The Æternity blockchain white paper

Aeternity development team   Thomas Arts   Yanislav Malahov

December 16, 2019

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 higly 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 Aeternity protocol description.

1 Introduction

Blockchain technology has been attracting a lot of attention since the first blockchain was proposed by Nakamoto [24] 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, 33] is an open source blockchain platform aiming at being 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 the state channels, the next generation of Nakamoto consensus algorithm and the very efficient FATE virtual machine for smart contract execution.

The plaform runs as a decentralized trustless distributed ledger with Proof-of-Work (PoW) protected consensus [30]. In Sect. 2 we explain how we improve scalability of traditional block mining by combining it with the Bitcoin NG technology [15]. The result of this change is a throughput of about 100 on-chain transactions per second.

Further scaling, towards thousands of transactions per second, is realized by state channels Sect. 3, an off-chain encrypted peer-to-peer communication protocol. After agreeing on-chain to collaborate in a state channel, parties communicate mutually signed transactions to each other. Closing the channel, with or without dispute resolution, is performed on-chain again.

The Æternity blockchain offers a variety of different transactions originating from commonly used applications on other blockchain implementations. For example, by identifying that some chains implement a way to claim a name for something on a 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. Another example is a set of transactions to register and query oracles. These transactions are explained in more detail in Sect. 5.

Many features are yet to be invented, but can already be implemented by users if they use the Aeternity smart 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. 4 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 Aeternity to meet the high security and efficiency demands, which we explain in more detail in Sect. 4.2.

The Aeternity protocol defines the Æternity blockchain, such that different implementations agree. At the moment there is one implementation using the functional language Erlang [4], This language originates from the telecommunication industry and 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 refer to an ordered list of transactions combined with a cryptographic hash of the previous block [7, 25]. A hash function maps a binary of any size to a fixed size binary. The Æternity blockchain uses the 32 byte Blake2b hash [6]. Cryptographic hashes are collision resistant, making it very unlikely that two binaries result in the same 32 bytes hash. It is also cryptographically hard to find a binary that results in a given hash.

The sequence of cryptographic hashes in a blockchain avoids that a transaction on the chain is tampered with. If a transaction changes with even one bit, the hash of the block will change and the next block will point to a non-existing hash. An attacker that modifies a transaction in a block will have to modify all consecutive blocks. If the hash is then also used to solve a cryptographic puzzle, that consumes time to solve, but is easy to verify, then each hash modification consumes time and is therefore increasingly costly the more hashes have to be changed.

Many blockchains provide such a Proof-of-Work (PoW) puzzle and the agent trying to solve the puzzle is called a *miner* (cf. [32] for a more refined view). In Aeternity the *Cuckoo* algorithm is used as cryptographic puzzle [30]. This algorithm was designed to be memory bound, meaning that large memories have an advantage over fast computing with little memory. This makes the algorithm less energy consuming than algorithms that are computation bound.

2.1 Mining to resolve solution conflicts

In a distributed system, like the Æternity blockchain, there are many computers (nodes) that together build the blockchain. Each of these nodes can receive user signed transactions. Some of them will try to build a block of these transactions and put it on the chain. Those nodes are sloppily called the *miners*. Adding the next block to the chain is rewarded directly with tokens. As soon as miners have created such a block and solved the corresponding cryptographic puzzle (solving the puzzle is the actual mining part), they gossip it to the other nodes in the network. This causes the evident challenge for each node to decide which of the blocks it receives is the one that is the next block for the chain. There must be a commonly agreed best block.

This challenge is addressed in two ways, on the one hand it is made far less likely that a node receives two competing blocks by adding the previously mentioned cryptographic puzzle. If a solution can on average only be found every 3 minutes (in Aeternity), then it is unlikely that two blocks are mined at almost the same instance.

Moreover, if the solution to the cryptographic problem is easy to verify, faking a solution and spamming the node with bad blocks is also much more difficult [31].

2.1.1 Cuckoo PoW

The Cuckoo algorithm has an adaptable complexity. Simply put it searches for cycles of specific length in a graph with a given length for the nodes. In the Æternity blockchain the length of the cycle is 42 and the size of the nodes is 30 bits. By also demanding a number of leading zeros in the solution (the difficulty), a solution can dynamically be made more or less easy to find. It is cryptographically unlikely that two solutions are exactly the same, since two miners will not create the same block (the block contains the address of the miner receiving the mining reward, as well as a 64-bit nonce that is selected at random to solve the PoW problem).

The adaptable complexity allows us to keep the average time it takes to find a solution around 3 minutes, independent of how many miners are trying to solve it. By measuring the average creation speed of the last 17 blocks, we can compute whether more or less miners are active and adjust the difficulty accordingly. The difficulty is deterministically computed and part of the block.

In addition a timestamp is provided in the block. This timestamp is unreliable, since the nodes have no synchronized clocks. However, the timestamp may not precede the timestamp of a previous block. A malicious miner that presents a block with difficulty that diverts from the target specified in the previous block will have its block refused by the chain. Likewise, if a miner presents the wrong target difficulty for the next block, this block will be discarded.

2.1.2 Forks

Due to network latency, not all nodes are guaranteed to receive all blocks at the same time. Therefore, a situation may occur in which part of the network first receives block A and then block B, whereas the other nodes receives them in a different order, first B and then A. The situation in which this happens and two nodes have different top blocks, is called a *fork*. In fact, due to for example a network split in which several nodes are isolated for a while, forks can be longer than just one block.

If blocks are gossiped to a node, then the total difficulty of the fork determines which branch to select. So in the unlikely case that a fork is created, there is a consensus over how to restore the branch with the highest difficulty.

A different kind of fork is created when one group of the nodes does not accept a block that the other nodes do accept as a valid block. This can be caused by having the nodes run incompatible versions of the protocol. These forks are not automatically restored and are referred to as *hard forks*.

2.2 Scaling transaction confirmations

A user that submits a transaction has to wait for the transaction to be part of a block, before it can be considered a valid transaction. In fact, due to the previously described fork possibility caused by network latency, one may have to see the transaction in a block followed by a few more blocks to be reasonably sure that the chain will perpetually keep that transaction. The longer one waits, the more certainty.

A transaction may be invalid for several reasons and even valid transactions may not always be accepted on chain. There might be other transactions in flight that have impact on the validity of the posted transaction. Since one cannot assume in which order miners put the transactions in a block, just looking at a pool of not yet included transactions gives very little certainty for the receiver of funds associated with that transaction [19].

In many applications a quick temporary confirmation with little definite certainty would be an advantage over more certainty, but waiting longer. A commonly used example is a coffee shop. The price of a cup of coffee may make a shop owner willing to serve the coffee if an early indication gives some guarantee that the transaction is at least acceptable for some miner.

2.2.1 Key-blocks and micro-blocks

In Aeternity we use Bitcoin NG [15] to address this problem. Bitcoin NG boils down to collectively accepting the miner of a block, called a *key-block* in this context, as the *leader*. This leader is allowed to periodically publish micro-blocks with transactions, until the next miner is elected the leader by presenting a new key-block. Only for key-blocks a cryptographic puzzle needs to be solved, thus there is very little cost in publishing micro-blocks with transactions. In Aeternity the leader is allowed to publish a micro-block including transactions every 3 seconds. This means that users can see an accepted transaction much faster than after 3 minutes. This is just a preliminary confirmation, but one that is stronger than having a transaction arrive in the transaction pool.

The derived 3 seconds per micro-block make sure that less powerful computation units can stay synchronized with the chain. The transactions in each micro-block need to be verified, which means computation time (verification of cryptographic signatures, running contracts, lookup data in previous blocks). In order to have a reasonably predictable computation task, each micro-block is limited to a maximum of 6 million gas and on the Æternity blockchain all transactions are associated with a gas cost. Even on a low-end computer that should not take more than a second to execute. The network latency and a good margin are accommodated in the other two seconds.

Terminology-wise, the height of the chain is determined by the key-blocks, incrementing the height by one for each new key-block put on the chain. Micro-blocks follow a key-block and are on the same height. A key-block and the micro-blocks on the same height are collectively called a *generation*. Since key-blocks do not contain transactions, but the micro-blocks do, one could conceptually see the generations as the blocks in the more traditional blockchain view.

The key-blocks should contain the hash of the previous micro-block as well as the hash of the previous key-block. This means that miners preferably update their crypto puzzle as soon as a micro-block arrives, because, if a miner is slow in updating the puzzle and finds a solution for an earlier micro-block, then all following micro-blocks are discarded. This is called a micro-fork, a key-block is mined on a micro-block that is not the last micro-block produced. Compared to forks on key-blocks, micro-forks occur relatively often (cf. red micro-block in Fig. 1). The transactions of the produced and 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.

Note that if a fork appears (cf. the red key-block following the red-micro-block in Fig. 1 then the highest difficulty of the fork counts to see which branch will make it on chain. Most likely, the red branch will win. In Aeternity there is an incentive to mine on top of the newest micro-block, because the fees are divided by rewarding 40% to the leader producing a micro block and 60% to the miner mining the 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 chain at all.

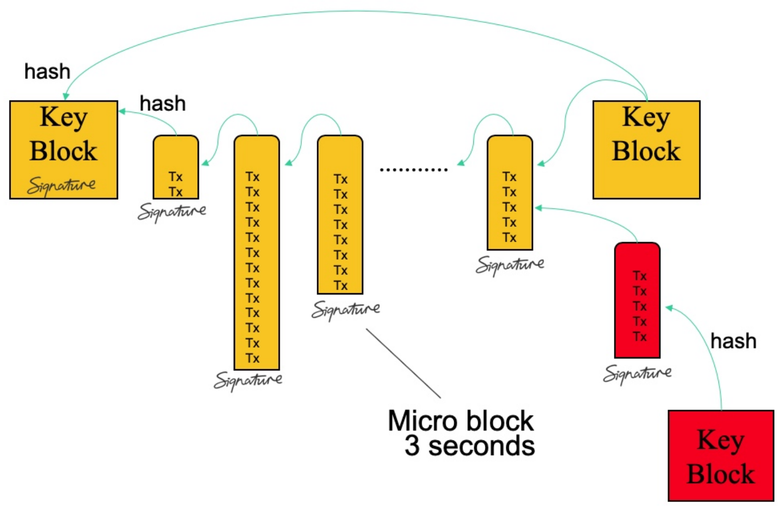

Figure 1: Next generation consensus

2.2.2 Fraudulent leaders

Only the leader is allowed to produce micro-blocks. Each key-block includes the public key of the next leader and consecutive micro-blocks are only valid if signed with the private key associated with it. This protects against third parties trying to post micro-blocks and all micro-blocks should form a well-defined sequence.

A malicious leader, however, could construct forks in a generation of micro-blocks, 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.

2.2.3 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 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. Yet another scaling advantage of NG.

3 Scaling with state channels

The NG technology explained in the previous section improves scalability by faster preliminary confirmation times for transactions. It also lifts the restriction of a maxiumum number of transactions in a generation; when it takes longer to produce the next key-block, additional micro-blocks can be generated containing new transactions. This benefits throughput and confirmation times. But if parties really want confirmation times in milliseconds range and perform more advanced computations than the total amount of gas in a micro-block would allow, then state channels come to the rescue.

Conceptually one can regard a state channel as an agreement between two parties, futher called *participants*, to build a chain of state changes on the side. Only the participants have access to this chain in the so called *state channel*. The parties agree on the initial state of this chain. Typically, the state would contain some initial accounts for the involved parties in which they reserve some tokens from the Æternity blockchain for transactions in the channel. Then they post a joinedly signed transaction on-chain to confirm that this is the initial state. The initial amounts are then reserved on-chain and cannot be used on-chain until they are released.

Apart from initial amounts, the participants also post the hash of the channel state-tree: the state the parties do agree upon. By only submitting the hash, the actual state is kept private to the channel, both parties know, but they need not reveal it.

If only accounts are used in a state channel, one can consider it a payment channel, a simpler purpose-specific form of a state channel. However, one can also agree upon a contract in the channel, either initially, or added later. In this way, one can also perform contract calls to update the state of the channel.

The state representation is determined by the state channel implementation and different implementations may in theory use their own representation. The state hash is what is visible on-chain and what parties agree upon in their transactions. In case of dispute resolution, the latest agreed, mutually signed, state hash can be posted to the Æternity blockchain.

3.1 On-chain and off-chain transactions

In a state channel users communicate peer-to-peer with each other, using mutually signed transactions that include state hashes. In this way parties agree upon a state, without any mining involved. Instead of creating key or micro-blocks, the transactions are just directly applied to update the state. Only states jointly agreed upon are recorded. This also means that there are no transaction or gas costs involved[2] and most notably that a transaction is confirmed as quickly as both parties can sign it. This confirmation time can therewith be reduced to milliseconds.

A typical use-case for a state channel, or rather a payment channel, is a subscription model, for example a coffee "account" with 25 cups of coffee. Each time one orders a coffee, the coffee shop creates a transaction to update the state and both shop and customer sign this transaction. They agree upon the new state. Since it would be annoying to have to wait 3 minutes or even 3 seconds on the payment of a cup of coffee, it is a good example for off-chain processing. At the same time, it also addresses privacy concerns, since only the involved parties and not all the users of the blockchain need to monitor the customers coffee consumption.

Scaling is achieved by performing transactions *off-chain*, i.e. transaction in the channel. One starts with an *on-chain* transaction agreeing and verifying a starting state and balance. After that many transaction can be performed without involving the Æternity blockchain. A coffee shop could have thousands of customers, all buying coffee and none of this puts transaction pressure on the blockchain. Only after terminating or topping up a subscription, a new on-chain transaction should be recorded. Having a thousand cups of coffee paid per second is no technical limitation, but possibly hard to achieve as a business.

The coffee subscription only includes two accounts and off-chain payments from one of these accounts to the other. Notably one can create contracts in a channel or refer in a channel to contracts created on the blockchain. We discuss contracts in Sect. 4, but one use case is to implement a game-playing contract (e.g. tic-tac-toe) and refer to that contract when playing the game in a channel. In such use-cases it is also an advantage to be able to play quickly and not having to wait 3 seconds before a participant's move is included in a transaction.

3.2 Disputes

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 a state channel is to mutually agree upon a withdrawal from the channel to an account of one of the parties. Typically, topping up a subscription would be done in combination with the coffee shop withdrawing part of the funds in the channel.

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

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

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.

Clearly, there is 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.

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 the above mentioned tic-tac-toe game contract, where the funds are re-distributed only after that 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.

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 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 a state has to be revealed that can be used by the Æternity blockchain to execute the next step in the contract. This requires to folllow a predefined format for the state-trees.

Clearly, a number of different counter measures are present for the other party in case a malicious force progress transaction is posted. It is also clear that the state of the state channel has to be made public (at least partly) to settle a contract dispute in a force progress transaction.

Users should keep in mind that, by using a state channel, they trade transaction efficiency and lower transaction fees for some reduction in safety. The on-chain transactions are under consensus, but on top of that, several implementations can be build that offer access to the state channel with their own implementation logic. For users of a state channel, it is important to know and understand the limitations of these off-chain implementations. For example, whether they monitor on-chain transactions that influence the state channel, such as a direct solo-close transaction on chain. Despite, a plurality of dispute handling primitives that are offered to mitigate the problems dealing with potentially malicious parties, users may not have the knowledge to use these primitives without a third party implementation. Therefore it is important to users to understand whether and how they are implemented.

4 Sophia smart contracts

Smart contracts [29, 12] 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 written in a programming language and for the aeternity a totally new language has been designed, called *Sophia*. There is a compiler that compiles Sophia contracts to byte code that is executed by the virtual machine part of the blockchain.

Contracts are put on chain by contract create transaction, specifying their byte code and data to compute the initial state. A contract on chain is called by contract call transactions[3]. Each contract creation and contract call requires computation effort and the user is paying for that effort by paying for *gas*. Each operation is associated with a certain amount of gas and the total used gas is paid for. The user specifies how much gas is provided in the transaction. A surplus is returned. If there is too little gas provided, the transaction is accepted (it is recorded on the chain), and the user pays for transaction fee and gas, but the contract computation has failed and its state is unchanged.

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 [21, 10].

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 to 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, 23, 13] have been studied with the goal to avoid the possibility to make such mistakes in Sophia.

4.1 The Sophia language

Sophia is a functional language [17]. 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

[3] in Aeternity contracts are executed by a call, there is no mechanism to automatically progress computation.

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.

4.1.1 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 on that height; 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 [27]. 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[4] is correctly paying the right price. Moreover, it is 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

4.2 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.

4.2.1 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

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

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 carefully 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 [26]). 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.

4.2.2 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 looks in itself as 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 also be run on IoT devices (cf. [14]) as well as to be able to get more computation into a micro-block.

5 Transaction types

The Æternity blockchain offers a plurality of different transactions types, designed to simplify application development for features that are common or popular in the blockchain sphere. By using smart contracts, users can develop completely new features themselves, but for already identified features, a general set of transactions is provided. These features are spending tokens from one account to another (Sect. 5.1), naming objects (Sect. 5.2), oracles (Sect. 5.3), and the transaction wrappers: generalized accounts (Sect. 5.4), and paying for (Sect. 5.5). There are also transactions for creating and calling contracts, as explained in Sect. 4 and for opening and closing state channels, deposit and withdraw funds from a state channel as well as solo closing, slash and force progress to deal with disputes as mentioned in Sect. 3.

All transactions are serialized and then cryptographically signed, by default using EdDSA [9] signatures with elliptic curve Curve25519 [8]. In order to protect against double spending, each transaction also includes a nonce, which is a stricly increasing counter connected to the account of the signer.

5.1 Aeternity accounts

The most basic transaction is a **spend transaction** that is used to transfer tokens from one account to another. An account is identified by the public key of the signer of a transaction. The spend transaction also serves as the basis for creating new accounts, by accepting any new recipient public key as a new account.

Posting a transaction to the chain can fail for many reasons, among which that the transaction is not correctly signed. But even correctly signed transactions may fail to become part of the chain.

Each transaction specifies a *fee* that the miner will eventually collect when including the transaction in a micro-block. This fee can be unattractive, in which case the transaction will stay in the transaction pool until it expires. Or the fee can be lower than the minimum fee defined by consensus, in which case it is an erroneous transaction that will be rejected. Including a transaction that is by consensus to be rejected, such as a transaction with lower fee than lowest agreed upon, is called fraud. The micro-block is invalid and rejected by all other nodes in the network[5].

Each transaction specifies a *nonce* to prevent it from replay attacks [28]. Each new account starts with nonce 1 and as soon as a transaction with that nonce is accepted on chain, only transactions with nonce 2 are accepted, etc. A user can post a number of different transactions with the same nonce in which case it is non-deterministic which of these transactions will result on chain, but only one of them will be accepted and the others thereafter rejected. This can be used as a feature by reposting a transaction with an unattractive fee with the same nonce and higher fee. Similarly, a user can post a whole series of transactions with increasing, but too large nonces. Only when the missing nonce is posted, all other transactions that possibly remained in transaction pools are enabled.

5.2 Aeternity naming system

Transferring tokens to a registered name instead of a hard to remember public key, is a feature that is supported natively by Aeternity. For example, in a spend transaction the recipient can be given as a name. For that to work, a collection of 5 transactions are provided that resemble internet domain name (DNS) registration.

One of the challenges in name registration is to offer a reasonably fair system for those that want a specific name. Imagine a user wants to reserve *emin* as a name. (Since there is currently only one name space on the Æternity blockchain *chain*, technically the user wants to claim *emin.chain*.) A short name of size 4, like *emin*, is a name that possibly many people like to claim. Just posting this name in a transaction would reveal to everyone monitoring the transaction pool that the name is attractive to at least someone. It is rather easy to immediately post the same name with a higher fee to have the leader pick that new transaction instead of the already posted request for the name *emin*. This is called *front running*, getting your transaction in front of

[5]Technically posting invalid micro-blocks is sometimes possible and a proof-of-fraud transaction is posted later to punish the node posting this transaction.

an already posted transaction by paying a larger fee. Transaction pools are not under consensus, so there are no guarantees that front running would work, but if there are many transaction in the system and one does this within the micro-block creation time of 3 seconds, there is a fair chance that front running would work.

In order to defend the users against front running, the first step in reserving a name is the **preclaim transaction**. In a pre-claim, one posts the hash of a combination of the name and a random number (called *salt*). When the preclaim is accepted, you can post a claim transaction to obtain the actual name. The **claim transaction** is then used to either immediately obtain the name if the name is long enough (with current governance values, longer than 12 characters), or to start an auction. In both cases the claim transaction reveals the name and the salt.

A fee is paid for the length of the name, shorter names are more expensive and the auction is open for a certain period expressed in blocks. For the 4 character name *emin*, the price starts with 134.6269 AE tokens and the auction would be open 29760 blocks (approx 62 days) after the last valid bid.

At the moment that someone posts a claim, the name is known. A potential front running by quickly posting both a preclaim and a claim for the same name is mitigated by demanding the preclaim and the claim to be in different generations. In other words, the original claim can be added to the generation, whereas the new claim needs to wait until the next generation.

After that the name is claimed, other users can see this name and claim it with a higher bid. Each bid must be at least 5% higher than the previous bid in order to be a valid bid. Note that the bidder must have enough tokens and that those tokens are reserved in a claim. The previous bidder gets the tokens returned as soon as a higher bid is accepted. This means that this users has the funds available for a possible next bid.

It is very well possible that two users preclaim the same name with different salt and then claim, but only one of them is accepted, the other is not. The rejected name claim transaction is not even seen as the next bid in an auction, even if the price would be higher. There is a subtle difference between a bidding claim and the original claim by the *salt* being zero for a bid and non-zero for an original claim.

After the auction, or for long names instantaneously, the highest bidder owns the name. An additional **update transaction** is needed to point the name to something (for example an account). Additionally there are a **transfer transaction** to change the owner of a name, and a **revoke transaction** to free the name.

Names have a specific lifetime and there is an agreed maximum number of blocks one can wait between a pre-claim and a claim in order to succeed with the claim. Obviously, registered names expire after a while, unless renewed in time with the 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. The name hash function first converts the name to a unique value using the internationalized domain name standard IDNA [20]. This makes names case-insensitive and, in particular, it helps to uniquely map non-ascii characters. After that, the Blake2b hash is applied. This unique hash is then part of the transaction and the way the name is internally represented. IDNA is used, since there are well known attacks by using names that look similar to the human eye, but are different.

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.

5.3 Aeternity oracles

Smart contracts only operate on data that is on the blockchain. Oracles are a mechanism to bring external data about real-world state and events onto the blockchain. Data can either be obtained from large data sources, real-time data, or heavy computations.

Typical external data may be useful for smart contracts. One can base a decision in a contract on the state of some external data. This can be sensor data or news events such as stock data, results of a match, supply chain data, etc. Researchers try to address the issue of trust in the authenticity of extrenal data [34, 16, 1], 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 ands in what format it is going to respond. Typically this is specified as a type signature. 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. The query fee is the economic incentive for the oracle to provide information.

When an oracle is registered on chain, any user can post a **query oracle transaction** with the rightly formatted query. The oracle supplier monitors the blockchain and will see this query and post an **oracle response transaction** with the answer to the query in the predefined format. In this way, the data becomes part of the data on the blockchain. This data can be referred to in a smart contract.

5.3.1 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.

5.3.2 Off-chain computations

Oracles can also be used to perform heavy computations off-chain and then post the actual result on-chain. After all, Sophia and the amount of gas available would make it impossible to implement a chess calculator to propose the next good move. Implementing this as an oracle would work. One queries for a certain position and gets a best move response. Clearly, the chess hints are already freely available, which harms this business model more than the fact that hints for specific positions becomes public data.

5.3.3 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**.

5.3.4 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[6].

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,
      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" )
```

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

```
    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[7], 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[8] 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) =
     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" )
```

[7]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

[8]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

```
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. An easier way to get a reasonable random number is to use the hash of the key-block at closing height.

5.4 Generalized accounts

Generalized accounts are a way to provide more flexibility to signing transactions. Both the nonce handling and the signature checking are done by a smart contract that is attached to the account. This can, for example, be useful when one would allow users to sign transactions with other cryptographic primitives than the default[9] EdDSA as mentioned in Sect. 5.

If a user wants to have a generalized account, then this user 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 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.

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

The following smart contract is an example that allows signing with the ECDSA algorithm [18] and the popular elliptic curve Secp256k1, used for example by Bitcoin and Ethereum [11, 22].

```
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.

5.4.1 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.

5.5 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 Æeternity 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 aeternity 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.

6 Experience

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 refered 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.

The blockchain has slowly attracted more traffic and we can now see over 1000 transactions an hour on busy days (Fig. 2). This is much below the theo-

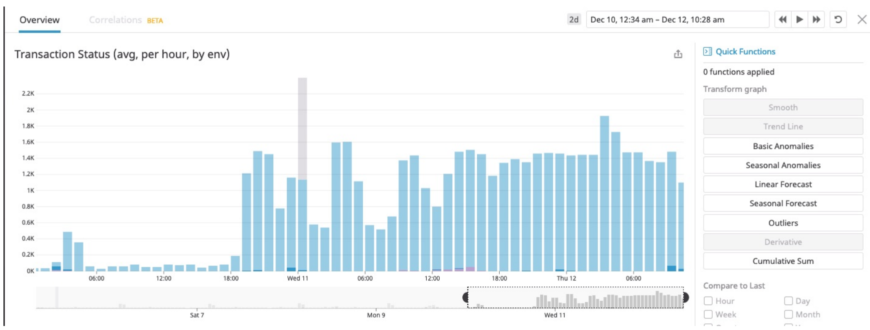

Figure 2: Transaction status

retical maximum of 100 on-chain transaction per second.

The confirmation time of transactions is the time between posting the transaction and seeing it in a micro-block on chain. We measure this over a longer time by posting a transaction each 3 seconds carrying the post time as timestamp in the payload. The micro-block in which this transaction appears also has a timestamp (set by the clock of the miner). The difference is observable for everyone, since these are transactions on chain. The mean of confirmation times is around the expected 3 seconds.

At height 183490 we had the following totals of different transaction types:

Transaction Type	count	Transaction Type	count
SpendTx	4117919	ChannelCloseMutualTx	15
GAAttachTx	2	ChannelCloseSoloTx	3
GAMetaTx	5	ChannelCreateTx	119
NameClaimTx	557012	ChannelDepositTx	2
NamePreclaimTx	605026	ChannelForceProgressTx	1
NameRevokeTx	1	ChannelSettleTx	2
NameTransferTx	59	ChannelSlashTx	1
NameUpdateTx	431037	ChannelSnapshotSoloTx	1
OracleExtendTx	276	ChannelWithdrawTx	1
OracleQueryTx	603	ContractCallTx	48063
OracleRegisterTx	25	ContractCreateTx	259
OracleResponseTx	590		

Which clearly shows that oracles are less popular than the naming system[10].

[10]Since the Lima release, name auctions are introduced, invalidating the names in the `.test`

It also shows that the state channels have not yet started to be widely used, but there are clearly some contracts that are heavily called.

7 Future ambitions

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-O'Hearn, 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] Bashir, I. *Mastering blockchain: Distributed ledger technology, decentralization, and smart contracts explained.* Packt Publishing Ltd, 2018.

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

[9] 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.

[10] 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.

domain. Most name transactions are from before this release.

[11] 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.

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

[13] 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.

[14] Ellul, J., and Pace, G. J. AlkylVM: A virtual machine for smart contract blockchain connected internet of things. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)* (2018), IEEE, pp. 1–4.

[15] 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.

[16] 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.

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

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

[19] Karame, G. O., Androulaki, E., and Capkun, S. Double-spending fast payments in bitcoin. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 906–917.

[20] Klensin, J. C. Internationalized domain names for applications (IDNA): Definitions and document framework. *RFC 5891–5894* (2010).

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

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

[23] 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.

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

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

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

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

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

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

[30] 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.

[31] Tsang, P. P., and Smith, S. W. Combating spam and denial-of-service attacks with trusted puzzle solvers. In *Information Security Practice and Experience* (Berlin, Heidelberg, 2008), Y. Chen, Liquncand Mu and W. Susilo, Eds., Springer Berlin Heidelberg, pp. 188–202.

[32] Wang, W., Hoang, D. T., Xiong, Z., Niyato, D., Wang, P., Hu, P., and Wen, Y. A survey on consensus mechanisms and mining management in blockchain networks. *arXiv preprint arXiv:1805.02707* (2018), 1–33.

[33] 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.

[34] 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.