Adamnite: A Dependable and Efficient Distributed Ledger Technology Development Platform

Archie Chaudhury
Founder, Adamnite and Co-Founder, Adamnite Labs
archchaudhury@adamnite.org

***Abstract*—Programmable Distributed Ledger Technologies have shown their potential with the recent growth of decentralized applications being governed by smart contracts. Platforms such as Ethereum, Solana, Cardano and more have established a standard model for permissionless computer systems operating in a decentralized manner. We formally define this paradigm as any programmable operating system with decentralized applications governed by immutable multi-party smart contracts recorded on a distributed ledger.**

Adamnite is an implementation of the aforementioned paradigm with features built specifically for ease of use and security. Additionally, it provides a framework for effective decentralized application development, allowing for the creation of complex multiparty smart contracts that are simultaneously powerful and safe. In this work, we produce a low-level discussion of Adamnite's design, its current implementation, and its long-term goals.

I. Introduction

Distributed Ledger Technologies are increasingly being viewed as a suitable platform for conducting financial transactions, issuing contractual rights, and developing applications. The latter has especially grown in popularity, with decentralized state-transition machines such as Ethereum essentially serving as global computers on which users can interact with interconnected and decentralized internet-based applications. While initially only used to provide and record financial instruments, these platforms have evolved to allow developers to create complex smart contracts that have come together to form the backbone of an ubiquitous decentralized application ecosystem. Use-cases such as Non Fungible Tokens, tokenized governance systems, and ledger-based loan mechanisms have metamorphized the manner in which users interact and share information with each other.

Adamnite is a protocol that aims to provide developers with a suitable and efficient platform for building such decentralized software. Like Ethereum and other multi-party smart contract development platforms, it can be modeled as a decentralized state-transition machine. However, Adamnite surpasses this common archetype through by making available both an intuitive modular programming language and efficient distributed ledger to developers, among other features, that allows to them to pursue use-cases and applications for the smart contracts paradigm that currently remain unexplored.

A. Reasoning

The main directive of the Adamnite protocol is to increase the usability of decentralized internet-based software applications by the general public, and thus encourage the adoption of distributed technology as whole. Current implementations of such applications are not only cumbersome to use, but more importantly, lack fundamental security features. Exploits on multi-party smart contracts are quite common, and due to both the legal ambiguity surrounding the space and the natural anonymity of the parties interacting with such contracts, are often difficult to rectify. This leads to a significant lack of trust of in distributed ledger applications by the general public, government legislature, and private businesses. Furthermore, the difficulty of architecting secure applications also significantly alienates otherwise capable developers, and acting as a significant roadblock in the development of decentralized applications. Through the creation of a modular and intuitive programming environment, Adamnite can serve to increase the adoption of decentralized technology.

Adamnite's proposed ecosystem will be at once similar and broadly different than other multi-party smart contract systems. While the Adamnite protocol fits the paradigm of an immutable distributed ledger supported by a state machine, its actual machine-level implementation is much more similar to that of a high-level general purpose programming language such as Python or Java. By doing so, we hope to bridge the current gap between traditional development and distributed ledger innovation , and encourage developers from the former to build applications on the latter. Finally, Adamnite hopes to increase the amount of native, decentralized, and usable applications on the internet.

B. Prior Work

Adamnite was initially defined formally in a white-paper (1) published in late 2021, which is where the core principle of a straightforward and secure programming environment combined with a traditional cryptographic distributed ledger was first espoused.

The first distributed ledger that could simultaneously process financial transactions secured by cryptography and be Byzantine Fault Tolerant was proposed by the anonymous developer Satoshi Nakamoto (2) in late 2008. Bitcoin was also the first implementation of the decentralized consensus mechanism commonly known as Nakamoto Consensus, which combines both hash-based Proof of Work with the dominance of the longest chain of timestamped blocks to serve as a computational proof. This was the first public implementation of the class of Distributed Ledger Technologies commonly known as a blockchain. Bitcoin was preceded by Hashcash (3), among others, and was followed by other currency-based systems such as Litecoin.

Bitcoin was preceded by other peer to peer currency systems such as B-Money, initially proposed by Dai. (4) While early iterations of such currency systems were never officially taken live, they were influential to the development of modern-day cryptocurrencies. For example, an early version of both distributed consensus and reputation-based punishment can be seen Dai's design for B-Money. These concepts will eventually be used in both Nakamoto Consensus and the contemporary slashing mechanisms seen in Proof of Stake (POS) currencies.

Buterin (5) proposed the first work that utilized an immutable distributed ledger with a state machine combined with a Turing Complete programming language in late 2013 for the Ethereum project, while Wood (6) provided a low-level technical specification of the same protocol in early 2014. Wood's work, which has been termed a "Yellow Paper", greatly influenced the layout of this paper. Ethereum was also the first such system to have a high-level script-like programming language (Solidity) that was accessible to all, and set a standard for distributed state systems and decentralized application development. A majority of decentralized applications today are built using Solidity, and are either defined on Ethereum or a distributed ledger dependent on Ethereum. Ethereum's virtual machine design has also become a standard, with many alternative protocols pursuing or leveraging compatibility with it to increase developer adoption.

Ethereum was followed by other multiparty smart contract platforms incorporated with distributed ledgers: most notably, Solana, Algorand, Cosmos, and others each proposed new protocols with unique advantages. For example, Solana, initially described by Yakovenko (7), proposes a distributed ledger and global state machine utilizing a universal timestamps to create a unique and performant consensus mechanism. On the other hand, Algorand, invented by Micalli (8), proposes an alternative consensus mechanism relying on cryptographic randomness to generate a secure and stable distributed ledger. This allows Algorand to be performant while retaining a significant amount of decentralization, and decreases the likelihood of a Distributed Denial of Service attack rendering the blockchain offline.

A common high-level smart contract scripting language was first extensively described by Szabo (9), who saw it as an algorithmic representation of a legally valid contract between two parties. These smart contracts were defined in a formal "mini" high-level language, and could be applied to financial, medical, and legal agreements. Simplicity was also key: contracts are meant to be readable by individuals coming from backgrounds such as law and medicine. Adamnite is meant to be a general distributed ledger and state platform which can process operations specified in such a high-level and universal contractual language.

II. The Distributed Ledger

Fundamentally, Adamnite is a distributed and decentralized network: it depends on the active participation of multiple geographically sparse parties to function as intended. To achieve this, Adamnite implements a distributed ledger that stores individual accounts, contracts, the current state, and other data. This data is manipulated and read through transactions that are continuously recorded on the distributed ledger. These transactions are often value exchanged between two parties who are either interacting directly or through a contract that is stored on the ledger. Transactions are approved through a decentralized consensus protocol, with each transaction being recorded via a timestamp and combined into data structures commonly known as a block. These blocks are then linked together via a cryptographic reference that points to the previous block. The consensus protocol can be assumed to be Byzantine Fault Tolerant; that is, we can reasonably expect that the failure of a single node or validator will not result in the failure of the entire network. We now move to a formal discussion of Adamnite's Distributed Ledger.

A. The Blockchain

Like other generalized multi-party smart contract development platforms, Adamnite can be defined as a state machine that calculates a new "canonical" state based upon the execution of some transactions applied to the previous state. This canonical state can store any sort of decentralized information, including but not limited to identities, on-chain assets, or rules governing the operation of some decentralized entity. While there are certain low level optimizations that have been made to Adamnite's underlying storage mechanism that make it more efficient, the protocol can still be formally defined as a state-transistion mechanism that operates through transactions:

$$\boldsymbol{\sigma}_{t+1} \equiv \Upsilon(\boldsymbol{\sigma}_t, t) \tag{1}$$

This is essentially the same as Ethereum's transaction-based state transition model, where external transactions define changes to the overall state of the distributed ledger. $\boldsymbol{\omega}$ represents the future canonical state, and Υ represents the typical state transition function. As with Ethereum, any amount of computation and storage can be carried out, with the only restriction being the economics of such an endeavor. In that sense, the Adamnite network can be seen as an implementation of a Turing Complete state machine.

Like other implementations of such systems, transactions on Adamnite are collected into blocks, which are then put together into a chain through secure cryptographic hashes. Thus, Adamnite can be defined as a blockchain, a specific implementation of the broader class of Distributed Ledger Technologies that aggregates blocks of data through cryptographic means. Blocks themselves contain a header which stores cryptographic references to the previous block, all the transactions in the current block, and a reference to the current state of the ledger. Note that the entire state or list of transactions is not stored; rather, a compact cryptographic proof that points to a storage mechanism such as a Merkle Tree is leveraged as the reference. Blocks themselves are proposed (and validated) by democratically elected witnesses who have an incentive to act in the best interest of the overall ecosystem by means of an implicit social contract: the ecosystem benefits from having dedicated and honest nodes validating the distributed ledger, while the elected nodes are rewarded with units of the underlying digital currency as a reward for their work. Not only does this currency carry with it some level of monetary value, it is also used as an indicator of an individual participant's voting power, thus giving elected nodes who act honestly (and are thus rewarded) more power in future elections. This process is an implementation of the Delegated Proof of Stake consensus mechanism, and is discussed in more detail later in section 3. The actual blocks on Adamnite's distributed ledger defined as follows:

$$\boldsymbol{\sigma}_{t+1} \equiv \Pi(\boldsymbol{\sigma}_t, B) \quad (2)$$

$$B \equiv (\Omega(B-1), t, \Delta, ..), (T_0, T_1, ..), (\omega), .. \quad (3)$$

Here, Π simply represents a state- transition within the block to account for the block reward, while B represents the block itself, which includes a block header, a list of all the transactions within the block, a list of the witnesses who had been elected for the particular block, and other low-level data. The block header contains $\Omega(B-1)$, which represents the hash of the previous block in the ledger (for now, we can consider Ω to be a secure black-box hash function that is both one-way and collision resistant), the timestamp at which the block was proposed, the signature of block proposer, an identifier, and other information related to both the storage of the individual block and the state of the entire ledger. While this definition is similar to that of Ethereum's, slight nuances are made for Adamnite's unique needs.

B. Digital Currency

Like with other decentralized consensus-based protocols, there is a need for an underlying currency that serves to add economic functionality to the system, reward validators as discussed earlier, and establish a method of exchange for the usage of the underlying computing system to host contracts or internet applications. Adamnite's underlying digital currency is called Nite, and is used as both an incentive to elected validators and as an internal medium of exchange. Like Ethereum, and its intrinsic currency Ether, Nite has several subdenominations named after prominent contributors to cryptography, digital assets, and more:

1) Micali: 1
2) Sunny: 10^{10}
3) Vitalik: 10^{12}
4) Nite: 10^{14}

In Adamnite, Nite is more than just a digital currency for processing payments. It is also a key for being able to participate in consensus: any individual with a valid Adamnite account will be able to use their Nite to vote for validators. Nite will also be the gateway to the entire class of decentralized internet based applications that exist on the Adamnite platform, with users being able to use their Nite in various ways to interact with these applications. In that sense, Nite can be thought of as an implementation of a digital key that enables access to the entirety of the Adamnite network.

III. Data Structures and Protocol Consensus

A blockchain or smart contract platform can be described more generally as a decentralized database storing various active and inactive data. In most blockchains, the underlying data is comprised of accounts, transactions, autonomous programs, and low level storage data. In order to both validate on-chain data and ensure that the ledger is kept updated with correct information without having to rely on a centralized mint or authority, there needs to be a procedure that allows for certain participants or parties within the network to both update and validate the information that is computed on the ledger. In most other implementations of currency-based blockchains, a measure of some form of contribution or dedication to the network is used to decide which node (we use the terms participant, account, and node interchangeably throughout this work; they all designate a player within the network that can execute the protocol and interact with others) has the ability to take such actions. Popular implementations such as Bitcoin and Ethereum measure the amount of raw computing power that a particular node has dedicated to solving a sufficiently hard problem in order to decide which blocks are added to the ledger in a process that is frequently termed Proof of Work (PoW), while other platforms use a variation of the amount of the native digital currency that an account has to decide how blocks are added to the underlying ledger. Both processes involving adding some degree of economic backing to the network, thus giving it long-term validity. They also double as issuance, as accounts are rewarded for their contribution through the native currency of the platform.

In Adamnite, as previously discussed, a ring of delegates are chosen by the broader network to both propose and approve blocks of transactions. This process can be defined as a variation of Delegated Proof of Stake (DPOS), a common consensus mechanism originally invented by Larmier (10) in July of 2014. Delegated Proof of Stake itself can be seen as a common implementation of a democratic peer to peer network protocol, or a democratic-crypto system. As both block validation and block proposal are, to a degree, concentrated, Adamnite takes additional measures to protect the network from common attacks that plague other stake-based consensus systems. We now describe both the core account structure and transaction types.

A. Formal Description of Accounts and Account Storage

Account data itself is stored in a binary merkle tree for simplicity, and should be stored as key-values within the trie itself. The account state, formally defined as α, has the following parameters:

nonce: A scalar representing the amount of transactions that the account has sent. For autonomous accounts, these include application-call transactions (transactions that are made in the context of a message call to the underlying code) Formally defined as α_n

balance: An unsigned integer representing the balance, in Micalli, owned by the account. Formally defined as α_b.

Rewards: The total amount, in Micalli, received by the account from participating in the staking process. Formally defined as as α_r.

Data: The 512-bit hash of the underlying data stored by the account; it is a mapping to the underlying binary merkle trie that actually stores the data for the account. Formally defined as α_d

ADVM Code: A hash of the virtual machine code for the account. The underlying code is executed in the event of an application call to the account; in the case of a manual account, this field is empty. Formally defined as α_c

It can be assumed that a standard serialization process allows the account state data to be stored in the underlying binary Merkle Trie, and that values can be retrieved by accessing the key-value pairs.

B. Transactions

Transactions (t) are cryptographically signed messages that relay information to the broader network. This information could be financial, such as the transfer of x nite from one non-autonomous account to another, or something else entirely, such as an manual participation transaction indicating which public addresses an account wants to represent them as delegates. Transactions can be sent by both autonomous or non-autonomous accounts; a non-autonomous account can send a transaction for the purpose of procuring some good or service, and an autonomous account can send another transaction for change, paid back to the original non-autonomous account. There are two main types of transactions: regular transactions, and application transactions. Regular transactions are the primary form of transactions; the example given above is a type of regular transaction. Application transactions are used to create an autonomous account (a chain-based smart contract), and are primarily sent by non-autonomous accounts (humans), although there is no physical or computational limitation that prevents one contract from creating another contract within its own predefined logical or procedural framework. Transactions in Adamnite have minimal fees due to the corresponding minimal computational power required to validate them. This fee is measured in ate, which simply another denomination of the nite subdenomination micalli. We define 1 ate = 1 micalli $*10^7$ Regardless, all transactions have similar parameters:

type: The type of transaction: an application-creation transaction or just a regular transaction. Formally defined as T_y

from: The 160-bit public address of the sender of the transaction; can belong to any type of account. Formally defined as T_t.

amount: The total amount, in Micalli, to be transferred in the transaction from the sender to the receiver. Formally defined as T_a.

timestamp: An UNIX timestamp specifying the time at which the transaction was sent.

message: The message is simply any data, encoded as a general byte array, that accompanies the transaction. This can be underlying operational code, or storage data pointing to the recipients underlying data. Formally defined as T_m.

fee: The value, in ate, that the sender of transaction is filling to pay for the transaction. Most client-side implementations and on-chain wallets will automatically calculate this value based on the computational size of the transaction. Formally defined as T_f.

r, s: r and s are cryptographic values derived from the Elliptic Curve used to sign transactions. This is discussed in more detail in subsection D of this section. Formally defined as T_0 and T_1.

To sign transactions, a combination of the SHA-512 Hash (from the SHA-3 Family) and the Secp256k1 curve order (from the broader category of elliptic curves) is used to generate a secure and recoverable signature derived from the sender's private key. This implementation is similar to the process described by Wood in Appendix F of Ethereum's Yellow Paper, and thus draws heavily from the general implementation described by Gura. (11)

We assume that the sender has a valid secret key P_s, which is a randomly selected unsigned integer from a source of relative entropy in the range of $[1, Secp256k1n - 1]$ of size 32. The derivation and recovery curve functions are essentially the same as Ethereum's: the ECDSA Functions ECDSASIGN (SIGN) and ECDSARECOVER (RECOVER) are used to sign transactions and recover the public key associated with the signature, respectively. To generate the public key P_p, a combination of the hashing functions SHA-512 and RIPMED-160 are used, defined formally as H_a and H_b respectively, along with the function ECDSAPUBKEY (PUBLIC). It can be assumed that all intermediate values are 32-byte unless specified otherwise. A formal description follows:

$$\texttt{PUBLIC}(p_\mathrm{s}) \equiv p_\mathrm{p} \in \mathbb{B}_{64} \tag{4}$$

$$\texttt{SIGN}(e, p_\mathrm{s}) \equiv (\mathrm{v} \in \mathbb{B}_1, r, s) \tag{5}$$

$$\texttt{RECOVER}(e, v, r, s) \equiv p_\mathrm{p} \in \mathbb{B}_{64} \tag{6}$$

Here, p_a, the public address, is derived by hashing the value p_p twice, first by SHA-512 and then by RIPEMD-160. This address is then encoded by a common encoding protocol such as BECH-32 to generate the public address that is seen in one's wallet or account.

The information that is actually mapped by the SIGN Function (e) is simply a hash of the transaction values, excluding the

ECDSA signature values r and s. These values are truncated (the first half is used), and then hashed with the hash function h_a:

$$A(p_{\mathrm{r}}) = \mathcal{B}_{0..256}\big(\mathtt{h_a}\big(\mathtt{PUBLIC}(p_{\mathrm{s}})\big)\big) \tag{7}$$

C. Blocks

Blocks in Adamnite are simply a collection of all the transactions within a particular framework, along with relevant information relating to consensus. A block is comprised of the block header U, which contains identifying information about the block itself, the transaction list T, which contains relevant information of all the transactions that in the block, and a witness list W, which contains information about the pool of witnesses chosen to act as validators for this particular block. The block header U contains:

Previous Hash: The SHA-512 hash of the previous block in the timestamp chain. Formally, U_p.

Timestamp: An unsigned scalar value that equals the UNIX time of this block's creation. Formally, U_t.

Witness: The public bit-address of the validator who proposed the block, defined as U_w. This address is also the recipient of all the transaction fees associated with the block.

Net Fee:: An unsigned integer representing the total sum of the fees for all the transactions included in the block, formally defined as U_f.

Storage Size: An unsigned integer representing the total bytes of storage and data in the block, formally defined as U_s.

Nonce: An unsigned integer representing the total amount of valid blocks that came before this block, starting from block 0. For example, a block with a nonce value of 5 will have 4 blocks: 0, 1, 2, 3, and 4, before it. Formally defined as U_n.

Signature: A cryptographic signature created through a common ring signature scheme that serves as a proof that the block was approved by the witnesses selected to serve as the validators for the block. Formally defined as U_i.

Transaction Root: The SHA-512 Hash of the root of the trie structure containing all the transactions for this block. Formally defined as U_r.

State Root: The SHA-512 Hash of the root of the trie structure containing the state after all transitions and transaction changes have been applied. Formally defined as U_m.

The block header U is combined with the transaction list T and the witness list W to generate a valid block. We can thus define a valid block by referring to the following tuple:

$$B = (B_u, B_t, B_w) \tag{8}$$

A block's validity is inherently dependent on the values contained within it, and standard logic. The state as dictated through the underlying transactions must be consistent; for example, a party P should not be double spending the same unit of currency twice. Essentially, transitions to the base state made throughout the serial execution of each transactions in the block must be consistent within the rules of the protocol. As described in equation 1, the underlying state is defined as σ, and transitions to σ are defined through the transactions contained in blocks. Throughout a block's execution, the changes made to the state as a result of each transaction must also be consistent with another. Rather than specifying a serialization function, we assume the existence of a black-box function SERIAL that serializes objects, including blocks and transactions, native to Adamnite's protocol to a common byte format that can be understood by computers interacting with each other in the context of the Adamnite Protocol. SERIAL thus represents a common function whose output can be understood by all participants interacting with the Adamnite Protocol. Thus, we define the preparation functions for both the block header U and the block B in the context of SERIAL:

$$L_H = SERIAL((B_p, B_t, B_w, B_f, B_s, B_n, B_i, B_m, B_o)) \tag{9}$$

$$L_B = SERIAL(B_u, B_t, B_w) \tag{10}$$

Thus, SERIAL can be considered to be the canonical method of translating block information (headers, transaction lists, and witness lists) into a consistent byte format that should be understood by different computer clients within the protocol, and provides a method for translating block and transaction information to a byte format that can be easily transferred via a P2P protocol.

D. Transaction Fees

As with Ethereum, transaction fees on Adamnite are used as a deterrent to network abuse, specifically when one party interacting with the protocol takes advantages of its decentralized nature to continuously execute computational operations. These fees have their own dedicated/ subunit: ate. Any operation on the Adamnite network, from sending transactions to creating a new on-chain contact, is associated with a specific and universal fee paid in ATE as soon as the operation executes.

Certain computational structures (such as a smart contract that continuously executes the SHA-256 hash function to cryptographically secure some arbitrary data) will be more expensive than others (a simple transaction sending k Nite to another account). As with other smart contract platforms, fees are ultimately a submarket of their own: participants can specify the fee they want to pay for the execution of their transaction, but a lower than average fee may result in their transaction taking longer to execute than similar transactions that have specified a higher fee. It is worth noting that due to Adamnite's consensus model allows for fees to be significantly lower than legacy Proof of Work systems such as Bitcoin.

Another difference in Adamnite's fee model is the inclusion of a minimum fee for average transaction:s t_f This fee is meant to represent the normative fee that guarantees a transaction will be accepted by the network, assuming average congestions.

A sender can increase this transaction fee for a time intensive transaction, such as a transaction that is time-intensive (an example will be a transaction containing the solution to some computational problem with x difficulty sent to a smart contract that automatically rewards the first sender that sends a transaction with the solution). However, for normal on-chain transactions, the average fee should guarantee acceptance within a reasonable time.

E. DPOS Consensus Mechanism

Adamnite's consensus mechanism follows a typical DPOS scheme, as discussed earlier. From that perspective, the Adamnite network can be described as a specification of a cryptographic peer to peer participatory democracy, where one's ownership stake in the network directly determines their ability to influence who the validators for the next round of blocks will be. The election process is used to add validity to the blockchain itself: it represents that the current canonical chain was created by and approved by nodes chosen by the broader network. There is also an incentive for nodes to be chosen as validators: block proposers are rewarded for proposing blocks, which acts as both an incentive for current validators to act honestly for the benefit for the network and for regular nodes to attempt to become validators in the future. Both block proposal and block validation is ultimately in the hands of these democratically elected validators, making Adamnite potentially more efficient than Proof of Work and most Proof of Stake alternatives.

Adamnite's DPOS scheme is actually used to elect two sets of validators, which we define as "chambers", in a parallel fashion. These chambers validate and reach consensus on different aspects of the distributed ledger. Chamber A is a set of validators tasked with executing the consensus and state transition protocol: they handle simple payment transactions that do not manipulate or take into account any code or storage tries and make overall changes to the state as necessary. Chamber B handles the execution of application-operations, such as the creation of a new autonomous account or a message-execution. Chamber B, upon completing a block of application operations, produces a state-transition batch that is then executed by Chamber A on the current state. Both chambers are elected through the same consensus mechanism in a parallel fashion. We focus on how A and B are elected and how consensus is reached in A in the remainder of this section. More information on the storage of contract code and data, its execution, and the general manner in which the various nodes elected to B validate application-related transactions cna be found in section IV.

It is extremely important that the DPOS process is secure and ensures that the compromise or malicious behavior of a single validator does not result in the failure of the entire network. Furthermore, the Adamnite network should prioritize decentralization, with any individual that has the minimal hardware requirements to participate in the peer to peer network being eligible to potentially become a validator.

One of the most attractive propositions of the DPOS system, its democratic nature, is also the reason for its largest drawbacks. A decentralized system relying on a DPOS consensus mechanism suffers from the same diseases as physical governments relying on Democratic elections: corruption and bribery. Just as malicious politicians gain an unfair advantage by corrupting the entities that handle elections and holding power over individuals with significant wealth, malicious nodes bribe would be voters and form cartels that allow them to fully control the consensus process. This could result in the approval of incorrect blocks that provide an unfair benefit to the current validators, or certain transactions being excluded entirely to either again provide a benefit to the current validators or harm accounts that belong to individuals that the validators consider to be an enemy. Centralization also becomes an issue; a situation in which consensus is ultimately controlled by a small group of individuals who own a significant portion of the network (regardless of whether they are malicious or not) to form an infallible group that constantly retains control of the consensus process. Not only is this ultimately a semi-centralized system (one's chances of becoming a validator is innately tied to their economic power, although to a degree less than traditional staking systems). Thus, it is important for any system that uses DPOS to take precautions to ensure that it continues to enjoy the efficiencies associated with traditional DPOS while not suffering from the same pitfalls.

While a significant part of a potential solution is simply concentrated in the underlying tokenomics (the distribution and allocation of the native token of the platform) and is thus out of the scope of this paper, we propose a technical solution within the consensus process itself. Consensus in Adamnite is actually a variation of traditional DPOS, using randomness and a reputation-based staking algorithm to protect against centralization and malicious validators respectively. This variation of DPOS is described in detail below.

We define a voting process V. V is simply a period of time in which snapshots of all accounts that are eligible to be witness is taken. To vote for a witness, an account only needs to have at least one micalli to participate in this process, needing only to possess a wallet that is directly interacting with the underlying peer to peer network to be able to convey their decision to the other nodes. We also define K, a black-box verifiable random function (VRF) that takes as its input a group of participants and an unlimited amount of tuples containing various variables and their weights. The specification of K is not important; any VRF that can be used to select n out of m, where n is a constant and m is variable, and that can take in multiple variables as weights, can be used. At the end of the voting process, validators are selected through an iterative random process that is weighed by several different variables.

$$G = V((x0, m0), (x1, m1), (x2, m2)...(xn, mn)) \tag{11}$$

$$A = Q(G) \tag{12}$$

$$B = K(A, (m0, m1, m2, ...mn), (w0, w1, w2, ...wn)) \tag{13}$$

$$C = K(A - B, (m0, m1, m2, ...mn), (w0, w1, w2, ...wn)) \tag{14}$$

$$\tag{15}$$

G is a tuple containing tuples that describe the public key and votes allocated to each candidate at the conclusion of the voting process. Again, V transforms the public keys and votes received for each candidate into a "tuple of tuples", G, that stores this information in an efficient manner. A is the first voting pool, and is simply the top Q percentile of G as determined by the amount of votes that were allocated to each candidate. Finally, B is the pool of validators selected for chamber X, the transaction and state tier, for the current round. The size of B should be a constant number, and should be able changed only through an on-chain fork. K is the verifiable random function that selects the candidates, weighed by the amount of votes they received and their reputation, an algorithmic representation of a node's behavior when they have previously served as validators. Actions that can impact a validator's reputation include inactivity or proposing an invalid block. C is the pool of validators chosen for chamber Y, simply be executing the VRF K again on the complement of B in A.

F. Fork Choice and Byzantine Fault Tolerance

Because both block proposal and block validation are restricted to a few democratically-elected and trustworthy individuals during consensus, Adamnite does not need as rigorous of a finality or security guarantee as alternative protocols that utilize Proof of Stake consensus. However, we do define several protocol-level parameters that provide some degree of certainty surrounding forks, finality, and misbehavior or collusion from the pool of elected validators. We propose that as long as a supermajority (2/3) of the total number of votes (coins) are allocated to honest validators during any given round, the proposed blocks will be both legitimate and reach finality. Furthermore, even if a significant portion of coins are allocated to malicious parties, Adamnite's usage of reputation and other factors as weights during the selection of validators for consensus should provide stronger guarantees of security than traditional BFT style protocols that only rely on the allocation of coins throughout the network.

We provide a short description of how the chosen validators q for a round r (where r contains c blocks) propose, certify, and reach consensus on blocks. At the beginning of each block proposal phase p (p is a factor of c), q merges all of their views of the current state into one, reaching agreement about the last valid block for the canonical chain that they are building on. The process that the q reaches to reach agreement about the current chain is similar to the Goldfish Algorithm, proposed by D'Amato, Neu, and others. At the beginning of a round, validators reach agreement on the current state of the chain by directly referencing the participation transactions that occurred during the first phase of the consensus process. Each participation transaction contains a reference to the last block that a particular node considered valid. Validators construct their view based on the weight of votes ascribed to each particular block (and henceforth, a particular fork): the block and corresponding chain with the most votes, as measured by the total amount of coins allocated by all voters in participation transactions referencing that particular block, is selected as the canonical chain on which blocks for that particular round are created. Once the validators reach agreement on the chain, they sign a certificate that signifies their agreement and references the chain being built upon. Because validators only use the most recent allocation of votes, and thus discount any previous weight assignments, a key component of Goldfish (specifically vote expiry) is utilized. Reaching agreement helps ensure that synchronous honest validators are aware in the chance that a malicious block producer creates a secret chain and tries to ascribe votes to it.

Once a block has been proposed by the block producer, validators check to ensure that it is valid, and more specifically, check to ensure that the block producer is not proposing blocks on a separate fork from the agreed upon core chain. If a proposer is found to be proposing nonsense blocks, is inactive (either maliciously or accidentally), or maliciously proposing blocks on the wrong chain, then a validator in the q can report them with the corresponding proof of their misbehavior. If the report is successful (based on the underlying cryptographic proof), then the block proposer is replaced with a validator from q, and a VRF is executed on the predefined pool to select a new validator to replace the block proposer in q. The block producer will also lose a significant portion of their reputation, which is described in the next section.

G. RepuStake

The specific reputation-based algorithm used by Adamnite is called RepuStake, first defined by the author of this work in late 2021 (13). It provides a basis for using reputation as a weight when selecting validators in a DPOS consensus system, and defines algorithmically how a reputation score may be calculated for an elected node. This algorithm is slightly modified for Adamnite as it focuses only on validators/witnesses, not regular nodes. Reputation-based staking systems were also discovered previously and independently by Hu (14), among others. A brief description of this algorithm follows:
There exists a tree L which stores the account information (balance, public address, nonce, etc) for all the accounts in the network. Among this information, a boolean value "Validator" exists. This value is true if the account has previously served as a validator, and false otherwise. If the account has previously served as a validator, an additional parameter, reputation (R), is added to L. Reputation is a score between 0-1 assigned to

the account, and is mutable whenever an account is selected to be a validator for a particular round. Proposing a correct block, for example, may increase a validator's reputation score by 0.1, while signing or proposing an invalid block decreases their score by 0.5, and being inactive decreases their score by 0.1. The specific amounts are not important; a valid RepuStake implementation must only have a large requirement to become trustworthy, and severely penalize behavior by untrustworthy nodes.

This algorithm is still under development, and will likely change over time as the Adamnite Protocol continues to become more formalized. Furthermore, reputation can also be used in contexts beyond the core Adamnite implementation, such as in clients or wallets that allow users to passively participate in consensus (and thus earn staking rewards) by simply dedicating their stake to candidates with high reputation scores. One potential integration of RepuStake within Adamnite's consensus mechanism is by using it as a weight within the VRF used to select validators. This can be a numerical measure, such as the percentage of blocks that the validator has proposed in its history. An integration such as this will allow for reputation to play a role without having to turn to off-chain storage or contract-based calculations.

H. Block Execution

The execution of a block, or the process by which a block is proposed and approved by the network, is perhaps the most critical part of the consensus protocol. Note that is only refers to the actual blockchain, and not to the off-chain database containing smart contract code and storage. State transitions (transactions and changes dependent on smart contract calls) are handled in a different fashion, and pushed to the main chain in batches. The specifcs of this are covered in later sections. The DPOS consensus mechanism ultimately serves to dictate an efficient and secure process through which a canonical chain (the one approved continuously by all elected validators since the first, or genesis, block) can be defined. This process is actually quite simple, and is described below in conjunction with a function `DPOS`, which describes the process through which a block is proposed and added to the canonical chain:

$$m = K_1(B) \quad \wedge \quad n \geq \frac{2}{3} \quad \text{with} \quad (m,n) = \texttt{DPOS}(H_b, H_x, d) \tag{16}$$

Here, K_1 is a VRF that is capable of simply picking one item from a list. K_1 can be completely identical to K, or can come from a different family of VRFs entirely. K_1 is used to randomly pick a block proposer for the next i blocks, where i is again some constant that can only be changed through a fork, and when multiplied by the total size of B, results in $blank$, the number of blocks in a round.// It can be assumed that each elected witness is given a witness specific public-private key pair used to approve newly proposed blocks, or if they are a proposer, signal that they are the ones who proposed the block. This private key can only be used in the context of the round, and once a new group of validators is elected, a new set of validator private keys is issued, rendering the ones used in the previous round unusable. For the purpose of block proposal, we leverage a generalized ring signature algorithm (as described by Rivest, Shamir, and Tauman (12)) H_x that allows the block proposer m to propose the block with their signature without having to expose their identity. Block validators can confirm that the block proposer was indeed among the current group of chosen validators, but will be unable to discern the exact identity of the block proposer. This applies to both the witnesses elected for the current round, and external nodes who are validating on-chain information through the peer to peer network. This serves two purposes: validators are encouraged to validate blocks based on their correctness rather than the identity of the individual who proposed the block, thus further combatting corruption and discouraging the development of cartels within the broader network of potential validators, and ensuring that malicious attackers are unable to determine the identity of the proposer m for the current sub-round i, thus protecting the network against DDOS attacks that target a specific validator. The specifics of this protocol are as follows: The block proposer for the current round creates a block containing transactions, state transitions as a result of contract calls, and other changes to the core blockchain. The block proposer then uses a ring signature to sign the block, signifying that they are a part of the current group of validators, but not revealing their true identity.

Once a block has been proposed, validators take turns appending their signatures to the block to signal their approval. n is simply the ratio of approvals to the total number of validators, and must be at least 2/3 in order for a block to be appended to the current canonical chain. In the case where 2/3 of validators cannot come to agreement, a delay occurs. In the event of an extended delay, an empty block is proposed, and a new chamber of validators are selected from the predefined pool using a VRF. Finally, d in the equation above is simply the associated block data.

We also assume that there is a schedule in place for block finality, the selection of block proposers, block validators, and the composition of the general voting pool. We provide a mathematical example below:

$$R_t = \Delta \tag{17}$$

$$B_t = \Delta + 1/126(x) \tag{18}$$

$$C_t = B_t + 1/252(x) \tag{19}$$

$$P_t = 6(C_t) \tag{20}$$

(21)

Here, R_t is the start of the round, x is the total amount of time allocated to the round, B_t is the first time a block is agreed upon and proposed to the network, C_t is a time period in which the network can reject the transaction (resulting in an empty block), and P_t is when a new block proposer is chosen.

IV. Storage and Execution

A. *State Storage*

The canonical state of the overall blockchain, formally comprised of account addresses, balances, data, rewards, and ADVM code, is the main data structure within the Adamnite Ecosystem beyond the blockchain itself. As with Ethereum, the state is stored in a database (full nodes will have to maintain a copy of this database), data is maintained in a variation of a traditional Merkle Trie, and addresses are mapped to the other fields. We define the current state of the blockchain as a mapping between addresses, balances, rewards, and their contract code and data. In Adamnite, the state root is conventional: it is simply the root of the Merkle Trie.

In Adamnite, a significant part of state storage, specifically the storage of code (for contracts) and the storage of data, is handled offchain. Contract code and data might be stored in an offchain database, and be validated on-chain through an one-way hash function. Thus, during a contract-based typical state transition (such as one caused by interacting with a smart contract or deploying a new smart contract), both the off-chain database and the on-chain hash are updated. For our implementation, the hashing function used is simply the $SHA-512$ hashing function. Furthermore, when multiple application calls are made in a batch (perhaps by a user interacting with a complex on-chain application), the validators approving the transaction must also check the sender's balance and state to ensure that it has not changed since the beginning of the batch.

Batch updates are pushed through optimistic pushes, thus working in a similar fashion to a layer-2 protocol on an alternative chain, although there are some slight differences: the database's sole purpose is to store contract information and execute state transitions associated with contract calls. Thus, the database's storage and push protocols are much less complex than a layer-2 such as Optimism. Batch updates are constructed in the following manner: once a set of state transitions for a batch is approved (this can be set to a predefined number), the state generator (described in more detail in the next section), upon confirming approval of the proposed state changes within their quoroum, packages them into a batch. A batch is essentially a list of state transitions, along with their requirements: for example, transferring one unit of a subcurrency from Bob's account to Alice's requires that Bob has at least one unit of said subcurrency. Creating new smart contracts or updating the code of an existing smart contract will have no requirements except for checking to ensure that the sender has enough NITE in their account to pay the associated fees.

Upon finalizing the batch of state transitions, the state generator pushes it to the peer to peer network. Chamber A receives this message, and the chosen block proposer proceeds to check its validity before making a state update block, which is then met with the same approval requirement (2/3) as regular blocks. Upon approval, a state block is pushed to the change, resulting in changes to account balances, storage hashes, and code hashes. This execution is documented mathematically below:

$$X_b = ((a_0 : ax_0, ax_1 ... ax_n), (a_1 : .....), (a_n : ax_0))Q_1 = \Upsilon(Q_0, X_t) \quad (2$$

Here, X_b represents a batch, with each entry representing a state transition and its associated requirements. Once the batch is approved, a state transition from Q_0 to Q_1 is defined, with X_t representing the state update block. It is worth noting that state update blocks have a higher maximum size, as they are designed for scalability. During a batch update, chamber A will also check for potential malicious activity or inconsistent logic as defined as by the requirements. This can be done by checking to see if the proposed storage updates are consistent with the logic in the underlying code and storage for each smart contract call. For example, if applying a state transition a requires some account to have a certain balance in NITE, then chamber A will check to ensure that the account currently has the required balance. This is to prevent a double-spend attack in the context of smart contracts: although contract calls are processed with little latency, there is the possibility that an account may attempt to send regular transactions to execute a double-spend attack in the hope that both their smart contract call and regular transaction will get approved. Checking to ensure that an account meets the requirements to actually completely a transaction is key to mitigate against this sort of attack.

Once an individual batch is uploaded to the core chain, the witnesses in chamber A have until the next batch to produce a fraud proof and challenge the batch due to suspicion of malicious activity. We assume that batches are pushed at a predefined schedule with respect to the core chain rounds: for example, batches can be pushed every 4 blocks. Once a fraud proof has been made, it is then executed to determine if malicious activity took place. If the fraud proof is correct, the batch is redone by chamber b, while the current state generator loses reputation and is replaced.

The data structure used to map state data is defined as a binary merkle trie. We propose a model similar to that proposed by Buterin and Ballet in EIP-3102 for Ethereum. In Adamnite, for each individual account, balance, rewards, code hash, and data hash are all stored as key-value pairs in a binary merkle trie. Using a binary Merkle Trie has specific advantages over current implementations such as the Merkle Patricia Trie, namely in terms of simplicity and saving overall disk size for clients who want to maintain a full node. A current shift in distributed ledger ecosystems, as noted in EIP-3102, is the move toward stateless clients, or at least clients that do not present as large of a computational requirement. Storing state in a binary merkle trie goes toward this goal. The typical state trie, because of Adamnite's account-based structure, will therefore contain a key-based reference to the account fields discussed previously. We define this structure as $TRIE$; for a singular account with address A_a, the structure is thus $TRIE(A_b, A_c, A_d, A_n)$, where A_b, A_c, A_d, A_n represent the balance of the account, the hash of the account's code, the hash of the account's data, and its nonce, respectively. Specific implementation details can be found in Section B of the appendix. In a future implementation, this may be transitioned to a tree dependent on vector commitments (stylized as a verkle trie).

B. *Contract Calls and Execution*

As discussed previously, Adamnite uses a second set of validators that execute applications and message calls. This was previously defined as chamber Y. Chamber Y executes operations on the off-chain database storing data and code, and provides a batch of state transitions to chamber x (which executes base transactions and updates the actual state of the public ledger) of all state-changes after executing a certain amount of application transactions, which also includes the creation of new applications and contracts. Note that both the validators and the off-chain database have access to on-chain state information (namely balance and nonce) to validate messages/transactions. The set of validators in chamber Y reach consensus on the state of the blockchain in a drastically different fashion than their counterparts chamber X. While the validators in X individually validate each individual change to state, the validators in Y use a more efficient and scalable process that is albeit more centralized. The validators in Y adopt an algorithm similar to that used by Solana for scalability: a single elected witness distributes packets of information pertaining to state changes, and uses a Verifiable Delay Function (VDF) to show that verifiable time has passed between different application related events.

A specification of this process follows. At the beginning of a round r in chamber y, a state generator (analogous to a POH generator in Solana's protocol) processes user messages and smart contract calls on their local copy of the state. After each state transition, the state generator generates a $SHA-512$ hash of the byteform of the resulting state. The state generator should then order the state transitions and their corresponding results in a way that maximizes verification throughput. The state generator then splits up the state transitions based on their proximity to one another; if an individual state transition overlaps with another (for example, if they are calling the same contract or impact the same account), they are grouped together. These groups, along with their corresponding proof of history records, are split up by the state generator and sent to the remaining witnesses in chamber Y. The proof of history records show that a certain amount of time has passed between each state transition, as determined by the amount of time it takes to execute the hash function and calculate a new version of the state.

Once the set of witnesses in chamber Y agree to each individual state transition by appending a signature with their participation key, the state generator assembles a batch consisting of the groups of state transitions, the signatures, and their stateproofs, and sends it over the peer to peer network to chamber X, who then performs the check described previously.

Chamber Y can be thought of as a computational specific execution layer; while witnesses representing chamber Y for any given round r are elected at the same time as the witnesses in chamber X, they play vastly different roles, with chamber X representing the base execution layer which approves all state changes. By splitting contract execution and storage, we significantly reduce both execution and storage overhead: smart contract calls can be processed at a faster rate without compromising on security by allowing for spam or overflow attacks, and individual nodes can choose to store just the core state that consists of accounts and transactions, without the offchain database. A witness that wants to run for chamber Y will need to run a full node with access to the database in order to be selected.

V. Programming Environment

We now move to a formal discussion of Adamnite's programming environment and execution process. At its core, Adamnite is a platform meant to allow for the efficient creation of multiparty smart contracts that are executed and stored entirely on a distributed ledger. The programming environment includes all the components needed to achieve this principle: an easy to use modular programming language, a semi-Turing complete stack-based virtual state machine, and a smart contract creation and message execution model. We leave the specification of the contract creation and message execution up to the developer; one can assume that it is essentially the same as other multiparty smart contract development platforms such as Ethereum or Solana. We now provide a formal definition of Adamnite's programming stack, $A1$, and a description of its virtual state machine.

A. *Programming Language and Execution*

A1 is Adamnite's high level programming language, and also serves as the basis for its high-level programming environment. A1 itself is an adoption of the functional programming model employed by Haskell and other popular programming languages used in third generation blockchains. However, in practice, A1 is more of a generalized functional language: the creation of functions and modules enables programmers to package reusable code that can be used by others, thus creating a programming ecosystem in which vetted modules of code are leveraged for both security and ease of use. Functional programming has been described in length by Hughes (15), among others. A1's structure and script are heavily inspired by E, a generalized contract-programming language that allowed developers to create smart contracts in a secure distributed computing framework. A1 can be thought of a modern implementation of a distributed message-oriented programming language with an emphasis on readability. A1 is dynamically-typed; types do need to be explicitly declared, and are only defined at runtime. A1's syntax is also extremely similar to that of Python's, thus allowing any developer familiar with writing programs in Python to easily use A1 to write powerful smart contracts.

In A1, developers write programs through multiple contracts, which themselves are defined by other contracts. This process continues until the most basic scripts are reached: these axiomatic scripts form the basis for which more complex objects are designed and executed. A1 borrows heavily from E, a smart contract language originally described by (16). E allowed developers to write functional Turing-Complete scripts in an object-message framework, and allowed those contracts to function in a distributed manner in the presence of mutually suspicious parties. For A1, this is extended to create multiparty smart contracts that function entirely on a distributed ledger, where all parties interacting with the distributed ledger are considered to be adversarial. This is a marked improvement over current implementations of current multi-party smart contract programming systems, which often leave much to be desired from both a security and functional perspective. Even the slightest error results in an adversarial party being able to access private information, or manipulate the smart contract in a way that should not be allowed by the average participant. Like with E, developers using A1 can create contracts with promises, where the execution of all the steps of a contract does not need to be immediate. Smart contracts can thus represent more than just rules dictating the transfer of subassets; they can be used to relay messages between two mutually suspicious parties while ensuring that neither one of them Furthermore, most modern smart contract programming languages are almost impossible to use for the average individual who interacts with contracts on a daily basis, thus being a far cry from the contract languages originally envisioned by Szabo. In A1, scripting is simplified. Programs can be executed through an object-oriented framework through the definition of different class sets, and contracts are able to be inline into other contracts, allowing for an efficient development platform that allows developers to utilize other contracts to create their own contracts, much like how standard forms are used in the legal industry.

A1 provides an easy-to-use script that is inherently readable, supports the use of data feeds and external APIs, and is modular, allowing for the use of packages and standards that can drive the creation of complex contracts. This standard library in particular will serve to make the development of contracts easier, and will also protect developers from making rudimentary errors that can compromise their contract.

B. Virtual Machine

The message execution model presents a framework through which alterations to Adamnite's canonical state are made in the context of contract execution. Contract execution within the Adamnite protocol means the serial execution of bytecode instructions along with the processing of data external to the contract itself (if needed). To accomplish this, we specify a formal stack-based virtual machine: the Adamnite Virtual Machine (ADVM). The ADVM, like the EVM and other common implementations of distributed virtual machines, is *quasi* Turing Complete because of the practical bound on net computation imposed by transaction fees.

1) Virtual Machine: The Adamnite Virtual Machine (ADVM) is a state virtual machine and bytecode interpeter based on the popular WebAssembly (WASM) paradigm. Specifically, we leverage WASM's modularity and standardization to allow developers to easily create scalable smart contracts, and for users to be able to efficiently interact with the underlying smart contracts. WASM is also preferred because of its native 32/64 bits support; one major pain point in EVM-based execution models is the execution of arithmetic and other basic operations in 256 bits, which is extremely inefficient and contributes to the slow execution of many EVM-based programs. The ADVM itself is a general extension of WASM: it includes all the standard WASM operations, with the addition of specific fee and execution based opcodes.
While we do not offer a specific implementation, either an extension of an existing WASM engine or a new implementation should accomplish this goal. Elliptic Curve and generalized cryptographic libraries are implemented as generalized WASM functions/modules within the WASM engine. A list of these operations, along with a description of their functionality, can be found in Section C of the appendix. The VM itself is stored in a virtual ROM, and is distributed among all the active nodes within the Adamnite network. The ADVM also supports exceptional execution; examples include stack overflows, incoherent instructions, or a simple lack of fees from the executor of the message. As with the EVM, any exception results in all state changes being voided, and the error is reported to the message executor. Fees follow a slightly different model than other blockchain-based virtual machines: due to WASM's predictability, we can actually create a system in which fees are deterministic and stable. The majority of WASM engine operations are consistent, and with hashing functions and other common cryptographic operations essentially being functional libraries, operations have consistent fees (almost all operations cost 1 micalli).

2) Execution: The execution model itself determines an output, a new canonical state, and an intermediate state based on the computations executed upon the inputs. The other variables provided by the execution agent are typical; we forgo explaining these in a formal context for brevity. Rather, we define a execution function Ξ that computes these outputs:

$$(\boldsymbol{\sigma}', t', I', \mathbf{O}) \equiv \Xi(\boldsymbol{\sigma}, t, I, IN) \tag{23}$$

Here, as in Ethereum's execution function, computation on the state, transaction fee, and intermediate state results in a new output. The intermediate state I is essentially the same as Ethereum's substate. Individual operations, defined as opcodes, are executed using this function until the entirety of the message is computed, or an exception that results in the execution halting is reached. The execution of a message can only be halted through an exception if the caller lacks the ATE needed to process the execution of all the operations within the message, if the execution of the message results in a stack overflow, or if the executor's own state changes during the execution of a complex batch of calls, as noted previously. Individual operations, barring any other errors, can never result in the machine halting.

Contracts, and their associated functions, are stored in a hash table that itself is stored in a database. Every contract is made up of various functions that define how it behaves and interacts with both manual and autonomous accounts. These sets of rules can be thought of as being analogous to clauses in traditional agreements that govern interactions in the physical world.

Just as physical agreements often cite or reuse parts of other agreements to create a standard often referred to as a boilerplate contract, smart contracts should be able to reuse parts of other contracts as they are enforcing various interactions between different parties. This is achieved in Adamnite through the concept of reusable functions, referenced through hashes. Imagine that a contract A, with a function set $x_0, x_1, x_2, ...x_n$ is deployed to the Adamnite state machine. Each of those functions are referenced by a hash in a hash table that references their storage location. When a contract B is deployed, the virtual machine checks to see if any of the functions used by B are computationally identical to A in some way. We define a function x_0 and a function y_0 to be identical if they produce the same outputs with the same inputs. For example, a function that adds 8 to some number will be considered identical to a function that adds 3 and then 5 to some number. For more complicated cases, the virtual machine can use test-cases to determine if two functions are identical. From the example above, when a function from contract B is found to be identical to a function from contract A, it is not stored, and instead, the function from contract A is used. This drastically reduces bloat in the overall chain, as the storage for every subsequent contract (assuming all contracts use the same sort of templates).

3) Contract Creation and Calling: Within Adamnite's protocol, contracts are created through an application transaction, similar to the framework employed by Algorand. The creation of a contract is essentially the creation of an autonomous account that dictates interactions between multiple parties. We use the terms autonomous account and smart contract interchangeably; they both define any account on the Adamnite Blockchain that is controlled by underlying code rather than a third-party. A simple exchange contract, for example, could dictate the exchange of two assets: an external party (represented by a manual account) will send some amount of one asset to the autonomous account and the autonomous account will automatically send an equal market value of the other asset back. The current exchange rate for the assets will be determined through the account's underlying ADVM code. If specified by the creator, this underlying code can actually be changed by the creator and other trusted parties, thus allowing for contracts to be updated. This does mean that some contracts do require a certain degree of trust in the creator; however, due to the readability of ADVM code, an interested party will easily be able to determine whether a contract has such a feature.

Another unique feature for contracts created on the ADVM is their modularity (not to be confused with the functional libraries provided by the A1 programming language). Contracts can be split into subsections, which can then be individually loaded by other contracts if needed. These subsections can also be defined as standard WASM libraries (a set of precompiled contracts may also work here). This is similar to how contracts work in the legal industry; often, when one contract utilizes language or terminology from another contract, it often only leverages one specific section or case. We apply this to programs acting on the distributed ledger that function as smart contracts. When a smart contract on the Adamnite protocol calls another smart contract, it can call and load specific sections within that contract. This serves two purposes: first, it serves to make execution more efficient by diverging from the top-down execution model used by the EVM (originally described by Pilmore in an article providing an overview of the Pact smart contract programming language (17)), and second, it protects contracts from external contract calls that might be malicious, even if a subsection within it is useful. Module calls are inlined during execution, thus foregoing the need to execute the entirety of an external contract just to execute one specific function.

VI. Looking Ahead and Conclusion

We now look ahead and provide potential directions for how the Adamnite project could expand. It is worth noting that the current work is a working paper, and will likely change as the protocol evolves. In particular, the section on the programming environment will likely evolve as A1 itself becomes more formalized and further research concerning the implementation of the ADVM done. Furthermore, an expansive appendix that further formalizes the details included in this work will also be included, and example programs and diagrams depicting how A1 works will likely be included in a future release of this work. However, despite the potential mutability of this work, we maintain that the core principle of an efficient and easy to use multiparty smart contract development platform will remain the same throughout the development of both this work and the Adamnite platform.

A. Scalability Plans

One of the core tenets of the Adamnite project is the idea of a simplified distributed ledger platform; this definition goes beyond just the programming environment used to create on-chain applications and contracts. The process for individuals to download nodes to validate the ledger should also be a simple process, and be open to any individual with access to the internet. Current blockchains are often too large, and require specialized hardware for node validation. A solution to this problem exists in the form of the succinct blockchain established by Bonneau, Meckler, Rao, and Shapiro in their technical paper describing the Mina Blockchain (18). Succinct blockchains leverage zero knowledge proofs to minimize the information needed to verify the blockchain; a node only needs to validate the latest block in order to verify the entire chain. Adamnite can leverage zero knowledge proofs either in its core protocol or via a subchain to make validating on-chain transactions more accessible to the average network participant. This is key to Adamnite's long-term goal of making blockchain technology more accessible to the general public while maintaining decentralization. Eventually, anyone poessesing a device with the ability to access the internet should be able to verify the entirety of the blockchain's history.
One of Adamnite's core features that was not discussed at length was formal verification, which will be a fundamental part of the A1 programming language. Formal verification allows for contract developers to declare assertions and dynamically check their code for common errors; an extension of this is knowledge verification, which allows for developers to define how various parties may interact with the contract. A common example of such a concept is a loop invariant, which allows a developer to declare via an assertion that a particular variable will remain constant over time. Knowledge verification allows for developers to make assertions that limit the actions that a party may take; this may be useful for a game or lottery system that is predicated on manual parties interacting with one another, with the underlying smart contract acting as the server and predefining the rules for the game. In such situations, it is essential to ensure that different parties have different roles and capabilities when interacting with the underlying contract. An administrative role may be able to redefine or update the contract, while others will only be restricted to a certain set of moves. It is also important to emphasize that this verification process only serves to mitigate errors contained in a contract's logic; it does not prevent human error (a developer losing his private key to a well-engineered scam is an example). This will be explained in more detail once A1 is more formalized.

B. Conclusion

We have formally defined Adamnite, an efficient distributed ledger and easy to use multiparty smart contract development platform. Adamnite allows developers, regardless of their prior experience with blockchain technologies, to easily create autonomous contracts and applications that are independent of any centralized power and are entirely autonomous. Further, changes to this work and the Adamnite protocol will be made to accomplish the core goals of the Adamnite protocol.

VII. Acknowledgements

This work would not have been possible without the contributions of my co-founders at Adamnite Labs: Thomas Petersen, Khalil Shanti, and Brent Gillett. Without them, the Adamnite project itself would have likely died out ago. Special thanks also goes to Adamnite's open-source contributors, namely Mayank Udani, Marin Peko, Jonah Wilsmeyer, and Ajai Karthik. They have all, at one point or another, helped formulate my theoretical ideas into actual software. Finally, a special shout out to Tsimafei for helping me greatly with protocol research, leading the development of the core blockchain, and of course keeping the spirit of anonymous open-source contribution alive. Finally, a general thanks to the broader blockchain Community for suggesting ideas such as the adoption of WASM and off-chain storage of code and data, which was suggested by Reid Rankin.

References

[1] A. Chaudhury, "Adamnite: A scalable and secure blockchain development platform", December 2021.
[2] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System", 2008.
[3] A. Back, "Hashcash - A Denial of Service Counter-Measure,", August 2002.
[4] W. Dai, "BMoney," 1998.
[5] V Buterin, "Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform", 2013.
[6] G. Wood, "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER", 2014.
[7] A. Yakovenko, "Solana: A new architecture for a high performance blockchain", 2017.
[8] S. Micalli, J. Chen, "ALGORAND" 2017.
[9] N. Szabo, "A Formal Language for Analyzing Contracts" 2002.
[10] D. Larmier, "Delegated Proof-of-Stake (DPOS)" 2014.
[11] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In Cryptographic Hardware and Embedded Systems-CHES 2004, pages 119–132. Springer, 2004.
[12] R. L. Rivest, A. Shamir, Y. Tauman, How to Leak a Secret 2001.
[13] A. Chaudhury, "RepuStake" 2021.
[14] Q. Hu, "An Improved Delegated Proof of Stake Consensus Algorithm" 2021.
[15] J. Hughes, "Why Functional Programming Matters" 1990.
[16] M. Miller, "Towards a Unified Approach to Access Control and Concurrency Controls" 2006.

[17] E. Pilmore, “The EVM Is Fundamentally Unsafe” 2019.
[18] J. Bonneau, I. Meckle, V Rao E. Shapiro, “Mina: Decentralized Cryptocurrency at Scale” 2020.

Appendix

A. Peer to Peer Network

Adamnite's peer to peer network is heavily optimized for Byzantine, Altruistic, and Rational (BAR) Resiliency. Adamnite's network utilizes a whitelist, w, a greylist, g, and a blacklist b. Individual nodes also maintain a POM (Proof of Misbehavior) score for each node they exchange information within the protocol. This is not stored on-chain, or even within the protocol. Rather, individual nodes maintain these records in their own memory. We thus define the possible of set of all nodes that a node a can interact with as the tuple (w, g, b), and the reputation scores of these nodes as p. Note that this reputation score is different from the on-chain reputation score used to evaluate validators through the RepuStake protocol. Nodes on the whitelist are nodes that a can interact with and exchange information with freely, nodes on the greylist are used as an alternative in case the whitelist is inactive. The blacklist is a temporary ban list that each node maintains; a node is demoted to the blacklist if it misbehaves (such as spamming with incorrect information) and kept there for the duration of a round of consensus.

Rational nodes, on the other hand, are incentivized to act in line with the rules of the protocol to avoid demotion and exclusion. It is in every node's best interest to be connected to other nodes and participate in fair exchanges of information. Altruistic nodes are incentivized to perform optimistic pushes of information for new nodes joining the protocol to not only stay connected to as many nodes as possible, but to also provide a social proof of their contributions to the protocol, and therefore increase their chance of being elected as a validator in the future.

B. Binary Merkle Trie

The binary merkle trie data structure, used to map key-value references from account addresses to balance, code hash, and storage hash, is an integral part of Adamnite. The Binary Merkle Trie is organized in a node structure, with leaf nodes containing specific values that map to given keypairs. In a manner similar to EIP-3102, we define an example of how an account's various values may be accessed.

Every account on the Adamnite Blockchain can be represented by the following tuple: $(A_a, A_b, A_c, A_d, A_n$, which represents the account's address, balance, code hash, data hash, and nonce, respectively. The merkle trie structure therefore maps A_a to the other values in the following fashion:

$$A_b = HASH(A_a)[0...253] + +0b00 \tag{24}$$
$$A_c = HASH(A_a)[0...253] + +0b01 \tag{25}$$
$$A_d = HASH(A_a)[0...253] + +0b10 \tag{26}$$
$$A_n = HASH(A_a)[0...253] + +0b11 \tag{27}$$
$$\tag{28}$$

This setup is very similar to that proposed in EIP-3102, and provides a deterministic mapping between addresses and other state values. The actual values for code and data are stored in a separate database, presumably one that also has direct access to the on-chain trie, and in some implementations, may even store a copy, thereby also storing the entirety of the state.

The binary merkle trie structure is used to store the state of individual state machines that track different aspects of the distributed ledger. The core account state stores all the accounts, along with the parameters described previously. A participant tracker keeps track of all candidates and elected witnesses, and a transaction tracker stores all finalized transactions. These individual trackers are stored on-disk, and should be updated at the conclusion of each respective finalized state transition.

C. Contract Storage

As discussed previously, contracts code and storage are stored in an off-chain database that is separate from the core state. This database is modelled as a distributed hash table, with an individual entry storing the address of the contract, its corresponding storage, and code. To improve efficiency, contract code is stored in a modular fashion: an individual contract storage slot only stores new functions, and references previously defined functions in other contracts. For example, a common addition function, that simply takes as input two unsigned integers and returns their sum, will only be stored once for any integers a and b. If the function is defined again for a different contract, it will not be stored, and the new contract will call the function from the original contract whenever it needs to use it.

Database witnesses and candidates each maintain a copy of the database; there also exists a canonical database that is updated after each batch upload, and is available to download for all new database witnesses if they wish to double check the contents passed to them over the gossip protocol.

D. Virtual Machine Specification and Instruction Set

A majority of the ADVM's numeric and operation instruction is directly derived from WebAssembly (WASM). This allows for computation that is native to most computers. WASM is natively little-endian, and thus by extent, a majority of the ADVM's operations are also little-endian. Unique operations (such as Elliptic Curve Cryptography or generalized crypto functions) can be implemented as WASM modules. The instruction set includes a MODULE opcode that specifies the use of a specific WASM module (say the cryptography module for using the SHA-256 hash). A set of WASM instructions can be found here. The ADVM native word and stack size is 64 bit, although basic operations also have 32 bit counterparts due to WASM's standard; this makes Adamnite's execution environment extremely similar to most modern computers.

Fees for contract calls are determined using an opcode metering model, similar to other ledger-based execution environments such as the Ethereum Virtual Machine and the Algorand Virtual Machine. Thus, the cost of interacting with a smart contract is the net sum of all the fees associated with executing all the opcodes associated with the call. The ADVM's execution model is also bounded: smart contracts can only be a certain size (8 kb) and have a maximum fee of 30000 ATE. However, due to the ADVMs modular nature (hashing and signature algorithms are implemented as WASM modules that developers can directly reference), these restrictions should be easy to stay within for the vast majority of multi-party smart contracts. The purpose of these restrictions are two-fold: first, they set a barrier against malicious users who wish to simply spam the network by creating a smart contract that just sends nonsense transactions back and forth, and second, they allow for Adamnite's execution to be relatively predictive with regard to both cost and time. By setting an upper limit on fees and size, we can predict with relative accuracy both the costs associated with a smart contract call and the amount of time it will take to process.

Individual opcode fees for the ADVM are quite simple: all native arithmetic (addition, subtraction, multiplication, division), store, load, and stack operations cost 1 ATE. More complex operations, such as square roots, truncations, ceiling/floor functions, and comparisons, cost 2 ATE. Boolean operations, block operations, and loops cost 4 ATE. Finally, the MODULE opcode, which essentially loads different WASM modules for the purpose of cryptographic hashing. An external module call has a fixed cost of 500 ATE, as these modules are often focused on expensive computations (hashing and elliptic curve cryptography). This is just a generalized guideline; different implementations of the protocol may contain different fee metrics. In general, the principle of keeping a minimal fee for the majority of the operations should be followed. The ADVM also implements specific blockchain related functionality through specific opcodes. A list of these functions, along with a short description, can be found below.

Adamnite Environment Opcodes		
Address	Name	Short Description
0xC0	ADDRESS	The Address of the contrac
0xC1	BALANCE	Balance of the contract.
0xC2	CALLER	The account that called t tract.
0xC3	DATASIZE	The size of the data being
0xC4	CODESIZE	The size of the code bei cuted.
0xC5	ATETOTAL	The total amount of ATE a to this call.
0xC6	COPYDATA	Copy the current data i memory.
0xC7	COPYCODE	Copy the current code i memory.
0xC8	GETCODE	Return a copy of the curre to read.
0xC9	VALUE	Return the amount of ATE in the message.
0XCA	TIMESTAMP	The current block's timest
0XCB	CALLERBALANCE	The balance of the caller.
0xD0	CALL	Initiates a new message/sm tract call into a specific co
0xD1	CREATE	Creates a new smart contra
0xD2	IMPORT	Imports specific functions external contract
0xD3	CALLOUTPUT	Call the contract based on put of another contract's c