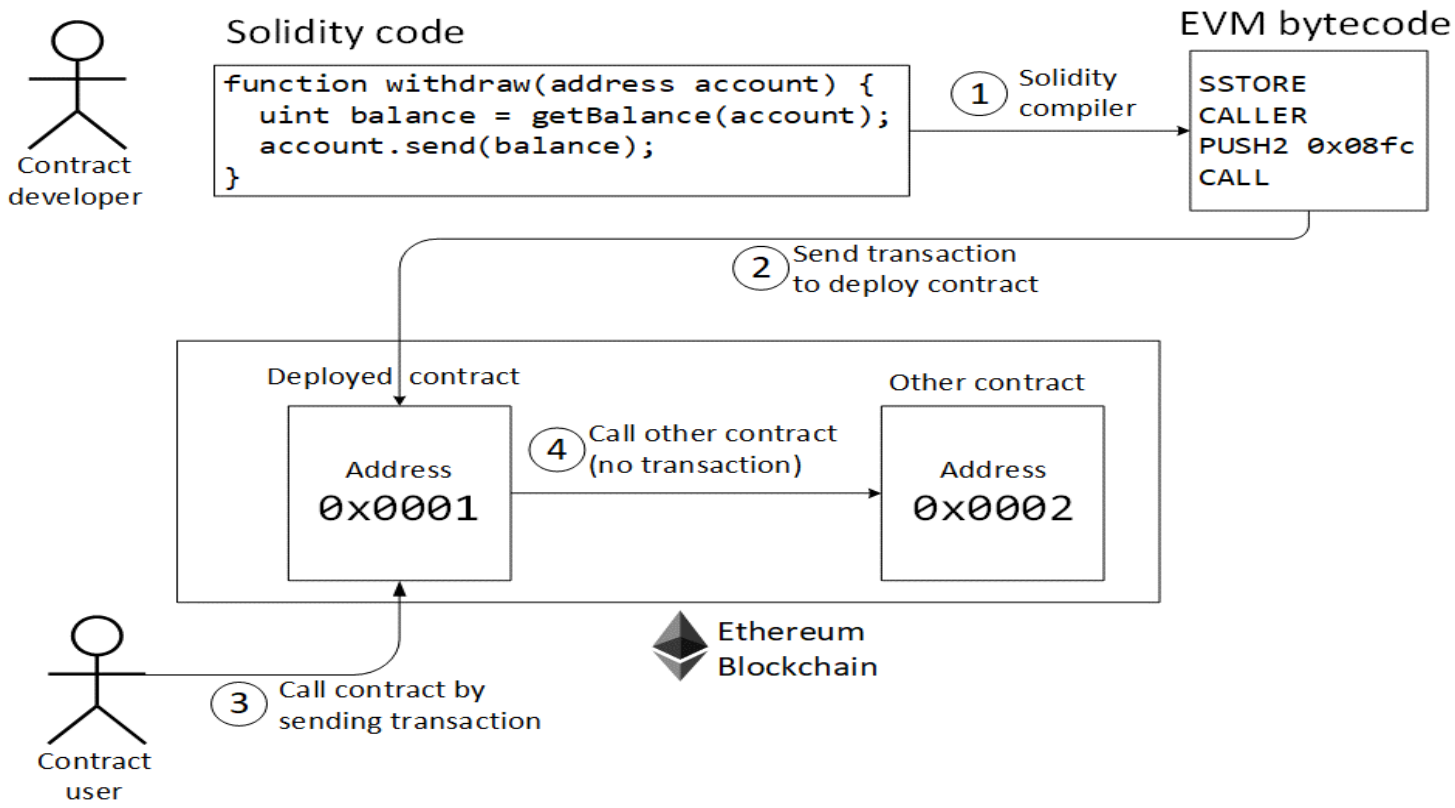
Broken Metre: Attacking Resource Metering in EVM

Daniel Perez and Benjamin Livshits

Imperial College London

Ethereum Smart Contracts



Gas Metering

- Each instruction consumes gas to execute
- Program gas cost = base cost + sum of instructions cost
- Program stops if it runs over its gas budget
- Transaction sender choses gas price and pays “gas cost x gas price”

Previous Attacks on Metering

EXTCODESIZE attack

- EXTCODESIZE is IO-intensive: needs to read the state
- Only cost 20 gas at time of attack
- Attacker spammed network with transactions performing many EXTCODESIZE
- Price was increased to 700 gas

SUICIDE attack

- SUICIDE kills a contract and sends all the Ether to a specified target
- SUICIDE was free at time of attack
- Specifying a new address when calling SUICIDE would create it for free
- Attacker spammed network with address creation/destruction
- SUICIDE priced changed to 5,000 and creating contract now consumes gas

Analysis Setup

- Fork aleth (C++ client)
- Instrument CPU
  - Record execution time/instruction
  - Aggregate over 1,000 instructions
- Instrument memory
  - Override new/delete
- Replay transactions and record stats

Gas and Resources Correlation

- Compute correlation between gas usage and different resources
- Correlation with CPU (execution time) alone is non-existent
- Adding **CPU decreases** the **correlation** with gas

Resource	Correlation
Memory	0.755
CPU	0.507
Storage	0.907
Storage/Memory	**0.938**
Storage/Memory/CPU	0.893

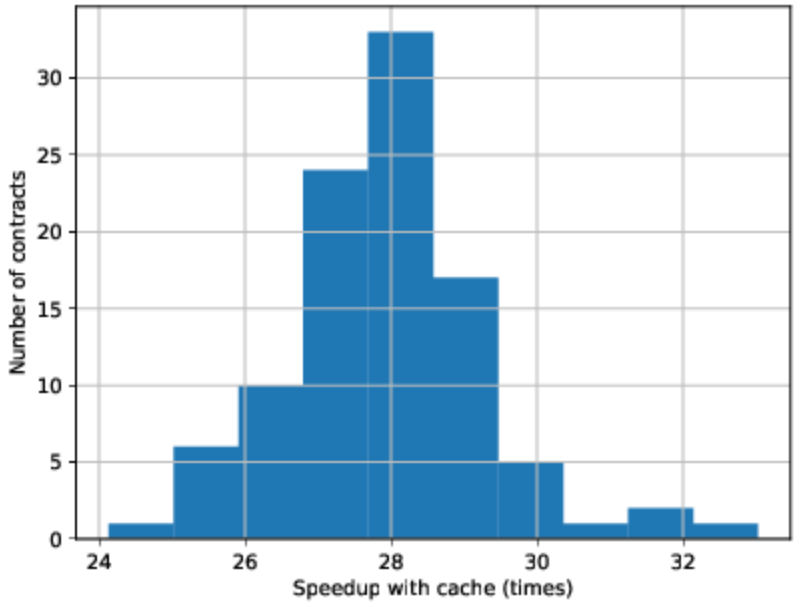
High-Variance Instructions

- Most high-variance instructions **depend on state**
- Even when aggregated over 1,000 calls, standard deviation is close to mean

Instruction	Mean (µs)	Stdev
BLOCKHASH	768	578
BALANCE	762	449
SLOAD	514	402
EXTCODECOPY	403	361
EXTCODESIZE	221	245

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



Resource Exhaustion Attack

- Goal is to find programs which minimize throughput (gas / second)

- 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
  - We use a **genetic algorithm** to approximate a solution

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. we do not include JUMP or JUMPI instructions

Initial Program Construction

- **Good initialization values** are important to converge in reasonable time
- To create initial program, we sample instructions as follow: given set of instructions *I*, we define the weight and probability of choosing an instruction with

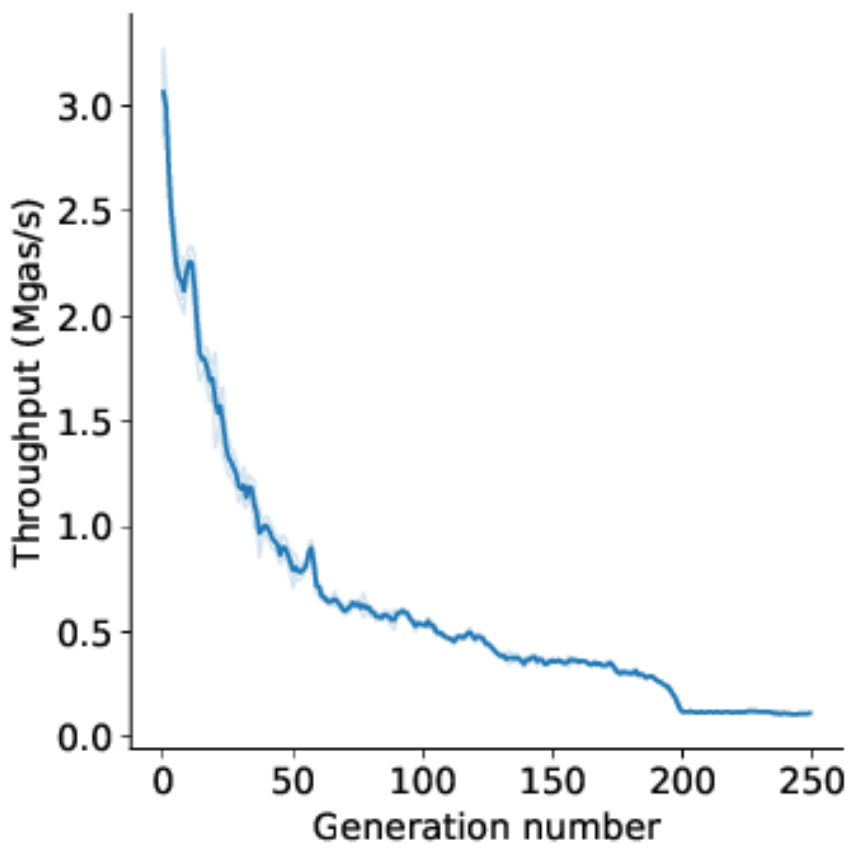
$$W(i \in I) = \log\left(1 + \frac{1}{throughput(i)}\right)$$

$$P(i \in I) = \frac{W(i)}{\sum_{i' \in I} W(i')}$$

Genetic Algorithm Results

- 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



DoS potential

- Implications
  - Nodes not being able to sync
  - Decrease in network throughput
- Probable attackers
  - Miners (selfish-mining)
  - Parties hostile to Ethereum (other chains)
  - Speculators
- Feasibility
  - Costs **only ~0.7 USD** to keep commodity hardware node out-of-sync for 1 block (~2M gas/block)
- Limitations
  - Current attack works best on commodity hardware
  - Hard to know what hardware full nodes are running

Evaluation on Different Clients

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

Improving Metering

Short term

- Increase cost of IO operations
  - Already seen in EIP 150 or EIP 2200
- Reduce number of required IO accesses
  - Flattened contracts state
  - Bloom filter to reduce search of inexistent contracts

Long term

- Stateless clients
  - Client do not need to keep track of all the state
  - Necessary data is sent with the transactions
- Sharding
  - Not a direct solution but less state needed per node

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
    - Thanks to Matthias Egli and Hubert Ritzdorf from PwC Switzerland

Supporting Slides

Responsible Disclosure

- 2019/10/3: Sent report to Ethereum Foundation through bounty program (thanks to Matthias Egli and Hubert Ritzdorf from PwC Switzerland)
- 2019/10/4: Reply from Ethereum Foundation
- 2019/10 – 2019/11: Tests with ongoing fixes
- 2019/11/17: Ethereum Foundation confirmed reward of 5000 USD
- 2020/1/7: Official bounty reward announcement

Arithmetic Instructions

Gas pricing for arithmetic instructions is **very inconsistent**

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**

Analysis Summary

- **Gas cost**: **Many inconsistencies**
- **IO operations**: very **high execution time variance**
- **Cache**: very important effect on speed
- **Overall**: **cannot model IO operations** very well