Contents

Chapter 1

The Argennon Virtual Machine

1.1 Introduction

The Argennon[1] Virtual Machine (AVM) is an abstract computing machine for executing Argennon's smart contracts. The Argennon Virtual Machine knows nothing of the Argennon blockchain, transactions or any concept of gas usage, only of Argennon's identifier tries and the concept of execution sessions.

Execution sessions are separate sessions of executing smart contract's code by the Argennon Virtual Machine. Every execution session starts with a single method invocation, but does not necessarily end when that method invocation completes. During an execution session additional method invocations could be scheduled, which will be executed after the initial method completion.

An execution session may complete normally or abruptly. When an execution session completes abruptly, it will not have any effects on the AVM state.

1.2 Data Types

The Argennon Virtual Machine expects that all type checking is done prior to run time, typically by a compiler, and does not have to be done by the Argennon Virtual Machine itself.

The Argennon Virtual Machine operates on two kinds of types: primitive types and identifier types. There are, correspondingly, two kinds of values that can be stored in memory locations, passed as arguments, returned by methods, and operated upon: primitive values and identifier values.

Values of primitive or identifier types need not be tagged or otherwise be inspectable to determine their types at run time, or to be distinguished from values of other types. Instead, the instruction set of the Argennon Virtual Machine distinguishes its operand types using instructions intended to operate on values of specific types. For instance,

[1]the classical pronunciation should be used:/ɑrˈgen.non/

`iadd64` assumes that its operands are two 64-bit signed integers or `delete` assumes that its operand is an identifier type.

An identifier value in the Argennon Virtual Machine is a variable length array of bytes. Some instructions that work with identifiers are able to determine the length of their identifier operands, while some other instructions, for performance reasons, require the length to be specified.

The Argennon virtual machine does not have a fixed word size. Any instruction of the AVM has its specific word size, and the addressable memory areas are byte addressable.

1.3 Identifiers

In the Argennon Virtual Machine four distinct identifier types exist: `applicationID`, `accountID`, `methodID` and `chunkID`.

All these identifiers are *prefix codes*, and hence can be represented by *prefix trees*[2]. The Argennon has three primitive prefix trees: *applications, accounts* and *local.* Any identifier in Argennon is a prefix code built by using one or more of these prefix trees:

- `applicationID` is a prefix code built by *applications* prefix tree.
- `accountID` is a prefix code built by *accounts* prefix tree.
- `methodID` is a composite prefix code built by concatenating an `applicationID` to a prefix code made by *local* prefix tree: (`|` is the concatenating operator)

  `methodID = (applicationID|<local-prefix-code>)`
- `chunkID` is a composite prefix code built by concatenating an `applicationID` to an `accountID` to a prefix code made by *local* prefix tree:

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

All Argennon prefix trees have an equal branching factor β. Therefore, we can represent an Argennon prefix tree as a sequence of fractional numbers[3] 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 $A^{(i)} < A^{(i+1)}$. A typical choice for β could be 2^8.

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 k first digits belong to an Argennon's identifier, but we don't know the value of k. 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.

[2] Also called tries.

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

Algorithm 1: Finding a prefixed identifier

input : A sequence of n digits in base β: $d_1 d_2 \ldots d_n$
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.d1d2...di)_β < A^(i) then
        return d1d2...di
    end
end
return NIL
```

In Argennon the shorter prefix codes are assigned to more active accounts and smart contracts 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.

1.4 Arithmetics

The Argennon Virtual Machine supports signed integer and signed floating point operations. The Argennon Virtual Machine does not support any type of unsigned arithmetics. All arithmetic operations in the Argennon Virtual Machine are checked and any type of overflow or underflow will cause a catchable exception to be thrown.

1.5 Architecture

1.5.1 The `pc` Register

The Argennon Virtual Machine always has a single thread of execution and exactly has one `pc` register. When the Argennon Virtual Machine is executing a method, if that method is not native, the `pc` register contains the address of the AVM instruction currently being executed. If the method currently being executed is native, the value of the Argennon Virtual Machine's `pc` register is undefined.

1.5.2 Call Stack Queue

Every AVM execution session has a queue of call stacks. A call stack contains all the information that is needed for restoring the AVM state, and continuing the execution after the method invocation completes. This information is represented by a `CallInfo` struct:

```
CallInfo {
    methodID,
```

```
    pc,
    localFrame,
    operandStack
}.
```

The `methodID` field is the unique identifier of a method's bytecode that is a composite prefix code built by concatenating an `applicationID` to a local identifier.

Every method invocation has a corresponding `CallInfo` struct and when a method is invoked its `CallInfo` struct is pushed onto the call stack. Hence, the top of the call stack always contains the `CallInfo` struct of the current method. When a method invocation completes, its call information is popped from the call stack.

An AVM execution session will continue as long as there is a non-empty call stack in the call stack queue. When the execution of the current call stack finishes, the execution of the next call stack in the call stack queue starts. It is possible that an AVM method invocation creates new call stacks. These newly created call stacks will be a part of the current execution session and are added at the end of the call stack queue. A newly created call stack always contains a single `CallInfo` struct. See Section 1.6.1 for more details.

1.5.3 Run-Time Data Areas

The Argennon Virtual Machine defines various run-time data areas that are used during an execution session (i.e. transaction), Some of these data areas are persistent and are stored on the blockchain, Other data areas are per execution session or per method invocation. These data areas are created when their context starts and destroyed when their context ends.

The Argennon Virtual Machine has five run-time data areas:

- Method Area
- Constant Area
- Local Frame
- Operand Stack
- Heap

All data areas except operand stack, have their own address space. Operand stack is a last-in-first-out (LIFO) stack and is not addressable. Every AVM instruction operates on its specific data areas.

Method Area

The method area contains the byte-code of any method which can be called by a non-native method invocation instruction. In the Argennon Virtual Machine every method has a unique identifier: `methodID`, and the method area is a map from method identifiers to their byte codes.

Every smart contract is required to have a special `dispatcher` method. The `dispatcher` method is the only method of a smart contract that can be called by other smart contracts and has a predefined `methodID`: `(ApplicationID|0)`.

> *The Argennon Virtual Machine does not have internal or external methods. All methods are considered internal except the `dispatcher` method, which is the only external method of any smart contract. This design enables the AVM to execute smart contracts with* RESTful *APIs more efficiently.*

Instructions that modify the code area can only be run in privileged mode, which means they can only be executed by the *root* smart contract. As a result, the access of smart contracts to the code area is essentially read-only.

Constant Area

Every smart contract has a single constant area which is stored in the AVM code area as a special type of non-executable method. The constant area of a smart contract contains user defined constants.

> *Instructions that modify the constant area can only be run in privileged mode.*

Local Frame

A local frame is used to store methods parameters and local variables. A new frame is created each time a method is invoked, and it is destroyed when its method invocation completes, whether the completion is normal or abrupt.

Operand Stack

Every time a local frame is created, a corresponding empty last-in-first-out (LIFO) stack is created too. The Argennon Virtual Machine is a stack machine and its instructions take operands from the operand stack, operate on them, and push the result back onto the operand stack. An operand stack is destroyed when its owner method completes, whether that completion is normal or abrupt.

Heap

The heap of the Argennon Virtual Machine is a persistent memory area which stores *memory chunks*. A Memory chunk is a byte addressable piece of continuous memory which has a separate address space starting from 0. Each chunk has a fixed size and different chunks need not be equally sized. Chunks are stored in the heap area and every chunk is assigned a unique identifier: `chunkID`. As a result, the address of every memory location inside the heap area can be considered as a pair: `(chunkID, offset)`.

A smart contract can read any chunk stored in the heap by having its `chunkID`. However, it can only modify those chunks that was created by itself. In other words, every memory chunk in the AVM heap has an owner. Only the owner can modify a chunk while anyone can read that chunk. The owner of a chunk can be easily determined by the `applicationID` part of the chunk identifier.

> *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. This design eliminates the need for implementing* trivial *getters.*

The AVM Heap is able to save snapshots of its state and later restore them. This will enable the Argennon Virtual Machine to have state reversion capability. The snapshot management of the heap area is done by the following functions:

- `Save()` saves a snapshot of the current state of the heap and pushes it onto the snapshot stack.
- `Restore()` pops the snapshot stored on the top of the snapshot stack and restores it. The current state of the heap will be lost.
- `Discard()` pops the snapshot stored on the top of the snapshot stack and discards it. The current state of the heap will not change.

> *These functions are internal functions of the AVM. They are not instructions, and can not be called by smart contracts.*

1.6 Instruction Set Overview

An Argennon Virtual Machine instruction consists of a **one-byte** opcode specifying the operation to be performed, followed by zero or more operands supplying arguments or data that are used by the operation. The number and size of the operands are determined solely by the opcode.

1.6.1 Method Invocation

The Argennon Virtual Machine has four types of method invocation instruction:

- `invoke_internal`: invokes a method without changing the context of method execution. In other words, the invoked method can only modify the state of the current smart contract. A smart contract can invoke any method by `invoke_internal` instruction, even if that method is an internal method of another smart contract. Since the invoked method will always be executed in the context of the invoking smart contract, a smart contract will not be able to modify another smart contract state by this instruction. This instruction facilitates code reuse and the usage of libraries. If the invoked method completes abruptly, that will cause the invoker method to complete abruptly too.

- `invoke_dispatcher`: invokes the `dispatcher` method of a smart contract and changes the context of method execution to that smart contract. If the invoked method completes abruptly, that will cause the invoker method to complete abruptly too.

- `i_invoke_dispatcher, i_invoke_internal`: independently invokes a method. These instructions are similar to their `invoke` counterparts, with the difference that when the invoked method completes abruptly, the invoker method can continue execution. When a method invoked by an `i_invoke` instruction completes abruptly, all changes made by the invoked method to the AVM state will be reverted and the instruction will act like a jump to the specified error handling code.

- `invoke_native`: invokes a method that is not hosted by the Argennon Virtual Machine. By this instruction, high performance native methods of the hosting machine, such as GPU accelerated operations, could become available to AVM smart contracts. This instruction will not modify the AVM state.

- `invoke_later`: invokes the `dispatcher` method of another smart contract by creating a new call stack. The created call stack is added at the end of the call stack queue, and the context of method execution will change after the execution of that call stack starts. This instruction does not modify the `pc` register and the execution of the invoker will continue after this instruction. As a result, `invoke_later` can not return any values to the caller and the return value of the called method (if any) will be discarded. If the invoked method completes abruptly, it will cause **all the call stack queue to complete abruptly.**

The `invoke_later` instruction can be used to call a smart contract while its reentracy lock is active. Because the invocation uses a separate call stack, it will not cause the lock to throw an exception.

Each time a method is invoked a new local frame and operand stack is created. If the method was invoked by an `i_invoke` instruction, the `Save()` function of the AVM heap is called before the execution of the method starts.

The Argennon Virtual Machine uses local frames to pass parameters on method invocation. On method invocation, any parameters are passed in consecutive local variables stored in the method's local frame starting from address 0. The invoker of a method writes the parameters in the local frame of the invoked method using `arg` instructions.

The AVM does not natively support polymorphism and virtual methods. A compiler could easily generates appropriate code for implementing these features.

1.6.2 Exceptions

An exception is thrown programmatically using the `athrow` instruction. Exceptions can also be thrown by various Argennon Virtual Machine instructions if they detect an abnormal condition.

Any AVM instruction may throw an exception. This happens when the execution cost of a transaction has exceeded the declared execution cost of the transaction.

1.6.3 Method Invocation Completion

A method invocation completes normally if that invocation does not cause an exception to be thrown, either directly from the AVM or as a result of executing an explicit throw statement. If the invocation of the current method completes normally and the invocation was not made by an `invoke_later` instruction, then a value may be returned to the invoking method. This occurs when the invoked method executes one of the return instructions, the choice of which must be appropriate for the type of the value being returned (if any). Execution then continues normally in the invoking method's local frame with the returned value (if any) pushed onto the operand stack. If the method was invoked by an `invoke_later` instruction, the execution of the next call stack (if any) begins and the returned value is discarded. When a method invocation that was made by an `i_invoke` instruction completes normally, the `Heap.Discard()` function is called as a part of the return instruction.

A method invocation completes abruptly when an exception is thrown by one its instructions. The `invoke_internal` and `invoke_dispatcher` instructions always throw an exception when their invoked method completes abruptly. A method invocation that completes abruptly never returns a value.

When a method completes normally, the call stack is used to restore the state of the invoker, including its local frame and operand stack, with the `pc` register appropriately restored and incremented to skip past the method invocation instruction. If the current call stack is empty the next call stack in the call stack queue will be used for continuing the execution, and when there are no call stacks left in the queue the current execution session ends.

When a method completes abruptly, the call stack is used to restore the `pc` register of the invoker. If the pc register points to an `invoke` instruction, the current method completes abruptly as well. Otherwise, if the `pc` register is pointing to an `i_invoke` instruction, the `Heap.Restore()` function is called to restore the state of the AVM heap. Then, the local frame and operand stack of the invoker is restored, and execution continues after branching to the error handling code specified by the instruction operand.

When an invocation which was made by an `invoke_later` instruction completes abruptly, the heap snapshot that was saved before the start of the execution of the first call stack is restored. This means that all changes made by the execution session to the AVM heap will be completely reverted.

1.6.4 Reentrancy Protection

The Argennon Virtual Machine provides low level reentrancy protection by defining a dedicated instruction: `enter`. When a smart contract executes the `enter` instruction, it can not execute that instruction again as long as the method executing that instruction completes (abruptly or normally). If a smart contract tries to execute the `enter` instruction, while the method executing that instruction for the first time has not yet completed, an exception will be thrown.

Smart contracts can completely prevent reentrancy by executing `enter` instruction as the first instruction of their `dispatcher` method.

1.6.5 Heap Allocation and Deallocation

Heap allocation and deallocation in the Argennon Virtual Machine is done by the following instructions:

- `alloc`: creates a new memory chunk of the specified size in the heap. the identifier of the new chunk will be the concatenation of the `applicationID` of the creator of the chunk and the `id` operand of the instruction. The `id` operand must be the concatenation of an `accountID` and a prefix code generated by the *local* prefix tree: `id = accountID|localID`. If the identifier is not valid or if it already exists in the heap a catchable exception will be thrown.

- `delete`: deletes the chunk that its identifier is the concatenation of the current smart contract `applicationID` and the specified `id` operand. If the identifier is not valid or if the chunk was not found a catchable exception will be thrown.

When a smart contract allocates a new memory chunk, the identifier of the new chunk is not generated by the AVM, instead the smart contract can choose an identifier itself. This is a very important feature of the AVM heap, which allows smart contracts to use the AVM heap as a dictionary (map) data structure. Since the `chunkID` is a prefix code, any smart contract has its own identifier space and a smart contract can easily generate unique identifiers for its chunks.

The AVM requires the chosen identifier to be a prefix code made by concatenating a code compatible with the *accounts* prefix tree to a code compatible with the *local* prefix tree. This requirement enables AVM to detect invalid identifiers, while smart contracts can easily associate data with account identifiers. At the same time, A smart contract is able to create maps with costume keys by appending local identifiers to the zero `accountID`. Zero `accountID` does not correspond to a valid account in the Argennon blockchain.

1.7 Authorizing Operations

In blockchain applications, we usually need to authorize certain operations. For example, for sending an asset from a user to another user, first we need to make sure that the sender has authorized this operation. The Argennon virtual machine has no built-in mechanism for authorizing operations, but it provides a rich set of cryptographic instructions for validating signatures and cryptographic entities. By using these instructions and passing cryptographic signatures as parameters to methods, a programmer can implement the required logic for authorizing any operation.

> *The Argennon virtual machine has no instructions for issuing cryptographic signatures.*

In addition to signatures, a method can verify its invoker by using `get_parent` instruction. This instruction gets the `applicationID` of the smart contract that is one level deeper than the current smart contract in the call stack or if there is none, the smart contract that created the current call stack. In other words, it returns the `applicationID` of the smart contract that has invoked or spawned the current smart contract.

1.8 The AVM Standard Library

As we explained in Section 1.6.1 by using `invoke_internal` instruction, a smart contract can invoke methods of another smart contract in its own context. AVM smart contracts use this instruction to invoke methods of the AVM standard library. Methods of the AVM standard library are stored with the `applicationID` zero in the AVM method area. This implies that the AVM standard library is actually a part of the root smart contract.

In Argennon, the root smart contract is an updatable smart contract, which can be updated by the Argennon consensus protocol. This means that bugs or security vulnerabilities in the AVM standard library could be quickly patched and smart contracts could benefit from bugfixes and improvements of the AVM 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 Argennon standard library.

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

Chapter 2

The Argon Language

2.1 Introduction

The Argon programming language is a class-based, object-oriented language designed for writing Argennon's 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 can 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`.

2.2 Features Overview

2.2.1 Access Level Modifiers

Access level modifiers determine whether other classes can use a particular field or invoke a particular method or if a method can be invoked externally by other smart contracts.

	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...
}
```

2.2.2 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.

Chapter 3

Persistence Layer

The Argennon Virtual Machine has two persistent memory areas: *method area*, and *heap*. Method area stores bytecodes of methods[1], and heap stores memory chunks. Both of these data elements, bytecodes and chunks, can be considered as continuous pieces of byte addressable memory. Throughout this chapter, we shall call these data elements *objects*.

3.1 Storage Pages

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

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

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

3.2 Zero-knowledge Databases

Pages of the AVM storage are persisted using updatable zero-knowledge elementary databases (ZK-EDB). Argennon has three zero-knowledge databases: *staking* database, which stores all the data that is associated with the Argennon consensus protocol.

[1] also it stores constant area blocks.

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

[3] and probably read.

method database, which stores the AVM method area, and *heap* database, which stores the AVM heap. The commitment of these three ZK-EDBs are included in every block of the Argennon blockchain.

We consider the following properties for a ZK-EDB:

- The ZK-EDB contains a mapping from a set of keys to a set of values.
- Every state of the database has a commitment C.
- The ZK-EDB 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 ZK-EDB.
- Having π and C a user can compute the commitment C' for the database in which D' is associated with x instead of D.

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

The commitments of the AVM ZK-EDBs are affected by the way data objects are clustered. Therefore, the Argennon clustering algorithm has to be a part of the consensus protocol.

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

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

3.2.1 Vector Commitments

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

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

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

not yet written...

3.3 Object Clustering Algorithm

not yet written...

Chapter 4

Networking Layer

4.1 Normal Mode

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

not yet written...

4.2 Censorship Resilient Mode

not yet written...

Chapter 5

The Argennon Blockchain

5.1 Applications

An Argennon application or a smart contract is a collection of method bytecodes and heap chunks that are stored in the AVM storage, identified by a unique application identifier. An application identifier, applicationID, is a unique prefix code generated by the *applications* prefix tree. (See Section 1.3.)

An application identifier can be considered as the address of an application and has the following standard symbolic representation:

```
<application-id> ::= "["<hex-prefix-code>"]"
<hex-prefix-code> ::= "0x"<hex-num>"."<hex-prefix-code> | "0x"<hex-num>
```

where <hex-num> is a hexadecimal number between 0 and 255, using lower case letters [a-f] for showing digits greater than 9.

For example [0x24.0xff.0xda], [0x0] and [0x3.0xa0.0x0.0x0], are valid application addresses.

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

5.1.1 The Root Application

The root application or the root smart contract, with applicationID=[0x0], is a privileged smart contract responsible for installation/uninstallation of other smart contracts. The Argennon root smart contract performs three main operations:

- Installation of AVM smart contracts and determining the update policy of a smart contract: if the contract is updatable or not, which accounts can update or uninstall the contract, and so on.

- Removing AVM smart contracts.

- Updating an AVM smart contract (if allowed).

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

5.1.2 The ARG Application

The ARG application or the ARG smart contract, with `applicationID=[0x1]`, controls the ARG token, the main currency of the Argennon blockchain. This smart contract also manages a database of public keys and handles signature verification.

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

5.2 Accounts

Argennon accounts are entities defined inside the ARG smart contract. Every Argennon account is uniquely identified by a prefix code generated using *accounts* prefix tree. (See Section 1.3) An account identifier can be considered as the address of an account and has the following standard symbolic representation:

```
<account-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]` and `[1.0.0.0.0]`, are valid standard symbolic representations of account addresses.

A new account can be created by invoking `createAccount` method from the Argennon 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,[1] and can not be already registered in the system.

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

> *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.*

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

5.3 Transactions

Every Argennon transaction consists of two `i_invoke_dispatcher` instructions, the first instruction always transfers the proposed fee of the transaction in ARGs from a sender account to the fee sink accounts, and the second performs the requested operation. If the first instruction fails, the transaction will not be added to the Argennon blockchain.

Users interact with AVM smart contracts using the second `i_invoke_dispatcher` instruction of a transaction. Transferring all assets, including ARG, is done by that instruction.

The `i_invoke_dispatcher` instruction invokes the `dispatcher` method of a smart contract. Every AVM smart contract has a `dispatcher` method, and the Argennon protocol requires the `dispatcher` method to accept an HTTP request as its argument and return an HTTP response.

As a result, every Argennon transaction contains two HTTP requests. In addition to these HTTP requests, the transaction is required to contain a resource declaration object, specifying the maximum amount of resources it needs for execution.

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

5.3.1 Resource Declaration

Every Argennon transaction is required to specify a cap for all the resources it needs. This includes memory, network and processor related resources. When a transaction reaches one of its pre-declared resource caps, executing any AVM instruction which uses that resource, will result in an AVM exception.

As we know, every Argennon transaction consists of two `i_invoke_dipatcher` instructions. The first instruction is always considered as a *free* instruction and resources spent during its execution session will not be counted. As a result, only the second instruction could fail due to exceeding resource limits.

The Argennon protocol defines an execution cost for every AVM instruction, reflecting the amount of resources its emulation needs. Every transaction is required to specify two maximum execution costs: `maxInternalCost` and `maxExternalCost`. The *external* execution cost of a transaction is the **overall** cost of its `invoke_dispatcher` and `invoke_later` instructions,[2] and the remaining execution cost will be considered as the *internal* cost. If a transaction reaches one of its maximum costs, executing any instruction which has that type of cost, will throw an AVM exception.

[2] By overall cost, we mean the execution cost needed for reaching the next instruction.

When a transaction reaches its `maxExternalCost`, it can still keep executing its own code, while it can not call other smart contracts. This way the execution cost of a smart contract is completely decoupled from the smart contract it calls, and a malicious contract can not make its invoker fail unconditionally by using infinite loops.

Also, Argennon transactions are required to specify what heap or code area addresses they will access. This will enable validators to parallelize transaction validation as we will see in Section 5.7. A transaction that tries to access a memory location that is not in its access list, will be rejected. Users could use off-chain *smart contract oracles* to predict the list of memory locations their transactions need.

A smart contract oracle is a full AVM emulator that keeps a full local copy of the AVM storage and can emulate AVM execution without accessing a ZK-EDB server. Smart contract oracles can be used for reporting useful information about Argennon transactions such as accessed AVM heap or code area locations, exact amount of execution cost, and so on.

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

- Maximum internal execution cost
- Maximum external execution cost
- A list of heap/code-area locations for reading
- A list of heap locations for writing
- A list of heap chunks it will deallocate
- A list of methods it will delete (if any)
- Number and size of heap chunks it will allocate
- Number and size of method bytecodes it will allocate (if any)

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

5.3.2 Authorization

Argennon transactions do not have a sender. The authorization of the requested operation is always done by checking the digital signatures that are provided as a part of the HTTP request to the `dispatcher` method.

While every block of the Argennon blockchain stores the commitment of the transaction list, Argennon does not enforce storage of the transaction history. To be able to detect replay attacks, we require every signature that a user creates to have a nonce.

This nonce consists of the issuance round of the signature and a sequence number: `(issuance, sequence)`. When a user creates more than one signature in a round, he must sequence his signatures starting from 0 (i.e. the sequence number restarts from 0 in every round). We define a maximum lifetime for signatures, so a signature is invalid if `currentRound - issuance > maxLifeTime` or if a signature of the same user with a bigger or equal nonce is already used (i.e. is recorded in the blockchain). A nonce is bigger than another nonce if it has an older issuance. If two nonces have an equal issuance, the nonce with the bigger sequence number will be considered bigger.

To be able to detect invalid signatures, we keep the maximum nonce of used digital signatures per user. This information is stored in the ARG smart contract and when the difference between `issuance` component of the nonce and the current round becomes bigger than the maximum allowed lifetime of a signature, it can be safely deleted.[3]

5.3.3 Transaction Fee

Every Argennon transaction is required to pay two types of fees: execution fee, which is paid for executing the transaction, and storage fee, which is paid for the amount of storage the transaction allocates.

A transaction pays its fees by providing digital signatures of one or more accounts, authorizing the transfer of the amount of fee in ARG from one or more accounts to the fee sink accounts, and the fee is transferred by the first `i_invoke_dispatcher` instruction of the transaction.

> *An Argennon transaction always pays all of its proposed fee, no matter how much of its predefined resources were not used in the final emulation. This will incentivize users to report the resource usage of their transactions more accurately.*

5.4 Blocks

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

A block of the Argennon blockchain contains the following information:

[3] in some conventional blockchains, the nonce data can never be deleted, even if the account has zero balance and is no longer used.

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

5.4.1 Block Validation

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

Interestingly, conditional block validation of multiple blocks can be done in parallel. If a node has enough bandwidth and computational resources, it can conditionally verify any number of blocks from a previously created blockchain simultaneously and in parallel. As we will see in Section 5.5.2, this property plays an important role in the Argennon consensus protocol.

To some extent, conditional validation of a single block could be parallelized as well. Many transactions in a block are actually independent and the order of their execution does not matter. These transactions can be safely validated in parallel. Section 5.7 further develops this concept.

5.4.2 Block Certificate

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

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

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

- An efficiently computable *non-degenerate* pairing $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$ in groups $\mathbb{G}_0$, $\mathbb{G}_1$ and $\mathbb{G}_T$ of prime order q. We let g_0 and g_1 be generators of $\mathbb{G}_0$ and $\mathbb{G}_1$ respectively.
- A hash function $H_0 : \mathcal{M} \to \mathbb{G}_0$, where $\mathcal{M}$ is the message space. The hash function will be treated as a random oracle.

The BLS signature scheme is defined as follows:

- **KeyGen**(): choose a random α from $\mathbb{Z}_q$ and set $h \leftarrow g_1^\alpha \in \mathbb{G}_1$. output $pk := (h)$ and $sk := (\alpha)$.
- **Sign**(sk, m): output $\sigma \leftarrow H_0(m)^\alpha \in \mathbb{G}_0$. The signature σ is a *single* group element.
- **Verify**(pk, m, σ): if $e(g_1, \sigma) = e\big(pk,\ H_0(m)\big)$ then output "accept", otherwise output "reject".

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

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

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

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

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

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

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

To defend against *rogue public key* attacks, Argennon uses Prove Knowledge of the Secret Key (KOSK) scheme. As we explained in Section 5.2, when an account is created its public keys need to be registered in the ARG smart contract. Therefore, the KOSK scheme can be easily implemented in Argennon.

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

Argennon block certificates constitute an ordered sequence based on the order of blocks they certify. If we show the i-th certificate[4] of committee C with $cert_i$, and the set of signers with S_i, then the block certificate $cert_i$ can be considered as a tuple:

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

[4]note that the i-th certificate is not necessarily the certificate of the i-th block.

where σ_i is the aggregate signature issued by S_i.

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

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

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

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

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

When an Argennon account is created, both its pk and pk^{-1} is registered in the ARG smart contract, so the inverse of any aggregate public key can be easily computed.[5]

5.5 Consensus

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

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

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

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

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

5.5.1 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

[5] since the group operator of a cyclic group is commutative, we have $(ab)^{-1} = a^{-1}b^{-1}$.

(ADAGs[6]). At the start of the Argennon mainnet, this committee will have five members, and later its size could be changed by the ADAGs in a procedure described in Section 6.1.

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

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

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

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

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

5.5.2 Validators

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

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

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

[6] pronounced /eɪ-dagz/.

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

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

- The total stake of `online` members of the committee is higher than `minOnlineStake` fraction of the total stake of the committee. This threshold can be changed by the ADAGs, but it can never be lower than 2/3.
- All signers of the certificate have `online` status.
- The sum of stake values of the certificate signers is higher than 3/4 of the total stake of the committee members that have `online` status.

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

As we mentioned before, the block with height n needs a certificate from the committee of validators with index n modulo m. To decide about signing a block certificate, a validator checks the conditional validity[8] of the block, and if the block is valid he issues an "accept" signature. If the block is invalid, he initiates the recovery protocol. The validator will broadcast the certificate **only after** he sees the certificate of the validators of the previous block. Some validators may also require seeing a certificate from some community driven committee. An honest validator never signs a certificate for two different blocks with the same height.

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

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

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

---

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

[8]See Section 5.4.1

5.5.3 Status Blocks

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

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

If the reshuffling can not solve the problem due to low total online stake, the protocol requires the next block of the blockchain to be a special *status block*. A status block is a special block which can only contain status change transactions. The status block need to be certified by the delegates and by 2/3 of the total stake of validators. The `online`/`offline` status of validators will not be considered in the validity of the status block certificate.[9] After applying the transactions of the status block, the total online stake of all committees of validators must go higher than `minOnlineStake` threshold.

5.5.4 Signature Aggregation

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

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

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

5.5.5 The Recovery Protocol

The recovery protocol is a resilient protocol designed for recovering the Argennon blockchain from critical situations. In the terminology of the CAP theorem, the recovery protocol is designed to choose consistency over availability, and is not a protocol supposed to be

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

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

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

Emergency Forking

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

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

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

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

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

To revoke a fork block with height $b+1$, the delegates need to seal the fork by adding a special *seal block* after the fork block.[10] The seal block has the height $b+2$ and needs to be certified by the validators' committee with the index $b+2$ modulo m. The fork block is considered revoked only after the seal block is certified by the validators.

The seal block is not a normal block and validators who signed a certificate for a seal block are allowed to sign a certificate for a block with the same height and vice versa.

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

However, it should be noted that generating a block with the same parent as the seal block is considered a malicious behaviour of the reserve committee and validators will not sign such a block.

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

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

Emergency Agreement

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

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

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

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

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

Voting Phase:

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

Confirmation Phase for Proposal p:

- user c-votes p.

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

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

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

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

Initiating the Recovery Protocol

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

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

In the next step, he will sign and broadcast an **emergency fork request** message, alongside some useful metadata such as the last valid block of his blockchain and the evidence of delegates' misbehaviour.[11]

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

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

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

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

5.5.6 Estimating Stake Values

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

Unfortunately, there is a subtle problem with this approach. It is not clear that 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 mechanisms 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:

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

where:

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

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

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

where

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

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

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

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

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

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

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

where:

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

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

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

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

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

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

5.5.7 Analysis

not yet written...

5.6 Incentive mechanism

5.6.1 Certificate Rewards

When the certificate of a committee of validators is included in a block, the signers of that certificate and the delegates will be rewarded. Every validator will be rewarded equally. The delegates will be rewarded proportional to the total stake of the signers of the certificate and their reward increases if the certificate has more signers.

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 AVM storage. Rewards of every staking period depends on the amount of fees that are collected during that period.

5.6.2 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 signature in the future.

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

- Signing a certificate for a block that is not conditionally valid.
- Signing a certificate for two different blocks at the same height if none of them is a fork block or a seal block.[12]

5.6.3 Incentives for ZK-EDB Servers

The incentive mechanism for ZK-EDB servers should have the following properties:

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

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

- `ownerAddr`: the account address of the ZK-EDB 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 ZK-EDB servers.

[12] 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 a honest user could mistakenly do that.

To incentivize ZK-EDB servers, a lottery will be held every round,[13] and a predefined amount of ARGs from `dbFeeSink` account will be distributed between winners as a prize. This prize will be divided equally between all *winning tickets* of the lottery.

One ZK-EDB server could own multiple winning tickets in a round.

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

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

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

`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.

5.7 Concurrent Transaction Validation

5.7.1 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 AVM state sequentially as they appear in the ordered list. This ordering is solely chosen by the block proposer, and users should not have any assumptions about the ordering of transactions in a block.

The fact that block transactions constitute a sequential list, does not mean they can not be executed and applied to the AVM 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 AVM state by modifying either the code area or the AVM 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 AVM state.
  - An ordering of transactions of T respects G_e if for every directed edge $(u, v) \in E_e$ the transaction u comes before the transaction v in the ordering.

Having the execution DAG of a set of transactions, using Algorithm 2, we can apply the transaction set to the AVM state concurrently, using multiple processor, while we can be sure that the resulted AVM state will always be the same no matter how many processor we have used.

Algorithm 2: Executing DAG transactions

Data: The execution dag $G_e = (V, E)$ of transaction set T
Result: The state of the AVM 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 3 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 paral-

Algorithm 3: Constructing an execution DAG

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

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

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

lelization a block can contain more transactions, a proposer is incentivized enough to find a good execution DAG for transactions.

5.7.2 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 just need to make sure u does not see any changes v is making in AVM 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.

5.7.3 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 4 implements a concurrent counter which returns an error when the value of the counter becomes negative.

Algorithm 4: Concurrent counter

Function `GetValue(Counter)`
- $s \leftarrow 0$
- `Lock.Acquire()`
- **for** $i \leftarrow 0$ **to** $\mathsf{Counter}.size - 1$ **do**
  - $s \leftarrow s + \mathsf{Counter}.cell[i]$
- **end**
- `Lock.Release()`
- **return** s

Function `Increment(Counter,` *value, seed*`)`
- $i \leftarrow seed \bmod \mathsf{Counter}.size$
- `AtomicIncrement(Counter.`$cell[i]$, *value*`)`

Function `Decrement(Counter,` *value, seed, attempt*`)`
- **if** $attempt = \mathsf{Counter}.size$ **then**
  - restore Counter by adding back the subtracted value
  - **return** *Error*
- **end**
- $i \leftarrow seed \bmod \mathsf{Counter}.size$
- $i \leftarrow (i + attempt) \bmod \mathsf{Counter}.size$
- **if** $\mathsf{Counter}.cell[i] \geq value$ **then**
  - `AtomicDecrement(Counter.`$cell[i]$, *value*`)`
- **else**
  - $r \leftarrow value - \mathsf{Counter}.cell[i]$
  - `AtomicSet(Counter.`$cell[i]$, 0`)`
  - `Decrement(Counter,` r, *seed*, $attempt + 1$`)`
- **end**

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 AVM standard library, and any smart contract can use this data structure for storing the balance record of highly active accounts.

Chapter 6

Governance

6.1 ADAGs

The Argennon Decentralized Autonomous Governance system (ADAGs)

not yet written...