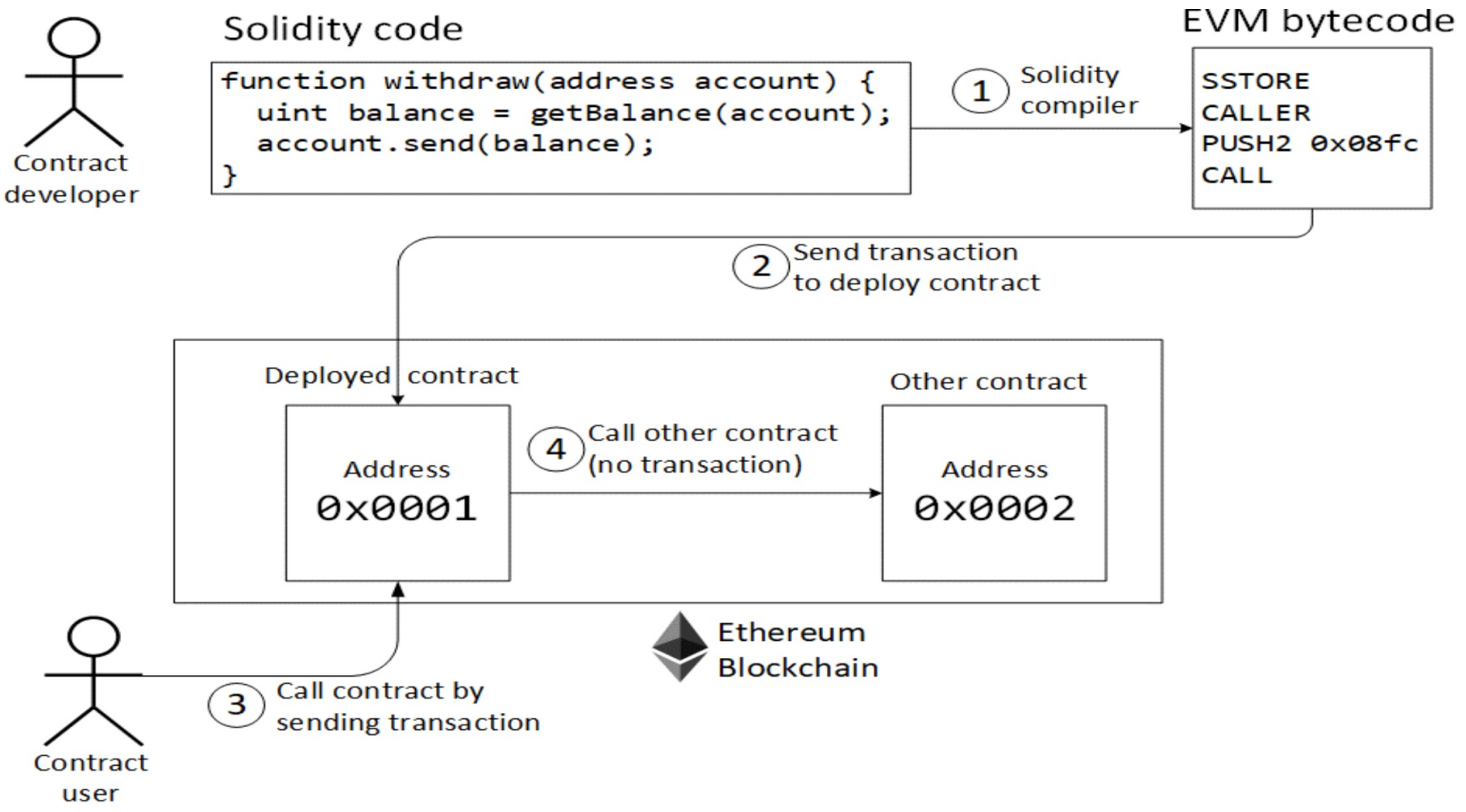
Broken Metre:
Attacking Resource Metering in EVM

Daniel Perez and Benjamin Livshits
Imperial College London

27th NDSS 2020

Background: EVM



⇒

DoS Attack?

Minners incentive?

Background: Gas Metering

- Each instruction consumes gas to execute, e.g. PUSH 3 gas, **EXTCODESIZE 20 gas**
- Program gas cost = base cost(2100 gas) + sum of instructions cost
- Program stops if it runs over its gas budget
- Transaction sender choses gas price and pays transaction fee = gas cost x gas price
- Minners takes transaction fee as an incentive

Problem

The Ethereum network has been victim of several Denial of Service (DoS) attacks due to instructions being underpriced.

EXTCODESIZE Attack

Attacker spammed network with transactions performing many EXTCODESIZE

Instruction	Gas Cost(gas)	IO ?
EXTCODESIZE	20	Y
SSTORE	20,000	Y
PUSH	3	N

EIP-150: EXTCODESIZE 20 gas ⇒ 700 gas, make attack very expensive

Work Overview

- Exploration of gas metering in EVM by Experiments
- Construct resource Exhaustion Attacks (REA) contract by generation strategy

Contributions

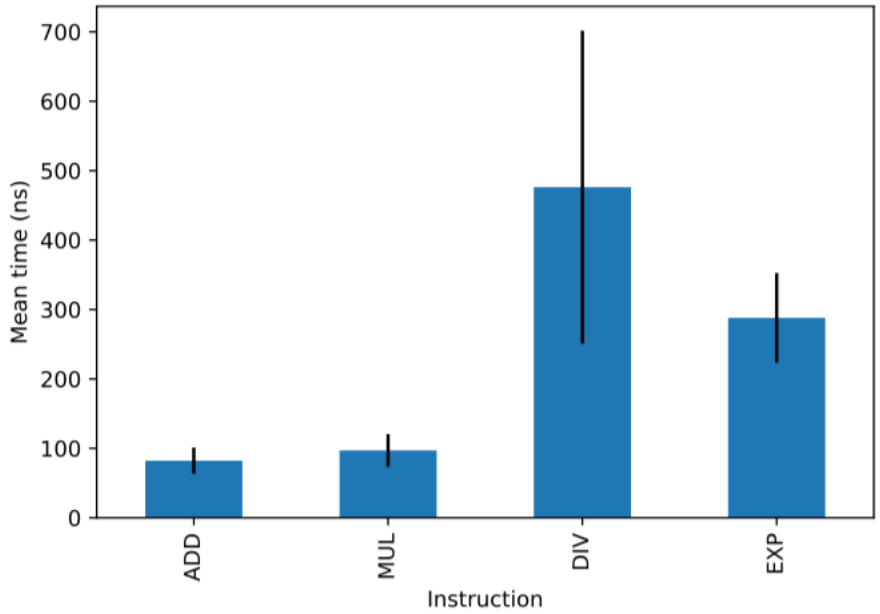
- Disclosed attack to Ethereum, and were awarded a bug bounty reward of 5,000 USD
- Present some of the short-term and long-term fixes in this paper

Experiment: Setup

- Hardware
  - 4 cores (8 threads) Intel Xeon at 2.20GHz,
  - 8 GB of RAM and an SSD with a 400MB/s throughput.
- Software
  - Aleth (C++ client)
  - Replay transactions and record stats

Experiment: Arithmetic Instructions

- ADD vs MUL: time similar, gas cost 65% higher
- DIV vs MUL: gas cost similar, time 5 times lower
- EXP: variable cost depending on arguments
- EXP vs DIV: gas 10 times, but time 40% faster



(a) Mean time for arithmetic instructions.

Conclution

simple instructions the execution time would not reflect the gas cost

Instruction	Gas cost	Count	Mean time (ns)	Throughput (gas / μs)
ADD	3	453,069	82.20	36.50
MUL	5	62,818	96.96	51.57
DIV	5	107,972	476.23	10.50
EXP	~51	186,004	287.93	177.1

(b) Execution time and gas usage for arithmetic instructions.

Experiment: gas and resources correlation

- Correlation with CPU (Execution Time) alone is non-existent
- Storage has the highest Pearson score
- Add CPU decreases the correlation with gas, while add Memory increases it

Phase	Resource	Pearson score
Pre EIP-150	Memory	0.545
	CPU	0.528
	Storage	0.775
	Storage/Memory	**0.845**
	Storage/Memory/CPU	0.759
Post EIP-150	Memory	0.755
	CPU	0.507
	Storage	0.907
	Storage/Memory	**0.938**
	Storage/Memory/CPU	0.893

Experiment: High-Variance Instructions

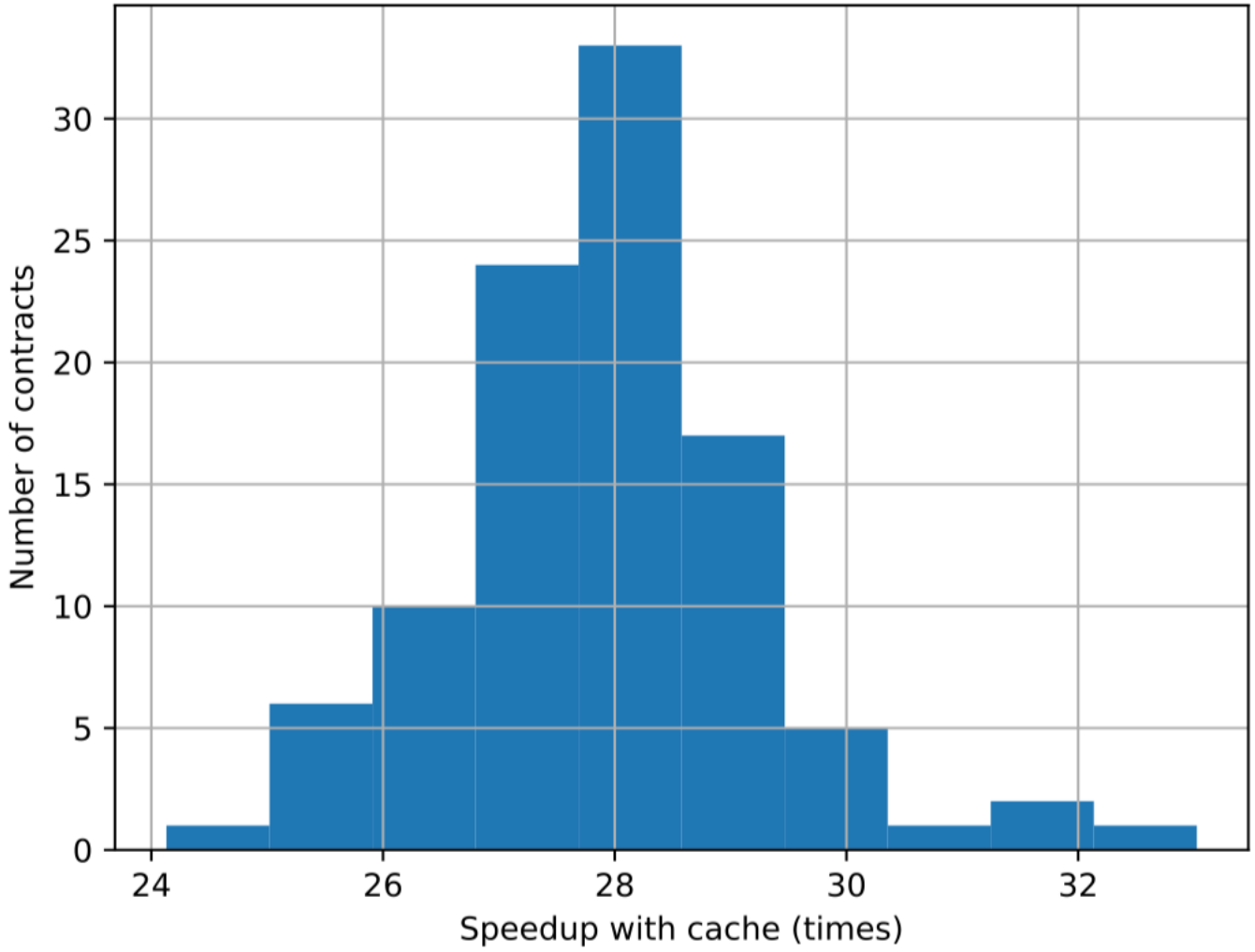
Reason

- Instruction with different parameters
- Instruction perform IO access, which can be influenced by many factors such as caching

Instruction	Mean time (μs)	Standard deviation	Measurements count
BLOCKHASH	768	578	240,000
BALANCE	762	449	8,625,000
SLOAD	514	402	148,687,000
EXTCODECOPY	403	361	23,000
EXTCODESIZE	221	245	16,834,000

Experiment: Effect of Cache on Execution Time

- Focus on OS page cache
- Generate random programs and measure speed with and without cache
- Programs run on average 28 times faster with page cache



Experiment: Conclusion

Execution time of instructions may not reflect the **gas cost**

IO causes a large gap in the execution time of instructions

Page cache has a significant impact on the execution speed

Possible Contract

- Low Gas Cost
- High Resources Use

Attack Model: Goal & Strategy

- Goal: Find a program which minimize possible throughput (gas / second)
- Strategy: Can be formulated as a search problem
  - Search space: Set of valid programs
  - Function to optimize: throughput
  - Constraint: gas budget
- Search space is too large to be explored entirely, so use a genetic algorithm to approximate a solution

Attack Model: Generated Programs

- We create programs valid by construction
  - Enough elements on stack
  - No stack overflows
  - Only access “reasonable” memory locations
- Cross-over and mutations also only create valid programs
- Generated programs do not contains loop, i.e. do not include JUMP or JUMPI instructions

Attack Model: Define functions

Functions function

- $r(I)$ returns the number of elements returned on the stack for an instruction I
- $a(I)$ returns the number of arguments consumed from the stack
- Generate instructions set by:
  - $\forall n \in [0,17], \mathbb{I}_s = \{ I \mid I \in \mathbb{I} \wedge a(I) \leq n \}$

Function to control memory access

- $true, false \leftarrow use\_memory(I)$ return if the given instruction accesses memory
- $P \leftarrow prepare\_stack(P, I)$ ensures memory accessed are below a low value (set 255)

Attack Model: Define functions

Fitness function

- $P \leftarrow$ thr$oughput(I)$ returns the measured throughput of a given instruction

Selection function

- $I \leftarrow biased\_sample(\mathbb{I}_s)$ returns a random instruction from the given instructions set

Define the weight and probability of choosing an instruction with:

$$W(I \in \mathbb{I}) = log\left(1 + \frac{1}{throughput(I)}\right)$$

$$P(I \in \mathbb{I}_n) = \frac{W(I)}{\sum_{I' \in \mathbb{n}} W(I')}$$

Attack Model: Initial Algorithm

Algorithm 1 Initial program construction

function GENERATEPROGRAM($size$)
    $P \leftarrow (\ )$ ▷ Initial empty program
    $s \leftarrow 0$ ▷ Stack size
    for 1 to $size$ **do**
        $I \leftarrow biased\_sample(\mathbb{I}_s)$
        if $uses\_memory(I)$ **then**
            $P \leftarrow prepare\_stack(P, I)$
        end if
        $P \leftarrow P \cdot (\ I\ )$ ▷ Append I to P
        $s \leftarrow s + (r(I) - a(I))$
    end for
    return P
end function

Attack Model: Cross-over

Algorithm 2 Cross-over function

function CREATESTACKSIZEMAPPING(P)
  $S \leftarrow$ empty mapping
  $pc \leftarrow 0$
  $s \leftarrow 0$
  for I in P **do**
    if $s \notin S$ **then**
      $S[s] \leftarrow \{\}$
    end if
    $S[s] \leftarrow S[s] \cup \{pc\}$
    $s \leftarrow s + (r(I) - a(I))$
    $pc \leftarrow pc + 1$
  end for
  return S
end function
function CROSSOVER(P_1, P_2)
  $S_1 \leftarrow$ CREATESTACKSIZEMAPPING(P_1)
  $S_2 \leftarrow$ CREATESTACKSIZEMAPPING(P_2)
  $S \leftarrow S_1 \cap S_2$ ▷ Intersection on keys
  $s \leftarrow sample(S)$
  $i_1 \leftarrow sample(S_1[s])$
  $i_2 \leftarrow sample(S_2[s])$
  $P_{11}, P_{12} \leftarrow split\_at(P_1, i_1)$
  $P_{21}, P_{22} \leftarrow split\_at(P_2, i_2)$
  $P_1' \leftarrow P_{11} \cdot P_{22}$ ▷ Concatenate
  $P_2' \leftarrow P_{21} \cdot P_{12}$
  return $P_1',\ P_2'$
end function

- $MAP \leftarrow CREATESTACKSIZEMAPPING(P)$

```
{
    0 : { 0 },          Key: stack size
    2 : { 1, 3 },
    5 : { 2 }           Value: pc
}
```

- $P_3, P_4 \leftarrow CROSSOVER(P_1, P_2)$

  - create mapping and randomly choose a stack size to split the program.
  - randomly choose a location from program with the selected stack size.
  - split program in two at the chosen position, then cross together.

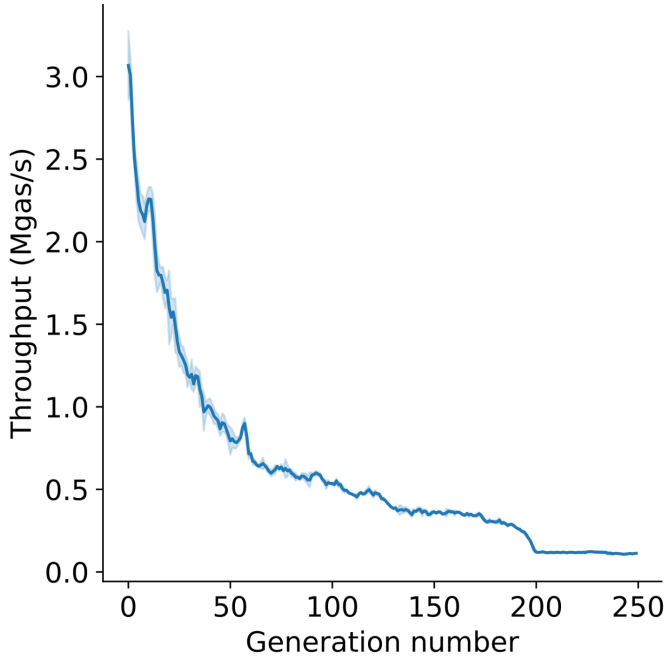
Attack Model: Mutation

We generate a set M Iof replacement candidate instructions defined as follow:

$$M_I = \{I' \mid I' \in \mathbb{I}_{a(I)} \wedge r(I') = r(I)\}$$

In other words, the replacement must require at most the same number of elements on the stack and put back the same number as the replaced instruction. Then, we replace the instruction I by I ′ , which we randomly sample from M I .

Attack Model: GA Result




```
PUSH9 0x57c2b11309b96b4c59
BLOCKHASH
SLOAD
CALLDATALOAD
PUSH7 0x25dfb360fa775a
BALANCE
MSTORE8
PUSH10 0x49f8c33edeea6ac2fe8a
PUSH14 0x1d18e6ece8b0cdbea6eb485ab61a
BALANCE
POP       ; prepare call to CALLDATACOPY
POP
POP
PUSH1 0xf7
PUSH1 0xf7
PUSH1 0xf7
CALLDATACOPY
PUSH7 0x421437ba67fe0e
ADDRESS
BLOCKHASH
```

- Initial program throughput: ~3M gas/s (compared to 20M on average)
- Decreases quickly to 500K
- Plateau at ~100K gas/s at generation 200

200x slower than average contract

- Many IO related instructions, e.g. BLOCKHASH and BALANCE
  - EIP-150: BALANCE 20 gas to 400 gas
  - This suggests that the instruction is still under-priced.
- Stack is replaced with small values before calling CALLDATACOPY.

Attack Model: Evaluation on Other Client

Client	Throughput (gas/s)	Time (s)	IO load (MB/s)
Aleth	107,349	93.6	9.12
Parity	210,746	47.1	10.0
Geth	131,053	75.6	6.57
Parity (bare-metal)	542,702	18.2	17.2
Geth (fixed)	3,021,038	3.33	0.72

Evaluation of different clients when executing 10M (1 block) gas worth of malicious transactions

The contracts crafted using our algorithm are also effective on the two most popular Ethereum clients:

- geth (v1.9.6)
- Parity Ethereum (v2.5.9)

Summary

- Re-execute several months of transactions and measure gas, CPU and memory consumption
  - Find several inconsistencies
  - Show the impact of caching on execution speed
- Present a new attack targeted at metering
  - Show that the attack works on all major clients
  - Disclosed attack to Ethereum Foundation and tested fixes