Adamnite Protocol Implementation

The Adamnite Protocol will be a permissionless blockchain platform designed for efficiency, security, and usability. In this document, core technical details of the Adamnite Protocol are explained, for the purpose of both streamlining new contributors and providing a basis for existing developers. The protocol client, from a minimal sense (without a virtual machine or smart contracts) should include the following: a distributed ledger that tracks changes to the overall state of the network (The Adamnite Blockchain), a peer to peer network that allows nodes interacting with the distributed ledger to also interact with another, and a native currency (NITE) that is used for inter-network transactions and consensus rewards. Finally, the consensus mechanism that dictates both the behavior of the blockchain and the majority of the communication that occurs between nodes in the network should be based on the Delegated Proof of Stake Protocol (DPOS) described in the Adamnite White Paper. A technical description of these features follows:

Blockchain

The blockchain should act as a ledger of all activity occurring within the network, and should serve as the basis of further development on the network. Several key structures define the blockchain's functionality: blocks, transactions, accounts (also referred to as nodes), and chains. At its core, the blockchain is simply a decentralized database containing all changes to the overall state of the network. In the context of Adamnite (and more popular blockchains such as Bitcoin and Ethereum), the overall state is dictated mostly by internal transactions of a native currency (NITE). Furthermore, like Ethereum, the overall state of the data is account-based. Blocks are composed of transactions, which are themselves composed of several parameters: sender, receiver, amount, input, output, fee, timestamp, and message. UTXOs are centered around the input/output; they track the amount of NITE that was not spent on transaction fees. A basic functioning client should include transactions (the ability for nodes to send NITE to one another), blocks that were created and approved by witnesses (more in the consensus mechanism section), and a way for anyone interacting with the client to see the current state of the blockchain/test it out themselves.

A blockchain is also inherently dependent on cryptography. Cryptographic structures determine how the blockchain stores and processes data, separating blockchain technology from other DLTs. This is where a majority of the protocol research for Adamnite will come into play: both security and efficiency depend largely on how cryptography is implemented. Blockchains such as Bitcoin often use a variation of merkle trees to store data, with the hashes of block headers being stored as nodes (not to be confused with nodes on the blockchain) in the tree. For a basic proof of concept, a merkle tree as used by Bitcoin should be enough to store the blocks and serve as reference for looking up both blocks/transactions.

Resources
Simple Implementation in Go, with tutorial (only focus on the blocks, accounts, and transactions, as the consensus mechanism will not be Proof of Work, but Delegated Proof of Stake)
Ethereum Implementation in Go, with DPOS! (Advanced, might take a really long time to understand, but very close to what we want)
C++ Example (again, not what we want with POW, but a good example to learn)
C++ DPOS, implementation with DPOS (A good example of what we are looking for)
Merkle Trees in Go

The key before oficial mainnet development begins (which will be after the crowdsale) is to develop a proof of concept and to learn the concepts necessary to build the full-fledged blockchain platform from scratch.

#This document will continue to have additions that describe the DPOS consensus mechanism, peer-to-peer network, and an advanced section detailing the smart contracts, virtual machine, and secure compiler.

Delegated Proof of Stake Implementation

Delegated Proof of Stake (DPOS) is an alternative consensus mechanism that allows nodes to vote on specific witnesses to handle both block proposals and block validation. In essence, a DPOS Proof of Stake implementation should allow nodes to communicate via a P2P protocol at specific times, ie, when new witnesses need to be elected. In Adamnite, this is after a full round has passed (after 5 blocks have been validated). A formal specification and step by step process follows:

1. New Voting Period begins. Nodes that have declared that they want to be a witness are put in witness pool A, which is a collection[1] of all nodes that can be voted to be a witness.
2. All accounts active in the network can send a participation transaction indicating which node they want to vote for to represent them as a witness. However, the default is an automatic vote integrating the RepuStake method for DPOS chains. This essentially allows accounts to automatically stake their NITE without having to continuously send participation transactions; their stake is evenly distributed among the top *i*th percentile of nodes in the witness pool, as determined by their reputation. As defined in RepuStake, a node's reputation is essentially a quantitative score describing their past validator behavior.

[1] Specific Data Structure to store witnesses TBD

3. Nodes in witness pool A that have the highest voter score[2] will be put in witness pool B, with the size of witness pool B being less than that of witness pool A.
4. The actual numerical size of witness pool B will be mutable: it will depend on the total size of the network (the size of witness pool B will grow as the network grows).
5. A verifiable random function should then select *n (n* will be constat) nodes from witness pool B to serve as the witnesses for a particular block. The process to select the witnesses should be slightly weighed based on the node's stake and the amount of votes they received during the most recent voting session. This process will repeat for *n* blocks, at which point voting will happen again.

DPOS implementation should be done over a P2P protocol, with account's weight/stake determining their overall voting power that they have within the network. A potential witness' reputation plays a large role as well; not only does it drive how automatic stakes are distributed, but it also serves as a statistic for manual voters to consider as they choose a candidate. Voting should happen every 5 blocks through a P2P protocol, with consensus for every block being determined through a traditional POS method. Advanced cryptographic primitives for both storing consensus data and for node validation are in development; these will be released with the technical white-paper in February. Namely, the plan is to leverage Zero-Knowledge Proofs to effectively store blocks. This should be similar to the MINA blockchain, which leverages ZK-Snarks to make its overall chain extremely lightweight (only 22kb), thus allowing anyone with a modern device and an internet connection to validate the chain. This greatly reduces the barrier to entry for most nodes, and also allows the blockchain to be scalable. The specific protocol for this will be described in the technical white-paper.

For the POC, the main goal is to have a suitable work product that demonstrates these principles. The POC does not need to have built-in smart contracts, a virtual machine, or even be fully expanded. However, what it does need to do is show the technical capabilities of the team, have the ability for developers to download and stress test it, and serve as a basic representation of what the end goal is.

Virtual Machine

The Adamnite Virtual Machine (ADVM) will be a stack-based Big-Endian virtual machine (VM) that processes on-chain ADVM code. ADVM code is a stack-based byte language that defines various interactions with the Adamnite Blockchain. Smart contracts on the Adamnite Blockchain, much like other smart contract platforms, essentially function like regular accounts with pre-defined instructions. The Virtual Machine essentially processes transactions that are encoded as application transactions; these are transactions that create a specific address that

[2]

serves as the application's main interaction point on the public ledger. An application transaction can also be thought of as a transaction that defines a new on-chain smart contract. During compilation, a compiler will translate the high-level code that a developer has written defining the application transaction into low-level byte code. This low-level byte code will be defined by various OPCODES that define how the smart contract interacts with or manipulates the current state of the blockchain. These OPCODES could be as simple as adding and removing items from the stack, or something more complex like fetching block information or the current state of the overall network.

The process for creating a VM is actually quite simple. First, the hardware-related components will need to be defined: the includes the memory (for Adamnite, this will be 256-bit based), the registers (which defines where values will be stored), and the actual OPCODES that define what instructions the VM can execute.

Second, the actual program execution will need to be defined. This will primarily deal with how smart contracts are compiled.

****Work in Progress****

Special Note: Public and Private Key Generation and Signatures

For Adamnite's account based transaction model, there are two main processes. First, each account needs to have a public-private key pair. The public key is essentially the public identifier of the account; it is encoded into an public address that acts as an identifier on the blockchain and can send or receive NITE and other on-chain assets. The private key certifies that the underlying user or program actually owns the account: it is used to actually sign transactions and recover the account in case it is lost. A private key is often encoded into an easy to remember mnemonic using a number system such as base-32 that allows users to import or recover their account easily (Note: There is a key distinction between account and wallet in cryptocurrency systems. An account is an individual public-private key pairing that can own, receive, and send on-chain assets. A wallet is an interface that allows the user to directly interact with the blockchain through different accounts).

On Adamnite, the primary signature algorithm is ECDSA; this defines how signatures are generated. Like Bitcoin and Ethereum, Adamnite will use **secp256k1** as the curve; this guarantees efficiency while ensuring that the signature is secure. The actual signature is generated by multiplying (under the ECDSA rules) the hash of the transaction information by the private-key. The hashing algorithm to hash the transaction information for Adamnite will be the first half of the

SHA-512 hash. The truncation ensures that the hash can be mathematically combined with the secp256k1 curve algorithm used to generate the actual signature. A detailed description of this process can be found here. A high-level implementation is also detailed below:

1. SHA-512 hash of transaction information is created (*z*)
2. The first half of *Z* is combined with the private key through the secp256k1 algorithm as described in the document linked above.
3. Signature Pair generated: (*r,s*)

For public-private key generation, a similar process is used. A private key is just a random number picked in a cryptographically secure manner; this can be a library based on Bitcoin encryption tools, or an implementation of a cryptographically secure library. The private key is usually 256 Bits, or 35 bytes, long. The private key can be mapped to a 24 word mnemonic phrase by using the BIP-0039 English word list (see Algorand's implementation here). To generate the public-key, the private-key simply needs to be hashed: for Adamnite, we will again use SHA-512, and then hash with the SHA-512 value with RIPEMED-160. This is very similar to both Bitcoin, Ethereum, and Litecoin. Finally, this value will need to be encoded using base-58, bech-32 to produce the public-address.

A detailed process follows:

1. Private Key generation in a cryptographically random manner.
2. Public Key generation by hashing the private key: first with SHA-512, second with RIPEMED-160.
3. Public address is derived from the public key by encoding the public key in base-58,bech 32.

This page is also a good resource to learn more about the cryptographic primitives powering the 30 cryptocurrencies today.

UPDATE 2/21/22:

For the initial development of the POC, it might be better to use the SHA3 Crypto Library, for which there is extensive support, and implement the SHA-512 hash from the SHA3 family rather than the SHA2. This is already available as a package in Go, under the crypto SHA-3 package. An alternative is to use the new SHAKE 256 hashing function with an output length of 128 bytes: this theoretically should provide the same 256 bit security as SHA2-512, while not being as slow to use as SHA3-512. This is available in the same SHA-3 package.