Verkle Trees

Impact study on existing smart contracts

Aug 10, 2021

Client: Ethereum Foundation

Introduction

Dedaub was commissioned by the Ethereum Foundation to investigate the impact of Vitalik Buterin's Verkle tree gas metering proposal on existing smart contracts. In order to appraise the impact of the proposed change, we performed extensive simulations of the proposed changes over past transactions; wrote a static analysis and applied it over most contracts deployed to the mainnet; examined bytecode, source, decompiled code, and low level traces of past transactions.

Vitalik Buterin's proposal introduces new gas changes to state access operations that closely reflect the worst case witness size that needs to be downloaded by a stateless client that needs to validate a block. As described in the original proposal, a Verkle tree is a cryptographically secure data structure with a high branching factor. Unlike Merkle trees, trace witnesses in Verkle trees do not need to include the siblings of each node in a path to the root node, which means that trees with a high branching factor are more efficient. Verkle trees can still retain the security properties by making use of a polynomial commitment scheme. By leveraging these properties, gas costs can therefore more easily reflect the upper bound of the cost of transmitting these witnesses across stateless clients, assuming these are appropriately designed.

In addition to gas changes that the proposal introduces, the Verkle tree proposal may break protocols that are too tied to the existing implementation, such as protocols that use Keydonix Oracles like Rune, but this consideration is outside the scope of this report.

In the report we briefly summarize the proposal with respect to gas costs. We then delve into the impact to existing contracts using two techniques: path-sensitive static program analysis, and dynamic analysis by modifying an Erigon node with the new gas semantics and analyzing the data per internal transaction. Finally we state our recommendations for lessening the impact of this proposal.

Background

The goal of the proposal [1] aims to make Ethereum *stateless clients* sustainable in terms of data transmission requirements.

Stateless clients should be able to verify the correctness of any individual block without any extra information except for the block's header and a small file, called *witness*, that contains the portion of the state accessed by the block along with

proofs of correctness. Witnesses can be produced by any state-holding node in the network. The benefits of stateless verification include allowing clients to run in low-disk-space environments, enabling semi-light-client setups where clients trust blocks by default but stand ready to verify any specific block in the case of an alarm, and secure sharding setups where clients jump between shards frequently without the need to keep up with the state of all shards.

Large witness sizes are the key problem for enabling stateless clients for Ethereum and the adoption of Verkle trees can reduce the witness sizes needed. More specifically, a witness accessing an account in the hexary Patricia tree is, in the average case, close to 3 kB, and in the worst case it may be three times larger. Assuming a worst case of 6000 accesses per block (15m gas / 2500 gas per access), this corresponds to a witness size of ~18 MB, which is too large to safely broadcast through a p2p network within a 12-second slot. Verkle trees reduce witness sizes to ~200 bytes per account in the average case, allowing stateless client witnesses to be acceptably small.

Finally, contract code needs to be included in a witness. A contract's code can contain up to 24000 bytes, and so a 2600 gas CALL can add ~24200 bytes to the witness size. This implies a worst-case witness size of over 100 MB. The proposal suggests breaking up contract code into chunks that can be proven separately; this can be done simultaneously with a move to a Verkle tree. Since a contract's code can contain up to 24000 bytes, and so a 2600 gas CALL can add ~24200 bytes to the witness size. This implies a worst-case witness size of over 100 MB. The solution is to move away from storing code as a single monolithic hash, and instead break it up into chunks that can be proven separately and adding gas rules[1] that accounts for these costs[2].

The small witness sizes described above are possible because Verkle trees rely on an efficient cryptographic commitment scheme, called *Polynomial Commitments*[3] [4]. Polynomial Commitments allow for logarithmic-sized (in respect to the tree's height) proof-of-inclusion of any number of leaves in a subtree – and this is independent to the branching factor of the tree. In comparison, traditional Merkle tree's proofs

1 https://notes.ethereum.org/@vbuterin/witness_gas_cost_2

2 https://notes.ethereum.org/@vbuterin/code_chunk_gas_cost

3 https://www.iacr.org/archive/asiacrypt2010/6477178/6477178.pdf

4 https://dankradfeist.de/ethereum/2020/06/16/kate-polynomial-commitments.html

depend on the branching factor of the tree in a linear fashion, since all siblings of a node-to-be-proved must be included in the proof.

Summary of Gas Changes

The current proposal suggests new gas costs which have no counterpart in the previous versions of Ethereum. These is a gas cost for every chunk of 31 bytes of bytecode which are accessed and some additional access events that apply to the entire transaction. An improvement in gas cost can however be also achieved by lowering the cost for SLOAD/SSTORE for when fewer subtrees of the tree structure underpinning the state are accessed. The following is a summary of the gas cost changes between EIP-2929, which introduces the notion of “access lists” and the current proposal.

SLOAD, but location previously accessed within the same tx using SLOAD or SSTORE

Verkle	`WARM_STORAGE_READ_COST (100)`
EIP-2929	`WARM_STORAGE_READ_COST (100)`

SLOAD, but location previously unread within the tx using SLOAD or SSTORE

Verkle	200 if previously visited subtree 2100 if subtree has not been visited (storage location is up to 64 / 256 elements away from the closest visited)
EIP-2929	`COLD_SLOAD_COST = 2100`

CALL an address for an account that has not been previously accessed within the tx

Verkle	Access list costs (typically 1900 + 200 + 200 + more if value bearing)
EIP-2929	`COLD_ACCOUNT_ACCESS_COST = 2600`

EIP-2929 mentions that precompiles are not charged `COLD_ACCOUNT_ACCESS_COST` but the Verkle specification omits this case at the time of writing.

CALL an address for an account that has been previously accessed within the tx

Verkle	`WARM_STORAGE_READ_COST = 100` (but this should be more if the previous cost was not value bearing or didn't access the balance and this one is)
`EIP-2929`	`WARM_STORAGE_READ_COST = 100`

SSTORE, but location previously accessed within the tx using SLOAD or SSTORE

Verkle	20000 if setting a location "from 0 to 1" 2900 if setting a location "from 1 to x**" ** refunds will be abolished in a separate EIP
EIP-2929	Similar to above

SSTORE, but location not previously accessed within the tx using SLOAD or SSTORE

Verkle	22100 or 20200* if setting a location "from 0 to 1" 5000 or 3100* if setting a location "from 1 to x**" * final cost depends on whether the subtree was previously accessed ** refunds will be abolished in a separate EIP
EIP-2929	22100 if setting a location "from 0 to 1" 5000 if setting a location "from 1 to 2"

In addition, the Verkle gas cost proposal also adds:
1) Code chunking costs, which have no counterpart in EIP-2929: there is a cost for accessing every 31-byte "chunk" of contract bytecode;
2) Additional access events per transaction.

At a high level, the reader should expect that the gas cost impact of Verkle trees will have two components:

- Bounded changes, where constant factors are slightly different, per above;
- Unbounded changes, especially due to code chunking. Code locality will become important for gas costs. There will be (many) instances of executing

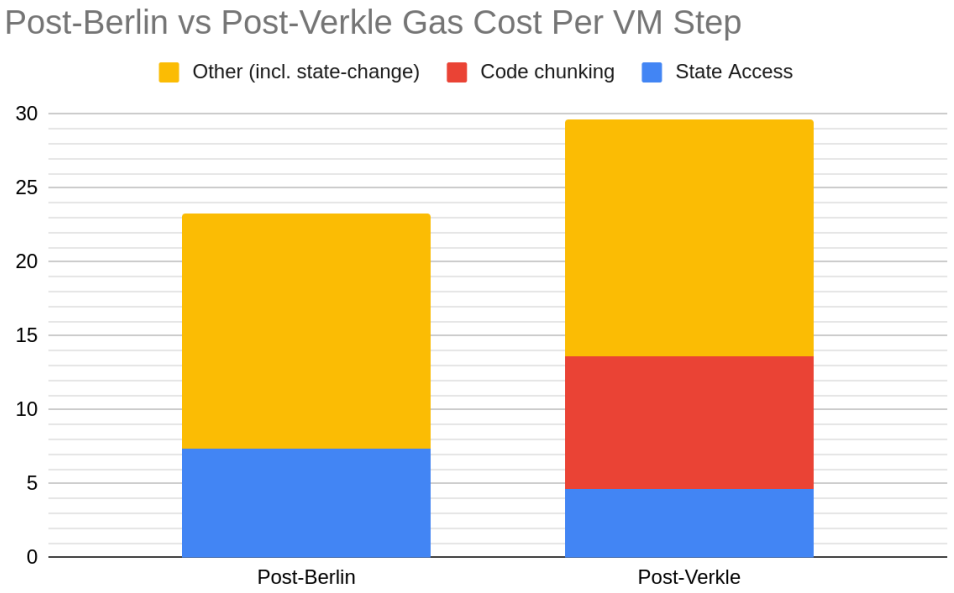
code that would incur zero cost before the Verkle trees proposal, yet have a cost after it, merely for access to the code itself.

Insights

The net gas cost increase of the proposed changes on current contracts is non-negligible, at around 26% on average, a number of pathological cases in existing DeFi projects, and 97% of internal transactions suffering higher gas costs. The impact is, however, manageable, especially if smart contract developers upgrade some of their contracts to optimize for the new cost model (e.g., avoid splitting public functions in different chunks) and optimizing compilers are developed to facilitate. The latter will likely be a gradual process: throughout software development history, tools have evolved at a slow pace and their adoption has been even slower. For instance, the majority of contracts that have been deployed in the first 40k blocks since block 12.9m still use older compiler versions, which the following table illustrates.

Compiler	solc 0.8	solc 0.6	solc 0.5	solc 0.7	other
# of contracts	6024	6003	2333	1946	660

If we drill down on the additional gas costs (per instruction executed), the following graph depicts this difference due to the different mechanisms in this proposal. This assumes no changes are made to a typical contract's code by the time this proposal is deployed.



As expected, the above graph illustrates that storage state access costs per VMStep (e.g., `COLD_*_COST` introduced by EIP-2929) will be significantly reduced (Blue), but there is also an even significantly higher increase in cost due to code chunking (Red). There are many reasons for why this has a significant effect on some contracts but not others, which we describe later, but overall we expect state access cost to go down further and code chunking costs to be reduced too. For instance, the former can be achieved by packing locations that are frequently used together (e.g., different kinds of balances) closer together.

The current Solidity and Vyper compilers were not designed to improve the locality of bytecode. Rather, these compilers reduce code size by removing common parts of a contract and reusing it in many different public functions. This is an unfortunate optimization for these new gas costs, because in many cases the reuse of this code will not be felt (typically transactions would involve one public function per contract). On the other hand, the compiled code now may need to "jump" too "far" from the rest of the code of the currently executing function, which can result in pathological behavior for library functions, getters, or other pure functions, leading to a possible cost increase by an order of magnitude.

Commonly used Vyper compiler versions generate worse function selector code (in terms of locality), compared to Solidity. Modern Solidity versions perform a binary search, whereas Vyper performs a linear search. Vyper functions, especially functions that are invoked at a deeper nesting level in function selectors, are more susceptible to additional costs. In addition, language constructs to reuse contract code are

absent. During our study we found many transactions (multiple per block) that call different addresses which reuse exactly the same code. The current proposal may optimize this case a little better. On the other hand, some of these contracts are small proxies, that would easily fit within the first 128 chunks, which is why the reduction in gas chunking cost would only be 3% with such an optimization.

Ultimately, we think that newly compiled code after deployment of this proposal with a well-designed optimizer, borrowing well-known techniques from compilation to resource constrained hardware, would not incur significant cost.

Static Analysis Experiments

As part of this study, we developed a static analysis tool to predict which code chunks will be accessed for every function in a smart contract, which can be applied to every contract deployed on the mainnet. This problem was phrased as a path-sensitive "code reachability" problem, running on top of the gigahorse-toolchain binary code lifter. An approximate upper bound of the additional gas cost of calling a function under the proposed gas cost changes is inferred. In order to minimize false positives, only inferences of successful execution paths (ending at STOP or RETURN) were considered.

The static analysis was applied to contracts that have transacted over the two weeks prior to the beginning of the study. The analysis is "path sensitive", meaning that it will consider each path of execution separately. For instance, in the function selector code, only the path that leads to the particular function with the corresponding signature will be considered, in order to more accurately approximate the additional gas incurred. There were some modifications to the contract-library.com infrastructure and related toolchains that were required for us to write this analysis:

a) Keep track of mapping an statement in the original bytecode to the Gigahorse IR.
b) For every basic block, keep track of the chunk ids that are accessed by it (e.g., by CODECOPY).
c) Add functionality to contract-library.com to handle analyses that return non-vulnerability information for many different parts of a smart contract.

The results of the static analysis can be queried through the contract-library api, e.g.,

https://contract-library.com/api/Ethereum/7a250d5630B4cF539739dF2C5dAcb4c659F2488D/code_chunk_cost

The results of the static analyses were also queried and interesting cases have been inspected.

Queries conducted: Impact of Chunking with Calls that have Fixed gas budgets

```
select hex(a.address), hex(td.selector), td.gas, cpv.int_value, hex(td.transaction_hash)
from ethereum.TransactionDetail td
straight_join address a on td.to_a = a.address
join code_proto_vulnerability cpv on a.md5_bytecode = cpv.md5_bytecode
where td.gas % 100 = 0 and td.vmsteps > 0 and a.network = "Ethereum" and cpv.selector = td.selector
and td.gas / cpv.int_value < 3 and td.block_number > 12895000
limit 1000;
```

Explanation: Find recent function calls that were passed a fixed amount of gas and for which the inferred additional cost of code chunking is significant compared to this gas budget.

These queries returned interesting cases of additional costs to proxies and fallbacks, including:

Breaking?	Description	Address
No	Uniswap V3 proxy	TODO
Will not result in additional gas cost as the same code would have been executed within a previous internal transaction.		

Breaking?	Description	Address
Unlikely	1inch exchange proxy	TODO
Will result in an out-of-gas error if called with 2300 gas but only when the computation would have otherwise reverted.		

It was however found out that many of the high-profile contracts will still work as the code-chunking costs would have been amortized away in previous internal transactions.

Some fallback functions will however still break, and this was also confirmed by looking at past transactions. One example is an unknown contract that acts like a Deposit aggregator: **0xBB44E3349C23CC430CAE6EBBAF0256C9F2A1872F.**

Another example of a breaking contract is a multi-sig wallet developed by Consensys. This is deployed at a number of addresses, including 0xAB613544D53173E65E75A2390F7512F6FE941257 and 0xCB102CBFAD94D71596D7D9172072DA8B86E60FBD.

We also found a currently active crowdsale contract that would break under these gas changes: 0xeDB29640F0B793c2A8DEcB5C4747BEa90580BC51.

Given that developers will have ample time to migrate contracts, and that Solidity's transfer is no longer the recommended way to send ETH, we believe that the impact to fallback functions is relatively low.

Dynamic analysis experiments

We have modified an Erigon client to analyze past transactions under the proposed gas semantics. This dynamic analysis, which compares the gas consumed by cold storage accesses, code chunks, was as faithful to the EIP-2929 and Verkle specification as possible. Unlike previous analyses, this dynamic analysis also:

- Considers the cost of moving between subtrees and correct costs for accessing each leaf (200)
- Compares this cost against EIP-2929's `COLD_ACCOUNT_ACCESS_COST` and EIP-2929's `COLD_SLOAD_COST`
- Considers the different costs of `SLOAD`s and `SSTORE`s under each specification.
- **Performs this analysis and reports gas information per internal transaction so that individual contracts and functions within them can be identified**

Information on gas usage of every internal transaction was computed and saved to a relational database to facilitate further querying. Queries on this database aimed to find the impact of code chunking on:

- Calls that have low or fixed gas budgets
- Extreme cases, by sorting internal transactions against the relative increase or decrease in gas consumed
- Average cases and other interesting exploratory queries

```
select hex(to_a), ifnull(signature, hex(selector)) as signature, hex(transaction_hash),
vmstep_start, ratio_gas_verkle, total_transactions from (
  select distinct
     FIRST_VALUE(td.to_a) OVER w as to_a,
     FIRST_VALUE(td.selector) OVER w as selector,
     FIRST_VALUE(ss.signature) OVER w as signature,
     FIRST_VALUE(td.transaction_hash) OVER w as transaction_hash,
     FIRST_VALUE(td.vmstep_start) OVER w as vmstep_start,
     FIRST_VALUE(gas_verkle_code / gd.gas_used) OVER w as ratio_gas_verkle,
     ROW_NUMBER() OVER w as total_transactions
  from GasDetail gd
  join TransactionDetail td on gd.block_number = td.block_number
                           and gd.transaction_hash = td.transaction_hash
                           and gd.vmstep_start = td.vmstep_start
  left join gigahorse.function_selector_signature ss on ss.selector = td.selector
  where gd.block_number > 12900000 and gd.failed = 0 and gd.gas_used > 1000
  and gas_verkle_code / gd.gas_used > 2
  window w as (partition by td.to_a, td.selector order by gas_verkle_code / gd.gas_used desc)
) temp
where total_transactions = 3;
```

Example Query, explanation: Find functions (and representative transactions) that would be severely negatively affected by the gas code chunking semantics that have at least 3 negatively affected txs since block 12.9m, and that would otherwise consume more than 1k unit of gas.

Impacted Projects and Functions

We queried the fine-grained data gathered by the modified Erigon client and combined it with existing data from our internal blockchain explorer database. We could therefore find some extreme cases, where the proposed gas metering mechanism significantly increases the gas requirements. Conversely, we could also find transactions with significant positive impact. This report documents some of these cases, however a larger list is available here.

Overall, the most extreme negative cost consequence seems to be limited to getters of immutable fields in some versions of Solidity, libraries, and other pure functions. Interestingly, Vyper code seems to be even more susceptible to edge cases. In the rest of the section, we shall look at examples of these. In each case, we will refer to individual functions in individual projects, and when we give an example of an internal transaction we use a key `[tx_hash, vmstep_start]` to uniquely refer to such a transaction. The `vmstep` is a counter that starts from 0 when executing a transaction and is incremented for each opcode that is executed within that transaction, the latter uniquely identified by the transaction hash (`tx_hash`).

In some cases, we will also include a *compacted trace representation*, decompiled code, or source. A compacted trace representation omits push, pop, swap and dup instructions and all notions of the stack. For example, the following line in this trace:

```
1123: 0xf: 0xad5c464800000000000000000000000000000000000000000000000000000000 = CALLDATALOAD(0)
```

Illustrates that at the 1123rd EVM execution step, the instruction at program counter `0xf` is a `CALLDATALOAD`, with an argument of `0`. The return value of this instruction is also indicated.

Examples of Negatively Impacted Contracts

The following examples are pathological cases were the cost of code chunking as proposed would yield an order of magnitude increase in gas costs from some functions.

Description	Project Name
Getters in Uniswap V2	Uniswap V2

The `WETH()` and `factory()` getters are very simple, however due to the generated code layout by the Solidity compiler, the gas consumed by the code chunking mechanism is an order of magnitude higher than the gas required to execute these getters. See example transaction:
`[0x0E966585E03B6FCCE274F4F5A1048320174D30D28859342673DCE020E96F2717, 1114]`

The compacted trace produced for this internal transaction is annotated as follows:

```
1116: 0x4: MSTORE(0x40, 0x80)
1118: 0x7: 4 = CALLDATASIZE()
1119: 0x8: 0 = LT(4, 4)
1121: 0xc: JUMPI(0x18f, 0)
1123: 0xf: 0xad5c4648000000000000000000000000000000000000000000000000000000000 = CALLDATALOAD(0)
1125: 0x12: 0xad5c4648 = SHR(0xe0, 0xad5c4648000000000000000000000000000000000000000000000000000000000)
1128: 0x19: 0 = GT(0x8803dbee, 0xad5c4648)
1130: 0x1d: JUMPI(0xd6, 0)
1133: 0x24: 1 = GT(0xc45a0155, 0xad5c4648)
1135: 0x28: JUMPI(0x7f, 1)
1136: 0x7f: JUMPDEST()
1139: 0x86: 1 = GT(0xaf2979eb, 0xad5c4648)
1141: 0x8a: JUMPI(0xb0, 1)
1142: 0xb0: JUMPDEST()
1145: 0xb7: 0 = EQ(0x8803dbee, 0xad5c4648)
1147: 0xbb: JUMPI(0x8af, 0)
1150: 0xc2: 1 = EQ(0xad5c4648, 0xad5c4648)
1152: 0xc6: JUMPI(0x954, 1)
// Function selector ends here
1153: 0x954: JUMPDEST()
1154: 0x955: 0 = CALLVALUE()
1156: 0x957: 1 = ISZERO(0)
1158: 0x95b: JUMPI(0x960, 1)
1159: 0x960: JUMPDEST()
1163: 0x968: JUMP(0x2671)
// The solidity compiler reused the !payable check, which ends here
1164: 0x2671: JUMPDEST()
1167: 0x2694: JUMP(0x969)
// The jump to the previous basic block was a superfluous artifact of the Solidity compiler (it reuses common instruction
sequences)
1168: 0x969: JUMPDEST()
1171: 0x96d: 0x80 = MLOAD(0x40)
1175: 0x985: 0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2 = AND(0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2,
0xffffffffffffffffffffffffffffffffffffffff)
1177: 0x987: MSTORE(0x80, 0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2)
1178: 0x988: 0x80 = MLOAD(0x40)
1182: 0x98c: 0 = SUB(0x80, 0x80)
1184: 0x98f: 0x20 = ADD(0x20, 0)
1186: 0x991: RETURN(0x80, 0x20)
```

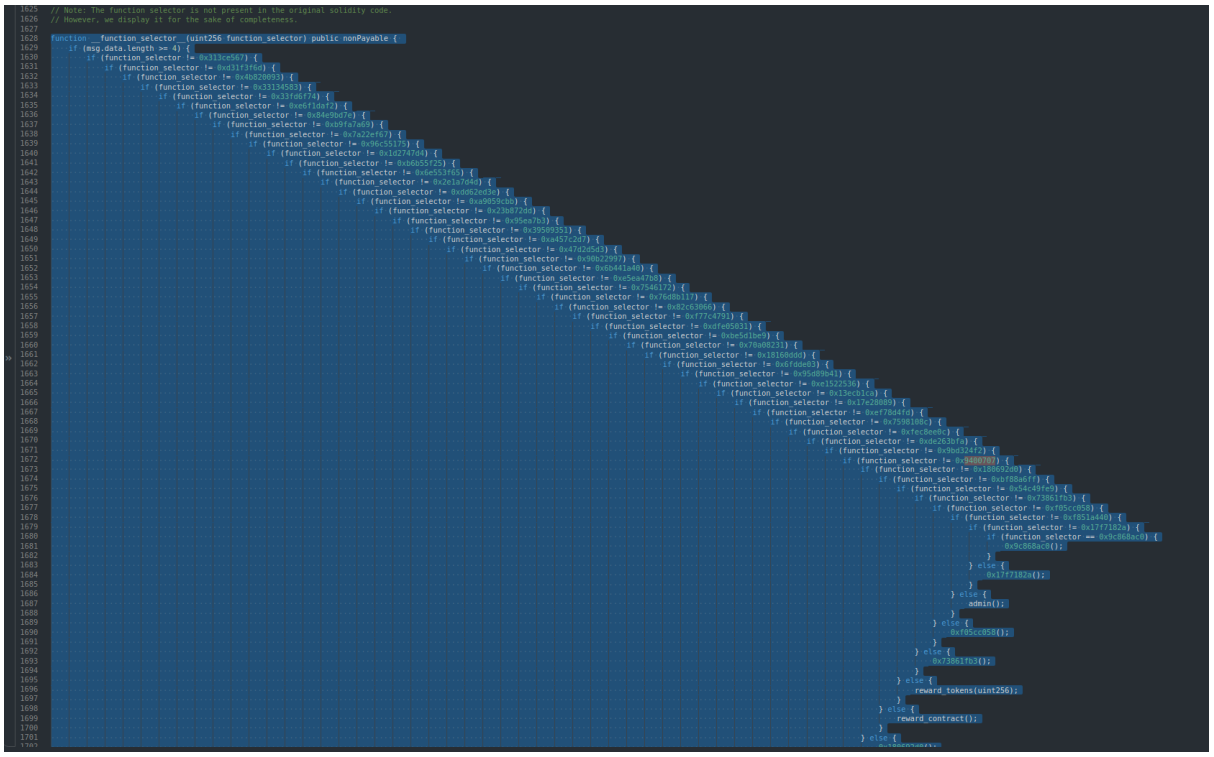
Description	Project Name
Uniswap V2 Libraries	Uniswap V2

Pure functions that have been compiled with suboptimal code paths can see the gas consumed relatively increased. For instance, `getAmountOut(uint256,uint256,uint256)` suffers from suboptimal code reuse (lowering the code size, but increasing jumps to many parts of the contract) and, since this is a library, the amount of gas consumed is small to begin with. The use of various other libraries, such as safeMath, laid out in different parts of the contract adds to the issue. See example transaction: `[0xDAB4D2C8E13AAF41C4F3F1815FEFF64F7E575E3A64E52202D3029BCD601BCBB3, 22299]`	

Description	Project Name
AAVE getters	AAVE V2
`01A3688D7D01390677E85256406B3156ACD59C64.getPriceInToken()`	

Description	Project Name
Curve Gauge Function Selector	Curve

The following Vyper code was decompiled at contract-library. Any function called within the most nested conditions of the function selector code incurs a heavy penalty since the function selector code is also scattered throughout the contract's code space.



Each line in the above decompiled code checks whether the function selector matches a fixed entry. **If not, the control flow jumps further away, which wastes the rest of the loaded code chunk.**

Examples of Positively Impacted Contracts

As expected, code that has been positively impacted would be getters of large tuples which didn't incur large code chunking overheads. This is because tuples are stored

over adjacent storage locations. Other example functions are ones that linearly search over an array or many values.

Description	Project Name
Synthetix multi-getter	Synthetix

The following is a decompiled getter from Synthetix

```
function getEntryAt(address varg0, bytes32 varg1, uint256 varg2) public nonPayable {
    assert(varg2 < _getLengthOfEntries[varg0]);
    v0 = (varg2 << 3) + keccak256(keccak256(varg1, keccak256(varg0, 3)));
    return STORAGE[0 + v0], STORAGE[1 + v0], STORAGE[2 + v0], STORAGE[3 + v0],
           STORAGE[4 + v0], STORAGE[5 + v0], STORAGE[6 + v0], STORAGE[7 + v0];
}
```

Note that the getter returns multiple adjacent storage locations.

Description	Project Name
Compound Open Oracle	Compound

This contract, mostly used in Sashimiswap, developed by Compound compares a large number of storage variables to some input, in an inefficient manner. Since many of these storage locations are adjacent to each other, this will result in gas savings overall.

```
function getSLTokenIndex(address slToken) internal view returns (uint) {
       if (slToken == slToken00) return 0;
       if (slToken == slToken01) return 1;
       if (slToken == slToken02) return 2;
       if (slToken == slToken03) return 3;
       if (slToken == slToken04) return 4;
       if (slToken == slToken05) return 5;
       // ... etc ...
       if (slToken == slToken27) return 27;
```

```
        if (slToken == slToken28) return 28;
        if (slToken == slToken29) return 29;

        return uint(-1);
    }
```

Description	Project Name
Options token parameters (OToken)	Opyn

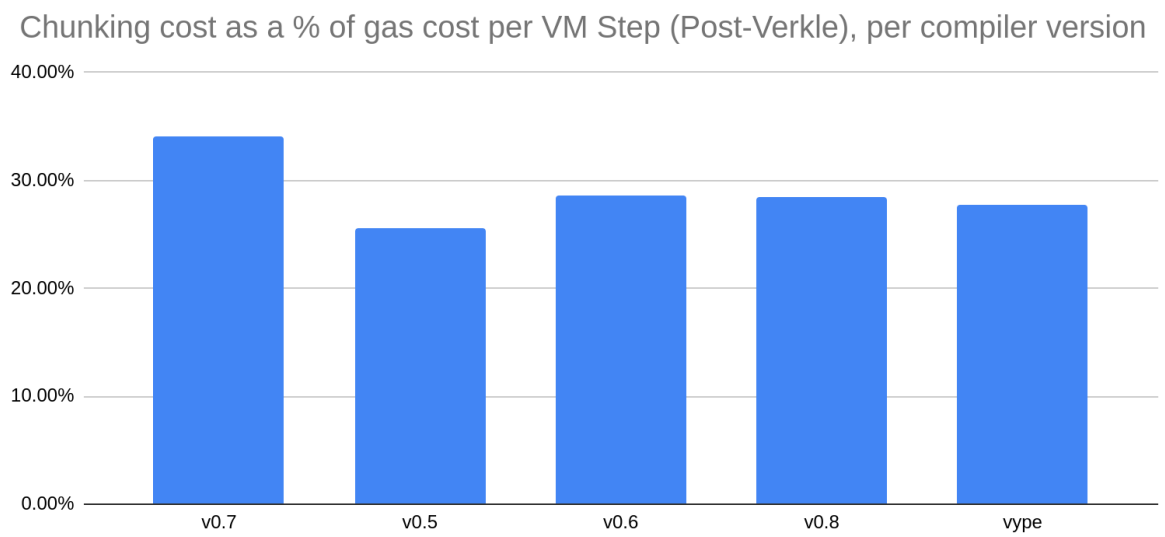
The following decompiled code shows a getter that returns adjacent storage locations.

```
function 0xaf0968fc() public payable {
    return _collateralAsset, _underlyingAsset, _strikeAsset,
           _strikePrice, _expiryTimestamp, _isPut;
}
```

Further Empirical Analyses

Further experiments were performed to statistically analyze the gathered data from past executions. Full insights are available in a separate repository here. These insights have established the average gas consumed in each VM step, in graph 3. In summary, **96% of internal transactions will be worse off**, and the average **additional cost consumed is close to 26%**. By partitioning the data into bins, we also saw that there are no surprises or spikes and that the data forms a relatively normal distribution, meaning that the number of extreme cases of transactions benefitting or negatively impacted is small. Finally graph 8 analyzes the impact of a proposed optimization to the access events that was communicated to the client, by indexing the verkle tree by the hash of the bytecode of the contract, instead of address for code chunks higher than 127. The benefit is modest, and only reduces 3% of the code chunking costs.

In addition, we examined the gathered statistics against specific compiler versions, to assess whether some versions were more susceptible to the new code chunking semantics, and the results indicate that Solidity 0.7 may be possibly worse off. In addition, Enabling today's optimizations results in overall gas savings with post-berlin semantics but will result in no benefit or loss for post-verkle semantics.



Dedaub's Recommendations

Our recommendations for this proposal are mostly intended towards smart contract developers and software tool developers (language designers and compiler developers). The Ethereum Foundation can, however, further consider the following options while the proposal is still being discussed:

- Increasing the Verkle tree's branching factor further, to capture a larger part of the code chunks within a single subtree.
- Optimizing the case for multiple contracts called within the same transaction with the same code, by emitting state access events of the form (codehash, (chunk_id - 128) // 256, (chunk_id - 128) % 128) for code chunks above 127.
- Decreasing the chunking gas cost to 100 for the first few months.

Recommendations for Developers

The recommendations for those developing code today are limited. Obviously, these recommendations must be evaluated against other security and code evolution

considerations. For instance, optimizing for proposed gas cost might be a bad idea, especially since the underlying platform will keep evolving.

Group frequently accessed storage locations by access. One way to achieve this is to fuse together data structures that are indexed by the same value. For instance, if a specialized token-like contract keeps different kinds of balances for each address, do not store each one of these in a separate map. It will be more optimal to store these in a struct in one mapping.

Do not initialize multiple copies of the same contract. Since code access costs are only paid once, these can be amortized by reusing the same contract and, rather than calling it directly, calling a proxy that `delegatecall`s into it.

Recommendations for Compiler Developers

The recommendations for those designing the next compiler or version of the language are more substantial. Unfortunately some of these recommendations also make it hard to reverse engineer contracts, or even analyze contracts at the bytecode level and may hamper exercises such as this one in the future :)

Develop higher level primitives and pack these within the first 128 chunks. High-level primitives such as “array append” or “safe add” can be compiled once to fit within a 31-byte chunk and reused across the entire contract. This has to be packed within the first subtree, however, in order to not introduce additional costs. By “calling” these primitives, one will incur the cost of two jumps (20), but this is an order of magnitude less than loading a new chunk (200).

Partition large functions into one subtree. Larger functions that do not fit within the first 128 blocks should still fit within a single subtree (although reusing functionality within the first 128 blocks shouldn’t be an issue).

Code within a “hot” path should be packed closer together. Thus, error-handling code, e.g., branches that lead to a revert can be placed further away from core function logic. This maximizes the usefulness of the code loaded in each chunk. The Vyper-compiled code that we inspected does the opposite for function selectors :(

Introduce better language features for using proxies. As things currently stand, Soldity's "new" keyword does not reuse implementations. Given the additional gas costs of deploying separate contracts, facilitating the deployment of proxies should be considered.

Optimize function selector code, especially. Function selector code currently consumes very little gas since it does not invoke expensive state-changing or loading operations. This proposal will, however, make function selector code at least an order of magnitude more expensive.

Conclusion

We studied the impact of Verkle trees on existing contracts. The impact is not negligible, with a 26% gas cost increase on average. Some of these projects affected, such as Curve, are hard to upgrade - but they will keep working. However a) the gas cost increases can take place more gradually, b) additional optimizations to the Verkle tree structure can lessen these costs, and c) optimizing compilers and code redesigns can lessen the cost dramatically. Well-designed contracts that take into consideration code chunking costs will potentially end up benefiting from these changes. As a result, we rate the impact of Verkle trees as "significant but manageable", yet acknowledge that the results are subject to interpretation.

Finally it is understood that the impact to current contracts will be gauged against the impact to the security and scalability of the consensus of the network, and Verkle trees could be a game-changer in this regard.

Disclaimer

The inspected contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The study is performed on a best-effort basis: Dedaub offers no promise or guarantee as to the accuracy of the findings, either in terms of completeness or in terms of precision. There may be contracts impacted that are missing from the report, as well as contracts included that, due to semantic complexities, are unaffected.

About Dedaub

Dedaub offers technology and auditing services for smart contract security. The founders, Neville Grech and Yannis Smaragdakis, are top researchers in program analysis. Dedaub's smart contract technology is demonstrated in the contract-library.com service, which decompiles and performs security analyses on the full Ethereum blockchain.