DRAFT: The æternity blockchain white paper

Aeternity development team   Thomas Arts   Yanislav Malahov

Sascha Hanse

February 18, 2020

Abstract

The æternity blockchain aims to be a development platform for advanced blockchain applications that can be used by millions of users. On the one hand the æternity blockchain is scalable due to a next generation consensus algorithm and the use of state channels. On the other hand, the æternity blockchain is a excellent development platform for applications by offering native support for commonly used blockchain features, as well as secure and highly efficient contract language and virtual machine.

In this white paper we explain the above mentioned concepts of the æternity blockchain and highlight the design decisions. The paper provides a high-level overview of the technology. For detailed and specific implementation details we refer to the æternity protocol description.

Contents

1 Introduction

Blockchain technology has been attracting significant attention since Bitcoin was proposed by Nakamoto [23] in 2008. Many blockchain platforms are being developed to serve the next generation of applications. The novelty of the field provides an interplay between developers offering new features, enabling new applications to be written by application developers, and application developers therewith requiring additional features or possibilities.

Aeternity [2, 30] is an open source blockchain platform aiming to be a development platform for advanced blockchain applications that can be used by millions of users. Several key technologies are put in place to meet these scaling requirements, most notably state channels, the next generation of Nakamoto consensus algorithm and the very efficient FATE virtual machine for smart contract execution.

The platform runs as a decentralized trustless distributed ledger using Proof-of-Work (PoW) [15, 7, 29] for leader election. In Sect. 2 we explain how we improve the scalability of original Nakamoto consensus [23] by leveraging Bitcoin-NG [16]. The result of this change is a throughput of about 100 on-chain transactions per second with low latency as opposed to the seven transactions per second of Bitcoin.

Further increases in transaction throughput, possibly thousands of transactions per second, can be achieved via state channels (Sect. 4), an off-chain encrypted peer-to-peer communication protocol. After agreeing on-chain to collaborate in a state channel, parties communicate mutually authenticated transactions to each other. These transactions don't have to be recorded on chain and thus can be exchanged at much higher speed. Closing the channel, with or without dispute resolution, is performed on-chain again.

The æternity blockchain offers a variety of different transaction types based on commonly used applications on other blockchain implementations. For example, by identifying that many blockchains have a need to give human readable, persistent names to objects on chain, the æternity blockchain provides a set of transactions that make this easy for developers, without the need to implement a smart contract for it (3.2). Another example is a set of transactions to register and query oracles, which provide data from outside the blockchain. These transactions are explained in more detail in Sect. 3.

Many features are yet to be invented, but can already be implemented by users if they use the æternitysmart contract language *Sophia*. Sophia is a Turing complete functional language designed with security in mind. Many mistakes that one can make in other contract languages are impossible to make or are easier to detect when using Sophia. In Sect. 5 we present some key ideas of the language.

Contracts are compiled to bytecode, which is executed on a highly efficient virtual machine *FATE*. Similar to other smart contract languages every operation has a gas cost associated to it. This cost reflects the amount of work needed to execute a contract. The FATE virtual machine is specifically designed for æternityto meet the high security and efficiency demands, which we explain in more detail in Sect. 5.2.

The reference implementation of the æternity protocol is written using the functional language Erlang [4]. This language originates from the telecommunication industry and is used in large distributed and concurrent systems (e.g. WhatsApp). However, the choice of implementation language has no further implications for the techniques used and described in this paper.

2 Mining Next Generation

Traditionally blocks in a blockchain contain an ordered list of transactions combined with a cryptographic hash—Blake2b [6] in our case—of the previous block [8, 24] and mining is the act of creating such blocks. Transactions are only added with the creation of a new block which. In [16] Ittay et al. propose to decouple the leader election from inclusion of transactions in blocks for scalability purposes. Their scheme dubbed "Bitcoin-NG" is what we adopted using Cuckoo Cycle for proof-of-work.

2.0.1 Cuckoo Cycle PoW

Cuckoo Cycle [29], a graph-theoretic problem of finding cycles in a graph, is used for proof-of-work puzzles. Finding solutions to this problem is memory bound, meaning that runtime is constrained by memory latency of accessing nodes of the graph. Cuckoo cycle was chosen because memory latency is believed to not allow as big performance gains from specialised integrated circuits (ASICs) versus general purpose hardware. Verifying the validity of a solution is also trivial, meaning that validating a block has less overhead.

Solving a cuckoo cycle instance requires finding a fixed length cycle in a bipartite graph of 2^N edges and 2^M nodes. The ratio of $\frac{M}{N}$, $\frac{1}{2}$ by default, is one of the parameters to control the hardness of the problem. Edges are represented as N bit strings. As of December 2019, æternity requires cycles of length 42 and 30 bit edges.

2.0.2 Difficulty

Adaptable difficulty allows us to control the expected time it takes to find a solution and thus a new valid block. To allow more fine-grained control over the difficulty of the proof-of-work, the final step after a solution to the graph problem has been found is to hash it. The hash is then compared to a difficulty target, which is adjusted with each new block. If output of the hash function and the difficulty target are interpreted as numbers, then in order to have a valid proof-of-work solution, the hash needs to be smaller than the target difficulty.

The difficulty for each block is deterministically computed based on timestamps of the last 17 blocks. This timestamp is unreliable, since the nodes have no synchronized clocks. However, the timestamp may not precede the timestamp of a previous block. A block submitted with difficulty not meeting the target specified in the previous block will be ignored by honest nodes. Likewise, if a miner presents the wrong target difficulty for the next block, this block will be discarded.

2.0.3 Forks

Whenever there are two or more blocks produced with the same parent block, we speak of a *fork*. This can happen for a variety of reasons, accidental or intended. Forks can last for multiple blocks with miners working on separate branches. A defining feature of every blockchain is the *fork choice rule*. It tells honest nodes which branch to choose in case of a fork. For æternitynodes, the rule is to prefer the longest branch use work committed to it, as measured via the difficulty, as tiebreaker.

A different kind of fork can be caused whenever nodes run different software and disagree about the rules of what constitutes a valid block. In this case manual intervention by node operators might be required.

2.1 Bitcoin-NG

Where in Bitcoin or Ethereum each block filled with transactions requires a proof-of-work solution, a miner in æternity has to find one proof-of-work solution and can then create multiple blocks with transactions until the next miner finds an admissible proof-of-work solution. This scheme was proposed by Ittay et al. [16] under the name "Bitcoin-NG".

2.1.1 Key- and micro-blocks

Bitcoin-NG requires two different kinds of blocks. One kind, called *key-block*, to elect the miner, or *leader*, who is allowed to include transactions on chain, and *micro-blocks* containing transactions. Every new key-block starts a new epoch—or *generation* in the æternity context.

The key-blocks contain the hash of a previous micro-block as well as the hash of the previous key-block. Micro-blocks are being created in rapid succession making it likely that any given new key-block does not include the hash of the latest micro-block produced by the previous round leader. This is called a micro-fork. (cf. red micro-block in Fig. 1). The transactions of the discarded micro-blocks are put back in the transaction pool and taken care of by the next leader. Since they have been seen in a micro-block before, they will very likely be valid in future micro-block as well.

In practice, miners are allowed to publish a micro-block every three seconds and the computational complexity—associated with every transaction and measured in gas—is limited per block. These limits are put in place to avoid miners flooding the network.

To give miners an incentive to work on top of the newest micro-block, transaction fees are divided by giving 40% to the leader producing a micro-block and 60% to the miner of next key-block after that micro-block[1].

[1]Note that if miners were to only mine key-blocks, there would be no transactions on the

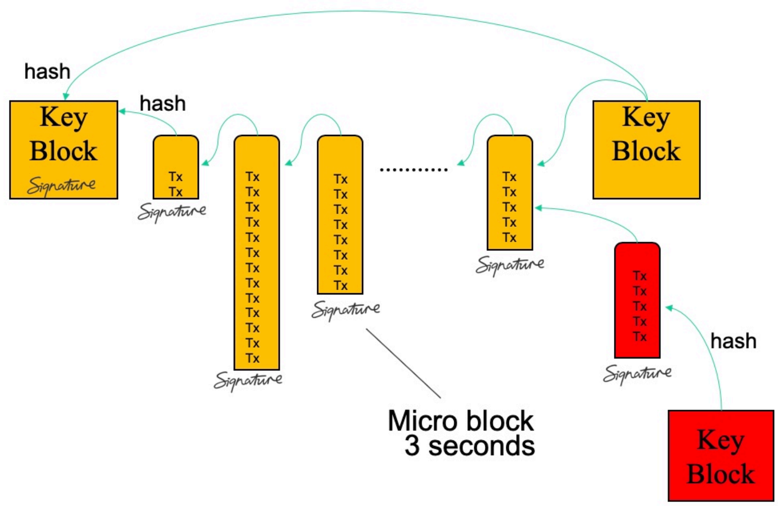

Figure 1: Next generation consensus

2.1.2 Fraudulent leaders

There is only one leader per generation creating micro-blocks. Each key-block includes the public key of the new leader and consecutive micro-blocks are only valid if signed with the associated private key. This protects against third parties trying to post micro-blocks.

A malicious leader, however, could construct forks either by forking directly from the key-block, or by forking on a micro-block. This could be done to disrupt the network or to perform double spend attacks. A malicious miner could also try to produce more micro-blocks than entitled to by fiddling with the micro-block creation timestamp.

In order to mitigate the risk of a leader forking its own sequence, there is a mechanism in place to detect and report this by submitting a Proof-of-Fraud. This is submitted in the next generation. The leader is punished by not receiving a block reward and in order to make that possible, block rewards are not immediately paid out but kept for the duration of 180 blocks.

2.1.3 Divide and conquer

Each transaction is a binary of a certain size. Internally both size and computation are expressed in gas and there is a maximum amount of gas per micro-block that allows for maximally 300 KB per micro-block. Thus, an additional advantage of the introduction of micro-blocks is that instead of a huge block with 180,000 transactions in 3 minutes, we can create 60 reasonably sized micro-blocks of maximally 300 KB in the same 3 minutes. This has several advantages

chain at all.

for network latency and smaller blocks are easier and faster to gossip through the network. Moreover, if it takes longer to compute the next key-block we are not bound to a maximum block size, because we generate new micro-blocks as long as no key-block has been found.

3 Transaction types

Transactions, more appropriately operations, specify state transitions to be applied to the state of the blockchain. As opposed to for example Ethereum, where only one type of transaction exists and all additional logic is enforced via smart contracts, æternity has many different kinds of transactions. These are provided as convenient built-in functionality for frequently used features, while all other functionality can still be realized via smart contracts. Every state transition caused by such a transaction has a computational complexity, both in terms of storage size and execution, and given that we are building an open, permissionless network, we need to measure and possibly regulate the amount of computation used. We refer to this measure as "gas" and each block has a maximum amount of gas it can contain. This gas must be bought via the natively available currency and therefore only accounts with enough currency can submit valid transactions, although it is possible to author transactions on behalf of others. All gas, or simply fees, are paid to miners. To prevent anyone from spending currency that is not theirs, among other things, transactions are authenticated either via digital signatures using Ed25519 [10, 9] or by user specified logic in generalized accounts (3.4). Replay protection is achieved via a strictly increasing nonce [27].

3.1 Sending coins

The most basic transaction is the **spend transaction** used to transfer coins from sender to receiver. The receiver can either be an account, oracle or contract. In addition to coin transfers, a sender can also attach an arbitrary binary payload, which can for example be used for proof of existence, registering a hash or file on the blockchain.

3.2 Aeternity naming system

By default all objects addressable within the blockchain are identified by 256 bit numbers. Just like users of the web prefer remembering DNS names over IP addresses, users of æternity have the option to using names. Currently all names have the extension *.chain*, e.g. *emin.chain*.

Major challenges for a naming system are to offer a reasonably fair system to distribute names and discourage name squatting. To achieve those two goals we settled on using a first price auction for short names, which are assumed to be more coveted. Names longer than twelve characters can be registered instantly.

The auction has two parameters, the initial starting bid and closing timeout after the last bit, which are adjusted based on the name being auctioned. For example, the 4 character name *emin*, starts with an initial bid of 134.6269 coins and the auction would be open 29760 blocks (approx 62 days) after the last valid bid. Each bid must be at least 5% higher than the previous one in order to be valid. Every successful bid will lock up the given number of coins and free up the coins of the previously highest bid.

The actual process of claiming a name requires a bit of setup to prevent *front running*, where an observer snatches up a name before someone else by observing unconfirmed transactions in the network and submitting a competing registration attempt.

To prevent against front running, the first step in reserving a name is the **preclaim transaction**. The pre-claim contains the hash of a combination of the desired name and a random number (called *salt*). An observer is unable to guess the combination of name and random number, which prevents the front running.

After the preclaim is accepted, the claimant reveals the name and salt in the **claim transaction**. If name and salt produce the hash in the preclaim, then they can either claim the name directly or an auction is started.

There is however still a potential for front running by putting a preclaim and claim transaction into a block upon seeing an unconfirmed claim. This is mitigated by requiring the preclaim and the claim for a given name to be in different generations and therefore different blocks.

If the claim triggered an auction, bids can be submitted by sending claims with the desired name, salt set to zero and a greater amount of coins than any previous bids.

Once the name has been registered, an **update transaction** is needed to point the name to something, for example an account. Additionally, there is a **transfer transaction** to change the owner of a name and a **revoke transaction** to free the name. And even without active revocation, names expire after a while, unless renewed in time with an update transaction.

When a name has been assigned to an owner and an update transaction has pointed this name to an account, then one can use the *name hash* of the name instead of an account in, for example, a spend transaction.

It is important to realize that the names are part of the blockchain logic. A user should not trust any third party to perform a name lookup on chain and then substitute the name by an account. If a user want to transfer tokens to Emin, the user should put the name hash of *emin.chain* in the transaction and sign this transaction.

3.3 Aeternity oracles

Oracles are a mechanism to bring arbitrary external data onto the blockchain, which can then be used in smart contracts. This can be sensor data, news events, stock prices, results of a match, supply chain data, etc. Assessing authenticity of external data [31, 17, 1] is still a somewhat open problem but can be solved if the availability of public key crypto systems are available to all parties. But in general oracles provide data without robust security guarantees.

Oracles are announced to the chain by a **register oracle transaction**. This specifies in what format the oracle expects its queries and in what format it is going to respond. The register oracle transaction also includes the fee of the queries to this oracle. Each query must supply that fee in order to be answered.

After an oracle has been registered on chain, any user can post a **query oracle transaction** with a properly formatted query. Oracle operators monitor the blockchain for queries and ideally post **oracle response transactions** with answers in the predefined format. This makes the answer available on the blockchain and thus also available to any smart contract. It is worth noting, that any oracle answers are by default publicly available and thus special care would need to be taken in order to make it private.

3.3.1 Data as a service

External data may come from a large database, possibly also accessible in different ways, but via the oracle made accessible on the blockchain. Typically one could think of supply chain data. If supply chain data is accessible via a trusted oracle, one could post an oracle query for last transaction on a specific item one ordered. Although the answer on such a query may be interesting and valuable in itself, the main purpose of asking for it would be to use it in a contract to transfer some tokens (goods have arrived in harbour, 20% of tokens are transferred).

The above supply chain data may be anonymous enough to appear on a blockchain. There is, however a privacy issue, external data that is put on chain is made public. So, even if there might be an interesting use case, one must be careful with for example personal data. If one would have an airline oracle that given a last name and booking reference returns flight data "date", "from" and "destination" airport, then this becomes public data. Having a contract pay the travel agent when the oracle returns that the correct date and flight has been booked, is therefore a bad idea. Even encrypting or decrypting the data in the contract would be a bad idea, since contract state and operations are visible.

Moreover, one cannot get paid for the same data twice, because the first time it is posted, it becomes public. Therefore, typical data normally is rather anonymous or invaluable to others than involved parties, or is already/will become public, such as the weather or the outcome of a match. Point is that one can use data that becomes available in the future to base contractual decisions upon.

3.3.2 Timing

Users that post a query would normally want a response rather quickly. Therefore, they can specify query TTL, either absolute or relative key-block heights. A relative query TTL of 2 assures that if the oracle does not answer within 2 key-blocks after the query is accepted onchain, the query fee is not paid. In fact, an answer that is too late, will not make it on chain and no contract can use it in a decision.

Oracles have a specific lifetime, supplied in block height when registering the oracle. After that block height, queries to the oracle are no longer resulting in a response. The lifetime of an oracle can be extended using an **extend oracle transaction**.

3.3.3 A lottery example

An example of the use of oracles and contracts can be illustrated with a little lottery example. Note that all computation on a blockchain should be deterministic in order to be able to validate the results. If not exactly the same, the corresponding state hashes will differ. As a consequence there is no random number generator in the Sophia language[2].

Running a simple lottery in which users buy a ticket and after a while one draws one of the tickets as the winner, is somehow depending on some kind of fair randomness. If the number is known or computable at the start of the buying process, one might be able to figure out what ticket to buy to win the lottery. But if we close the lottery and then ask an oracle for a random number, then a trusted, but disconnected computation can be used to draw the winner.

So assume there is such an oracle, monitoring the chain and registered with a reasonable query fee covering for its cost of operation. The oracle has an identifier, for the example say

"ok_shEHMV8Q2F1HR86pcyF7DYpudg8hnvJwJuVE3berWpbktnL2R".

We can now write a Sophia contract that takes this oracle identifier as input of its initialization and uses it for random numbers in the lottery game.

```
include "List.aes"

contract Lottery =

  record state = { participants : list(address),
                   price_sum : int(),
                   close_height : int(),
                   oracle : oracle(int, int),
                   query : option(oracle_query(int, int))  }

  entrypoint init(rand : oracle(int, int)) : state =
    { participants = [],
      price_sum = 0,
      close_height = 0,
      oracle = rand,
```

[2]Even if some kind of random function would be offered, it would be deterministic and hence have a predictable outcome

```
    query = None }
```

The contract stores a list of participants in its state, a price sum that increases for each ticket bought, a closing height after which no new tickets can be purchased. The initial closing height is zero, because initially there is no ongoing lottery. The state also contains an oracle and an optional query. This query will be instantiated when the lottery is closed and a ticket is drawn.

The creator of this contract may now start the lottery by supplying a relative closing height, for example, 20 key-blocks from that the transaction gets on chain; approximately one hour. Any user can then buy a ticket.

```
  stateful payable entrypoint start(n : int) =
    require( Call.caller == Contract.creator, "not creator" )
    require( state.price_sum == 0, "lottery ongoing" )
    require( n > 1, "block in future")
    put(state{ participants = [],
               close_height = Chain.block_height + n })

  stateful payable entrypoint buy() =
    require( state.close_height > Chain.block_height, "lottery closed" )
    require( Call.value == 10, "price ticket 10" )
    put(state{ participants = Call.caller :: state.participants,
               price_sum = state.price_sum + 8 })  // we take 20%
```

Note that a lot of conditions are checked, resulting in abortion of the contract when falsified. In general, making the contract safe requires thinking through a large number of possible scenarios in which things may not work out as expected. For example, if no users buy a ticket[3], the contract creator must be able to restart the lottery at a later time, but the creator should not be able to restart as soon as there are participants. Similarly, one should not allow participants to buy tickets when the lottery is closed.

The keyword `payable` expresses that we expect the participants to add a token amount to the contract call transaction, which is checked by comparing `Call.value`. Similarly, the contract creator needs to pay something into the contract to cover the oracle query fee[4] in case there are too few participants.

When the closing height is reached, someone, most likely the creator of the contract, asks the oracle to draw a number between 1 and the number of participants. This call returns the query., such that anyone can monitor the chain to see if the query has arrived. When that's the case, the winner, or anyone else, can call the claim function, which will transfer the price sum to the winner.

```
 stateful entrypoint draw() : oracle_query(int, int) =
```

[3]The price of the ticket is set to 10 aettos instead of a more realistic 10_000_000_000_000_000 to make the code more readable

[4]Note that there is no check in the contract that it has enough funds to query the oracle after starting a lottery. The participants are at risk here

```
    require( Chain.block_height > state.close_height, "ongoing lottery" )
    require( state.price_sum > 0, "no ongoing lottery" )
    require( state.query == None, "already drawn" )
    let q =
       Oracle.query(state.oracle, List.length(state.participants) - 1,
                    Oracle.query_fee(state.oracle),
                    RelativeTTL(5), RelativeTTL(480))
    put(state{query = Some(q)})
    q

  stateful entrypoint claim() : option(address) =
    switch(state.query)
        None => abort( "no drawing" )
        Some(query) =>
          switch( Oracle.get_answer(state.oracle, query))
             None => abort("waiting for query")
             Some(winner) =>
               let winner_account = List.get(winner, state.participants)
               // Spend to winner
               Chain.spend(winner_account, state.price_sum)
               put(state{ price_sum = 0, query = None })
               Some(winner_account)
```

The reason to make it possible for anyone to call these functions is to ensure that the contract creator can force progress to start a new lottery and the winner to be able to claim even if the contract owner is not around. The TTLs in the query assure that the oracle has approximately 15 minutes to answer, long enough to even get it out in busy times. Within a day one then has to claim the price sum, otherwise, because the query answer is then no longer on chain.

This contract is far from fully secure, but illustrates an example of how oracles and contracts can be used together.

3.4 Generalized accounts

Generalized accounts are a way to provide more flexibility to authenticating transactions. This can, for example, be useful when one would allow users to sign transactions with other cryptographic primitives than the default[5] EdDSA as mentioned in Sect. 3.

If a user wants to have a generalized account, then they must provide a smart contract in a **attach transaction**. This contract is thereafter attached to the given account. The contract must have an authentication function that returns a boolean whether or not authentication is successful. The attach transaction

[5]Some hardware devices may be restricted to other cryptographic signing algorithms than the default on the æternity blockchain.

itself is just like all previously mentioned transactions signed in the default way. It turns a normal account into a generalized account, and *there is no way back*.

When an account is a generalized account, any transaction can be wrapped in a so called **meta transaction**. That is, one prepares an ordinary transaction in the usual way, but with a nonce set to zero. After that, one adds additional fee, gas and authentication data to run the smart contract. When this transaction is processed, the authentication function in the smart contract associated with the account is called with the provided authentication data as input. If the authentication fails the transaction is discarded, otherwise its inner transaction is processed.

The following smart contract is an example that allows signing with the ECDSA algorithm [19] and the popular elliptic curve Secp256k1, used for example by Bitcoin and Ethereum [12, 21].

```
contract ECDSAAuth =
  record state = { nonce : int, owner : bytes(20) }

  entrypoint init(owner' : bytes(20)) = { nonce = 1, owner = owner' }

  stateful entrypoint authorize(n : int, s : bytes(65)) : bool =
    require(n >= state.nonce, "Nonce too low")
    require(n =< state.nonce, "Nonce too high")
    put(state{ nonce = n + 1 })
    switch(Auth.tx_hash)
      None          => abort("Not in Auth context")
      Some(tx_hash) =>
        Crypto.ecverify_secp256k1(to_sign(tx_hash, n), state.owner, s)

  function to_sign(h : hash, n : int) : hash =
    Crypto.blake2b((h, n))
```

The contract is initialized by providing the public key used for signing and the nonce (in the contract state) is set to 1. The authentication function takes two parameters, the nonce and the signature. The authorization function checks that the nonce is correct, and then proceeds to fetch the TX hash from the contract environment using `Auth.tx_hash`. In this example the signature is for the Blake2b hash of the tuple of the transaction hash and the nonce). The authorization finally checks that the private key used for signing the hash was from the owner.

By attaching this contract to an aeternity account, users can sign aeternity transactions with their bitcoin private key. They need to keep track, of course, what nonces they have used for this contract, to provide the right next nonce.

3.4.1 Security considerations

Before the authentication is performed, there is no account that one can charge for the computational effort of running the authentication function. After all, anyone could wrap a transaction in a **meta transaction** and submit it. It would be an easy attack to empty a generalized account if the account had to pay for failed authorization attempts. So, the gas for authentication is only charged when successful. This opens up for another unpleasant attack.

Since there is no cost involved for the user to run an authentication function, but the miner needs to spend execution cycles, one could potentially write a complex function as authentication function and extract resources from a miner by calling one's own authentication function with failing input data. This is mitigated by not allowing expensive chain operations in an authentication call. Moreover, miners are free to implement any sophisticated rules for accepting transactions in their mining pool, such that they can reject this behaviour when observed.

Using different signature algorithms is only one of many possible uses of generalized accounts. Other uses cases can be multi-sig, spending limits (per week/month), limiting the transaction types, and more. For these applications smart contracts have to be written. Utmost care needs to be taken when implementing the authorization function in these smart contracts. If the contract does not enforce integrity checking or replay protection, then it will be vulnerable to abuse.

3.5 Financing transaction costs

In order to make users enthusiastic about a blockchain application, one may want them to try it for free. However, there are always costs involved for transaction and gas. This means that a new user has to buy tokens at some exchange to pay for the fees. This can be considered a hurdle for adoption. Of course, one can ask a user for an account and put some tokens on it, but then those tokens can be used for anything. The æternity solution is more powerful and can be used to pay for just specific transactions. It can be used to pay for both transaction fee and gas cost of a contract call.

Assume a game played via a contract on the blockchain. One interacts with the game, by calling the contract. In order to get more users for the game, the game provider could make an App that visualizes the game and asks for a next move. This App could automatically create an æternity account, even without the user being aware of it. This account can be used to sign transactions on the blockchain, but there are no tokens in the account. This move is then encoded in a transaction signed by the players account in the app. The transaction is submitted to the game provider, who inspects it to see that this is indeed a move in the game and wraps it in a **payingfor transaction** signed by the game provider. The gas and fee are now paid by the game provider's account. Clearly this is also a way to have some trustful cross-chain activity. The App user could provide the game provider with funds on a different blockchain.

One can pay for any transaction apart from the payingfor transaction itself. So even a generalized accounts meta transaction can be paid for, as long as it recursively does not contain any other payingfor transaction.

4 Scaling with state channels

Conceptually one can regard a state channel as an agreement between two parties, further called *participants*, to build a chain of state changes just between themselves. Communication happens peer to peer with the blockchain acting as the ultimate source of truth. The lifecycle of a state channel is defined by two different state-machines tracking the off- and on-chain state.

Opening a channel involves two participants agreeing on and then posting a mutually authenticated channel create transaction on-chain. This transaction specifies the involved parties' on-chain accounts, amount of coins they want to lock in the channel and hash of the initial state.

In case neither party want to dispute any operations executed in the channel, all that ends up on chain are exactly just amounts, accounts and a state hash. This makes state channels not just a possible throughput but also a privacy improvement.

4.1 Off-chain

Anything happening off-chain is not part of consensus and thus, at least in theory, it is completely irrelevant what the participants do off-chain as long as they can agree on the on-chain transactions.

In practice, the current implementation is a fairly complex finite state machine that needs to deal with constant disconnects and all the other problems of distributed computation. To be able to re-use as much logic as possible, the off-chain logic closely resembles what would happen on-chain. Participants exchange transactions modifying the available state trees, for example smart contract interactions. The actual consensus between the participants is established via a simple two-phase commit protocol instead of Bitcoin-NG. By default, there are no transaction or gas costs involved[6] and most notably transactions are confirmed as quickly as both parties can sign them. Confirmation time can therefore be reduced to milliseconds, significantly increasing throughput compared to on-chain.

4.2 On-chain

A state channel requires both parties to sign each state to make sure the state is agreed upon. Typically a state channel can be closed under mutual agreement and then a closing transaction is used to re-distribute and return the reserved balances to the on-chain accounts. Another way to extract reserved funds from

[6] In a state channel participants can agree to have a kind of transaction cost, but it is not the default setup

a state channel is to mutually agree upon a withdrawal from the channel to an account of one of the parties.

But there might be disputes. For example, a customer could decide not to cancel a subscription, but keep an empty account in the state channel forever. This is disadvantageous for the shop, because it cannot extract the funds from the already paid coffee in mutual agreement. For this reason, there is a solo close transaction, a way for one party to close the channel on-chain and use the last signed and agreed state as a proof on how the funds should be divided.

Dealing with disputes is a considerable part of the logic and implementation of state channels. This becomes even more evident in the context of using contracts in a channel. A contract may be build in such a way that it re-distributes balances after a certain state has been reached. Imagine s tic-tac-toe game contract, where the funds are re-distributed only after one party has won. It could then be beneficial for a party to quit the game when loosing and solo close, or to simply refuse to sign the last transaction.

Clearly, there are a plethora of scenarios in which one can try to cheat. One could buy a coffee off-chain and at the same time solo close the channel on-chain. That would mean a free coffee. Therefore, funds are not immediately returned after a solo close, but kept for a certain period, called a *lock period*. During this period the other party can post a transaction to refute this claim and show a later state obtained by a mutually signed transaction (the one after buying the additional coffee). Which then again could possibly be refuted, etc.

Quitting when one expects to lose harms the other party, because there is no next state that is more beneficial than the initial state. For this purpose the other party can then force progress the contract and move to the winning state. That is, the party can perform a contract call on-chain and show that it ends up in state that can be claimed the actual final state for which the channel should be closed. This force progress requires more than just the state hash. Here enough of the state has to be revealed such that a miner can execute the next step in the contract.

Finally, it is important to keep in mind that just like when interacting with smart contracts there is always the possibility of losing all deposited coins to a maliciously crafted contract inside a channel.

5 Sophia smart contracts

Smart contracts [28, 13] are programs on the blockchain that can perform tasks with the data on that blockchain. Typically contracts have state (data), which is recorded on the chain. A call to a function in the contract results in a return value and updated state, both put back on the chain as a result of the call.

Smart contracts are an active research field [3] and a substantial amount of effort goes in to studying the verification and validation of smart contracts [20, 11].

There are two major technical challenges for smart contract implementations. The first challenge is to make the contracts execute fast without requiring too many resources. In a blockchain implementation, contract execution is performed in a *virtual machine*. This is an execution engine with formally defined semantics, such that all implementations perform exactly the same computation steps, with exactly the same result and charging exactly the same amount of gas. The æternity blockchain defines two virtual machines, the AEVM, compatible with the Ethereum blockchain, and the more efficient FATE virtual machine.

The second challenge is to design a language to express contracts in such a way that one can understand and reason about the contract both as a human, but also mechanically by computer programs. The language should by design protect contract designers against vulnerabilities that can be exploited. Sophia is a functional language to accommodate for these properties. It is designed as a contract language with security and user comfort in mind. In particular, vulnerabilities in contracts in other languages [5, 22, 14] have been studied with the goal to avoid the possibility to make such mistakes in Sophia.

5.1 The Sophia language

Sophia is a functional programming language [18]. The main unit of code in Sophia is the *contract*. A contract implementation, or simply a contract, is the code for a smart contract and consists of a list of types, entrypoints and local functions. Only the entrypoints can be called from outside the contract. A *contract instance* is an entity living on the blockchain (or in a state channel). Each instance has an address that can be used to call its entrypoints, either from another contract or in a call transaction. A contract may define a type `state` encapsulating its local state. When creating a new contract the `init` entrypoint is executed and the state is initialized to its return value.

5.1.1 Dutch auction contract

As an example, let us consider an auction contract. In such an auction contract, a user could auction an object in the real world by creating and posting a contract to the blockchain, using a **contract create transaction**. Let us assume that this is a Dutch auction, then the initial price would be set high and for each new key-block that is mined (representing time) the price is decreased. Someone buys the object by calling a `bid` function. When this bidding **contract call transaction** executes, the contract computes the price given the current block number; if the caller has supplied enough funds in that call, the seller is paid, the bidder is charged (possibly refunded the extras) and the contract enters a non sellable state for the object. The next bidder will fail the call and only pays for transaction costs, not for the object.

The complete Sophia code for a Dutch auction is presented here:

```
contract DutchAuction =

  record state = { amount : int,
                   height : int,
```

```
                 dec    : int,
                 sold   : bool }

  entrypoint init(price, decrease) : state =
    { amount = price,
      height = Chain.block_height,
      dec    = decrease,
      sold   = false }

  stateful payable entrypoint bid() =
    require( !state.sold, "sold"  )
    let price = demanded_price()
    require( Contract.balance >= price, "not enough tokens" )
    Chain.spend(Contract.creator, price)
    Chain.spend(Call.origin, Contract.balance)
    put(state{sold = true})

  function demanded_price() : int =
    state.amount - (Chain.block_height - state.height) * state.dec
```

The contract languages and hence the evaluation in the virtual machine, must have access to blockchain primitives like the height of the chain and caller accounts. Typically, all blockchain primitives are available from within a contract.

Note that the contract create transaction includes the contract byte code, not the source code, together with information on which version of the compiler is used. Compiled for FATE this contract results in 254 bytes, whereas compiled for the AEVM 2092 bytes are needed. The gas needed to compute the initial state is 240 for FATE compared to 741 for the AEVM.

The `init` function is called when the contract is created to compute the initial state of the contract. The `init` function is not part of the byte code, such that it cannot accidentally be called again. If one wants to reset the state, this has to be explicitly programmed to avoid expensive exploits [26]. The contract designer also has to explicitly mention whether the state of the contract is changed in a call (using the `stateful` keyword). Entrypoints can be called from outside the contract, whereas functions are only accessible from within the contract.

The keyword `payable` is added to explicitly state which function calls expect to come with additional tokens in the contract call transaction. These tokens are added to the contract balance before the call is made. If, however, the call is reverted by a failing `require`d condition, then the provided tokens are returned.

The transparency of the blockchain guarantees that it is verifiable that the first valid bid on chain[7] is correctly paying the right price. Moreover, it is

[7]The æternity blockchain does not guarantee that the first one posting a valid bid becomes the first one on chain.

verifiable that the bidding call transactions accepted later are only charged a transaction fee and the cost of execution. The sale conditions are transparent, but whether the actual object ever arrives is outside the scope of the blockchain

5.2 The FATE virtual machine

The Fast Aeternity Transaction Engine (FATE) VM uses transactions as its basic operations and operates directly on the state tree of the æternity chain. This enables native integration with first class objects such as oracles, the naming system, and state channels since those are all managed by specific types of transactions described on the protocol level. FATE is a simple-to-use machine language, superior to the more traditional byte-code virtual machines currently used on other platforms. It enables easier, safe coding, faster transactions, and smaller code sizes. It is custom-built to seamlessly integrate with the functional smart contract language Sophia.

5.2.1 More secure

Every operation and every value is typed. Any type violation results in an exception and reverts all state changes. This prevents people to circumvent the compiler and write or modify their own FATE code to use type violations as an attack vector.

The instruction memory is divided into functions and basic blocks. Only basic blocks can be indexed and used as jump destinations. This is a precaution to be unable to jump to arbitrary positions in memory. It also fit FATE's function style by having function calls instead of jumps. Moreover, data and control flow are separated, one cannot possibly modify the running contract, since the code memory cannot be written to.

FATE is functional in the sense that updates of data structures, such as tuples, lists or maps do not change the old values of the structure, instead a new version is created. FATE does have the ability to write the value of an operation back to the same register or stack position as one of the arguments, in effect updating the memory.

FATE solves a fundamental problem programmers run into when coding for Ethereum: integer overflow, weak type checking and poor data flow. FATE checks all arithmetic operations to keep the right meaning of it. Integers cannot overflow, since FATE uses unbounded integer arithmetic (cf. Bignums [25]). Floats are not part of the language, avoiding a bunch of issues associated with floating point arithmetic. Also you cannot cast types (e.g integers to booleans). This makes FATE ultimately a safer coding platform for smart contracts.

5.2.2 More efficient

FATE uses high level instructions. There are instructions to operate on the chain state tree in a safe and formalized way. Likewise the virtual machine has high-level support for most of the transactions available on the æternity blockchain.

There are operations such as 'ORACLE_CHECK_QUERY' for querying an oracle or 'AENS_CLAIM' for claiming a name.

Having higher level instructions makes the code deployed smaller and it reduces the blockchain size. FATE contracts use on average ten times less space than the same contract compiled to the AEVM, the Ethereum compatible VM. At the same time, it performs on average much faster and uses therefore less gas.

FATE byte code by itself is already a readable program. For example, the bid function of the Dutch auction contract compiles to this code:

```
FUNCTION bid( ) : {tuple,[]}
  ;; BB : 0
          ELEMENT a 3 store1
          NOT a a
          JUMPIF a 2
  ;; BB : 1
          ABORT "sold"
  ;; BB : 2
          CALL "(h:p"
  ;; BB : 3
          POP var1
          BALANCE a
          EGT a a var1
          JUMPIF a 5
  ;; BB : 4
          ABORT "not enough tokens"
  ;; BB : 5
          CREATOR a
          SPEND a var1
          BALANCE a
          ORIGIN a
          SPEND a a
          SETELEMENT store1 3 store1 true
          RETURNR ()
```

The notion `BB` stands for basic block and jumps are always to such a basic block. Note that for example 'CREATOR', 'SPEND' and 'BALANCE' are native instructions used in basic block 5. The instruction `CALL "(h:p"` in basic block 2 looks a bit cryptic for a call to the function `demanded_price()`. Each function name is hashed to 4 bytes that are printed as a string.

Both memory constraints and computation efficiency are important to enable smaller contracts to to get more computation into a micro-block.

6 Future ambitions

The æternity blockchain went live on November 28th, 2018. For the curious reader, there is a timestamp in each block and the first mined key-block has timestamp "1543373685748", which is the time in milliseconds using POSIX time. Since that first date, 3 major protocol updates have successfully be applied, enriching the æternity blockchain with new features. The new protocols are effective at a certain height and the software supports the old protocol under that height and the new protocol from that height. Each protocol is referred to by name for ease in communication with developers of blockchain applications: *Roma*, effective at height 0, *Minerva*, effective at height 47800, *Fortuna*, effective at height 90800, and *Lima*, effective at height 161150.

In the future, there will be more protocol upgrades with additional features, some of which we are going to outline in this section.

6.1 Formal verification

Over the course of its existence Ethereum has had many big flaws on deployed contracts uncovered and abused. Some of these were simple developer errors and others very subtle due to the complexity of both the EVM and Solidity.

Formal verification is one of the approaches to prevent these problems by allowing code to be proven correctly with regards to a given specification. Formal verification is used in many areas where high assurance is of vital importance, e.g. cryptographic systems. Take the parity multi-sig contract flaw, which allowed anyone to destroy one of the libraries used by the multi-sig contract rendering it useless, as an example. A formal specification of this contract could include the assumption that destruction functions can only be called by authorized accounts or not at all. Checking this specification against the code would then have raised an error. But there is certainly still the problem of writing good specifications.

Since Sophia was written with formal verification in mind, we have laid most of the ground work required to provide these tools. Together with Sophia being a functional language, this should prevent many classes of bugs plaguing Solidity smart contracts.

6.2 Native tokens

The ERC-20 standard, which specified an interface for fungible tokens, was arguably one of the biggest drivers of adoption for Ethereum. With that came the ability for anyone to design and test economic systems, which was a great catalyst for the innovation happening on Ethereum. A big drawback of the token contracts deployed was and still is the requirement to pay gas and thus own Eth, which can be a major hurdle for users, who don't necessarily want to own any Eth or even know what that is. In addition to that, interacting or integrating with token can be a pain, especially since the interface evolved over time and if upgradability was not baked into the contract there are only very clumsy upgrade paths for old contracts.

To address these drawbacks, æternity will make tokens native to the blockchain. That opens the way to be able to use tokens to pay transaction fees, which frees users from the requirement to own any other tokens. It will make usage cheaper since the basic token logic and storage can be optimized in the virtual machine. Finally, this will allow tokens to benefit from updates and upgrades for free, without needing complex upgrade strategies.

6.3 Computational integrity

Computational integrity assures that a given function was computed correctly. This could mean that a coin transfer correctly deducts coins from one account while adding it to another or the correct execution of a smart contract call. Currently assuring these correct executions is done by miners and node operators via complex consensus rules. It also requires everyone who wants to check the correct execution to re-run the full computation, which could be very costly.

There are different ways that allow checking the integrity of a function execution but we are going to assume that the prover, the agent running the computation and trying to prove that they executed it correctly, can generate a succinct proof. Succinctness implies that checking the proof is computationally less expensive than actually re-running the computation. This proof can then be read and validated by another party, the verifier, who wants to check the integrity of the computation.

The existence of such primitives then allows scaling, since checking the proof takes less resources than running the computation. Additionally, these proofs can be generated in such a way that the function itself can be private, while still being verifiable by a third party, which would be a big gain in privacy.

Sophia already comes with some primitives to write efficient proof verifiers and in the future we want to integrate these primitives even further into the æternity protocol, to make it faster and private.

6.4 Scaling

The Bitcoin-NG consensus described in section 2 allows us to handle around 120 transaction per second which is sufficient to process everything almost without delay or block congestion at the time of writing. But in a future where millions of people want to use æternity we will need higher throughput. One partial solution, state channels, are already available today and will become more relevant as usability improves.

Besides state channels exist many other different approaches, which can be used alongside. The most obvious solution is to improve the consensus algorithm in such a way that it can handle higher throughput and there are already many other consensus algorithms, which can improve on Bitcoin-NG.

Another direction to go is to split the blockchain into distinct parts, which is also common for databases. These parts are then called shards and each one will then be responsible for only a subset of all available transactions. Dividing up the work like this would then offer each shard more room to scale just by virtue of only having to handle a fraction of the previous load. Shards will still have to communicate with each other, which is a possible bottleneck, and have to know of each others existence. But overall the system could present a big gain in throughput.

The third widely researched solution are then side-chains, app-chains or child-chains. Unlike sharding, side-chains do not divide a global state space but each one has its own state. The idea of this approach is to have many specific, maybe even single purpose, chains which do not necessarily have to know of each other. The name side-chain usually implies that they are pegged to some sort of main or parent chain, which can be thought of as a communication hub. Side-chain then imply that each app could get its own chain, thus only having to handle transactions for this one app and not be concerned with all the other applications. This in turn would then allow each app much more throughput and maybe even specific optimizations.

All the solutions described here offer different trade-offs but could certainly be used in tandem. æternity will most likely take elements from all the presented approaches in order to meet demands by users.

References

[1] Adler, J., Berryhill, R., Veneris, A., Poulos, Z., Veira, N., and Kastania, A. Astraea: A decentralized blockchain oracle. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)* (2018), IEEE, pp. 1145–1152.

[2] Aeternity. Aeternity protocol. `https://aeternity.com/`, 2019. Accessed: 2019-05-20.

[3] Alharby, M., and van Moorsel, A. Blockchain-based smart contracts: A systematic mapping study. *CoRR abs/1710.06372* (2017).

[4] Armstrong, J. Erlang. *Commun. ACM 53*, 9 (Sept. 2010), 68–75.

[5] Atzei, N., Bartoletti, M., and Cimoli, T. A survey of attacks on ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust* (2017), Springer, pp. 164–186.

[6] Aumasson, J.-P., Neves, S., Wilcox-OHearn, Z., and Winnerlein, C. Blake2: simpler, smaller, fast as md5. In *International Conference on Applied Cryptography and Network Security* (2013), Springer, pp. 119–135.

[7] Back, A. Hashcash, 1997.

[8] Bashir, I. *Mastering blockchain: Distributed ledger technology, decentralization, and smart contracts explained*. Packt Publishing Ltd, 2018.

[9] Bernstein, D. J. Curve25519: new diffie-hellman speed records. In *International Workshop on Public Key Cryptography* (2006), Springer, pp. 207–228.

[10] Bernstein, D. J., Duif, N., Lange, T., Schwabe, P., and Yang, B.-Y. High-speed high-security signatures. *Journal of Cryptographic Engineering 2*, 2 (2012), 77–89.

[11] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., et al. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security* (2016), ACM, pp. 91–96.

[12] Bos, J. W., Halderman, J. A., Heninger, N., Moore, J., Naehrig, M., and Wustrow, E. Elliptic curve cryptography in practice. In *International Conference on Financial Cryptography and Data Security* (2014), Springer, pp. 157–175.

[13] Cong, L. W., and He, Z. Blockchain Disruption and Smart Contracts. *The Review of Financial Studies 32*, 5 (04 2019), 1754–1797.

[14] Delmolino, K., Arnett, M., Kosba, A., Miller, A., and Shi, E. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security* (2016), Springer, pp. 79–94.

[15] Dwork, C., and Naor, M. Pricing via processing or combatting junk mail. In *Annual International Cryptology Conference* (1992), Springer, pp. 139–147.

[16] Eyal, I., Gencer, A. E., Sirer, E. G., and Van Renesse, R. Bitcoin-ng: A scalable blockchain protocol. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2016), NSDI'16, USENIX Association, pp. 45–59.

[17] Guarnizo, J., and Szalachowski, P. Pdfs: practical data feed service for smart contracts. In *European Symposium on Research in Computer Security* (2019), Springer, pp. 767–789.

[18] Hughes, J. Why functional programming matters. *The computer journal 32*, 2 (1989), 98–107.

[19] Johnson, D., Menezes, A., and Vanstone, S. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security 1*, 1 (2001), 36–63.

[20] Magazzeni, D., McBurney, P., and Nash, W. Validation and verification of smart contracts: A research agenda. *Computer 50*, 9 (2017), 50–57.

[21] Mayer, H. ECDSA security in bitcoin and ethereum: a research survey. *CoinFaabrik, June 28* (2016), 126.

[22] Mehar, M. I., Shier, C. L., Giambattista, A., Gong, E., Fletcher, G., Sanayhie, R., Kim, H. M., and Laskowski, M. Understanding a revolutionary and flawed grand experiment in blockchain: the dao attack. *Journal of Cases on Information Technology (JCIT) 21*, 1 (2019), 19–32.

[23] Nakamoto, S., et al. Bitcoin: A peer-to-peer electronic cash system. *White Paper* (2008).

[24] Raikwar, M., Gligoroski, D., and Kralevska, K. Sok of used cryptography in blockchain. *arXiv preprint arXiv:1906.08609* (2019).

[25] Serpette, B., Vuillemin, J., and Hervé, J.-C. *BigNum: a portable and efficient package for arbitrary-precision arithmetic*. Digital. Paris Research Laboratory, 1989.

[26] Suiche, M. The $280m ethereums parity bug. *A critical security vulnerability in Parity multi-sig wallet* (2017).

[27] Syverson, P. A taxonomy of replay attacks [cryptographic protocols]. In *Proceedings The Computer Security Foundations Workshop VII* (June 1994), pp. 187–191.

[28] Szabo, N. Smart contracts: building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought,(16) 18* (1996), 2.

[29] Tromp, J. Cuckoo cycle: A memory bound graph-theoretic proof-of-work. In *Financial Cryptography and Data Security* (Berlin, Heidelberg, 2015), M. Brenner, N. Christin, B. Johnson, and K. Rohloff, Eds., Springer Berlin Heidelberg, pp. 49–62.

[30] Wiger, U. Building a blockchain in erlang — code mesh ldn 18. `https://www.youtube.com/watch?v=I4_xX_Zs2eE&feature=youtu.be&t=1730`, 2018.

[31] Zhang, F., Cecchetti, E., Croman, K., Juels, A., and Shi, E. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 aCM sIGSAC conference on computer and communications security* (2016), ACM, pp. 270–282.