Towards Efficient Hashing in Ethereum Smart Contracts

Emanuel Onica and Cosmin-Ionuţ Schifirneţ
Faculty of Computer Science, Alexandru Ioan Cuza University of Iaşi, Romania

Keywords: Blockchain, Hashing Algorithms, Ethereum.

Abstract: Ethereum is a popular public blockchain platform and currently the most significant featuring smart contract functionality. Smart contracts are small programs that are executed on the blockchain nodes, which can be used to implement complex transaction logic. Several high-level programming languages are available for writing Ethereum smart contracts, the most used being Solidity. The high-level code is further translated into a bytecode executed by a dedicated runtime environment, the Ethereum Virtual Machine (EVM). A few operations are, however, externalized as precompiled contracts, and run by the native implementation of the Ethereum node. These are typically computationally intensive operations such as cryptographic hash functions. Various smart contract patterns require hash computations. In such contexts, the current hash functions supported by Ethereum have a direct impact in both the performance and cost inflicted on the blockchain users. In this paper we investigate the available options for hashing in smart contracts, we discuss the implications regarding some patterns and we evaluate possible improvements. In particular we focus on the recent Blake family of cryptographic hash functions, which show promising performance results, but has yet limited support in the Ethereum platform.

1 INTRODUCTION

Blockchain networks are a type of distributed architectures, widely used during the last decade in providing a variety of decentralized services. The essential role of the nodes in a blockchain network is to maintain a replicated data structure: the blockchain ledger. Clients of the network submit transactions, which are grouped in blocks, cryptographically linked and added to the ledger. The nodes mutually agree on new blocks using mechanisms that vary depending on the blockchain architecture, a common set of guarantees including immutability and integrity of confirmed transactions. The transactions can be for instance simple payments in a virtual currency. Some blockchain platforms also permit clients to execute more complex transactions in the form of operations over a state maintained as part of the blockchain replicated data. Such operations are grouped in *smart contracts*, essentially small programs executed on the blockchain nodes.

Ethereum (Wood, 2021) is currently the main public blockchain platform that offers support for execution of smart contracts. Smart contracts for Ethereum can be written in several specific high-level languages. The most used in practice is Solidity (Sol, 2021), which provides an object oriented, Turing complete specification. In essence, a transaction corresponds to calling a function in Solidity that modifies the state of the blockchain data. Network clients must pay a fee that depends on the execution and storage cost of the called function. The denomination used for measuring this cost are *gas* units and the gas cost calculation is based on the executed function code. The price to be paid per gas unit is set in the Ethereum native currency and varies depending on the blockchain network usage. This mechanism regulates the processing load on the nodes executing the functions and prevents Denial of Service attacks.

The Solidity language provides a relatively comprehensive API, which can be used in programming complex logic for transaction functions. The smart contract code is compiled into a low-level bytecode, which is run by the Ethereum Virtual Machine (EVM), the core part of an Ethereum node implementation. The EVM essentially operates as a stack machine executing opcodes resulted from the compilation and charging their fees according to gas cost specifications fixed in the Ethereum Yellow Paper (Wood, 2021). However, this does not apply for all operations. Some, which are typically computationally intensive, such as cryptographic hashes, are provided as separate precompiled contracts in the Ethereum node implementations. This makes their execution more efficient, but also their defined cost evaluation more debatable.

Onica, E. and Schifirneţ, C.
Towards Efficient Hashing in Ethereum Smart Contracts.
DOI: 10.5220/0010606606600666
In *Proceedings of the 16th International Conference on Software Technologies (ICSOFT 2021)*, pages 660-666
ISBN: 978-989-758-523-4

Besides their typical use in programming, cryptographic hashes have an important role also in some specific smart contract patterns (e.g., commit-reveal). This and the specific context of Ethereum where each full node executes all confirmed transactions to verify these makes performance a critical factor. Ethereum node implementations currently support four types of hashes: Keccak-256 (Bertoni et al., 2011), SHA-256 (NIST, 2002), RIPEMD-160 (Dobbertin et al., 1996) and Blake2b-512 (Aumasson et al., 2013)[1]. The first three are directly callable from Solidity in a smart contract implementation, although only Keccak-256 has a corresponding EVM opcode and associated cost. SHA-256 and RIPEMD-160 are implemented as precompiled contracts, their cost being defined in the Ethereum Yellow Paper.

The main component of Blake2, the compression part, is also provided as a precompiled contract, although there is no default instruction available in Solidity for executing the complete hash function. Blake3 (O'Connor et al., 2020) is a novel iteration of the Blake hashing algorithm, showing also promising performance results. However, to our knowledge, there is no current support for using Blake3 in Ethereum smart contracts. In this paper we provide an early look over an extension for supporting the complete Blake2 function and we consider also the integration of Blake3 with Geth (Get, 2013), the most popular Ethereum node implementation. We provide a performance comparison between the Blake functions and the other hashes. Our evaluation is performed both in respect to raw CPU performance as well as for *gas* costs as defined in the Ethereum specification, revealing also valuable information on the correspondence between the two.

In Section 2 we provide background details and more information about the implications of enhancing hashing performance in the context of smart contracts. We briefly discuss some implementation details in Section 3. We present our evaluation results in Section 4. Finally, we conclude in Section 5 discussing future steps related to hashing support for the Ethereum platform.

[1]Keccak and Blake2 were both finalists of the SHA-3 NIST competition. Keccak was chosen as a winner due to less similarity with SHA-2 algorithms although showing weaker performance than Blake2.

2 BACKGROUND AND IMPLICATIONS

The native support for Keccak-256 and inclusion of the precompiled SHA-256 and RIPEMD-160 hashes is present since early versions of Ethereum. Addition of the Blake2 hash was only later considered, following the typical Ethereum standards adoption methodology, in EIP-152 (Hess et al., 2016). The addition was motivated by both the enhanced performance of the hash and the interoperability in smart contracts with Zcash (Zca, 2016), another blockchain using Blake2. As mentioned in Section 1, only the Blake2 main component, the compression function, was introduced as a precompiled contract. To complete a hash computation this function must be iteratively called for chunks of 128 bytes of the input followed by a few other operations like padding. We believe this part was left out of the addition for providing flexibility in the implementation depending on need (e.g., offering optional keyed support). However, as we will discuss in Section 4, this can severely impact the costs inflicted by adding the missing part.

Blake2 is also mentioned in another Ethereum standard proposal, EIP-3102 (Ballet and Buterin, 2020), discussing improvements in the storage of the blockchain which uses a Merkle-Patricia tree where hashing is a frequent operation. The proposal considers Blake2 as the preferred choice for a hash function to replace the currently used Keccak-256. It actually mentions Blake3 as an even better potential candidate in terms of performance, but reverts to Blake2 due to lack of Blake3 Go implementations at the time of the proposal. A precompiled contract is nothing more than an implementation of a function integrated natively in the language of the Ethereum node, i.e., Go in the case of a Geth based node. Therefore, integrating a complete Blake2 implementation or adding Blake3 as precompiled contracts would permit the use of the functions also in the various blockchain node operations such as EIP-3102.

The main topic of our paper is, however, the use of hashing in smart contracts, where the integration of functions such as the Blake family can have an impact both in generic programming and in established patterns that require efficient hashing. For instance, we can refer to the *commit-reveal* pattern, used to prevent transaction order dependence attacks as referenced in the Smart Contract Weakness (SWC) registry (SWC, 2020). An attacker that is able to monitor transactions to a smart contract, can issue transactions of its own, leveraging any advantage (e.g., higher payments, ownership of mining nodes) in attempt to overcome the other transactions observed. A typical example is a smart contract offering a reward for solving a problem. An attacker could try to steal an answer seen in other transaction by issuing its own transaction with the same answer, getting it approved before the other, and being rewarded. Another example is where an attacker can try to overcome transactions on financial operations like changes in credits approvals before these take effect. This is a known issue in ERC-20 (Vladimirov and Khovratovich, 2017), the most common token implementation (essentially used for custom currencies implemented through Ethereum smart contracts).

In essence, the commit-reveal pattern implies submitting first a transaction including the hashed version of its content binding it to the sender identity. Only in a second step the submitter would reveal the transaction content, by sending another transaction. This would overcome the attack attempt. However, the smart contract processing the transactions would need to compute verification hashes very often. Therefore, the hashing performance is essential in this pattern.

We believe that adding support for efficient cryptographic hash functions in the Ethereum platform can be of use both in smart contracts as described and also for future blockchain node optimizations. For this purpose we investigate the potential for a complete integration of the Blake family of functions as precompiled contracts, which we discuss in the following sections.

3 INTEGRATION OVERVIEW

In this section we briefly discuss some aspects regarding the possibility of integrating complete support for the Blake2 and Blake3 hash functions in Ethereum. We consider Geth (Get, 2013), an Ethereum node implementation written in Go which is currently used by more than 80 percent of the operating nodes (Eth, 2021), as base for our proposed integration. The integration dependence on the node implementation is due to the precompiled part of the hash. We further provide evaluation on our integration status for these two potential hash function candidates, showing their efficiency in Section 4.

3.1 Blake2 Integration Details

As previously discussed, the main compression function used in the Blake2 hash has been supported in Ethereum following the adoption of EIP-152 (Hess et al., 2016) in the form of a precompiled contract. However, the combining algorithm that iterates the compression function over a larger input and calculates the final hash output was not provided in the EIP-152 (Hess et al., 2016). The existing implementations are scarce and obsolete (e.g., integrally implementing Blake2 in Solidity and not making use of the newer precompiled integration (Consensys, 2017)).

In order to obtain a complete hash implementation of Blake2, one option is to include the rest of the function algorithm as part of its own smart contract that can be invoked by other smart contracts. Being the most straightforward approach, we have completed the Blake2 implementation in this manner using Solidity and low level EVM assembly for performance optimizations. Despite the latter, this approach impacts negatively the cost towards the final user.

As mentioned in Section 1 a fee is charged by summing up the predefined gas cost of all instructions in a smart contract function. Completing the Blake2 implementation involves gas costly operations such as iterating over the data that must be hashed. When issuing a function call corresponding to a transaction, the user must pay a price per gas unit multiplied with the total gas cost of the function. The caveat is that calling a function that does not alter the blockchain state, i.e., just for computing a hash value, is not considered a transaction, and therefore, not charged. However, if such a call is a subcall of an altering function, which is the most probable usage idiom, its cost will be added to the transaction cost.

A second option for a complete Blake2 implementation is to provide this integrally as a precompiled module, which could be called directly in a Solidity smart contract via a single instruction, i.e., similarly to the other supported hashes in Ethereum. This would require, however, establishing a gas cost for the call, as done for the other hashes. Setting this is not a trivial matter, typically being part of a Ethereum standardization process. We discuss more over the gas cost in Section 4, where we evaluate our implementation.

3.2 Blake3 Integration Details

Essentially, the mechanism of Blake3 (O'Connor et al., 2020) splits the hash input in chunks of 1024 bytes, further separated in blocks of up to 64 bytes, each block being compressed using a compression function. Applying the compression function on a block produces a block chaining value. These block chaining values are further used when compressing the other blocks in the chunk to obtain chunk chaining values, which are finally combined using a tree structure for calculating the hash output.

Similarly to the existing integration of Blake2 hash discussed in the previous section, we could consider as precompiled part only the compression function, the most computationally intensive in the Blake3 algorithm. However, adding the combining part in Solidity would lead to similar issues as discussed in the case before, and would require heavy optimizations for obtaining an acceptable gas cost.

Therefore, the option of integrating Blake3 integrally as a precompiled module seems more appealing. Unfortunately, as mentioned also in Section 2, in reference to (Ballet and Buterin, 2020) there are not many implementations of Blake3 available in Go, and none audited for security to our knowledge, at the time of writing of this paper. This makes Blake3 integration with Geth rather problematic. We have considered, however, one of the versions available in the open domain (Champine, 2020) as a potential candidate. A rationale for our chosen variant is that it is currently the most popular Blake3 Go implementation on the Github platform, and that it foresees future optimizations on using several Intel Advanced Vector Extensions (AVX) (Intel, 2015) sets for enhancing performance. We include this implementation in our initial evaluation discussed in the following section.

4 EVALUATION

The first criteria we consider in our evaluation is the effective CPU time cost of computing the result of the hash functions. Since typically transactions in Ethereum do not carry large amounts of data, we focus in particular on small input sizes, up to at most 4096 bytes. We compare the cost of hashing by evaluating the implementations currently available in the Ethereum node implementation Geth (Get, 2013) v1.9.25, for the Keccak-256, SHA-256, RIPEMD-160 and Blake2b-512 hashes as well as an external Blake3-256[2].

We note that the Geth source code references the official implementation for Blake2 provided as part of the Go language packages, despite exposing only the main compression function for use in smart contracts. Therefore, we choose to evaluate the complete hash. For Blake3, we consider the public domain implementation referenced in the previous section (Champine, 2020) with minor adjustments. Because at the date of writing this Blake3 implementation lists some shortcomings that seem to result specifically in no enhancement from Intel AVX assembly extensions on small input sizes, we choose to disable these both in Blake3 as well as in Blake2, using, therefore, a pure Go implementation.

We perform our tests on a machine equipped with an Intel Core i7-4771, 3.5 GHz CPU, 16 GB DDR3 800 Mhz, running Lubuntu 18.04.5 LTS as an operating system. We execute the hash functions over an increasing size of the input, calculating the average hash computation time cost over one million runs. The results are depicted in Figure 1.

We observe that both Blake functions perform better than all other hashes exposed in the Ethereum implementation via Solidity instructions. Interestingly, the Blake2 module provided with the Geth source code proved more efficient than our selected Blake3 alternative, contrary to the results described in (O'Connor et al., 2020). We assume the reason for this resides mainly in the fact that the used Blake2 package is a more mature implementation including optimizations not present yet in our selected Blake3 candidate package. It is also true that Blake3 is particularly fit for larger data sizes showing a constant better hash rate over Blake2 starting from 16KiB with AVX optimizations activated according to official tests (O'Connor et al., 2020).

Another interesting observation is that for smaller data inputs (128 and 256 bytes), which are of particular interest in our context, SHA-256 performs equally well, or even slightly better than Keccak-256. This does not correspond to the gas costs defined in the Ethereum specifications, as we detail further below.

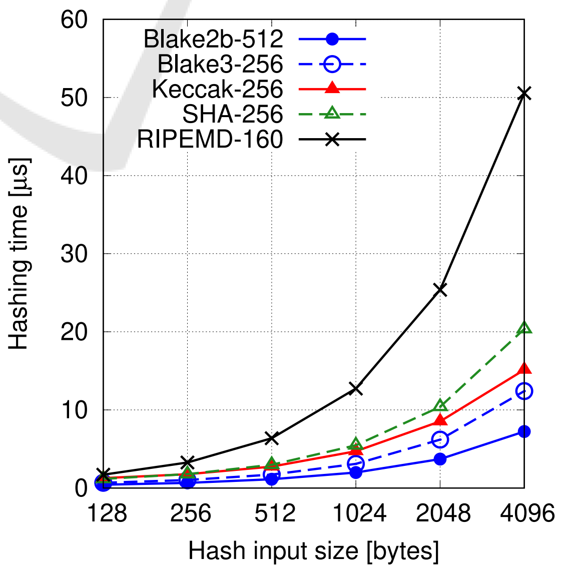

Figure 1: Evaluation of hash computation time cost.

The second criteria we evaluate is the gas cost of the hash functions. This has a direct impact in the price paid by users, and should normally reflect the

[2]The selected output bit size version is the most common for the considered hashes, where multiple options are available, e.g., Blake2b-512 is the version used by the exposed precompiled module.

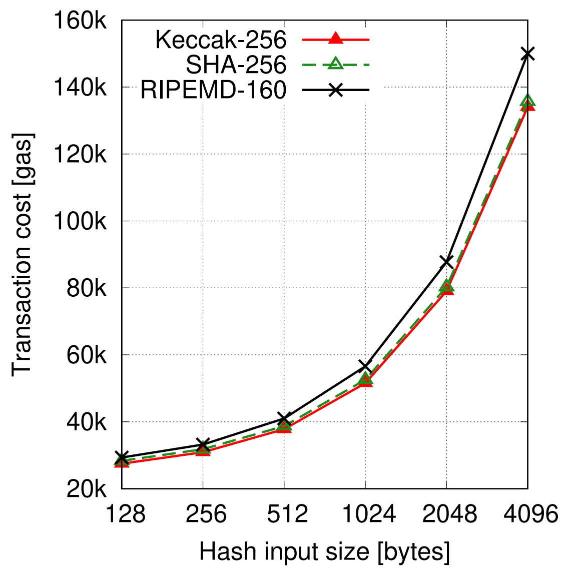

Figure 2: Evaluation of transactions gas cost for Ethereum supported hashes.

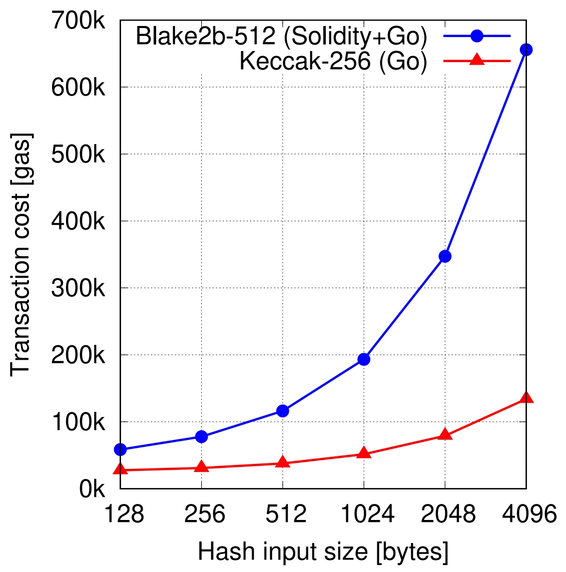

Figure 3: Comparison of transaction gas cost between Keccak-256 and Blake2b-512.

same results as the CPU cost comparison. We test this simulating a smart contract that executes simple transactions that trigger the computation of the hash functions.

We evaluate first the complete hashes supported by Ethereum, by calling the provided Solidity instruction exposed for the Keccak-256, SHA-256 and RIPEMD-160 functions for a similar increasing input sequence as in the performance evaluation above. A total transaction gas cost in Ethereum is composed of the transaction submission cost and of its execution cost. It is notoriously difficult to accurately estimate the individual costs for different parts of an Ethereum transaction, or even its total costs in absence of an effective execution. This is why we decided to perform the evaluation of the total transaction cost on our private Ethereum node setup running Geth, which would effectively charge the transactions.

We first initialize our test contract with the hash input data in order to obtain a more accurate gas cost approximation in respect to the execution. Otherwise our cost evaluation would be severely impacted by the variable input size submitted with each transaction. Another overhead, also constant, is resetting a boolean flag at each function call to change the contract state. Without this the calls to our contract would not count as transactions and we will not be charged with their gas cost. We observe the gas costs for our transactions in Figure 2.

We note that the evaluated gas cost represents the total cost, therefore including the transaction submission overhead mentioned above. In addition, possible other internal operations in the contract execution are counted. The charged gas cost mostly seems to respect the expected trend and proportions, according to the CPU costs measured above. The only disparity is observed in the small input size area, where SHA-256 is charged slightly more than Keccak-256. The difference is more obvious if we consider only a synthetically evaluation by counting exclusively just the cost of invoking the hash operations as defined in the Ethereum Yellow Paper (Wood, 2021), which we examine further below.

We continue our evaluation by comparing with the gas costs of our Blake2 complete implementation in Solidity, which makes use of the available precompiled compression function. For clarity, we keep in the comparison only the evaluation of Keccak-256 from the other functions, which is the most used in practice, showing the difference in Figure 3. The discrepancy with the CPU time cost evaluation is striking, despite optimizations done in the Solidity implementation via invoking low cost EVM assembly instructions when possible. A user would essentially be charged three times or more for submitting a Blake2 transaction in comparison with a Keccak transaction, despite the effective hash real lower cost in execution. This situation could be avoided by integrating a complete precompiled Blake2 implementation. As mentioned in Section 3 this would require standardizing a fixed gas cost per hash invocation.

In order to observe what would be an appropriate cost for a complete implementation of Blake2, we perform a synthetically analysis of the current costs set in the Ethereum Yellow Paper (Wood, 2021). The Blake2 precompiled compression function has

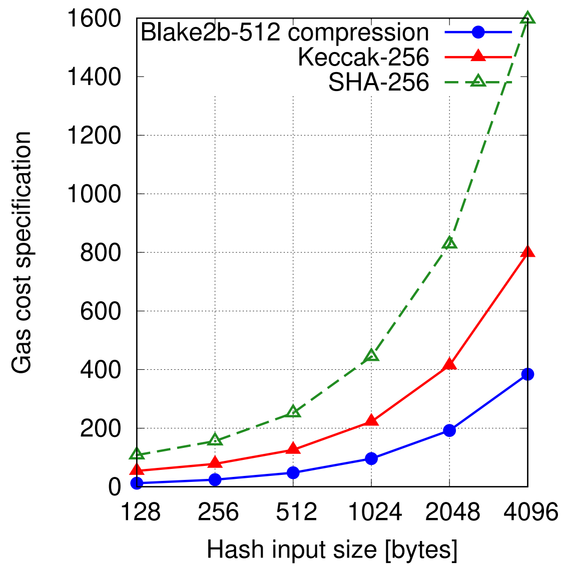

Figure 4: Synthetically comparison of hash gas costs according to Ethereum Yellow Paper.

the cost fixed to the number of rounds necessary for iterating the compression over one block of 128 bytes, which by default is 12. This should be multiplied with the number of blocks when applying the complete hash over a larger input. We emphasize that this is not the complete cost, not including the Solidity part for completing the function, which we added in our evaluation above. The Keccak-256 function is charged at 30 gas base cost adding 6 gas per input word (32 bytes in the EVM specification). The cost defined for the SHA-256 function is two times the cost of Keccak-256, and the cost of RIPEMD-160 is ten times the cost of SHA-256. Figure 4 summarizes the current hash costs as defined in the Ethereum Yellow Paper (we leave out RIPEMD for clarity).

A first observation is that the standardized gas costs charged exclusively for the hash computation operation in case of Keccak-256 and SHA-256 seem marginal to the total transaction costs depicted in Figure 2, representing less than one percent. This makes the high costs inflicted by our Blake2 Solidity addition on top of the precompiled function observable in Figure 3 even more artificial. We can observe an interesting fact though: the cost charged for only the precompiled part of the Blake2 function is mostly similar with the complete Blake2 CPU cost observed in Figure 1 if we consider its proportion of the Keccak cost (our evaluation shows even an exact 47 percent of the Keccak cost for both cases at 4096 bits). Therefore, we can conclude that the cost currently set for the precompiled Blake2 compression function could be preserved unchanged for a precompiled form of a complete Blake2 function implementation.

5 CONCLUSION AND FUTURE WORK

We have discussed and evaluated the current supported hash functions in Ethereum smart contracts, showing the difficulty of reducing inflicted artificial costs to users when trying to provide new implementations via Solidity. We focused on the Blake family of functions, and demonstrated their good performance results in practice by comparing to the other hash implementations supported by Ethereum. We performed our evaluation on top of Geth (Get, 2013), the most popular Ethereum blockchain node implementation, which strengthens the validity of our practical results in a general context. Our conclusion is that currently the best route towards integrating a new, more effcient hash to be used in Ethereum smart contracts, is in a precompiled form with a standardized gas cost. The complicated issue is a thorough evaluation for establishing such a cost. Our experiments helped us to provide a suggestion for a complete implementation of Blake2, where the cost could be kept similar to the one currently set just for the compression part. Integrating such support for new hashes permits to use these in smart contracts and can also have a role in the internal functionality of the blockchain platform.

As future work, we intend to explore more the potential integration of Blake3 (O'Connor et al., 2020) in the Ethereum blockchain, by checking also other implementations as well as its potential use in optimizing internal storage operations (Ballet and Buterin, 2020). There are other new promising hash functions that can be investigated too for a possible integration (Grassi et al., 2019). Finally, the EVM is currently operating in a single threaded fashion for executing smart contracts. However, several approaches and studies have been recently proposed over a possible concurrent execution (Saraph and Herlihy, 2019; Buterin, 2017; Dor, 2018). According to its specification Blake3 has shown a more promising performance in a multithreaded environment, therefore we believe this direction would be interesting to investigate.

REFERENCES

(2013). Geth - Go Ethereum - Official Go implementation of the Ethereum protocol. https://geth.ethereum.org/ - Accessed on 05/02/2021.

(2016). Zcash. https://z.cash/ - Accessed on 05/02/2021.

(2018). Dora Network. http://www.dora.network/index-en.html - Accessed on 05/02/2021.

(2020). SWC Registry - SWC 114: Transaction order dependence. https://swcregistry.io/docs/SWC-114 - Accessed on 05/02/2021.

(2021). Ethernodes.org - Ethereum Mainnet Statistics - Clients. https://www.ethernodes.org/ - Accessed on 05/02/2021.

(2021). Solidity - the official language description page. https://docs.soliditylang.org/en/v0.8.1/ - Accessed on 05/02/2021.

Aumasson, J.-P., Neves, S., Wilcox-O'Hearn, Z., and Winnerlein, C. (2013). Blake2: simpler, smaller, fast as MD5. https://www.blake2.net/blake2.pdf - Accessed on 05/02/2021.

Ballet, G. and Buterin, V. (2020). EIP-3102: Binary trie structure. https://eips.ethereum.org/EIPS/eip-3102 - Accessed on 05/02/2021.

Bertoni, G., Daemen, J., Peeters, M., and van Aasche, G. (2011). The Keccak SHA-3 submission - version 3. https://keccak.team/files/Keccak-submission-3.pdf - Accessed on 05/02/2021.

Buterin, V. (2017). EIP-648 easy parallelizability. https://github.com/ethereum/EIPs/issues/648 - Accessed on 05/02/2021.

Champine, L. (2020). Blake3 - Pure Go implementation. https://github.com/lukechampine/blake3 - Accessed on 05/02/2021.

Consensys (2017). Project Alchemy - Blake2b implementation. https://github.com/ConsenSys/Project-Alchemy/tree/master/contracts/BLAKE2b - Accessed on 05/02/2021.

Dobbertin, H., Bosselaers, A., and Preneel, B. (1996). RIPEMD-160: A strengthened version of RIPEMD. In *International Workshop on Fast Software Encryption (FSE '96)*, pages 71–82.

Grassi, L., Khovratovich, D., Rechberger, C., Roy, A., and Schofnegger, M. (2019). Poseidon: A new hash function for zero-knowledge proof systems. Cryptology ePrint Archive, Report 2019/458. https://eprint.iacr.org/2019/458 - Accessed on 05/02/2021.

Hess, T., Luongo, M., Dyraga, P., and Hancock, J. (2016). EIP-152: Add BLAKE2 compression function F precompile. https://eips.ethereum.org/EIPS/eip-152 - Accessed on 05/02/2021.

Intel (2015). Intel Advanced Vector Extensions - AVX 512. https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html - Accessed on 05/02/2021.

NIST (2002). Secure Hash Standard - FIPS Pub 180-2.

O'Connor, J., Aumasson, J.-P., Neves, S., and Wilcox-O'Hearn, Z. (2020). Blake3 one function, fast everywhere. https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf - Accessed on 05/02/2021.

Saraph, V. and Herlihy, M. (2019). An empirical study of speculative concurrency in Ethereum smart contracts. In *Tokenomics 2019*.

Vladimirov, M. and Khovratovich, D. (2017). ERC20 API: An attack vector on Approve/TransferFrom methods.

Wood, G. (2021). Ethereum: A secure decentralised generalised transaction ledger (Petersburg version) - Yellow Paper. https://ethereum.github.io/yellowpaper - Accessed on 05/02/2021.