Argennon: A Scalable Cloud Based Smart Contract Platform

aybehrouz

January 2021

Abstract

Argennon is a next generation cloud based blockchain and smart contract platform. The Argennon blockchain uses a hybrid proof of stake (HPoS) consensus protocol, which is capable of combining the benefits of a centralized and a decentralized system. In Argennon, ledger storage and transaction processing are outsourced to the cloud and normal personal computers or smartphones, with limited hardware capabilities, are able to validate transactions and actively participate in the Argennon consensus protocol. This property makes Argennon a truly decentralized and democratic blockchain and one of the most secure existing platforms.

The Argennon cloud is trustless and publicly verifiable. Computational Integrity (CI) is achieved by using succinct cryptographic proofs (STARK/SNARK) and data integrity is guaranteed by cryptographic accumulators. At the same time, a smart clustering algorithm keeps the bandwidth usage and the overhead of cryptography manageable for validators.

The Argennon protocol strongly incentivizes the formation of a permission-less network of Publicly Verifiable Cloud (PVC) servers. A PVC server in Argennon, is a conventional data server which uses its computational and storage resources to help the Argennon network process transactions.

Contents

Chapter 1

Introduction

The most common use for blockchains is in financial applications. This gives a crucial importance to the security of the consensus protocol used in a blockchain. Unfortunately, many currently used blockchains are vulnerable to a certain type of consensus attack, known as the bribery attack. In a bribery attack, an adversary tries to corrupt participants of a protocol by offering them money and making them violate the protocol.

At the time of writing of this document, the total mining reward for a bitcoin block is around $150,000. If we assume, in decision theoretic terminology, that the mining reward accurately defines the utility function of a bitcoin miner, one could hire all hashing power of the bitcoin network for one hour by spending only $750,000. The situation is not much different for PoS blockchains, as long as the total stake of the validator set is a relatively small value. Here by stake, we mean a real number measuring the total interest of a user in the system. We are not referring, in particular, to some locked amount of a user's money that is known as stake in some PoS protocols.

This problem is more severe in blockchains that use randomly selected small sets of validators. These small sets usually have low total stake and could be easily bribed and corrupted. Selecting these random sets by hidden random procedures would not help, since the validator himself knows he has been selected, before casting his vote.

It appears that the only solution to this important vulnerability is to effectively participate all the stakeholders of the system in the consensus protocol. This large participation makes the protocol resilient against bribery or collusion because the adversary would need to spend unrealistic amounts of money to bribe enough users.

However, for effective participation in the consensus protocol, a validator needs to be able to detect illegal transactions. Detecting illegal transactions can be done by accessing the ledger state and executing transactions according to the protocol rules. The ledger state, even for small blockchains, could be several hundreds of gigabytes, and executing transactions could easily become costly when a blockchain is acting as a smart contract platform. This computational and storage overhead, in practice, could prevent most of ordinary users from any type of participation in the process of securing a blockchain.

Although a fully decentralized blockchain based on the participation of every user looks appealing, it is not as perfect as it might seem. The consensus protocol of a blockchain relies on a network of computers, not humans. Ordinary users use simple and similar computer systems. That means, they all have similar vulnerabilities and weaknesses which could be used by an adversary to catastrophically attack the consensus protocol. For instance, if a malware, probably using a common zero-day vulnerability, has the ability to infect a large portion of normal personal computers, it could be used by an adversary to control the majority of participants in the consensus protocol and compromise the security of the blockchain.

Securing a computer system against cyberattacks needs planning ahead and access to engineering resources. Special software and hardware, like custom-built operating systems and isolated specialized hardware is required. This is not something a normal user can afford. Only powerful centralized entities having large financial and technical resources, could build systems that are resilient against sophisticated cyberattacks. In this regard, we have to admit, a centralized system is arguably superior to a decentralized[1] system.

To overcome these difficulties, Argennon[2] uses a Hybrid Proof of Stake consensus protocol, which is capable of combining the benefits of a centralized and a decentralized system. A small committee of delegates is democratically elected by users via the Argennon Decentralized Autonomous Governance system (ADAGs). This committee usually[3] is elected for a one-year term, has five members, and is responsible for minting new blocks of the Argennon blockchain. Each minted block is required to get approved by its corresponding validator committee. Validator committees are very large sets of validators including at least three percent of the maximum possible stake of the system. A block is approved if it takes approval votes from at least 2/3 of the total stake of its validator committee.

Every block of the Argennon blockchain needs a certificate from both the committee of delegates and the committee of validators. A block is considered final after its **next** block gets **both** of its certificates. The Argennon protocol ensures that as long as more than half of the total stake of validators is controlled by honest users, the probability of discarding a final block is near zero even if all the delegates are malicious.

Centralized block generation brings some interesting features to the Argennon platform, such as flexible and lower fees, off chain fee payment, optimistic instant transaction confirmation, and runoff protection. However, it also increases the possibility of transaction censorship. In Argennon, this problem is addressed by a special High Priority Request (HPR) protocol.

Each block of the Argennon blockchain contains a set of Computational Integrity (CI) statements and a commitment to the final ledger state of the block. The CI set defines how the final ledger state of the block can be reached from the state of its previous block via a set of intermediate states. Each individual CI statement defines an ordered list of

[1] Note that decentralized and distributed are two different concepts.

[2] The classical pronunciation should be used:/ɑrˈgen.non/

[3] The election term and the number of committee members can be changed by the ADAGs.

external requests[4] and determines the state before and after executing those requests:

$$\tau_{(s_i,s_{i+1},\mathbb{R})} \coloneqq \text{“}s_{i+1} \text{ is the next state after executing external requests } \mathbb{R} \text{ on state } s_i\text{”}$$

Validators can[5] receive succinct cryptographic proofs of these CI statements (STARK/SNARK) from the Argennon cloud. They can also receive the state of the previous block (the required part of it) and proofs that show the received data is consistent with the previous block commitment.

Verifying a succinct proof can be exponentially faster than replaying the computation. Moreover, verifications of different CI statements are independent of each other and can be done in parallel. As a result, a validator can use multiple cores for verifying CI statements of a **single** block and different validator committees can simultaneously and independently verify **different** blocks. In addition, proof generation of different CI statements similarly can be done in parallel. However, for parallel proof generation, the state transition needs to be known in advance. That's why in Argennon, delegates do not try to generate proofs and focus all their computational power on executing external requests and generating the state transition as fast as possible.

Argennon applications (i.e smart contracts) are stored in a high level text based language on the blockchain. This language is intended for preserving the high level information of the application logic to facilitate platform specific compiler optimization at a host machine. This enables delegates to compile and optimize Argennon applications for their specific hardware platforms and execute applications efficiently, ensuring the state transition can be found as fast as possible.

Independence of CI statements is useful, but is not enough for having a truly scalable blockchain. To increase parallelism, the Argennon protocol enforces all external requests to pre-declare their memory access locations. That would enable a block proposer[6] to use Data Dependency Analysis[7] (DDA) to indicate independent sets of external requests (i.e. transactions) and use those sets for parallel processing. More importantly, these sets can be used for generating CI statements that are defined on the **same initial state** and their proof can be generated independently without the need to calculate the state transition in advance.

The Argennon protocol strongly incentivizes the formation of a **permission-less** network of Publicly Verifiable Cloud (PVC) servers. To do so, the Argennon protocol conducts repetitive automatic lotteries between PVC servers. A PVC server can increase its chance of winning by (i) generating proofs for more CI statements, (ii) providing the state data to more validators and, (iii) storing all parts of the state instead of more frequently used parts.

A PVC server in Argennon, is a conventional data server which uses its computational and storage resources to help the Argennon network process transactions. This encourages the development of conventional networking, storage and compute hardware,

[4]External requests in Argennon are similar to transactions in older blockchains.

[5]Using the Argennon cloud is optional for a validator.

[6]In Argennon, delegates are the only block proposers.

[7]See Section 2.11

which can benefit all areas of information technology. This contrasts with the approach of some older blockchains that incentivized the development of a totally useless technology of hash calculation.

Chapter 2

The Argennon Smart Contract Execution Environment

This chapter is outdated and does not include CI proofs. Currently a new version is being rewritten.

2.1 Introduction

The Argennon Smart Contract Execution Environment (AscEE) is an abstract execution environment for executing Argennon smarts contracts (a.k.a Argennon applications) in a safe and secure environment. An Argennon application essentially is an HTTP server whose state is kept in the Argennon blockchain and its logic is described using an Argennon Standard Representation (ASR).

An Argennon Standard Representation (ASR) is a programming language for describing Argennon applications, optimized for the architecture and properties of the Argennon platform. Argennon supports two standard representations: One is a high level text based representation which needs costly compilation before being executed on a hardware machine. The other is a low level binary representation which usually can be executed, with minimal pre-processing, by a JIT compiler or an emulator. The high level language is intended for preserving the high level information of the application logic to facilitate platform specific compiler optimization at a host machine. The low level language, on the oder hand, is designed for efficient direct execution of applications that are not frequently used and do not benefit from compiler optimizations.

The state of an Argennon application is stored in byte addressable finite arrays of memory called *heap chunks*. An application may have several heap chunks with different sizes, and can remove or resize its heap chunks or allocate new chunks. Every chunk belongs to exactly one application and can only be modified by its owner. In addition to heap chunks, every application has an amount of non-persistent local memory for storing temporary data.

The AscEE executes the requests contained in each block of the Argennon blockchain in a three-step procedure. The first step is the *preprocessing step*. In this step, the required data for executing requests are retrieved from the Argennon cloud, the cryptographic proofs are verified and the helper data structures for next steps are constructed. This step is designed in a way that can be done fully in parallel for each request without any risk of data races. The second step is the *Data Dependency Analysis (DDA) step*. In this step by analyzing data dependency between requests, the AscEE determines requests that can be run in parallel and requests that need to be run sequentially. This information is represented using an *execution DAG* data structure and in the final step, the requests are executed using this data structure.

2.2 Execution Sessions

The Argennon Smart Contract Execution Environment can be seen as a machine for executing Argennon applications to fulfill HTTP requests, produce their HTTP responses and update related heap chunks. The execution of requests can be considered sequential[1] and each request has a separate *execution session*. Therefore, an execution session is a separate session of executing smart contract's code in order to fulfill an *external* HTTP request. External requests are requests that are not made by other Argennon applications.

The state of an execution session will be destroyed at the end of the session and only the state of heap chunks is preserved. If a session fails and does not complete normally, it will not have any effect on any heap chunks.

During an execution session an application can make *internal* HTTP requests to other applications. Those requests will not start a new execution session and will be executed within the current session. In AscEE making an HTTP request to an application is similar to a function invocation, and for that reason, we also refer to them as application calls.

The AscEE is designed based on *optional decoupling principle*. When an application makes a request to another application, optionally it can choose to be decoupled from the called application. That would mean the called application could not affect its caller's state by reentrancy, or could not abort the session by using excessive resources or performing illegal operations.

2.3 Identifiers

In Argennon a unique identifier is assigned to every application, heap chunk and account. Consequently, three distinct identifier types exist: `appID`, `accountID`, and `chunkID`. All these identifiers are *prefix codes*, and hence can be represented by *prefix trees*[2].

[1] Actually requests are executed in parallel but by performing data dependency analysis the result is guaranteed to be identical with sequential execution of requests.

[2] Also called tries.

Algorithm 1: Finding a prefixed identifier

input : A sequence of n digits in base β: $d_1 d_2 \dots d_n$
A prefix tree: $< A^{(1)}, A^{(2)}, A^{(3)}, \cdots >$

output: Valid identifier prefix of the sequence.

for $i = 1$ **to** n **do**
    if $(0.d_1 d_2 \dots d_i)_\beta < A^{(i)}$ **then**
        return $d_1 d_2 \dots d_i$
    end
end
return *NIL*

Argennon has four primitive prefix trees: *applications, accounts, local* and *varUint*. All these trees are in base 256, with the maximum height of 8.

An Argennon identifier may be simple or compound. A simple identifier is generated using a single tree, while a compound identifier is generated by concatenating prefix codes generated by two or more trees:

- `appID` is a prefix code built by *applications* prefix tree. An `appID` cannot be `0x0`.
- `accountID` is a prefix code built by *accounts* prefix tree. An `accountID` cannot be `0x0` or `0x1`.
- `chunkID` is a composite prefix code built by concatenating an `applicationID` to an `accountID` to a prefix code made by *local* prefix tree:

  `chunkID = (applicationID|accountID|<local-prefix-code>)` .

All Argennon prefix trees have an equal branching factor β[3], and we can represent an Argennon prefix tree as a sequence of fractional numbers in base β:

$$(A^{(1)}, A^{(2)}, A^{(3)}, \dots) ,$$

where $A^{(i)} = (0.a_1 a_2 \dots a_i)_\beta$, and we have $A^{(i)} \leq A^{(i+1)}$. [4]

One important property of prefix identifiers is that while they have variable and unlimited length, they are uniquely extractable from any sequence. Assume that we have a string of digits in base β, we know that the sequence starts with an Argennon identifier, but we do not know the length of that identifier. Algorithm 1 can be used to extract the prefixed identifier uniquely. Also, we can apply this algorithm multiple times to extract a composite identifier, for example `chunkID`, from a sequence.

When we have a prefixed identifier, and we want to know if a sequence of digits is marked by that identifier, we use Algorithm 2 to match the prefixed identifier with the

[3] A typical choice for β is 2^8 .

[4] It's possible to have $a_i = 0$. For example, $A^{(4)} = (0.2000)_{10}$ is correct.

start of the sequence. The matching can be done with only three comparisons, and an invalid prefixed identifier can be detected and will not match any sequence.

In Argennon the shorter prefix codes are assigned to more active accounts and applications which tend to own more data objects in the system. The prefix trees are designed by analyzing empirical data to make sure the number of leaves in each level is chosen appropriately.

Algorithm 2: Matching a prefixed identifier

---

input : A prefixed identifier in base β with n digits: $id = a_1a_2 \ldots a_n$
A sequence of digits in base β: $d_1d_2d_3 \ldots$
A prefix tree: $< 0, A^{(1)}, A^{(2)}, A^{(3)}, \cdots >$

output: *TURE* if and only if the identifier is valid and the sequence starts with the identifier.

if $(0.a_1 \ldots a_n)_\beta = (0.d_1 \ldots d_n)_\beta$ **then**
    if $A^{(n-1)} \leq (0.a_1a_2 \ldots a_n)_\beta < A^{(n)}$ **then**
        return *TRUE*
    end
end
return *FALSE*

2.4 Heap Chunks

The persistent data of an Argennon application is stored in continuous finite arrays of bytes, called heap chunks. An application may have several heap chunks with different sizes, and can remove or resize its chunks or allocate new chunks. Every chunk belongs to exactly one application. Only the owner application can modify a chunk but there is no restrictions for reading a chunk[5].

When an application allocates a new heap chunk, the identifier of the new chunk is not generated by the AscEE. Instead, the application can choose an identifier itself, provided it has a correct format. This is an important feature of the AscEE heap, which allows applications to use the AscEE heap as a dictionary data structure[6]. Since the `chunkID` is a prefix code, any application has its own identifier space, and an application can easily find unique identifiers for its chunks.

[5]The reason behind this type of access control design is the fact that smart contract code is usually immutable. That means if a smart contract does not implement a getter mechanism for some parts of its internal data, this functionality can never be added later, and despite the internal data is publicly available, there will be no way for other smart contracts to use this data on-chain.

[6]also called a map.

2.4.1 Chunk Resizing

At the start of executing requests of a block, a validator can consider two values for every heap chunk, the size: `chunkSize` and a size upper bound: `sizeUpperBound`. The value of `chunkSize` can be determined uniquely at the start of every execution session, and it may be updated during the session by the owner application. On the other hand, the value of `sizeUpperBound` is determined and fixed for each block and is proposed by the block proposer. This value indicates the upper bound of `chunkSize` during the execution of requests of the block, and it can be used by the validator for safe memory allocation.

The value of `chunkSize`, can be modified during an execution session. However, the new size values can only be increasing or decreasing. More precisely, if a request declares that it wants to expand (shrink) a chunk, it can only increase (decrease) the value of `chunkSize` and any specified value during the execution session, needs to be greater (smaller) than the previous value of the chunk's size. Any request that wants to expand (shrink) a chunk needs to specify a max size (min size). The value of `chunkSize` can not be set higher (lower) than this value.

The value of `chunkSize` at the end of an execution session will determine if a memory location at an offset is persistent or not: Offsets lower than the chunk size are persistent, and higher offsets are not. Non-persistent locations will be re-initialized with zero at the start of every execution session.

Usually an application should not have any assumption about the content of memory locations that are outside the chunk. While these locations are zero initialized at the start of every execution session, it should be noted that multiple invocations of an application may occur in a single execution session, and if one of them modifies a location outside the chunk, the changes can be seen by next invocations.

There is no way for an application to query `sizeUpperBound` of a chunk. As a result, for an application, accessing offsets higher than `chunkSize` results in undefined behaviour, while the behaviour is well-defined in the view of validators. This enables validators to determine the validity of an offset at the start of the block validation in a parallelized preprocessing phase without actually executing requests.

While an application can use `chunkSize` to determine if an offset is persistent or not, that is not considered a good practice. Reading `chunkSize` decreases transaction parallelization, and should be avoided. Instead, applications should use a built-in AscEE function for checking the persistence status of memory addresses.

An application can load any chunk with a valid prefix identifier even if that chunk does not exist. For a non-existent chunk the value of `chunkSize` is always zero.

The address space of a chunk starts from zero and only offsets lower than `sizeUpperBound` are valid. Trying to access any offset higher than this value will always result in a revert for the application.

2.4.2 Access Blocks

Memory locations inside a chunk can only be accessed through access blocks. An access block is an access interval defined on a chunk, which determines the accessible memory locations inside the chunk. Each access block has an offset and a size. Multiple access blocks can be defined on a single chunk, but they must be non overlapping. Access blocks are byte addressable and may have different access types:

- `check_only`: only allows check operations. These operations query the persistence status of a memory location.
- `read_only`: only allows read and check operations.
- `writable`: allows reading and writing.
- `additive`: only allows additive operations. These operations are addition-like operations without any overflow checking. These operations must be associative and commutative. Note that the content of these access blocks cannot be read.

Chunks and access blocks have different purposes. Chunks are intended for simplifying proof checking of the data stored in the Argennon cloud[7] and access blocks are required for better parallelization of the request execution. An application should put the data needed for validating a single block in the same chunk and the data needed in a single execution session in the same access block.

2.5 Request Attachments

The attachment of a request is a list of request identifiers of the current block that are "attached" to the request. That means, for validating the request a validator first needs to "inject" the digest of attached requests into the HTTP request text. By doing so, the called application will have access to the digest of attachments in a secure way.

The main usage of this feature is for fee payment. A request that wants to pay the fees for a number of requests, declares those requests as its attachments. For paying fees the payer signs the digest of requests for which he wants to pay fees. After injecting the digest of those request by validators, that signature can be validated correctly and securely by the application that handles fee payment.

2.6 Authorizing Operations

In blockchain applications, we usually need to authorize certain operations. For example, for sending an asset from a user to another user, first we need to make sure that the sender has authorized this operation.

[7] Note that the Argennon cloud uses cryptographic accumulators for preserving data integrity.

The AscEE uses *Authenticated Message Passing* for authorizing operations. In this method, every execution session has a set of authenticated messages, and those messages are **explicitly** passed in requests to applications for authorizing operations. These messages act exactly like digital signatures and applications can ensure that they are issued by specific accounts. The only difference is that the process of message authentication is performed by the AscEE internally and is not performed by the application. For accounts, the AscEE uses digital signatures to authenticate messages. Signatures for each request can be validated in parallel during the preprocessing step of the block validation, and any type of cryptographic signature scheme can be used. For example, we could use BLS aggregate signatures to authenticate all messages of a block of the blockchain in bulk.

Moreover, applications can use built-in functions of the AscEE to generate authenticated messages in run-time. This enables an application to authorize operations for another application even if it is not calling that application directly.

In addition to authenticated messages, the AscEE provides a set of cryptographic functions for validating signatures and cryptographic entities. By using these functions and passing cryptographic signatures as parameters to methods, a programmer, having users' public keys, can implement the required logic for authorizing operations.

Authorization by explicit authenticated messages and signatures eliminates the need for approval mechanisms or call back patterns in Argennon.

The AscEE has no instructions for issuing cryptographic signatures.

2.7 Reentrancy Protection

The Argennon Smart Contract Execution Environment provides optional low level reentrancy protection by providing low level *entrance locks*. When an application acquires an entrance lock it cannot acquire that lock again and trying to do so will result in a revert. The entrance lock of an application will be released when the application explicitly releases its lock or when the call that had acquired that lock completes.

The AscEE reentrancy protection mechanism is optional. An application can allow reentrancy, it can protect only certain areas of its code, or can completely disallow reentrancy.

2.8 Deferred Calls

...

2.9 Resource Management

Completing an execution session requires computational resources. The amount of resources used by an execution session should be monitored and managed, otherwise a malicious user would be able to easily spam and exhaust resources of the execution environment.

In most consensus protocols, we can assume that the block proposer has enough incentive to filter out transactions that spam run-time resources. Here by a run-time resource, we mean a resource that at run-time, a limited amount of it is available, but its surplus can not be stored for later use. Execution time and local memory are examples of such a resource but permanent storage is not.

If a proposed block contains many transactions which need a lot of run-time resources, validators would not be able to validate all transactions in a timely manner. Consequently, they may decide to reject the block or if they spend enough resources, the confirmation of that block could take more than usual. Longer block time is not favoured by block proposers, because it means less throughput of the system which usually means less overall rewards for them. In the Argennon protocol the management of run-time resources, is left to the block proposer.

Resource usage can be measured per session or per application call. Obviously per session measurement is easier and more efficient. However, when we measure resources per session if a session violates its resource caps, determining the point of failure may require precise and error-free resource measurement. For example, assume that a session containing an application call violates a 2 milliseconds execution time cap. If in the caller's code, the call happens exactly after 2 milliseconds, a small fluctuation in the execution time measurement can change the point of failure between the called and the caller application. Note that for implementing optional decoupling principle in Argennon, we need to determine the exact application call which has failed in a call chain, and this fluctuation introduces a nondeterministic behaviour which could make block validation impossible. That's why we need per application call resource management sometimes. If we define the resource caps per application call and perform our measurements for each application call separately, by using caps that are larger than the measurement error, errors in the measurements would not change the point of failure in the call chain.

The AscEE has two types of execution sessions: *optimistic* and *monitored*. Resource usage of an optimistic session is always measured per session and **default** pre-defined resource caps are used. On the other hand, for a monitored session, resources that can not be measured precisely are measured per application call and their caps are determined per application call by the external HTTP request (i.e. transaction). The block proposer decides the type of the execution session for each transaction.

Different computational resources are measured and monitored during an AscEE session:

- **execution time**: is the amount of cpu time that is required for executing a session or an application call. The execution time is measured in *AscEE clocks*. One AscEE clock is defined as 1/1000 of the amount of **cpu time** needed for executing a predefined standard application which is used for benchmarking a host's performance. In other words, by definition, the AscEE imaginary standard machine completes the standard benchmark in 1000 clocks.

Optimistic sessions have a predefined `maxClocks` value which is determined by the Argennon protocol. This value defines a bound on the **total** AscEE clocks of the session, and no per application measurement is done.

Monitored sessions perform per application call cpu-time measurement, and every application call during a monitored session has a separate `maxClocks` value. This value determines the maximum amount of time that the cpu can be used for executing that particular application call. It should be noted that the cpu timer of the application call is paused when the application makes a call to another application, and is resumed when the control returns. An application call needs at least 10 clocks and if the value of its `maxClock` is lower than this value the call will be considered a failed call.

Each application call has some amount of `externalClocks`. When an application makes a request to another application it has to *forward* a portion of its **external** clocks to the called application. This amount will determine the value of `maxClocks` for the called application, and is subtracted from the value of `externalClocks` of the caller.

The amount of external clocks of an application call is defined to be 2/3 of its `maxClocks`. As a result, the total number of clocks of a monitored session is always less than $3 \times$ `maxClocks` of the root application call. The value of `maxClocks` for the root application call is determined by the external request (i.e. transaction).

- **local memory**: any memory usage of an application that is not part of a heap chunk and is not part of another measured resource will be considered as local memory usage. Local memory is not persistent and when an application finishes serving a request and returns the HTTP response (i.e. the application call completes) its local memory is deleted. This resource is measured in bytes.

  Optimistic sessions measure local memory usage per session and enforce a protocol-defined cap on the total amount of local memory a session can use. Monitored sessions measure local memory usage per application call and enforce a protocol-defined cap for each application call separately. An application call which tries to use more local memory than the cap, fails.

- **heap access list**: every session can only access heap locations that are declared in its access list. In addition, resizing heap chunks can only be done in the range of the pre-declared lower bound and upper bound.

- **app access list**: a session may only make requests to applications that are declared in its application access list.

- **call depth**: during a session the number of nested application calls can not be more than a threshold. This threshold is determined by the Argennon protocol. It should be noted that a differed call is considered like a normal call and increases the call depth by one level.

- **differed calls**: every application call can have a limited number of differed calls which is determined by the protocol. To simplify the implementation, this limit is defined per application call. Since in Argennon the call depth is limited, a per-application call limit will also define an implicit limit for the total number of active differed calls.
- **virtual signatures**:
- **number of entrance locks**:

In Argennon, execution time and local memory are considered *nondeterministic* resources. A nondeterministic resource is a resource that can not be measured precisely and its measurement always contains a random error. Optimistic sessions are not allowed to fail because of violating nondeterministic resource limits. As a result, the block proposer must always choose a monitored session for a transaction that is included in the block and violates a nondeterministic resource cap.

When a transaction fails due to the violation of a limit for a nondeterministic resource, the proposer is required to exactly specify the application call which violates that limit in the execution session. When validators are executing a monitored session, for each application call, they enforce considerably larger limits for nondeterministic resources. Only in case the proposer has declared that an application call violates a limit, the validators will enforce the actual value of the limit. This simple mechanism ensures that, with a very high probability, validators agree with the proposer, although their measurements could be different.

2.10 The ArgC Language

2.10.1 The ArgC Standard Library

In Argennon, some applications (smart contracts) are updatable. The ArgC Standard Library is an updatable smart contract which can be updated by the Argennon governance system. This means that bugs or security vulnerabilities in the ArgC Standard Library could be quickly patched and applications could benefit from bugfixes and improvements of the ArgC Standard Library even if they are non-updatable. Many important and useful functionalities, such as fungible and non-fungible assets, access control mechanisms, and general purpose DAOs are implemented in the ArgC Standard Library.

All Argennon standards, for instance ARC standard series, which defines standards regarding transferable assets, are defined based on how a contract should use the AVM standard library. As a result, Argennon standards are different from conventional blockchain standards. Argennon standards define some type of standard logic and behaviour for a smart contract, not only a set of method signatures. This enables users to expect certain type of behaviour from a contract which complies with an Argennon standard.

2.11 Data Dependency Analysis

2.11.1 Memory Dependency Graph

Every block of the Argennon blockchain contains a list of transactions. This list is an ordered list and the effect of its contained transactions must be applied to the AscEE state sequentially as they appear in the ordered list. This ordering is solely chosen by the block proposer, and users should not have any assumptions about the ordering of transactions in a block.

The fact that block transactions constitute a sequential list, does not mean they can not be executed and applied to the AscEE state concurrently. Many transactions are actually independent and the order of their execution does not matter. These transactions can be safely validated in parallel by validators.

A transaction can change the AscEE state by modifying either the code area or the AscEE heap. In Argennon, all transactions declare the list of memory locations they want to read or write. This will enable us to determine the independent sets of transactions which can be executed in parallel. To do so, we define the *memory dependency graph* G_d as follows:

- G_d is an undirected graph.
- Every vertex in G_d corresponds to a transaction and vice versa.
- Vertices u and v are adjacent in G_d if and only if u has a memory location L in its writing list and v has L in either its writing list or its reading list.

If we consider a proper vertex coloring of G_d, every color class will give us an independent set of transactions which can be executed concurrently. To achieve the highest parallelization, we need to color G_d with minimum number of colors. Thus, the *chromatic number* of the memory dependency graph shows how good a transaction set could be run concurrently.

Graph coloring is computationally NP-hard. However, in our use case we don't need to necessarily find an optimal solution. An approximate greedy algorithm will perform well enough in most circumstances.

After constructing the memory dependency graph, we can use it to construct the *execution DAG* of transactions. The execution DAG of transaction set T is a directed acyclic graph $G_e = (V_e, E_e)$ which has the *execution invariance* property:

- Every vertex in V_e corresponds to a transaction in T and vice versa.
- Executing the transactions of T in any order that *respects* G_e will result in the same AscEE state.
  - An ordering of transactions of T respects G_e if for every directed edge $(u, v) \in E_e$ the transaction u comes before the transaction v in the ordering.

Having the execution DAG of a set of transactions, using Algorithm 3, we can apply the transaction set to the AscEE state concurrently, using multiple processor, while we can be sure that the resulted AscEE state will always be the same no matter how many processor we have used.

Algorithm 3: Executing DAG transactions

Data: The execution dag $G_e = (V, E)$ of transaction set T
Result: The state after applying T with any ordering respecting G_e

```
R_e ← the set of all vertices of V with in degree 0
while V ≠ ∅ do
    wait until a new free processor is available
    if the execution of a transaction was finished then
        remove the vertex of the finished transaction v_f from G_e
        for each vertex u ∈ Adj[v_f] do
            if u has zero in degree then
                R_e ← R_e ∪ u
            end
        end
    end
    if R_e ≠ ∅ then
        remove a vertex from R_e and assign it to a processor
    end
end
```

By replacing every undirected edge of a memory dependency graph with a directed edge in such a way that the resulted graph has no cycles, we will obtain a valid execution DAG. Thus, from a memory dependency graph different execution DAGs can be constructed with different levels of parallelization ability.

If we assume that we have unlimited number of processors and all transactions take equal time for executing, it can be shown that by providing a minimal graph coloring to Algorithm 4 as input, the resulted DAG will be optimal, in the sense that it results in the minimum overall execution time.

The block proposer is responsible for proposing an efficient execution DAG alongside his proposed block. This execution DAG will determine the ordering of block transactions and help validators to validate transactions in parallel. Since with better parallelization a block can contain more transactions, a proposer is incentivized enough to find a good execution DAG for transactions.

2.11.2 Memory Spooling

When two transactions are dependant and they are connected with an edge (u, v) in the execution DAG, the transaction u needs to be run before the transaction v. However, if v does not read any memory locations that u modifies, we can run u and v in parallel. We

Algorithm 4: Constructing an execution DAG

input : The memory dependency graph $G_d = (V_d, E_d)$ of transaction set T
A proper coloring of G_d

output: An execution dag $G_e = (V_e, E_e)$ for the transaction set T

```
V_e ← V_d
E_e ← ∅
define a total order on colors of G_d
for each edge {u, v} ∈ E_d do
    if color[u] < color[v] then
        E_e ← E_e ∪ (u, v)
    else
        E_e ← E_e ∪ (v, u)
    end
end
```

just need to make sure u does not see any changes v is making in AscEE memory. This can be done by appropriate versioning of the memory locations which is shared between u and v. We call this method *memory spooling.* After enabling memory spooling between two transactions the edge connecting them can be safely removed from the execution DAG.

2.11.3 Concurrent Counters

We know that in Argennon every transaction needs to transfer its proposed fee to the `feeSink` accounts first. This essentially makes every transaction a reader and a writer of the memory locations which store the balance record of the `feeSink` accounts. As a result, all transactions in Argennon will be dependant and parallelism will be completely impossible. Actually, any account that is highly active, for example the account of an exchange or a payment processor, could become a concurrency bottleneck in our system which makes all transactions interacting with them dependant.

This problem can be easily solved by using a concurrent counter for storing the balance record of this type of accounts. A concurrent counter is a data structure which improves concurrency by using multiple memory locations for storing a single counter. The value of the concurrent counter is equal to the sum of its sub counters and it can be incremented or decremented by incrementing/decrementing any of the sub counters. This way, a concurrent counter trades concurrency with memory usage.

Algorithm 5 implements a concurrent counter which returns an error when the value of the counter becomes negative.

It should be noted that in a blockchain application we don't have concurrent threads and therefore we don't need atomic functions. For usage in a smart contract, the atomic functions of this pseudocode can be implemented like normal functions.

Concurrent counter data structure is a part of the ArgC standard library, and any

Algorithm 5: Concurrent counter

```
Function GetValue(Counter)
    s ← 0
    Lock.Acquire()
    for i ← 0 to Counter.size − 1 do
        s ← s + Counter.cell[i]
    end
    Lock.Release()
    return s

Function Increment(Counter, value, seed)
    i ← seed mod Counter.size
    AtomicIncrement(Counter.cell[i], value)

Function Decrement(Counter, value, seed, attempt)
    if attempt = Counter.size then
        restore Counter by adding back the subtracted value
        return Error
    end
    i ← seed mod Counter.size
    i ← (i + attempt) mod Counter.size
    if Counter.cell[i] ≥ value then
        AtomicDecrement(Counter.cell[i], value)
    else
        r ← value − Counter.cell[i]
        AtomicSet(Counter.cell[i], 0)
        Decrement(Counter, r, seed, attempt + 1)
    end
```

smart contract can use this data structure for storing the balance record of highly active accounts.

Chapter 3

Persistence Layer

The Argennon Smart Contract Execution Environment has two persistent memory areas: *code area*, and *heap*. Code area stores the Argennon Standard Representation of applications[1], and heap stores heap chunks. Both of these data elements, ASRs and heap chunks, can be considered as continuous arrays of bytes. Throughout this chapter, we shall call these data elements *objects*.

3.1 Storage Pages

In the AscEE persistence layer, similar objects are clustered together and constitute a bigger data element which we call a *page*.[2] A page is an ordered list of an arbitrary number of objects, which their order reflects the order they were added to the page:

$$P = (O_1, O_2, \ldots, O_n), \quad i < j \;\Leftrightarrow\; O_i \text{ was added before } O_j \;\cdot$$

A page of the AscEE storage should contain objects that have very similar access patterns. Ideally, when a page is needed for validating a block, almost all of its objects should be needed for either reading or writing. We prefer that the objects are needed for the same access type. In other words, the objects of a page are chosen in a way that for validating a block, we usually need to either read all of them or modify[3] all of them.

3.2 Publicly Verifiable Database Servers

Pages of the AscEE storage are persisted using *dynamic cryptographic accumulators*. Argennon has three dynamic cryptographic accumulators: *staking* database, which stores all the data that is associated with the Argennon consensus protocol, *code* database, which stores the AscEE code area, and *heap* database, which stores the AscEE heap.

[1] also it stores applications' constants.

[2] we avoid calling them clusters, because usually a cluster refers to a *set*. AscEE object clusters are not sets. They are ordered lists, like a page containing an ordered list of words or sentences.

[3] and probably read.

The commitment of these three accumulators are included in every block of the Argennon blockchain. These accumulators, in the Argennon network, are hosted on special database nodes called Publicly Verifiable Database (PV-DB) servers. We consider the following properties for a PV-DB:

- The PV-DB contains a mapping from a set of keys to a set of values.
- Every state of the database has a commitment C.
- The PV-DB has a method $(D, \pi) = \text{get}(x)$, where x is a key and D is the associated data with x, and π is a proof.
- A user having C and π can verify that D is really associated with x, and D is not altered. Consequently, a user who can obtain C from a trusted source does not need to trust the PV-DB.
- Having π and C a user can compute the commitment C' for the database in which D' is associated with x instead of D.

Pages of the AscEE storage are stored in the PV-DBs, with an index: `pageIndex` as their key. The `pageIndex` is required to be smaller than a certain value, determined by the protocol, to facilitate the usage of PV-DBs that are based on vector commitments. For this reason, the AscEE clustering algorithm always tries to reuse indices and keep the number of used indices as low as possible.

The commitments of the AscEE cryptographic accumulators are affected by the way data objects are clustered. Therefore, the Argennon clustering algorithm has to be a part of the consensus protocol.

Every block of the Argennon blockchain contains a set of *clustering directives*. These directives can only modify pages that were used for validating the block, and can include directives for moving an object from one page to another or directives specifying which pages will contain the newly created objects. These directives are always executed by nodes at the end of block validation.

A block proposer could obtain clustering directives from any third party source[4]. This will not affect Argennon security, since the integrity of a database can not be altered by clustering directives. Those directives can only affect the performance of the Argennon network, and directives of a single block can not affect the performance considerably.

3.2.1 Vector Commitments

Informally, vector commitments allow committing to an ordered sequence of q values (i.e. a vector), rather than to single messages. This is done in a way such that it is later possible to open the commitment with respect to specific positions (e.g., to prove that m_i is the i-th committed message). More precisely, vector commitments are required to

[4]we can say the AscEE clustering algorithm is essentially off-chain.

satisfy what is called position binding. Position binding states that an adversary should not be able to open a commitment to two different values at the same position. While this property, by itself, would be trivial to realize using standard commitment schemes, what makes vector commitments interesting is that they are concise, i.e., the size of the commitment string as well as the size of each opening **is independent of the vector length**.

not yet written...

3.3 Object Clustering Algorithm

not yet written...

Chapter 4

Networking Layer

4.1 Normal Mode

Unlike conventional blockchains, Argennon does not use a P2P network architecture. Instead, it uses a client-server topology, based on a permission-less list of ZK-EDB servers. ZK-EDB servers are a crucial part of the Argennon ecosystem, and they form the backbone of the Argennon networking layer.

not yet written...

4.2 Censorship Resilient Mode

not yet written...

Chapter 5

The Argennon Blockchain

Currently a new version is being rewritten to include CI proofs.

5.1 Blocks

The Argennon blockchain is a sequence of blocks. Every block represents an ordered list of transactions, intended to be executed by the Argennon Virtual Machine (AVM). The first block of the blockchain, the *genesis* block, is a spacial block that fully describes the initial state of the AVM. Every block of the Argennon blockchain thus corresponds to a unique AVM state which can be calculated deterministically from the genesis block.

A block of the Argennon blockchain contains the following information:

Block
commitment to the staking database
commitment to the method database
commitment to the heap database
commitment to the set of transactions
a consecutive list of block certificates issued by validators' committee (if any)
clustering directives
random seed
previous block hash

5.1.1 Block Validation

Having the previous AVM state, the transaction list and the clustering directives of a block, a node can calculate commitments to the staking, method and heap databases of the current block by emulating the AVM execution. If the node can obtain the previous block commitments from a trusted source, it does not need to have a trusted local copy of the AVM state, and it can reliably retrieve the required storage pages from a ZK-EDB server. We call this type of block verification *conditional* block validation, since the validity of the current block is conditioned on the validity of the previous block.

Interestingly, conditional block validation of multiple blocks can be done in parallel. If a node has enough bandwidth and computational resources, it can conditionally verify any number of blocks from a previously created blockchain simultaneously and in parallel. As we will see in Section 5.2.2, this property plays an important role in the Argennon consensus protocol.

To some extent, conditional validation of a single block could be parallelized as well. Many transactions in a block are actually independent and the order of their execution does not matter. These transactions can be safely validated in parallel. Section 2.11 further develops this concept.

5.1.2 Block Certificate

An Argennon block certificate is an aggregate signature of some predefined subset of accounts. This predefined subset is called the certificate committee and their signature ensures that the certified block is conditionally valid given the validity of some previous block.

Argennon uses BLS aggregate signatures to represent block certificates. To better understand block certificates and the Argennon consensus protocol, we need to briefly review the BLS signature scheme and its aggregation mechanism.

The BLS signature scheme operates in a prime order group and supports simple threshold signature generation, threshold key generation, and signature aggregation. To review, the scheme uses the following ingredients:

- An efficiently computable *non-degenerate* pairing $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$ in groups $\mathbb{G}_0$, $\mathbb{G}_1$ and $\mathbb{G}_T$ of prime order q. We let g_0 and g_1 be generators of $\mathbb{G}_0$ and $\mathbb{G}_1$ respectively.

- A hash function $H_0 : \mathcal{M} \to \mathbb{G}_0$, where $\mathcal{M}$ is the message space. The hash function will be treated as a random oracle.

The BLS signature scheme is defined as follows:

- **KeyGen**(): choose a random α from $\mathbb{Z}_q$ and set $h \leftarrow g_1^{\alpha} \in \mathbb{G}_1$. output $pk := (h)$ and $sk := (\alpha)$.

- **Sign**(sk, m): output $\sigma \leftarrow H_0(m)^{\alpha} \in \mathbb{G}_0$. The signature σ is a *single* group element.

- **Verify**(pk, m, σ): if $e(g_1, \sigma) = e\big(pk,\ H_0(m)\big)$ then output "accept", otherwise output "reject".

Given triples $(pk_i,\ m_i,\ \sigma_i)$ for $i = 1, \ldots, n$, anyone can aggregate the signatures $\sigma_1, \ldots, \sigma_n \in \mathbb{G}_0$ into a short convincing aggregate signature σ by computing

$$\sigma \leftarrow \sigma_1 \cdots \sigma_n \in \mathbb{G}_0 \ . \tag{5.1}$$

Verifying an aggregate signature $\sigma \in \mathbb{G}_0$ is done by checking that

$$e(g_1, \sigma) = e\big(pk_1,\ H_0(m_1)\big) \cdots e\big(pk_n,\ H_0(m_n)\big) \ . \tag{5.2}$$

When all the messages are the same ($m = m_1 = \ldots = m_n$), the verification relation (5.2) reduces to a simpler test that requires only two pairings:

$$e(g_1, \sigma) = e\Big(pk_1 \cdots pk_n,\ H_0(m)\Big) \ . \tag{5.3}$$

We call $apk = pk_1 \cdots pk_n$ the aggregate public key.

To defend against *rogue public key* attacks, Argennon uses Prove Knowledge of the Secret Key (KOSK) scheme. As we explained in Section 5.4, when an account is created its public keys need to be registered in the ARG smart contract. Therefore, the KOSK scheme can be easily implemented in Argennon.

Because it is not usually possible to collect the signatures of all members of a certificate committee, an Argennon block certificate essentially is an Accountable-Subgroup Multi-signature (ASM). Argennon uses a simple ASM scheme based on BLS aggregate signatures.

Argennon block certificates constitute an ordered sequence based on the order of blocks they certify. If we show the i-th certificate[1] of committee C with $cert_i$, and the set of signers with S_i, then the block certificate $cert_i$ can be considered as a tuple:

$$cert_i = (\sigma_i,\ C - S_i) \ , \tag{5.4}$$

where σ_i is the aggregate signature issued by S_i.

The aggregate public key of the certificate can be calculated from:

$$apk_i = apk_C\, apk_{C-S_i}^{-1} \ , \tag{5.5}$$

where apk_A shows the aggregate public key of all accounts in A.

Alternately, we can use apk_{i-1} to calculate the aggregate public key:

$$apk_i = apk_{i-1}\, apk_{S_i - S_{i-1}}\, apk_{S_{i-1} - S_i}^{-1} \ . \tag{5.6}$$

When an Argennon account is created, both its pk and pk^{-1} is registered in the ARG smart contract, so the inverse of any aggregate public key can be easily computed.[2]

[1] note that the i-th certificate is not necessarily the certificate of the i-th block.
[2] since the group operator of a cyclic group is commutative, we have $(ab)^{-1} = a^{-1}b^{-1}$.

5.2 Consensus

The credibility of a block of the Argennon blockchain is determined by the certificates it has received from different sets of users, known as committees. There are two primary type of certificate committees in Argennon: the committee of *delegates* and the committee of *validators*. Argennon has *one* committee of delegates and m committees of validators.

The committee of delegates issues a certificate for every block of the Argennon blockchain, and each committee of validators issues a certificate every m blocks. A validators' committee will certify a block only if it has already been certified by the committee of delegates. Every committee of validators has an index between 0 and $m-1$, and it issues a certificate for block number n, if n modulo m equals the committee index.

Every block of the Argennon blockchain needs a certificate from both the committee of delegates and the committee of validators. A block is considered final after its **next** block receives **both** of its certificates. In Argennon as long as more than half of the total stake of validators is controlled by honest users, the probability of discarding a final block is near zero even if all the delegates are malicious.

In addition to primary committees, Argennon has several community driven committees. Certificates of these committees are not required for block finality, but they could be used by members of the validators' committee to better decide about the validity of a block.

When an anomaly is detected in the consensus mechanism, the *recovery* protocol is initiated by validators. The recovery protocol is designed to be resilient to many types of attacks in order to be able to restore the normal functionality of the system.

5.2.1 The Committee of Delegates

The committee of delegates is a small committee of trusted delegates, elected by Argennon users through the Argennon Decentralized Autonomous Governance system (ADAGs[3]). At the start of the Argennon mainnet, this committee will have five members, and later its size could be changed by the ADAGs in a procedure described in Section 6.1.

The committee of delegates is responsible for creating new blocks of the Argennon blockchain, and it issues a certificate for every block of the Argennon blockchain. A certificate needs to be signed by **all** of the committee members in order to be considered valid.

Besides the main delegates' committee, a reserve committee of delegates consisting of three members is elected by users either through the ADAGs or by *emergency agreement* during the recovery protocol. In case the main committee fails to generate new blocks or behaves maliciously, the task of block generation will be assigned to the reserve committee until a new main delegates' committee is elected through the ADAGs.

[3]pronounced /eɪ-dagz/.

Usually, the delegates are large organizations, and they have enough computational resources to generate blocks very fast. However, a block is not completely final if it does not have the certificate of its validators. A certified block by the delegates will not be accepted by the network, if the last block certified by the validators is behind it more than a certain number of blocks.

The committee of delegates may use any type of agreement protocol to reach consensus on the next block. Usually a very simple and fast protocol can do the job: one of the members is randomly chosen as the proposer, and other members vote "yes" or "no" on the proposed block.

If one of the delegates lose its network connectivity, no new blocks can be generated. For this reason, the delegates should invest on different types of communication infrastructure, to make sure they never lose connectivity to each other and to the Argennon network.

5.2.2 Validators

The Argennon protocol calculates a stake value for every account, which is an estimate of a user's stake in the system, and is measured in ARGs. Any account whose stake value is higher than `minValidatorsStake` threshold is considered a *validator*. The `minValidatorsStake` threshold is determined by the ADAGs, but it can never be higher than 500 ARGs.

Every `committeeLifeTime` number of blocks, randomly m committees are selected from validators, in a way that the total stakes of committees are almost equal, and every account is a member of **at least** one committee.

Every validator has a status which can be either `online` or `offline`. This status is stored in the ARG smart contract and is a part of the staking database. A validator can change his status through a method invocation from the ARG smart contract. When an account sets its status to `offline`, it receives a small reward, and it can not change it back to `online` for `statusCoolDown` number of blocks.

When a validator changes his status, the change has no effect until the block containing the status change transaction gets certified by his committee.

A block certificate issued by some members of a validators' committee is considered valid, if according to the staking database of the previous block **certified by the same committee**, we have:[4]

- The total stake of `online` members of the committee is higher than `minOnlineStake` fraction of the total stake of the committee. This threshold can be changed by the ADAGs, but it can never be lower than 2/3.
- All signers of the certificate have `online` status.

[4]If we calculate the stake values based on the previous block a malicious committee can select the validators of the next block.

- The sum of stake values of the certificate signers is higher than 3/4 of the total stake of the committee members that have `online` status.

The delegates can generate blocks very fast. Therefore, the Argennon blockchain always has an unvalidated part, which contains blocks that have a certificate from the committee of delegates but have not yet received a certificate from the validators.

As we mentioned before, the block with height n needs a certificate from the committee of validators with index n modulo m. To decide about signing the certificate of a block which already has a certificate from the delegates, a validator checks the conditional validity[5] of the block, and if the block is valid he issues an "accept" signature. If the block is invalid, he initiates the recovery protocol. The validator will broadcast the certificate **only after** he sees the certificate of the validators of the previous block. Some validators may also require seeing a certificate from some community driven committee. An honest validator never signs a certificate for two different blocks with the same height.

So in Argennon, the block validation by committees is performed in parallel, and validators do not wait for seeing the certificate of the previous block validators to start transaction validation. On the other hand, the block certificates are published and broadcast sequentially. A validator does not publish its certificate if the certificate of the previous block has not been published yet. This ensures that an invalid fork made by malicious delegates will not receive any certificates from validators.

The value of m is determined by the ADAGs. but it can never be higher than 25. This way, it is guaranteed that on average, any block of the Argennon blockchain is validated by at least 0.02 of the total ARG supply.

block certificates issued by committees of validators are included in the blocks of the Argennon blockchain. A block can contain multiple certificates, provided that those certificates belong to consecutive blocks.

5.2.3 Status Blocks

If according to the staking database of block n, the total online stake of the committee with index n modulo m is lower than `minOnlineStake` threshold, the block $n + m$ can never be certified by validators.

To prevent the blockchain from halting in such situations, the protocol performs a predefined partial reshuffling of committee members. In this reshuffling which is based on the block random seed, some online members from other committees will be moved to the committee without enough online stake to make it active again.

If the reshuffling can not solve the problem due to low total online stake, the protocol requires the next block of the blockchain to be a special *status block*. A status block is a special block which can only contain status change transactions. The status block needs to be certified by the delegates and by 2/3 of the total stake of validators. The `online/offline` status of validators will not be considered in the validity of the status

[5] See Section 5.1.1

block certificate.[6] After applying the transactions of the status block, the total online stake of all committees of validators must go higher than `minOnlineStake` threshold.

5.2.4 Signature Aggregation

In Argennon, signature aggregation is mostly performed by ZK-EDB servers. To distribute the aggregation workload between different servers, Every committee of validators is divided into pre-determined groups, and each ZK-EDB server is responsible for signature aggregation of one group. To make sure that there is enough redundancy, the total number of groups should be less than the number of ZK-EDB servers and each group should be assigned to multiple ZK-EDB servers.

Any member of a group knows all the servers that are responsible for signature aggregation of his group. When a member signs a block certificate, he sends his signature to **all** the severs that aggregate the signatures of his group. These servers aggregate the signatures they receive and then send the aggregated signature to the delegates. Furthermore, the delegates aggregate these signatures to produce the final block certificate and then include it in the next block.

The role of the delegates in the signature aggregation is limited. The important part of the work is done by ZK-EDB servers. As long as there are enough honest ZK-EDB servers, the network will be able to perform signature aggregation even if the delegates are malicious.

5.2.5 The Recovery Protocol

The recovery protocol is a resilient protocol designed for recovering the Argennon blockchain from critical situations. In the terminology of the CAP theorem, the recovery protocol is designed to choose consistency over availability, and is not a protocol supposed to be executed occasionally. Ideally this protocol should never be used during the lifetime of the Argennon blockchain.

The recovery protocol can recover the functionality of the Argennon blockchain as long as more than 2/3 of the total stake of the system is controlled by honest users and any network partition resolves after a finite amount of time. The recovery protocol uses two main emergency procedures to recover the functionality of the Argennon blockchain: *emergency forking* and *emergency agreement* protocol.

Emergency Forking

The reserve committee of delegates is able to fork the Argennon blockchain, if it receives a valid fork request from the validators. This fork needs to be confirmed by validators and can not discard any blocks that has been already certified by validators. A valid fork request is an unexpired request signed by more than half of the total `online` stake of the validators.

[6] Theoretically at the status block, the total online stake of the system could be very low. Therefore, the status block should not be certified only by online stake.

For forking at block b, the reserve committee of delegates makes a special *fork block* which only contains a valid fork request, and its parent is the block b. The height of the fork block is $b+1$ so it needs a valid certificate from the committee of validator with the index $b+1$ modulo m.

For signing a fork block, a validator ensures that the block is signed by the reserve committee and contains a valid fork request. The parent of the fork block does not necessarily need a validators' certificate. If the parent does not have a certificate, the validator checks the certificate of the block before the parent and the conditional validity of the parent instead. This enables the reserve committee to recover the liveliness of the blockchain in a situation where a malicious committee has generated multiple blocks at the same height.

A validator always chooses a valid fork block over a block of the main chain. However, as we mentioned before, a validator never signs a certificate for two different blocks with the same height. Consequently, if a validator has already signed the block b of the main chain, he will not sign the fork block and vice versa.

When the fork block is broadcast, it is possible that the validators of the committee with index $b+1$ modulo m get divided between the fork block and the block $b+1$ of the main chain, in a way that no block gets enough validators. This will cause the blockchain to halt. To prevent this, the protocol allows the reserve delegates to revoke a fork block. After a fork block is revoked, the validators who voted for it are allowed to vote for another block with the same height.

To revoke a fork block with height $b+1$, the delegates need to seal the fork by adding a special *seal block* after the fork block.[7] The seal block has the height $b+2$ and needs to be certified by the validators' committee with the index $b+2$ modulo m. The fork block is considered revoked only after the seal block is certified by the validators.

The seal block is not a normal block and validators who signed a certificate for a seal block are allowed to sign a certificate for a block with the same height and vice versa. However, it should be noted that generating a block with the same parent as the seal block is considered a malicious behaviour of the reserve committee and validators will not sign such a block.

As long as more than half of the total online stake of every committee of validators is controlled by honest users, a malicious committee of delegates can not use the emergency forking procedure to discard blocks that have a certificate from validators.

Moreover, an honest committee of delegates will always try to perform the emergency forking in such a way that valid blocks do not get discarded, including blocks that are not certified by the validators yet.

Emergency Agreement

The emergency agreement protocol is a resilient protocol for deciding between a set of proposals when no committee of delegates can be trusted. For initiating the protocol, a validator signs a message containing the subject of the agreement and a start time.

[7] The parent of the seal block must be the fork block.

A validator enters the agreement protocol if he receives a request that is signed by more than half of the total stake of the validators and its start time has not passed. The validator calculates the stake values based on the staking database of the last final block in his blockchain without considering the `online`/`offline` status of validators.

The agreement protocol consists of two phases: the *voting phase*, which selects a single proposal and the *confirmation phase* which confirms the selected proposal. The voting phase is done in rounds. Each round lasts for approximately λ units of time, and after k rounds, the current agreement session ends and a new session starts. All votes and messages are tagged in a way that the messages of one session can not be used in another session.

Users cast three type of votes: *i-votes*, which are votes that are valid only in round i, *final-votes*, which are votes that are valid in any round, and *c-votes*, which are votes used only in the confirmation phase.

A user executes the following procedure in round r of the voting phase:

Voting Phase:

- if the user has not yet final-voted any value, he r-votes a single desired proposal.
- if he sees more than 2/3 r-votes for a proposal p, he final-votes p.
- if he sees more than 2/3 final-votes for a proposal, he goes to the confirmation phase for p.
- if $clock > r \cdot \lambda$ and $r < k$ user goes to the round $r + 1$ and if $r = k$ user starts a new agreement session.

Confirmation Phase for Proposal p:

- user c-votes p.
- if he sees more than 2/3 c-votes for p he selects p and ends the agreement protocol.
- if $clock > k \cdot \lambda$ user starts a new agreement session.

We assume that users have clocks with the same speed, and $\lambda \gg \epsilon$, where ϵ is the maximum clock difference between users. We also assume that more than 2/3 of the total stake of the system is controlled by honest users, and network partitions are resolved after a finite amount of time. With these assumptions it can be shown that the emergency recovery protocol has the following important properties:

- no two users will end the agreement protocol with two different proposals as the result of the agreement.
- if honest users can agree upon some proposal value, the agreement protocol will converge to that value after a finite number of sessions.

When the emergency agreement protocol is used for electing a new reserve committee to fork the blockchain, the confirmation phase could be skipped. In that case, the confirmation of the fork block by the appropriate committee of validators acts like the confirmation phase.

Initiating the Recovery Protocol

When a validator does not receive any blocks for `blockTimeOut` amount of time, or when he sees an evidence which proves the delegates are malicious, he initiates the recovery protocol.

To do so, first the validator activates the censorship resilient mode of his networking module, then he checks the validity of the blocks that do not have a validators' certificate and determines the last valid block of his blockchain.

In the next step, he will sign and broadcast an **emergency fork request** message, alongside some useful metadata such as the last valid block of his blockchain and the evidence of delegates' misbehaviour.[8]

If the reserve committee of delegates is already active, or if a validator sees a valid fork request signed by more than half of the total online stake of the validators, but does not receive the fork block after a certain amount of time, he will sign and broadcast a request for **emergency agreement** on a new reserve committee. The agreement on new delegates usually needs user interaction and is not a fully automatic process.

The evidence which proves a committee of delegates is malicious is an invalid block that is signed by at least one delegate:

- a block that is not conditionally valid
- two different blocks with the same parent
- a block that has an invalid format

5.2.6 Estimating Stake Values

In a proof of stake system the influence of a user in the consensus protocol should be proportional to the amount of stake the user has in the system. Conventionally in these systems, a user's stake is considered to be equal with the amount of native system tokens, he has "staked" in the system. A user stakes his tokens by locking them in his account or a separate staking account for some period of time. During this time, he will not be able to transfer his tokens.

Unfortunately, there is a subtle problem with this approach. It is not clear in a real world economic system how much of the main currency of the system can be locked and kept out of the circulation indefinitely. It seems that this amount for currencies like US dollar, is quite low comparing to the total market cap of the currency. This means that for a real world currency this type of staking mechanism will result in putting the fate of the system in the hands of the owners of a small fraction of the total supply.

[8] this metadata is not a part of the fork request.

To mitigate this problem, Argennon uses a hybrid approach for estimating the stake of a user. Every `stakingDuration` blocks, which is called a *staking period*, Argennon calculates a *trust value* for each user.

The user's stake at time step t, is estimated based on the user's trust value and his ARG balance:

$$S_{u,t} = \min(B_{u,t}, Trust_{u,k}) \ , \tag{5.7}$$

where:

- $S_{u,t}$ is the stake of user u at time step t.
- $B_{u,t}$ is the ARG balance of user u at time step t.
- $Trust_{u,k}$ is an estimated trust value for user u at staking period k.

Argennon users can lock their ARG tokens in their account for any period of time. During this time a user will not be able to transfer his tokens and there is no way for cancelling a lock. The trust value of a user is calculated based on the amount of his locked tokens and the Exponential Moving Average (EMA) of his ARG balance:

$$Trust_{u,k} = L_{u,k} + M_{u,t_k} \ , \tag{5.8}$$

where

- $L_{u,k}$ is the amount of locked tokens of user u, whose release time is **after the end** of the staking period $k+1$.
- M_{u,t_k} is the Exponential Moving Average (EMA) of the ARG balance of user u at time step t_k. t_k is the start time of the staking period k.

In Argennon a user who held ARGs and participated in the consensus for a long time is more trusted than a user with a higher balance whose balance has increased recently. An attacker who has obtained a large amount of ARGs, also needs to hold them for a long period of time before being able to attack the system.

For calculating the EMA of a user's balance at time step t, we can use the following recursive formula:

$$M_{u,t} = (1-\alpha)M_{u,t-1} + \alpha B_{u,t} = M_{u,t-1} + \alpha(B_{u,t} - M_{u,t-1}) \ ,$$

where the coefficient α is a constant smoothing factor between 0 and 1, which represents the degree of weighting decrease. A higher α discounts older observations faster.

Usually an account balance will not change in every time step, and we can use older values of EMA for calculating $M_{u,t}$: (In the following equations the u subscript is dropped for simplicity)

$$M_t = (1-\alpha)^{t-k}M_k + [1-(1-\alpha)^{t-k}]B \ ,$$

where:

$$B = B_{k+1} = B_{k+2} = \cdots = B_t \ .$$

We know that when $|nx| \ll 1$ we can use the binomial approximation $(1+x)^n \approx 1+nx$. So, we can further simplify this formula:

$$M_t = M_k + (t-k)\alpha(B - M_k) \ .$$

For choosing the value of α we can consider the number of time steps that the trust value of a user needs for reaching a specified fraction of his account balance. We know that for large n and $|x| < 1$ we have $(1+x)^n \approx e^{nx}$, so by letting $M_{u,k} = 0$ and $n = t-k$ we can write:

$$\alpha = -\frac{\ln\left(1 - \frac{M_{n+k}}{B}\right)}{n} \ . \tag{5.9}$$

The value of α for a desired configuration can be calculated by this equation. For instance, we could calculate the α for a relatively good configuration in which $M_{n+k} = 0.8B$ and n equals to the number of time steps of 10 years.

5.2.7 Analysis

not yet written...

5.3 Applications

An Argennon application or smart contract is an HTTP server which is represented by an Argennon Standard Representation (ASR) and whose state is stored in the Argennon blockchain. Each Argennon application is identified by a unique application identifier.

An application identifier, `applicationID`, is a unique prefix code generated by the *applications* prefix tree. (See Section 2.3.) An application identifier can be considered as the address of an application and has the following standard symbolic representation:

```
<application-id> ::= <decimal-prefix-code>
<decimal-prefix-code> ::= <dec-num>"."<decimal-prefix-code> | <dec-num>
```

where `<dec-num>` is a normal decimal number between 0 and 255. For example `21.255.37`, `0`, `11.6` and `2.0.0.0.0`, are valid application addresses.

Argennon has two special smart contracts: the *root smart contract*, also called the *root application*, and the *ARG smart contract*, which is also called the *Argennon smart contract* or the *ARG application*.

Argennon application use HTTP as the application protocol and they are advised to have a RESTful API design.

5.3.1 The Root Application

The root application or the root smart contract, with `applicationID = 0`, is a privileged smart contract responsible for installation/uninstallation of other smart contracts. The Argennon's root smart contract performs three main operations:

- Installation of new Argennon applications and determining the update policy of a smart contract: if the contract is updatable or not, which accounts or smart contracts can update or uninstall the contract, and so on.
- Removing an Argennon application (if allowed).
- Updating an Argennon application (if allowed).

The root smart contract is a mutable smart contract and can be updated by the Argennon governance system. (See Section 6.1)

5.3.2 The ARG Application

The ARG application or the ARG smart contract, with `applicationID = 1`, controls the ARG token, the main currency of the Argennon blockchain. This smart contract also manages a database of public keys and handles signature verification.

The ARG smart contract is a mutable smart contract and can be updated by the Argennon governance system.

5.4 Accounts

Argennon accounts are entities defined inside the ARG application. Every Argennon account is uniquely identified by a prefix code generated using *accounts* prefix tree. (See Section 2.3) An account identifier can be considered as the address of an account and has the following standard symbolic representation:

```
<account-id> ::= "0x"<hex-num>
```

where `<hex-num>` is a hexadecimal number, using lower case letters `[a-f]` for showing digits greater than 9.

For example `0x24ffda`, `0x0` and `0x03a0000`, are valid standard symbolic representations of account addresses.

A new account can be created by sending a proper HTTP request to the ARG smart contract. For creating a new account two public keys need to be provided by the caller and registered in the Argennon smart contract. One public key will be used for issuing digital signatures, and the other one will be used for voting. The provided public keys need to meet certain cryptographic requirements,[9] and can not be already registered in the system.

If the owner of the new account is an application, the `applicationID` of the owner will be registered in the ARG smart contract and no public keys are needed. An application can own an arbitrary number of accounts.

[9] Argennon uses Prove Knowledge of the Secret Key (KOSK) scheme.

Explicit key registration enables Argennon to decouple cryptography from the blockchain design. In this way, if the cryptographic algorithms used become insecure for some reason, for example because of the introduction of quantum computers, they could be easily upgraded.

5.5 Transactions

An Argennon transaction consist of an HTTP request made by a user to an Argennon application, a resource declaration object and a list of signed messages. Transactions can only be issued by users and applications can not create transactions. An Argennon transaction is also called an *external request*.

5.5.1 Resource Declaration

Every Argennon transaction is required to provide the following information as an upper bound for the resources it needs:

- Maximum AscEE clocks
- The list of applications the request will call
- The list of access blocks the request needs
- `maxSize` for chunks it wants to expand
- `minSize` for chunks it wants to shrink
- A list of applications it will update (if any)

If a transaction tries to violate any of these predefined limitations, it will be considered failed, and the network can receive the proposed fee of that transaction.

5.6 Incentive mechanism

5.6.1 Fees

The Argennon protocol does not explicitly define any fees for normal transactions. Only for high priority transactions a fixed fee is determined by the governance system (See Section ...). Because the protection of the Argennon network against spams and DOS attacks is mostly done by the delegates, they are also responsible for determining and collecting transaction fees. A good fee collection policy could considerably increase the chance of delegates for being reelected in the next terms, therefore they are incentivized to use creative and effective methods[10].

[10]For example, they may allow a limited number of free transactions per month for every account.

An Argennon transaction in YAML format


```yaml
---
request: |
    PATCH /balances/0x95ab HTTP/1.1
    Content-Type: application/json; charset=utf-8
    Content-Length: 46

    {"to":0xaabc,"amount":1399,"sig":0}

messages:
  - issuer: 0x95ab000000000000
    msg: {"to":0xaabc000000000000,"amount":1399,"forApp":0x100000000000000,"nonce":11}
    sig: LNUC49Lhyz702uszzNcfaU3BhPIbdaSgzqDUKzbJzLPTlFS2J9GzHl-cDKb

caps:
    maxClocks: 150 # maximum number of AVM execution clocks
    apps: [1,124.16]
    read: [(2654,3),(15642,0),(15642,1),(15642,3)]
    write: [(15642,0),(20154,0),(20154,1)]
```


In Argennon fee payment can be done off-chain or on-chain. Off-chain fee payment is more efficient and flexible but requires some level of trust in the delegates. For trust-less fee payment, the Argennon protocol provides the concept of request attachments (See Section 2.5). When a user does not want to use off-chain fee payment methods, he can simply define his transaction as the attachment of the fee payment transaction. That way, the fee payment transaction will be performed only if the attached transaction is also included in the same block.

While transaction fee is not enforced by the Argennon protocol, there are other types of fee that are mandatory: the *database fee* and the *block fee*. Both of these fees are required to be paid for every block of the Argennon blockchain and are paid by the delegates. The block fee is a constant fee that is paid for each new block of the blockchain and its amount is determined by the ADAGs. The database fee depends on the data access and storage overhead that a new block is imposing on the Argennon storage cloud. The amount of this fee is determined by the ADAGs, and is collected in a special account: the `dbFeeSink`.

5.6.2 Certificate Rewards

The validators who sign the certificate of a block will receive the block fee paid for that block. Every validator will be rewarded proportional to his stake (i.e voting power). As we mentioned before the block fee is a constant fee which the delegates pay for each block.

Rewards will not be distributed instantly, instead they will be distributed at the end of the staking period. This will facilitate efficient implementations which avoid frequent updates in the Argennon storage.

As long as ARG is allowed to be minted and its cap is not reached, the delegates will receive a reward at the **end** of their election term. This reward will consist of newly minted ARGs, and its amount will be determined by the ADAGs. In addition, for each block certificate that is added to the Argennon blockchain some amount of ARGs will be minted and added to the `dbFeeSink` account.

5.6.3 Penalties

If an account behaves maliciously, and that behaviour could not have happened due to a mistake, by providing a proof in a block, the account will be disabled forever in the ARG smart contract. Disabling an account in the ARG smart contract will prevent that account from signing any valid signatures in the future.

Only behaviours will be punished that can not happen due to a mistake or an attack. These behaviours include:

- Signing a certificate for a block that is not conditionally valid.
- Signing a certificate for two different blocks at the same height if none of them is a fork block or a seal block.[11]

5.6.4 Incentives for PV-DB Servers

The incentive mechanism for PV-DB servers should have the following properties:

- It incentivizes storing all storage pages and not only those pages that are used more frequently.
- It incentivizes PV-DB servers to actively provide the required storage pages for validators.
- Making more accounts will not provide any advantage for a PV-DB server.

For our incentive mechanism, we require that every time a validator receives a storage page from a PV-DB, after validating the data, he gives a receipt to the PV-DB server. In this receipt the validator signs the following information:

- `ownerAddr`: the account address of the PV-DB server.
- `receivedPageID`: the ID of the received page.
- `round`: the current block number.

[11]Signing a fork block and a normal block at the same height usually is a malicious behaviour. However, it will not be penalized because there are circumstances that an honest user could mistakenly do that.

In a round, an honest validator never gives a receipt for an identical page to two different PV-DB servers.

To incentivize PV-DB servers, a lottery will be held every round,[12] and a predefined amount of ARGs from `dbFeeSink` account will be distributed between the winners as a prize. This prize will be divided equally between all *winning tickets* of the lottery.

One PV-DB server could own multiple winning tickets in a round.

To run this lottery, every round, based on the current block seed, a collection of *valid* receipts will be selected randomly as the *winning receipts* of the round. A receipt is *valid* in round r if:

- The signer was a member of the validators' committee of the block $r-1$ and signed the block certificate.
- The page in the receipt was needed for validating the **previous** block.
- The receipt round number is $r-1$.
- The signer did not sign a receipt for the same storage page for two different PV-DB servers in the previous round.

For selecting the winning receipts we could use a random generator:

```
IF random(seed|validatorPK|receivedPageID) < winProbability THEN
    the receipt issued by validatorPK for receivedPageID is a winner
```

- `random()` produces uniform random numbers between 0 and 1, using its input argument as a seed.
- `validatorPK` is the public key of the signer of the receipt.
- `receivedPageID` is the ID of the storage page that the receipt was issued for.
- `winProbability` is the probability of winning in every round.
- `seed` is the current block seed.
- `|` is the concatenation operator.

Also, based on the current block seed, a random storage page is selected as the challenge of the round. A PV-DB server that owns a winning receipt needs to broadcast a *winning ticket* to claim his prize. The winning ticket consists of a winning receipt and a *solution* to the round challenge. Solving a round challenge requires the content of the

[12] A round is the time interval between two consecutive blocks.

storage page which was selected as the round challenge. This will encourage PV-DB servers to store all storage pages.

A possible choice for the challenge solution could be the cryptographic hash of the content of the challenge page combined with the server account address:

`hash(challenge.content|ownerAddr)`

The winning tickets of the lottery of round r need to be included in the block of the round r, otherwise they will be considered expired. However, finalizing and prize distribution for the winning tickets should be done in a later round. This way, **the content of the challenge page could be kept secret during the lottery round.** Every winning ticket will get an equal share of the lottery prize.

Chapter 6

Governance

6.1 ADAGs

The Argennon Decentralized Autonomous Governance system (ADAGs)

not yet written...

Chapter 7

The Argon Language

7.1 Introduction

The Argon programming language is a class-based, object-oriented language designed for writing Argennon smart contracts. The Argon programming language is inspired by Solidity and is similar to Java, with a number of aspects of them omitted and a few ideas from other languages included. Argon is designed to be fully compatible with the Argennon Virtual Machine and be able to use all advanced features of the Argennon blockchain.

Argon applications (i.e. smart contracts) are organized as sets of packages. Each package has its own set of names for types, which helps to prevent name conflicts. Every package can contain an arbitrary number of classes. Every Argon application is required to have exactly one `main` method and one `initialize` method. The `main` method is the only method of an Argon application which would be called by other smart contracts.

The `main` method is required to have a single parameter named `request`. The type of this parameter should be `RestRequest` or `HttpRequest`. The return value of the `main` function needs to be a `RestResponse` or `HttpResponse`.

7.2 Features Overview

7.2.1 Access Level Modifiers

Access level modifiers determine whether other classes can use a particular field or invoke a particular method.

	Class	Package	Subclass	Program
private	yes	no	no	no
protected	yes	no	yes	no
package	yes	yes	yes	no
public	yes	yes	yes	yes

```
public class MirrorToken {
    private static SimpleToken token;
    private static SimpleToken reflection;

    // ‘initialize‘ is a special static method that is called by the AVM after the code of a contract
    // is stored in the AVM code area.
    public static void initialize(double supply1, double supply2) {
        // ‘new‘ does not create a new smart contract. It just makes an ordinary object.
        token = new SimpleToken(supply1);
        reflection = new SimpleToken(supply2);
    }
    // ‘main‘ is the only method of the application (i.e. smart contract) that can be called
    // by other applications. Every application should have exactly one main method defined
    // in some class. Alternatively, the keyword ‘dispatcher‘ could be used instead of ‘main‘.
    public static RestResponse main(RestRequest request) {
        RestResponse response = new RestResponse();
        if (request.pathMatches("/balances/{user}")) {
            Account sender = request.getParameter<Account>("user");
            if (request.operationIsPUT()) {
                sender.authorize(request.toMessage(), request.getParameter<byte[]>("sig"));
                Account recipient = request.getParameter<Account>("to");
                double amount = request.getParameter<double>("amount");
                token.transfer(sender, recipient, amount);
                reflection.transfer(recipient, sender, Math.sqrt(amount));
                return response.setStatus(Http.Status.OK);
            } else if (request.operationIsGET()) {
                response.append<double>("balance", token.balanceOf(sender));
                response.append<double>("reflection", reflection.balanceOf(user));
                return response.setStatus(Http.Status.OK);
            } else {
                return response.setStatus(Http.Status.MethodNotAllowed);
            }
        }
    }
}

package class SimpleToken {
    private Map(Account −> double) balances;

    // The visibility of a member without an access modifier will be the package level.
    constructor(double initialSupply) {
        // initializes the object
    }

    void transfer(Account sender, Account recipient, double amount) {
        if (balances[sender] < amount) throw("Not enough balance.");
        // implements the required logic...
    }
    // implements other methods...
}
```

7.2.2 Shadowing

If a declaration of a type (such as a member variable or a parameter name) in a particular scope (such as an inner block or a method definition) has the same name as another declaration in the enclosing scope, it will result in a compiler error. In other words, the Argon programming language does not allow shadowing.