Argennon: A Scalable Cloud Based Smart Contract Platform Using Argument of Knowledge Systems

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 argument of knowledge systems (STARK/SNARK) and data integrity is guaranteed by cryptographic accumulators.

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 seducing them to violate the protocol.

At the time of writing of this document, the total mining reward for a Bitcoin block is around $150,000, and the Bitcoin network approximately produces 5 blocks per hour. If we assume, in decision theoretic terminology, that the mining reward solely defines the utility function[1] of a Bitcoin miner, one should be able to 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[2] of the validator set is a relatively small value. This problem is even 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

[1] In decision theory, the utility function measures the preference, worth or value of different alternatives for a decision maker.

[2] Here by stake, we mean a real number measuring the total interest of a user in the system, and we are not referring, in particular, to some locked amount of a user's money that is known as stake in some PoS protocols.

platform. This computational and storage overhead, in practice, could prevent most of ordinary users from any type of participation in the consensus protocol of a blockchain.

A fully decentralized blockchain based on the participation of every user looks appealing, though 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, an adversary can use it to take control of 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 are required. This is not something a normal user can afford. 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[3] system.

To overcome these difficulties, Argennon[4] 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[5] is elected for a one-year term, has five members, and is responsible for minting new blocks of the Argennon blockchain. Each minted block gets certified by all members of the delegates committee and after that it must get approved by its corresponding validator assembly.

Validator assemblies are very large sets of validators including at least three percent of the maximum possible stake of the Argennon network. Every validator assembly has an index between 0 and $m - 1$, and is responsible for validating block number n, if n modulo m equals the index of the assembly. A block is approved if it takes approval votes from at least $2/3$ of the total stake of its validator assembly.

In case the main committee fails to generate new blocks or behaves maliciously, a special *recovery protocol* is initiated by validators. This recovery protocol can recover the functionality of the Argennon blockchain as long as more than $2/3$ of the total stake of every validator assembly 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.

In Argennon, a block is considered final after its **next** block gets certified by **both** the delegates committee and its validator assembly. The Argennon protocol ensures that

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

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

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

as long as more than 2/3 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.

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 (i.e. the ledger state after executing all transactions of the block). The CI set defines how the final state of the block can be reached from the state of its previous block via a set of intermediate state transitions. Each individual CI statement defines an ordered list of external requests[6] and determines the state before and after executing those requests.

More formally, each CI statement has the form $\hat{\mathbb{S}} \coloneqq \tau(\mathbb{S}, \mathbb{R})$, and states that $\hat{\mathbb{S}}$ is the commitment of the next state after executing an ordered list of external requests with commitment $\mathbb{R}$ on a state which has commitment $\mathbb{S}$. Commitments to states are produced by a cryptographic accumulator. τ is a transition function and is known to both the validator and the prover.

Validators can[7] receive succinct cryptographic proofs (STARK/SNARK) of these CI statements from the Argennon cloud and validate blocks without storing the ledger state or performing costly computation.

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 assemblies 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.

On the blockchain, Argennon applications (i.e smart contracts) are stored in a high level text based language, called the Argennon Standard Application Representation (ASAR). The ASAR 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.

In addition to the ASAR, an Argennon application is stored on the blockchain in the Argennon Prover Machine (APM) code. This code is used by validators for CI proof verifications. The Argennon Prover Machine is a virtual machine tailored for efficient verification of AscEE computations by argument systems. The APM has a minimal RISC architecture with a very compact instruction set. This ensures that its transition function has an efficient circuit complexity. However, the Argennon protocol does not enforce the usage of the APM. Validators and PVC servers can use any argument system with any arithmetization, and if required, the ASAR of an application can be used for generating the appropriate arithmetization instead of the APM.

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

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

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[8] to use Data Dependency Analysis[9] (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.

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 front running protection. However, it also increases the possibility of transaction censorship. In Argennon, this problem is addressed by a special High Priority Request (HPR) protocol.

Using Succinct Argument of Knowledge systems makes the main functionalities of an Argennon validator light enough to be implemented as a smart contract. By deploying a validator contract on another platform, Argennon could use more established blockchains as an extra layer of security, specially during the bootstrapping phase, when ARG is not well distributed yet. In addition, this contract will facilitate trustless bridging of assets from that platform to the Argennon blockchain. To reduce execution fee, only roll-ups of the state transition can be validated by the contract.

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) storing all parts of the ledger state and providing *proofs of storage*.

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, 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.

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

[9] See Section 2.10

Chapter 2

The Argennon Smart Contract Execution Environment

2.1 Introduction

The Argennon Smart Contract Execution Environment (AscEE) is an abstract high level execution environment for executing Argennon smarts contracts a.k.a Argennon applications in an efficient and isolated 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 Application Representation (ASAR).

An Argennon Standard Application Representation is a programming language for describing Argennon applications, optimized for the architecture and properties of the Argennon platform. This text based representation is low level enough to enable easy compilation from any high level programming language and is high level enough to preserve the high level information of the application logic and facilitate platform specific compiler optimization at a host machine. In this regard, the ASAR can be considered as an Intermediate Representation (IR).

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. Besides heap chunks, every application has a limited 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 and 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*

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 *external* HTTP requests[1], produce their HTTP responses and update related heap chunks. The execution of requests can be considered sequential[2] and each request has a separate *execution session*. An execution session is a separate session of executing application's code in order to fulfill an external HTTP request. This external request is the *initiator* of the execution session.

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 effects on any heap chunks.

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

2.3 Memory

Every Argennon application has two types of memory: local memory and heap. Local memory is not persistent and is destroyed when the application call ends. Heap, on the other hand, is persistent and can be used for persisting data between application calls. Local memory is used for storing local variables and is not directly addressable. Heap is addressable and provides low level direct access. Both local memory and heap are limited, but the limit is not specified by the AscEE. If an application tries to use too much memory, that may cause the execution session to end abruptly. In that case, the execution session will not have any effects on the state of heap.

2.3.1 Heap Chunks

The AscEE heap is split into chunks. Each heap chunk is a continuous finite array of bytes, has a unique identifier, and is byte addressable. An application may have several heap chunks with different sizes and can remove or resize its chunks or allocate new ones. Every chunk belongs to exactly one application. Only the owner application can modify a chunk but there is no restrictions for reading a chunk[3].

[1] External requests are requests that are not made by other Argennon applications.

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

[3] 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.

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 the new identifier has a correct format. This is an important feature of the AscEE heap which enables applications to use the AscEE heap as a map data structure[4]. 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. (See Section 2.4)

During an execution session every heap chunk has an access-type which may disallow certain operations during that session. This access-type is declared by the initiator request of the execution session for every chunk:

- `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. By additive, we mean an addition-like operator without overflow checking which is associative and commutative. Note that the content of the chunk cannot be read.

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` can be determined uniquely at the start of block validation and is proposed by the block proposer for each block. This value is calculated based on resizing values declared by external requests (i.e. transactions) that want to perform chunk resizing and needs to be an upper bound of all the declared resizing values, indicating an upper bound of `chunkSize` during the execution of the requests of a block.

The address space of a chunk starts from zero and only offsets lower than `sizeUpperBound` are valid. Trying to access any offset higher than `sizeUpperBound` will always cause the execution session to end abruptly. The value of `sizeUpperBound` is always greater than `chunkSize` and there is no way for an application to query `sizeUpperBound`. As a result, in the view of an application, accessing offsets higher than `chunkSize` results in undefined behaviour, while the behaviour is well-defined in the view of a validator. This enables validators to determine the validity of an offset at the start of the block validation in a preprocessing phase without actually executing requests, while `sizeUpperBound` can be determined in a simple parallelized algorithm.

The value of `chunkSize`, can be modified during an execution session. However, the new values of size can only be increasing or decreasing. More precisely, if a request

[4]also called a dictionary.

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 `chunkSize`. 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 assumptions about the content of memory locations that are outside the chunk. While these locations are zero initialized at the start of every execution session, 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.

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 may 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.

2.4 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*[5].

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 the *applications* prefix tree. An `appID` cannot be `0x0`.
- `accountID` is a prefix code built by the *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 the *local* prefix tree:

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

All Argennon prefix trees have an equal branching factor β[6], and we can represent

[5] Also called tries.

[6] A typical choice for β is 2^8.

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 ... d_i)_β < A^(i) then
        return d_1 d_2 ... d_i
    end
end
return NIL
```

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 $A^{(i)} \leq A^{(i+1)}$. [7]

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 start of the sequence. The matching can be done with only three comparisons, while invalid identifiers 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.

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 this request a validator first needs to *inject* the digest of attachments into its HTTP request text. By doing so, the called application will have access to the digest of attachments in a secure way, while it is ensured that the attached requests are included in the current block.

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

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

Algorithm 2: Matching a prefixed identifier

input : A prefixed identifier in base β with n digits: $id = a_1 a_2 \dots a_n$
A sequence of digits in base β: $d_1 d_2 d_3 \dots$
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 $(0.a_1 \dots a_n)_\beta = (0.d_1 \dots d_n)_\beta$ **then**
  if $A^{(n-1)} \leq (0.a_1 a_2 \dots a_n)_\beta < A^{(n)}$ **then**
    return *TRUE*
  end
end
return *FALSE*

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 by the application that handles fee payment, and it is guaranteed that the attached requests are actually included in the current block.

2.6 Authorization

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

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 their claimed issuer accounts. The only difference is that the process of message authentication is performed by the AscEE internally and an application does not explicitly verify cryptographic signatures.

Each execution session has a list of authenticated messages. Each authenticated message has an index which will be used for passing the message to an application as a request parameter. The AscEE uses cryptographic signatures to authenticate messages for user accounts. The signatures are validated during the preprocessing step in parallel, and any type of cryptographic signature scheme can be used.

Also, 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 calculating 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.

Authorizing operations by Authenticated Message Passing and explicit signatures eliminates the need for approval mechanisms or call back patterns in Argennon.[8]

2.7 Reentrancy Protection

The AscEE 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 has 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 The ArgC Language

This section is outdated...

2.9.1 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 ArgC 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.

[8] The AscEE has no instructions for issuing cryptographic signatures.

2.10 Data Dependency Analysis

2.10.1 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.[9]

The fact that block transactions constitute a sequential list, does not imply 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.

[9] This ordering is solely chosen by the block proposer, and users should not have any assumptions about the ordering of transactions in a block.

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

$R_e \leftarrow$ the set of all vertices of V with in degree 0
while $V \neq \varnothing$ **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* $u \in Adj[v_f]$ **do**
      if u *has zero in degree* **then**
        $R_e \leftarrow R_e \cup u$
      end
    end
  end
  if $R_e \neq \varnothing$ **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.

2.10.2 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

$V_e \leftarrow V_d$
$E_e \leftarrow \varnothing$
define a total order on colors of G_d
for *each edge* $\{u, v\} \in E_d$ **do**
  if $color[u] < color[v]$ **then**
    $E_e \leftarrow E_e \cup (u, v)$
  else
    $E_e \leftarrow E_e \cup (v, u)$
  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.

2.10.3 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

The Argennon Prover Machine

3.1 Introduction

The Argennon Prover Machine (APM) is a virtual machine tailored for efficient verification of AscEE computations by argument systems. The APM has a minimal RISC architecture with a very compact instruction set. This ensures that its transition function has an efficient circuit complexity. Here by circuit complexity we mean the size of the smallest circuit that, given two adjacent states in the trace, verifies that the transition between the two states indeed respects the APM specification. The APM is a stack machine and has a random access key-value memory. The word size of the APM is 64-bit.

The Argennon Prover Machine gets as its primary input a vector $(\mathbf{C}_H, \mathbf{C}_P, \mathbf{C}_R)$ where $\mathbf{C}_H$ is a commitment to the AscEE heap, $\mathbf{C}_P$ is a commitment to the AscEE program area and $\mathbf{C}_R$ is a commitment to an ordered list of requests. The APM then outputs the updated commitments to the AscEE heap and program area and a final **accept** flag which indicates if the execution has ended successfully or not: $(\mathbf{C}_{H'}, \mathbf{C}_{P'}, \mathbf{accept})$.

Producing the required outputs from these inputs is not computationally feasible, so the APM receives an auxiliary input vector $(H, \pi_H, P, \pi_P, R, \pi_R, h)$, where π_X is a proof that proves X can be opened w.r.t $\mathbf{C}_X$ and h is a hint that helps the APM make nondeterministic choices correctly.

The APM consists of four main modules:

- **Preprocessor:** This module prepares the input data for other modules. It verifies that H, P, R are valid w.r.t the provided commitments. It also processes the input data and ensures it has the correct format and is valid. For example, it verifies the signatures of authenticated messages or checks that the proposed chunk bounds are valid.

- **Normal Execution Unit (NEU):** This module executes the requests whose execution completes normally. It outputs the updated heap chunks and an $accept_N$ flag. More formally it evaluates $H', accept_N = P(H, R_{good})$, where H' is the updated heap chunks and $accept_N$ flag will be set to false, If an application terminates

abruptly.

- **Failure Repeater Unit (FRU):** This module evaluates $accept_F = P(H', R_{bad})$, where R_{bad} is those requests whose execution terminates abnormally, H' is the output of the NEU and $accept_F$ flag is set to false if a request execution completes normally.

  The FRU has a higher circuit complexity than the NEU. Unlike the NEU, the FRU is able to restore the initial state of the heap when an application fails. It also calculates the execution cost of every instruction and if the application's execution cost exceeds its predefined cap, the FRU will terminate the application.

- **Postprocessor:** This module is responsible for calculating the updated commitments $(\mathbf{C}_{H'}, \mathbf{C}_{P'})$ and the final **accept** flag. Installing new applications or updating the code of existing application is performed by this module.

The configuration of the APM can be considered as a tuple:

$$(\mathcal{S}, \mathcal{L}_{\text{NEU}}, \mathcal{L}_{\text{FRU}}, \mathcal{T}_{\text{NEU}}, \mathcal{T}_{\text{FRU}}) \ ,$$

where $\mathcal{S}$ is the size of the internal stack, $\mathcal{L}_{\text{NEU}}, \mathcal{L}_{\text{FRU}}$ are the amount of local memory of the normal execution and failure repeater units respectively, and $\mathcal{T}_{\text{NEU}}, \mathcal{T}_{\text{FRU}}$ are the number of cycles that the NEU and FRU runs for. It should be noted that the APM does not use a different local memory for each application call.

3.2 Simplifying Complex Components

Assume that we have a component which has a considerable arithmetic circuit complexity. If this complex component is needed in some arbitrary computation steps, we will need to repeat its circuit in every step of the computation. This will increase the complexity of proof generation considerably. Fortunately, there is a workaround. Instead of repeating the circuit of the component, we can use a simpler cryptographic hash calculator circuit during the computation, and use a final verification circuit at the end to verify the functionality of the component.

Every component accepts an input *in* and produces an output *out*. As a result we can denote the component by a deterministic function that maps an input sequence with any arbitrary length k, to an output sequence with the same length:

$$f((in_1, in_2, \ldots, in_k)) \coloneqq (out_1, out_2, \ldots, out_k)$$

In every step of the computation, we replace this component with a cryptographic hash calculator which receives *in* and *out* as its inputs and gets activated in the same computation steps that the component must be active. When the hash calculator is active, it hashes its inputs, so at the end of the computation it has computed a digest:

$$h(in_1, out_1, in_2, out_2, \ldots, in_k, out_k) \coloneqq digest_f$$

where k is the number of steps that our component must have been active.

Now the prover needs to prove that he knows values $in_1, out_1, \ldots, in_k, out_k$ such that they are correctly produced by the component:

$$f((in_1, \ldots, in_k)) = (out_1, \ldots, out_k)$$

and their digest is also correct:

$$h(in_1, out_1, \ldots, in_k, out_k) = digest_f$$

where functions f and h, are both known to the prover and verifier.

The circuit for verifying this assessment usually is straight forward. When the computation involves a large number of steps and the component is complex, this approach can reduce the cost of proof generation considerably. Memory components are good candidates for being simplified by this method.

Interestingly, this approach can reduce the number of computation steps as well. When we use a hash calculator circuit instead of the component, *out* will be available in the same computation step that *in* is available. This will eliminate the computation step that is needed for generating *out* by our component's circuit.

Chapter 4

Persistence Layer

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

4.1 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 chunks. Every page has a *native* chunk that has the same identifier as the page. In addition to the native chunk, a page can host any number of *migrant* chunks. A page of the AscEE storage should consist of chunks that have very similar access patterns. Ideally, when a page is needed for validating a block, almost all of its chunks should be needed for either reading or writing. We prefer that the chunks are needed for the same access type. In other words, the chunks of a page should be chosen in a way that for validating a block, we need to either read all of them or modify[3] all of them.

When a page contains migrants, its native chunk can not be migrated. If the page does not have any migrants, its native chunk can be migrated and after that the page will be converted into a special `<moved>` page. When a non-native chunk is migrated to another page, it will be simply removed from the page.

4.2 Publicly Verifiable Database Servers

Pages of the AscEE storage are persisted using *dynamic universal accumulators*. Argennon has two dynamic accumulators: *program* database, which stores the AscEE program

[1] also it stores applications' constants.

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

[3] and probably read.

area, and *heap* database, which stores the AscEE heap. The commitment of these accumulators are included in every block of the Argennon blockchain. In the Argennon cloud, nodes that store these accumulators are called Publicly Verifiable Database (PV-DB) servers.[4]

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.

The commitments of the AscEE cryptographic accumulators are affected by the way chunks are clustered in pages. 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 a chunk from one page to another or directives specifying which pages will contain newly created chunks. These directives are applied at the end of block validation, after executing requests.

A block proposer is allowed to obtain clustering directives from any third party source.[5] 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.

4.3 Object Clustering Algorithm

not yet written...

[4] Usually PVC servers are also PV-DB servers.

[5] We can say the AscEE clustering algorithm is essentially off-chain.

Chapter 5

Networking Layer

5.1 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 network of PVC servers. PVC servers are a crucial part of the Argennon ecosystem, and they form the backbone of the Argennon networking layer.

not yet written...

5.2 Censorship Resilient Mode

not yet written...

Chapter 6

The Argennon Blockchain

6.1 Blocks

The Argennon blockchain is a sequence of blocks. Every block represents an ordered list of external requests, intended to be executed by the Argennon Smart Contract Execution Environment (AscEE). The first block of the blockchain, the *genesis* block, is a spacial block that fully describes the initial state of the AscEE and every block of the Argennon blockchain thus corresponds to a unique AscEE state which can be calculated deterministically from the genesis block.

A block of the Argennon blockchain contains the following information:

Block
height: h
commitment to the program database: $\mathbf{C}_P$
commitment to the heap database: $\mathbf{C}_H$
commitment to the ordered list of requests: $\mathbf{C}_R$
clustering directives: dir
certificate of the validator assembly for the block with height $h-k$: $v\text{-}cert_{h-k}$
previous block hash

6.1.1 Block Certificate

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

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).

The Argennon network uses a parallel algorithm to produce block certificates and therefore the signature scheme needs to satisfy certain properties:

- **Associative aggregation:** the signature aggregation operator is associative.
- **Efficient cancellation:** if S is a predefined and fixed set of users and U is an arbitrary subset of S, verifying an aggregate signature of $S - U$ can be done in time $O(T + |U|)$, if the aggregate signature of S can be verified in $O(T)$.

An example for a signature scheme that supports all these properties is the BLS signature scheme.

BLS Signatures

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

$$\sigma \leftarrow \sigma_1 \cdots \sigma_n \in \mathbb{G}_0 \ . \tag{6.1}$$

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

$$e(g_1, \sigma) = e\big(pk_1,\ H_0(m_1)\big) \cdots e\big(pk_n,\ H_0(m_n)\big) \ . \tag{6.2}$$

When all the messages are the same ($m = m_1 = \ldots = m_n$), the verification relation (6.2) reduces to a simpler test that requires only two pairings:

$$e(g_1, \sigma) = e\Big(pk_1 \cdots pk_n,\ H_0(m)\Big) \cdot \tag{6.3}$$

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 6.4, 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.

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:

$$cert_i = (\sigma_i,\ C - S_i)\ , \tag{6.4}$$

where σ_i is the aggregate signature issued by S_i.

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

$$apk_i = apk_C apk_{C-S_i}^{-1}\ , \tag{6.5}$$

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:

$$apk_i = apk_{i-1} apk_{S_i - S_{i-1}} apk_{S_{i-1}-S_i}^{-1}\ . \tag{6.6}$$

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]

6.1.2 Block Validation

To validate a block three main conditions need to be validated: (i) commitments to the program and heap database are resulted from applying the request list and clustering directives of the block to the previous AscEE state, (ii) previous block is valid and has height $h-1$, (iii) $v\text{-}cert_{h-k}$ is valid.

We can denote condition (i) by a Computational Integrity (CI) statement:

$$\mathbf{C}_{P_h}, \mathbf{C}_{H_h} \coloneqq \tau(\mathbf{C}_{P_{h-1}}, \mathbf{C}_{H_{h-1}}, \mathbf{C}_{R_h}, dir_h)\ , \tag{6.7}$$

where τ is a transition function that encodes the AscEE computation logic and the necessary preprocessing and postprocessing steps[3].

[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 $(ab)^{-1} = a^{-1}b^{-1}$.

[3] including opening and updating the state commitments

Verifying statement 6.7 can be done by either replaying the AscEE computation and performing the required preprocessing and postprocessing steps, or alternatively by verifying a computational integrity proof[4]. To use computational integrity proofs the verifier and prover need to share an arithmetized representation of τ and use it for both proof generation and verification.

The Argennon Prover Machine provides a convenient and universal way for arithmetization of any Argennon application. Moreover, since a compiled version of every Argennon application to the APM code is stored in the AscEE program area, validators that use the APM do not need to locally store the APM code of applications.

The Argennon protocol does not enforce the usage of the APM. Validators and PVC servers can use any argument system with any arithmetization, and if required, the ASAR of an application can be used for generating the appropriate arithmetization instead of the APM arithmetization.

For validating the previous block, instead of directly validating the contents of the block, a validator only verifies the block certificates. Each block of the Argennon blockchain has two certificates: the certificate of validators, *v-cert*, and the certificate of delegates, *d-cert*. Verifying $d\text{-}cert_{h-1}$ is straightforward but verifying $v\text{-}cert_{h-1}$ is more challenging. $v\text{-}cert_{h-1}$ is an aggregate signature and validating it requires accessing the staking database at block $h-1-m$, where m is the total number of validator assemblies, to obtain public keys and stake values. Again, in addition to direct verification, a validator can use computational integrity proofs, received from the Argennon cloud, to perform cheaper verification of this certificate.

The block at height h includes a certificate of validators for block $h-k$ which is used to record the participation of validators and facilitate reward calculation. This certificate needs to be validated based on stake and public key database at block $h-k-m$, and can be cheaply done by using computational proofs, obtained from the Argennon cloud. Here k is the maximum allowed length of the unvalidated part of the Argennon blockchain. (See Section 6.2.2)

This type of block validation only validates the transition from block $h-1$ to block h, and the block is valid only if its previous block is valid. We call this type of block validation *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. Moreover, the required proofs can be generated independently and by different PVC servers. As we will see in Section 6.2.2, this property plays an important role in the Argennon consensus protocol.

In summary, a validator validates a block by verifying three computational integrity proofs: π_τ which proves the transition is correct, π_{h-1} which proves the previous block has a correct certificate of validators and π_{h-k} which proves the validity of the included certificate of validators of block $h-k$. It should be noted that these proofs are not part of the block contents and therefore the exact argument system used for generating them

[4]More accurately we should call it an argument instead of a proof. However, using the word argument can confuse a reader who is not familiar with the subject, so we avoid it here.

is not specified by the Argennon protocol.

6.2 Consensus

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

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

Every block of the Argennon blockchain needs a certificate from both the committee of delegates and the assembly of validators. A block is considered final after its **next** block receives **both** of its certificates. In Argennon as long as more than 2/3 of the total stake of validators is controlled by honest users, the probability of discarding a final block is zero even if all the delegates are malicious.

Having multiple assemblies gives validators some "resting" time and allows a validator to be out of sync with the network for some time, without losing the opportunity to vote for any blocks.

In addition to primary committees, Argennon could have several community driven committees. Certificates of these committees are not required for block finality, but they could be used by members of validator assemblies 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.

6.2.1 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[5]). At the start of the Argennon main-net, this committee will be elected for one-year terms and will have five members. Later, this can be changed by the ADAGs in a procedure described in Section 7.1.

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

Besides the main committee, a reserve committee of delegates consisting of three members is elected by validators either through the ADAGs or by the *emergency agree-*

[5]pronounced /eɪ-dagz/.

ment 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 the main committee comes back online or a new committee is elected through the ADAGs.

A block certified only by the committee of delegates is relatively credible, but it is not considered final until its next block receives the certificate of its validator assembly. Since a block at height h contains the validators certificate of the block at height $h - k$, the unvalidated part of the Argennon blockchain can not be longer than k blocks.

The committee of delegates may use any type of agreement protocol to reach consensus on the next block. Usually the delegates are large organizations, and they can communicate with each other efficiently using their reliable networking infrastructure. This mostly eliminates the complications of their consensus protocol and any protocol could have a good performance in practice. 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. For better performance, the delegates should run their agreement protocol for reaching consensus about small batches of transactions in their mem-pools, instead of the whole block.

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

6.2.2 The Assemblies of 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 `minValidatorStake` threshold is considered a *validator*. The `minValidatorStake` threshold is determined by the ADAGs, but it can never be higher than 1000 ARG.

Every `AssemblyLifeTime` number of blocks, randomly m assemblies are selected from validators, in a way that the total stakes of different assemblies are almost equal, and every account is a member of **at least** one assembly.[6]

The value of m is determined by the ADAGs, but it can never be higher than 32. This way, it is guaranteed that on average, any block of the Argennon blockchain is validated by at least 2% of the total ARG supply.

Signing the Block Certificate

The delegates can generate blocks very fast. Consequently, the Argennon blockchain always has an unvalidated part which contains the 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 h needs a certificate from the assembly of validators with index h modulo m. To decide about signing the certificate of a block

[6] An account can be a member of multiple assemblies.

which already has a certificate from the delegates, a validator checks the conditional validity of the block (See Section 6.1.2), 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 signature **only after** he sees the certificate of the validator assembly of the previous block. Some validators may also require seeing a certificate from some community based committees. An honest validator never signs a certificate for two different blocks with the same height.

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

Signature Aggregation

The validators certificate of a block is an aggregate signature of members of the corresponding assembly of validators. Validator assemblies could include millions of users and calculating their aggregate signature requires an efficient distributed algorithm.

In Argennon, signature aggregation is mostly performed by PVC servers. To distribute the aggregation workload between different servers, every validator assembly is divided into pre-determined groups, and each PVC 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 PVC servers and each group should be assigned to multiple PVC 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 PVC 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 broadcast it to the PVC servers network.

The role of the delegates in the signature aggregation algorithm is limited. The important part of the work is done by PVC servers and slightly modified versions of this algorithm can perform signature aggregation even if all the delegates are malicious, as long as there are enough honest PVC servers.

Activity Status

Every validator has a status which can be either `online` or `offline`. This status is stored in the ARG application and is part of the staking database. A validator can change his status to `offline` through an external request (transaction) to the ARG application. In this request he exactly specifies for how long he wants to be offline and after this period his status will be automatically considered `online` again. When a validator sets

his status to `offline` for some period of time, he will receive a small portion of the maximum possible reward that a validator can receive in that period of time by actively participating in the consensus protocol. This ensures that a validator has an incentive for changing his status to offline rather than simply becoming inactive.

The staking database of a validator assembly can be updated only by the assembly itself. That means, an external request which changes the status of a validator can be included only in a block that is validated by the assembly of that validator.

There is no transaction type for changing the status of a validator to `online`. A malicious committee of delegates would be able to censor this type of transactions and prevent honest validators from coming back online. For this reason the status of a validator is considered online automatically when the specified period of time for being offline ends.

A block certificate issued by some members of a validators assembly is considered valid, if according to the staking database of the previous block **certified by the same assembly**, we have:[7]

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

If according to the staking database of block h, the total online stake of the assembly with index h modulo m is lower than `minOnlineStake` threshold, the block $h + m$ can never be certified by validators. To prevent the blockchain from halting in such situations, the validator assembly with index h modulo m will get merged into the assembly that has the most online stake at block h. This will decrease the number of assemblies to $m - 1$, and the indices of assemblies will be updated appropriately.

The merging will continue recursively until the online stake of all remaining assemblies is higher than `minOnlineStake` fraction. If eventually all assemblies get merged together and only one assembly remains, the condition for validity of block certificates changes: A certificate of validators will be considered valid if the sum of stakes of the certificate signers is higher than 2/3 of the total stake of validators and `online/offline` status of validators becomes ineffective. This prevents the system from going into temporary deadlocks and the community will always be able to preserve the liveliness without waiting for the expiration of offline status of some accounts.

Analysis

We analyze the minimum amount of stake that is required for conducting different types of attacks against the Argennon blockchain. In these attack scenarios, we assume that a

[7] If we calculate the stake values based on the previous block, a malicious assembly can select the validators of the next block by manipulating the staking database.

single validator assembly is corrupted, all the delegates are malicious and the adversary is able to fully control message transmission between nodes and partition the network arbitrarily.

We denote the total stake of the corrupted validator assembly with s and the total stake of malicious users of the assembly with m. We use d to denote the stake of users of the assembly who have `offline` status and h to denote the stake of users of the assembly who have `online` status and do not participate in the protocol.[8] We assume that a certificate is accepted if it is signed by more than r fraction of the total online stake of the assembly. We obtain the minimum required malicious stake for three types of attacks:

- Certifying an invalid block:
$$m > r(s - d)$$
- Forking the blockchain by double voting and network partitioning:
$$m > (2r - 1)(s - d) + h$$
- Halting the blockchain by refusing to vote:
$$m > (1 - r)(s - d) - h$$

In Argennon we have $r = \frac{3}{4}$ and $d < \frac{1}{3}s$. Consequently, in Argennon to confirm an invalid block, the adversary needs at least $\frac{1}{2}$ of the total stake of an assembly. For forking the blockchain, interestingly m is minimized when $h = 0$, thus the minimum required stake is $\frac{1}{3}s$. For halting the blockchain, an adversary requires a stake higher than $\frac{1}{6}s - h$.

In particular, we are interested in comparing the Argennon protocol with a simple protocol that accepts a certificate if it is signed by more than $\frac{2}{3}$ of the total stake of the assembly and there is no `online/offline` status for users.

We observe that the minimum required stake for halting the blockchain in the Argennon protocol will be higher, as long as the following inequality holds:

$$\frac{1}{4}(s - d) - h > \frac{1}{3}s - (d + h) \cdot$$

So if $d > \frac{1}{9}s$, the minimum required stake for halting the blockchain is higher in the Argennon protocol and the value of h does not matter.

6.2.3 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

[8] d stands for *deactivated* and h stands for *hidden*.

executed occasionally. Ideally this protocol should never be used during the lifetime of the Argennon blockchain.

We assume that an adversary is able to fully control message transmission between users and is able to partition the network arbitrarily for finite periods of time. Under these circumstances, the recovery protocol can recover the functionality of the Argennon blockchain as long as more than 2/3 of the total stake of every validator assembly is controlled by honest users. The recovery protocol uses two main emergency procedures to recover the functionality of the Argennon blockchain: the *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 validators. A valid fork request is an unexpired request signed by more than half of the total stake of validators.

A fork created by the reserve committee needs to be confirmed by validators and can never discard more than one block which has received a certificate from validators.

For forking at block h, the reserve committee of delegates makes a special *fork block* which only contains a valid fork request, and its parent is the block h. The height of the fork block therefore is $h + 1$, and the fork block needs a valid certificate from the assembly of validators with index $h + 1$ modulo m. When a fork block gets certified by validators, its parent is also confirmed and will become a part of the blockchain, even if it does not have a certificate of validators.

For signing a fork block at height $h + 1$, a validator ensures that the following conditions hold:

- the fork block is signed by the reserve committee.
- the fork block contains a valid fork request.
- the parent block of the fork block is issued by the previous committee.
- the parent block of the fork block is certified by validators, or the parent block is conditionally valid and there is a fork block with height h which is certified by validators, or the parent block is conditionally valid and the parent block of the parent has a validators certificate.
- the validator has not already signed a certificate for a fork or normal block at height $h + 1$.

The parent of the fork block does not necessarily need a validators certificate. 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. Notice that the block before the parent always needs a validators certificate.

A validator always chooses a valid fork block over a block of the main chain and may sign different fork blocks with different heights. However, as we mentioned before, an

honest validator **never signs a certificate for two different blocks with the same height**. Consequently, a validator never signs two fork blocks at the same height, and if he has already signed the fork block at height $h + 1$, he will not sign the block $h + 1$ of the main chain and vice versa.

The reserve committee of delegates is allowed to generate multiple fork blocks with different heights, as long as the parent block is generated by the previous committee. When the reserve committee generates multiple fork blocks at different heights, the next normal block must be always added after the fork block with the highest height.

The reserve committee of delegates should try to perform the emergency forking in such a way that valid blocks do not get discarded, including blocks that have not been certified by the validators yet.

For forking the blockchain, the reserve committee uses a straightforward algorithm: let h_v be the height of the last block with a validator certificate and $h_v + k$ be the height of the last valid block that the reserve committee has seen. For forking the main chain, the reserve committee generates all fork blocks with heights $h_v + 1, h_v + 2, \ldots, h_v + k + 1$. The parent of the fork block with height $h_v + i$ will be the block $h_v + i - 1$ of the main chain. The reserve committee will wait until the fork block with height $h_v + k + 1$ receives a certificate from validators and then will continue the normal chain after that fork block. Hence, the fork block with height $h_v + k + 1$ will be the parent of the first normal block generated by the reserve committee.

Analysis When the reserve committee gets activated, the main committee might have been malicious, so any number of blocks could exist at each height. However, at each height at max one block can have a validators certificate. Moreover, if at some height there are not any blocks with a validators certificate, then no blocks at higher heights can have a validators certificate either, because validators do not sign the certificate of a normal block if its parent does not have a certificate.

If $h_{\max}$ denotes the height of the highest block with a validators certificate, as long as more than $2/3$ stake of every assembly of validators is honest, for a fork block with height h_f we have:[9]

- if $h_f \leq h_{\max}$, the fork block can not receive a certificate from validators.
- if $h_f = h_{\max} + 1$, when there is no main block with height h_f, the fork block can always receive a certificate from validators, otherwise the fork block may receive a certificate or not. It is possible that the validators of the assembly with index h_f modulo m get divided between the fork block and a block at height h_f of the main chain.
- if $h_f = h_{\max} + 2$, the fork block can always receive a certificate from validators, if network partitions do not last forever.
- if $h_f \geq h_{\max} + 3$, the fork block can always receive a certificate from validators **only if** a fork block at height $h_f - 1$ gets certified by validators.

[9] This fork block forks the blockchain at height $h_f - 1$.

The reserve committee must be able to create at least one fork block which gets a certificate from validators. The reserve committee may not know the value of $h_{\max}$.[10] However, if the reserve committee creates all fork blocks with heights $h_0, h_0+1, h_0+2, \ldots$ for some $h_0 \leq h_{\max} + 2$, then every fork block with height $h \geq h_{\max} + 2$ will surely get a certificate. Obviously $h_v \leq h_{\max}$, so if the main chain contains a block with height $h_{\max}+1$ the reserve committee should be able to eventually find this block and generate the fork block with height $h_{\max} + 2$, which will surely get a certificate. If there is no block in the main chain with height $h_{\max} + 1$, then the fork block with height $h_{\max} + 1$ can get a certificate. This way, the reserve committee will always be able to continue the chain after the certified fork block with the highest height.

If two fork blocks at heights h_0 and $h_0 + k$ are generated by the reserve committee and both blocks receive a certified from validators, we must have $h_0 > h_{\max}$, and there must exist fork blocks with heights $h_0+1, \ldots, h_0+k-1$ which are certified by validators. That means if a malicious reserve committee generates a normal block after any fork block with height less than $h_0 + k$, that normal block can not receive a certificate from validators.

If a malicious reserve committee creates two fork blocks with the same height, either only one of them can get a certificate, which makes the other one ineffective or validators get split between blocks and the chain will halt. In this case the emergency agreement protocol will start.

During the emergency forking, no more than one block with a validators certificate can be discarded. A fork block with height $h + 1$ can not receive a certificate without a certified normal parent block with height h or $h - 1$, or another certified fork block with height h.

One certified block may be discarded when the malicious main committee of delegates has forked the main chain by producing blocks b_1 and b_2 with heights h; the block b_1 has received a validators certificate and validators have not certified any blocks at height $h + 1$. The reserve committee adds a fork block whose parent is block b_2, and that will essentially discard b_1. Notice that if a block at height $h + 1$ had been certified by validators the fork block after b_2 could not have received a certificate from validators.

Emergency Agreement

The emergency agreement protocol is a resilient protocol for deciding between a set of proposals when the committee of delegates is not available or can not be trusted. For initiating the protocol, a validator signs a message containing the subject of the agreement and a start time.

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 emergency agreement protocol is essentially an election procedure and involves

[10] This could happen when the network is partitioned.

human interaction. Users need to determine who they want to vote for by interacting with the software. As long as users can not agree upon electing a candidate, the voting process has to continue.

The voting process is done in rounds and each round usually lasts for approximately λ units of time. λ is selected by the ADAGs and could be several hours. All votes and messages **are tagged** in a way that a vote cast in a round can not be used in another round. Votes are weighted based on users' stakes and `online/offline` status of users is not considered. When we say 2/3 votes, we mean the sum of the stake of voters is 2/3 of the total stake.

A user executes the following procedure in each agreement session:

Voting Phase in Round r:

- if the user has locked his vote on a proposal p, he votes p, otherwise he votes a single desired proposal.
- when $clock = \lambda$, if the user has seen more than 2/3 votes for a proposal p, he votes p-win, otherwise he votes $draw$. A user votes either p-win or $draw$, not both.
- if the user sees more than 2/3 $draw$ votes, he goes to the round $r + 1$ and sets $clock = 0$.
- if the user sees more than 2/3 p-lock votes, he goes to the round $r+1$, sets $clock = 0$ and locks his vote on p.
- when $clock = k\lambda$ for $k = 2, 3, \ldots$, if the user has seen more than 2/3 votes for a proposal p he votes p-lock.

Termination:

- as soon as the user sees more than 2/3 p-win votes for p, he selects p and ends the agreement protocol. The p-win votes can be for any round, but all must belong to the same round.

We assume that users have clocks with the same speed, and $\lambda \gg \delta$, where δ 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 rounds.

A honest user during a round only votes a single proposal. This ensures that as long as more than 2/3 of the total stake of the system is controlled by honest users, no two different proposals can get more than 2/3 votes. As a result, we can not have 2/3 votes for both p-lock and p'-lock if $p \neq p'$. Also, a honest user either votes p-win or *draw*, so only one of p-win or *draw* can get more than 2/3 votes and when a user sees more than 2/3 p-win votes, he can be sure that *draw* has less than 2/3 votes. Therefore, for going to the next round we will need 2/3 p-lock votes and all honest users will lock their vote on p when they start the next round. As a result only p can be confirmed by the agreement protocol.

Note that when a single honest user terminates the protocol, he can convince all other honest users to terminate their protocol by sending those 2/3 p-win votes that he has seen.

If honest users can agree upon some proposal value, the agreement protocol will converge to that value. When a round ends and we go to the next round, all honest users will lock their vote on the same proposal or no one will lock his vote, so an agreement could be reached in next rounds. We will never get stuck in a round. If at some round *draw* gets less than 2/3 votes, that means at least ϵ honest stake has voted p-win.[11] That means there must be more than 2/3 votes for some proposal p which convinced the ϵ honest stake to vote p-win. Therefore, after waiting long enough, all the honest stake will see those votes and will eventually vote for p-lock, and p-lock will get more than 2/3 votes.

Initiating the Recovery Protocol

When the validator software does not receive any blocks for `blockTimeOut` amount of time, or when it observes an evidence which proves the delegates are malicious, after prompting the user and after his confirmation, it will initiate the recovery protocol.

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

In the next step, it will sign and broadcast an **emergency fork request** message, alongside some useful metadata such as the last valid block of its blockchain and the evidence of delegates' misbehaviour.[12] Before starting the recovery protocol, validators try to synchronize their blockchains as much as possible.

If the reserve committee of delegates is already active, or if the validator software 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, after user confirmation, it 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.

[11] It is possible that the ϵ stake has not voted yet. However based on the finite time partitioning assumption, at some point that honest stake should get connected to the network and vote.

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

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

6.2.4 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.

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:

$$S_{u,t} = \min(B_{u,t}, Trust_{u,k}) \ , \tag{6.8}$$

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:

$$Trust_{u,k} = L_{u,k} + M_{u,t_k} \ , \tag{6.9}$$

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:

$$\alpha = -\frac{\ln\left(1 - \frac{M_{n+k}}{B}\right)}{n} \ . \tag{6.10}$$

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.

6.2.5 Analysis

not yet written...

6.3 Applications

An Argennon application or smart contract is an HTTP server which is represented by an Argennon Standard Application Representation (ASAR) 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 2.4) 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 symbolic representations of 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 applications use HTTP as the application protocol, and they are advised to have a RESTful API design.

6.3.1 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 7.1)

6.3.2 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 stake values.

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

6.4 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 2.4) 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.[13]

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.

> *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.*

6.5 External Requests

An Argennon *external request* (i.e. transaction) consist of an HTTP request made by a user to an Argennon application, a resource declaration object and a list of signed messages. External requests can only be issued by users and requests created by applications are called *internal request*.

6.5.1 Resource Declaration Object

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

- Maximum execution cost
- The list of applications the request will call

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

An Argennon transaction in YAML format

```
---
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: LNUC49Lhyz702uszzNcfaU3BhPIbdaSgzqDUKzbJzLPTIFS2J9GzHI-cDKb

caps:
    maxCost: 150 # the cost of execution by the APM
    apps: [1,124.16]
    read: [(2654,3),(15642,0),(15642,1),(15642,3)]
    write: [(15642,0),(20154,0),(20154,1)]
```

- The list of chunks 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.

6.6 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. In Argennon the reference for resource usage is the Argennon Prover Machine. The APM has two different execution units: the NEU which is used for executing external requests that the block proposer claims will complete successfully and the FRU which executes requests that the proposer has classified as failed requests. The FRU has a more restricted resource management and less resources are available for requests that use the FRU.

We recall from Chapter 3 that the APM configuration is a tuple:

$$(\mathcal{S}, \mathcal{L}_{\text{NEU}}, \mathcal{L}_{\text{FRU}}, \mathcal{T}_{\text{NEU}}, \mathcal{T}_{\text{FRU}}).$$

When requests are executed by the APM, the following run-time resources should be considered:

- **execution cost**: each APM instruction has a protocol defined cost and the execution cost of a program can be calculated deterministically. Only the FRU performs these calculations and verifies that an external request does not exceed its predeclared execution cost.

  For executing the whole requests of a block, at max, The NEU will run for $\mathcal{T}_{\text{NEU}}$ steps and the FRU will run for $\mathcal{T}_{\text{FRU}}$ steps.

- **local memory**: denoted by $\mathcal{L}_{\text{NEU}}, \mathcal{L}_{\text{FRU}}$. The Argennon protocol requires: $\mathcal{L}_{\text{NEU}} = 3\mathcal{L}_{\text{FRU}}$, so the Normal Execution Unit has three times more local memory.

- **stack size**: both the FPU and the NEU has the same stack size: $\mathcal{S}$

- **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. Both the NEU and the FRU enforce this restriction.

- **app access list**: a session may only make requests to applications that are declared in its application access list. Both the NEU and the FRU enforce this restriction.

- **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. Only the FRU enforces this restriction.

In Argennon the proposer does not use the APM for executing requests and directly executes the ASAR of an application. That is much more efficient. Only when the proposer wants to reject a transaction due to excessive resource usage, he will emulate the APM to make sure the request will be rejected by the FRU. The differences between the NEU and FRU ensures that this policy is safe.

6.7 Incentive mechanism

6.7.1 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[14].

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`.

6.7.2 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.

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

6.7.3 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.

Punishable 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.[15]

6.7.4 Incentives for PVC Servers

The incentive mechanism for PVC servers should have the following properties:

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

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

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

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

To incentivize PVC servers, a lottery will be held every round,[16] 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 PVC server could own multiple winning tickets in a round.

[15] 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.

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

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 PVC 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 PVC 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 storage page which was selected as the round challenge. This will encourage PVC 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 7

Governance

7.1 ADAGs

The Argennon Decentralized Autonomous Governance system (ADAGs)

not yet written...

Chapter 8

The Argon Language

8.1 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`.

8.2 Features Overview

8.2.1 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...
}
```

8.2.2 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.