TRUST

Trustmachine Blackpaper Draft version 0.6

Rebuild Trust With The Decentralized World Computer

TRUST-TECH.org

CONTENTS

TRUSTMACHINE: A GENERALIZED MATHEMATICALLY-SECURED QUASI-DETERMINISTIC DECENTRALIZED STATE-TRANSITION SYSTEM

TRUST-TECH LABORATORY
TRUST-TECH@PROTONMAIL.COM
0X6AEBE3BCE00AA58D

Abstract. Blockchain, which refers to the cryptographically-secured state-transition singleton system with shared state that guarantees high probability or even complete sureness of the validness of information storage, transfer and processing among multiple entities, demonstrated great utilities through bitcoin, ethereum, ripple and some other projects removing unnecessary, redundant and even hostile states via facilitating transactions between consenting individuals to enforce the smart contracts. Due to the low possibility or impossibility of absolute trust in face of complicated human or machine based informational operations, some dynamically balanced trusted order built upon the chaotic decentralized systems between mankind and cyber world has already become an urge.

Trustmachine implements an even more generalised yet purely functional future of trust regarding human and machine based informational activities as a whole and it is an ongoing project that wish to build a generalized mathematically-secured quasi-deterministic decentralized state-transition system based on the theory of typed λ-calculus and various technologies including the blockchain paradigm to rebuild trust across different informational existence in the objective reality to further remove the redundant and malicious states. It hereby not only needs to extend the previous special theory of blockchain paradigm into a general theory of trustmachine, but as well needs to scale the global state-transistion system inclusively and securely in a compositional manner, and construct more balanced and trusted infrastructure to support the entire ecosystem of trusted applications on the wild course of decentralization.

1. INTRODUCTION

Mankind itself is genetic programmed, bio-chemical supported and neural engineered continuously self-aware computer which has its own decentralized and complex nature of computation, storage and communication on the very perspective of classical or quantum information science, most fundamentally, he who has already built extraordinarily comprehensive trust-based intelligence-collective informational order which is still evolving. If we consider human and its ambiguous creations like financial systems and social systems as part of the universal informational system, then we should think of the theory of blockchain paradigm which has blockchain alone as a special theory of distributed singleton system instead of general.

The blockchain paradigm as a reverse-prone cryptographically-secured transactional singleton machine with shared-state has already been abstracted and generalized by Dr. Gavin Wood and Vitalik. Such technologies have unleashed great potentials through the utilities of hundreds of thousands of decentralized applications and have now been gradually mirroring the human-centric chaotic yet partial-digitalized real world to their own, hence they will eventually capture extensive financial actvities and informational actvities as their predecessors like bitcoin both on volume and complexness according to their design rationale of a turing-complete machine, despite the fact that they has their own scalability, security and other issues. Padoras box has been opened while the rapid adoption of such system into the human society whereas some blockchain infrastructure are only decentralizing instead of reconstructing the new dynamically achieved balance, and most of the rest are utilizing federated, hybrid, private and even fully centralized approaches. All these systems are eventually backed by different combinations of trust, and now heavily flooded with frauds and flaws. It is under such circumstances that the valid arc should be built to bridge together value and its underlying trust. Hence it is also necessary to implement a decomposition framework of trust for the entire system to limit fraws of absorbing trust or value through the public.

Trustmachine is a series of theories built upon the generalized yet purely functional mathematically-secured quasi-deterministic decentralized state-transition system that breaks and includes the traditional paradigm of blockchain. It tries to unify all computation forms with numerous securing and trusted approaches to build a distributed canonically recognized quasi-deterministic state-transistion system that has various state existence forms and representations, flexible and adaptive state-transtion abilities, diverse state data storage and communication methods to forge canonically-recognized shared-state. Blockchain as an original form of trustmachine paradigm would be further abstracted and generalized to explain the quasi-deterministic nature of most computation forms and guide our design of a near-perfect state-of-the-art quasi-deterministic dynamically-balanced trustmachine.

We believe that there exists no absolute decentralization, balance or determinism in most cases, but these are three ultimate goals for our project. All of the following properties become inevitable in our design of the future enhanced trustmachine: open-source licenced infrastructure; machine dependent language with small footprint, segfaults prevention and thread safety; all public APIs documented and all code reviewed by multiple peers; follow a pipelined N-ish-week release cycle; on-chain and multi-chain scalability; million txs per second distributed cache layer; hardened security against sophisticated attack; permission control with privacy guarantee and atomic selective transparency; robust plug and play consensus module; help scientific computing; proof of trust; compartmentation techniques to reduce attack factors; turing-complete execution environment; formally verifiable smart contract; state database with high throughput and querying flexibility; financial-friendly representation; javascript full stack compatibility; the trust arc; decomposition framework of trust; security-enforced continuous liquidity and realtime price discovery; cryptolaw and legal framework...etc. And there will be more and more.

With greater scalability, security, flexibility, determinism and inclusiveness implemented to reach the dynamiclly achieved balance, our publicly accessible trustmachine would soon be able to provide unprecedented opportunities for use as the trusted backbone of future global industry, commerce and economy where much greater complexities has been involved. We will discuss the theory of trustmachine, its design rationale, implementation and security issues, forefront ideas from the community, obstacles that we envisage and unlimited opportunities it provides.

1.1. **Driving Factors.** One of the ultimate goals of this project is to solve as many trust issues as possible in the human society. The quantum existence of the entire universe, the inderterministic nature of we ourselves and our natural desire of control and manipulation have already brought great uncertainties, risks, trust issues and unstabilities to most systems we created, even including blockchain.

Abstractly, trust issues may come from the unambigurousness or unstability of the state, invalid arc and loose connection between deterministic and indeterministic state-transition systems, asymmetry among different state-transition systems and incapability of systematic synchronicity, security issues over the state transition functions in the deterministic state-transition system, the misbehaviors in the indeterministic system, the uncertainty of the future states especially for value and risk, not enough scalability coverage and some security issues that result in the distortion of the ought-to-be tightly coupled deterministic state transition system.

In the previous blockchain paradigm, we find it almost deterministic itself to facilitate transactions and smart contracts between consenting individuals who would otherwise have no means to trust one another, but indeterministic when one is trying to couple with some other state transition systems which may cause complex trust issues, especially when the should-be-trusted blockchain is deliberately complexified to verify in a givin time, sometimes even partially or completely unverifiable, compromised, manipulated, close-sourced, or completely centralized, just like a blackbox. Frauds, flaws, crimes and manipulation spread fast where complexities, uncertainties and blackbox exists. We assume that trust issues can not be thoroughly eliminated, but the boosting of technology-rooted trust economy is a clear trend. We would not only depend on the transparency, stability, security and determinism of the trusted state transition system, but in urgent need of more publicly verifiable yet secure and deterministic infrastructures as well. Sometimes where trust can not be established in time through technology, we ourselves should stand naked in the sunlight and dare face all criticisms before the trustmachine has been eventually built to cover all the issues.

Since trust can not be easily or ultimately achieved in the current society, and rare as frauds become flooded, the need of trust would become a must. I wish to design trustmachine such that most trust issues could be solved in a divide-and-conquer effort and trustmachine itself is an ongoing project that would rapidly become convenient in usage, secure in design, large in scale and quasi-deterministic through time.

1.2. **Previous Work.** Blockchain is a special case of trustmachine that resolves trust issues without centralized parties to achieve decentralization. But seldom has tried carefully to define what blockchain is, what trust is and what decentralization is in a fomal way. Blockchain, first originated from Bitcoin when Nakamoto [2008] proposed a peer-to-peer electronic cash system to the double spending problem via hashing the network timestamp transactions into a hash-based proof-of-work-secured ongoing chain of blocks. Nakamoto adopted the idea provided by Dwork and Naor [1992], and Back [2002] of the usage of transimitting cryptographic proof of computaional expenditure secured value-signal over the internet, and later referenced the first example of such idea from Vivek Vishnumurthy and Sirer [2003].

Nakamoto believed that any trusted thrid party would suffer from the inherent weaknesses of the trust based model with the possibility of reversing the transactions. Thus two willing parties should transact directly with each other without the need for a trusted third party, and they should rely on a peer-to-peer distributed timestamp server to generate computational proof and form chronological order of transactions, assuming the system would stay secure as long as honest nodes collectively control more computational power than any cooperating group of attacker nodes. The truth is Satoshi Nakamoto actually had not ultimately eliminated the need of trusted third party, and instead, he created one special form of trusted thrid party which is Bitcoin itself if we consider the economically-incentivized proof-of-work-secured distributed system as a whole. Therefore trust, as well as the trusted thrid parties, is impossible to be eleminated, it merely changed into a more deterministic existence, sometimes, mathematics.

Buterin [2013a] then first proposed the kernel of the design of Trustmachine, an idea of Turing-complete language with effectively unlimited inter-transaction stroage capability. Inspired by Buterin, Wood [2014], the mastermind of Trustmachine, since the very earliest of 2013, drafted the yellow paper of a secure decentralized transaction ledger which became the cornerstone of the theory of blockchain. His effort of abstracting the blockchain paradigm into a generalised cryptographically-secured transactional singleton machine with shared state has been proven right through time when the design of Trustmachine gradually achieved enormous adoption in the developper community. The idea of algorithmic enforcement of agreements proposed by Szabo [1997] and Miller [1997] also led to his design of smart contracts and the programmable economic phenomena sprang up globally. Wood [2017] is now working on Polkadot, a heterogeneous multi-chain framework to resolve extensibility issues and scalability issues horizontally where some other properties like privacy, isolatability, developability, governance and applicability has also been greatly emphasized to eventually connect trust among divergent types of consensus systems. And he invented the idea Fisherman in his design of Polkadot which could be a major breakthrough in removing malicious states via incentivization. Similarly, Jae Kwon [2017] is another attempt to implement interoperability of blockchains trying to scale horizontally with Hubs and Zones, having Tendermint at its core. Tendermint was proposed by Kwon [2014], and it is a partially synchronous BFT consensus module derived from the DLS consensus algorithm. The blockchain paradigm as well as the iterative and heterogeneous nature of the Blockchain of The Blockchain paradigm might work well to effectively scale the entire state-transition system and form a rather deterministic infrastructure.

Tezos, proposed by Goodman [2014], is trying to implement a generic and self-amending crypto-ledger with purely functional programming language as OCaml from ground up. It is designed to instantiate any blackchain based ledger and the operations of a regular blockchain are implemented as a purely functional module abstracted into a shell responsible for network operation. It begins with a seed protocol defining a procedure for stateholders to approve amendments to the protocol, including amendments to the voting procedure itself. The seed protocol is based on a pure proof-of-stake system and designed to support Turing complete smart contracts.

Ouroboros, proposed by Aggelos Kiayias and Oliynykov [2017], is the first blockchain protocol based on proof of stake with rigorous security guarantees. The security properties for the protocol is established to be comparable to those achieved by the bitcoin blockchain protocol. As the protocol provides a proof of stake blockchain discipline, it offers qualitative efficiency advantages over blockchains based on proof of physical resources (e.g., proof of work). It also presents a novel reward mechanism for incentivizing Proof of Stake protocols and it has been proved that, given this mechanism, honest behavior is an approximate Nash equilibrium, thus neutralizing attacks such as selfish mining. Initial evidence of the practicality of the protocol has also been presented in real world settings by providing experimental results on transaction confirmation and processing. A fundamental problem for PoS-based blockchain protocols is to simulate the leader election process. In order to achieve a fair randomized election among stakeholders, entropy must be introduced into the system, and mechanisms to introduce entropy may be prone to manipulation by the adversary. For instance, an adversary controlling a set of stakeholders may attempt to simulate the protocol execution trying different sequences of stakeholder participants so that it finds a protocol continuation that favors the adversarial stakeholders. This leads to a so called grinding vulnerability, where adversarial parties may use computational resources to bias the leader election. Ouroboros presents a provably secure proof of stake system with a rigorous security analysis. The model of the problem of realizing the PoS-Based blockchain protocol has been formalized and a noval blockchain protocol based on POS is described.

Lightning Network is a proposal from Joseph Poon [2016]. Its implementation of Revocable Sequence Maturity Contracts (RSMCs) and Hashed Timelock Contracts (HTLCs) with bidirectional payment channels which allow payments to be securely routed across multiple peer-to-peer payment channels without constantly interacting with the blockchain. This also allows the formation of a network where any peer on the network can pay any other peer even if they don't directly have a channel open between each other. Lightning Network has aroused the spirit of innovation in Bitcoin community and Segregated Witness, a significant breakthrough at the time, was proposed by Wuille [2016] both to pave way for Lightning Network and to resolve dozens of security and performance issues, including transaction malleability, related with Bitcoin. Later projects like Aeternity, proposed by Zackary Hess [2017], visioned a future where general smart contracts could be executed in an off-chain state channel coupling with decentralized oracle system. However, debating in the community occurred since such system would ultimately rely on the trusted blockchain infrastructure to secure all sub-state transitions. Centralization, security, scalability, implementation and trust issues are inevitable in such a sub-state transition system when handling massive amount of transactions, great number of channels and huge hubs. Every change occurred off-chain is the result of some sub-state transition in the state channel and every such scalar, mostly vector, would take certain amount of time, which is the subset of the locked-up time window of the state channel.

Plasma, visioned by Poon and Buterin [2017], is a proposed framework for incentivized and enforced execution of smart contracts which is scalable to a significant amount of state updates per second (potentially billions) enabling the blockchain to be able to represent a significant amount of decentralized financial applications worldwide. These smart contracts are incentivized to continue operation autonomously via network transaction fees, which is ultimately reliant upon the underlying blockchain to enforce transactional state transitions. A method for decentralized autonomous applications to scale to process not only financial activity, but also construct economic incentives for globally persistent data services, which may produce an alternative to centralized server farms has been proposed. Plasma is composed of two key parts of the design: Reframing all blockchain computation into a set of MapReduce functions, and an optional method to do Proof-of-Stake token bonding on top of existing blockchains with the understanding that the Nakamoto Consensus incentives discourage block withholding. This construction is achieved by composing smart contracts on the main blockchain using fraud proofs whereby state transitions can be enforced on a parent blockchain. Blockchains are composed into a tree hierarchy, and treat each as an individual branch blockchain with enforced blockchain history and MapReducable computation is committed into merkle proofs. By framing ones ledger entry into a child blockchain which is enforced by the parent chain, one can enable incredible scale with minimized trust (presuming root blockchain availability and correctness). The greatest complexity around global enforcement of non-global data revolves around data availability and block withholding attacks, Plasma has mitigations for this issue by allowing for exiting faulty chains while also creating mechanisms to incentivize and enforce continued correct execution of data. As only merkleized commitments are broadcast periodically to the root blockchain during non-faulty states, this can allow for incredibly scalable, low cost transactions and computation. Plasma enables persistently operating decentralized applications at high scale.

Zcash, first originated from Zerocoin, is proposed by an elite team of cryptographers, Ben-Sasson et al. [2014]. Compared with Zerocoin, Zcash has brought brand new features, bunch of optimazations and extensions, and many security fixes regarding transaction malleability and indistinguishability. It leverages the recent advances in zero-knowledge Succinct Non-interactive Arguments of Knowledge (zk-SNARKS) and the fruit is a full-fledged ledger-based digital currency with strong privacy guarantees. A DAP(decentralized anonymous payment scheme) is achieved and selective transparency becomes the new standard of privacy protection. It works well with the bitcoin utxo model and Zcashs genesis block started with trusted-setup. Equihash as a asymmetric memory-hard proof-of-work algorithm based on the general birthday problem was ultilized by zcash and it also drew worlds attention. It may very well lead to the innovation of new FPGA or ASIC devices that have certain memory capacity and throughput. The most excite part however, is its cooperation with Trustmachine to bring zk-SNARKS to selectively anonymify smart contracts with precompiled code. Baby ZOE, as part of Alchemy project, also brings great potentials to bridge together these two projects. We believe that privacy technology would always be the major part of a trustmachine against certain attack factors of our end clients.

Ripple, proposed by Schwartz et al. [2014], is known as the first federated approach for efficient currency clearing and real-time settlement of cross-border payments among banks and merchants. And in 2017, Ripple proposed Inter Ledger Protocol(ILP) to achieve horizontal scalability in the federation where there exists multiple bank ledgers. Delegated Proof-of-Stake(DPOS), proposed by Larimer [2014], is another consensus-level vertical scalability solution and it had once adopted by Ripple. Compared with normal POW and POS approaches, DPOS has obvious performance upgradation and less incentivation expenditures on securing the network, but at the cost of being more centralized, less secured and less trusted due to the high dependence of node election and node uptime. Such design brought heated discussions in the community at that time and none persuaded the other side. We believe that every security model in the blockchain has its own benefits and drawbacks especially when the use cases and issues to eliminate varies on different perspectives. Rather than a federated approach from Ripple, Skycoins opinion dynamics based consenus proposed by Chen et al. [2015] is a new prototype that wish to minimum the cost and maximum the efficiency of securing the network via mapping the dynamics of the human-based opinions to the consensus layer without any centralized coordinators or global clock. And Trustmachines Casper, proposed by Zamfir and L.G.Meredith [2016], wishes to build economics-centered consensus to absorb opinions and incentivize good behaviors as well as disincentivize bad behaviors.

Quorum is a private and permissioned blockchain based on the official Golang implementation of the Trustmachine protocol by JPMorganChase [2016]. It uses a majority voting based consensus algorithm based RBFT, the voting role allows a node to vote on which block should be the canonical head at a particular height. Quorum also achieves Data Privacy through the introduction of a new private transaction identifier and segmentation. Cryptography is applied to the data in the transactions, which everyone sees on the blockchain. Segmentation is applied to each nodes local state database which contains the contract storage and is only accessible to the node. Such private blockchain works well in securing the state transition in a rather controlled and centralized environment when the majority of the trust is backed the underlying brand and bank, which ultimately comes from the credibility and predictability through its financial and human-relational activities.

Hyperledger is another open source collaborative effort created to advance cross-industry blockchain technologies. It is a global collaboration, hosted by The Linux Foundation, including leaders in finance, banking, IoT, supply chain, manufacturing and technology. There are several blockchain frameworks hosted with Hyperledger: Burrow, Fabric, Iroha, Sawtooth and Indy. Burrow is a permissionable smart contract machine. Burrow provides a modular blockchain client with a permissioned smart contract interpreter built in part to the specification of the Trustmachine Virtual Machine (TVM). Fabric, initially contributed by Digital Asset and IBM, is intended as a foundation for developing applications or solutions with a modular architecture, and it allows components, such as consensus and membership services, to be plug-and-play, it also leverages container technology to host smart contracts called chaincode that comprise the application logic of the system. Iroha, initially contributed by Soramitsu, Hitachi, NTT Data and Colu, features a simple construction; modern, domain-driven C++ design, emphasis on mobile application development and a new, chain-based Byzantine Fault Tolerant consensus algorithm, called Sumeragi. In Sawtooth the data model and transaction language are implemented in a transaction family and users would build custom transaction families that are sufficient for building, testing and deploying a marketplace for digital assets, it also have a hardware-level Proof of Elapased Time(PoET) as its consensus algorithm to secure the entire infrastructure securely. Indy provides tools, libraries, and reusable components for providing digital identities rooted on blockchains or other distributed ledgers so that they are interoperable across administrative domains, applications, and any other silo. And Corda is another blockchain-inspired distributed ledger platform developped by R3cev with the joint effort of the largest blockchain-oriented financial consortium. It requires notaries to validate transactions and merge several blockchains via establishing two-way connectivity between their nodes after configuring each side to trust each others notaries and certificate authorities and may use secure signing devices to guard against consensus level attacks.

EOS, proposed by Bytemaster [2017], explained that public blockchains should have properties of supporting millions of users, free usage, easy upgrades and bug recovery, low latency, sequential and parallel performance. He emphasizes much on DPOS algorithm, accounts and role based permission management, deterministic parallel execution of applications, token model and resource usage, governance, scripts and virtual machines, inter-blockchain communication and detailed about how everything works in the design of the EOS framework. About DPOS consensus algorithm, he proposes that a transaction should be 99.9% certainty after an average of 1.5 seconds from time of broadcast. And EOS would require every transation to include the hash of a recent block header to make it difficult to forge counterfeit chains. About permission defination, he suggested that every post on the blockchain should have some permission control abilities like hard-coded named permission levels, named message handler groups, permission mapping and permission evaluation. About permission controlling, he proposed that messages should have mandatory delay which depends upon developers and users to randomize the attacking factor against time or reversing related attacks, the message handlers should be read-only. About performance, he suggested that an application should be deterministically executed in parallel. For convience of usage and scalability, transaction messages should be delivered to and accepted by multiple accounts atomically, blockchain state should be partially evaluated so that everyone dont have to run everything if they only need a small subset of the applications, and each block producer should make their own subjective measurement of the computational complexity and time required to process a transaction. He then discussed token model and resource usage: objective and subjective measurements, receivers pays, delegating capacity, separating transaction costs from token value, state storage costs, block rewards, community benefit applications. About Governance, some special requirements from the users might be freezing accounts, changing account code, establishing a peer-to-peer terms of service agreement or a binding contract among those users who sign it, and defines a process by which the protocol as defined by the canonical source code and its constitution, especially for emergency changes. About scripts and virtual machines, there should be schema defined messages which allows seamless conversion between binary and JSON representation of the messages. Authentication should be separated from the application. The whole architecure should be virtual machine independent. He also emphasized much on inter blockchain communication and Merkle proofs for light client validation to gain horizonal scalability, and the latency of interchain communication is minimized to 45 seconds with 3 second blocks and 21 producers DPOS. As for completeness of the blockchain, he proposed that via assigning a sequence number to every message delivered to every account, a user can use these sequence numbers to prove that all messages intended for a particular account have been processed and that they were processed in order.

2. THE GENERAL THEORY OF TRUSTMΛCHINE

Trustmachine is the generalized theory of distributed mathematically-secured state transition systems that execute as typed λ-calculi machines and has transactions to represent valid transitions between two states which stores arbitrary data of the states of some other state transition systems.

2.1. Turing Machine, λ-Calculus And Church-Turing Thesis.

A turing Machine is a mathematical model of a theoretical computing machine thought of by the mathematician Alan Turing in

1937 that can use a predefined set of rules to determine a result from a set of input variables. To complete the program, the state changes in each of its step during the execution of the program on the machine. Despite its simplicity, the machine can simulate any computer algorithm, no matter how complicated it is. In the quantum world where non-determinism is natural, probabilistic, non-deterministic and even quantum turing machine grows solid in theory and can efficiently simulate any realistic model of computation theoretically.

λ-calculus is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution. It is a universal model of computation that can be used to simulate any single-taped Turing machine and was first introduced by mathematician Alonzo Church in the 1930s as part of his research of the foundations of mathematics. λ-calculus consists of constructing λ-terms and performing reduction operations on them. λ-calculus is Turing complete, that is, it is a universal model of computation that can be used to simulate any single-taped Turing machine. In typed λ-calculus, functions can be applied only if they are capable of accepting the given input's "type" of data. And a typed λ-calculus is a typed formalism that uses the symbol λ to denote anonymous function abstraction. In this context, types are usually objects of a syntactic nature that are assigned to λ-terms; the exact nature of a type depends on the calculus considered. The -calculus is a universal model of computation. This was first observed by Milner in his paper "Functions as Processes", in which he presents two encodings of the λ-calculus in the -calculus.

In computability theory, the Church-Turing thesis (also known as computability thesis) is a hypothesis about the nature of computable functions. Church and Turing proved that these three formally defined classes of computable functions coincide: a function is λ-computable if and only if it is Turing computable if and only if it is general recursive. After the development of complexity theory, probabilistic Turing machine is thought to be able to efficiently simulate any realistic model of computation according to Feasibility Thesis.The word 'efficiently' above means up to polynomial-time reductions. This thesis was originally called Computational Complexity-Theoretic Church-Turing Thesis by Ethan Bernstein and Umesh Vazirani (1997). The Complexity-Theoretic Church-Turing Thesis, then, posits that all 'reasonable' models of computation yield the same class of problems that can be computed in polynomial time. Assuming the conjecture that probabilistic polynomial time (BPP) equals deterministic polynomial time (P), the word 'probabilistic' is optional in the Complexity-Theoretic Church-Turing Thesis. Eugene Eberbach and Peter Wegner later invalidated the Complexity-Theoretic Church-Turing Thesis, and they claimed in Quantum Complexity-Theoretic Church-Turing thesis with the term Super-Turing Computation: "A quantum Turing machine can efficiently simulate any realistic model of computation."

In mathematics and physics, a deterministic system is a system in which no randomness is involved in the development of future states of the system. A deterministic model will thus always produce the same output from a given starting condition or initial state. However, a deterministic system is a conceptual model of the philosophical doctrine of determinism applied to a system for understanding everything that has and will occur in the system, based on the physical outcomes of causality. In a deterministic system, every action, or cause, produces a reaction, or effect, and every reaction, in turn, becomes the cause of subsequent reactions. The totality of these cascading events can theoretically show exactly how the system will exist at any moment in time, which is rather valuably rare, where random quantum events dominate the world. Most systems are non-deterministic due to the fact of their being the subsystems or derivative systems of some non-deterministic systems , hence would be trustless because of their inherent uncertainties and risks.

We envisioned a future where trustmachine could be a solid series of theories of building systems that follows church-turing thesis when strongly coupled with the quasi-deterministic mathematically-secured state-transition systems designed under the theory of typed λ-calculus. Our future work would focus on extending this theory and building up certainties through the guideline of the theory against all uncertainties and bring the world trust as our persevere goal.

2.2. **The Trustmachine Paradigm.** Trustmachine is a type-safe generalized mathematically-secured quasi-deterministic state-transition system that is built upon multiple coupled cryptographically-secured transactional state-transition singleton systems with shared state as well as their sub-state transition systems and all other secured systems derived or coupled. The uppercase Greek letter Λ is used in λ-expressions and λ-terms to denote binding a variable in a function of trustmachine and the term trustmachine can be viewed as the superset of all trusted or quasi-deterministic systems that is both distributed and strongly coupled with or derived from the mathematically-secured state transititon systems.

We first start with the syntax of the pure typed λ-calculi which is defined as follows:

$$e ::= x \mid \lambda x{:}\tau.e \mid e\, e \mid c \tag{1}$$

where c is a term constant.That is, variable reference, abstractions, application, and constant. A variable reference x is bound if it is inside of an abstraction binding x. A term is closed if there are no unbound variables. The type rules are given as:

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \tag{2}$$

$$\frac{c \text{ is a constant of type } T}{\Gamma \vdash c : T} \tag{3}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\lambda x : \sigma.\ e) : (\sigma \to \tau)} \tag{4}$$

$$\frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\ e_2 : \tau} \tag{5}$$

Note that $\Gamma \vdash e : \sigma$ indicates e is a term of type σ in context Γ, which is typing environment. Then we have: a) If x has type σ in the context, we know that x has type σ; b)term constants have the appropriate base types; c)if, in a certain context with x having type σ, e has type τ, then, in the same context without x, $\lambda\ x : \sigma.\ e$ has type $\sigma \to \tau$; d)If, in a certain context, e_1 has type $\sigma \to \tau$, and e_2 has type σ , then $e_1\ e_2$ has type τ; e)If, in a certain context with x having type σ, e has type τ, then, in the same context without x, $\lambda\ x : \sigma.\ e$ has type $\sigma \to \tau$; f)If, in a certain context, e_1 has type $\sigma \to \tau$, and e_2 has type σ, then $e_1\ e_2$ has type τ.

π-calculi can be fully encoded with λ-calculi and it is a process calculus which allows channel names to be communicated along the channels themselves, and in this way it is able to describe concurrent computations whose network configuration may change during the computation. Let X be a set of objects called names and formally:

$$P, Q, R ::= x(y).P \mid \overline{x}\langle y\rangle.P \mid P|Q \mid (\nu x)P \mid\ !P \mid 0 \tag{6}$$

The syntax above defines the following process: Receive on channel x, bind the result to y, then run P; Send the value y over channel x, then run P; run P and Q simultaneously; Create a new channel x and run P; Repeatedly spawn copies of P; Terminate the process. Similarly, π-calculi designed systems can be stongly typed and we will explore this in later sections. Similarly, π-calculi can be strongly typed and we will explore the polymorphic features and the usage of λ-calculi and π-calculi in our design of trustmachine in later sections.

The blockchain paradigm, as previously illustrated by Dr. Gavin Wood, works as follows and is typed-friendly by design: The blockchain is initiated from a genesis state and all the transactions that carry changes of the state would be incrementally executed to be morphed into some final state. It is this final state that would be publicly accepted as the canonical shared state through consensus algorithm. Formally:

$$\boldsymbol{\sigma}_{t+1} \equiv \Upsilon(\boldsymbol{\sigma}_t, T) \tag{7}$$

where Υ is the state transition function. Υ allows components to carry out arbitrary computation, while $\boldsymbol{\sigma}$ allows components to store arbitrary state between transactions. Transactions are collated into blocks; blocks are chained together using a cryptographic hash as a means of reference. Blocks function as a journal, recording a series of transactions together with the previous block and an identifier for the final state. In real world design, the final state itself doesnt have to be storedthat would be far too big. And in some special design cases, the transaction series will also be punctuated with incentives for nodes to form consensus. This incentivisation takes place as a state-transition function, adding value to a nominated account. Forming consensus is the process of dedicating effort (working) to bolster one series of transactions (a block) over any other potential competitor block. It is achieved thanks to a cryptographically secure proof. Not only proof-of-work, consensus can be achieved in multiple ways and is discussed in detail in section 16. Formally:

$$\boldsymbol{\sigma}_{t+1} \equiv \Pi(\boldsymbol{\sigma}_t, B) \tag{8}$$

$$B \equiv (..., (T_0, T_1, ...)) \tag{9}$$

$$\Pi(\boldsymbol{\sigma}, B) \equiv \Omega(B, \Upsilon(\Upsilon(\boldsymbol{\sigma}, T_0), T_1)...) \tag{10}$$

Where Ω is the block-finalisation state transition function (a function that rewards a nominated party); B is this block, which includes a series of transactions amongst some other components; and Π is the block-level state-transition function.

3. DESIGN RATIONALES

3.1. **Introducing TRUSTMΛCHINE.** Trustmachine is an ongoing project that ultimately aims to resolve trust issues and forge determinism. We are building a system that could be mathematically-proved secure, publicly verified and constantly validated. It has multiple projects in parallel, each of which has a different sub direction, and all is crucial to exist, support and extend. In this section, we will explore the design rationales of trustmachine, and the principles that we stick to.

3.2. **Security first.** We believe that trust ultimately comes from the quasi-determinism of the state transition system which is secured mathematically. As long as the cost of securing the system is substantically lower than the cost of implementing a successful attack, the system is safe and thus trust-worthy. That's why security first is our major principle in designing the entire system of trustmachine.

3.2.1. *Code Review.* We strive for 100% important logic unit-tested and code reviewed by multiple peers to guarantee security and trustworthiness. We would combine tool-based peer reviewing techniques with specialized machine-generated analyzers to detect defects and investigate functionality, style/coding guidelines, architecture, exception handling, timing, validation and hardware in a continuous way. And some formal methods will be applied to achieve above-industrial safety standards. Certain integrated analysis framework and code review workflow will be designed specifically for trustmachine project.

3.3. **Open Source.**

3.3.1. *GPL Licence.* We are using GNU General Public License 3.0 for all the softwares under project trustmachine to gurantee end users the freedom to run, study, share and modify the software. GPL 3.0 is copyleft license, which means that derivative work can only be distributed under the same license terms. GPL 3.0 is hostile to patents and it assures that patents cannot be used to render the program non-free and would never be allowed to restrict development and use of software on general-purpose computers.

3.4. **High Efficiency and Throughput.**

3.4.1. *Small Footprint And Machine Dependent.* While the modules of the trustmachine execute, the runtime memory it requires will be tightly restricted and controlled. Small footprint means minimized active memory regions like code segment containing (mostly) program instructions (and occasionally constants), data segment (both initialized and uninitialized), heap memory, call stack, plus memory required to hold any additional data structures, such as symbol tables, debugging data structures, open files, shared libraries mapped to the current process, etc., that the program ever needs while executing and will be loaded at least once during the entire run. The language of constructing the trustmachine and the language executed within trustmachine virtual machine will be machine dependent to achieve maximum efficiency.

3.4.2. *Latency, Concurrency And Throughput.* Latency is the time for a transaction to confirm. One-way lantency and round-trip lantency are two major concerns if millions of users are interacting with the trustmachine simultaneously. The maximum throughput is the maximum rate at which the trustmachine can confirm transactions. This number is constrained by the maximum mana limit and the inter-block time. Huge throughput as well as low lantency should be achieved through multiple approaches combined, one of which is to decompose problems into order-independant or parially-ordered componenets or units, and execute the concurrent units in parallel.

3.5. **Compatability.**

3.5.1. *Backward Compatability.* Prior to the trustmachine project, there are several decent projects in blockchain industry trying to eliminate trust issues on different perspectives. We are looking for backward compatability to extend the developpers community and ease the shifting from an unsecured platform to a much secured and trusted one.

3.5.2. *Extensive Libraries.* Libraries are extensively listed in an elegantly sorted way to be easily referenced. Trust issues are complex in reality, and we hope libraries like such could help with solving the trust issues conveniently on top of the shoulders of the giants. Trustmachine, as a secure general purpose distributed system, libraries may also boost the development of some much more complex projects.

3.5.3. *Maximum Inclusiveness.* Trustmachine project looks for maximum inclusiveness to absorb as much theories and innovations in the open source community as possible to lead the research and development of a giant series of projects which will focus on the unchangable common goal of mankind, which is to build a mathematically-secured quasi-deterministic decentralized state transition system that is safe and certain enough to be canonically interacted with, in short, to rebuild trust.

3.6. **Development.**

3.6.1. *Distributed Version Management.* Git is the version control system that we are using for tracking changes of the files in the project and coordinating distributed development on those files. It has strong support for non-linear development, agility in distributed development, compatibility with extant systems and protocols, efficient handling of large projects, cryptographic authentication of history, toolkit-based design, pluggable merge strategies, garbage accumulates until collected, periodic explicit object packing and it also snapshots directory trees of files.

3.6.2. *Pipelined Release Circle.* To give users the latest improvements as quickly as possible and more certainty of development and upgrading, the trustmachine project is following a pipelined N-ish-week release cycle and maintain three different releases at any one time. The stable release which is our most mature and tested software. The beta release has additional features and better performance but may yet have quirks and issues to be fixed. Finally, the nightly release is our cutting-edge software build but comes with a strong caveat against using it for managing anything of value. As the schedules become more frequent, the competitiveness in the marketplace becomes better. And in the meantime, we are also applying continuous delivery as our strategy to deliver new features to users as fast and efficiently as possible to enable a constant flow of changes into production via an automated software production line.

3.7. **Five Principles.** In the blackpaper of trustmachine, we stick to five major principles that form the backbone of our future project.

3.7.1. *Security First Principle.* Security first is the fundamental layer of all principles and it is the ultimate one that shape the current project. Threats of trustmachine might come from several completely different angles which wish to destroy or manipulate trust of the system and even create flaws. Sometimes we may even turn to some active methods that act automatically to defend the system.

3.7.2. *Freedom Of Speech Principle.* Freedom of speech is the second principle and trustmachine is a completely open project that contributes all the information to the trust community including papers and source code. Advanced technical approches will be applied to make sure that project is a collective effort with the right of absolute freedom.

3.7.3. *Objectively Neutral Principle.* Objectively neutral principle is the one that derived from the idea of "code is law". Trustmachine is an ongoing project with an open set of code that behaves neutrally and objectively, despite the fact that such decentralized computers are evolving rapidly and would soon be out of control.

3.7.4. *Inclusive Collaboration Principle.* Collaboration should be inclusive since day one of the project. We respect every effort of everyone who is willing to share his ideas to the world and with such principle instead of exclusively competition, we hope to bring together the brilliant minds and push forward the common goal of creating a costless trusted future.

3.7.5. *Radical Innovation Principle.* Trustmachine never stops moving forward. We are at the dawn of the trust evolution, the previous four principles are the guarantees of inspiring and implementing radical innovations that could reshape everything. This principle, at last, means our determined view of accelerating changes.

4. TYPOLOGICAL CONVENTIONS

Typographical conventions for the formal notations have been used and some of which are quite particular to the present work:

(A)Bold Lowercase Greek Letters denote the set of highly structured, 'top level', state values.
e.g. $\boldsymbol{\sigma}$, the state-mapping; $\boldsymbol{\mu}$, the machine-state.

(B)Upper-Case Greek Letters denote functions operating on highly structured values.
e.g. Υ, the state transition function.

(C)Uppercase Letters denote most functions; Subscripted Uppercase Letters denote specialised variants of functions,
e.g. C, the general cost function; C_{SSTORE}, the cost function for the SSTORE operation.

(D)Typewriter Text are defined for specialised and possibly externally defined functions,
e.g. `KEC`, the Keccak-256 hash function; `KEC512`, the Keccak 512 hash function.

(E)Upper-Case Letters denote tuples; Subscripted Upper-case Letters refer to an individual component; the form of the Subscript is used to denote its type; Uppercase Subscripts refer to tuples with subscriptable components,
e.g. T, a transaction; T_n, the nonce of said transaction.

(F)Normal Lower-Case Letters denote scalars and fixed-size byte sequences (or, synonymously, arrays); Greek Letters denote those with special meaning,
e.g. n, a transaction nonce; δ, the number of items required on the stack for a given operation.

(G)Bold Lower-Case Letters denote arbitrary-length sequences; Bold Uppercase Letters denote particularly important values,
e.g. $\mathbf{o}$, the byte sequence given as the output data of a message call.

(H)$\mathbb{P}$ is the set of scalars which are positive integers; $\mathbb{B}$ is the set of all byte sequences, formally defined in Appendix B.

(I)Subscript of a set indicates the sequences within is restricted to a particular length,

e.g. $\mathbb{P}_{256}$ is the set of all positive integers smaller than 2^{256}; $\mathbb{B}_{32}$ is the set of all byte sequences of length 32.

(J)Square Brackets are used to index into and reference individual components or subsequences of sequences,

e.g. $\boldsymbol{\mu}_{\mathbf{s}}[0]$ denotes the first item on the machine's stack.

(K)Ellipses are used to specify the intended range and to include elements at both limits for subsequences,

e.g. $\boldsymbol{\mu}_{\mathbf{m}}[0..31]$ denotes the first 32 items of the machine's memory.

(L)Square Brackets are used to reference an individual account of tuples from a sequence of accounts in the case of the global state $\boldsymbol{\sigma}$.

(M)Placeholder $\Box$ denotes the unmodified 'input' value within a given scope for definition when considering variants of existing values; $\Box'$ denotes modified and utilisable value; $\Box^*$, $\Box^{**}$ &c denote intermediate values.

(N)Alpha-Numeric Subscripts denotes intermediate values on very particular occasions to maximise readability and only if unambiguous in meaning.

(O)Function f^* denotes an element-wise version of the function mapping of the existing function f, instead between sequences. It is formally defined in section 5.3.

(P) ℓ evaluates to the last item in the given sequence:

$$\ell(\mathbf{x}) \equiv \mathbf{x}[\|\mathbf{x}\| - 1] \tag{11}$$

5. STATE, TRANSACTION AND BLOCK

The meaning of the state, the transaction and the block will be discussed in more detail in later sections.

5.1. The State.

5.1.1. *State-Mapping.* State-Mapping is a mapping between addresses and account states. Addresses are 160-bit identifiers. Account states is a data structure serialised as RLP, see Appendix B. State-Mapping is not stored on the blockchain, but assumed to be in a modified Merkle Patricia tree (*trie*, see Appendix D).

5.1.2. *State-Database.* State-Database is the underlying database that maintains State-Mapping, which is in a trie of bytearrays to bytearrays. Benefits: a) The hash can be used as a secure identity for the entire system state, because the root node of this structure is cryptographically dependent on all internal data; b)It allows any previous state (whose root hash is known) to be recalled by simply altering the root hash accordingly, while the data structure is immutable; c) Reverting to old states trivially is possible since all such root hashes are stored in the blockchain.

5.1.3. *Account-State.* The account state comprises the following four fields:

nonce: A scalar value equal to the number of transactions sent from this address or, in the case of accounts with associated code, the number of contract-creations made by this account. For account of address a in state $\boldsymbol{\sigma}$, this would be formally denoted $\boldsymbol{\sigma}[a]_n$.

balance: A scalar value equal to the number of Wei owned by this address. Formally denoted $\boldsymbol{\sigma}[a]_b$.

storageRoot: A 256-bit hash of the root node of a Merkle Patricia tree that encodes the storage contents of the account (a mapping between 256-bit integer values), encoded into the trie as a mapping from the Keccak 256-bit hash of the 256-bit integer keys to the RLP-encoded 256-bit integer values. The hash is formally denoted $\boldsymbol{\sigma}[a]_s$.

codeHash: The hash of the TVM code of this account—this is the code that gets executed should this address receive a message call; it is immutable and thus, unlike all other fields, cannot be changed after construction. All such code fragments are contained in the state database under their corresponding hashes for later retrieval. This hash is formally denoted $\boldsymbol{\sigma}[a]_c$, and thus the code may be denoted as $\mathbf{b}$, given that $\texttt{KEC}(\mathbf{b}) = \boldsymbol{\sigma}[a]_c$.

The underlying set of key/value pairs stored within the trie is used as references instead of the root hash, formally:

$$\texttt{TRIE}\big(L_I^*(\boldsymbol{\sigma}[a]_{\mathbf{s}})\big) \equiv \boldsymbol{\sigma}[a]_s \tag{12}$$

The collapse function for the set of key/value pairs in the trie, L_I^*, is defined as the element-wise transformation of the base function L_I, given as:

$$L_I\big((k, v)\big) \equiv \big(\texttt{KEC}(k), \texttt{RLP}(v)\big) \tag{13}$$

where:

$$k \in \mathbb{B}_{32} \quad \wedge \quad v \in \mathbb{P} \tag{14}$$

It shall be understood that $\boldsymbol{\sigma}[a]_{\mathbf{s}}$ is not a 'physical' member of the account and does not contribute to its later serialisation.

If the **codeHash** field is the Keccak-256 hash of the empty string, i.e. $\boldsymbol{\sigma}[a]_c = \texttt{KEC}\big(()\big)$, then the node represents a simple account, sometimes referred to as a "non-contract" account.

Thus we may define a state-mapping collapse function L_S:

$$L_S(\boldsymbol{\sigma}) \equiv \{p(a) : \boldsymbol{\sigma}[a] \neq \varnothing\} \tag{15}$$

where

$$p(a) \equiv \big(\texttt{KEC}(a), \texttt{RLP}\big((\boldsymbol{\sigma}[a]_n, \boldsymbol{\sigma}[a]_b, \boldsymbol{\sigma}[a]_s, \boldsymbol{\sigma}[a]_c)\big)\big) \tag{16}$$

This function, L_S, is used alongside the trie function to provide a short identity (hash) of the state mapping. We assume:

$$\forall a : \boldsymbol{\sigma}[a] = \varnothing \ \vee \ (a \in \mathbb{B}_{20} \ \wedge \ v(\boldsymbol{\sigma}[a])) \tag{17}$$

where v is the account validity function:

$$v(x) \equiv x_n \in \mathbb{P}_{256} \wedge x_b \in \mathbb{P}_{256} \wedge x_s \in \mathbb{B}_{32} \wedge x_c \in \mathbb{B}_{32} \tag{18}$$

5.2. **The Transaction.** A transaction (formally, T) is a single cryptographically-signed instruction constructed by an actor externally to the scope of Trustmachine. While it is assumed that the ultimate external actor will be human in nature, software tools will be used in its construction and dissemination[1]. There are two types of transactions: those which result in message calls and those which result in the creation of new accounts with associated code (known informally as 'contract creation'). Both types specify a number of common fields:

nonce: A scalar value equal to the number of transactions sent by the sender; formally T_n.

manaPrice: A scalar value equal to the number of Wei to be paid per unit of *mana* for all computation costs incurred as a result of the execution of this transaction; formally T_p.

manaLimit: A scalar value equal to the maximum amount of mana that should be used in executing this transaction. This is paid up-front, before any computation is done and may not be increased later; formally T_g.

to: The 160-bit address of the message call's recipient or, for a contract creation transaction, $\varnothing$, used here to denote the only member of $\mathbb{B}_0$; formally T_t.

value: A scalar value equal to the number of Wei to be transferred to the message call's recipient or, in the case of contract creation, as an endowment to the newly created account; formally T_v.

v, r, s: Values corresponding to the signature of the transaction and used to determine the sender of the transaction; formally T_w, T_r and T_s. This is expanded in Appendix F.

Additionally, a contract creation transaction contains:

init: An unlimited size byte array specifying the TVM-code for the account initialisation procedure, formally $T_\mathbf{i}$.

init is an TVM-code fragment; it returns the **body**, a second fragment of code that executes each time the account receives a message call (either through a transaction or due to the internal execution of code). **init** is executed only once at account creation and gets discarded immediately thereafter.

In contrast, a message call transaction contains:

data: An unlimited size byte array specifying the input data of the message call, formally $T_\mathbf{d}$.

Appendix F specifies the function, S, which maps transactions to the sender, and happens through the ECDSA of the SECP-256k1 curve, using the hash of the transaction (excepting the latter three signature fields) as the datum to sign. For the present we simply assert that the sender of a given transaction T can be represented with $S(T)$.

$$(19) \quad L_T(T) \equiv \begin{cases} (T_n, T_p, T_g, T_t, T_v, T_\mathbf{i}, T_w, T_r, T_s) & \text{if } T_t = \varnothing \\ (T_n, T_p, T_g, T_t, T_v, T_\mathbf{d}, T_w, T_r, T_s) & \text{otherwise} \end{cases}$$

Here, we assume all components are interpreted by the RLP as integer values, with the exception of the arbitrary length byte arrays $T_\mathbf{i}$ and $T_\mathbf{d}$.

$$(20) \quad \begin{array}{lllll} T_n \in \mathbb{P}_{256} & \wedge & T_v \in \mathbb{P}_{256} & \wedge & T_p \in \mathbb{P}_{256} \quad \wedge \\ T_g \in \mathbb{P}_{256} & \wedge & T_w \in \mathbb{P}_5 & \wedge & T_r \in \mathbb{P}_{256} \quad \wedge \\ T_s \in \mathbb{P}_{256} & \wedge & T_\mathbf{d} \in \mathbb{B} & \wedge & T_\mathbf{i} \in \mathbb{B} \end{array}$$

where

$$(21) \quad \mathbb{P}_n = \{P : P \in \mathbb{P} \wedge P < 2^n\}$$

The address hash $T_\mathbf{t}$ is slightly different: it is either a 20-byte address hash or, in the case of being a contract-creation transaction (and thus formally equal to $\varnothing$), it is the RLP empty byte sequence and thus the member of $\mathbb{B}_0$:

$$(22) \quad T_\mathbf{t} \in \begin{cases} \mathbb{B}_{20} & \text{if} \quad T_t \neq \varnothing \\ \mathbb{B}_0 & \text{otherwise} \end{cases}$$

5.3. **The Block.** The block in Trustmachine is the collection of relevant pieces of information (known as the block *header*), H, together with information corresponding to the comprised transactions, $\mathbf{T}$, and a set of other block headers $\mathbf{U}$ that are known to have a parent equal to the present block's parent's parent (such blocks are known as *ommers*[2]). The block header contains several pieces of information:

parentHash: The Keccak 256-bit hash of the parent block's header, in its entirety; formally H_p.

ommersHash: The Keccak 256-bit hash of the ommers list portion of this block; formally H_o.

beneficiary: The 160-bit address to which all fees collected from the successful mining of this block be transferred; formally H_c.

stateRoot: The Keccak 256-bit hash of the root node of the state trie, after all transactions are executed and finalisations applied; formally H_r.

transactionsRoot: The Keccak 256-bit hash of the root node of the trie structure populated with each transaction in the transactions list portion of the block; formally H_t.

receiptsRoot: The Keccak 256-bit hash of the root node of the trie structure populated with the receipts of each transaction in the transactions list portion of the block; formally H_e.

logsBloom: The Bloom filter composed from indexable information (logger address and log topics) contained in each log entry from the receipt of each transaction in the transactions list; formally H_b.

difficulty: A scalar value corresponding to the difficulty level of this block. This can be calculated from the previous block's difficulty level and the timestamp; formally H_d.

number: A scalar value equal to the number of ancestor blocks. The genesis block has a number of zero; formally H_i.

manaLimit: A scalar value equal to the current limit of mana expenditure per block; formally H_l.

manaUsed: A scalar value equal to the total mana used in transactions in this block; formally H_g.

timestamp: A scalar value equal to the reasonable output of Unix's time() at this block's inception; formally H_s.

extraData: An arbitrary byte array containing data relevant to this block. This must be 32 bytes or fewer; formally H_x.

mixHash: A 256-bit hash which proves combined with the nonce that a sufficient amount of computation has been carried out on this block; formally H_m.

nonce: A 64-bit hash which proves combined with the mix-hash that a sufficient amount of computation has been carried out on this block; formally H_n.

The other two components in the block are simply a list of ommer block headers (of the same format as above) and a series of the transactions. Formally, we can refer to a block B:

$$(23) \quad B \equiv (B_H, B_\mathbf{T}, B_\mathbf{U})$$

5.3.1. *Transaction Receipt.* In order to encode information about a transaction concerning which it may be useful to form a zero-knowledge proof, or index and search, we encode a receipt of each transaction containing certain information from concerning its execution. Each receipt, denoted $B_{\mathbf{R}}[i]$ for the ith transaction, is placed in an index-keyed trie and the root recorded in the header as H_e.

The transaction receipt is a tuple of four items comprising the post-transaction state, $R_{\boldsymbol{\sigma}}$, the cumulative mana used in the block containing the transaction receipt as of immediately after the transaction has happened, R_u, the set of logs created through execution of the transaction, $R_{\mathbf{l}}$ and the Bloom filter composed from information in those logs, R_b:

(24) $$R \equiv (R_{\boldsymbol{\sigma}}, R_u, R_b, R_{\mathbf{l}})$$

The function L_R trivially prepares a transaction receipt for being transformed into an RLP-serialised byte array:

(25) $$L_R(R) \equiv (\texttt{TRIE}(L_S(R_{\boldsymbol{\sigma}})), R_u, R_b, R_{\mathbf{l}})$$

thus the post-transaction state, $R_{\boldsymbol{\sigma}}$ is encoded into a trie structure, the root of which forms the first item.

We assert R_u, the cumulative mana used is a positive integer and that the logs Bloom, R_b, is a hash of size 2048 bits (256 bytes):

(26) $$R_u \in \mathbb{P} \quad \wedge \quad R_b \in \mathbb{B}_{256}$$

The log entries, $R_{\mathbf{l}}$, is a series of log entries, termed, for example, $(O_0, O_1, ...)$. A log entry, O, is a tuple of a logger's address, O_a, a series of 32-bytes log topics, $O_{\mathbf{t}}$ and some number of bytes of data, $O_{\mathbf{d}}$:

(27) $$O \equiv (O_a, (O_{\mathbf{t}0}, O_{\mathbf{t}1}, ...), O_{\mathbf{d}})$$

(28) $$O_a \in \mathbb{B}_{20} \quad \wedge \quad \forall_{t \in O_{\mathbf{t}}} : t \in \mathbb{B}_{32} \quad \wedge \quad O_{\mathbf{d}} \in \mathbb{B}$$

We define the Bloom filter function, B_{FILTER}, to reduce a log entry into a single 256-byte hash:

(29) $$B_{\text{FILTER}}(O) \equiv \bigvee_{t \in \{O_a\} \cup O_{\mathbf{t}}} (B_{\text{FILTER 3:2048}}(t))$$

where $B_{\text{FILTER 3:2048}}$ is a specialised Bloom filter that sets three bits out of 2048, given an arbitrary byte sequence. It does this through taking the low-order 11 bits of each of the first three pairs of bytes in a Keccak-256 hash of the byte sequence. Formally:

$$\begin{aligned}
B_{\text{FILTER 3:2048}}(\mathbf{x} : \mathbf{x} \in \mathbb{B}) &\equiv \mathbf{y} : \mathbf{y} \in \mathbb{B}_{256} \quad \text{where:} \\
\mathbf{y} &= (0, 0, ..., 0) \quad \text{except:} \\
\forall_{i \in \{0,2,4\}} &: \mathcal{B}_{m(\mathbf{x},i)}(\mathbf{y}) = 1 \\
m(\mathbf{x}, i) &\equiv \texttt{KEC}(\mathbf{x})[i, i+1] \bmod 2048
\end{aligned}$$

where $\mathcal{B}$ is the bit reference function such that $\mathcal{B}_j(\mathbf{x})$ equals the bit of index j (indexed from 0) in the byte array $\mathbf{x}$.

5.3.2. *Holistic Validity.* We can assert a block's validity if and only if it satisfies several conditions: it must be internally consistent with the ommer and transaction block hashes and the given transactions $B_{\mathbf{T}}$ (as specified in sec 12), when executed in order on the base state $\boldsymbol{\sigma}$ (derived from the final state of the parent block), result in a new state of the identity H_r:

(34)
$$\begin{array}{lcll}
H_r & \equiv & \texttt{TRIE}(L_S(\Pi(\boldsymbol{\sigma}, B))) & \wedge \\
H_o & \equiv & \texttt{KEC}(\texttt{RLP}(L_H^*(B_{\mathbf{U}}))) & \wedge \\
H_t & \equiv & \texttt{TRIE}(\{\forall i < \|B_{\mathbf{T}}\|, i \in \mathbb{P} : p(i, L_T(B_{\mathbf{T}}[i]))\}) & \wedge \\
H_e & \equiv & \texttt{TRIE}(\{\forall i < \|B_{\mathbf{R}}\|, i \in \mathbb{P} : p(i, L_R(B_{\mathbf{R}}[i]))\}) & \wedge \\
H_b & \equiv & \bigvee_{\mathbf{r} \in B_{\mathbf{R}}} (\mathbf{r}_b) &
\end{array}$$

where $p(k, v)$ is simply the pairwise RLP transformation, in this case, the first being the index of the transaction in the block and the second being the transaction receipt:

(35) $$p(k, v) \equiv \big(\texttt{RLP}(k), \texttt{RLP}(v)\big)$$

Furthermore:

(36) $$\texttt{TRIE}(L_S(\boldsymbol{\sigma})) = P(B_H)_{H_r}$$

Thus $\texttt{TRIE}(L_S(\boldsymbol{\sigma}))$ is the root node hash of the Merkle Patricia tree structure containing the key-value pairs of the state $\boldsymbol{\sigma}$ with values encoded using RLP, and $P(B_H)$ is the parent block of B, defined directly.

The values stemming from the computation of transactions, specifically the transaction receipts, $B_{\mathbf{R}}$, and that defined through the transactions state-accumulation function, Π, are formalised later in section 12.4.

5.3.3. *Serialisation.* The function L_B and L_H are the preparation functions for a block and block header respectively. Much like the transaction receipt preparation function L_R, we assert the types and order of the structure for when the RLP transformation is required:

$$\begin{array}{lcl}
L_H(H) & \equiv & (\ H_p, H_o, H_c, H_r, H_t, H_e, H_b, H_d, \\
 & & \quad H_i, H_l, H_g, H_s, H_x, H_m, H_n\ ) \\
L_B(B) & \equiv & \big(L_H(B_H), L_T^*(B_{\mathbf{T}}), L_H^*(B_{\mathbf{U}})\big)
\end{array}$$

With L_T^* and L_H^* being element-wise sequence transformations, thus:

(39) $$f^*\big((x_0, x_1, ...)\big) \equiv \big(f(x_0), f(x_1), ...\big) \quad \text{for any function } f$$

The component types are defined thus:

(40)
$$\begin{array}{llllll}
H_p \in \mathbb{B}_{32} & \wedge & H_o \in \mathbb{B}_{32} & \wedge & H_c \in \mathbb{B}_{20} & \wedge \\
H_r \in \mathbb{B}_{32} & \wedge & H_t \in \mathbb{B}_{32} & \wedge & H_e \in \mathbb{B}_{32} & \wedge \\
H_b \in \mathbb{B}_{256} & \wedge & H_d \in \mathbb{P} & \wedge & H_i \in \mathbb{P} & \wedge \\
H_l \in \mathbb{P} & \wedge & H_g \in \mathbb{P} & \wedge & H_s \in \mathbb{P}_{256} & \wedge \\
H_x \in \mathbb{B} & \wedge & H_m \in \mathbb{B}_{32} & \wedge & H_n \in \mathbb{B}_8 &
\end{array}$$

where

(41) $$\mathbb{B}_n = \{B : B \in \mathbb{B} \wedge \|B\| = n\}$$

We now have a rigorous specification for the construction of a formal block structure. The RLP function $\texttt{RLP}$ (see Appendix B) provides the canonical method for transforming this structure into a sequence of bytes ready for transmission over the wire or storage locally.

5.3.4. *Block Header Validity.* We define $P(B_H)$ to be the parent block of B, formally:

(42) $$P(H) \equiv B' : \texttt{KEC}(\texttt{RLP}(B'_H)) = H_p$$

The block number is the parent's block number incremented by one:

(43) $$H_i \equiv P(H)_{H_i} + 1$$

The canonical difficulty of a block of header H is defined as $D(H)$:

(44) $$D(H) \equiv \begin{cases} D_0 & \text{if} \quad H_i = 0 \\ \max(D_0, P(H)_{H_d} + x \times \varsigma_2 + \epsilon) & \text{otherwise} \end{cases}$$

where:

(45) $$D_0 \equiv 131072$$

(46) $$x \equiv \left\lfloor \frac{P(H)_{H_d}}{2048} \right\rfloor$$

(47) $$\varsigma_2 \equiv \max\left(1 - \left\lfloor \frac{H_s - P(H)_{H_s}}{10} \right\rfloor, -99\right)$$

(48) $$\epsilon \equiv \left\lfloor 2^{\lfloor H_i \div 100000 \rfloor - 2} \right\rfloor$$

The canonical mana limit H_l of a block of header H must fulfil the relation:

$$\begin{aligned} H_l &< P(H)_{H_l} + \left\lfloor \frac{P(H)_{H_l}}{1024} \right\rfloor \quad \wedge \\ H_l &> P(H)_{H_l} - \left\lfloor \frac{P(H)_{H_l}}{1024} \right\rfloor \quad \wedge \\ H_l &\geqslant 125000 \end{aligned}$$

H_s is the timestamp of block H and must fulfil the relation:

(52) $$H_s > P(H)_{H_s}$$

This mechanism enforces a homeostasis in terms of the time between blocks; a smaller period between the last two blocks results in an increase in the difficulty level and thus additional computation required, lengthening the likely next period. Conversely, if the period is too large, the difficulty, and expected time to the next block, is reduced.

The nonce, H_n, must satisfy the relations:

(53) $$n \leqslant \frac{2^{256}}{H_d} \quad \wedge \quad m = H_m$$

with $(n, m) = \texttt{PoW}(H_{\cancel{n}}, H_n, \mathbf{d})$.

Where $H_{\cancel{n}}$ is the new block's header H, but *without* the nonce and mix-hash components, $\mathbf{d}$ being the current DAG, a large data set needed to compute the mix-hash, and PoW is the proof-of-work function (see section 17.1): this evaluates to an array with the first item being the mix-hash, to proof that a correct DAG has been used, and the second item being a pseudo-random number cryptographically dependent on H and $\mathbf{d}$. Given an approximately uniform distribution in the range $[0, 2^{64})$, the expected time to find a solution is proportional to the difficulty, H_d.

This is the foundation of the security of the blockchain and is the fundamental reason why a malicious node cannot propagate newly created blocks that would otherwise overwrite ("rewrite") history. Because the nonce must satisfy this requirement, and because its satisfaction depends on the contents of the block and in turn its composed transactions, creating new, valid, blocks is difficult and, over time, requires approximately the total compute power of the trustworthy portion of the mining peers.

Thus we are able to define the block header validity function $V(H)$:

$$\begin{aligned} V(H) \quad \equiv \quad & n \leqslant \frac{2^{256}}{H_d} \wedge m = H_m \quad \wedge \\ & H_d = D(H) \quad \wedge \\ & H_g \leq H_l \quad \wedge \\ & H_l < P(H)_{H_l} + \left\lfloor \frac{P(H)_{H_l}}{1024} \right\rfloor \quad \wedge \\ & H_l > P(H)_{H_l} - \left\lfloor \frac{P(H)_{H_l}}{1024} \right\rfloor \quad \wedge \\ & H_l \geqslant 125000 \quad \wedge \\ & H_s > P(H)_{H_s} \quad \wedge \\ & H_i = P(H)_{H_i} + 1 \quad \wedge \\ & \|H_x\| \leq 32 \end{aligned}$$

where $(n, m) = \texttt{PoW}(H_{\cancel{n}}, H_n, \mathbf{d})$

Noting additionally that **extraData** must be at most 32 bytes.

6. MANA AND PAYMENT

In order to avoid issues of network abuse and to sidestep the inevitable questions stemming from Turing completeness, all programmable computation in Trustmachine is subject to fees. The fee schedule is specified in units of *mana* (see Appendix G for the fees associated with various computation). Thus any given fragment of programmable computation (this includes creating contracts, making message calls, utilising and accessing account storage and executing operations on the virtual machine) has a universally agreed cost in terms of mana.

Every transaction has a specific amount of mana associated with it: **manaLimit**. This is the amount of mana which is implicitly purchased from the sender's account balance. The purchase happens at the according **manaPrice**, also specified in the transaction. The transaction is considered invalid if the account balance cannot support such a purchase. It is named **manaLimit** since any unused mana at the end of the transaction is refunded (at the same rate of purchase) to the sender's account. Mana does not exist outside of the execution of a transaction. Thus for accounts with trusted code associated, a relatively high mana limit may be set and left alone.

In general, Trust used to purchase mana that is not refunded is delivered to the *beneficiary* address, the address of an account typically under the control of the miner. Transactors are free to specify any **manaPrice** that they wish, however miners are free to ignore transactions as they choose. A higher mana price on a transaction will therefore cost the sender more in terms of Trust and deliver a greater value to the miner and thus will more likely be selected for inclusion by more miners. Miners, in general, will choose to advertise the minimum mana price for which they will execute transactions and transactors will be free to canvas these prices in determining what mana price to offer. Since there will be a (weighted) distribution of minimum acceptable mana prices, transactors will necessarily have a trade-off to make between lowering the mana price and maximising the chance that their transaction will be mined in a timely manner.

7. TRANSACTION EXECUTION

The execution of a transaction defines the state transition function Υ. It is assumed that any transactions executed first pass the initial tests of intrinsic validity. These include:

(1) The transaction is well-formed RLP, with no additional trailing bytes;
(2) the transaction signature is valid;
(3) the transaction nonce is valid (equivalent to the sender account's current nonce);
(4) the mana limit is no smaller than the intrinsic mana, g_0, used by the transaction;
(5) the sender account balance contains at least the cost, v_0, required in up-front payment.

Formally, we consider the function Υ, with T being a transaction and $\boldsymbol{\sigma}$ the state:

(63) $$\boldsymbol{\sigma}' = \Upsilon(\boldsymbol{\sigma}, T)$$

Thus $\boldsymbol{\sigma}'$ is the post-transactional state. We also define Υ^g to evaluate to the amount of mana used in the execution of a transaction and $\Upsilon^{\mathbf{l}}$ to evaluate to the transaction's accrued log items, both to be formally defined later.

7.1. Substate.

Throughout transaction execution, we accrue certain information that is acted upon immediately following the transaction. We call this *transaction substate*, and represent it as A, which is a tuple:

(64) $$A \equiv (A_{\mathbf{s}}, A_{\mathbf{l}}, A_r)$$

The tuple contents include $A_\mathbf{s}$, the self-destruct set: a set of accounts that will be discarded following the transaction's completion. $A_\mathbf{l}$ is the log series: this is a series of archived and indexable 'checkpoints' in VM code execution that allow for contract-calls to be easily tracked by onlookers external to the Trustmachine world (such as decentralised application front-ends). Finally there is A_r, the refund balance, increased through using the SSTORE instruction in order to reset contract storage to zero from some non-zero value. Though not immediately refunded, it is allowed to partially offset the total execution costs.

For brevity, we define the empty substate A^0 to have no self-destructs, no logs and a zero refund balance:

(65) $$A^0 \equiv (\varnothing, (), 0)$$

7.2. **Execution.** We define intrinsic mana g_0, the amount of mana this transaction requires to be paid prior to execution, as follows:

(66) $$g_0 \equiv \sum_{i \in T_\mathbf{i}, T_\mathbf{d}} \begin{cases} M_{txdatazero} & \text{if} \quad i = 0 \\ M_{txdatanonzero} & \text{otherwise} \end{cases}$$

(67) $$+ \begin{cases} M_{\text{txcreate}} & \text{if} \quad T_t = \varnothing \\ 0 & \text{otherwise} \end{cases}$$

(68) $$+ M_{transaction}$$

where $T_\mathbf{i}, T_\mathbf{d}$ means the series of bytes of the transaction's associated data and initialisation TVM-code, depending on whether the transaction is for contract-creation or message-call. M_{txcreate} is added if the transaction is contract-creating, but not if a result of TVM-code. M is fully defined in Appendix G.

The up-front cost v_0 is calculated as:

(69) $$v_0 \equiv T_g T_p + T_v$$

The validity is determined as:

(70) $$\begin{array}{rcl} S(T) & \neq & \varnothing \quad \wedge \\ \boldsymbol{\sigma}[S(T)] & \neq & \varnothing \quad \wedge \\ T_n & = & \boldsymbol{\sigma}[S(T)]_n \quad \wedge \\ g_0 & \leqslant & T_g \quad \wedge \\ v_0 & \leqslant & \boldsymbol{\sigma}[S(T)]_b \quad \wedge \\ T_g & \leqslant & B_{Hl} - \ell(B_\mathbf{R})_u \end{array}$$

Note the final condition; the sum of the transaction's mana limit, T_g, and the mana utilised in this block prior, given by $\ell(B_\mathbf{R})_u$, must be no greater than the block's **manaLimit**, B_{Hl}.

The execution of a valid transaction begins with an irrevocable change made to the state: the nonce of the account of the sender, $S(T)$, is incremented by one and the balance is reduced by part of the up-front cost, T_gT_p. The mana available for the proceeding computation, g, is defined as $T_g - g_0$. The computation, whether contract creation or a message call, results in an eventual state (which may legally be equivalent to the current state), the change to which is deterministic and never invalid: there can be no invalid transactions from this point.

We define the checkpoint state $\boldsymbol{\sigma}_0$:

$$\begin{array}{rcl} \boldsymbol{\sigma}_0 & \equiv & \boldsymbol{\sigma} \quad \text{except:} \\ \boldsymbol{\sigma}_0[S(T)]_b & \equiv & \boldsymbol{\sigma}[S(T)]_b - T_g T_p \\ \boldsymbol{\sigma}_0[S(T)]_n & \equiv & \boldsymbol{\sigma}[S(T)]_n + 1 \end{array}$$

Evaluating $\boldsymbol{\sigma}_P$ from $\boldsymbol{\sigma}_0$ depends on the transaction type; either contract creation or message call; we define the tuple of post-execution provisional state $\boldsymbol{\sigma}_P$, remaining mana g' and substate A:

(74)
$$(\boldsymbol{\sigma}_P, g', A) \equiv \begin{cases} \Lambda(\boldsymbol{\sigma}_0, S(T), T_o, & \\ \quad g, T_p, T_v, T_\mathbf{i}, 0) & \text{if} \quad T_t = \varnothing \\ \Theta_3(\boldsymbol{\sigma}_0, S(T), T_o, & \\ \quad T_t, T_t, g, T_p, T_v, T_v, T_\mathbf{d}, 0) & \text{otherwise} \end{cases}$$

where g is the amount of mana remaining after deducting the basic amount required to pay for the existence of the transaction:

(75) $$g \equiv T_g - g_0$$

and T_o is the original transactor, which can differ from the sender in the case of a message call or contract creation not directly triggered by a transaction but coming from the execution of TVM-code.

Note we use Θ_3 to denote the fact that only the first three components of the function's value are taken; the final represents the message-call's output value (a byte array) and is unused in the context of transaction evaluation.

After the message call or contract creation is processed, the state is finalised by determining the amount to be refunded, g^* from the remaining mana, g', plus some allowance from the refund counter, to the sender at the original rate.

(76) $$g^* \equiv g' + \min\{\Big\lfloor \frac{T_g - g'}{2} \Big\rfloor, A_r\}$$

The total refundable amount is the legitimately remaining mana g', added to A_r, with the latter component being capped up to a maximum of half (rounded down) of the total amount used $T_g - g'$.

The Trust for the mana is given to the miner, whose address is specified as the beneficiary of the present block B. So we define the pre-final state $\boldsymbol{\sigma}^*$ in terms of the provisional state $\boldsymbol{\sigma}_P$:

$$\begin{array}{rcl} \boldsymbol{\sigma}^* & \equiv & \boldsymbol{\sigma}_P \quad \text{except} \\ \boldsymbol{\sigma}^*[S(T)]_b & \equiv & \boldsymbol{\sigma}_P[S(T)]_b + g^* T_p \\ \boldsymbol{\sigma}^*[m]_b & \equiv & \boldsymbol{\sigma}_P[m]_b + (T_g - g^*) T_p \\ m & \equiv & B_{Hc} \end{array}$$

The final state, $\boldsymbol{\sigma}'$, is reached after deleting all accounts that appear in the self-destruct set:

$$\begin{array}{rcl} \boldsymbol{\sigma}' & \equiv & \boldsymbol{\sigma}^* \quad \text{except} \\ \forall i \in A_\mathbf{s} : \boldsymbol{\sigma}'[i] & \equiv & \varnothing \end{array}$$

And finally, we specify Υ^g, the total mana used in this transaction and $\Upsilon^\mathbf{l}$, the logs created by this transaction:

$$\begin{array}{rcl} \Upsilon^g(\boldsymbol{\sigma}, T) & \equiv & T_g - g' \\ \Upsilon^\mathbf{l}(\boldsymbol{\sigma}, T) & \equiv & A_\mathbf{l} \end{array}$$

These are used to help define the transaction receipt, discussed later.

8. CONTRACT CREATION

There are a number of intrinsic parameters used when creating an account: sender (s), original transactor (o), available mana (g), mana price (p), endowment (v) together with an arbitrary length byte array, $\mathbf{i}$, the initialisation TVM code and finally the present depth of the message-call/contract-creation stack (e).

We define the creation function formally as the function Λ, which evaluates from these values, together with the state $\boldsymbol{\sigma}$ to the tuple containing the new state, remaining mana and accrued transaction substate $(\boldsymbol{\sigma}', g', A)$, as in section 7:

$$(85)\qquad (\boldsymbol{\sigma}', g', A) \equiv \Lambda(\boldsymbol{\sigma}, s, o, g, p, v, \mathbf{i}, e)$$

The address of the new account is defined as being the rightmost 160 bits of the Keccak hash of the RLP encoding of the structure containing only the sender and the nonce. Thus we define the resultant address for the new account a:

$$(86)\qquad a \equiv \mathcal{B}_{96..255}\Big(\mathtt{KEC}\Big(\mathtt{RLP}\big(\;(s, \boldsymbol{\sigma}[s]_n - 1)\;\big)\Big)\Big)$$

where $\mathtt{KEC}$ is the Keccak 256-bit hash function, $\mathtt{RLP}$ is the RLP encoding function, $\mathcal{B}_{a..b}(X)$ evaluates to binary value containing the bits of indices in the range $[a, b]$ of the binary data X and $\boldsymbol{\sigma}[x]$ is the address state of x or $\varnothing$ if none exists. Note we use one fewer than the sender's nonce value; we assert that we have incremented the sender account's nonce prior to this call, and so the value used is the sender's nonce at the beginning of the responsible transaction or VM operation.

The account's nonce is initially defined as zero, the balance as the value passed, the storage as empty and the code hash as the Keccak 256-bit hash of the empty string; the sender's balance is also reduced by the value passed. Thus the mutated state becomes $\boldsymbol{\sigma}^*$:

$$(87)\qquad \boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma} \quad \text{except:}$$

$$\begin{aligned} \boldsymbol{\sigma}^*[a] &\equiv \big(0, v + v', \mathtt{TRIE}(\varnothing), \mathtt{KEC}\big(()\big)\big) \\ \boldsymbol{\sigma}^*[s]_b &\equiv \boldsymbol{\sigma}[s]_b - v \end{aligned}$$

where v' is the account's pre-existing value, in the event it was previously in existence:

$$(90)\qquad v' \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\sigma}[a] = \varnothing \\ \boldsymbol{\sigma}[a]_b & \text{otherwise} \end{cases}$$

Finally, the account is initialised through the execution of the initialising TVM code $\mathbf{i}$ according to the execution model (see section 10). Code execution can effect several events that are not internal to the execution state: the account's storage can be altered, further accounts can be created and further message calls can be made. As such, the code execution function Ξ evaluates to a tuple of the resultant state $\boldsymbol{\sigma}^{**}$, available mana remaining g^{**}, the accrued substate A and the body code of the account $\mathbf{o}$.

$$(91)\qquad (\boldsymbol{\sigma}^{**}, g^{**}, A, \mathbf{o}) \equiv \Xi(\boldsymbol{\sigma}^*, g, I)$$

where I contains the parameters of the execution environment as defined in section 10, that is:

$$\begin{aligned} I_a &\equiv a \\ I_o &\equiv o \\ I_p &\equiv p \\ I_{\mathbf{d}} &\equiv () \\ I_s &\equiv s \\ I_v &\equiv v \\ I_{\mathbf{b}} &\equiv \mathbf{i} \\ I_e &\equiv e \end{aligned}$$

$I_{\mathbf{d}}$ evaluates to the empty tuple as there is no input data to this call. I_H has no special treatment and is determined from the blockchain.

Code execution depletes mana, and mana may not go below zero, thus execution may exit before the code has come to a natural halting state. In this (and several other) exceptional cases we say an out-of-mana (OOG) exception has occurred: The evaluated state is defined as being the empty set, $\varnothing$, and the entire create operation should have no effect on the state, effectively leaving it as it was immediately prior to attempting the creation.

If the initialization code completes successfully, a final contract-creation cost is paid, the code-deposit cost, c, proportional to the size of the created contract's code:

$$(100)\qquad c \equiv M_{codedeposit} \times |\mathbf{o}|$$

If there is not enough mana remaining to pay this, i.e. $g^{**} < c$, then we also declare an out-of-mana exception.

The mana remaining will be zero in any such exceptional condition, i.e. if the creation was conducted as the reception of a transaction, then this doesn't affect payment of the intrinsic cost of contract creation; it is paid regardless. However, the value of the transaction is not transferred to the aborted contract's address when we are out-of-mana.

If such an exception does not occur, then the remaining mana is refunded to the originator and the now-altered state is allowed to persist. Thus formally, we may specify the resultant state, mana and substate as $(\boldsymbol{\sigma}', g', A)$ where:

$$(101)\qquad g' \equiv \begin{cases} 0 & \text{if} \quad F \\ g^{**} - c & \text{otherwise} \end{cases}$$

$$(102)\qquad \boldsymbol{\sigma}' \equiv \begin{cases} \boldsymbol{\sigma} & \text{if} \quad F \\ \boldsymbol{\sigma}^{**} \quad \text{except:} & \\ \quad \boldsymbol{\sigma}'[a]_c = \mathtt{KEC}(\mathbf{o}) & \text{otherwise} \end{cases}$$

where

$$(103)\qquad F \equiv \big(\boldsymbol{\sigma}^{**} = \varnothing \ \vee\ g^{**} < c \ \vee\ |\mathbf{o}| > 24576\big)$$

The exception in the determination of $\boldsymbol{\sigma}'$ dictates that $\mathbf{o}$, the resultant byte sequence from the execution of the initialisation code, specifies the final body code for the newly-created account.

Note that intention is that the result is either a successfully created new contract with its endowment, or no new contract with no transfer of value.

8.1. **Subtleties.** Note that while the initialisation code is executing, the newly created address exists but with no intrinsic body code. Thus any message call received by it during this time causes no code to be executed. If the initialisation execution ends with a SELFDESTRUCT instruction, the matter is moot since the account will be deleted before the transaction is completed. For a normal STOP code, or if the code returned is otherwise empty, then the state is left with a zombie account, and any remaining balance will be locked into the account forever.

9. MESSAGE CALL

In the case of executing a message call, several parameters are required: sender (s), transaction originator (o), recipient (r), the account whose code is to be executed (c, usually the same as recipient), available mana (g), value (v) and mana price (p) together with an arbitrary length byte array, $\mathbf{d}$, the input data of the call and finally the present depth of the message-call/contract-creation stack (e).

Aside from evaluating to a new state and transaction substate, message calls also have an extra component—the output data denoted by the byte array $\mathbf{o}$. This is ignored when executing transactions, however message calls can be initiated due to VM-code execution and in this case this information is used.

$$(104)\qquad (\boldsymbol{\sigma}', g', A, \mathbf{o}) \equiv \Theta(\boldsymbol{\sigma}, s, o, r, c, g, p, v, \tilde{v}, \mathbf{d}, e)$$

Note that we need to differentiate between the value that is to be transferred, v, from the value apparent in the execution context, $\tilde{v}$, for the DELEGATECALL instruction.

We define $\boldsymbol{\sigma}_1$, the first transitional state as the original state but with the value transferred from sender to recipient:

$$\boldsymbol{\sigma}_1[r]_b \equiv \boldsymbol{\sigma}[r]_b + v \quad \wedge \quad \boldsymbol{\sigma}_1[s]_b \equiv \boldsymbol{\sigma}[s]_b - v \tag{105}$$

unless $s = r$.

Throughout the present work, it is assumed that if $\boldsymbol{\sigma}_1[r]$ was originally undefined, it will be created as an account with no code or state and zero balance and nonce. Thus the previous equation should be taken to mean:

$$\boldsymbol{\sigma}_1 \equiv \boldsymbol{\sigma}'_1 \quad \text{except:} \tag{106}$$

$$\boldsymbol{\sigma}_1[s]_b \equiv \boldsymbol{\sigma}'_1[s]_b - v \tag{107}$$

$$\text{and} \quad \boldsymbol{\sigma}'_1 \equiv \boldsymbol{\sigma} \quad \text{except:} \tag{108}$$

$$\begin{cases} \boldsymbol{\sigma}'_1[r] \equiv (v, 0, \mathtt{KEC}(()), \mathtt{TRIE}(\varnothing)) & \text{if} \quad \boldsymbol{\sigma}[r] = \varnothing \\ \boldsymbol{\sigma}'_1[r]_b \equiv \boldsymbol{\sigma}[r]_b + v & \text{otherwise} \end{cases} \tag{109}$$

The account's associated code (identified as the fragment whose Keccak hash is $\boldsymbol{\sigma}[c]_c$) is executed according to the execution model (see section 10). Just as with contract creation, if the execution halts in an exceptional fashion (i.e. due to an exhausted mana supply, stack underflow, invalid jump destination or invalid instruction), then no mana is refunded to the caller and the state is reverted to the point immediately prior to balance transfer (i.e. $\boldsymbol{\sigma}$).

$$\begin{aligned}
\boldsymbol{\sigma}' &\equiv \begin{cases} \boldsymbol{\sigma} & \text{if} \quad \boldsymbol{\sigma}^{**} = \varnothing \\ \boldsymbol{\sigma}^{**} & \text{otherwise} \end{cases} \\
g' &\equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\sigma}^{**} = \varnothing \\ g^{**} & \text{otherwise} \end{cases} \\
(\boldsymbol{\sigma}^{**}, g^{**}, A, \mathbf{o}) &\equiv \begin{cases} \Xi_{\mathtt{ECREC}}(\boldsymbol{\sigma}_1, g, I) & \text{if} \quad r = 1 \\ \Xi_{\mathtt{SHA256}}(\boldsymbol{\sigma}_1, g, I) & \text{if} \quad r = 2 \\ \Xi_{\mathtt{RIP160}}(\boldsymbol{\sigma}_1, g, I) & \text{if} \quad r = 3 \\ \Xi_{\mathtt{ID}}(\boldsymbol{\sigma}_1, g, I) & \text{if} \quad r = 4 \\ \Xi(\boldsymbol{\sigma}_1, g, I) & \text{otherwise} \end{cases} \\
I_a &\equiv r \\
I_o &\equiv o \\
I_p &\equiv p \\
I_{\mathbf{d}} &\equiv \mathbf{d} \\
I_s &\equiv s \\
I_v &\equiv \tilde{v} \\
I_e &\equiv e \\
\text{Let } \mathtt{KEC}(I_{\mathbf{b}}) &= \boldsymbol{\sigma}[c]_c
\end{aligned}$$

It is assumed that the client will have stored the pair $(\mathtt{KEC}(I_{\mathbf{b}}), I_{\mathbf{b}})$ at some point prior in order to make the determination of $I_{\mathbf{b}}$ feasible.

As can be seen, there are four exceptions to the usage of the general execution framework Ξ for evaluation of the message call: these are four so-called 'precompiled' contracts, meant as a preliminary piece of architecture that may later become *native extensions*. The four contracts in addresses 1, 2, 3 and 4 execute the elliptic curve public key recovery function, the SHA2 256-bit hash scheme, the RIPEMD 160-bit hash scheme and the identity function respectively.

Their full formal definition is in Appendix E.

10. Execution Model

The execution model specifies how the system state is altered given a series of bytecode instructions and a small tuple of environmental data. This is specified through a formal model of a virtual state machine, known as the Trustmachine Virtual Machine (TVM). It is a *quasi*-Turing-complete machine; the *quasi* qualification comes from the fact that the computation is intrinsically bounded through a parameter, *mana*, which limits the total amount of computation done.

10.1. Basics.

The TVM is a simple stack-based architecture. The word size of the machine (and thus size of stack item) is 256-bit. This was chosen to facilitate the Keccak-256 hash scheme and elliptic-curve computations. The memory model is a simple word-addressed byte array. The stack has a maximum size of 1024. The machine also has an independent storage model; this is similar in concept to the memory but rather than a byte array, it is a word-addressable word array. Unlike memory, which is volatile, storage is non volatile and is maintained as part of the system state. All locations in both storage and memory are well-defined initially as zero.

The machine does not follow the standard von Neumann architecture. Rather than storing program code in generally-accessible memory or storage, it is stored separately in a virtual ROM interactable only through a specialised instruction.

The machine can have exceptional execution for several reasons, including stack underflows and invalid instructions. Like the out-of-mana exception, they do not leave state changes intact. Rather, the machine halts immediately and reports the issue to the execution agent (either the transaction processor or, recursively, the spawning execution environment) which will deal with it separately.

10.2. Fees Overview.

Fees (denominated in mana) are charged under three distinct circumstances, all three as prerequisite to the execution of an operation. The first and most common is the fee intrinsic to the computation of the operation (see Appendix G). Secondly, mana may be deducted in order to form the payment for a subordinate message call or contract creation; this forms part of the payment for CREATE, CALL and CALLCODE. Finally, mana may be paid due to an increase in the usage of the memory.

Over an account's execution, the total fee for memory-usage payable is proportional to smallest multiple of 32 bytes that are required such that all memory indices (whether for read or write) are included in the range. This is paid for on a just-in-time basis; as such, referencing an area of memory at least 32 bytes greater than any previously indexed memory will certainly result in an additional memory usage fee. Due to this fee it is highly unlikely addresses will ever go above 32-bit bounds. That said, implementations must be able to manage this eventuality.

Storage fees have a slightly nuanced behaviour—to incentivise minimisation of the use of storage (which corresponds directly to a larger state database on all nodes), the execution fee for an operation that clears an entry in the storage is not only waived, a qualified refund is given; in fact, this refund is effectively paid up-front since the initial usage of a storage location costs substantially more than normal usage.

See Appendix H for a rigorous definition of the TVM mana cost.

10.3. **Execution Environment.** In addition to the system state $\boldsymbol{\sigma}$, and the remaining mana for computation g, there are several pieces of important information used in the execution environment that the execution agent must provide; these are contained in the tuple I:

- I_a, the address of the account which owns the code that is executing.
- I_o, the sender address of the transaction that originated this execution.
- I_p, the price of mana in the transaction that originated this execution.
- $I_{\mathbf{d}}$, the byte array that is the input data to this execution; if the execution agent is a transaction, this would be the transaction data.
- I_s, the address of the account which caused the code to be executing; if the execution agent is a transaction, this would be the transaction sender.
- I_v, the value, in Wei, passed to this account as part of the same procedure as execution; if the execution agent is a transaction, this would be the transaction value.
- $I_{\mathbf{b}}$, the byte array that is the machine code to be executed.
- I_H, the block header of the present block.
- I_e, the depth of the present message-call or contract-creation (i.e. the number of CALLs or CREATEs being executed at present).

The execution model defines the function Ξ, which can compute the resultant state $\boldsymbol{\sigma}'$, the remaining mana g', the accrued substate A and the resultant output, $\mathbf{o}$, given these definitions. For the present context, we will defined it as:

$$(\boldsymbol{\sigma}', g', A, \mathbf{o}) \equiv \Xi(\boldsymbol{\sigma}, g, I) \tag{121}$$

where we will remember that A, the accrued substate is defined as the tuple of the suicides set $\mathbf{s}$, the log series $\mathbf{l}$ and the refunds r:

$$A \equiv (\mathbf{s}, \mathbf{l}, r) \tag{122}$$

10.4. **Execution Overview.** We must now define the Ξ function. In most practical implementations this will be modelled as an iterative progression of the pair comprising the full system state, $\boldsymbol{\sigma}$ and the machine state, $\boldsymbol{\mu}$. Formally, we define it recursively with a function X. This uses an iterator function O (which defines the result of a single cycle of the state machine) together with functions Z which determines if the present state is an exceptional halting state of the machine and H, specifying the output data of the instruction if and only if the present state is a normal halting state of the machine.

The empty sequence, denoted $()$, is not equal to the empty set, denoted $\varnothing$; this is important when interpreting the output of H, which evaluates to $\varnothing$ when execution is to continue but a series (potentially empty) when execution should halt.

$$\begin{aligned}
\Xi(\boldsymbol{\sigma}, g, I) &\equiv (\boldsymbol{\sigma}', \boldsymbol{\mu}'_g, A, \mathbf{o}) \\
(\boldsymbol{\sigma}, \boldsymbol{\mu}', A, ..., \mathbf{o}) &\equiv X\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A^0, I)\big) \\
\boldsymbol{\mu}_g &\equiv g \\
\boldsymbol{\mu}_{pc} &\equiv 0 \\
\boldsymbol{\mu}_{\mathbf{m}} &\equiv (0, 0, ...) \\
\boldsymbol{\mu}_i &\equiv 0 \\
\boldsymbol{\mu}_{\mathbf{s}} &\equiv ()
\end{aligned}$$

$$X\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\big) \equiv \begin{cases} (\varnothing, \boldsymbol{\mu}, A^0, I, ()) & \text{if} \quad Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \\ O(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \cdot \mathbf{o} & \text{if} \quad \mathbf{o} \neq \varnothing \\ X\big(O(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\big) & \text{otherwise} \end{cases} \tag{130}$$

where

$$\begin{aligned}
\mathbf{o} &\equiv H(\boldsymbol{\mu}, I) \\
(a, b, c, d) \cdot e &\equiv (a, b, c, d, e)
\end{aligned}$$

Note that, when we evaluate Ξ, we drop the fourth element I' and extract the remaining mana $\boldsymbol{\mu}'_g$ from the resultant machine state $\boldsymbol{\mu}'$.

X is thus cycled (recursively here, but implementations are generally expected to use a simple iterative loop) until either Z becomes true indicating that the present state is exceptional and that the machine must be halted and any changes discarded or until H becomes a series (rather than the empty set) indicating that the machine has reached a controlled halt.

10.4.1. *Machine State.* The machine state $\boldsymbol{\mu}$ is defined as the tuple $(g, pc, \mathbf{m}, i, \mathbf{s})$ which are the mana available, the program counter $pc \in \mathbb{P}_{256}$, the memory contents, the active number of words in memory (counting continuously from position 0), and the stack contents. The memory contents $\boldsymbol{\mu}_{\mathbf{m}}$ are a series of zeroes of size 2^{256}.

For the ease of reading, the instruction mnemonics, written in small-caps (e.g. ADD), should be interpreted as their numeric equivalents; the full table of instructions and their specifics is given in Appendix H.

For the purposes of defining Z, H and O, we define w as the current operation to be executed:

$$w \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{pc}] & \text{if} \quad \boldsymbol{\mu}_{pc} < \|I_{\mathbf{b}}\| \\ \text{STOP} & \text{otherwise} \end{cases} \tag{133}$$

We also assume the fixed amounts of δ and α, specifying the stack items removed and added, both subscriptable on the instruction and an instruction cost function C evaluating to the full cost, in mana, of executing the given instruction.

10.4.2. *Exceptional Halting.* The exceptional halting function Z is defined as:

$$\begin{aligned}
Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \equiv \quad & \boldsymbol{\mu}_g < C(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \quad \vee \\
& \delta_w = \varnothing \quad \vee \\
& \|\boldsymbol{\mu}_{\mathbf{s}}\| < \delta_w \quad \vee \\
& (w \in \{\text{JUMP}, \text{JUMPI}\} \quad \wedge \\
& \quad \boldsymbol{\mu}_{\mathbf{s}}[0] \notin D(I_{\mathbf{b}})) \quad \vee \\
& \|\boldsymbol{\mu}_{\mathbf{s}}\| - \delta_w + \alpha_w > 1024
\end{aligned} \tag{134}$$

This states that the execution is in an exceptional halting state if there is insufficient mana, if the instruction is invalid (and therefore its δ subscript is undefined), if there are insufficient stack items, if a JUMP/JUMPI destination is invalid or the new stack size would be larger then 1024. The astute reader will realise that this implies that no instruction can, through its execution, cause an exceptional halt.

10.4.3. *Jump Destination Validity.* We previously used D as the function to determine the set of valid jump destinations given the code that is being run. We define this as any position in the code occupied by a JUMPDEST instruction.

All such positions must be on valid instruction boundaries, rather than sitting in the data portion of PUSH operations and must appear within the explicitly defined portion of the code (rather than in the implicitly defined STOP operations that trail it).

Formally:

(135) $$D(\mathbf{c}) \equiv D_J(\mathbf{c}, 0)$$

where:

(136)
$$D_J(\mathbf{c}, i) \equiv \begin{cases} \{\} & \text{if} \quad i \geqslant |\mathbf{c}| \\ \{i\} \cup D_J(\mathbf{c}, N(i, \mathbf{c}[i])) & \text{if} \quad \mathbf{c}[i] = \text{JUMPDEST} \\ D_J(\mathbf{c}, N(i, \mathbf{c}[i])) & \text{otherwise} \end{cases}$$

where N is the next valid instruction position in the code, skipping the data of a PUSH instruction, if any:

(137)
$$N(i, w) \equiv \begin{cases} i + w - \text{PUSH1} + 2 & \text{if} \quad w \in [\text{PUSH1}, \text{PUSH32}] \\ i + 1 & \text{otherwise} \end{cases}$$

10.4.4. *Normal Halting.* The normal halting function H is defined:

(138)
$$H(\boldsymbol{\mu}, I) \equiv \begin{cases} H_{\text{RETURN}}(\boldsymbol{\mu}) & \text{if} \quad w = \text{RETURN} \\ () & \text{if} \quad w \in \{\text{STOP}, \text{SELFDESTRUCT}\} \\ \varnothing & \text{otherwise} \end{cases}$$

The data-returning halt operation, RETURN, has a special function H_{RETURN}, defined in Appendix H.

10.5. **The Execution Cycle.** Stack items are added or removed from the left-most, lower-indexed portion of the series; all other items remain unchanged:

$$\begin{aligned} O\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\big) &\equiv (\boldsymbol{\sigma}', \boldsymbol{\mu}', A', I) \\ \Delta &\equiv \alpha_w - \delta_w \\ \|\boldsymbol{\mu}'_{\mathbf{s}}\| &\equiv \|\boldsymbol{\mu}_{\mathbf{s}}\| + \Delta \\ \forall x \in [\alpha_w, \|\boldsymbol{\mu}'_{\mathbf{s}}\|) : \boldsymbol{\mu}'_{\mathbf{s}}[x] &\equiv \boldsymbol{\mu}_{\mathbf{s}}[x + \Delta] \end{aligned}$$

The mana is reduced by the instruction's mana cost and for most instructions, the program counter increments on each cycle, for the three exceptions, we assume a function J, subscripted by one of two instructions, which evaluates to the according value:

$$\begin{aligned} \boldsymbol{\mu}'_g &\equiv \boldsymbol{\mu}_g - C(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \\ \boldsymbol{\mu}'_{pc} &\equiv \begin{cases} J_{\text{JUMP}}(\boldsymbol{\mu}) & \text{if} \quad w = \text{JUMP} \\ J_{\text{JUMPI}}(\boldsymbol{\mu}) & \text{if} \quad w = \text{JUMPI} \\ N(\boldsymbol{\mu}_{pc}, w) & \text{otherwise} \end{cases} \end{aligned}$$

In general, we assume the memory, self-destruct set and system state don't change:

$$\begin{aligned} \boldsymbol{\mu}'_{\mathbf{m}} &\equiv \boldsymbol{\mu}_{\mathbf{m}} \\ \boldsymbol{\mu}'_i &\equiv \boldsymbol{\mu}_i \\ A' &\equiv A \\ \boldsymbol{\sigma}' &\equiv \boldsymbol{\sigma} \end{aligned}$$

However, instructions do typically alter one or several components of these values. Altered components listed by instruction are noted in Appendix H, alongside values for α and δ and a formal description of the mana requirements.

11. BLOCKTREE TO BLOCKCHAIN

The canonical blockchain is a path from root to leaf through the entire block tree. In order to have consensus over which path it is, conceptually we identify the path that has had the most computation done upon it, or, the *heaviest* path. Clearly one factor that helps determine the heaviest path is the block number of the leaf, equivalent to the number of blocks, not counting the unmined genesis block, in the path. The longer the path, the greater the total mining effort that must have been done in order to arrive at the leaf. This is akin to existing schemes, such as that employed in Bitcoin-derived protocols.

Since a block header includes the difficulty, the header alone is enough to validate the computation done. Any block contributes toward the total computation or *total difficulty* of a chain.

Thus we define the total difficulty of block B recursively as:

$$\begin{aligned} B_t &\equiv B'_t + B_d \\ B' &\equiv P(B_H) \end{aligned}$$

As such given a block B, B_t is its total difficulty, B' is its parent block and B_d is its difficulty.

12. BLOCK FINALISATION

The process of finalising a block involves four stages:

(1) Validate (or, if mining, determine) ommers;
(2) validate (or, if mining, determine) transactions;
(3) apply rewards;
(4) verify (or, if mining, compute a valid) state and nonce.

12.1. **Ommer Validation.** The validation of ommer headers means nothing more than verifying that each ommer header is both a valid header and satisfies the relation of Nth-generation ommer to the present block where $N \leq 6$. The maximum of ommer headers is two. Formally:

(151) $$\|B_{\mathbf{U}}\| \leqslant 2 \bigwedge_{U \in B_{\mathbf{U}}} V(U) \ \wedge \ k(U, P(B_H)_H, 6)$$

where k denotes the "is-kin" property:

(152) $$k(U, H, n) \equiv \begin{cases} false & \text{if} \quad n = 0 \\ s(U, H) & \\ \quad \vee \, k(U, P(H)_H, n - 1) & \text{otherwise} \end{cases}$$

and s denotes the "is-sibling" property:

(153) $$s(U, H) \equiv (P(H) = P(U) \ \wedge \ H \neq U \ \wedge \ U \notin B(H)_{\mathbf{U}})$$

where $B(H)$ is the block of the corresponding header H.

12.2. **Transaction Validation.** The given **manaUsed** must correspond faithfully to the transactions listed: B_{H_g}, the total mana used in the block, must be equal to the accumulated mana used according to the final transaction:

(154) $$B_{H_g} = \ell(\mathbf{R})_u$$

12.3. **Reward Application.** The application of rewards to a block involves raising the balance of the accounts of the beneficiary address of the block and each ommer by a certain amount. We raise the block's beneficiary account by R_b; for each ommer, we raise the block's beneficiary by an additional $\frac{1}{32}$ of the block reward and the beneficiary of the ommer gets rewarded depending on the block number. Formally we define the function Ω:

$$\begin{aligned} \Omega(B, \boldsymbol{\sigma}) &\equiv \boldsymbol{\sigma}' : \boldsymbol{\sigma}' = \boldsymbol{\sigma} \quad \text{except:} \\ \boldsymbol{\sigma}'[B_{H_c}]_b &= \boldsymbol{\sigma}[B_{H_c}]_b + (1 + \frac{\|B_{\mathbf{U}}\|}{32}) R_b \\ \forall_{U \in B_{\mathbf{U}}} &: \\ \boldsymbol{\sigma}'[U_c]_b &= \boldsymbol{\sigma}[U_c]_b + (1 + \frac{1}{8}(U_i - B_{H_i})) R_b \end{aligned}$$

If there are collisions of the beneficiary addresses between ommers and the block (i.e. two ommers with the same beneficiary address or an ommer with the same beneficiary address as the present block), additions are applied cumulatively.

We define the block reward as 5 Trust:

(158) $$\text{Let} \quad R_b = 5 \times 10^{18}$$

12.4. **State & Nonce Validation.** We may now define the function, Γ, that maps a block B to its initiation state:

(159)
$$\Gamma(B) \equiv \begin{cases} \boldsymbol{\sigma}_0 & \text{if} \quad P(B_H) = \varnothing \\ \boldsymbol{\sigma}_i : \texttt{TRIE}(L_S(\boldsymbol{\sigma}_i)) = P(B_H)_{H_r} & \text{otherwise} \end{cases}$$

Here, $\texttt{TRIE}(L_S(\boldsymbol{\sigma}_i))$ means the hash of the root node of a trie of state $\boldsymbol{\sigma}_i$; it is assumed that implementations will store this in the state database, trivial and efficient since the trie is by nature an immutable data structure.

And finally define Φ, the block transition function, which maps an incomplete block B to a complete block B':

$$\begin{aligned} \Phi(B) &\equiv B' : \quad B' = B^* \quad \text{except:} \\ B'_n &= n : \quad x \leqslant \frac{2^{256}}{H_d} \\ B'_m &= m \quad \text{with } (x, m) = \texttt{PoW}(B^*_{\not{n}}, n, \mathbf{d}) \\ B^* &\equiv B \quad \text{except:} \quad B^*_r = r(\Pi(\Gamma(B), B)) \end{aligned}$$

With $\mathbf{d}$ being a dataset as specified in appendix J.

As specified at the beginning of the present work, Π is the state-transition function, which is defined in terms of Ω, the block finalisation function and Υ, the transaction-evaluation function, both now well-defined.

As previously detailed, $\mathbf{R}[n]_{\boldsymbol{\sigma}}$, $\mathbf{R}[n]_{\mathrm{l}}$ and $\mathbf{R}[n]_u$ are the nth corresponding states, logs and cumulative mana used after each transaction ($\mathbf{R}[n]_b$, the fourth component in the tuple, has already been defined in terms of the logs). The former is defined simply as the state resulting from applying the corresponding transaction to the state resulting from the previous transaction (or the block's initial state in the case of the first such transaction):

$$(164) \qquad \mathbf{R}[n]_{\boldsymbol{\sigma}} = \begin{cases} \Gamma(B) & \text{if} \quad n < 0 \\ \Upsilon(\mathbf{R}[n-1]_{\boldsymbol{\sigma}}, B_{\mathbf{T}}[n]) & \text{otherwise} \end{cases}$$

In the case of $B_{\mathbf{R}}[n]_u$, we take a similar approach defining each item as the mana used in evaluating the corresponding transaction summed with the previous item (or zero, if it is the first), giving us a running total:

$$(165) \qquad \mathbf{R}[n]_u = \begin{cases} 0 & \text{if} \quad n < 0 \\ \Upsilon^g(\mathbf{R}[n-1]_{\boldsymbol{\sigma}}, B_{\mathbf{T}}[n]) & \\ \quad + \mathbf{R}[n-1]_u & \text{otherwise} \end{cases}$$

For $\mathbf{R}[n]_{\mathrm{l}}$, we utilise the Υ^{l} function that we conveniently defined in the transaction execution function.

$$(166) \qquad \mathbf{R}[n]_{\mathrm{l}} = \Upsilon^{\mathrm{l}}(\mathbf{R}[n-1]_{\boldsymbol{\sigma}}, B_{\mathbf{T}}[n])$$

Finally, we define Π as the new state given the block reward function Ω applied to the final transaction's resultant state, $\ell(B_{\mathbf{R}})_{\boldsymbol{\sigma}}$:

$$(167) \qquad \Pi(\boldsymbol{\sigma}, B) \equiv \Omega(B, \ell(\mathbf{R})_{\boldsymbol{\sigma}})$$

Thus the complete block-transition mechanism, less `PoW`, the proof-of-work function is defined.

12.5. **Mining Proof-of-Work.** The mining proof-of-work (PoW) exists as a cryptographically secure nonce that proves beyond reasonable doubt that a particular amount of computation has been expended in the determination of some token value n. It is utilised to enforce the blockchain security by giving meaning and credence to the notion of difficulty (and, by extension, total difficulty). However, since mining new blocks comes with an attached reward, the proof-of-work not only functions as a method of securing confidence that the blockchain will remain canonical into the future, but also as a wealth distribution mechanism.

For both reasons, there are two important goals of the proof-of-work function; firstly, it should be as accessible as possible to as many people as possible. The requirement of, or reward from, specialised and uncommon hardware should be minimised. This makes the distribution model as open as possible, and, ideally, makes the act of mining a simple swap from electricity to Trust at roughly the same rate for anyone around the world.

Secondly, it should not be possible to make super-linear profits, and especially not so with a high initial barrier. Such a mechanism allows a well-funded adversary to gain a troublesome amount of the network's total mining power and as such gives them a super-linear reward (thus skewing distribution in their favour) as well as reducing the network security.

One plague of the Bitcoin world is ASICs. These are specialised pieces of compute hardware that exist only to do a single task. In Bitcoin's case the task is the SHA256 hash function. While ASICs exist for a proof-of-work function, both goals are placed in jeopardy. Because of this, a proof-of-work function that is ASIC-resistant (i.e. difficult or economically inefficient to implement in specialised compute hardware) has been identified as the proverbial silver bullet.

Two directions exist for ASIC resistance; firstly make it sequential memory-hard, i.e. engineer the function such that the determination of the nonce requires a lot of memory and bandwidth such that the memory cannot be used in parallel to discover multiple nonces simultaneously. The second is to make the type of computation it would need to do general-purpose; the meaning of "specialised hardware" for a general-purpose task set is, naturally, general purpose hardware and as such commodity desktop computers are likely to be pretty close to "specialised hardware" for the task. For Trustmachine 1.0 we have chosen the first path.

More formally, the proof-of-work function takes the form of `PoW`:

(168)
$$m = H_m \quad \land \quad n \leqslant \frac{2^{256}}{H_d} \quad \text{with} \quad (m, n) = \texttt{PoW}(H_{\not{n}}, H_n, \mathbf{d})$$

Where $H_{\not{n}}$ is the new block's header but *without* the nonce and mix-hash components; H_n is the nonce of the header; $\mathbf{d}$ is a large data set needed to compute the mixHash and H_d is the new block's difficulty value (i.e. the block difficulty from section 17.1.3). `PoW` is the proof-of-work function which evaluates to an array with the first item being the mixHash and the second item being a pseudo-random number cryptographically dependent on H and $\mathbf{d}$. The underlying algorithm is called Ethash and is described below.

12.5.1. *Ethash.* Ethash is the PoW algorithm for Trustmachine 1.0. It is the latest version of Dagger-Hashimoto, introduced by Buterin [2013b] and Dryja [2014], although it can no longer appropriately be called that since many of the original features of both algorithms have been drastically changed in the last month of research and development. The general route that the algorithm takes is as follows:

There exists a seed which can be computed for each block by scanning through the block headers up until that point. From the seed, one can compute a pseudorandom cache, $J_{cacheinit}$ bytes in initial size. Light clients store the cache. From the cache, we can generate a dataset, $J_{datasetinit}$ bytes in initial size, with the property that each item in the dataset depends on only a small number of items from the cache. Full clients and miners store the dataset. The dataset grows linearly with time.

Mining involves grabbing random slices of the dataset and hashing them together. Verification can be done with low memory by using the cache to regenerate the specific pieces of the dataset that you need, so you only need to store the cache. The large dataset is updated once every J_{epoch} blocks, so the vast majority of a miner's effort will be reading the dataset, not making changes to it. The mentioned parameters as well as the algorithm is explained in detail in appendix J.

13. IMPLEMENTING CONTRACTS

There are several patterns of contracts engineering that allow particular useful behaviours; two of these that I will briefly discuss are data feeds and random numbers.

13.1. **Data Feeds.** A data feed contract is one which provides a single service: it gives access to information from the external world within Trustmachine. The accuracy and timeliness of this information is not guaranteed and it is the task of a secondary contract author—the contract that utilises the data feed—to determine how much trust can be placed in any single data feed.

The general pattern involves a single contract within Trustmachine which, when given a message call, replies with some timely information concerning an external phenomenon. An example might be the local temperature of New York City. This would be implemented as a contract that returned that value of some known point in storage. Of course this point in storage must be maintained with the correct such temperature, and thus the second part of the pattern would be for an external server to run a Trustmachine node, and immediately on discovery of a new block, creates a new valid transaction, sent to the contract, updating said value in storage. The contract's code would accept such updates only from the identity contained on said server.

13.2. **Random Numbers.** Providing random numbers within a deterministic system is, naturally, an impossible task. However, we can approximate with pseudo-random numbers by utilising data which is generally unknowable at the time of transacting. Such data might include the block's hash, the block's timestamp and the block's beneficiary address. In order to make it hard for malicious miner to control those values, one should use the BLOCKHASH operation in order to use hashes of the previous 256 blocks as pseudo-random numbers. For a series of such numbers, a trivial solution would be to add some constant amount and hashing the result.

14. SCALABILITY

14.1. **Light Clients.**

14.1.1. *Light Subprotocol.* Light client is a way of interacting with a blockchain that only requires a very small amount of computational resources, keeping track of only the block headers of the chain by default and acquiring any needed information about transactions, state or receipts by asking for and verifying Merkle proofs of the relevant data on an as-needed basis. The light subprotocol is the protocol used by "light" clients, which only download block headers as they appear and fetch other parts of the blockchain on-demand. This provides full functionality in terms of safely accessing the blockchain, but do not mine and therefore do not take part in the consensus process. Full and archive nodes can also support the light subprotocol in order to be able to serve light nodes. It has been decided to create a separate sub-protocol in order to avoid interference with the consensus-critical mainnet and make it easier to update during the development phase. Via introducing the Canonical Hash Trie (CHT) that maps the historical block numbers to their canonical hashes in a merkle trie, the block headers themselves in favor of a trie root which encompasses the accumulation of their hashes could be discarded and proofs that a specific block hash is the one that verified earlier could be fetched. ReqId is also introduced for coordinating request packets with responses to enable multiple requests of a peer at a time. The CHT is generated once every 2048 blocks and a 32-byte trie root would be stored for every range. This makes it a little more difficult to bootstrap a client with a checkpoint, but the "CHT-chain" would be kept up to date on the light nodes without imposing any additional overhead on full nodes. Light subprotocol also enables fast syncing and warp syncing, which solved another specific part of the scaling problem: initial full node synchronization. Mimblewimble and strategies based on ZK-SNARKs are another two existing approaches that use advanced cryptography to help.

14.2. **Horizontal Scaling.** Horizontal scalability is an experimental effort of processing transactions and executing smart contracts simultanesouly and concurrently via spliting data into multiple channels or applications. The fundamental protocol might be redesigned for the need of significantly scaling without sacrificing security while retaining the decentralization.

14.3. **Vertical Scaling.** We wish to operate clusters with hundreds of thousands of cores in a decentalized OS-like trustmachine, all connected with specialized network faciliates. We assume the entire system will stay secure through organizing all these in a compartmentated virtually separated environment and all connected with the security-enhanced minimized protocols like TCP/IP, to reduce attack factors after introducing great complexities such as accounts, authentication, databases, asynchronous communication and the scheduling of applications.

14.4. **Truebit.** Truebit consists of a financial incentive layer atop a dispute resolution layer where the latter takes form of a versatile verification game. In addition to secure outsourced computation, immediate applications include decentralized mining pools whose operator is an Ethereum smart contract, a cryptocurrency with scalable transaction throughput, and a trustless means for transferring currency between disjoint cryptocurrency systems. Its design allows for compact proofs to be created to submit to the blockchain, so nearly all of the heavy lifting done by the TrueBit paper and team is directly applicable in this design. The use of the Verification Game, which generates merklized proofs provides increased benefits with reducing the scale of computation. Similar assumptions as TrueBit applies, namely that computational state must be computable and broadcastable online (large pieces of data must be split in multiple rounds), data availability problems needs to be mitigated, failure must be disclosed.

14.5. **Sharding.**

14.5.1. *The Trilemma Of Decentralization, Scalability And Security.* Currently, each node stores all states and processes all transactions. This provides a large amount of security, but greatly limits scalability: a blockchain cannot process more transactions than a single node can. Several methods are considered flawed such as increasing the throughput via using multiple chains to settle transacions, increasing the block size(eg: bitcoin) or "merged mining", all suffered from the issue of increasing the computational and storage load on each "miner" by a factor of N at the cost of a consequencial N-factor decrease. "Miners" that secure the network are not "tied together" and only a small amount of economic incentive is required to convince them into abandoning or compromising one specific chain. The trilemma claims that blockchain systems can only at most have two of the following three properties:

- Decentralization (defined as the system being able to run in a scenario where each participant only has access to $O(c)$ resources, ie. a regular laptop or small VPS)
- Scalability (defined as being able to process $O(n) > O(c)$ transactions)
- Security (defined as being secure against attackers with up to $O(n)$ resources)

Here c will be used to refer to the size of computational resources (including computation, bandwidth and storage) available to each node, and n to refer to the size of the ecosystem in some abstract sense; we assume that transaction load, state size, and the market cap of a cryptocurrency are all proportional to n. Such claim backed by Metcalfes law assuming the marketcap of cryptocurrency should be proportional to n^2 instead of n is flawed. Because once a system gets bigger, two effects slow the growth down. Since in practice, empirical research suggests that the value of a network with n users is close to n^2 proportionality for small values of n and $(nlogn)$ proportionality for large values of n. Hence, assuming $n = O(k * log(k))$ and basing everything off of n (size of the ecosystem) and c (a single nodes computing power) is a perfectly fine model for us to use.

14.5.2. *State Size Growth And Transaction Processing Limits.* Without either forcing every node to be a supercomputer or forcing every node to store a terabyte of state data(aka: vertical scaling), it requires a comprehensive solution where the workloads of state storage, transaction processing and even transaction downloading and re-broadcasting are all spread out across nodes. Thus changes at the P2P level is required, as a broadcast model is not scalable since it requires every node to download and re-broadcast $O(n)$ data (every transaction that is being sent), whereas our decentralization criterion assumes that every node only has access to $O(c)$ resources of all kinds. Bitcoin-NG-style approaches alleviate the problem via increasing the portion of computational time verifying the blocks. However, all solutions mentioned above do little or nothing to alleviate state size growth or the limits of online transaction processing. Particularly, note that if an attacker comes up with worst-case transactions whose ratio between processing time and block space expenditure (bytes, gas, etc) is much higher than usual, then the system will experience very low performance, and so a safety factor is necessary to account for this possibility. In traditional blockchains, the fact that block processing only takes 1-5% of block time has the primary role of protecting against centralization risk but serves double duty of protecting against denial of service risk. In the specific case of Bitcoin, its current worst-case known quadratic execution vulnerability arguably limits any scaling at present to 5-10x, and in the case of Ethereum, while all known vulnerabilities are being or have been removed after the denial-of-service attacks, there is still a risk of further discrepancies particularly on a smaller scale. In Bitcoin NG, the need for the former is removed, but the need for the latter is still there. A further reason to be cautious is that increased state size corresponds to reduced throughput, as nodes will find it harder and harder to keep state data in RAM and so need more and more disk accesses, and databases, which often have an $O(log(n))$ access time, will take longer and longer to access. This was an important lesson from the last Ethereum denial-of-service attack, which bloated the state by 10 GB by creating empty accounts and thereby indirectly slowed processing down by forcing further state accesses to hit disk instead of RAM. In sharded blockchains, there may not necessarily be in-lockstep consensus on a single global state, and so the protocol never asks nodes to try to compute a global state root; in fact, in the protocols presented in later sections, each shard has its own state, and for each shard there is a mechanism for committing to the state root for that shard, which represents that shards state

14.5.3. *The Basic Design Of Sharded Blockchain.* In the design of sharding, state is splited up into $K = O(n/c)$ partitions called shards. For example, a sharding scheme might put all addresses starting with "0x00" into one shard, all addresses starting with "0x01" into another shard, etc. In the simplest form of sharding, each shard also has its own transaction history, and the effect of transactions in some shard k are limited to the state of shard k. One simple example would be a multi-asset blockchain, where there are K shards and each shard stores the balances and processes the transactions associated with one particular asset. In more advanced forms of sharding, some form of cross-shard communication capability, where transactions on one shard can trigger events on other shards, is also included. There exist nodes called collators that accept transactions on shard k (depending on the protocol, collators either choose which k or are randomly assigned some k) and create collations. A collation has a collation header, a short message of the form "This is a collation of transactions on shard k. It expects the previous state root of shard k to be 0x12bc57, the Merkle root of the transactions in this collation is 0x3f98ea, and the state root after processing these transactions should be 0x5d0cc1. Signed, collators "1, 2, 4, 5, 6, 8, 11, 13 ... 98, 99". A block must then contain a collation header for each shard. A block is valid if:

- The pre-state root given in each collation matches the current state root of the associated shard
- All transactions in all collations are valid
- The post-state root given in the collation matches the result of executing the transactions in the collation on top of the given pre-state
- The collation is signed by at least two thirds of the collators registered for that shard

Note that there are now several "levels" of nodes that can exist in such a system:

- Super-full node - processes all transactions in all collations and maintains the full state for all shards.
- Top-level node - processes all top-level blocks, but does not process or try to download the transactions in each collation. Instead, it accepts it on faith that a collation is valid if two thirds of the collators in that shard say that it is valid.
- Single-shard node - acts as a top-level node, but also processes all transactions and maintains the full state for some specific shard.
- Light node - downloads and verifies the block headers of the top-level blocks only; does not process any collation headers or transactions unless it needs to read some specific entry in the state of some specific shard, in which case it downloads the Merkle branch to the most recent collation header for that shard and from there downloads the Merkle proof of the desired value in the state.

14.5.4. *Challenges Of Sharding Design.*

- Cross shard communication - the above design supports no cross-shard communication. How do we add cross-shard communication safely?
- Single-shard takeover attacks - what if an attacker takes over the majority of the collators in one single shard,

either to prevent any collations from getting enough signatures or, worse, to submit collations that are invalid?
- Fraud detection - if an invalid collation does get made, how can nodes (including light nodes) be reliably informed of this so that they can verify the fraud and reject the collation if it is truly fraudulent?
- The data availability problem - as a subset of fraud detection, what about the specific case where data is missing from a collation?
- Superquadratic sharding - in the special case where $n > c^2$, in the simple design given above there would be more than $O(c)$ collation headers, and so an ordinary node would not be able to process even just the top-level blocks. Hence, more than two levels of indirection between transactions and top-level block headers are required (ie. we need "shards of shards"). What is the simplest and best way to do this?

However, the effect of a transaction may depend on events that earlier took place in other shards; a canonical example is transfer of money, where money can be moved from shard i to shard j by first creating a debit transaction that destroys coins in shard i, and then creating a credit transaction that creates coins in shard j, pointing to a receipt created by the debit transaction as proof that the credit is legitimate.

14.5.5. *Cross-Shard Communication.* Cross-shard communication is facilitated via breaking up each transaction into a "debit" and a "credit". For example, suppose that we have a transaction where account A on shard M wishes to send 100 coins to account B on shard N. The steps would looks as follows:

- Send a transaction on shard M which (i) deducts the balance of A by 100 coins, and (ii) creates a receipt. A receipt is an object which is not saved in the state directly, but where the fact that the receipt was generated can be verified via a Merkle proof.
- Wait for the first transaction to be included (sometimes waiting for finalization is required; this depends on the system).
- Send a transaction on shard N which includes the Merkle proof of the receipt from (1). This transaction also checks in the state of shard N to make sure that this receipt is "unspent"; if it is, then it increases the balance of B by 100 coins, and saves in the state that the receipt is spent.
- Optionally, the transaction in (3) also saves a receipt, which can then be used to perform further actions on shard M that are contingent on the original operation succeeding.

However, most scenarios individually do not have too many users, and only very occasionally and loosely interact with each other; in this case, applications can live on separate shards and thus requires no cross-shard interaction at all.

If applications do need to talk to each other, the challenge is much easier if the interaction can be made asynchronous - that is, if the interaction can be done in the form of the application on shard A generating a receipt, a transaction on shard B consuming the receipt and performing some action based on it, and possibly sending a callback to shard A containing some response. Generalizing this pattern is easy, and is not difficult to incorporate into a high-level programming language.

Note that the in-protocol mechanisms that used for asynchronous cross-shard communication would be different and have weaker functionality compared to the mechanisms that are available for intra-shard communication. Some of the functionality that is currently available in non-scalable blockchains would, in a scalable blockchain, only be available for intra-shard communication.

14.5.6. *Cross-Shard Atomic Synchronous Transactions.* The problem of asynchronousness of messages on different shards is decidedly nontrivially resolved by the design of cross-shard atomic synchronous transactions, which are indivisible and irreducible series of operations such that either all occur, or nothing occurs.

If an individual application has more than $O(c)$ usage, then that application would need to exist across multiple chains. The feasibility of doing this depends on the specifics of the application itself; some applications (eg. currencies) are easily parallelizable, whereas others (eg. certain kinds of market designs) cannot be parallelized and must be processed serially.

There are properties of sharded blockchains that is known to be impossible to achieve. Amdahls law states that in any scenario where applications have any non-parallelizable component, once parallelization is easily available the non-parallelizable component quickly becomes the bottleneck. In a general computation platform like Ethereum, it is easy to come up with examples of non-parallelizable computation: a contract that keeps track of an internal value x and sets $x = sha3(x, tx_data)$ upon receiving a transaction is a simple example. No sharding scheme can give individual applications of this form more than $O(c)$ performance. Hence, it is likely that over time sharded blockchain protocols will get better and better at being able to handle a more and more diverse set of application types and application interactions, but a sharded architecture will always necessarily fall behind a single-shard architecture in at least some ways at scales exceeding $O(c)$.

14.5.7. *Synchronous And Semi-Synchronous Cross-shard Messages.* The process becomes much easier if you view the transaction history as being already settled, and are simply trying to calculate the state transition function. There are several approaches; one fairly simple approach can be described as follows:

- A transaction may specify a set of shards that it can operate in
- In order for the transaction to be effective, it must be included at the same block height in all of these shards.
- Transactions within a block must be put in order of their hash (this ensures a canonical order of execution)

A client on shard X, if it sees a transaction with shards (X, Y), requests a Merkle proof from shard Y verifying (i) the presence of that transaction on shard Y, and (ii) what the pre-state on shard Y is for those bits of data that the transaction will need to access. It then executes the transaction and commits to the execution result. Note that this process may be highly inefficient if there are many transactions with many different block pairings in each block; for this reason, it may be optimal to simply require blocks to specify sister shards, and then calculation can be done more efficiently at a per-block level. This is the basis for how such a scheme could work; one could imagine more complex designs. However, when making a new design, its always important to make sure that low-cost denial of service attacks cannot arbitrarily slow state calculation down.

Vlad Zamfir created a scheme by which asynchronous messages could still solve the book a train and hotel problem. This works as follows. The state keeps track of all operations that have been recently made, as well as the graph of which operations were triggered by any given operation (including cross-shard operations). If an operation is reverted, then a receipt is created which can then be used to revert any effect of that operation on other shards; those reverts may then trigger their own reverts and so forth. The argument is that if one biases the system so that revert messages can propagate twice as fast as other kinds of messages, then a complex cross-shard transaction that finishes executing in K rounds can be fully reverted in another K rounds.

The overhead that this scheme would introduce has arguably not been sufficiently studied; there may be worst-case scenarios that trigger quadratic execution vulnerabilities. It is clear that if transactions have effects that are more isolated from each other, the overhead of this mechanism is lower; perhaps isolated executions can be incentivized via favorable mana cost rules. All in all, this is one of the more promising research directions for advanced sharding.

14.5.8. *Guaranteed Cross-Shard Calls.* One of the challenges in sharding is that when a call is made, there is by default no hard protocol-provided guarantee that any asynchronous operations created by that call will be made within any particular timeframe, or even made at all; rather, it is up to some party to send a transaction in the destination shard triggering the receipt. This is okay for many applications, but in some cases it may be problematic for several reasons:

- There may be no single party that is clearly incentivized to trigger a given receipt. If the sending of a transaction benefits many parties, then there could be tragedy-of-the-commons effects where the parties try to wait longer until someone else sends the transaction (ie. play "chicken"), or simply decide that sending the transaction is not worth the transaction fees for them individually.
- Mana prices across shards may be volatile, and in some cases performing the first half of an operation compels the user to follow through on it, but the user may have to end up following through at a much higher mana price. This may be exacerbated by DoS attacks and related forms of griefing.
- Some applications rely on there being an upper bound on the latency of cross-shard messages (eg. the train-and-hotel example). Lacking hard guarantees, such applications would have to have inefficiently large safety margins.

One could try to come up with a system where asynchronous messages made in some shard automatically trigger effects in their destination shard after some number of blocks. However, this requires every client on each shard to actively inspect all other shards in the process of calculating the state transition function, which is arguably a source of inefficiency. The best known compromise approach is this: when a receipt from shard A at height height_a is included in shard B at height height_b, if the difference in block heights exceeds MAX_HEIGHT, then all validators in shard B that created blocks from height_a + MAX_HEIGHT + 1 to height_b - 1 are penalized, and this penalty increases exponentially. A portion of these penalties is given to the validator that finally includes the block as a reward. This keeps the state transition function simple, while still strongly incentivizing the correct behavior.

14.5.9. *Attack On Cross-Shard Call.* If an attacker sends a cross-shard call from every shard into shard X at the same time, it might be mathematically impossible to include all of these calls in time. Here is a proposed solution. In order to make a cross-shard call from shard A to shard B, the caller must pre-purchase congealed shard B mana (this is done via a transaction in shard B, and recorded in shard B). Congealed shard B mana has a fast demurrage rate: once ordered, it loses $1/k$ of its remaining potency every block. A transaction on shard A can then send the congealed shard B mana along with the receipt that it creates, and it can be used on shard B for free. Shard B blocks allocate extra mana space specifically for these kinds of transactions. Note that because of the demurrage rules, there can be at most $MANA\_LIMIT * k$ worth of congealed mana for a given shard available at any time, which can certainly be filled within k blocks (in fact, even faster due to demurrage, but we may need this slack space due to malicious validators). In case too many validators maliciously fail to include receipts, we can make the penalties fairer by exempting validators who fill up the receipt space of their blocks with as many receipts as possible, starting with the oldest ones.

Under this pre-purchase mechanism, a user that wants to perform a cross-shard operation would first pre-purchase mana for all shards that the operation would go into, over-purchasing to take into account the demurrage. If the operation would create a receipt that triggers an operation that consumes 100000 mana in shard B, the user would pre-buy $100000 * e$ (ie. 271818) shard-B congealed mana. If that operation would in turn spend 100000 mana in shard C (ie. two levels of indirection), the user would need to pre-buy $100000 * e^2$ (ie. 738906) shard-C congealed mana. Notice how once the purchases are confirmed, and the user starts the main operation, the user can be confident that they will be insulated from changes in the mana price market, unless validators voluntarily lose large quantities of money from receipt non-inclusion penalties.

14.5.10. *Congealed mana.* One could buy congealed shard A gas inside of shard A, and send a guaranteed cross-shard call from shard A to itself. Though note that this scheme would only support scheduling at very short time intervals, and the scheduling would not be exact to the block; it would only be guaranteed to happen within some period of time.

14.5.11. *Sharding Security Models.* There are several competing models under which the safety of blockchain designs is evaluated:

- Honest majority (or honest supermajority): we assume that there is some set of validators and up to $\frac{1}{2}$ (or $\frac{1}{3}$ or $\frac{1}{4}$ of those validators are controlled by an attacker, and the remaining validators honestly follow the protocol
- Uncoordinated majority: we assume that all validators are rational in a game-theoretic sense (except the attacker, who is motivated to make the network fail in some way), but no more than some fraction (often between $\frac{1}{4}$ and $\frac{1}{2}$ are capable of coordinating their actions.
- Coordinated choice: we assume that all validators are controlled by the same actor, or are fully capable of coordinating on the economically optimal choice between themselves. We can talk about the cost to the coalition (or profit to the coalition) of achieving some undesirable outcome.
- Bribing attacker model: we take the uncoordinated majority model, but instead of making the attacker be one of the participants, the attacker sits outside the protocol, and has the ability to bribe any participants to change their behavior. Attackers are modeled as having a budget, which is the maximum that they are willing to pay, and we can talk about their cost, the amount that they end up paying to disrupt the protocol equilibrium.

Bitcoin proof of work with Eyal and Sirers selfish mining fix is robust up to under the honest majority assumption, and up to $\frac{1}{4}$ under the uncoordinated majority assumption. Schellingcoin is robust up to $\frac{1}{2}$ under the honest majority and uncoordinated majority assumptions, has ϵ (ie. slightly more than zero) cost of attack in a coordinated choice model, and has a $P + \epsilon$ budget requirement and cost in a bribing attacker model due to P + ϵ attacks.

Hybrid models also exist; for example, even in the coordinated choice and bribing attacker models, it is common to make an honest minority assumption that some portion (perhaps 1-15%) of validators will act altruistically regardless of incentives. We can also talk about coalitions consisting of between 50-99% of validators either trying to disrupt the protocol or harm other validators; for example, in proof of work, a 51%-sized coalition can double its revenue by refusing to include blocks from all other miners.

The honest majority model is arguably highly unrealistic and has already been empirically disproven - see Bitcoin's SPV mining fork for a practical example. It proves too much: for example, an honest majority model would imply that honest miners are willing to voluntarily burn their own money if doing so punishes attackers in some way. The uncoordinated majority assumption may be realistic; there is also an intermediate model where the majority of nodes is honest but has a budget, so they shut down if they start to lose too much money.

The bribing attacker model has in some cases been criticized as being unrealistically adversarial, although its proponents argue that if a protocol is designed with the bribing attacker model in mind then it should be able to massively reduce the cost of consensus, as 51% attacks become an event that could be recovered from. We will evaluate sharding in the context of both uncoordinated majority and bribing attacker models.

14.5.12. *Solution To Single Shard Takeover.* In short, random sampling. Each shard is assigned a certain number of collators (eg. 150), and the collators that approve blocks on each shard are taken from the sample for that shard. Samples can be reshuffled either semi-frequently (eg. once every 12 hours) or maximally frequently (ie. there is no real independent sampling process, collators are randomly selected for each shard from a global pool every block).

The result is that even though only a few nodes are verifying and creating blocks on each shard at any given time, the level of security is in fact not much lower, in an honest/uncoordinated majority model, than what it would be if every single node was verifying and creating blocks. The reason is simple statistics: if you assume a $\frac{2}{3}$ honest supermajority on the global set, and if the size of the sample is 150, then with 99.999% probability the honest majority condition will be satisfied on the sample. If you assume a $\frac{3}{4}$ honest supermajority on the global set, then that probability increases to 99.999999998% (see here for calculation details).

Hence, at least in the honest / uncoordinated majority setting, we have:

- Decentralization (each node stores only $O(c)$ data, as its a light client in $O(c)$ shards and so stores $O(1)*O(c) = O(c)$ data worth of block headers, as well as $O(c)$ data corresponding to the full state and recent history of one or several shards that it is assigned to at the present time)
- Scalability (with $O(c)$ shards, each shard having O(c) capacity, the maximum capacity is $n = O(c^2)$)
- Security (attackers need to control at least $\frac{1}{3}$ of the entire O(n)-sized validator pool in order to stand a chance of taking over the network).

In the Zamfir model (or alternatively, in the very very adaptive adversary model), things are not so easy, but we will get to this later. Note that because of the imperfections of sampling, the security threshold does decrease from $\frac{1}{2}$ to $\frac{1}{3}$, but this is still a surprisingly low loss of security for what may be a 100-1000x gain in scalability with no loss of decentralization.

14.5.13. *Sampling In POW And POS.* In proof of stake, it is easy. There already is an active validator set that is kept track of in the state, and one can simply sample from this set directly. Either an in-protocol algorithm runs and chooses 150 validators for each shard, or each validator independently runs an algorithm that uses a common source of randomness to (provably) determine which shard they are at any given time. Note that it is very important that the sampling assignment is compulsory; validators do not have a choice of what shard they go into. If validators could choose, then attackers with small total stake could concentrate their stake onto one shard and attack it, thereby eliminating the systems security.

In proof of work, it is more difficult, as with direct proof of work schemes one cannot prevent miners from applying their work to a given shard. It may be possible to use proof-of-file-access forms of proof of work to lock individual miners to individual shards, but it is hard to ensure that miners cannot quickly download or generate data that can be used for other shards and thus circumvent such a mechanism. The best known approach is through a technique invented by Dominic Williams called puzzle towers, where miners first perform proof of work on a common chain, which then inducts them into a proof of stake-style validator pool, and the validator pool is then sampled just as in the proof-of-stake case.

One possible intermediate route might look as follows. Miners can spend a large $(O(c) - sized)$ amount of work to create a new cryptographic identity. The precise value of the proof of work solution then chooses which shard they have to make their next block on. They can then spend an $O(1) - sized$ amount of work to create a block on that shard, and the value of that proof of work solution determines which shard they can work on next, and so on8. Note that all of these approaches make proof of work stateful in some way; the necessity of this is fundamental.

The probabilities given are for one single shard; however, the random seed affects $O(c)$ shards and the attacker could potentially take over any one of them. If we want to look at $O(c)$ shards simultaneously, then there are two cases. First, if the grinding process is computationally bounded, then this fact does not change the calculus at all, as even though there are now $O(c)$ chances of success per round, checking success takes $O(c)$ times as much work. Second, if the grinding process is economically bounded, then this indeed calls for somewhat higher safety factors (increasing N by 10-20 should be sufficient) although its important to note that the goal of an attacker in a profit-motivated manipulation attack is to increase their participation across all shards in any case, and so that is the case that we are already investigating.

14.5.14. *Tradeoffs In Making Sampling More Or Less Frequent.* Selection frequency affects just how adaptive adversaries can be for the protocol to still be secure against them; for example, if you believe that an adaptive attack (eg. dishonest validators who discover that they are part of the same sample banding together and colluding) can happen in 6 hours but not less, then you would be okay with a sampling time of 4 hours but not 12 hours. This is an argument in favor of making sampling happen as quickly as possible.

The main challenge with sampling taking place every block is that reshuffling carries a very high amount of overhead. Specifically, verifying a block on a shard requires knowing the state of that shard, and so every time validators are reshuffled, validators need to download the entire state for the new shard(s) that they are in. This requires both a strong state size control policy (ie. economically ensuring that the size of the state does not grow too large, whether by deleting old accounts, restricting the rate of creating new accounts or a combination of the two) and a fairly long reshuffling time to work well.

Currently, the Parity client can download and verify a full Ethereum state snapshot via warp-sync in 3 minutes; if we increase by 20x to compensate for increased usage (10 tx/sec instead of the current 0.5 tx/sec) (well assume future state size control policies and dust accumulated from longer-term usage roughly cancel out) , we get 60 minute state sync time, suggesting that sync periods of 12-24 hours but not less are safe.

There are two possible paths to overcoming this challenge. The subset of objects in a Merkle tree that would need to be provided in a Merkle proof of a transaction that accesses several state objects

Implementing this scheme in its pure form has two flaws. First, it introduces $O(log(n))$ overhead, although one could argue that this $O(log(n))$ overhead is not as bad as it seems because it ensures that the validator can always simply keep state data in memory and thus it never needs to deal with the overhead of accessing the hard drive. Second, it can easily be applied if the addresses that are accessed by a transaction are static, but is more difficult to apply if the addresses in question are dynamic - that is, if the transaction execution has code of the form $read(f(read(x)))$ where the address of some state read depends on the execution result of some other state read. In this case, the address that the transaction sender thinks the transaction will be reading at the time that they send the transaction may well differ from the address that is actually read when the transaction is included in a block, and so the Merkle proof may be insufficient.

A compromise approach is to allow transaction senders to send a proof that incorporates the most likely possibilities for what data would be accessed; if the proof is sufficient, then the transaction will be accepted, and if the state unexpectedly changes and the proof is insufficient then either the sender must resend or some helper node in the network resends the transaction adding the correct proof. Developers would then be free to make transactions that have dynamic behavior, but the more dynamic the behavior gets the less likely transactions would be to actually get included into blocks.

Note that validators transaction inclusion strategies under this approach would need to be complicated, as they may spend millions of mana processing a transaction only to realize that the last step accesses some state entry that they do not have. One possible compromise is for validators to have a strategy that accepts only (i) transactions with very low mana costs, eg. $< 100k$, and (ii) transactions that statically specify a set of contracts that they are allowed to access, and contain proofs for the entire state of those contracts. Note that this only applies when transactions are initially broadcasted; once a transaction is included into a block, the order of execution is fixed, and so only the minimal Merkle proof corresponding to the state that actually needs to be accessed can be provided.

If validators are not reshuffled immediately, there is one further opportunity to increase efficiency. We can expect validators to store data from proofs of transactions that have already been processed, so that that data does not need to be sent again; if k transactions are sent within one reshuffling period, then this decreases the average size of a Merkle proof from $log(n)$ to $log(n) - log(k)$.

14.5.15. *User Side Validation.* The techniques here tend to involve requiring users to store state data and provide Merkle proofs along with every transaction that they send. A transaction would be sent along with a Merkle proof-of-correct-execution, and this proof would allow a node that only has the state root to calculate the new state root. This proof-of-correct-execution would consist of the subset of objects in the trie that would need to be traversed to access and verify the state information that the transaction must verify; because Merkle proofs are $O(log(n))$ sized, the proof for a transaction that accesses a constant number of objects would also be $O(log(n))$ sized.

14.5.16. *Randomness for Random Sampling.* First of all, it is important to note that even if random number generation is heavily exploitable, this is not a fatal flaw for the protocol; rather, it simply means that there is a medium to high centralization incentive. The reason is that because the randomness is picking fairly large samples, it is difficult to bias the randomness by more than a certain amount.

The simplest way to show this is through the binomial distribution, as described above; if one wishes to avoid a sample of size N being more than 50% corrupted by an attacker, and an attacker has p% of the global stake pool, the chance of the attacker being able to get such a majority during one round is:

$$\sum_{k=\frac{N}{2}}^{N} p^k(1-p)^{N-k} \left(\begin{array}{c} N \\ k \end{array} \right)$$

When $N >= 150W$,the chance that any given random seed will lead to a sample favoring the attacker is very small. This means from the perspective of security of randomness is that the attacker needs to have a very large amount of freedom in choosing the random values order to break the sampling process outright. Most vulnerabilities in proof-of-stake randomness do not allow the attacker to simply choose a seed; at worst, they give the attacker many chances to select the most favorable seed out of many pseudorandomly generated options. If one is very worried about this, one can simply set N to a greater value, and add a moderately hard key-derivation function to the process of computing the randomness, so that it takes more than 2^100 computational steps to find a way to bias the randomness sufficiently.

For purposes of profit rather than outright takeover, there exists some risks of attacks being made that try to influence the randomness more marginally. For example, suppose that there is an algorithm which pseudorandomly selects 1000 validators out of some very large set (each validator getting a reward of $1), an attacker has 10% of the stake so the attackers average honest revenue 100, and at a cost of $1 the attacker can manipulate the randomness to re-roll the dice (and the attacker can do this an unlimited number of times).

Due to the central limit theorem, the standard deviation of the number of samples, and based on other known results in math the expected maximum of N random samples is slightly under $M + S * sqrt(2 * log(N))$ where M is the mean and S is the standard deviation. Hence the reward for manipulating the randomness and effectively re-rolling the dice (ie. increasing N) drops off sharply, eg. with 0 re-trials your expected reward is $100, with one re-trial it's $105.5, with two it's $108.5, with three it's $110.3, with four it's $111.6, with five it's $112.6 and with six it's $113.5. Hence, after five retrials it stops being worth it. As a result, an economically motivated attacker with ten percent of stake will (socially wastefully) spend $5 to get an additional revenue of $13, for a net surplus of $8.

However, this kind of logic assumes that one single round of re-rolling the dice is expensive. Many older proof of stake algorithms have a stake grinding vulnerability where re-rolling the dice simply means making a computation locally on ones computer; algorithms with this vulnerability are certainly unacceptable in a sharding context. Newer algorithms (see the validator selection section in the proof of stake FAQ) have the property that re-rolling the dice can only be done by voluntarily giving up ones spot in the block creation process, which entails giving up rewards and fees. The best way to mitigate the impact of marginal economically motivated attacks on sample selection is to find ways to increase this cost. One method to increase the cost by a factor of sqrt(N) from N rounds of voting is the majority-bit method devised by Iddo Bentov; the Mauve Papers sharding algorithm expects to use this approach.

Another form of random number generation that is not exploitable by minority coalitions is the deterministic threshold signature approach most researched and advocated by Dominic Williams. The strategy here is to use a deterministic threshold signature to generate the random seed from which samples are selected. Deterministic threshold signatures have the property that the value is guaranteed to be the same regardless of which of a given set of participants provides their data to the algorithm, provided that at least $\frac{2}{3}$ of participants do participate honestly. This approach is more obviously not economically exploitable and fully resistant to all forms of stake-grinding, but it has several weaknesses:

- It relies on more complex cryptography (specifically, elliptic curves and pairings). Other approaches rely on nothing but the random-oracle assumption for common hash algorithms.
- It fails when many validators are offline. A desired goal for public blockchains is to be able to survive very large portions of the network simultaneously disappearing, as long as a majority of the remaining nodes is honest; deterministic threshold signature schemes at this point cannot provide this property.
- Its not secure in a Zamfir model where more than of validators are colluding. The other approaches described in the proof of stake FAQ above still make it expensive to manipulate the randomness, as data from all validators is mixed into the seed and making any manipulation requires either universal collusion or excluding other validators outright.

One might argue that the deterministic threshold signature approach works better in consistency-favoring contexts and other approaches work better in availability-favoring contexts.

14.5.17. *Concerns Over Sharding And Improvements.* In a bribing attacker or coordinated choice model, the fact that validators are randomly sampled doesnt matter: whatever the sample is, either the attacker can bribe the great majority of the sample to do as the attacker pleases, or the attacker controls a majority of the sample directly and can direct the sample to perform arbitrary actions at low cost ($O(c)$ cost, to be precise).

At that point, the attacker has the ability to conduct 51% attacks against that sample. The threat is further magnified because there is a risk of cross-shard contagion: if the attacker corrupts the state of a shard, the attacker can then start to send unlimited quantities of funds out to other shards and perform other cross-shard mischief. All in all, security in the bribing attacker or coordinated choice model is not much better than that of simply creating $O(c)$ altcoins.

Basically, by comprehensively solving the problem of fraud detection.

One major category of solution to this problem is the use of challenge-response mechanisms. Challenge-response mechanisms generally rely on a principle of escalation: fact X (eg. "collation #17293 in shard #54 is valid") is initially accepted as true if at least k validators sign a claim (backed by a deposit) that it is. However, if this happens, there is some challenge period during which 2k validators can sign a claim stating that it is false. If this happens, 4k validators can sign a claim stating that the claim is in fact true, and so forth until one side either gives up or most validators have signed claims, at which point every validator and client themselves checks whether or not X is true. If X is ruled true, everyone who made a claim saying so is rewarded and everyone who made a contradictory claim is penalized, and vice versa.

Looking at this mechanism, you can prove that malicious actors lose an amount of money proportional to the number of actors that they forced to look at the given piece of data. Forcing all users to look at the data requires a large portion of validators to sign a claim which is false, which can be used to penalize all of them, so the cost of forcing all users to look at a piece of data is O(n); this prevents the challenge-response mechanism from being used as a denial-of-service vector.

14.5.18. *The Data Availability Problem.* There is one trivial attack by which an attacker can always burn $O(c)$ capital to temporarily reduce the quality of a shard: spam it by sending transactions with high transaction fees, forcing legitimate users to outbid you to get in. This attack is unavoidable; you could compensate with flexible mana limits, and you could even try transparent sharding schemes that try to automatically re-allocate nodes to shards based on usage, but if some particular application is non-parallelizable, Amdahls law guarantees that there is nothing you can do. The attack that is opened up here (reminder: it only works in the Zamfir model, not honest/uncoordinated majority) is arguably not substantially worse than the transaction spam attack. Hence, we've reached the known limit for the security of a single shard, and there is no value in trying to go further.

14.5.19. *Instant Shuffling.* First of all, its worth nothing that proof of work and simple proof of stake, even without sharding, both have very low security in a bribing attacker model; a block is only truly finalized in the economic sense after $O(n)$ time (as if only a few blocks have passed, then the economic cost of replacing the chain is simply the cost of starting a double-spend from before the block in question). Casper solves this problem by adding its finality mechanism, so that the economic security margin immediately increases to the maximum. In a sharded chain, if we want economic finality then we need to come up with a chain of reasoning for why a validator would be willing to make a very strong claim on a chain based solely on a random sample, when the validator itself is convinced that the bribing attacker and coordinated choice models may be true and so the random sample could potentially be corrupted.

14.5.20. *Transparent Sharding.* The concept of shards will not be exposed directly to developers, and it does not permanently assign state objects to specific shards. The protocol has an ongoing built-in load-balancing process that shifts objects around between shards. If a shard gets too big or consumes too much mana it can be split in half; if two shards get too small and talk to each other very often they can be combined together; if all shards get too small one shard can be deleted and its contents moved to various other shards, etc. There are some advantages and disadvantages though:

- Developers no longer need to think about shards
- Theres the possibility for shards to adjust manually to changes in mana prices, rather than relying on market mechanics to increase mana prices in some shards more than others
- There is no longer a notion of reliable co-placement: if two contracts are put into the same shard so that they can interact with each other, shard changes may well end up separating them
- More protocol complexity

The co-placement problem can be mitigated by introducing a notion of sequential domains, where contracts may specify that they exist in the same sequential domain, in which case synchronous communication between them will always be possible. In this model a shard can be viewed as a set of sequential domains that are validated together, and where sequential domains can be rebalanced between shards if the protocol determines that it is efficient to do so.

14.5.21. *Majority Collusions Trying To Censor Transactions.* If a user fails to get a transaction in because colluding validators are filtering the transaction and not accepting any blocks that include it, then the user could send a series of messages which trigger a chain of guaranteed scheduled messages, the last of which reconstructs the transaction inside of the TVM and executes it. Preventing such circumvention techniques is practically impossible without shutting down the guaranteed scheduling feature outright and greatly restricting the entire protocol, and so malicious validators would not be able to do it easily.

14.5.22. *Sharded Blockchains.* The schemes described in this document would offer no improvement over non-sharded blockchains; realistically, every shard would end up with some nodes on both sides of the partition. There have been calls (eg. from IPFSs Juan Benet) for building scalable networks with the specific goal that networks can split up into shards as needed and thus continue operating as much as possible under network partition conditions, but there are nontrivial cryptoeconomic challenges in making this work well.

One major challenge is that if we want to have location-based sharding so that geographic network partitions minimally hinder intra-shard cohesion (with the side effect of having very low intra-shard latencies and hence very fast intra-shard block times), then we need to have a way for validators to choose which shards they are participating in. This is dangerous, because it allows for much larger classes of attacks in the honest/uncoordinated majority model, and hence cheaper attacks with higher griefing factors in the Zamfir model. Sharding for geographic partition safety and sharding via random sampling for efficiency are two fundamentally different things.

Second, more thinking would need to go into how applications are organized. A likely model in a sharded blockchain as described above is for each app to be on some shard (at least for small-scale apps); however, if we want the apps themselves to be partition-resistant, then it means that all apps would need to be cross-shard to some extent.

One possible route to solving this is to create a platform that offers both kinds of shards - some shards would be higher-security global shards that are randomly sampled, and other shards would be lower-security local shards that could have properties such as ultra-fast block times and cheaper transaction fees. Very low-security shards could even be used for data-publishing and messaging.

14.5.23. *Challenges Of Pushing Further Scaling.* There are several considerations. First, the algorithm would need to be converted from a two-layer algorithm to a stackable n-layer algorithm; this is possible, but is complex. Second, n/c (ie. the ratio between the total computation load of the network and the capacity of one node) is a value that happens to be close to two constants: first, if measured in blocks, a timespan of several hours, which is an acceptable maximum security confirmation time, and second, the ratio between rewards and deposits (an early computation suggests a 32 token deposit size and a 0.05 token block reward for Casper). The latter has the consequence that if rewards and penalties on a shard are escalated to be on the scale of validator deposits, the cost of continuing an attack on a shard will be O(n) in size.

Going above c^2 would likely entail further weakening the kinds of security guarantees that a system can provide, and allowing attackers to attack individual shards in certain ways for extended periods of time at medium cost, although it should still be possible to prevent invalid state from being finalized and to prevent finalized state from being reverted unless attackers are willing to pay an $O(n)$ cost. However, the rewards are large - a super-quadratically sharded blockchain could be used as a general-purpose tool for nearly all decentralized applications, and could sustain transaction fees that makes its use virtually free.

14.6. **Introducing SUPERNOVΛ.** SUPERNOVΛ is an experimental project that aims to eliminate scalability issues with all the technology available to serve millions of users.

14.7. **Research And Implementation.** Trustmachine is an open project, we welcome everyone to support SUPERNOVΛ.

15. SECURITY

15.1. **Cryptography.**

15.1.1. *Schnorr Signature Overview.* Digital signatures have been around since Diffie and E.Hellman [1976] first described them. Early implementations based on the prime-number based RSA algorithm were dominant in the 90s. In recent years discrete logarithm and elliptic curve approaches have become more popular because they allow for improved computational efficiency and smaller key size without diminishing security.

In the discrete logarithm space, ElGamal-based signature schemes have been the most deployed thanks in large part to the National Institute of Standards and Technology (NIST) releasing their patented DSA under a royalty-free license when implementing the FIPS 186 standard in '93. Even Bitcoin as adopted a variant of DSA: the elliptic-curve based ECDSA, which uses the secp256k1 elliptic curve.

However, DSA isnt the only answer for modern digital signatures. Schnorr signatures are an alternative thats less widely used primarily because they were under patent until 2008. Schnorr signatures offer numerous benefits over traditional methodologies, including:

- Strong Security. They have a stronger security proof.

- Nice Simplicity. Theyre considered the simplest form of digital signature.
- Fast & Efficient. They can be implemented in blindingly quick ways on Intel hardware.
- NIKE (aka "Non-Interactive Key Exchange"). A Diffie-Helman secret between two parties can be created without any additional round-trips.

Schnorr signatures can also be added together in a very simple manner, which creates a number of interesting multisignature (aka "multisig") possibilities. These multisigs look exactly like a single signature, which means among other things that these mulitisig approaches have high privacy and low accountability:

High privacy. Third parties cant see the participants, and only the participants can see the policy.

Low accountability. For an M of N multisig that only requires some parties to sign, you cant see who the signors are. For example, in a 3 of 5 multisig, you can see that 3 of the allowed parties signed, but not who they are.

This high privacy is seen as an almost-universal win, but the low accountability has both advantages and disadvantages. It supports the implementation of group and ring signatures, both of which depend on anonymity, but it might be a deficit when used for financial multisigs. Fortunately, some recent implementations of Schnorr signatures are beginning to demonstrate new ways to adjust these cryptographic choices.

Much of the recent deployment of Schnorr signatures has come thanks to stream of non-NIST-based cryptographic work: in the early 00s, Bernstein [2006] and his team were working on Curve25519. This elliptic curve offers a number of advantages over curve secp256k1 including CM field discriminants, ladders, completeness, and indistinguishability from uniform random strings. Curve25519 also has a birationally equivalent Twisted Edwards Curve that can be implemented very efficiently on Intel hardware.

Bernstein applied his elliptic curve work to Schnorr signatures after their patent expired in order to create a new signature standard: EdDSA, the Edwards-curve Digital Signature Algorithm, as proposed by Josefsson [2016]. The recommended curve for EdDSA is Bernsteins Curve25519; using it results in a variant of EdDSA called Ed25519. It combines the speed, security, and simplicity of Schnorr signatures with the size advantages of Elliptic Curve Cryptography, using all non-NIST algorithms, producing an enviable system for digital signatures.

Since its creation, Ed25519 has been implemented in a few different toolkits. Bernstein produced the first implementation for SUPERCOP. He then created the NaCl crypto library (2012) and the shorter and simpler TweetNaCl (2013), which is intended to be an auditable crypto library in 100 tweets. Meanwhile, user-oriented software has begun to appear based on Ed25519, including the OpenBSD OpenSSH 6.5 (2014) and the miniLock file encryption software (2014).

Though Curve25519 researchers have initiated much of the recent work on Schnorr signatures, its also entering the Bitcoin world thanks to Greg Maxwell and Pieter Wuille who implemented them in libsecp256k1, a library that is planned to replace Bitcoin's use of elliptic curve cryptography. Their Schnorr signature code allows Bitcoin multisigs to support more signers and to be less costly than the Threshold ECDSA signature scheme that was introduced in 2014.

In fact, the Bitcoin community appears to be currently on the leading edge of Schnorr multisig development. Maxwell and Wuille are looking for ways to make Schnorr multisig more accountable. Theyre also trying to solve usability issues, as M of N Schnorr multisigs currently require too many round trips.

Polycheck Signatures offer one intriguing possibility. They implement M of N multisigs as mathematical formulas; the formulas can be manipulated to zero out some of their factors, removing some signers from the (literal) equation.

However, Pieter Wuilles Tree Signatures appear to offer even more capabilities. Tree signatures implement M of N multisigs as a Merkle tree filled with N of N multisigs; together, this combined structure represents many additional options for multisigs, as as "Either A and B signed, or 2 out of C, D and E signed".

15.1.2. *Schnorr Signature Scheme.* Schnorr, named after its inventor Claus-Peter Schnorr, is a digital signature scheme that connect private key, public key and signature with a series of mathematical rules. Its security is based on the intractability of certain discrete logarithm problems and offers a strong level of correctness, they do not suffer from malleability and relatively fast to verify. It is considered the simplest signature scheme that efficiently generates short signatures to be provably secure in a random oracle model, and most importantly, it also supports multiple signatures to be aggregated into a signle new signature. Usually the transactions would all be signed using the ECDSA algorithm with secp256k1 curve.

(A)Choosing parameters:

All users of the signature scheme agree on a group, G, of prime order, q, with generator, g, in which the discrete log problem is assumed to be hard. Typically a Schnorr group is used. All users agree on a cryptographic hash function $H{:}\{0,1\}^* \to \mathbb{Z}q$.

(B)Notation:

- Exponentiation stands for repeated application of the group operation
- Juxtaposition stands for multiplication on the set of congruence classes or application of the group operation (as applicable)
- Subtraction stands for subtraction on set of equivalence groups
- $M \in \{0,1\}^*$, the set of finite bit strings
- $s, e, e_v \in \mathbb{Z}_q$, the set of congruence classes modulo q
- $x, k \in \mathbb{Z}_q^\times$, the multiplicative group of integers modulo q (for prime q, $\{q\}^\times = \mathbb{Z}_q \setminus \overline{0}_q$
- $y, r, r_v \in G$.

(C)Key generation:

- Choose a private signing key, x, from the allowed set.
- The public verification key is $y = g^x$.

(D)Signing: To sign a message, M:

- Choose a random k from the allowed set.
- Let $r = g^k$.
- Let $e = H(r \parallel M)$, where $\parallel$ denotes concatenation and r is represented as a bit string.
- Let $s = k - xe$.

The signature is the pair, (s, e).

Note that $s, e \in \mathbb{Z}_q$; if $q < 2^{160}$, then the signature representation can fit into 40 bytes.

(E)Verifying:

- Let $r_v = g^s y^e$
- Let $e_v = H(r_v \parallel M)$

If $e_v = e$ then the signature is verified.

(F)Proof of correctness:

It is relatively easy to see that $e_v = e$ if the signed message equals the verified message:

$r_v = g^s y^e = g^{k-xe} g^{xe} = g^k = r$, and hence $e_v = H(r_v \parallel M) = H(r \parallel M) = e$.

Public elements: G, g, q, y, s, e, r. Private elements: k, x.

This shows only that a correctly signed message will verify correctly; many other properties are required for a secure signature algorithm.

(G)Security argument:
The signature scheme was constructed by applying the FiatShamir transform to Schnorr's identification protocol. Therefore, (per Fiat and Shamir's arguments), it is secure if H is modeled as a random oracle.

Its security can also be argued in the generic group model, under the assumption that H is "random-prefix preimage resistant" and "random-prefix second-preimage resistant". In particular, H does not need to be collision resistant.

In 2012, Seurin provided an exact proof of the Schnorr signature scheme. In particular, Seurin shows that the security proof using the Forking lemma is the best possible result for any signature schemes based on one-way group homomorphisms including Schnorr-Type signatures and the Guillou-Quisquater signature schemes. Namely, under the ROMDL assumption, any algebraic reduction must lose a factor $f(\epsilon_F)q_h$ in its time-to-success ratio, where $f \leq 1$ is a function that remains close to 1 as long as "$\{\epsilon\}_F$ is noticeably smaller than 1", where $\{\epsilon\}_F$ is the probability of forging an error making at most $q_\{h\}$ queries to the random oracle.

15.1.3. *Schnorr MultiSignature.* Micali et al. [2006] proposed an accountable-subgroup multisignatures scheme based on schnorr signature that is both provable and efficient:

- Only three rounds of communication are required per signature. The signing time per signer is the same as for the single-signer
- Schnorr scheme, regardless of the number of signers.
- The verification time is only slightly greater than that for the single-signer Schnorr scheme.
- The signature length is the same as for the single-signer Schnorr scheme, regardless of the number of signers.

Accountable-subgroup multisignatures(ASM) is formalized and a variant multi-signature scheme has been implemented. In essence, ASM schemes enable any subgroup, S, of a given group, G, of potential signers, to sign efficiently a message M so that the signature provably reveals the identities of the signers in S to any verifier.

15.1.4. *Boneh-Lynn-Shacham Signature Overview.* In cryptography, the Boneh-Lynn-Shacham (BLS) signature scheme allows a user to verify that a signer is authentic. The scheme uses a bilinear pairing for verification, and signatures are elements of an elliptic curve group. Working in an elliptic curve group provides some defense against index calculus attacks (with the caveat that such attacks are still possible in the target group G_T of the pairing), allowing shorter signatures than FDH signatures for a similar level of security. Signatures produced by the BLS signature scheme are often referred to as short signatures, BLS short signatures, or simply BLS signatures. The signature scheme is provably secure (the scheme is existentially unforgeable under adaptive chosen-message attacks) assuming both the existence of random oracles and the intractability of the computational Diffie-Hellman problem in a gap DiffieHellman group. A gap group is a group in which the computational DiffieHellman problem is intractable but the decisional DiffieHellman problem can be efficiently solved. Non-degenerate, efficiently computable, bilinear pairings permit such groups.

Let $e\colon G \times G \to G_T$ be a non-degenerate, efficiently computable, bilinear pairing where G, GT are groups of prime order, r. Let g be a generator of G. Consider an instance of the CDH problem, g, g^x, g^y. Intuitively, the pairing function e does not help us compute g^{xy}, the solution to the CDH problem. It is conjectured that this instance of the CDH problem is intractable. Given g^z, we may check to see if $g^z = g^{xy}$ without knowledge of x, y, and z, by testing whether $e(g^x, g^y) = e(g, g^z)$ holds.

By using the bilinear property $x + y + z$ times, we see that if $e(g^x, g^y) = e(g, g)^{xy} = e(g, g)^z = e(g, g^z)$, then since G_T is a prime order group, $xy = z$.

15.1.5. *Boneh-Lynn-Shacham Signature Scheme.* A signature scheme consists of three functions: generate, sign, and verify.

- Key generation The key generation algorithm selects a random integer x in the interval $[0, r1]$. The private key is x. The holder of the private key publishes the public key, g^x.
- Signing Given the private key x, and some message m, we compute the signature by hashing the bitstring m, as $h = H(m)$. We output the signature $\sigma = h^x$.
- Verification Given a signature σ and a public key g^x, we verify that $e(\sigma, g) = e(H(m), g^x)$.

15.1.6. *Lamport Signature Overview.* In cryptography, a Lamport signature or Lamport one-time signature scheme is a method for constructing a digital signature. Lamport signatures can be built from any cryptographically secure one-way function; usually a cryptographic hash function is used.

Although the potential development of quantum computers threatens the security of many common forms of cryptography such as RSA, it is believed that Lamport signatures with large hash functions would still be secure in that event. Unfortunately, each Lamport key can only be used to sign a single message. However, combined with hash trees, a single key could be used for many messages, making this a fairly efficient digital signature scheme.

The Lamport signature cryptosystem was invented in 1979 and named after its inventor, Leslie Lamport.

Alice has a 256-bit cryptographic hash function and some kind of secure random number generator. She wants to create and use a Lamport key pair, that is, a private key and a corresponding public key. Making the key pair

To create the private key Alice uses the random number generator to produce 256 pairs of random numbers (2×256 numbers in total), each number being 256 bits in size, that is, a total of 2×256×256 bits = 16 KiB in total. This is her private key and she will store it away in a secure place for later use.

To create the public key she hashes each of the 512 random numbers in the private key, thus creating 512 hashes, each 256 bits in size. (Also 16 KiB in total.) These 512 numbers form her public key, which she will share with the world. Signing the message

Later Alice wants to sign a message. First she hashes the message to a 256-bit hash sum. Then, for each bit in the hash, based on the value of the bit, she picks one number from the corresponding pairs of numbers that comprise her private key (i.e., if the bit is 0, the first number is chosen, and if the bit is 1, the second is chosen). This produces a sequence of 256 random numbers. As each number is itself 256 bits long the total size of her signature will be 256×256 bits = 8 KiB. These random numbers are her signature and she publishes them along with the message.

Note that now that Alice's private key is used, it should never be used again. The other 256 random numbers that she did not use for the signature she must destroy. Otherwise, each additional signature reusing the private key halves the security level against adversaries that might later create false signatures from them. Verifying the signature

Then Bob wants to verify Alice's signature of the message. He also hashes the message to get a 256-bit hash sum. Then he uses the bits in the hash sum to pick out 256 of the hashes in Alice's public key. He picks the hashes in the same manner that Alice picked the random numbers for the signature. That is, if the first bit of the message hash is a 0, he picks the first hash in the first pair, and so on.

Then Bob hashes each of the 256 random numbers in Alice's signature. This gives him 256 hashes. If these 256 hashes exactly match the 256 hashes he just picked from Alice's public key then the signature is ok. If not, then the signature is wrong.

Note that prior to Alice publishing the signature of the message, no one else knows the 2×256 random numbers in the private key. Thus, no one else can create the proper list of 256 random numbers for the signature. And after Alice has published the signature, others still do not know the other 256 random numbers and thus can not create signatures that fit other message hashes.

15.1.7. *Lamport Signature Scheme.* Below is a short description of how Lamport signatures work, written in mathematical notation. Note that the "message" in this description is a fixed sized block of reasonable size, possibly (but not necessarily) the hash result of an arbitrary long message being signed.

- Keys Let k be a positive integer and let $P = \{0,1\}^k$ be the set of messages. Let $f : Y \to Z$ be a one-way function. For $1 \leq i \leq k$ and $j \in \{0,1\}$ the signer chooses $y_{i,j} \in Y$ randomly and computes $z_{i,j} = f(y_{i,j})$. The private key, K, consists of $2k$ values $y_{i,j}$. The public key consists of the $2k$ values $z_{i,j}$.
- Signing a message Let $m = m_1 \ldots m_k \in \{0,1\}^k$ be a message. The signature of the message is $\mathrm{sig}(m_1 \ldots m_k) = (y_{1,m_1}, \ldots, y_{k,m_k}) = (s_1, \ldots, s_k)$.
- Verifying a signature The verifier validates a signature by checking that $f(s_i) = z_{i,m_i}$ for all $1 \leq i \leq k$. In order to forge a message Eve would have to invert the one-way function f. This is assumed to be intractable for suitably sized inputs and outputs.

15.1.8. *Lamport Signature Security Parameters.* The security of Lamport signatures is based on security of the one way hash function, the length of its output and the quality of the input.

For a hash function that generates an n-bit message digest, the ideal preimage and 2nd preimage resistance on a single hash function invocation implies on the order of 2n operations and 2n bits of memory effort to find a collision under a classical computing model. According to Grover's algorithm, finding a preimage collision on a single invocation of an ideal hash function is upper bound on $O(2n/2)$ operations under a quantum computing model. In Lamport signatures, each bit of the public key and signature is based on short messages requiring only a single invocation to a hash function.

For each private key yi,j and its corresponding zi,j public key pair, the private key length must be selected so performing a preimage attack on the length of the input is not faster than performing a preimage attack on the length of the output. For example, in a degenerate case, if each private key yi,j element was only 16 bits in length, it is trivial to exhaustively search all 216 possible private key combinations in 215 operations to find a match with the output, irrespective of the message digest length. Therefore a balanced system design ensures both lengths are approximately equal.

Based on Grover's algorithm, a quantum secure system, the length of the public key elements zi,j, the private key elements yi,j and the signature elements si,j must be no less than 2 times larger than the security rating of the system. That is:

- An 80-bit secure system uses element lengths of no less than 160 bit;
- A 128-bit secure system uses element lengths of no less than 256 bit;

However caution should be taken as the idealistic work estimates above assume an ideal (perfect) hash function and are limited to attacks that target only a single preimage at a time. It is known under a conventional computing model that if $23n/5$ preimages are searched, the full cost per preimage decreases from $2n/2$ to $22n/5$. Selecting the optimum element size taking into account the collection of multiple message digests is an open problem. Selection of larger element sizes and stronger hash functions, such as 512-bit elements and SHA-512, ensures greater security margins to manage these unknowns.

15.2. **Environment.** Basic risk-based threat modeling should be a standard best practice for DevSecOps. We will leverage the DevSecOps Toolchain to automate and secure the virtualized compartmented environment for every node of trustmachine.

15.2.1. *Type Safe And Formally Verifiable.*

15.3. **Virtual Machine.** For a long time, mathematical proofs were read, understood and then checked. In some cases you can do calculations like "$a + b - b = a$" by just looking at the form, saying something like "this cancels that". Still, in the usual practice of mathematics, you ought to be able to explain what is going on. You need to understand.

In the first half of 20th century, rigid languages appeared where proofs can be checked by machines. You can see examples here. Maybe the computers do not understand the meaning, but they can check the proofs.

When you translate mathematical texts into such machine-readable proofs, you are "formalizing" mathematics. Recent decades saw a tantalizing progress in this area. The Flyspeck project and CoqFiniteGroups formalized big results from the past centuries. The Homotopy Type Theory was formalized from its very early stages.

So far this was about mathematics. The take away is, you can obtain infinitely many truths at one shot. The equality "$a+b-b = a$" is true for any natural numbers. By the way, here we have a smart contract, whose input is actually only finitely many. Can we do something about this?

A) Verification of TVM bytecodes One way is to look at the TVM bytecodes. They are executed on a simple virtual machine. The rules of the virtual machine is well understood by different clients which usually match (otherwise they fix the difference with uttermost priority). Obstacles:

- Coq proofs currently might be lengthy. Isabelle/HOL seems to made the proofs much shorter. Isabelle/HOL may provide much easier user experience (because it has a well-polished machine word library).

- At the bytecode level, it's harder to see what the code is doing and what to expect. One solution is to make the Solidity compiler annotate the bytecode with the expected properties at specific code location.
- At the bytecode level, Solidity array access looks like storage access with Kaccek hashes, and collisions should be assumed none. Assume $a, b.keccak(a) = keccak(b) -> a = b$, using the pigeon hole argument, one can prove $0 = 1$ and everything.

The steps of verify tvm bytecodes:

(1) try Isabelle/HOL to see if proofs are really 5 times easier there (less than a week) By the way, about the trustworthiness, I would put Isabelle/HOL at least as good as Coq because Isabelle/HOL is based on a simpler deduction system
(2) implement With Isabelle/HOL #Coq.
(3) #verify Deed and some other simple bytecode programs against simple properties (6-10 days) This is finished and there is a report.
(4) #translate the definitions into Lem (2 weeks; almost done except the Keccak hash) done.
(5) #develop a method how to verify assertions between opcodes (this is tvmverif#5) (a week; there is a milestone for this item)
(6) #cover all opcodes (3 days): this is already in progress. A milestone. mostly done, but LOGx does nothing right now
(7) #test the new TVM against the standard VM tests (4 weeks; started, now parsing the tests) mostly done. but VM tests involving multiple contracts are still skipped
(8) NOW here: build a simple Hoare logic
(9) develop ABI
(10) implement the packaging ABI
(11) extend the model so that it can run state-tests
(12) implement verification-condition generation – the desired post-condition and invariant annotations at JUMPDESTs would generate the formulas to prove.
(13) implement a wallet (might be between 8. and 9.)
(14) build decompilation of TVM bytecode into a recursive function in a theorem prover
(15) try to automate the process of verification / finding vulnerabilities

B) Verification of Solidity programs: Another way would be to verify Solidity sources somehow, not looking at the TVM bytecode. Obstacles

- The only way to know the meaning of a Solidity program is to compile it into bytecodes.
- The Solidity compiler is changing fast and the verification tools would need to follow the changes
- The verification tools tend to believe a different behavior from that of actually deployed bytecode

The steps that have to be taken:

- Know Solidity program's meaning without checking the bytecode. For this, one way is to write a Solidity interpreter in Coq or in an ML language. Another way is to compile a Solidity program into WhyML or F*.
- Check the above against the reality. We need a big set of tests to compare the above translation against the bytecode.
- Translate the ML implementation into a proof assistant (or we rely on Why3, in that case we need to trust the SMT solvers)
- Try to assert properties in a shallow-embedded way and write proofs
- Automate the process, maybe by deep-embedding some kind of Hoare logic

C)Formal Verificaton of Proof-of-Stake Protocols

15.4. Smart Contract.

It is possible to develop a formally verifiable smart contract suitable for agile development as well as strongly typed and prone to attacks. However this alone does not verify smart contracts, static analyzer as well as some other infrastructures should be built. One of the implementation available is the bamboo compiler, it is now producing snippets of bytecode for the empty contract (still lots to be done).

The language has:

- no loops or recursions this makes life much easier for static analyzers gas consumption is stabler users would need to call it again and again
- persistent program counter this discourages multiple workflows in a single contract reentrancy does not jump to far away in the source code, but reentrancy arrives around the last sentence executed
- protection against reentrancy
- explcit storage setting at every exit

Dr-Y's contract analyzer The contract analyzer needs an overhaul in the functionality and the UX. The meaning of each basic block and the execution can jump around the basic blocks is yet to be explored.

15.5. Contract ABI.

The Application Binary Interface (ABI) is strongly typed, known at compilation time and static, no introspection mechanism will be provided. All contracts will have the interface definitions of any contracts they call available at compile-time. Function Selector The first four bytes of the call data for a function call specifies the function to be called. It is the first (left, high-order in big-endian) four bytes of the Keccak (SHA-3) hash of the signature of the function. The signature is defined as the canonical expression of the basic prototype, i.e. the function name with the parenthesised list of parameter types. Parameter types are split by a single comma - no spaces are used. Argument Encoding Starting from the fifth byte, the encoded arguments follow. This encoding is also used in other places, e.g. the return values and also event arguments are encoded in the same way, without the four bytes specifying the function. Types

The following elementary types exist:

- uint<M>: unsigned integer type of M bits, $0 < M \leq 256, M\%8 == 0.e.g.uint32, uint8, uint256.$
- int<M>: two's complement signed integer type of M bits, $0 < M \leq 256, M\%8 == 0.$
- address: equivalent to uint160, except for the assumed interpretation and language typing.
- uint, int: synonyms for uint256, int256 respectively (not to be used for computing the function selector).
- bool: equivalent to uint8 restricted to the values 0 and 1
- fixed<M>x<N>: signed fixed-point decimal number of M bits, $0 < M \leq 256, M\%8 == 0$, and $0 < N \leq 80$, which denotes the value v as $v/(10 * *N)$.
- ufixed<M>x<N>: unsigned variant of fixed<M>x<N>.
- fixed, ufixed: synonyms for fixed 128×19, ufixed128x19 respectively (not to be used for computing the function selector).

- bytes<M>: binary type of M bytes, $0 < M \leq 32$.
- function: equivalent to bytes24: an address, followed by a function selector

The following (fixed-size) array type exists:

- <type>[M]: a fixed-length array of the given fixed-length type.

The following non-fixed-size types exist:

- bytes: dynamic sized byte sequence.
- string: dynamic sized unicode string assumed to be UTF-8 encoded.
- <type>[]: a variable-length array of the given fixed-length type.

Formal Specification of the Encoding We will now formally specify the encoding, such that it will have the following properties, which are especially useful if some arguments are nested arrays:

Properties:

(1) The number of reads necessary to access a value is at most the depth of the value inside the argument array structure, i.e. four reads are needed to retrieve a_i[k][l][r]. In a previous version of the ABI, the number of reads scaled linearly with the total number of dynamic parameters in the worst case.
(2) The data of a variable or array element is not interleaved with other data and it is relocatable, i.e. it only uses relative "addresses"

We distinguish static and dynamic types. Static types are encoded in-place and dynamic types are encoded at a separately allocated location after the current block.

Definition: The following types are called "dynamic":

- bytes
- string
- T[] for any T
- T[k] for any dynamic T and any k ¿ 0

All other types are called "static".

Definition: len(a) is the number of bytes in a binary string a. The type of len(a) is assumed to be uint256. We define enc, the actual encoding, as a mapping of values of the ABI types to binary strings such that len(enc(X)) depends on the value of X if and only if the type of X is dynamic.

Definition: For any ABI value X, we recursively define enc(X), depending on the type of X being:

- T[k] for any T and k:

  ```
  enc(X) =
  head(X[0]) ... head(X[k-1])
  tail(X[0]) ... tail(X[k-1])
  ```

  where head and tail are defined for `X[i]` being of a static type as `head(X[i]) = enc(X[i])` and `tail(X[i]) = ""` (the empty string) and as:

  ```
  head(X[i]) =
  enc(
      len(
      head(X[0]) ... head(X[k-1])
      tail(X[0]) ... tail(X[i-1])
      )
  ) tail(X[i]) = enc(X[i])
  ```

  Note that in the dynamic case, head(`X[i]`) is well-defined since the lengths of the head parts only depend on the types and not the values. Its value is the offset of the beginning of tail(`X[i]`) relative to the start of `enc(X)`.
- `T[]` where X has k elements (k is assumed to be of type uint256):

  `enc(X) = enc(k) enc([X[1], ..., X[k]])`

  i.e. it is encoded as if it were an array of static size k, prefixed with the number of elements.
- bytes, of length k (which is assumed to be of type uint256): `enc(X) = enc(k) pad_right(X)`, i.e. the number of bytes is encoded as a uint256 followed by the actual value of X as a byte sequence, followed by the minimum number of zero-bytes such that `len(enc(X))` is a multiple of 32.
- string: `enc(X) = enc(enc_utf8(X))`, i.e. X is utf-8 encoded and this value is interpreted as of bytes type and encoded further. Note that the length used in this subsequent encoding is the number of bytes of the utf-8 encoded string, not its number of characters. uint<M>: `enc(X)` is the big-endian encoding of X, padded on the higher-order (left) side with zero-bytes such that the length is a multiple of 32 bytes.
- address: as in the uint160 case
- int<M>: `enc(X)` is the big-endian two's complement encoding of X, padded on the higher-oder (left) side with 0xff for negative X and with zero bytes for positive X such that the length is a multiple of 32 bytes.
- bool: as in the uint8 case, where 1 is used for true and 0 for false
- fixed<M>x<N>: `enc(X)` is `enc(X * 2**N)` where `X * 2**N` is interpreted as a int256.
- fixed: as in the fixed128x128 case
- ufixed<M>x<N>: `enc(X)` is `enc(X * 2**N)` where `X * 2**N` is interpreted as a uint256.
- ufixed: as in the ufixed128x128 case
- bytes<M>: enc(X) is the sequence of bytes in X padded with zero-bytes to a length of 32.

Note that for any X, `len(enc(X))` is a multiple of 32.

Function Selector and Argument Encoding All in all, a call to the function f with parameters `a_1, ..., a_n` is encoded as:

`function_selector(f) enc([a_1, ..., a_n])`

and the return values `v_1, ..., v_k` of f are encoded as:

`enc([v_1, ..., v_k])`

where the types of `[a_1, ..., a_n]` and `[v_1, ..., v_k]` are assumed to be fixed-size arrays of length n and k, respectively. Note that strictly, `[a_1, ..., a_n]` can be an "array" with elements of different types, but the encoding is still well-defined as the assumed common type T (above) is not actually used. Events are an abstraction of the Ethereum logging/event-watching protocol. Log entries provide the contract's address, a series of up to four topics and some arbitrary length binary data. Events leverage the existing function ABI in order to interpret this (together with an interface spec) as a properly typed structure. Given an event name and series of event parameters, we split them into two sub-series: those which are indexed and those which are not. Those which are indexed, which may number up to 3, are used alongside the Keccak hash of the event signature to form the topics of the log entry. Those which as not indexed form the byte array of the event. In effect, a log entry using this ABI is described as:

- `address`: the address of the contract;
- `topics[0]`:`keccak(EVENT_NAME+"("+EVENT_ARGS.map(canonical_type_of).join(",")+")")` `canonical_type_of` is a function that simply returns the canonical type of a given argument, e.g. for uint

indexed foo, it would return uint256. If the event is declared as anonymous the topics[0] is not generated;
- topics[n]: EVENT_INDEXED_ARGS[n - 1] EVENT_INDEXED_ARGS is the series of EVENT_ARGS that are indexed;
- data: abi_serialise(EVENT_NON_INDEXED_ARGS) EVENT_NON_INDEXED_ARGS is the series of EVENT_ARGS that are not indexed, abi_serialise is the ABI serialisation function used for returning a series of typed values from a function, as described above.

The JSON format for a contract's interface is given by an array of function and/or event descriptions. A function description is a JSON object with the fields:

- type: "function", "constructor", or "fallback" (the unnamed "default" function);
- name: the name of the function;
- inputs: an array of objects, each of which contains: name: the name of the parameter; type: the canonical type of the parameter.
- outputs: an array of objects similar to inputs, can be omitted if function doesn't return anything;
- constant: true if function is specified to not modify blockchain state;
- payable: true if function accepts ether, defaults to false.

Constructor and fallback function never have name or outputs. Fallback function doesn't have inputs either.

Sending non-zero ether to non-payable function will throw. Don't do it.

An event description is a JSON object with fairly similar fields:

- type: always "event"
- typename: the name of the event;
- typeinputs: an array of objects, each of which contains: name: the name of the parameter; type: the canonical type of the parameter. indexed: true if the field is part of the log's topics, false if it one of the log's data segment.
- typeanonymous: true if the event was declared as anonymous.

For example,

```
contract Test {
    function Test(){
        b = 0x12345678901234567890123456789012;
    }
    event Event(uint indexed a, bytes32 b);
    event Event2(uint indexed a, bytes32 b);
    function foo(uint a){
        Event(a, b);
    }
    bytes32 b;
}
```

would result in the JSON:

```
[{"type":"event",
  "inputs":
  [{"name":"a","type":"uint256","indexed":true},
   {"name":"b","type":"bytes32","indexed":false}],
  "name":"Event"
},
{ "type":"event",
  "inputs":
  [{"name":"a","type":"uint256","indexed":true},
   {"name":"b","type":"bytes32","indexed":false}],
  "name":"Event2"
},
{ "type":"function",
  "inputs":
  [{"name":"a","type":"uint256"}],
"name":"foo",
"outputs": []
}]
```

15.6. Networking.

15.6.1. *Whisper Proposal.* Whisper is a communication protocol for DApps to communicate with each other.

- DApps that need to publish small amounts of information to each other and have the publication last some substantial amount of time. For example, a currency exchange DApp may use it to record an offer to sell some currency at a particular rate on an exchange. In this case, it may last anything between tens of minutes and days. The offer wouldn't be binding, merely a hint to get a potential deal started.
- DApps that need to signal to each other in order to ultimately collaborate on a transaction. For example, a currency exchange DApp may use it to coordinate an offer prior to creating one (or two, depending on how the exchange is structured) transactions on the exchange.
- DApps that need to provide non-real-time hinting or general communications between each other. E.g. a small chat-room app.
- DApps that need to provide dark (plausible denial over perfect network traffic analysis) comms to two correspondents that know nothing of each other but a hash. This could be a DApp for a whistleblower to communicate to a known journalist exchange some small amount of verifiable material and arrange between themselves for some other protocol (Swarm, perhaps) to handle the bulk transfer.

In general, think transactions, but without the eventual archival, any necessity of being bound to what is said or automated execution & state change. Specs:

- Low-level API only exposed to DApps, never to users.
- Low-bandwidth Not designed for large data transfers.
- Uncertain-latency Not designed for RTC.
- Dark No reliable methods for tracing packets or
- Typical usage: Low-latency, 1-1 or 1-N signalling messages. High latency, high TTL 1-* publication messages.

Messages less than 64K bytes, typically around 256 bytes.

Existing solutions:

- UDP: Similar in API-level, native multicasting. No TTL, security or privacy safeguards.
- 0MQ: A distributed messaging system, no inherent privacy safeguards.
- Bitmessage: Similar in the basic approach of P2P network exchanging messages with baseline PKI for dark comms. Higher-level (e-mail replacement, only "several thousand/day", larger mails), fixed TTL and no hinting to optimise for throughput. Unclear incentivisation.
- TeleHash: Secure connection-orientated RTC comms. Similar in approach to BitTorrent (uses modified Kademila tech), but rather than discovering peers for a given hash, it routes to the recipient given its hash. Uses DHT to do deterministic routing therefore insecure against simple statistical packet-analysis attacks against a large-scale

attacker. Connection oriented, so no TTL and not designed for asynchronous data publication.
- Tox: Higher-level (IM & AV chat) replacement.

Basic Design: Uses the "shh" protocol string of DEVp2p.

Rest coming soon, once I've finished prototyping. Gav.

Considerations for Defeating Traffic Analysis:

(from lokiverloren) All existing protocols for location obscured instant messaging have complicated problems to do with routing.

The Bitmessage protocol propagates messages blindly across the network, and the proper recipient knows how to decrypt it and receives it (just like everyone else) but then stores it and lets its' user know it's got a new message. The problem with this of course is that it greatly increases the exposure of the whole network to a body of encrypted material that ideally should not be easily accessed at all.

None of the others listed above particularly have any means to hide the source and destination of messages. I believe it is one of the core objectives of the Whisper protocol to hide location of sender and receiver and in transit, make it difficult if not impossible to establish one, the other or both.

The Tor system has a protocol for enabling connections between two nodes without either knowing where the other is, it is the Rendezvous protocol used for hidden services. Most readers of this will already understand how it works, but what happens is that a hidden service (the equivalent of a server listening on a port in the TCP/IP protocol) selects at random a number (I think, usually 6) 'introducer' nodes. In order to do this it establishes the standard 3-hop chain to each of the introducers, and when a user wants to establish a connection with a hidden service, they propagate a request to connect with hidden service that is associated with a particular public key. I'm not sure about how this request is propagated, obviously it would also have to be done through a circuit or the rendezvous introducer node knows who might be about to establish a circuit. Once one's client knows a valid rendezvous node it then establishes a connection to it and requests to have traffic relayed through its' circuit to the hidden service.

I think there really isn't a better way to implement 'dark' parts of the connection protocol, because there has to be a different identity for a relay node as a client node, the same principle applies in Ethereum. However, using the rendezvous, it again makes possible something for connections between nodes that know each other (though they don't know the account or identity of the initiator) something other than the uniform 3 hop circuit. For generating a circuit to an exit node in Tor, it is of no danger that your client knows the identity of the relays in each hop along the way, but to do that within the network compromises location obfuscation security, tying your ethereum identity with your router identity (I think it may need to be explicitly pointed out that routing function is something that Ethereum nodes will perform). When connecting to a rendezvous, you do not thereby reveal your location to the rendezvous, and the rendezvous does not know the location of the hidden service either, and establishing these connections only requires a public directory to be created for routers. It is not consequential information for an attacker and can be revealed directly by probing for a relevant server at your IP addresses anyway.

The contribution I would like to make to the discussion is this:

(1) It is implied that the whisper protocol has to work through a system of location obfuscation relays like Tor. Not only this, but it must therefore also be using some form of the rendezvous protocol used in Tor for hidden services.

(2) However, as Tor was originally designed first to be a way to stop webservers logging your IP address in association with a session cookie and record, then secondarily added the ability to implement the equivalent of TCP/IP routing (listening ports) within the context of location obfuscation in the form of the rendezvous protocol, it is an artifact, a legacy of the first purpose that leads to the uniform utilisation of 3 hop nodes and connections that remain open rather than a session (connectionless) protocol like used in HTTP. What I suggest is that, depending on the needs of a particular connection, there can be occasion to mix up the obfuscation process so that it is further obscured from traffic analysis in the case of malicious nodes performing data gathering for an attacker.

Instead of a connection like is normal between a browser and web server, for example, which is usually left open because of the latency of reestablishing such a connection, to make further requests, instead there can be a session cookie, and then you can alter the way your client sends data through the 'connection' to a hidden node. The connection, between two routing nodes, could be direct, 1 proxy intermediary, 2, or 3, it could use shared secrets instead and route fragments of datagrams across multiple heteregenous circuits, as well, and the receiver would then wait for sufficient fragments to assemble the original packets. Indeed, it could be possible in the case of (not quite related to whisper, but to the distributed data storage protocol) larger streams, break the stream up into parts and alter the obfuscation method through the process, further confusing the traffic analysis data.

The same considerations apply, fundamentally, the only differences have to do with the size of the data being transmitted, whether it's for a messaging system or distributed filesystem. The other criteria for deciding how to scramble the routing is latency. For some purposes one wants lower latency, and other purposes, greater security is vitally important. When in the process streams are fragmented into parts, it can also increase security to apply an All Or Nothing Transform to the entire package, then if part is intercepted but not the complete message, it is impossible to assemble the data, not even for cryptanalysis purposes.

15.6.2. *Whisper Overview.* Basic operation:

`post` takes a JSON object containing four key parameters:

```
shh.post({ "topics": t, "payload": p,
"ttl": ttl, "workToProve": work });
```

- `topic`, provided as either a list of, or a single, arbitrary data items that are used to encode the abstract topic of this message, later used to filter messages for those that are of interest;
- `payload`, provided similarly to topic but left as an unformatted byte array provides the data to be sent.
- `ttl` is a time for the message to live on the network, specified in seconds. This defaults to 50.
- `work` is the amount of priority you want the packet to have on the network. It is specified in milliseconds of processing time on your machine. This defaults to 50.

Two other parameters optionally specify the addressing: recipient (`to`), sender (`from`). The latter is meaningless unless a recipient has been specified.

Use cases:

- `shh.post({ "topic": t, "payload": p });` No signature, no encryption: Anonymous broadcast; a bit like an anonymous subject-filtered twitter feed.
- `shh.post({ "from": myIdentity, "topic": t, "payload": p });` Open signature, no encryption: Clear-signed broadcast; a bit like a normal twitter feed - anyone interested can see a particular identity is sending particular stuff out to no-one in particular.
- `shh.post({ "to": recipient, "topic": t, "payload": p });` No signature, encryption: Encrypted anonymous message; a bit like an anonymous drop-box - message is private to the owner of the dropbox. They can't tell from whom it is.
- `shh.post({ "from": myIdentity, "to": recipient, "topic": t, "payload": p });` Secret signature, encryption: Encrypted signed message; like a secure e-mail. One identity tells another something - nobody else can read it. The recipient alone knows it came from the sender.
- `shh.post({ "from": myIdentity, "to": recipient, "topic": t, "payload": p, "deniable": d });` Secret signature, encryption with optional plausible deniability. If boolean parameter `d` is false, it is equivalent to the previous call. If `d` is true, recipient cannot prove to any third party that the message originates from sender, though still can verify it for herself. This is achieved by the digital signature being calculated on the symmetric session encryption key instead of the message body.

In addition to the basic use cases, there will also be support for secure multi-casting. For this, you set up a group with `shh.newGroup`:

```
var group = shh.newGroup(
eth.key, [ recipient1, recipient2 ]
);
```

Then can use this as a recipient as you would normally:

```
shh.post({
    "from": eth.key,
    "to": group,
    "topic": t,
    "payload": p
});
```

The newGroup actually does something like:

```
var group = shh.newIdentity();
shh.post([
    "from": myIdentity,
    "to": recipient1,
    "topic":[invSHA3(2^255),recipient1],
    "payload": secretFromPublic(group)
]);
shh.post([
    "from": myIdentity,
    "to": recipient2,
    "topic": [invSHA3(2^255), recipient2],
    "payload": secretFromPublic(group)
]);
return keypair;
```

Here, the `invSHA3(2^255)` topic is a sub-band topic (intercepted by the Whisper protocol layer) which takes the key and adds it to the key database. When a packet is addressed to group, it encrypts with group's public key. group is not generally used for signing. `secretFromPublic` obviously isn't a public API and invSHA3 is only possible because we know each item of the topic is SHA3ed prior to amalgamation in the final topic.

When signing a message (one with a from parameter), the message-hash is the hash of the clear-text (unencrypted) payload.

Topics is constructed from a number of components - this simply compresses (sha3 + crop) each into a final set of 4-byte crypto-secure hashes. When composing filters, it's the same process. Importantly, all such hashes given in the filters must be includes in.

To filter on sender/recipient, they should be encoded within the topic by the sender.

Silent Operation

In normal operation (and assuming a non-degenerate attack condition), there is a trade-off between true anonymity/plausible deniability over ones communications and efficiency of operation. The more one advertises to ones peers attempting to "fish" for useful messages and steer such message towards oneself, the more one reveals to ones peers.

For a securely anonymous dynamic two-way conversation, this trade-off becomes problematic; significant topic-advertising would be necessary for the point-to-point conversation to happen with sensible latency and yet so little about the topic can be advertised to guide messages home without revealing substantial information should there be adversary peers around an endpoint. (If substantial numbers of adversary peers surround both endpoints, a tunnelling system similar to TOR must be used to guarantee security.)

In this situation, dynamic topic generation would be used. This effectively turns the datagram-orientated channel into a connection-oriented channel. The endpoint to begin the conversation sends a point-to-point (i.e. signed and encrypted) conversation-begin message that contains a randomly chosen 256-bit topic seed. The seed is combined (by both endpoints) with a message nonce (beginning at 0 and incrementing over the course of the conversation) to provide a secure chain of single-use topics. It then generates a bloom filter using randomly selected bits from the new topic to match against and gives the filter to its peers; once a randomly selected minimum of messages fitting this filter have been collected (we assume one of these is the message we are interested in), we send our reply, deriving a new topic (incrementing the nonce), then advertise for that with yet another topic (another nonce increment) with another randomly selected group of bits.

15.6.3. *DEVp2p Wire Protocol.* Peer-to-peer communications between nodes running Ethereum/Whisper/&c. clients are designed to be governed by a simple wire-protocol making use of existing technologies and standards such as RLP wherever practical.

This document is intended to specify this protocol comprehensively. Low-Level

DEVp2p nodes communicate by sending messages using RLPx, an encrypted and authenticated transport protocol. Peers are free to advertise and accept connections on any TCP ports they wish, however, a default port on which the connection may be listened and made will be 30303. Though TCP provides a connection-oriented medium, DEVp2p nodes communicate in terms of packets. RLPx provides facilities to send and receive packets. For more information about RLPx, refer to the protocol specification.

DEVp2p nodes find peers through the RLPx discovery protocol DHT. Peer connections can also be initiated by supplying the endpoint of a peer to a client-specific RPC API. Payload Contents

There are a number of different types of payload that may be encoded within the RLP. This "type" is always determined by the first entry of the RLP, interpreted as an integer.

DEVp2p is designed to support arbitrary sub-protocols (aka capabilities) over the basic wire protocol. Each sub-protocol is given as much of the message-ID space as it needs (all such protocols must statically specify how many message IDs they require). On connection and reception of the Hello message, both peers have equivalent information about what subprotocols they share (including versions) and are able to form consensus over the composition of message ID space.

Message IDs are assumed to be compact from ID 0x10 onwards (0x00-0x10 is reserved for DEVp2p messages) and given to each shared (equal-version, equal name) sub-protocol in alphabetic order. Sub-protocols that are not shared are ignored. If multiple versions are shared of the same (equal name) sub-protocol, the numerically highest wins, others are ignored. DEVp2p Hello Message

```
0x00 [
    p2pVersion: P,
    clientId: B,
    [
        [cap1: B_3, capVersion1: P],
        [cap2: B_3, capVersion2: P],
        ...
    ],
    listenPort: P,
    nodeId: B_64
]
```

First packet sent over the connection, and sent once by both sides. No other messages may be sent until a Hello is received.

- `p2pVersion` Specifies the implemented version of the DEVp2p protocol. Now must be 1.
- `clientId` Specifies the client software identity, as a human-readable string (e.g. "Ethereum(++)/1.0.0").
- `cap` Specifies a peer capability name as a length-3 ASCII string. Current supported capabilities are eth, shh.
- `capVersion` Specifies a peer capability version as a positive integer. Current supported versions are 34 for eth, and 1 for shh.
- `listenPort` specifies the port that the client is listening on (on the interface that the present connection traverses). If 0 it indicates the client is not listening.
- `nodeId` is the Unique Identity of the node and specifies a 512-bit hash that identifies this node.

Disconnect 0x01 [reason: P] Inform the peer that a disconnection is imminent; if received, a peer should disconnect immediately. When sending, well-behaved hosts give their peers a fighting chance (read: wait 2 seconds) to disconnect to before disconnecting themselves.

reason is an optional integer specifying one of a number of reasons for disconnect: `0x00` Disconnect requested; `0x01` TCP subsystem error; `0x02` Breach of protocol, e.g. a malformed message, bad RLP, incorrect magic number &c.; `0x03` Useless peer; `0x04` Too many peers; `0x05` Already connected; `0x06` Incompatible DEVp2p protocol version; `0x07` Null node identity received - this is automatically invalid; `0x08` Client quitting; `0x09` Unexpected identity (i.e. a different identity to a previous connection/what a trusted peer told us). `0x0a` Identity is the same as this node (i.e. connected to itself); `0x0b` Timeout on receiving a message (i.e. nothing received since sending last ping); `0x10` Some other reason specific to a subprotocol.

Ping `0x02` [] Requests an immediate reply of Pong from the peer.

Pong `0x03` [] Reply to peer's Ping packet.

NotImplemented (was GetPeers) `0x04`

NotImplemented (was Peers) `0x05` Node identity and reputation

The identity of a DEVp2p node is a secp256k1 public key.

Nodes are free to store ratings for given IDs (how useful the node has been in the past) and give preference accordingly. Nodes may also track node IDs (and their provenance) in order to help determine potential man-in-the-middle attacks. Clients are free to mark down new nodes and use the node ID as a means of determining a node's reputation. Session Management

Upon connecting, all clients (i.e. both sides of the connection) must send a `Hello` message. Upon receiving the `Hello` message and verifying compatibility of the network and versions, a session is active and any other DEVp2p messages may be sent.

At any time, a Disconnect message may be sent.

15.6.4. *Whisper Wire Protocol.* Peer-to-peer communications between nodes running Whisper clients run using the underlying DEVp2p Wire Protocol. This is a preliminary wire protocol for the Whisper subsystem. It will change. Whisper Sub-protocol

Status [`+0x00: P, protocolVersion: P`] Inform a peer of the whisper status. This message should be send after the initial handshake and prior to any whisper related messages.

`protocolVersion` is one of: `0x02` for PoC-7.

Messages [+0x01: P, [expiry1: P, ttl1: P, [topic1x1: B_4, topic1x2: B_4, ...], data1: B, nonce1: P], [expiry2: P, ...], ...] Specify one or more messages. Nodes should not resend the same message to a peer in the same session, nor send a message back to a peer from which it received. This packet may be empty. The packet must be sent at least once per second, and only after receiving a Messages message from the peer.

TopicFilter [`+0x02: P, bloom: B_64`] Specify the bloom filter of the topics that the sender is interested in. The bloom method is defined below. Session Management

For the Whisper sub-protocol, upon an active session, a Status message must be sent. Following the reception of the peer's Status message, the Whisper session is active. The peer with the greatest Node Id should send a Messages message to begin the message rally. From that point, peers take it in turns to send (possibly empty) Messages packets. Topics and Abridged Topics

Topics are 32-byte hashes, typically generated from App-specified data. The abridged topic is the first 4 bytes of the topic. Abridged topics allow the identification (within the bounds of an acceptably high probability) of a given topic, useful for determining the utility of a topic for a given peer, yet do not give away the full topic information itself, allowing it to be used as a strong encryption key whose secret is not inherently available. Bloomed Topics

The Bloom filter used in the `TopicFilter` message type is a means a identifying a number of topics to a peer without compromising (too much) privacy over precisely what topics are of interest. Precise control over the information content (and thus efficiency of the filter) may be maintained through the addition of bits.

Blooms are formed by the bitwise OR operation on a number of bloomed topics. The bloom function takes the abridged topic (the first four bytes of the SHA3 of the App/user level topic description) and projects them onto a 512-bit slice; in total, three bits are marked for each bloomed topic.

The projection function is defined as a mapping from a a 4-byte slice S to a 512-bit slice D; for ease of explanation, S will dereference to bytes, whereas D will dereference to bits.

```
LET D[*] = 0
FOREACH i IN { 0, 1, 2 } DO
LET n = S[i]
IF S[3] & (2 ** i) THEN n += 512
D[n] = 1
END FOR
```

In formal notation:

```
D[i] := f(0) == i || f(1) == i || f(2) == i
```

where:

```
f(x) := S[x] + S[3][x] * 512
```

(assuming a byte value `S[i]` may be further dereferenced into a bit value)

15.6.5. *Whisper Protocol Spec.* Whisper combines aspects of both DHTs and datagram messaging systems (e.g. UDP). As such it may be likened and compared to both, not dissimilar to the matter/energy duality (apologies to physicists for the blatant abuse of a fundamental and beautiful natural principle).

Whisper is a pure identity-based messaging system. Whisper provides a low-level (non-application-specific) but easily-accessible API without being based upon or prejudiced by the low-level hardware attributes and characteristics, particularly the notion of singular endpoints.

Alternatively, Whisper may be likened to a DHT with a per-entry configurable TTL and conventions for the signing and encryption of values. In this sense, Whisper provides the ability to have multiply-indexable, non-unique entries (i.e. the same entry having multiple keys, some or all of which may be the same as other entries).

As such, Whisper is not a typical communications system. It is not designed to replace or substitute TCP/IP, UDP, HTTP or any other traditional protocols; it is not designed to provide a connection oriented system, nor for simply delivering data betwixt a pair of specific network endpoints; it does not have a primary goal of maximising bandwidth or minimising latency (though as with any transmission system, these are concerns).

Whisper is a new protocol designed expressly for a new paradigm of application development. It is designed from the ground up for easy and efficient multi-casting and broadcasting. Similarly, low-level partially-asynchronous communications is an important goal. Low-value traffic reduction or retardation is another goal (which might also be likened to the quest for QoS). It is designed to be a building block in next generation DApps which require large-scale many-to-many data-discovery, signal negotiation and modest transmissions with an absolute minimum of fuss and the expectation that one has a very reasonable assurance of complete privacy.

Whisper operates around the notion of being user-configurable with regard to how much information it leaks concerning the DApp content and ultimately, the user activities. To understand information leakage, it is important to distinguish between mere encryption, and darkness. Many protocols, both those designed around p2p and more traditional client/server models provide a level of encryption. For some, encryption forms an intrinsic part of the protocol and, applied alone, delivers its primary requirement. While decentralising and encrypting is a great start on building a legitimately "post-Snowden" Web, it is not the end.

Even with encrypted communications, well-funded attackers are still able to compromise ones privacy, often quite easily. Bulk metadata collection becomes the new battleground, and is at once dismissed as a privacy concern by authorities yet trumpeted as the Next Big Thing by big-data outfits. In the case of a simple client/server model, metadata betrays with which hosts one communicates - this is often plenty enough to compromise privacy given that content is, in many cases, largely determinable from the host.

With decentralised communications systems, e.g. a basic non-routed but encrypted VoIP call or Telehash communication, a network packet-sniffing attacker may not be able to determine the specific content of a transmission, but with the help of ISP IP address logs they would be able to determine to whom one communicated, when and how often. For certain types of applications in various jurisdictions, this is enough to be a concerning lack of privacy.

Even with encryption and packet forwarding through a third relay node, there is still ample room for a determined bulk transmissions-collector to execute statistical attacks on timing and bandwidth, effectively using their knowledge of certain network invariants and the fact that only a finite amount of actors are involved (i.e. that nodes tend to forward messages between the same two peers with minimal latency and that there aren't all that many pairs of nodes that are trying to communicate via the same relay). There are ways to mitigate this attack vector, such as using multiple third-party relays and switching between them randomly or to use very strict framing, however both are imperfect and can lead to substantial inefficiencies.

A truly dark system is one that is utterly uncompromising in information leakage from metadata. At its most secure mode of operation, Whisper can (at a considerable cost of bandwidth and latency) deliver 100% dark operation. Even better, this applies not only for metadata collection from inter-peer conduits (i.e. backbone dragnet devices), but even against a much more arduous "100% - 2" attack; i.e. where every node in the network were compromised (though functional) save a pair running DApps for people that wanted to communicate without anybody else knowing.

Routing and Lack Thereof

Fundamentally, at least with the present state of computer science, all systems present a trade off between the efficiency of deterministic (and thus supposedly optimal) routing and darkness (or, put another way, routing privacy). One of Whisper's differences is in providing a user-configurable trade-off between ones routing privacy and ones routing efficiency.

At its most dark, Whisper nodes are entirely reactive - they receive and record pieces of data and forward them trying to maximise the utility of information transmission to the peers. These pieces of information include what is known as a topic, which may be viewed both as a secure-probabilistic routing endpoint and/or a DHT multi-key.

However, Whisper is also designed to be able to route probabilistically using two methods, both giving away minimal routing information and both being exceptionally resilient to statistical attacks from large-scale metadata collection.

The first builds on the functionality of the DEVp2p backend. This backend provides the ability of Whisper to rate peers and, over time, probabilistically alter (or steer) its set of peers to those which tend to deliver useful (on-topic, timely, required for ones DApps to function) information. Ultimately, as the network evolves and the peer-set is steered, the number of hops between this peer and any others that tend to be good conduits of useful information (be they the emitters or simply the well-positioned hubs) will tend to 0.

Peer steering also provides the incentive for nodes to provide useful information to peers. The fear of being identified as an under-performing peer and thus being rotated out in favour of other nodes gives all nodes incentive to cooperate and share the most useful information. Useful in this case means provably difficult to author (use of a proof-of-work function helps here); a low time-to-live; and well-corresponding to any provided information on what might be useful (read on for more).

The second is more dynamic. Nodes are informed by their DApps over what sort of topics are useful. Nodes are then allowed to advertise to each peer describing these topics. Topics may be described, using masks or Bloom filters, and thus in an incomplete manner, to reduce the amount of information leaked and make passive statistical attacks arbitrarily difficult to execute. The determination of the amount of information to give to peers is both made through an explicit setting from the user and through topic/traffic modelling: "Given the information coming from this peer and my models of information distribution made from total traffic, am I receiving the amount of useful valuable information that I would expect to receive? If so, narrowing it down with additional description information to the peer may be warranted."

These settings can even be configured per-peer (more trusted/longer-lasting peers may be afforded greater amounts of information), and per-DApp (those DApps that may be more sensitive could be afforded less advertising than others). We can also make use of proof-of-work algorithms to minimise the changes of both DoS attacks and 'everything-but' attacks (where a peer is flooded with almost-useful information prompting it to give away more about its true requirements than it would do otherwise).

Through combining and reducing the Blooms/masks, weaker Nth-level information can be provided to peers about their peers' interests, forming a probabilistic topic-reception vortex around nodes, the "topic-space" gravity-well getting weaker and less certain the farther away with the network hop distance from any interested peers. Basic Protocol Elements Envelopes

There are two key concepts in Whisper: Envelopes and Messages. Envelopes may be considered items should Whisper be considered a DHT. Should Whisper be considered a datagram messaging system then envelopes become the packet format which house the potentially encrypted datagrams. Envelopes are necessarily comprehensible by any node (i.e. they themselves are unencrypted).

They contain information to the entry/datagram such as the original time to live (TTL), the absolute time for expiry and, importantly, the topics. Topics are a set of indexes to this item, recorded in order to help an interested party ("recipient") find it or have it routed to them. They might be likened to binary tags or keywords. Envelopes additionally contain a nonce allowing the original insertion node ("sender") to prove that some arbitrary amount of work was done in its composition. Finally, envelopes contain a message-data field; this stores the actual payload, together with some information and potentially a signature; together this forms a message. This field may be encrypted.

Envelopes are transmitted as RLP-encoded structures. The precise definition is given by:

```
[expiry: P,
 ttl: P,
    [topic0: B_4, topic1: B_4, ...],
 data: B,
 nonce: P
]
```

Here, ttl is given in seconds, expiry is the Unix time of the intended expiry date/time for the envelope. Following this point in time the envelope should no longer be transmitted (or stored, unless there is some extenuating circumstance). Prior to this point in time less the ttl (i.e. the implied insertion time), the envelope is considered utterly invalid and should be dropped immediately and the transmitting peer punished.

nonce is an arbitrary value. We say the work proved through the value of the SHA3 of the concatenation of the nonce and the SHA3 of the RLP of the packet save the nonce (i.e. just a 4-item RLP list), each of the components as a fixed-length 256-bit hash. When this final hash is interpreted as a BE-encoded value, the smaller it is, the higher the work proved. This is used later to judge a peer. Topics

Topics are cryptographically secure, probabilistic partial-classifications of the message. Each topic in the set (order is unimportant) is determined as the first (left) 4 bytes of the SHA3-256 hash of some arbitrary data given by the original author of the message. These might e.g. correspond to "twitter" hash-tags or an intended recipient's public key hashed with some session nonce or application-identity.

Four bytes was chosen to minimise space should a large number of topics be mentioned while still keeping a sufficiently large space to avoid large-scale topic-collision (though it may yet be reviewed and possibly made dynamic in later revisions of the protocol). Messages

A message is formed as the concatenation of a single byte for flags (at present only a single flag is used), followed by any additional data (as stipulated by the flags) and finally the actual payload. This series of bytes is what forms the `data` item of the envelope and is always encrypted.

In the present protocol version, no explicit authentication token is given to indicate that the data field is encrypted; any would-be readers of the message must know ahead of time, through the choice of topic that they have specifically filtered for, that the message is encrypted with a particular key. This is likely to be altered in a further PoC to include a MAC.

Any determination that the message is indeed from a particular sender is left for a higher-level to address. This is noted through the Javascript API allowing the to parameter to be passed only at the point of specifying the filter. Since the signature is a part of the message and not outside in the envelope, those unable to decrypt the message data are also unable to access any signature.

- `flags`: 1 byte
- (`signature`: 65 bytes)
- `payload`: not fixed

Bit 0 of the flags determines whether the signature exists. All other bits are not yet given a purpose and should be set randomly. A message is invalid if bit 0 is set but the total data is less than 66 bytes (since this wouldn't allow it to contain a signature).

Payloads are encrypted in one of two ways. If the message has a specific recipient, then by using ECIES with the specific recipient's SECP-256k1 public key. If the message has no recipient, then by AES-256 with a randomly generated key. This key is then XORed with each of the full topics to form a salted topic. Each salted topic is stored prior to the encrypted data in the same order as the corresponding topics are in the envelope header.

As a recipient, payloads are decrypted in one of two ways. Through use of topics, it should be known whether the envelope is encrypted to a specific recipient (in which case use the private key to decrypt) or to a general multicast audience. In the latter case, we assume that at least one topic is known (since otherwise, the envelope could not be properly "identified"). In this case, we match the known full topic to one of the abridged topics in the envelope, determine the index and de-salt the according salted-key at the beginning of the data segment in order to retrieve the final key.

Encryption using the full topic with "routing" using the abridged topic ensures that nodes which are merely transiently storing the message and have no interest in the contents (thus have access only to routing information via the abridged topics) have no intrinsic ability to read the content of the message.

The signature, if provided, is the SHA3-256 hash of the unencrypted payload signed using ECDSA with the insertion-identity's secret key.

The signature portion is formed as the concatenation of the r, s and v parameters of the SECP-256k1 ECDSA signature, in that order. v is non-normalised and should be either 27 or 28. r and s are both big-endian encoded, fixed-width 32-byte unsigned.

The payload is otherwise unformatted binary data.

In the Javascript API, the distinction between envelopes and messages is blurred. This is because DApps should know nothing about envelopes whose message cannot be inspected; the fact that nodes pass envelopes around regardless of their ability to decode the message (or indeed their interest in it at all) is an important component in Whisper's dark communications strategy.
Basic Operation

Nodes are expected to receive and send envelopes continuously, as per the protocol specification. They should maintain a map of envelopes, indexed by expiry time, and prune accordingly. They should also efficiently deliver messages to the front-end API through maintaining mappings between topics and envelopes.

When a node's envelope memory becomes exhausted, a node may drop envelopes it considers unimportant or unlikely to please its peers. Nodes should consider peers good that pass them envelopes with low TTLs and high proofs-of-work. Nodes should consider peers bad that pass then expired envelopes or, worse, those that have an implied insertion time prior to the present.

Nodes should always keep messages that its DApps have created. Though not in PoC-1, later editions of this protocol may allow DApps to mark messages as being "archived" and these should be stored and made available for additional time.

Nodes should retain a set of per-DApp topics it is interested in. Inserting (Authoring) Messages

To insert a message, little more is needed than to place the envelope containing it in the node's envelope set that it maintains; the node should, according to its normal heuristics retransmit the envelope in due course. Composing an envelope from a basic payload, possible identities for authoring and access, a number of topics, a time-to-live and some parameters concerning work-proving targets is done though a few steps:

- Compose data through concatenating the relevant flag byte, a signature of the payload if the user specified a valid author identity, and the user-given payload.
- Encrypt the data if an access ("destination") identity's public key is given by the user.
- Compose topics from the first 4 bytes of the SHA3 of each topic.
- Set user-given attribute ttl.
- Set the expiry as the present Unix time plus the time-to-live.
- Set the nonce as that which provides the most work proved as per the previous definition, after some fixed amount of time of cycling through candidates or after a candidate surpasses some boundary; either should be given by the user.

Topic Masking and Advertising

Nodes can advertise their topics of interest to each other. For that purpose they use a special type of Whisper message (TopicFilterPacket). The size of Bloom Filter they send to each other must be 64 bytes. Subsequently the rating system will be introduced – peers sending useful messages will be rated higher then those sending random messages.

A message matches the bloom filter, if any one of the topics in this message, converted to the Whisper bloom hash, will match the bloom filter.

Whisper bloom function accepts AbridgedTopic as a parameter (size: 4 bytes), and produces a 64-byte hash, where three bits (at the most) are set to one, and the rest are set to zeros, according to the following algorithm:

(1) Set all the bits in the resulting 64-byte hash to zero.
(2) We take 9 bits form the AbridgedTopic, and convert to integer value (range: 0 - 511).
(3) This value defines the index of the bit in the resulting 512-bit hash, which should be set to one.
(4) Repeat steps 1 & 2 for the second and third bit to be set in the resulting hash.

Thus, in order to produce the bloom, we use 27 bits out of 32 in the AbridgedTopic. For more details, please see the implementation of the function: TopicBloomFilterBase::bloom() [libwhisper/BloomFilter.h]. Coming changes

Also being considered for is support for plausible deniability through the use of session keys and a formalisation of the multicast mechanism.

15.6.6. *Named Data Networking.* Named Data Networking (NDN) is a Future Internet architecture inspired by years of empirical research into network usage and a growing awareness of unsolved problems in contemporary internet architectures like IP. NDN has its roots in an earlier project, Content-Centric Networking (CCN), which Van Jacobson first publicly presented in 2006. The NDN project is investigating Jacobsons proposed evolution from todays host-centric network architecture IP to a data-centric network architecture (NDN). The belief is that this conceptually simple shift will have far-reaching implications for how people design, develop, deploy, and use networks and applications. Its premise is that the Internet is primarily used as an information distribution network, which is not a good match for IP, and that the future Internet's "thin waist" should be based on named data rather than numerically addressed hosts. The underlying principle is that a communication network should allow a user to focus on the data he or she needs, named content, rather than having to reference a specific, physical location where that data is to be retrieved from, named hosts. The motivation for this is derived from the fact that the vast majority of current Internet usage (a "high 90% level of traffic") consists of data being disseminated from a source to a number of users. Named-data networking comes with potential for a wide range of benefits such as content caching to reduce congestion and improve delivery speed, simpler configuration of network devices, and building security into the network at the data level.

15.7. **Consensus Layer.**

15.8. **Client Side.**

15.9. **Introducing EΛGLE.**

15.10. Research And Implementation.

16. FORMAL VERIFICATION

16.1. Previous Work.

16.2. Introducing SOTΛ (STATE-OF-THE-ΛRT):.

16.3. Research And Implemention.

17. CONSENSUS

17.1. Proof of work.

17.1.1. *Bitcoin proof of work.*

17.1.2. *Equihash.*

17.1.3. *Ghost Protocol.*

17.1.4. *Proof of Storage.*

17.1.5. *Proof of Bandwidth.*

17.1.6. *Proof of DDOS.*

17.1.7. *Proof of Tor Bandwidth.*

17.1.8. *Scientific Computing.*

17.2. Proof of Stake.

17.2.1. *Proof of Stake.*

17.2.2. *Delegated Proof of Stake.*

17.2.3. *Ouroboros.*

17.2.4. *Casper.*

17.3. Permissioned Consensus.

17.3.1. *Paxos Byzantine Fault Tolerance.*

17.3.2. *Raft Byzantine Fault Tolerance.*

17.4. Asynchronous Consensus.

17.4.1. *Tendermint.*

17.4.2. *Honey Badger.*

17.5. Opinion Dynamics Based Consensus.

17.5.1. *Opinion Dynamics Framework And Model.*

17.5.2. *Sky, Skyhash and Sznajd2.*

17.5.3. *Oracles in Opinion Dynamics.*

17.6. BLS Threshold Relaying.

17.6.1. *BLS Threshold Signature.*

17.6.2. *Threshold Relaying.*

17.6.3. *Random Number Based On BLS Threshold Relaying.*

17.7. Merkle Directed Acyclic Graph.

17.8. Proof of Authority.

17.9. Plug And Play.

17.10. Introducing ΛCCORD: Adapative Consensus.

17.11. Research And Implemention.

18. PRIVACY PROTECION

18.1. zk-SNARKS.

18.2. zk-STARKs.

18.3. notation.

18.4. Concepts.

18.5. Abstraction Protocol.

18.6. Concrete Protocol.

18.7. Consensus Layer.

18.8. Baby Zoe.

18.9. Hawk.

18.10. Sigma Protocol.

18.11. Introducing ZION-GUΛRD.

18.12. Research And Implementation.

19. SUBSTATE TRANSITION SYSTEM

19.1. **Lightning Network.** Bitcoin Lightning Network requires a new sighash type that addresses malleability to build an unsigned yet trusted funding transaction that creates a timelock-decremented window for RSMC(Revocable Sequence-Maturity Contract) and HTLC(Hashed Time-Locked Contract) to route within network of channels incentivised by optimized amount fees with shortest path and minimum time. Despite the fact that LN's network typology might reach some decentralization that still remains trust-worthy, the security of micro state-transitions within LN still ultimately depends upon the underlying state transition system which forms the locked-up time window, and Bitcoin itself might not be that flexible or scalable enough due to its heavy nature, which greatly restricted possible use cases of LN to cover complex economic activities.

19.2. **Lightning Smart Contracts And State Channel.** Enforced by the security of a general-purpose state-transition system, complex smart contracts executed off-chain that satisfies diverse needs might achieve great scalability and throughput routing smartly within the trusted and intelligent state channels.

19.3. Plasma Framework.

19.3.1. *Scalable Multi-Party Computation.*

19.3.2. *The Plasma Blockchain, or Externalized Multiparty Channels.*

19.3.3. *Enforcible Blockchains in Blockchains.*

19.3.4. *Plasma Proof-of-Stake.*

19.3.5. *Blockchains as MapReduce.*

19.3.6. *Economic Incentives.*

19.3.7. *Design Stack and Smart Contracts.*

19.4. Introducing BEΛSTS: Blind Elastic Accelerated Substate Transition System.

19.5. Research And Implementation.

20. INTERCONNECTED STATE COUPLING SYSTEM

20.1. Cosmos.

20.2. Polkadot.

20.3. Introducing THE MΛTRIX: The Universal State Coupling System.

33. FUTURE DIRECTIONS

The state database won't be forced to maintain all past state trie structures into the future. It should maintain an age for each node and eventually discard nodes that are neither recent enough nor checkpoints; checkpoints, or a set of nodes in the database that allow a particular block's state trie to be traversed, could be used to place a maximum limit on the amount of computation needed in order to retrieve any state throughout the blockchain.

Blockchain consolidation could be used in order to reduce the amount of blocks a client would need to download to act as a full, mining, node. A compressed archive of the trie structure at given points in time (perhaps one in every 10,000th block) could be maintained by the peer network, effectively recasting the genesis block. This would reduce the amount to be downloaded to a single archive plus a hard maximum limit of blocks.

Finally, blockchain compression could perhaps be conducted: nodes in state trie that haven't sent/received a transaction in some constant amount of blocks could be thrown out, reducing both Trust-leakage and the growth of the state database.

33.1. **Scalability.** Scalability remains an eternal concern. With a generalised state transition function, it becomes difficult to partition and parallelise transactions to apply the divide-and-conquer strategy. Unaddressed, the dynamic value-range of the system remains essentially fixed and as the average transaction value increases, the less valuable of them become ignored, being economically pointless to include in the main ledger. However, several strategies exist that may potentially be exploited to provide a considerably more scalable protocol.

Some form of hierarchical structure, achieved by either consolidating smaller lighter-weight chains into the main block or building the main block through the incremental combination and adhesion (through proof-of-work) of smaller transaction sets may allow parallelisation of transaction combination and block-building. Parallelism could also come from a prioritised set of parallel blockchains, consolidated each block and with duplicate or invalid transactions thrown out accordingly.

Finally, verifiable computation, if made generally available and efficient enough, may provide a route to allow the proof-of-work to be the verification of final state.

34. CONCLUSION

I have introduced, discussed and formally defined the protocol of Trustmachine. Through this protocol the reader may implement a node on the Trustmachine network and join others in a decentralised secure social operating system. Contracts may be authored in order to algorithmically specify and autonomously enforce rules of interaction.

35. ACKNOWLEDGEMENTS

Many thanks to Aeron Buchanan for authoring the Homestead revisions, Christoph Jentzsch for authoring the Ethash algorithm and Yoichi Hirai for doing most of the EIP-150 changes. Important maintenance, useful corrections and suggestions were provided by a number of others from the Trustmachine DEV organisation and Trustmachine community at large including Gustav Simonsson, Paweł Bylica, Jutta Steiner, Nick Savers, Viktor Trón, Marko Simovic, Giacomo Tazzari and, of course, Vitalik Buterin.

36. AVAILABLILITY

The PDF of this paper is located at `https://github.com/trust-tech/blackpaper`.

References

Bernardo David Aggelos Kiayias, Alexander Russell and Roman Oliynykov. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. 2017. URL {https://eprint.iacr.org/2016/889.pdf}.

Adam Back. Hashcash - Amortizable Publicly Auditable Cost-Functions. 2002. URL {http://www.hashcash.org/papers/amortizable.pdf}.

Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. 2014. URL {http://zerocash-project.org/media/pdf/zerocash-extended-20140518.pdf}.

Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. 2006. URL {https://cr.yp.to/ecdh/curve25519-20060209.pdf}.

Vitalik Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. 2013a. URL {https://github.com/ethereum/wiki/wiki/White-Paper}.

Vitalik Buterin. Dagger: A Memory-Hard to Compute, Memory-Easy to Verify Scrypt Alternative. 2013b. URL {http://vitalik.ca/ethereum/dagger.html}.

Bytemaster. EOS.IO Technical White Paper. 2017. URL {https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md}.

Houwu Chen, Jiwu Shu, and Brandon. Sky: an Opinion Dynamics Framework and Model for Consensus over P2P Network. 2015. URL {https://github.com/skycoin/whitepapers}.

Whitfield Diffie and Martin E.Hellman. New Directions in Cryptography. 1976. URL {https://www-ee.stanford.edu/~hellman/publications/24.pdf}.

Thaddeus Dryja. Hashimoto: I/O bound proof of work. 2014. URL {https://mirrorx.com/files/hashimoto.pdf}.

Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *In 12th Annual International Cryptology Conference*, pages 139–147, 1992.

Phong Vo Glenn Fowler, Landon Curt Noll. FowlerNollVo hash function. 1991. URL {https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function#cite_note-2}.

L.M Goodman. Tezos: A Self-amending Crypto-ledger. 2014. URL {https://www.tezos.com/static/papers/white_paper.pdf}.

Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In *Cryptographic Hardware and Embedded Systems-CHES 2004*, pages 119–132. Springer, 2004.

Ethan Buchman Jae Kwon. Cosmos: A Network of Distributed Ledgers. 2017. URL {https://github.com/cosmos/cosmos/blob/master/WHITEPAPER.md}.

S. Josefsson. Edwards-Curve Digital Signature Algorithm (EdDSA). 2016. URL {https://tools.ietf.org/pdf/draft-irtf-cfrg-eddsa-05.pdf}.

Thaddeus Dryja Joseph Poon. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. 2016. URL {https://lightning.network/lightning-network-paper.pdf}.

JPMorganChase. Quorum Whitepaper. 2016. URL {https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ}.

Jae Kwon. Tendermint: Consensus Without Mining. 2014. URL {https://tendermint.com/static/docs/

tendermint.pdf}.

Daniel Larimer. Delegated Proof-of-Stake (DPOS). 2014. URL {https://steemit.com/bitshares/@testz/bitshares-history-delegated-proof-of-stake-dpos}.

Sergio Demian Lerner. Strict Memory Hard Hashing Functions. 2014. URL {http://www.hashcash.org/papers/memohash.pdf}.

Silvio Micali, Kazuo Ohta, and Leonid Reyzin. Curve25519: New Diffie-Hellman Speed Records. 2006. URL {https://cr.yp.to/ecdh/curve25519-20060209.pdf}.

Mark Miller. The Future of Law. In *paper delivered at the Extro 3 Conference (August 9)*, 1997.

Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Consulted*, 1:2012, 2008.

Joseph Poon and Vitalik Buterin. Plasma: Scalable Autonomous Smart Contracts. 2017. URL {http://plasma.io/plasma.pdf}.

David Schwartz, Noah Youngs, and Arthur Britt. The Ripple Protocol Consensus Algorithm. 2014. URL {https://ripple.com/files/ripple_consensus_whitepaper.pdf}.

Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

Sangeeth Chandrakumar Vivek Vishnumurthy and Emin Gn Sirer. Karma: A secure economic framework for peer-to-peer resource sharing, 2003.

Dr Gavin Wood. Ethereum: A Secure Decentralized Generalized Transaction Ledger. 2014. URL {http://gavwood.com/paper.pdf}.

Dr Gavin Wood. Polkadot: Vision For A Heterogeneous Multichain Framework. 2017. URL {https://github.com/polkadot-io/polkadotpaper/raw/master/PolkaDotPaper.pdf}.

Pieter Wuille. Segregated Witness. 2016. URL {https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki}.

Jack Pettersson Zackary Hess, anislav Malahov. ternity blockchain: The trustless, decentralized and purely functional oracle machine. 2017. URL {https://blockchain.aeternity.com/%C3%A6ternity-blockchain-whitepaper.pdf}.

Vlad Zamfir and L.G.Meredith. Casper. 2016. URL {https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ}.

Appendix A. Terminology

External Actor: A person or other entity able to interface to a Trustmachine node, but external to the world of Trustmachine. It can interact with Trustmachine through depositing signed Transactions and inspecting the blockchain and associated state. Has one (or more) intrinsic Accounts.

Address: A 160-bit code used for identifying Accounts.

Account: Accounts have an intrinsic balance and transaction count maintained as part of the Trustmachine state. They also have some (possibly empty) TVM Code and a (possibly empty) Storage State associated with them. Though homogenous, it makes sense to distinguish between two practical types of account: those with empty associated TVM Code (thus the account balance is controlled, if at all, by some external entity) and those with non-empty associated TVM Code (thus the account represents an Autonomous Object). Each Account has a single Address that identifies it.

Transaction: A piece of data, signed by an External Actor. It represents either a Message or a new Autonomous Object. Transactions are recorded into each block of the blockchain.

Autonomous Object: A notional object existent only within the hypothetical state of Trustmachine. Has an intrinsic address and thus an associated account; the account will have non-empty associated TVM Code. Incorporated only as the Storage State of that account.

Storage State: The information particular to a given Account that is maintained between the times that the Account's associated TVM Code runs.

Message: Data (as a set of bytes) and Value (specified as Trust) that is passed between two Accounts, either through the deterministic operation of an Autonomous Object or the cryptographically secure signature of the Transaction.

Message Call: The act of passing a message from one Account to another. If the destination account is associated with non-empty TVM Code, then the VM will be started with the state of said Object and the Message acted upon. If the message sender is an Autonomous Object, then the Call passes any data returned from the VM operation.

Mana: The fundamental network cost unit. Paid for exclusively by Trust (as of PoC-4), which is converted freely to and from Mana as required. Mana does not exist outside of the internal Trustmachine computation engine; its price is set by the Transaction and miners are free to ignore Transactions whose Mana price is too low.

Contract: Informal term used to mean both a piece of TVM Code that may be associated with an Account or an Autonomous Object.

Object: Synonym for Autonomous Object.

App: An end-user-visible application hosted in the Trustmachine Browser.

Trustmachine Browser: (aka Trustmachine Reference Client) A cross-platform GUI of an interface similar to a simplified browser (a la Chrome) that is able to host sandboxed applications whose backend is purely on the Trustmachine protocol.

Trustmachine Virtual Machine: (aka TVM) The virtual machine that forms the key part of the execution model for an Account's associated TVM Code.

Trustmachine Runtime Environment: (aka ERE) The environment which is provided to an Autonomous Object executing in the TVM. Includes the TVM but also the structure of the state mapping on which the TVM relies for certain I/O instructions including CALL & CREATE.

TVM Code: The bytecode that the TVM can natively execute. Used to formally specify the meaning and ramifications of a message to an Account.

TVM Assembly: The human-readable form of TVM-code.

- **LLL:** The Lisp-like Low-level Language, a human-writable language used for authoring simple contracts and general low-level language toolkit for trans-compiling to.

Appendix B. Recursive Length Prefix

This is a serialisation method for encoding arbitrarily structured binary data (byte arrays).

We define the set of possible structures $\mathbb{T}$:

$$\begin{aligned}
\mathbb{T} &\equiv \mathbb{L} \cup \mathbb{B} \\
\mathbb{L} &\equiv \{\mathbf{t} : \mathbf{t} = (\mathbf{t}[0], \mathbf{t}[1], ...) \;\wedge\; \forall_{n<\|\mathbf{t}\|}\; \mathbf{t}[n] \in \mathbb{T}\} \\
\mathbb{B} &\equiv \{\mathbf{b} : \mathbf{b} = (\mathbf{b}[0], \mathbf{b}[1], ...) \;\wedge\; \forall_{n<\|\mathbf{b}\|}\; \mathbf{b}[n] \in \mathbb{O}\}
\end{aligned}$$

Where $\mathbb{O}$ is the set of bytes. Thus $\mathbb{B}$ is the set of all sequences of bytes (otherwise known as byte-arrays, and a leaf if imagined as a tree), $\mathbb{L}$ is the set of all tree-like (sub-)structures that are not a single leaf (a branch node if imagined as a tree) and $\mathbb{T}$ is the set of all byte-arrays and such structural sequences.

We define the RLP function as $\mathtt{RLP}$ through two sub-functions, the first handling the instance when the value is a byte array, the second when it is a sequence of further values:

(172)
$$\mathtt{RLP}(\mathbf{x}) \equiv \begin{cases} R_b(\mathbf{x}) & \text{if} \quad \mathbf{x} \in \mathbb{B} \\ R_l(\mathbf{x}) & \text{otherwise} \end{cases}$$

If the value to be serialised is a byte-array, the RLP serialisation takes one of three forms:

- If the byte-array contains solely a single byte and that single byte is less than 128, then the input is exactly equal to the output.
- If the byte-array contains fewer than 56 bytes, then the output is equal to the input prefixed by the byte equal to the length of the byte array plus 128.
- Otherwise, the output is equal to the input prefixed by the minimal-length byte-array which when interpreted as a big-endian integer is equal to the length of the input byte array, which is itself prefixed by the number of bytes required to faithfully encode this length value plus 183.

Formally, we define R_b:

$$\begin{aligned}
R_b(\mathbf{x}) &\equiv \begin{cases} \mathbf{x} & \text{if} \quad \|\mathbf{x}\| = 1 \wedge \mathbf{x}[0] < 128 \\ (128 + \|\mathbf{x}\|) \cdot \mathbf{x} & \text{else if} \quad \|\mathbf{x}\| < 56 \\ \big(183 + \big\|\mathtt{BE}(\|\mathbf{x}\|)\big\|\big) \cdot \mathtt{BE}(\|\mathbf{x}\|) \cdot \mathbf{x} & \text{otherwise} \end{cases} \\
\mathtt{BE}(x) &\equiv (b_0, b_1, ...) : b_0 \neq 0 \wedge x = \sum_{n=0}^{n<\|\mathbf{b}\|} b_n \cdot 256^{\|\mathbf{b}\|-1-n} \\
(a) \cdot (b, c) \cdot (d, e) &= (a, b, c, d, e)
\end{aligned}$$

Thus $\mathtt{BE}$ is the function that expands a positive integer value to a big-endian byte array of minimal length and the dot operator performs sequence concatenation.

If instead, the value to be serialised is a sequence of other items then the RLP serialisation takes one of two forms:

- If the concatenated serialisations of each contained item is less than 56 bytes in length, then the output is equal to that concatenation prefixed by the byte equal to the length of this byte array plus 192.
- Otherwise, the output is equal to the concatenated serialisations prefixed by the minimal-length byte-array which when interpreted as a big-endian integer is equal to the length of the concatenated serialisations byte array, which is itself prefixed by the number of bytes required to faithfully encode this length value plus 247.

Thus we finish by formally defining R_l:

$$\begin{aligned}
R_l(\mathbf{x}) &\equiv \begin{cases} (192 + \|s(\mathbf{x})\|) \cdot s(\mathbf{x}) & \text{if} \quad \|s(\mathbf{x})\| < 56 \\ \big(247 + \big\|\mathtt{BE}(\|s(\mathbf{x})\|)\big\|\big) \cdot \mathtt{BE}(\|s(\mathbf{x})\|) \cdot s(\mathbf{x}) & \text{otherwise} \end{cases} \\
s(\mathbf{x}) &\equiv \mathtt{RLP}(\mathbf{x}_0) \cdot \mathtt{RLP}(\mathbf{x}_1)...
\end{aligned}$$

If RLP is used to encode a scalar, defined only as a positive integer ($\mathbb{P}$ or any x for $\mathbb{P}_x$), it must be specified as the shortest byte array such that the big-endian interpretation of it is equal. Thus the RLP of some positive integer i is defined as:

(178)
$$\mathtt{RLP}(i : i \in \mathbb{P}) \equiv \mathtt{RLP}(\mathtt{BE}(i))$$

When interpreting RLP data, if an expected fragment is decoded as a scalar and leading zeroes are found in the byte sequence, clients are required to consider it non-canonical and treat it in the same manner as otherwise invalid RLP data, dismissing it completely.

There is no specific canonical encoding format for signed or floating-point values.

Appendix C. Hex-Prefix Encoding

Hex-prefix encoding is an efficient method of encoding an arbitrary number of nibbles as a byte array. It is able to store an additional flag which, when used in the context of the trie (the only context in which it is used), disambiguates between node types.

It is defined as the function HP which maps from a sequence of nibbles (represented by the set $\mathbb{Y}$) together with a boolean value to a sequence of bytes (represented by the set $\mathbb{B}$):

$$\begin{aligned}
\texttt{HP}(\mathbf{x}, t) : \mathbf{x} \in \mathbb{Y} &\equiv \begin{cases} (16f(t), 16\mathbf{x}[0] + \mathbf{x}[1], 16\mathbf{x}[2] + \mathbf{x}[3], ...) & \text{if} \quad \lVert\mathbf{x}\rVert \text{ is even} \\ (16(f(t)+1) + \mathbf{x}[0], 16\mathbf{x}[1] + \mathbf{x}[2], 16\mathbf{x}[3] + \mathbf{x}[4], ...) & \text{otherwise} \end{cases} \\
f(t) &\equiv \begin{cases} 2 & \text{if} \quad t \neq 0 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

Thus the high nibble of the first byte contains two flags; the lowest bit encoding the oddness of the length and the second-lowest encoding the flag t. The low nibble of the first byte is zero in the case of an even number of nibbles and the first nibble in the case of an odd number. All remaining nibbles (now an even number) fit properly into the remaining bytes.

Appendix D. Modified Merkle Patricia Tree

The modified Merkle Patricia tree (trie) provides a persistent data structure to map between arbitrary-length binary data (byte arrays). It is defined in terms of a mutable data structure to map between 256-bit binary fragments and arbitrary-length binary data, typically implemented as a database. The core of the trie, and its sole requirement in terms of the protocol specification is to provide a single value that identifies a given set of key-value pairs, which may be either a 32 byte sequence or the empty byte sequence. It is left as an implementation consideration to store and maintain the structure of the trie in a manner that allows effective and efficient realisation of the protocol.

Formally, we assume the input value $\mathfrak{I}$, a set containing pairs of byte sequences:

(181) $$\mathfrak{I} = \{(\mathbf{k}_0 \in \mathbb{B}, \mathbf{v}_0 \in \mathbb{B}), (\mathbf{k}_1 \in \mathbb{B}, \mathbf{v}_1 \in \mathbb{B}), ...\}$$

When considering such a sequence, we use the common numeric subscript notation to refer to a tuple's key or value, thus:

(182) $$\forall_{I \in \mathfrak{I}} I \equiv (I_0, I_1)$$

Any series of bytes may also trivially be viewed as a series of nibbles, given an endian-specific notation; here we assume big-endian. Thus:

$$\begin{aligned}
y(\mathfrak{I}) &= \{(\mathbf{k}_0' \in \mathbb{Y}, \mathbf{v}_0 \in \mathbb{B}), (\mathbf{k}_1' \in \mathbb{Y}, \mathbf{v}_1 \in \mathbb{B}), ...\} \\
\forall_n \quad \forall_{i : i < 2\lVert\mathbf{k}_n\rVert} \quad \mathbf{k}_n'[i] &\equiv \begin{cases} \lfloor \mathbf{k}_n[i \div 2] \div 16 \rfloor & \text{if } i \text{ is even} \\ \mathbf{k}_n[\lfloor i \div 2 \rfloor] \bmod 16 & \text{otherwise} \end{cases}
\end{aligned}$$

We define the function TRIE, which evaluates to the root of the trie that represents this set when encoded in this structure:

(185) $$\texttt{TRIE}(\mathfrak{I}) \equiv \texttt{KEC}(c(\mathfrak{I}, 0))$$

We also assume a function n, the trie's node cap function. When composing a node, we use RLP to encode the structure. As a means of reducing storage complexity, for nodes whose composed RLP is fewer than 32 bytes, we store the RLP directly; for those larger we assert prescience of the byte array whose Keccak hash evaluates to our reference. Thus we define in terms of c, the node composition function:

(186) $$n(\mathfrak{I}, i) \equiv \begin{cases} () & \text{if} \quad \mathfrak{I} = \varnothing \\ c(\mathfrak{I}, i) & \text{if} \quad \lVert c(\mathfrak{I}, i)\rVert < 32 \\ \texttt{KEC}(c(\mathfrak{I}, i)) & \text{otherwise} \end{cases}$$

In a manner similar to a radix tree, when the trie is traversed from root to leaf, one may build a single key-value pair. The key is accumulated through the traversal, acquiring a single nibble from each branch node (just as with a radix tree). Unlike a radix tree, in the case of multiple keys sharing the same prefix or in the case of a single key having a unique suffix, two optimising nodes are provided. Thus while traversing, one may potentially acquire multiple nibbles from each of the other two node types, extension and leaf. There are three kinds of nodes in the trie:

Leaf: A two-item structure whose first item corresponds to the nibbles in the key not already accounted for by the accumulation of keys and branches traversed from the root. The hex-prefix encoding method is used and the second parameter to the function is required to be $true$.

Extension: A two-item structure whose first item corresponds to a series of nibbles of size greater than one that are shared by at least two distinct keys past the accumulation of nibbles keys and branches as traversed from the root. The hex-prefix encoding method is used and the second parameter to the function is required to be $false$.

Branch: A 17-item structure whose first sixteen items correspond to each of the sixteen possible nibble values for the keys at this point in their traversal. The 17th item is used in the case of this being a terminator node and thus a key being ended at this point in its traversal.

A branch is then only used when necessary; no branch nodes may exist that contain only a single non-zero entry. We may formally define this structure with the structural composition function c:

(187)
$$c(\mathfrak{I}, i) \equiv \begin{cases} \texttt{RLP}\Big(\big(\texttt{HP}(I_0[i..(\lVert I_0 \rVert - 1)], true), I_1 \big) \Big) & \text{if} \quad \lVert \mathfrak{I} \rVert = 1 \quad \text{where } \exists I : I \in \mathfrak{I} \\ \texttt{RLP}\Big(\big(\texttt{HP}(I_0[i..(j - 1)], false), n(\mathfrak{I}, j) \big) \Big) & \text{if} \quad i \neq j \quad \text{where } j = \arg\max_x : \exists \mathbf{l} : \lVert \mathbf{l} \rVert = x : \forall_{I \in \mathfrak{I}} : I_0[0..(x - 1)] = \mathbf{l} \\ \texttt{RLP}\Big((u(0), u(1), ..., u(15), v) \Big) & \text{otherwise} \quad \text{where } \begin{array}{rcl} u(j) & \equiv & n(\{I : I \in \mathfrak{I} \wedge I_0[i] = j\}, i + 1) \\ v & = & \begin{cases} I_1 & \text{if} \quad \exists I : I \in \mathfrak{I} \wedge \lVert I_0 \rVert = i \\ () & \text{otherwise} \end{cases} \end{array} \end{cases}$$

D.1. **Trie Database.** Thus no explicit assumptions are made concerning what data is stored and what is not, since that is an implementation-specific consideration; we simply define the identity function mapping the key-value set $\mathfrak{I}$ to a 32-byte hash and assert that only a single such hash exists for any $\mathfrak{I}$, which though not strictly true is accurate within acceptable precision given the Keccak hash's collision resistance. In reality, a sensible implementation will not fully recompute the trie root hash for each set.

A reasonable implementation will maintain a database of nodes determined from the computation of various tries or, more formally, it will memoise the function c. This strategy uses the nature of the trie to both easily recall the contents of any previous key-value set and to store multiple such sets in a very efficient manner. Due to the dependency relationship, Merkle-proofs may be constructed with an $O(\log N)$ space requirement that can demonstrate a particular leaf must exist within a trie of a given root hash.

Appendix E. Precompiled Contracts

For each precompiled contract, we make use of a template function, $\Xi_{\texttt{PRE}}$, which implements the out-of-mana checking.

(188)
$$\Xi_{\texttt{PRE}}(\boldsymbol{\sigma}, g, I) \equiv \begin{cases} (\varnothing, 0, A^0, ()) & \text{if} \quad g < g_r \\ (\boldsymbol{\sigma}, g - g_r, A^0, \mathbf{o}) & \text{otherwise} \end{cases}$$

The precompiled contracts each use these definitions and provide specifications for the $\mathbf{o}$ (the output data) and g_r, the mana requirements.

For the elliptic curve DSA recover VM execution function, we also define $\mathbf{d}$ to be the input data, well-defined for an infinite length by appending zeroes as required. Importantly in the case of an invalid signature ($\texttt{ECDSARECOVER}(h, v, r, s) = \varnothing$), then we have no output.

$$\begin{array}{rcl} \Xi_{\texttt{ECREC}} & \equiv & \Xi_{\texttt{PRE}} \quad \text{where:} \\ g_r & = & 3000 \\ |\mathbf{o}| & = & \begin{cases} 0 & \text{if} \quad \texttt{ECDSARECOVER}(h, v, r, s) = \varnothing \\ 32 & \text{otherwise} \end{cases} \\ \text{if} \quad |\mathbf{o}| = 32 : & & \\ \mathbf{o}[0..11] & = & 0 \\ \mathbf{o}[12..31] & = & \texttt{KEC}\big(\texttt{ECDSARECOVER}(h, v, r, s)\big)[12..31] \quad \text{where:} \\ \mathbf{d}[0..(|I_{\mathbf{d}}| - 1)] & = & I_{\mathbf{d}} \\ \mathbf{d}[|I_{\mathbf{d}}|..] & = & (0, 0, ...) \\ h & = & \mathbf{d}[0..31] \\ v & = & \mathbf{d}[32..63] \\ r & = & \mathbf{d}[64..95] \\ s & = & \mathbf{d}[96..127] \end{array}$$

The two hash functions, RIPEMD-160 and SHA2-256 are more trivially defined as an almost pass-through operation. Their mana usage is dependent on the input data size, a factor rounded up to the nearest number of words.

$$\begin{array}{rcl} \Xi_{\texttt{SHA256}} & \equiv & \Xi_{\texttt{PRE}} \quad \text{where:} \\ g_r & = & 60 + 12\Big\lceil \dfrac{|I_{\mathbf{d}}|}{32} \Big\rceil \\ \mathbf{o}[0..31] & = & \texttt{SHA256}(I_{\mathbf{d}}) \\ \Xi_{\texttt{RIP160}} & \equiv & \Xi_{\texttt{PRE}} \quad \text{where:} \\ g_r & = & 600 + 120\Big\lceil \dfrac{|I_{\mathbf{d}}|}{32} \Big\rceil \\ \mathbf{o}[0..11] & = & 0 \\ \mathbf{o}[12..31] & = & \texttt{RIPEMD160}(I_{\mathbf{d}}) \end{array}$$

For the purposes here, we assume we have well-defined standard cryptographic functions for RIPEMD-160 and SHA2-256 of the form:

$$\begin{aligned}\texttt{SHA256}(\mathbf{i} \in \mathbb{B}) &\equiv o \in \mathbb{B}_{32} \\ \texttt{RIPEMD160}(\mathbf{i} \in \mathbb{B}) &\equiv o \in \mathbb{B}_{20}\end{aligned}$$

Finally, the fourth contract, the identity function $\Xi_{\mathtt{ID}}$ simply defines the output as the input:

$$\begin{aligned}\Xi_{\mathtt{ID}} &\equiv \Xi_{\mathtt{PRE}} \quad \text{where:} \\ g_r &= 15 + 3\left\lceil \frac{|I_{\mathbf{d}}|}{32} \right\rceil \\ \mathbf{o} &= I_{\mathbf{d}}\end{aligned}$$

Appendix F. Signing Transactions

The method of signing transactions is similar to the 'Electrum style signatures'; it utilises the SECP-256k1 curve as described by Gura et al. [2004].

It is assumed that the sender has a valid private key p_r, which is a randomly selected positive integer (represented as a byte array of length 32 in big-endian form) in the range $[1, \mathtt{secp256k1n} - 1]$.

We assert the functions ECDSASIGN, ECDSARESTORE and ECDSAPUBKEY. These are formally defined in the literature.

$$\begin{aligned}\texttt{ECDSAPUBKEY}(p_r \in \mathbb{B}_{32}) &\equiv p_u \in \mathbb{B}_{64} \\ \texttt{ECDSASIGN}(e \in \mathbb{B}_{32}, p_r \in \mathbb{B}_{32}) &\equiv (v \in \mathbb{B}_1, r \in \mathbb{B}_{32}, s \in \mathbb{B}_{32}) \\ \texttt{ECDSARECOVER}(e \in \mathbb{B}_{32}, v \in \mathbb{B}_1, r \in \mathbb{B}_{32}, s \in \mathbb{B}_{32}) &\equiv p_u \in \mathbb{B}_{64}\end{aligned}$$

Where p_u is the public key, assumed to be a byte array of size 64 (formed from the concatenation of two positive integers each $< 2^{256}$) and p_r is the private key, a byte array of size 32 (or a single positive integer in the aforementioned range). It is assumed that v is the 'recovery id', a 1 byte value specifying the sign and finiteness of the curve point; this value is in the range of $[27, 30]$, however we declare the upper two possibilities, representing infinite values, invalid.

We declare that a signature is invalid unless all the following conditions are true:

(217) $$0 < r < \mathtt{secp256k1n}$$

(218) $$0 < s < \mathtt{secp256k1n} \div 2 + 1$$

(219) $$v \in \{27, 28\}$$

where:

(220) $$\mathtt{secp256k1n} = 115792089237316195423570985008687907852837564279074904382605163141518161494337$$

For a given private key, p_r, the Trustmachine address $A(p_r)$ (a 160-bit value) to which it corresponds is defined as the right most 160-bits of the Keccak hash of the corresponding ECDSA public key:

(221) $$A(p_r) = \mathcal{B}_{96..255}\big(\texttt{KEC}\big(\texttt{ECDSAPUBKEY}(p_r)\big)\big)$$

The message hash, $h(T)$, to be signed is the Keccak hash of the transaction without the latter three signature components, formally described as T_r, T_s and T_w:

$$\begin{aligned}L_S(T) &\equiv \begin{cases}(T_n, T_p, T_g, T_t, T_v, T_{\mathbf{i}}) & \text{if } T_t = 0 \\ (T_n, T_p, T_g, T_t, T_v, T_{\mathbf{d}}) & \text{otherwise}\end{cases} \\ h(T) &\equiv \texttt{KEC}(L_S(T))\end{aligned}$$

The signed transaction $M(T, p_r)$ is defined as:

$$\begin{aligned}M(T, p_r) &\equiv T \quad \text{except:} \\ (T_w, T_r, T_s) &= \texttt{ECDSASIGN}(h(T), p_r)\end{aligned}$$

We may then define the sender function S of the transaction as:

(226) $$S(T) \equiv \mathcal{B}_{96..255}\big(\texttt{KEC}\big(\texttt{ECDSARECOVER}(h(T), T_w, T_r, T_s)\big)\big)$$

The assertion that the sender of a signed transaction equals the address of the signer should be self-evident:

(227) $$\forall T : \forall p_r : S(G(T, p_r)) \equiv A(p_r)$$

Appendix G. Fee Schedule

The fee schedule M is a tuple of 31 scalar values corresponding to the relative costs, in mana, of a number of abstract operations that a transaction may effect.

Name	Value	Description*
M_{zero}	0	Nothing paid for operations of the set W_{zero}.
M_{base}	2	Amount of mana to pay for operations of the set W_{base}.
$M_{verylow}$	3	Amount of mana to pay for operations of the set $W_{verylow}$.
M_{low}	5	Amount of mana to pay for operations of the set W_{low}.
M_{mid}	8	Amount of mana to pay for operations of the set W_{mid}.
M_{high}	10	Amount of mana to pay for operations of the set W_{high}.
$M_{extcode}$	700	Amount of mana to pay for operations of the set $W_{extcode}$.
$M_{balance}$	400	Amount of mana to pay for a BALANCE operation.
M_{sload}	200	Paid for a SLOAD operation.
$M_{jumpdest}$	1	Paid for a JUMPDEST operation.
M_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
M_{sreset}	5000	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
$R_{selfdestruct}$	24000	Refund given (added into refund counter) for self-destructing an account.
$M_{selfdestruct}$	5000	Amount of mana to pay for a SELFDESTRUCT operation.
M_{create}	32000	Paid for a CREATE operation.
$M_{codedeposit}$	200	Paid per byte for a CREATE operation to succeed in placing code into state.
M_{call}	700	Paid for a CALL operation.
$M_{callvalue}$	9000	Paid for a non-zero value transfer as part of the CALL operation.
$M_{callstipend}$	2300	A stipend for the called contract subtracted from $M_{callvalue}$ for a non-zero value transfer.
$M_{newaccount}$	25000	Paid for a CALL or SELFDESTRUCT operation which creates an account.
M_{exp}	10	Partial payment for an EXP operation.
$M_{expbyte}$	50	Partial payment when multiplied by $\lceil \log_{256}(exponent) \rceil$ for the EXP operation.
M_{memory}	3	Paid for every additional word when expanding memory.
M_{txcreate}	32000	Paid by all contract-creating transactions after the *Homestead transition*.
$M_{txdatazero}$	4	Paid for every zero byte of data or code for a transaction.
$M_{txdatanonzero}$	68	Paid for every non-zero byte of data or code for a transaction.
$M_{transaction}$	21000	Paid for every transaction.
M_{log}	375	Partial payment for a LOG operation.
$M_{logdata}$	8	Paid for each byte in a LOG operation's data.
$M_{logtopic}$	375	Paid for each topic of a LOG operation.
M_{sha3}	30	Paid for each SHA3 operation.
$M_{sha3word}$	6	Paid for each word (rounded up) for input data to a SHA3 operation.
M_{copy}	3	Partial payment for *COPY operations, multiplied by words copied, rounded up.
$M_{blockhash}$	20	Payment for BLOCKHASH operation.

Appendix H. Virtual Machine Specification

When interpreting 256-bit binary values as integers, the representation is big-endian.

When a 256-bit machine datum is converted to and from a 160-bit address or hash, the rightwards (low-order for BE) 20 bytes are used and the left most 12 are discarded or filled with zeroes, thus the integer values (when the bytes are interpreted as big-endian) are equivalent.

H.1. **Mana Cost.** The general mana cost function, C, is defined as:

(228)

$$C(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \equiv C_{mem}(\boldsymbol{\mu}'_i) - C_{mem}(\boldsymbol{\mu}_i) + \begin{cases} C_{\text{SSTORE}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{if} \quad w = \text{SSTORE} \\ M_{exp} & \text{if} \quad w = \text{EXP} \wedge \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ M_{exp} + M_{expbyte} \times (1 + \lfloor \log_{256}(\boldsymbol{\mu}_{\mathbf{s}}[1]) \rfloor) & \text{if} \quad w = \text{EXP} \wedge \boldsymbol{\mu}_{\mathbf{s}}[1] > 0 \\ M_{verylow} + M_{copy} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[2] \div 32 \rceil & \text{if} \quad w = \text{CALLDATACOPY} \vee \text{CODECOPY} \\ M_{extcode} + M_{copy} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[3] \div 32 \rceil & \text{if} \quad w = \text{EXTCODECOPY} \\ M_{log} + M_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] & \text{if} \quad w = \text{LOG0} \\ M_{log} + M_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + M_{logtopic} & \text{if} \quad w = \text{LOG1} \\ M_{log} + M_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 2M_{logtopic} & \text{if} \quad w = \text{LOG2} \\ M_{log} + M_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 3M_{logtopic} & \text{if} \quad w = \text{LOG3} \\ M_{log} + M_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 4M_{logtopic} & \text{if} \quad w = \text{LOG4} \\ C_{\text{CALL}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{if} \quad w = \text{CALL} \vee \text{CALLCODE} \vee \text{DELEGATECALL} \\ C_{\text{SELFDESTRUCT}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{if} \quad w = \text{SELFDESTRUCT} \\ M_{create} & \text{if} \quad w = \text{CREATE} \\ M_{sha3} + M_{sha3word} \lceil \text{s}[1] \div 32 \rceil & \text{if} \quad w = \text{SHA3} \\ M_{jumpdest} & \text{if} \quad w = \text{JUMPDEST} \\ M_{sload} & \text{if} \quad w = \text{SLOAD} \\ M_{zero} & \text{if} \quad w \in W_{zero} \\ M_{base} & \text{if} \quad w \in W_{base} \\ M_{verylow} & \text{if} \quad w \in W_{verylow} \\ M_{low} & \text{if} \quad w \in W_{low} \\ M_{mid} & \text{if} \quad w \in W_{mid} \\ M_{high} & \text{if} \quad w \in W_{high} \\ M_{extcode} & \text{if} \quad w \in W_{extcode} \\ M_{balance} & \text{if} \quad w = \text{BALANCE} \\ M_{blockhash} & \text{if} \quad w = \text{BLOCKHASH} \end{cases}$$

(229)
$$w \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{pc}] & \text{if} \quad \boldsymbol{\mu}_{pc} < \|I_{\mathbf{b}}\| \\ \text{STOP} & \text{otherwise} \end{cases}$$

where:

(230)
$$C_{mem}(a) \equiv M_{memory} \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor$$

with C_{CALL}, $C_{\text{SELFDESTRUCT}}$ and C_{SSTORE} as specified in the appropriate section below. We define the following subsets of instructions:
W_{zero} = {STOP, RETURN}
W_{base} = {ADDRESS, ORIGIN, CALLER, CALLVALUE, CALLDATASIZE, CODESIZE, MANAPRICE, COINBASE, TIMESTAMP, NUMBER, DIFFICULTY, MANALIMIT, POP, PC, MSIZE, MANA}
$W_{verylow}$ = {ADD, SUB, NOT, LT, GT, SLT, SGT, EQ, ISZERO, AND, OR, XOR, BYTE, CALLDATALOAD, MLOAD, MSTORE, MSTORE8, PUSH*, DUP*, SWAP*}
W_{low} = {MUL, DIV, SDIV, MOD, SMOD, SIGNEXTEND}
W_{mid} = {ADDMOD, MULMOD, JUMP}
W_{high} = {JUMPI}
$W_{extcode}$ = {EXTCODESIZE}

Note the memory cost component, given as the product of M_{memory} and the maximum of 0 & the ceiling of the number of words in size that the memory must be over the current number of words, $\boldsymbol{\mu}_i$ in order that all accesses reference valid memory whether for read or write. Such accesses must be for non-zero number of bytes.

Referencing a zero length range (e.g. by attempting to pass it as the input range to a CALL) does not require memory to be extended to the beginning of the range. $\boldsymbol{\mu}'_i$ is defined as this new maximum number of words of active memory; special-cases are given where these two are not equal.

Note also that C_{mem} is the memory cost function (the expansion function being the difference between the cost before and after). It is a polynomial, with the higher-order coefficient divided and floored, and thus linear up to 724B of memory used, after which it costs substantially more.

While defining the instruction set, we defined the memory-expansion for range function, M_{EX}, thus:

(231)
$$M_{\text{EX}}(s, f, l) \equiv \begin{cases} s & \text{if} \quad l = 0 \\ \max(s, \lceil (f + l) \div 32 \rceil) & \text{otherwise} \end{cases}$$

Another useful function is "all but one 64th" function L defined as:

(232) $$L(n) \equiv n - \lfloor n/64 \rfloor$$

H.2. **Instruction Set.** As previously specified in section 10, these definitions take place in the final context there. In particular we assume O is the TVM state-progression function and define the terms pertaining to the next cycle's state $(\boldsymbol{\sigma}', \boldsymbol{\mu}')$ such that:

(233) $$O(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \equiv (\boldsymbol{\sigma}', \boldsymbol{\mu}', A', I) \quad \text{with exceptions, as noted}$$

Here given are the various exceptions to the state transition rules given in section 10 specified for each instruction, together with the additional instruction-specific definitions of J and C. For each instruction, also specified is α, the additional items placed on the stack and δ, the items removed from stack, as defined in section 10.

0s: Stop and Arithmetic Operations

All arithmetic is modulo 2^{256} unless otherwise noted. The zero-th power of zero 0^0 is defined to be one.

Value	Mnemonic	δ	α	Description
0x00	STOP	0	0	Halts execution.
0x01	ADD	2	1	Addition operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1]$
0x02	MUL	2	1	Multiplication operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0] \times \boldsymbol{\mu}_{\mathbf{s}}[1]$
0x03	SUB	2	1	Subtraction operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0] - \boldsymbol{\mu}_{\mathbf{s}}[1]$
0x04	DIV	2	1	Integer division operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ \lfloor \boldsymbol{\mu}_{\mathbf{s}}[0] \div \boldsymbol{\mu}_{\mathbf{s}}[1] \rfloor & \text{otherwise} \end{cases}$
0x05	SDIV	2	1	Signed integer division operation (truncated). $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ -2^{255} & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] = -2^{255} \wedge \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = -1 \\ \mathbf{sgn}(\boldsymbol{\mu}_{\mathbf{s}}[0] \div \boldsymbol{\mu}_{\mathbf{s}}[1]) \lfloor \lvert \boldsymbol{\mu}_{\mathbf{s}}[0] \div \boldsymbol{\mu}_{\mathbf{s}}[1] \rvert \rfloor & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers. Note the overflow semantic when -2^{255} is negated.
0x06	MOD	2	1	Modulo remainder operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ \boldsymbol{\mu}_{\mathbf{s}}[0] \bmod \boldsymbol{\mu}_{\mathbf{s}}[1] & \text{otherwise} \end{cases}$
0x07	SMOD	2	1	Signed modulo remainder operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ \mathbf{sgn}(\boldsymbol{\mu}_{\mathbf{s}}[0])(\lvert \boldsymbol{\mu}_{\mathbf{s}}[0] \rvert \bmod \lvert \boldsymbol{\mu}_{\mathbf{s}}[1] \rvert) & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers.
0x08	ADDMOD	3	1	Modulo addition operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] = 0 \\ (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1]) \mod \boldsymbol{\mu}_{\mathbf{s}}[2] & \text{otherwise} \end{cases}$ All intermediate calculations of this operation are not subject to the 2^{256} modulo.
0x09	MULMOD	3	1	Modulo multiplication operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] = 0 \\ (\boldsymbol{\mu}_{\mathbf{s}}[0] \times \boldsymbol{\mu}_{\mathbf{s}}[1]) \mod \boldsymbol{\mu}_{\mathbf{s}}[2] & \text{otherwise} \end{cases}$ All intermediate calculations of this operation are not subject to the 2^{256} modulo.
0x0a	EXP	2	1	Exponential operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]^{\boldsymbol{\mu}_{\mathbf{s}}[1]}$
0x0b	SIGNEXTEND	2	1	Extend length of two's complement signed integer. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_i \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{s}}[1]_t & \text{if} \quad i \leqslant t \quad \text{where } t = 256 - 8(\boldsymbol{\mu}_{\mathbf{s}}[0] + 1) \\ \boldsymbol{\mu}_{\mathbf{s}}[1]_i & \text{otherwise} \end{cases}$

$\boldsymbol{\mu}_{\mathbf{s}}[x]_i$ gives the ith bit (counting from zero) of $\boldsymbol{\mu}_{\mathbf{s}}[x]$

10s: Comparison & Bitwise Logic Operations

Value	Mnemonic	δ	α	Description
0x10	LT	2	1	Less-than comparison. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] < \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$
0x11	GT	2	1	Greater-than comparison. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] > \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$
0x12	SLT	2	1	Signed less-than comparison. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] < \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers.
0x13	SGT	2	1	Signed greater-than comparison. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] > \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers.
0x14	EQ	2	1	Equality comparison. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] = \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$
0x15	ISZERO	1	1	Simple not operator. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] = 0 \\ 0 & \text{otherwise} \end{cases}$
0x16	AND	2	1	Bitwise AND operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_i \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]_i \land \boldsymbol{\mu}_{\mathbf{s}}[1]_i$
0x17	OR	2	1	Bitwise OR operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_i \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]_i \lor \boldsymbol{\mu}_{\mathbf{s}}[1]_i$
0x18	XOR	2	1	Bitwise XOR operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_i \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]_i \oplus \boldsymbol{\mu}_{\mathbf{s}}[1]_i$
0x19	NOT	1	1	Bitwise NOT operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_i \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0]_i = 0 \\ 0 & \text{otherwise} \end{cases}$
0x1a	BYTE	2	1	Retrieve single byte from word. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_i \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{s}}[1]_{(i + 8\boldsymbol{\mu}_{\mathbf{s}}[0])} & \text{if} \quad i < 8 \land \boldsymbol{\mu}_{\mathbf{s}}[0] < 32 \\ 0 & \text{otherwise} \end{cases}$ For Nth byte, we count from the left (i.e. N=0 would be the most significant in big endian).

20s: SHA3

Value	Mnemonic	δ	α	Description
0x20	SHA3	2	1	Compute Keccak-256 hash. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \texttt{Keccak}(\boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1] - 1)])$ $\boldsymbol{\mu}'_i \equiv M_{\text{EX}}(\boldsymbol{\mu}_i, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[1])$

30s: Environmental Information

Value	Mnemonic	δ	α	Description
0x30	ADDRESS	0	1	Get address of currently executing account. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_a$
0x31	BALANCE	1	1	Get balance of the given account. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0]]_b & \text{if} \quad \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \mod 2^{160}] \neq \varnothing \\ 0 & \text{otherwise} \end{cases}$
0x32	ORIGIN	0	1	Get execution origination address. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_o$ This is the sender of original transaction; it is never an account with non-empty associated code.
0x33	CALLER	0	1	Get caller address. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_s$ This is the address of the account that is directly responsible for this execution.
0x34	CALLVALUE	0	1	Get deposited value by the instruction/transaction responsible for this execution. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_v$
0x35	CALLDATALOAD	1	1	Get input data of current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathbf{d}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + 31)] \quad \text{with} \quad I_{\mathbf{d}}[x] = 0 \quad \text{if} \quad x \geqslant \lVert I_{\mathbf{d}} \rVert$ This pertains to the input data passed with the message call instruction or transaction.
0x36	CALLDATASIZE	0	1	Get size of input data in current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \lVert I_{\mathbf{d}} \rVert$ This pertains to the input data passed with the message call instruction or transaction.
0x37	CALLDATACOPY	3	0	Copy input data in current environment to memory. $\forall_{i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[2]-1\}} \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] + i] \equiv \begin{cases} I_{\mathbf{d}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] + i < \lVert I_{\mathbf{d}} \rVert \\ 0 & \text{otherwise} \end{cases}$ The additions in $\boldsymbol{\mu}_{\mathbf{s}}[1] + i$ are not subject to the 2^{256} modulo. $\boldsymbol{\mu}'_i \equiv M_{\text{EX}}(\boldsymbol{\mu}_i, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[2])$ This pertains to the input data passed with the message call instruction or transaction.
0x38	CODESIZE	0	1	Get size of code running in current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \lVert I_{\mathbf{b}} \rVert$
0x39	CODECOPY	3	0	Copy code running in current environment to memory. $\forall_{i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[2]-1\}} \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] + i] \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] + i < \lVert I_{\mathbf{b}} \rVert \\ \text{STOP} & \text{otherwise} \end{cases}$ $\boldsymbol{\mu}'_i \equiv M_{\text{EX}}(\boldsymbol{\mu}_i, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[2])$ The additions in $\boldsymbol{\mu}_{\mathbf{s}}[1] + i$ are not subject to the 2^{256} modulo.
0x3a	MANAPRICE	0	1	Get price of mana in current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_p$ This is mana price specified by the originating transaction.
0x3b	EXTCODESIZE	1	1	Get size of an account's code. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \lVert \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \mod 2^{160}]_c \rVert$
0x3c	EXTCODECOPY	4	0	Copy an account's code to memory. $\forall_{i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[3]-1\}} \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] \equiv \begin{cases} \mathbf{c}[\boldsymbol{\mu}_{\mathbf{s}}[2] + i] & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] + i < \lVert \mathbf{c} \rVert \\ \text{STOP} & \text{otherwise} \end{cases}$ where $\mathbf{c} \equiv \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \mod 2^{160}]_c$ $\boldsymbol{\mu}'_i \equiv M_{\text{EX}}(\boldsymbol{\mu}_i, \boldsymbol{\mu}_{\mathbf{s}}[1], \boldsymbol{\mu}_{\mathbf{s}}[3])$ The additions in $\boldsymbol{\mu}_{\mathbf{s}}[2] + i$ are not subject to the 2^{256} modulo.

40s: Block Information

Value	Mnemonic	δ	α	Description
0x40	BLOCKHASH	1	1	Get the hash of one of the 256 most recent complete blocks. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv P(I_{H_p}, \boldsymbol{\mu}_{\mathbf{s}}[0], 0)$ where P is the hash of a block of a particular number, up to a maximum age. 0 is left on the stack if the looked for block number is greater than the current block number or more than 256 blocks behind the current block. $P(h, n, a) \equiv \begin{cases} 0 & \text{if} \quad n > H_i \vee a = 256 \vee h = 0 \\ h & \text{if} \quad n = H_i \\ P(H_p, n, a+1) & \text{otherwise} \end{cases}$ and we assert the header H can be determined as its hash is the parent hash in the block following it.
0x41	COINBASE	0	1	Get the block's beneficiary address. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{Hc}$
0x42	TIMESTAMP	0	1	Get the block's timestamp. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{Hs}$
0x43	NUMBER	0	1	Get the block's number. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{Hi}$
0x44	DIFFICULTY	0	1	Get the block's difficulty. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{Hd}$
0x45	MANALIMIT	0	1	Get the block's mana limit. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{Hl}$

50s: Stack, Memory, Storage and Flow Operations

Value	Mnemonic	δ	α	Description
0x50	POP	1	0	Remove item from stack.
0x51	MLOAD	1	1	Load word from memory. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + 31)]$ $\boldsymbol{\mu}'_i \equiv \max(\boldsymbol{\mu}_i, \lceil(\boldsymbol{\mu}_{\mathbf{s}}[0] + 32) \div 32\rceil)$ The addition in the calculation of $\boldsymbol{\mu}'_i$ is not subject to the 2^{256} modulo.
0x52	MSTORE	2	0	Save word to memory. $\boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + 31)] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$ $\boldsymbol{\mu}'_i \equiv \max(\boldsymbol{\mu}_i, \lceil(\boldsymbol{\mu}_{\mathbf{s}}[0] + 32) \div 32\rceil)$ The addition in the calculation of $\boldsymbol{\mu}'_i$ is not subject to the 2^{256} modulo.
0x53	MSTORE8	2	0	Save byte to memory. $\boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \equiv (\boldsymbol{\mu}_{\mathbf{s}}[1] \bmod 256)$ $\boldsymbol{\mu}'_i \equiv \max(\boldsymbol{\mu}_i, \lceil(\boldsymbol{\mu}_{\mathbf{s}}[0] + 1) \div 32\rceil)$ The addition in the calculation of $\boldsymbol{\mu}'_i$ is not subject to the 2^{256} modulo.
0x54	SLOAD	1	1	Load word from storage. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\sigma}[I_a]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]]$
0x55	SSTORE	2	0	Save word to storage. $\boldsymbol{\sigma}'[I_a]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$ $C_{\text{SSTORE}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} M_{sset} & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] \neq 0 \ \wedge \ \boldsymbol{\sigma}[I_a]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] = 0 \\ M_{sreset} & \text{otherwise} \end{cases}$ $A'_r \equiv A_r + \begin{cases} R_{sclear} & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \ \wedge \ \boldsymbol{\sigma}[I_a]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \neq 0 \\ 0 & \text{otherwise} \end{cases}$
0x56	JUMP	1	0	Alter the program counter. $J_{\text{JUMP}}(\boldsymbol{\mu}) \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$ This has the effect of writing said value to $\boldsymbol{\mu}_{pc}$. See section 10.
0x57	JUMPI	2	0	Conditionally alter the program counter. $J_{\text{JUMPI}}(\boldsymbol{\mu}) \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{s}}[0] & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] \neq 0 \\ \boldsymbol{\mu}_{pc} + 1 & \text{otherwise} \end{cases}$ This has the effect of writing said value to $\boldsymbol{\mu}_{pc}$. See section 10.
0x58	PC	0	1	Get the value of the program counter *prior* to the increment corresponding to this instruction. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{pc}$
0x59	MSIZE	0	1	Get the size of active memory in bytes. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv 32\boldsymbol{\mu}_i$
0x5a	MANA	0	1	Get the amount of available mana, including the corresponding reduction for the cost of this instruction. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_g$
0x5b	JUMPDEST	0	0	Mark a valid destination for jumps. This operation has no effect on machine state during execution.

60s & 70s: Push Operations

Value	Mnemonic	δ	α	Description
0x60	PUSH1	0	1	Place 1 byte item on stack. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv c(\boldsymbol{\mu}_{pc} + 1)$ where $c(x) \equiv \begin{cases} I_{\mathbf{b}}[x] & \text{if} \quad x < \lVert I_{\mathbf{b}} \rVert \\ 0 & \text{otherwise} \end{cases}$ The bytes are read in line from the program code's bytes array. The function c ensures the bytes default to zero if they extend past the limits. The byte is right-aligned (takes the lowest significant place in big endian).
0x61	PUSH2	0	1	Place 2-byte item on stack. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{c}\big((\boldsymbol{\mu}_{pc} + 1) \dots (\boldsymbol{\mu}_{pc} + 2)\big)$ with $\boldsymbol{c}(\boldsymbol{x}) \equiv (c(\boldsymbol{x}_0), ..., c(\boldsymbol{x}_{\lVert x \rVert - 1}))$ with c as defined as above. The bytes are right-aligned (takes the lowest significant place in big endian).
⋮	⋮	⋮	⋮	⋮
0x7f	PUSH32	0	1	Place 32-byte (full word) item on stack. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{c}\big((\boldsymbol{\mu}_{pc} + 1) \dots (\boldsymbol{\mu}_{pc} + 32)\big)$ where $\boldsymbol{c}$ is defined as above. The bytes are right-aligned (takes the lowest significant place in big endian).

80s: Duplication Operations

Value	Mnemonic	δ	α	Description
0x80	DUP1	1	2	Duplicate 1st stack item. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$
0x81	DUP2	2	3	Duplicate 2nd stack item. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$
⋮	⋮	⋮	⋮	⋮
0x8f	DUP16	16	17	Duplicate 16th stack item. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[15]$

90s: Exchange Operations

Value	Mnemonic	δ	α	Description
0x90	SWAP1	2	2	Exchange 1st and 2nd stack items. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$ $\boldsymbol{\mu}'_{\mathbf{s}}[1] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$
0x91	SWAP2	3	3	Exchange 1st and 3rd stack items. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[2]$ $\boldsymbol{\mu}'_{\mathbf{s}}[2] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$
⋮	⋮	⋮	⋮	⋮
0x9f	SWAP16	17	17	Exchange 1st and 17th stack items. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[16]$ $\boldsymbol{\mu}'_{\mathbf{s}}[16] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$

a0s: Logging Operations

For all logging operations, the state change is to append an additional log entry on to the substate's log series:
$A'_{\mathbf{l}} \equiv A_{\mathbf{l}} \cdot (I_a, \mathbf{t}, \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1] - 1)])$
and to update the memory consumption counter:
$\boldsymbol{\mu}'_i \equiv M_{\mathsf{EX}}(\boldsymbol{\mu}_i, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[1])$
The entry's topic series, $\mathbf{t}$, differs accordingly:

Value	**Mnemonic**	δ	α	**Description**
0xa0	LOG0	2	0	Append log record with no topics. $\mathbf{t} \equiv ()$
0xa1	LOG1	3	0	Append log record with one topic. $\mathbf{t} \equiv (\boldsymbol{\mu}_{\mathbf{s}}[2])$
⋮	⋮	⋮	⋮	⋮
0xa4	LOG4	6	0	Append log record with four topics. $\mathbf{t} \equiv (\boldsymbol{\mu}_{\mathbf{s}}[2], \boldsymbol{\mu}_{\mathbf{s}}[3], \boldsymbol{\mu}_{\mathbf{s}}[4], \boldsymbol{\mu}_{\mathbf{s}}[5])$

f0s: System operations

Value	Mnemonic	δ	α	Description
0xf0	CREATE	3	1	Create a new account with associated code. $\mathbf{i} \equiv \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[1] \dots (\boldsymbol{\mu}_{\mathbf{s}}[1] + \boldsymbol{\mu}_{\mathbf{s}}[2] - 1)]$ $(\boldsymbol{\sigma}', \boldsymbol{\mu}'_g, A^+) \equiv \begin{cases} \Lambda(\boldsymbol{\sigma}^*, I_a, I_o, L(\boldsymbol{\mu}_g), I_p, \boldsymbol{\mu}_{\mathbf{s}}[0], \mathbf{i}, I_e + 1) & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] \leqslant \boldsymbol{\sigma}[I_a]_b \ \wedge \ I_e < 1024 \\ (\boldsymbol{\sigma}, \boldsymbol{\mu}_g, \varnothing) & \text{otherwise} \end{cases}$ $\boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma} \quad \text{except} \quad \boldsymbol{\sigma}^*[I_a]_n = \boldsymbol{\sigma}[I_a]_n + 1$ $A' \equiv A \Cup A^+$ which implies: $A'_{\mathbf{s}} \equiv A_{\mathbf{s}} \cup A^+_{\mathbf{s}} \quad \wedge \quad A'_{\mathbf{l}} \equiv A_{\mathbf{l}} \cdot A^+_{\mathbf{l}} \quad \wedge \quad A'_{\mathbf{r}} \equiv A_{\mathbf{r}} + A^+_{\mathbf{r}}$ $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv x$ where $x = 0$ if the code execution for this operation failed due to an exceptional halting $Z(\boldsymbol{\sigma}^*, \boldsymbol{\mu}, I) = \top$ or $I_e = 1024$ (the maximum call depth limit is reached) or $\boldsymbol{\mu}_{\mathbf{s}}[0] > \boldsymbol{\sigma}[I_a]_b$ (balance of the caller is too low to fulfil the value transfer); and otherwise $x = A(I_a, \boldsymbol{\sigma}[I_a]_n)$, the address of the newly created account, otherwise. $\boldsymbol{\mu}'_i \equiv M_{\text{EX}}(\boldsymbol{\mu}_i, \boldsymbol{\mu}_{\mathbf{s}}[1], \boldsymbol{\mu}_{\mathbf{s}}[2])$ Thus the operand order is: value, input offset, input size.
0xf1	CALL	7	1	Message-call into an account. $\mathbf{i} \equiv \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[3] \dots (\boldsymbol{\mu}_{\mathbf{s}}[3] + \boldsymbol{\mu}_{\mathbf{s}}[4] - 1)]$ $(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}, I_a, I_o, t, t, & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] \leqslant \boldsymbol{\sigma}[I_a]_b \ \wedge \\ \quad C_{\text{CALLMANA}}(\boldsymbol{\mu}), I_p, \boldsymbol{\mu}_{\mathbf{s}}[2], \boldsymbol{\mu}_{\mathbf{s}}[2], \mathbf{i}, I_e + 1) & \quad I_e < 1024 \\ (\boldsymbol{\sigma}, g, \varnothing, ()) & \text{otherwise} \end{cases}$ $n \equiv \min(\{\boldsymbol{\mu}_{\mathbf{s}}[6], \lvert\mathbf{o}\rvert\})$ $\boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[5] \dots (\boldsymbol{\mu}_{\mathbf{s}}[5] + n - 1)] = \mathbf{o}[0 \dots (n - 1)]$ $\boldsymbol{\mu}'_g \equiv \boldsymbol{\mu}_g + g'$ $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv x$ $A' \equiv A \Cup A^+$ $t \equiv \boldsymbol{\mu}_{\mathbf{s}}[1] \mod 2^{160}$ where $x = 0$ if the code execution for this operation failed due to an exceptional halting $Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) = \top$ or if $\boldsymbol{\mu}_{\mathbf{s}}[2] > \boldsymbol{\sigma}[I_a]_b$ (not enough funds) or $I_e = 1024$ (call depth limit reached); $x = 1$ otherwise. $\boldsymbol{\mu}'_i \equiv M_{\text{EX}}(M_{\text{EX}}(\boldsymbol{\mu}_i, \boldsymbol{\mu}_{\mathbf{s}}[3], \boldsymbol{\mu}_{\mathbf{s}}[4]), \boldsymbol{\mu}_{\mathbf{s}}[5], \boldsymbol{\mu}_{\mathbf{s}}[6])$ Thus the operand order is: mana, to, value, in offset, in size, out offset, out size. $C_{\text{CALL}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv C_{\text{MANACAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) + C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu})$ $C_{\text{CALLMANA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} C_{\text{MANACAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) + M_{callstipend} & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] \neq 0 \\ C_{\text{MANACAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{otherwise} \end{cases}$ $C_{\text{MANACAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} \min\{L(\boldsymbol{\mu}_g - C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu})), \boldsymbol{\mu}_{\mathbf{s}}[0]\} & \text{if} \quad \boldsymbol{\mu}_g \geq C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \\ \boldsymbol{\mu}_{\mathbf{s}}[0] & \text{otherwise} \end{cases}$ $C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv M_{call} + C_{\text{XFER}}(\boldsymbol{\mu}) + C_{\text{NEW}}(\boldsymbol{\sigma}, \boldsymbol{\mu})$ $C_{\text{XFER}}(\boldsymbol{\mu}) \equiv \begin{cases} M_{callvalue} & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] \neq 0 \\ 0 & \text{otherwise} \end{cases}$ $C_{\text{NEW}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} M_{newaccount} & \text{if} \quad \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[1] \mod 2^{160}] = \varnothing \\ 0 & \text{otherwise} \end{cases}$
0xf2	CALLCODE	7	1	Message-call into this account with an alternative account's code. Exactly equivalent to CALL except: $(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}^*, I_a, I_o, I_a, t, & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] \leqslant \boldsymbol{\sigma}[I_a]_b \ \wedge \\ \quad C_{\text{CALLMANA}}(\boldsymbol{\mu}), I_p, \boldsymbol{\mu}_{\mathbf{s}}[2], \boldsymbol{\mu}_{\mathbf{s}}[2], \mathbf{i}, I_e + 1) & \quad I_e < 1024 \\ (\boldsymbol{\sigma}, g, \varnothing, ()) & \text{otherwise} \end{cases}$ Note the change in the fourth parameter to the call Θ from the 2nd stack value $\boldsymbol{\mu}_{\mathbf{s}}[1]$ (as in CALL) to the present address I_a. This means that the recipient is in fact the same account as at present, simply that the code is overwritten.
0xf3	RETURN	2	0	Halt execution returning output data. $H_{\text{RETURN}}(\boldsymbol{\mu}) \equiv \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1] - 1)]$ This has the effect of halting the execution at this point with output defined. See section 10. $\boldsymbol{\mu}'_i \equiv M_{\text{EX}}(\boldsymbol{\mu}_i, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[1])$

0xf4	DELEGATECALL	6	1	Message-call into this account with an alternative account's code, but persisting the current values for *sender* and *value*. Compared with CALL, DELEGATECALL takes one fewer arguments. The omitted argument is $\boldsymbol{\mu}_{\mathbf{s}}[2]$. As a result, $\boldsymbol{\mu}_{\mathbf{s}}[3]$, $\boldsymbol{\mu}_{\mathbf{s}}[4]$, $\boldsymbol{\mu}_{\mathbf{s}}[5]$ and $\boldsymbol{\mu}_{\mathbf{s}}[6]$ in the definition of CALL should respectively be replaced with $\boldsymbol{\mu}_{\mathbf{s}}[2]$, $\boldsymbol{\mu}_{\mathbf{s}}[3]$, $\boldsymbol{\mu}_{\mathbf{s}}[4]$ and $\boldsymbol{\mu}_{\mathbf{s}}[5]$. Otherwise exactly equivalent to CALL except: $(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}^*, I_s, I_o, I_a, t, \\ \quad \boldsymbol{\mu}_{\mathbf{s}}[0], I_p, 0, I_v, \mathbf{i}, I_e + 1) & \text{if} \quad I_v \leqslant \boldsymbol{\sigma}[I_a]_b \ \wedge \ I_e < 1024 \\ (\boldsymbol{\sigma}, g, \varnothing, ()) & \text{otherwise} \end{cases}$ Note the changes (in addition to that of the fourth parameter) to the second and ninth parameters to the call Θ. This means that the recipient is in fact the same account as at present, simply that the code is overwritten *and* the context is almost entirely identical.
0xfe	INVALID	$\varnothing$	$\varnothing$	Designated invalid instruction.
0xff	SELFDESTRUCT	1	0	Halt execution and register account for later deletion. $A'_{\mathbf{s}} \equiv A_{\mathbf{s}} \cup \{I_a\}$ $\boldsymbol{\sigma}'[\boldsymbol{\mu}_{\mathbf{s}}[0] \mod 2^{160}]_b \equiv \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \mod 2^{160}]_b + \boldsymbol{\sigma}[I_a]_b$ $\boldsymbol{\sigma}'[I_a]_b \equiv 0$ $A'_r \equiv A_r + \begin{cases} R_{selfdestruct} & \text{if} \quad I_a \notin A_{\mathbf{s}} \\ 0 & \text{otherwise} \end{cases}$ $C_{\text{SELFDESTRUCT}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv M_{selfdestruct} + \begin{cases} M_{newaccount} & \text{if} \quad \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \mod 2^{160}] = \varnothing \\ 0 & \text{otherwise} \end{cases}$

Appendix I. Genesis Block

The genesis block is 15 items, and is specified thus:

(234) $$\big((0_{256}, \texttt{KEC}\big(\texttt{RLP}\big(()\big)\big), 0_{160}, stateRoot, 0, 0, 0_{2048}, 2^{17}, 0, 0, 3141592, time, 0, 0_{256}, \texttt{KEC}\big((42)\big)\big), (), () \big)$$

Where 0_{256} refers to the parent hash, a 256-bit hash which is all zeroes; 0_{160} refers to the beneficiary address, a 160-bit hash which is all zeroes; 0_{2048} refers to the log bloom, 2048-bit of all zeros; 2^{17} refers to the difficulty; the transaction trie root, receipt trie root, mana used, block number and extradata are both 0, being equivalent to the empty byte array. The sequences of both ommers and transactions are empty and represented by $()$. $\texttt{KEC}\big((42)\big)$ refers to the Keccak hash of a byte array of length one whose first and only byte is of value 42, used for the nonce. $\texttt{KEC}\big(\texttt{RLP}\big(()\big)\big)$ value refers to the hash of the ommer lists in RLP, both empty lists.

The proof-of-concept series include a development premine, making the state root hash some value $stateRoot$. Also $time$ will be set to the initial timestamp of the genesis block. The latest documentation should be consulted for those values.

Appendix J. Ethash

J.1. **Definitions.** We employ the following definitions:

Name	Value	Description
$J_{wordbytes}$	4	Bytes in word.
$J_{datasetinit}$	2^{30}	Bytes in dataset at genesis.
$J_{datasetgrowth}$	2^{23}	Dataset growth per epoch.
$J_{cacheinit}$	2^{24}	Bytes in cache at genesis.
$J_{cachegrowth}$	2^{17}	Cache growth per epoch.
J_{epoch}	30000	Blocks per epoch.
$J_{mixbytes}$	128	mix length in bytes.
$J_{hashbytes}$	64	Hash length in bytes.
$J_{parents}$	256	Number of parents of each dataset element.
$J_{cacherounds}$	3	Number of rounds in cache production.
$J_{accesses}$	64	Number of accesses in hashimoto loop.

J.2. **Size of dataset and cache.** The size for Ethash's cache $\mathbf{c} \in \mathbb{B}$ and dataset $\mathbf{d} \in \mathbb{B}$ depend on the epoch, which in turn depends on the block number.

(235) $$E_{epoch}(H_i) = \left\lfloor \frac{H_i}{J_{epoch}} \right\rfloor$$

The size of the dataset growth by $J_{datasetgrowth}$ bytes, and the size of the cache by $J_{cachegrowth}$ bytes, every epoch. In order to avoid regularity leading to cyclic behavior, the size must be a prime number. Therefore the size is reduced by a multiple of $J_{mixbytes}$, for the dataset, and $J_{hashbytes}$ for the cache. Let $d_{size} = \|\mathbf{d}\|$ be the size of the dataset. Which is calculated using

(236) $$d_{size} = E_{prime}(J_{datasetinit} + J_{datasetgrowth} \cdot E_{epoch} - J_{mixbytes}, J_{mixbytes})$$

The size of the cache, c_{size}, is calculated using

(237) $$c_{size} = E_{prime}(J_{cacheinit} + J_{cachegrowth} \cdot E_{epoch} - J_{hashbytes}, J_{hashbytes})$$

(238) $$E_{prime}(x, y) = \begin{cases} x & \text{if} \quad x/y \in \mathbb{P} \\ E_{prime}(x - 1 \cdot y, y) & \text{otherwise} \end{cases}$$

J.3. **Dataset generation.** In order the generate the dataset we need the cache $\mathbf{c}$, which is an array of bytes. It depends on the cache size c_{size} and the seed hash $\mathbf{s} \in \mathbb{B}_{32}$.

J.3.1. *Seed hash.* The seed hash is different for every epoch. For the first epoch it is the Keccak-256 hash of a series of 32 bytes of zeros. For every other epoch it is always the Keccak-256 hash of the previous seed hash:

(239) $$\mathbf{s} = C_{seedhash}(H_i)$$

(240) $$C_{seedhash}(H_i) = \begin{cases} \texttt{KEC}(\mathbf{0}_{32}) & \text{if} \quad E_{epoch}(H_i) = 0 \\ \texttt{KEC}(C_{seedhash}(H_i - J_{epoch})) & \text{otherwise} \end{cases}$$

With $\mathbf{0}_{32}$ being 32 bytes of zeros.

J.3.2. *Cache.* The cache production process involves using the seed hash to first sequentially filling up c_{size} bytes of memory, then performing $J_{cacherounds}$ passes of the RandMemoHash algorithm created by Lerner [2014]. The initial cache $\mathbf{c}'$, being an array of arrays of single bytes, will be constructed as follows.

We define the array $\mathbf{c}_i$, consisting of 64 single bytes, as the ith element of the initial cache:

(241) $$\mathbf{c}_i = \begin{cases} \texttt{KEC512}(\mathbf{s}) & \text{if} \quad i = 0 \\ \texttt{KEC512}(\mathbf{c}_{i-1}) & \text{otherwise} \end{cases}$$

Therefore $\mathbf{c}'$ can be defined as

(242) $$\mathbf{c}'[i] = \mathbf{c}_i \quad \forall \quad i < n$$

(243) $$n = \left\lfloor \frac{c_{size}}{J_{hashbytes}} \right\rfloor$$

The cache is calculated by performing $J_{cacherounds}$ rounds of the RandMemoHash algorithm to the inital cache $\mathbf{c}'$:

(244) $$\mathbf{c} = E_{cacherounds}(\mathbf{c}', J_{cacherounds})$$

(245) $$E_{cacherounds}(\mathbf{x}, y) = \begin{cases} \mathbf{x} & \text{if} \quad y = 0 \\ E_{\text{RMH}}(\mathbf{x}) & \text{if} \quad y = 1 \\ E_{cacherounds}(E_{\text{RMH}}(\mathbf{x}), y - 1) & \text{otherwise} \end{cases}$$

Where a single round modifies each subset of the cache as follows:

(246) $$E_{\text{RMH}}(\mathbf{x}) = \big(E_{rmh}(\mathbf{x}, 0), E_{rmh}(\mathbf{x}, 1), ..., E_{rmh}(\mathbf{x}, n - 1)\big)$$

(247) $$E_{rmh}(\mathbf{x}, i) = \texttt{KEC512}(\mathbf{x}'[(i - 1 + n) \mod n] \oplus \mathbf{x}'[\mathbf{x}'[i][0] \mod n]) \\ \text{with} \quad \mathbf{x}' = \mathbf{x} \quad \text{except} \quad \mathbf{x}'[j] = E_{rmh}(\mathbf{x}, j) \quad \forall \quad j < i$$

J.3.3. *Full dataset calculation.* Essentially, we combine data from $J_{parents}$ pseudorandomly selected cache nodes, and hash that to compute the dataset. The entire dataset is then generated by a number of items, each $J_{hashbytes}$ bytes in size:

(248) $$\mathbf{d}[i] = E_{datasetitem}(\mathbf{c}, i) \quad \forall \quad i < \left\lfloor \frac{d_{size}}{J_{hashbytes}} \right\rfloor$$

In order to calculate the single item we use an algorithm inspired by the FNV hash (Glenn Fowler [1991]) in some cases as a non-associative substitute for XOR.

(249) $$E_{\text{FNV}}(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot (0\text{x}01000193 \oplus \mathbf{y})) \mod 2^{32}$$

The single item of the dataset can now be calculated as:

(250) $$E_{datasetitem}(\mathbf{c}, i) = E_{parents}(\mathbf{c}, i, -1, \varnothing)$$

(251) $$E_{parents}(\mathbf{c}, i, p, \mathbf{m}) = \begin{cases} E_{parents}(\mathbf{c}, i, p + 1, E_{mix}(\mathbf{m}, \mathbf{c}, i, p + 1)) & \text{if} \quad p < J_{parents} - 2 \\ E_{mix}(\mathbf{m}, \mathbf{c}, i, p + 1) & \text{otherwise} \end{cases}$$

(252) $$E_{mix}(\mathbf{m}, \mathbf{c}, i, p) = \begin{cases} \texttt{KEC512}(\mathbf{c}[i \mod c_{size}] \oplus i) & \text{if} \quad p = 0 \\ E_{\text{FNV}}\big(\mathbf{m}, \mathbf{c}[E_{\text{FNV}}(i \oplus p, \mathbf{m}[p \mod \lfloor J_{hashbytes}/J_{wordbytes} \rfloor]) \mod c_{size}]\big) & \text{otherwise} \end{cases}$$

J.4. **Proof-of-work function.** Essentially, we maintain a "mix" $J_{mixbytes}$ bytes wide, and repeatedly sequentially fetch $J_{mixbytes}$ bytes from the full dataset and use the E_{FNV} function to combine it with the mix. $J_{mixbytes}$ bytes of sequential access are used so that each round of the algorithm always fetches a full page from RAM, minimizing translation lookaside buffer misses which ASICs would theoretically be able to avoid.

If the output of this algorithm is below the desired target, then the nonce is valid. Note that the extra application of $\mathtt{KEC}$ at the end ensures that there exists an intermediate nonce which can be provided to prove that at least a small amount of work was done; this quick outer PoW verification can be used for anti-DDoS purposes. It also serves to provide statistical assurance that the result is an unbiased, 256 bit number.

The PoW-function returns an array with the compressed mix as its first item and the Keccak-256 hash of the concatenation of the compressed mix with the seed hash as the second item:

$$\mathtt{PoW}(H_{\not n}, H_n, \mathbf{d}) = \{\mathbf{m}_c(\mathtt{KEC}(\mathtt{RLP}(L_H(H_{\not n}))), H_n, \mathbf{d}), \mathtt{KEC}(\mathbf{s}_h(\mathtt{KEC}(\mathtt{RLP}(L_H(H_{\not n}))), H_n) + \mathbf{m}_c(\mathtt{KEC}(\mathtt{RLP}(L_H(H_{\not n}))), H_n, \mathbf{d}))\} \tag{253}$$

With $H_{\not n}$ being the hash of the header without the nonce. The compressed mix $\mathbf{m}_c$ is obtained as follows:

$$\mathbf{m}_c(\mathbf{h}, \mathbf{n}, \mathbf{d}) = E_{compress}(E_{accesses}(\mathbf{d}, \sum_{i=0}^{n_{mix}} \mathbf{s}_h(\mathbf{h}, \mathbf{n}), \mathbf{s}_h(\mathbf{h}, \mathbf{n}), -1), -4) \tag{254}$$

The seed hash being:

$$\mathbf{s}_h(\mathbf{h}, \mathbf{n}) = \mathtt{KEC512}(\mathbf{h} + E_{revert}(\mathbf{n})) \tag{255}$$

$E_{revert}(\mathbf{n})$ returns the reverted bytes sequence of the nonce $\mathbf{n}$:

$$E_{revert}(\mathbf{n})[i] = \mathbf{n}[\|\mathbf{n}\| - i] \tag{256}$$

We note that the "+"-operator between two byte sequences results in the concatenation of both sequences.

The dataset $\mathbf{d}$ is obtained as described in section J.3.3.

The number of replicated sequences in the mix is:

$$n_{mix} = \left\lfloor \frac{J_{mixbytes}}{J_{hashbytes}} \right\rfloor \tag{257}$$

In order to add random dataset nodes to the mix, the $E_{accesses}$ function is used:

$$E_{accesses}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i) = \begin{cases} E_{mixdataset}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i) & \text{if} \quad i = J_{accesses} - 2 \\ E_{accesses}(E_{mixdataset}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i), \mathbf{s}, i + 1) & \text{otherwise} \end{cases} \tag{258}$$

$$E_{mixdataset}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i) = E_{\mathsf{FNV}}(\mathbf{m}, E_{newdata}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i)) \tag{259}$$

$E_{newdata}$ returns an array with n_{mix} elements:

$$E_{newdata}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i)[j] = \mathbf{d}[E_{\mathsf{FNV}}(i \oplus \mathbf{s}[0], \mathbf{m}[i \mod \left\lfloor \frac{J_{mixbytes}}{J_{wordbytes}} \right\rfloor]) \mod \left\lfloor \frac{d_{size}/J_{hashbytes}}{n_{mix}} \right\rfloor \cdot n_{mix} + j] \quad \forall \quad j < n_{mix} \tag{260}$$

The mix is compressed as follows:

$$E_{compress}(\mathbf{m}, i) = \begin{cases} \mathbf{m} & \text{if} \quad i \geqslant \|\mathbf{m}\| - 8 \\ E_{compress}(E_{\mathsf{FNV}}(E_{\mathsf{FNV}}(E_{\mathsf{FNV}}(\mathbf{m}[i+4], \mathbf{m}[i+5]), \mathbf{m}[i+6]), \mathbf{m}[i+7]), i + 8) & \text{otherwise} \end{cases} \tag{261}$$

Appendix K. IMPORTANT: Signature Verification And Message Encryption

Digital signature is a process ensuring that a certain package was generated by its developers and has not been tampered with. Below we explain why it is important and how to verify that the document as well as some other software you download is the one we have created and has not been modified by some attacker. Digital signature is a cryptographic mechanism. If you want to learn more about how it works see https://en.wikipedia.org/wiki/Digital_signature.

K.1. **What Is A Signature And Why Should I Check It?** How do you know that the document or software you download is really the one we made? Digital signatures ensure that the package you are downloading was created by our developers. It uses a cryptographic mechanism to ensure that the software package that you have just downloaded is authentic. For every user it is a must to verify that the software is authentic as they have very real adversaries who might try to give them a fake version. If the software package as well as the document has been modified by some attacker it is not safe to use. It doesn't matter how secure our package is if you're not running the real one. Before you go ahead and download something, there are a few extra steps you should take to make sure you have downloaded an authentic version. Below is an example of how to verify whether the DOWNLOADED_FILE is the one that we produce: 1.Import the public key of trust-tech.org(trust-tech@protonmail.com):

```
gpg --keyserver pgp.mit.edu --recv-keys 4AAB89C770CF38E68C9D41BE6AEBE3BCE00AA58D
```

2.After importing the key, you can verify that the fingerprint is correct:

```
gpg --fingerprint 4AAB89C770CF38E68C9D41BE6AEBE3BCE00AA58D
```

3.You Should See:

```
pub   rsa4096 2017-06-13 [SC]
      4AAB 89C7 70CF 38E6 8C9D  41BE 6AEB E3BC E00A A58D
uid           [ultimate] trust-tech <trust-tech@protonmail.com>
sub   rsa4096 2017-06-13 [E]
```

4.To verify the signature of the file you downloaded, you will need to download the ".asc" file as well. Assuming the file is at the current directory, run:

```
gpg --verify DOWNLOADED_FILE.asc DOWNLOADED_FILE
```

5.The output should say "Good signature":

```
gpg: Signature made Wed 23 Aug 2017 04:22:39 PM CST
gpg:                using RSA key 6AEBE3BCE00AA58D
gpg: Good signature from "trust-tech <trust-tech@protonmail.com>" [ultimate]
```

K.2. **How To Use GPG to Encrypt and Sign Messages?** It is recommended that all the important messages sent between you and us are encrypted with gpg and assuming you have properly generated gpg key pairs and well configured and secured. 1.Import the public key of trust-tech.org(trust-tech@protonmail.com):

```
gpg --keyserver pgp.mit.edu --recv-keys 4AAB89C770CF38E68C9D41BE6AEBE3BCE00AA58D
```

2.After importing the key, you can verify that the fingerprint is correct:

```
gpg --fingerprint 4AAB89C770CF38E68C9D41BE6AEBE3BCE00AA58D
```

3.You Should See:

```
pub   rsa4096 2017-06-13 [SC]
      4AAB 89C7 70CF 38E6 8C9D  41BE 6AEB E3BC E00A A58D
uid           [ultimate] trust-tech <trust-tech@protonmail.com>
sub   rsa4096 2017-06-13 [E]
```

4.To encrypt the messages that you wish to send to us:

```
gpg --output DOCUMENT_FILE.gpg --encrypt --recipient trust-tech@protonmail.com DOCUMENT_FILE
```

5.To decrypt the messages that we sent to you:

```
gpg --output DOCUMENT_FILE --decrypt DOCUMENT_FILE.gpg
```

For More detail, please refer to the guidelines described in the The GNU Privacy Handbook:
https://www.gnupg.org/gph/en/manual/book1.html

K.3. **Trust-Tech Public Key.** Below is the official pubkey of Trust-Tech:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBFk/TqoBEADiv/06kpQFjF4uGMsxgNGXEHjbh9CdJ5qfCdDk+FyRFcjTKLeb
Gl4x/AqCsokOWHRy8t1xUjQHTn2m+6ATRx3rn/MrMMAEz9slX2MLAbEwzxV2+wkx
vu9k5FylthmhouLObNjRRBjAOCBKVKgLWbII1m4T6qHiTIpkaOhSJdr7phheaxPO
SLyBnlwmUsYC79E8rZw2hWIoTFkQUJw++1egiy0qBb8Z+ezSCezwKfEKCIkz3H9O
/8lsC/Wy9R4ol6L4Wcof5p8PVYYPEwaRRNAGVC+6ODEjC8X7uYSXCAaRRGzVWJEt
yqMnDJf9jYFfhOf/BwwU7biBrYVdOnwYrEm+sXk5osN2WxpEVDJQMEdHm9JlDH2F
Q0UlY2pfvJNWs8VyGzj4zwLT1qA9Ju2bxG4UOOwSpQcqNWEU/2J13k4ikCc1q6+o
cYKhqPhQnYrIgL1+HvU1VZ/IDbDQZBTktZa7DENuRHmOwN2VB5SjgdZOcksf5oWJ
HOjSiPTQEz518z0s1nPslOeuup5PnBfBn04bnXxAhoG1rjvsyyNrBgFBJ5835dk/
PhzY772e2keMWGaygIZsN5BEExOY3W8h04RCMXlxeAY56P9cP6ORnYsHC5AN3f5M
11nvIr5Aqxy8k3h4WScTJFk7Ce0lLDDiXUIBQvd27I5UgXwWlkCjne4biQARAQAB
tCZ0cnVzdC10ZWNoIDx0cnVzdC10ZWNoQHByb3Rvbm1haWwuY29tPokCTgQTAQgA
OBYhBEqricdwzzjmjJ1Bvmrr47zgCqWNBQJZP06qAhsDBQsJCAcCBhUICQoLAgQW
AgMBAh4BAheAAAoJEGrr47zgCqWNmsMP/iENF/4mbK5z1IE4Lkh/AxSiZ1aETVMf
2IZBuYlPOqj1WzPeR9tKJ2/9fyglzXalGHhimSFhTYe8KBJIvzqmGOZgG9kuxhTs
c5nCLZUEZX75flRRpwVLMTAagibm+8h5CbLCmaO2Y35K5awBoxO/BC2Xkpgdrp2S
htixyznKl+H4UGdFVAnI5HafzojTNmZhnmqR5cN9hI7VNm+GtdiiqPX08/4a69vF
PpAXiQLpaq0AwKxpGoY6sBgNHmSQ6+heXxmU2nBOjREe6cUfHOMWT2W1TyZZuOI1
Fcd7FkVt0KolCbJUMEXTbtQuaEQA4cGuBeP6Edlkdkwo2brngJ7qv0ijoZ4aWViB
Y4HawDpv+ZUFOeiRr6sxMwBpcKJ13hyWUAN+wY4vynhU8kEWf+qydwOOObrIgSD8
WDulXcWHR47hGv0d/8sEgKS2vJA11ZtZWDTcKIXfP/AvJYUp+ympw6/+dsyAhh1m
U+L77zj6kE1ddIHWTz/s0BUdZVm9vqy9XGgxvWW88PLd7DVxzLKGe9JkUzOQPxRH
XXOCo2pkjiWsAebOOEx825zw3G0a5CS/O2l5LMwrX0icd9I5iHtcQiEvvaf+uctd
1SZrBTKFQqhULdZTKBNNGbyaB2DPa85Gf0hYxkYniB1VQBdRai9Y9yvt1ps1OFg+
HkIQPp9mAlzTuQINBFk/TqoBEAC4/XoMnT1pEfEkCLOMOKmDy9WiMBVp1ZBRjzZS
r+rE1Sq+8mVSX1CVvsSBdO6FcobjIG2Sb9kaxJLvuX7+14IfoSyIUj96q+7wicCw
X956LumOXsLig3r2z3IGxg9u5kYpfRWKYvLFZWXmezB4nIZGqFEuO/HxDjXA7doz
mhWRwiU+1lmHT2oVrS6j3PojtGBbuXu2Y6pFmYghpCu1hGt/s9yojU7++/KPtu/U
hnP3A+JjGXDHcotw19afgX8Ngs3WCz/ghKfA6aiYEVVDPNB5m5U/PbvyewYIeitR
i0sIw/UltOPGflN4wAs0VGnPHxdYTyKnvXsxChNabv4L05IdxIdzAYu+/mX4SA3v
GVJ/bPL6Cu4fCN6x6rIzlpPuoDHVxEjzY+xsl96glxNRTupOiontkpcT62gIw0sr
BRAEBq4xAW9HhIAcir76xvsNXU1qq6T43PA+r7Xou9ym83oN99YI1LfnjO8jk7Ao
3FSrzhIRsRAvAG5S1CsVYlkoQj+rWQw4rBhboMmCcQpOaF92IwLYkdJKGmUkBoBy
P6T9fllZiWTmS1E6+1ZHWhQ3KiCCZQMmMFmwyIbnYlakNlesr/+IFUrMcU+53K2w
Qa/22iSRHwyjhbXoiNe8fIdg3w4rHucg3ihG4eZGRjyOORy4F7ZZ3uQOZIJRYoxp
dV9vEQARAQABiQI2BBgBCAAgFiEESquJx3DPOOaMnUG+auvjvOAKpYOFAlk/TqoC
GwwACgkQauvjvOAKpYOrrRAAjllxUmNvxWcNBBIKps7eqlNoqLaBow5UYydpovq0
vEJ/FigxS+at14HCjyltvSb/4E4GI4Kl5oadHHk+H4v6NnJpPqhCxEjMFZl7MMmr
20YY8Ho1LYKBY+OxkFccN9LOqwIHkNwWcpupdybdT7ZDLYOZHm2YG3DRpjcbGegp
sok0Ew3utqG2rOXYdStH64+t5wFE0JS2KHvG1QX4sKpMchKloGYpnIt2qrPzZC44
kcbSgu3dkoVurCCPRF12Ej2gDzzr5DkH3yq36SfRsEGfJ5IGlwGPR05AM2ytsKM2
GkLzRwbKfH0px+9iYQCnbfefr0ew+whqoCY/pBYC0Zp25n0Mdirkn8CegLGwMPRz
pQdr8YCd7Cmmb9cXXx/HyvnvsHnySzOyvV1iBPGnJz5q5b6sneQiovi3vMhOw2SD
dAMhn7o4loigT3KRO7+mjyYF372F8mXMdY8lxn1hmNewJVy4U2+BYVONzLqxRDaw
qfpolp/TdWyOMINZSyhIuHK+KF4BbLWjKklw5mfAcVjGPJuAQGNh9XA0LXYktsAb
5FTlAIVSbt8bk8faZde/lzBbhzTHO6BLY57GdmgerNu3Rced6K8+qmywG5CgupY2
5XscysLOfRbhW2Oy8Wd6tySv2meo/cl/HBrCgpKMor0fCq4MZeOTC5/krPELWNPc
iBs=
=nAME
-----END PGP PUBLIC KEY BLOCK-----
```