){

            orders[game][i].player.transfer(toPlayer);
          }
       }
  }
```

Transfer ETH within the loop

Related work

MadMaX [OOPSLA, 2018]

- Uses pre-specified code patterns and rules
- Fails to detect variations in the patterns; results in high false-positives

```
for(uint i=0; i< orders[game].length; i++){
```

MadMax’s rule

```
.decl PossibleArrayIterator(loop: Block, resVar:Variable, arrayId:Value)
// A loop, looping through an array
// Firstly, the loop has to be dynamically bound by some storage var(resVar)
// And this must be the array's size variable.
PossibleArrayIterator(loop, resVar, arrayId) :-
  StorageDynamicBound(loop, resVar),
  PossibleArraySizeVariable(resVar, arrayId).
```

Our goal

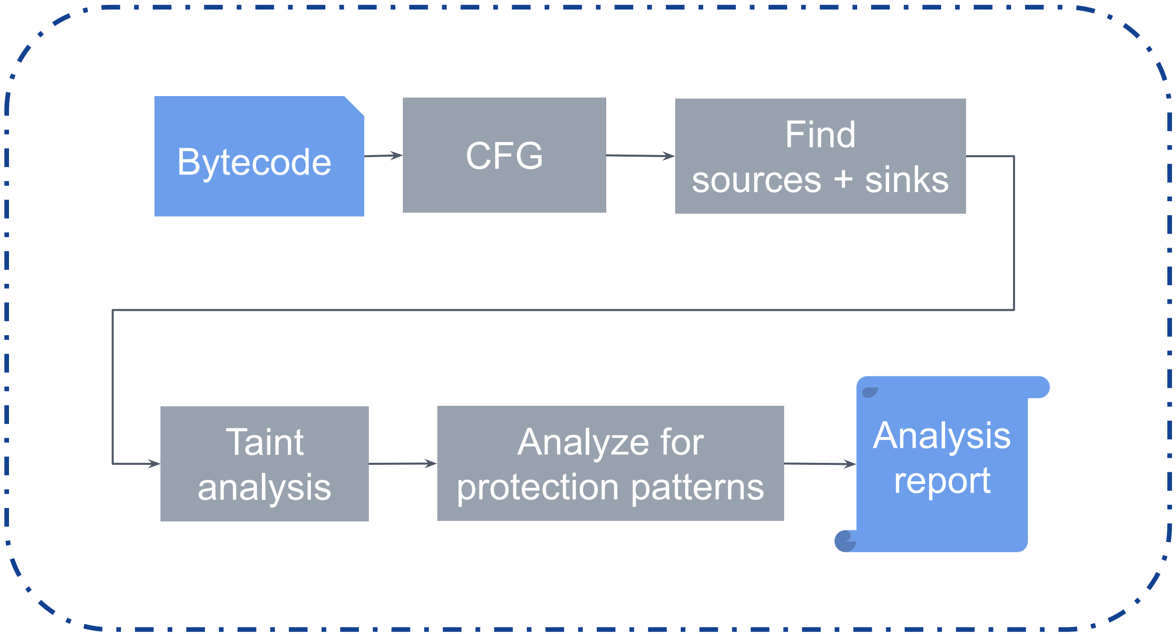
Goal:

- An approach for **detecting** smart contract **gas-related vulnerabilities using static taint analysis**

Observation:

- Gas-related vulnerabilities:
    - Caused by dependency on **user data sources manipulated by users**
    - Can be discovered by **tracking taints** without any pre-existing rules

Proposed approach: **eTainter**



Research challenges

- Patterns of use (e.g., implicit benign dynamic loops to handle strings)
- Propagating taints through persistent contract storage
- Unconventional way to access storage dynamic items (e.g. arrays) through hash calculation

Tainted data flow: Example

```
contract PIPOT {
  struct order {
    address player;
    uint betPrice;
   }
  mapping (uint => order[]) orders ;

  function buyTicket (uint betPrice) public payable {
    orders[game].push(order(msg.sender, betPrice));
     //some code
    }

  function pickTheWinner(uint winPrice) public {
     //some code
     for(uint i=0; i< orders[game].length; i++){
        if (orders[game][i].betPrice == winPrice){
           orders[game][i].player.transfer(toPlayer);
          }
      }
    }
```

Taint tracking

Sink: i< orders[game].length

Sources:

msg.sender
betPrice
winPrice

Tainted data flow: Example

```
contract PIPOT {
  struct order {
    address player;
    uint betPrice;
   }
  mapping (uint => order[]) orders ;

  function buyTicket (uint betPrice) public payable {
    orders[game].push(order(msg.sender, betPrice));
     //some code
    }

  function pickTheWinner(uint winPrice) public {
     //some code
     for(uint i=0; i< orders[game].length; i++){
        if (orders[game][i].betPrice == winPrice){
           orders[game][i].player.transfer(toPlayer);
          }
       }
    }
```

Taint tracking

Sink: i< orders[game].length

Sources:

- msg.sender
- betPrice
- winPrice
- orders[game]<needs validation>

Storage sink: orders[game]

Tainted data flow: Example

```
contract PIPOT {
  struct order {
    address player;
    uint betPrice;
   }
  mapping (uint => order[]) orders ;

  function buyTicket (uint betPrice) public payable {
   orders[game].push(order(msg.sender, betPrice));
     //some code
    }

  function pickTheWinner(uint winPrice) public {
     //some code
     for(uint i=0; i< orders[game].length; i++){
        if (orders[game][i].betPrice == winPrice){
           orders[game][i].player.transfer(toPlayer);
        }
     }
  }
```

Taint written to orders[game] array

Taint tracking

Sink: i< orders[game].length

Sources:

msg.sender
betPrice
winPrice
orders[game]<needs validation>

Storage sink: orders[game] tainted

Tainted data flow: Example

```
contract PIPOT {
  struct order {
    address player;
    uint betPrice;
   }
  mapping (uint => order[]) orders ;

  function buyTicket (uint betPrice) public payable {
    orders[game].push(order(msg.sender, betPrice));
     //some code
    }

  function pickTheWinner(uint winPrice) public {
     //some code
     for(uint i=0; i< orders[game].length; i++){
        if (orders[game][i].betPrice == winPrice){
           orders[game][i].player.transfer(toPlayer);
         }
      }
    }
```

Taint written to orders[game] array

Taint tracking

Sink: i< orders[game].length

Sources:

msg.sender
betPrice
winPrice
orders[game]<source of taints>

Storage sink: orders[game] tainted

Tainted data flow: Example

```
contract PIPOT {
  struct order {
    address player;
    uint betPrice;
   }
  mapping (uint => order[]) orders ;

  function buyTicket (uint betPrice) public payable {
    orders[game].push(order(msg.sender, betPr
     //some code
    }

  function pickTheWinner(uint winPrice) public {
     //some code
     for(uint i=0; i< orders[game].length; i++){
        if (orders[game][i].betPrice == winPrice){
           orders[game][i].player.transfer(toPlayer);
          }
      }
    }
```

Loop is unbounded

Taint tracking

Sink: i< orders[game].length
Sources:

msg.sender
betPrice
winPrice
orders[game]<source of taints>

Storage sink: orders[game] tainted

Taint reaches sink (loop exit condition)

eTainter evaluation

RQ1: Effectiveness of eTainter compared to prior work (MadMax)?

RQ2: Performance of eTainter?

RQ3: Prevalence of gas-related vulnerabilities in the wild?

Dataset	Contract Num.	Used for
Annotated dataset	28	RQ1
Ethereum dataset	60, 612	RQ2 & RQ3
Popular–contracts dataset	3,000	RQ3

eTainter evaluation

RQ1 (Effectiveness compared to MadMax)

	MadMax	eTainter
Precision	64.9%	90.4%
Recall	74%	94%
F1 score	69.2%	92.2%

eTainter achieves better recall and precision

eTainter evaluation

RQ2 (Performance of eTainter)

- <u>**Average analysis time:**</u> **8 seconds**
- <u>**Memory:**</u> **118 MB**

Practical analysis time

< 60 seconds for 97% of successfully analyzed contracts

eTainter evaluation

RQ3 (Prevalence of gas-related vulnerabilities)

Vulnerability	Ethereum dataset	Popular-contracts dataset
Unbounded loops	4.1%	1.8%
DoS with failed call	1.2%	0.8%
Total vulnerable contracts	2,763	71

Gas-related vulnerabilities are prevalent in both datasets

Summary

Goal: Effective approach for detecting gas-related vulnerabilities

- Introduced eTainter; a static taint analyzer
- eTainter achieved 92% F1 score compared to 69% for MadMax
- Flagged 2,763 contracts on Ethereum as vulnerable

eTainter code & artifact: https://github.com/DependableSystemsLab/eTainter

Asem Ghaleb, PhD Candidate at University of British Columbia

aghaleb@ece.ubc.ca