Contents

Chapter 1

The Argennon Virtual Machine

1.1 Introduction

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

Execution sessions are separate sessions of executing smart contact's code by the Argennon Virtual Machine. These sessions are usually correspondent to the concept of a transaction in a blockchain. (See Section 1.7)

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, `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 Argennon Virtual Machine has its specific word size, and the addressable memory areas are byte addressable.

1.3 Identifiers

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

All these identifiers are *prefix codes*, and hence can be represented by *prefix trees*[1]. The AVM 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 in base β:

$$< A^{(1)}, A^{(2)}, A^{(3)}, \cdots >$$

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 data. Assume that we have a string of digits which are 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 extract the prefixed identifier uniquely. We can apply this algorithm multiple times to a sequence to extract a composite identifier, for example `chunkID`, from a sequence.

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] Also called a trie.

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
```

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 state, and continuing the execution after the method invocation completes. This information is represented by a `CallInfo` struct:

```
CallInfo {
    applicationID,
    methodID,
    pc,
    canModify,
    localFrame,
    operandStack
}
```

The `applicationID` field is the unique identifier that the AVM assigns to every smart contract, `methodID` is the unique identifier of a method's bytecode and `canModify` is a flag indicating whether a method is allowed to execute state modifying instructions or not.

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's 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.

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 several kinds of constants, ranging from user defined constants to method address tables. A method address table stores the list of methods of a smart contract and their access type. The access type of method can be either `external` or `internal`. Only external methods can be invoked by `invoke_external` instruction.

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 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. In addition, for modifying a chunk the `canModify` flag of the current method's `callInfo` struct must be true. 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. Although 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, `applicationID` and `canModify` fields of the invoked method in the call stack will be the same as the invoker. 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.

- `invoke_external`: invokes a method and changes the context of method execution to another smart contract. Only external methods of another smart contract can be called by this instruction. For changing the context of method execution this instruction sets the `applicationID` field of the invoked method to the `applicationID` of the called smart contract. Then it searches the call stack and if the `applicationID` of the called smart contract is found in the current call stack it sets `canModify` to `false`, otherwise `canModify` is set to `true`. As a result, if the called smart contract is already called, the invoked method will not be able to modify the heap area.
- `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 could become available to AVM smart contracts. This instruction will not modify the AVM state.
- `spawn`: spawns a new method invocation. Spawning a method invocation is the invocation of a method with changing the context of method execution and creating a new call stack. Only external methods can be spawned. The created call stack is added at the end of the call stack queue and the `canModify` field of the pushed `CallInfo` struct will always be set to `true`. This instruction does not modify the `pc` register and the execution of the caller will continue after this instruction. As a result, `spawn` can not return any value to the caller and the return value of the called method (if any) will be added to the transaction output. If the invoked method completes abruptly, it will cause all the call stack queue to complete abruptly.

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

Each time a method is invoked the `Save()` function of the AVM heap is called, then a new local frame and operand stack is created. 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.

For AVM smart contracts reentrancy *can only happen in read-only mode and is totally safe. At the same time call-back patterns can be easily implemented by using* ***`spawn`*** *instruction.*

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. Some exceptions are not catchable and will always abort the method invocation.

By using the `athrow` *instruction properly, a programmer can make any method act like an atomic operation.*

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 made by an `invoke` 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 a `spawn` instruction, the execution of the next call stack (if any) begins and the returned value is appended at the end of the transaction output. When a method invocation completes normally, the `Heap.Discard()` function is called as a part of the return instruction.

A method invocation completes abruptly if an exception is thrown and is not caught by the current method. A method invocation that completes abruptly never returns a value.

When a method completes, whether normally or abruptly, 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.

A thrown exception causes methods in the call stack to complete *abruptly* one by one, as long as the `pc` register is not pointing to a `catch` instruction. The `catch` instruction acts like a branch instruction that branches only if an exception is caught. When an exception is thrown Algorithm 2 is used to restore the state of the AVM heap.

A closer investigation of Algorithm 2 shows that when an invocation which was made by a `spawn` instruction completes abruptly, the heap snapshot that was saved before the start of the execution of the first call stack of the execution session is restored. Since the root smart contract calls `Heap.Save()` function before spawning the requested method, this means that all changes made to the AVM heap during the execution session will be restored, but the changes made by the root smart contract including the transferring of the transaction fee will not be restored. See Section 1.7 for more details.

Algorithm 2: Abrupt method completion

```
while CallStack is not empty do
    CallStack.pop()
    if CallStack is not empty and CallStack[head].pc → catch then
        Heap.Restore()
    else
        Heap.Discard()
    end
end
Heap.Restore()
```

1.6.4 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 specifies 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. The specified identifier must be 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` to the specified `id`. 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. Because 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 will enable 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 `accountID` zero, which does not correspond to a valid account in the Argennon.

1.7 Execution Sessions

The execution of smart contracts code in the AVM is done in execution sessions. Every execution session in the Argennon Virtual Machine consists of a single method invocation

from the *root* smart contract and a list of resource allocation requests. These allocation requests always include a request for allocating some amount of execution time and a request for accessing some memory locations for reading or writing. If after the start an execution session tries to violate its allocated resources, an **uncatchable** exception will be thrown by the AVM.

> *The AVM does not have a* `calldata` *memory area. The arguments of the root smart contract method will be copied from the transaction to the local frame of the invoked method, exactly like a normal method invocation.*

The root smart contract with the `applicationID` of zero, is a special smart contract in the Argennon Virtual Machine. Its code is immutable and is a part of the AVM implementation. The root smart contract is always run in the privileged mode, and every execution session starts with a call to a method of the root smart contract. This method always transfers some amount of fee from an account ot the `feeSink` account and then it performs the requested operation.

> *Argennon transactions do not have a sender, and the payer of the transaction fee could be any account who has provided the required digital signature.*

If the requested operation is a method invocation, the root smart contract will call the `Heap.Save()` function once, and then it will invoke the requested method using `spawn` instruction. This means that the invoked method will have a separate call stack and if it completes abruptly the changes made by the root smart contract to the state will not be reverted.

1.8 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.

1.9 The AVM Standard Library

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 Programs 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 and a single contract. Every Argon package corresponds to at max one AVM smart contract.

2.2 Features Overview

2.2.1 Static Classes

In Argon a static class is an uninstantiable class which all its members and methods are static. Every static class can define a special method `initialize` which can only be called in the `initialize` method of the single contract of the package the static class is defined in.

A Static class is like a normal class which can only have one instance. So like normal classes, static classes can be a super class or a subclass of other static classes and can implement interfaces.

2.2.2 Contracts

Every package of an Argon program can have one or zero contracts. A contact is a special static class which is allowed to define methods with `external` visibility. When an Argon package is deployed as an AVM smart contract, these external methods will define the contract's public interface.

2.2.3 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	World
private	yes	no	no	no	no
protected	yes	no	yes	no	no
package	yes	yes	yes	no	no
public	yes	yes	yes	yes	no
external	no	no	no	no	yes

2.2.4 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.

```
// A contract is a static class and all its methods are automatically static.
contract MirrorToken {
    private SimpleToken token;
    private 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. It can not be called after that.
    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);
    }

    external void transfer(account sender, account recipient,
        double amount, signature sig) {
        // Checks if the signature is issued by the sender and verifies the calling of this function
        // with the current parameters.
        sig.verifyCallBy(sender);
        token.transfer(sender, recipient, amount);
        reflection.transfer(recipient, sender, Math.sqrt(amount));
    }

    external double balanceOf(account user) { return token.balanceOf(user); }

    external double balanceOfReflection(account user) { return reflection.balanceOf(user); }
}

public static class Math {
    public double sqrt(double x) {
        return // calculate square root of x
    }
}

package class SimpleToken {
    private map(account -> double) balances;

    // The visibility of a member without an access modifier will be the same as its defining
    // class so 'SimpleToken' constructor has package visibility.
    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...
}
```

Chapter 3

The Argennon Blockchain

3.1 Persistence

For implementing the persistence layer of the AVM, we assume that we have access to an updatable zero-knowledge elementary database (ZK-EDB) with the following properties:

- 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 $(D, p) = get(x)$, where x is a key and D is the associated data with x, and p is a proof.
- A user can use C and p to 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 p and C a user can compute the commitment C' for the database in which D' is associated with x instead of D.

We use a ZK-EDB for storing the AVM heap. We include the commitment of the current state of this DB in every block of the Argennon blockchain, so ZK-EDB servers need not be trusted servers.

Every page of AVM heap will be stored with a key of the form: `applicationID|pageIndex` (the `|` operator concatenates two numbers). Nodes do not keep a full copy of the AVM heap and for validating block certificates or emulating the AVM (i.e. validating transactions) they need to connect to a ZK-EDB and retrieve the required pages of AVM heap. For better performance, nodes keep a cache of heap pages to reduce the amount of ZK-EDB access.

We also use a ZK-EDB for storing the code area of each segment, and we include the commitment of this DB in every block. Every code area will be divided into blocks and every block will be stored in the DB with `applicationID|blockID` as its key. Like heap pages, nodes keep a cache of code area blocks.

Unlike heap pages, the AVM is not aware of different blocks of code area.

3.2 Transactions

Argennon has four types of transaction:

- `avmCall` essentially is an `invoke_external` instruction that invokes a method from an AVM smart contract. Users interact with AVM smart contracts using these transactions. Transferring all assets, including ARGs, is done by these transactions.
- `installApp` installs an AVM smart contract and determines the update policy of the smart contract: if the contract is updatable or not, which accounts can update or uninstall the contract, and so on.
- `unInstallApp` removes an AVM smart contract.
- `updateApp` updates an AVM smart contract.

All types of Argennon transactions contain an `invoke_external` instruction which calls a special method from ARG smart contract that transfers the proposed fee of the transaction in ARGs from a sender account to the fee sink accounts.

Every transaction is required to exactly specify what heap locations or code area addresses it will access. This enables validators to start retrieving the required memory blocks from available ZK-EDB servers as soon as they see a transaction, and they won't need to wait for receiving the new proposed block. A transaction that tries to access a memory location that is not included in its access lists, will be rejected. Users could use smart contract oracles to predict the list of memory blocks their transactions need. See Section 3.7 for more details.

3.3 Blockchain

Every block of the Argennon blockchain corresponds to a set of transactions. We store the commitment of this transaction set in every block, but we don't keep the set itself. 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. When the difference between `issuance` component of this nonce and the current round becomes bigger than the maximum allowed lifetime of a signature, this information can be safely deleted. **As a result, we will not have the problem of "un-removable empty accounts" like Ethereum.**

The only information that Argennon nodes are required to store is **the most recent block** of the Argennon blockchain. Every block of the Argennon blockchain contains the following information:

Block
commitment to the ZK-EDB storing heap pages
commitment to the ZK-EDB storing code areas
commitment to the set of transactions
previous block hash
random seed

For confirming a new block, nodes that are not validators only need to verify the block certificate. For verifying a block certificate, a node needs to know the ARG balances of validators, but it doesn't need to emulate the AVM execution.

On the other hand, nodes that are chosen to be validators, for validating a new block, need to emulate the execution of the Argennon virtual machine. To do so, first they retrieve all heap pages and code area blocks they need from available ZK-EDBs. Then, they emulate the execution of AVM instructions and validate all the transactions included in the new block. This will modify some pages of the AVM memory, so they update the ZK-EDB commitments based on the modified pages and verify the commitments included in the new block. Validators also calculate and verify the commitment to the new block's transaction set.

Validators do not need to write the modified pages back to ZK-EDB servers. ZK-EDB servers will receive the new block, and they will update their database by emulating the AVM execution.

3.4 Consensus

It will be a randomized proof of stake which is under development....

3.4.1 Estimating A User's Stake

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, for estimating a user's stake, we use the amount of native system tokens the user is holding. Unfortunately, one problem with this approach is that a strong attacker may be able to obtain a considerable amount of system tokens, for example by borrowing from a DEFI application, and use this stake to attack the system.

To mitigate this problem, for calculating a user's stake at time step t, instead of using the raw ARG balance, we use the minimum of a *trust value* the system has calculated for the user and the user's ARG balance:

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

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.
- $Trust_{u,t}$ is an estimated trust value for user u at time step t.

The agreement protocol, at time step t, will use $\sum_u S_{u,t}$ to determine the required number of votes for the confirmation of a block, and we let $Trust_{u,t} = M_{u,t}$, where $M_{u,t}$ is the exponential moving average of the ARG balance of user u at time step t.

In our system 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 exponential moving average 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:

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

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.

In our system a newly created account will not have voting power for some time, no matter how high its balance is. While this is a desirable property, in case a large proportion of total system tokens are transferred to newly created accounts, it can result in too much voting power for older accounts. This may decrease the degree of decentralization in our system.

However, this situation is easily detectable by comparing the total stake of the system with the total balance of users. If after confirming a block the total stake of the system goes too low and we have:

$$\sum_u S_{u,t} < \gamma \sum_u B_{u,t}$$

The protocol will perform a *time shift* in the system: the time step of the system will be incremented for m steps while no blocks will be confirmed. This will increase the value of $M_{u,t}$ for new accounts with a non-zero balance, giving them more influence in the agreement protocol.

For calculating the value of m which determines the amount of time shift in the system, we should note that when $B_{u,t} = B_{u,t-1} = B_u$, we can derive a simple recursive rule for the stake of a user:

$$S_{u,t} = (1-\alpha)S_{u,t-1} + \alpha B_u$$

Therefore, we have:

$$\sum_u S_{u,t} = (1-\alpha)\sum_u S_{u,t-1} + \alpha \sum_u B_u$$

This equation shows that when the balance of users is not changing over time the total stake of the system is the exponential average of the total ARGs of the system. Consequently, when we shift the time for m steps, we can calculate the new total stake of the system from the following equation:

$$\sum_u S_{u,t+m} = (1-\alpha)^m \sum_u S_{u,t} + [1-(1-\alpha)^m] \sum_u B_u$$

Hence, if we want to increase the total stake of the system from $\gamma \sum_u B_u$ to $\lambda \sum_u B_u$, we can obtain m from the following formula, assuming α is small enough:

$$m = \frac{1}{\alpha} \ln\left(\frac{1-\gamma}{1-\lambda}\right)$$

3.5 Incentive mechanism

3.5.1 Transaction Fee

Every transaction in the Argennon blockchain starts with an `invoke_external` instruction which calls a special method from ARG smart contract. This method will transfer the proposed fee of the transaction in ARGs from a sender account to the fee sink accounts. Argennon has two fee sink accounts: `execFeeSink` collects execution fees and `dbFeeSink` collects fees for ZK-EDB servers. The Protocol decides how to distribute the transaction fee between these two fee sink accounts.

When a block is added to the blockchain, the proposer of that block will receive a share of the block fees. Consequently, a block proposer is always incentivized to include more transactions in his block. However, if he puts too many transactions in his block and the validation of the block becomes too difficult, some validators may not be able to validate all transactions on time. If a validator can not validate a block in the required time, he will consider the block invalid. So, when a proposed block contains too many transactions, the network may reach consensus on another block, and the proposer of that block will not receive any fees. As a result, a proposer is incentivized to use network transaction capacity optimally.

On the other hand, we believe that the proposer does not have enough incentives for optimizing the storage size of the transaction set. Therefore, we require that **the size of the transaction set of every block in bytes be lower than a certain threshold.**

Validators need to spend resources for validating transactions. When a validator starts the emulation of the AVM to validate a transaction, solely from the code he can't predict the time the execution will finish. This will give an adversary an opportunity to attack the network by broadcasting transactions that never ends. Since, validators can not finish the execution of these transactions, the network will not be able to charge the attacker any fees, and he would be able to waste validators resources for free.

To mitigate this problem, we require that every transaction specify a cap for all the resources it needs. This will include memory, network and processor related resources. Also, the protocol defines an execution cost for every AVM instruction reflecting the amount of resources its emulation needs. This will define a standard way for measuring the execution cost of any `avmCall` transaction. Every `avmCall` transaction is required to specify a maximum execution cost. If during emulation it reaches this maximum cost, the transaction will be considered failed and the network can receive the proposed fee of that transaction.

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

- Execution cost
- A list of heap/code-area locations for reading
- A list of heap locations for writing
- A list of heap pages it will deallocate

- Number and size of heap pages it will allocate

If a transaction tries to violate any of these predefined limitations, for example, if it tries to read a memory location that is not included in its reading list, it will be considered failed and the network can receive the proposed fee of that transaction.

A transaction always pays all of its proposed fee, no matter how much of its predefined resources were not used in the final emulation.

3.5.2 Incentives for ZK-EDB Servers

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

- It incentivizes storing all memory blocks, whether a heap page or a code area block, and not only those which are used more frequently.
- It incentivizes ZK-EDB servers to actively provide the required memory blocks for validators.
- Making more accounts will not provide any advantages for a ZK-EDB server.

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

- `ownerAddr`: the ARG address of the ZK-EDB.
- `receivedBlockID`: the ID of the received memory block.
- `round`: the current round number.

In a round, an honest validator never gives a receipt for an identical memory block to two different ZK-EDBs.

To incentivize ZK-EDB servers, a lottery will be held every round 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 validator in the round $r-1$ and voted for the agreed-upon block.

- The data block 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 data block for two different ZK-EDBs in the previous round.

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

```
IF random(seed|validatorPK|receivedBlockID) < winProbability THEN
    the receipt issued by validatorPK for receivedBlockID 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.
- `receivedBlockID` is the ID of the memory block that the receipt was issued for.
- `winProbability` is the probability of winning in every round.
- `seed` is the current block seed.
- `|` is a concatenation operator.

The winners of the lottery were validators one round before the lottery round.

Also, based on the current block seed, a random memory block, whether a heap page or a code area block, is selected as the challenge of the round. A ZK-EDB 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 memory block which was selected as the round challenge. This will encourage ZK-EDBs to store all memory blocks.

A possible choice for the challenge solution could be the cryptographic hash of the content of the challenge memory block combined with the ZK-EDB ARG 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. Validation and prize distribution for the winning tickets of round r will be done in the round $r+1$. This way, **the content of the challenge memory block could be kept secret during the lottery round.** Every winning ticket will get an equal share of the lottery prize.

3.5.3 Memory Allocation and De-allocation Fee

Every k round the protocol chooses a price per byte for AVM memory. When a smart contract executes a heap allocation instruction, the protocol will automatically deduce the cost of the allocated memory from the ARG address of the smart contract.

To determine the price of AVM memory, Every k round, the protocol calculates `dbFee` and `memTraffic` values. `dbFee` is the aggregate amount of collected database fees, and `memTraffic` is the total memory traffic of the system. For calculating the memory traffic of the system the protocol considers the total size of all the memory pages that were accessed for either reading or writing during a time period. These two values will be calculated for the last k rounds and the price per byte of AVM memory will be a linear function of `dbFee/memTraffic`

When a smart contract executes a heap de-allocation instruction, the protocol will refund the cost of de-allocated memory to the smart contract. Here, the current price of AVM memory does not matter and the protocol calculates the refunded amount based on the average price the smart contract had paid for that allocated memory. This will prevent smart contracts from profit taking by trading memory with the protocol.

3.6 Concurrency

3.6.1 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 3, 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 3: 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 execu-

tion 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.

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

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.

3.6.2 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.

3.6.3 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.

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
```

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.

3.6.4 Memory Chunks

In order to further increase the concurrency level of Argennon, we can divide the AVM memory into *chunks*. Each memory chunk can be persisted using a different ZK-EDB, hence having its own commitment. Then, the consensus on new values of the commitment of any chunk can be achieved by a different voting committee.

If a transaction does not modify a memory chunk and in the transaction ordering of the block it comes after any transaction which modifies that chunk, then the execution of that transaction is not needed for calculating the new commitment of the chunk. Consequently, the voting committee of that memory chunk can safely ignore such a transaction. The execution DAG of transactions can be used for finding and pruning these transactions as we see in Algorithm 6.

Algorithm 6: Pruning an execution DAG

input : An execution dag G_e and a memory chunk C_k
output: The pruned execution dag based on C_k

```
for each vertex v_0 with out degree 0 do
    RecursivePrune(v_0)
end

Function RecursivePrune(v_0)
    if Adj[v_0] = ∅ and v_0 does not modify C_k then
        remove v_0 from G_e
        for each u such that edge (u, v_0) was in G_e do
            RecursivePrune(u)
        end
    end
```

If we choose chunks in a way that most transactions only modify memory locations of one chunk, likely many transactions of a block only need to be validated by one voting committee and can be validated in parallel by different committees.

Because the voting committees are selected by random sampling, by choosing large enough samples we can make sure that having multiple voting committees will not change the security properties of the Argennon agreement protocol.

3.7 Smart Contract Oracle

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