## TL;DR:

We're presenting:

- an opcode tracer

for TurboGeth that can process and export EVM-level transaction details at ~28 blocks per second

- a script that calculates fixed-size chunking merklization outcomes

from the basic blocks of transaction executions

- the results of running these tools on over 500K mainnet blocks: 1 byte seems to be the optimum length for chunking

, as far as witness-size is concerned.

ConsenSys' TeamX is moving on to other subjects, so we hope that these tools can be useful to the community.

# Merklization: the problem statement

One of the fundamental ideas of Stateless Ethereum is the use of witnesses, which provide just enough state context for a client to execute a block without any more state information.

Witnesses need to be as small as possible, so reducing their size is a priority. And contract code is a major contributor to witnesses size. Which brings us to code merklization: a variety of strategies to break down contract code to smaller chunks, so that only the necessary ones are included in the witness.

Such chunks need to still be verifiable by the stateless client, so they are accompanied by structured data (Merkle proofs) that allow the client to make sure that the code it receives corresponds to what the full state contains.

And the interaction between contract behavior, chunking strategies, Merkle proofs and witness size is what makes it tricky to decide what is the best way to merklize code.

In this post we're exploring the outcomes of the simplest chunking strategy: fixed-size chunking. In it, the contract is broken down into fixed-size segments, with the only variable being the size.

Given the peculiarities of EVM behavior, and as discussedin other posts, these chunks might additionally need some metadata, so that a stateless client can distinguish whether some bytecode in a chunk is part of PUSHDATA in the full contract.

# The results

BEWARE: THE OVERHEAD RESULTS ARE VERY DUBIOUS. THE PYTHON TOOL HAD FLAWS AND HAS BEEN EVOLVING. See comments for additional information.

We have run our merklization script on the traces of over 500K blocks of Mainnet transactions (from block 9M onwards). The key insights are:

- Real-world transactions' basic blocks have a median length of ~16

, with a statistical distribution very skewed towards smaller sizes

- The closest that chunks are to each other, the more that their proof hashes are amortized / the least hashes that are needed to build a multiproof. N sequential chunks need <= hashes than 1 chunk.

- The following overhead numbers are the result of running the merklization process on >500K blocks, but the same results are already apparent and stable within a ±1% margin in smaller batches of only 1000 blocks.

Merklization overhead is caused by contract bytes included in chunks even if they are unused, plus bytes used up by the hashes of the multiproof for those chunks:

overheadBytes = \frac{chunkBytes + hashesBytes - executedBytes}{executedBytes}

Assuming 32-byte hashes and a binary trie, this is the cumulative overhead for various chunk sizes:

Chunk size in bytes

Cumulative overhead

(TABLE DELETED)

A few observations are in order:

- We are addressing exclusively the byte size outcome for a witness using a given chunking strategy, not other costs (computational, storage, etc). Still, this hints towards big size reductions to be had by using small chunk sizes.
- For the smaller chunk sizes, it probably makes little sense to use 32-byte hashes. Using a faster, smaller hash would yield better speed and even smaller overheads.

# The tools

## An opcode tracer based on TurboGeth

We created a TurboGeth-based tracer in Go, which is capable of extracting any trace data from the execution of transactions in the EVM. As of this writing, TurboGeth can process over 28 blocks per second (on an AWS i3.xlarge-type instance); our tracer demonstrates how to process trace data while it's being gathered and extract it with minimal slow-down, depending mainly on the quantity of data to be exported. This means that one can potentially extract information for 1M blocks in ~12 hours, or the whole history of Mainnet in less than 1 week

.

The careful interfacing with TurboGeth also means that, if TurboGeth keeps getting faster, our tracer will automatically get faster too

As a concrete example, the detection of contiguous code segments (AKA basic blocks) run by the EVM and the exporting of this data as JSON files recording a variety of metadata is done at over 20 blocks/sec.

This speed, and the fact that part of the process is done right at the source, alleviates the typical need to collect extra data "just in case". This allows further steps to be more streamlined, and greatly improves the agility to iterate in research projects.

We expect the code to be useful as is, but it's also a good entry point for other projects to customize the kind of processing that is done and the data that is exported. The tracer has already been merged into TurboGeth.

More generally, we hope that this will allow for faster iteration of Ethereum research projects and reproducibility of their results by other teams, since it reduces the barrier to entry that was caused by the slowness of data gathering of previous approaches and the consequent need to store enormous quantities of data for intermediate steps.

And of course, we hope you'll double-check our results

## A script to merklize Mainnet transaction traces

We wrote a tool that takes the opcode tracer output and a chunk size, and calculates the per-block merklization code witnesses, gathering the resulting statistics at the transaction, block, batch and global level. These statistics were key to get an understanding of the surprising runtime characteristics of real world transactions.

An example of output for 32-byte chunks:

```
Block 9904995: segs=816 median=14 2▁▂▁▂▃▄▅▁▁▂▁▂26 (+29% more) Block 9904995: overhead=42.4%
exec=12K merklization= 16.6 K = 15.6 + 1.0 K Block 9904996: segs=8177 median=18 2▁▂▁▃▅▃▄▂▃▂▂▁▂▂34
(+24% more) Block 9904996: overhead=34.1% exec=101K merklization= 136.0 K = 135.1 + 0.9 K Block 9904997:
segs=14765 median=14 2▁▂▂▃▅▂▄▂▂▂▁▂▁26 (+25% more) Block 9904997: overhead=31.5% exec=106K
merklization= 139.8 K = 139.0 + 0.8 K Block 9904998: segs=20107 median=14 2▁▁▂▃▄▃▅▂▂▂▁▂26 (+27% more)
Block 9904998: overhead=36.4% exec=76K merklization= 103.8 K = 103.0 + 0.8 K Block 9904999: segs=10617 median=12
2▁▂▃▄▄▅▂▂▂22 (+29% more) Block 9904999: overhead=37.4% exec=59K merklization= 81.4 K = 80.8 + 0.7 K file
/data/traces2/segments-9904000.json.gz: overhead=32.1% exec=74493K merklization=98368.4K = 97481.3 K chunks +
887.1 K hashes segment sizes:median=14 2▁▂▅▆▆▅▂▂▂▂26 (+28% more) running total: blocks=905000
overhead=30.8% exec=65191590K merklization=85278335.0K = 84535640.5 K chunks + 742694.5 K hashes estimated
median:15.0
```

The script assumes a plain Merkle tree, with configurable arity and hash sizes, but no special nodes as those defined in the Yellow Paper (branch, extension, etc).

Looking forward to see what the community does with these tools!

# Acknowledgements

Big thank you to @AlexeyAkhunov and the rest of TurboGeth team

for their vision and hard work, without which any of this wouldn’t have been possible!

Also:

O. Tange (2011): GNU Parallel - The Command-Line Power Tool, ;login: The USENIX Magazine, February 2011:42-47.