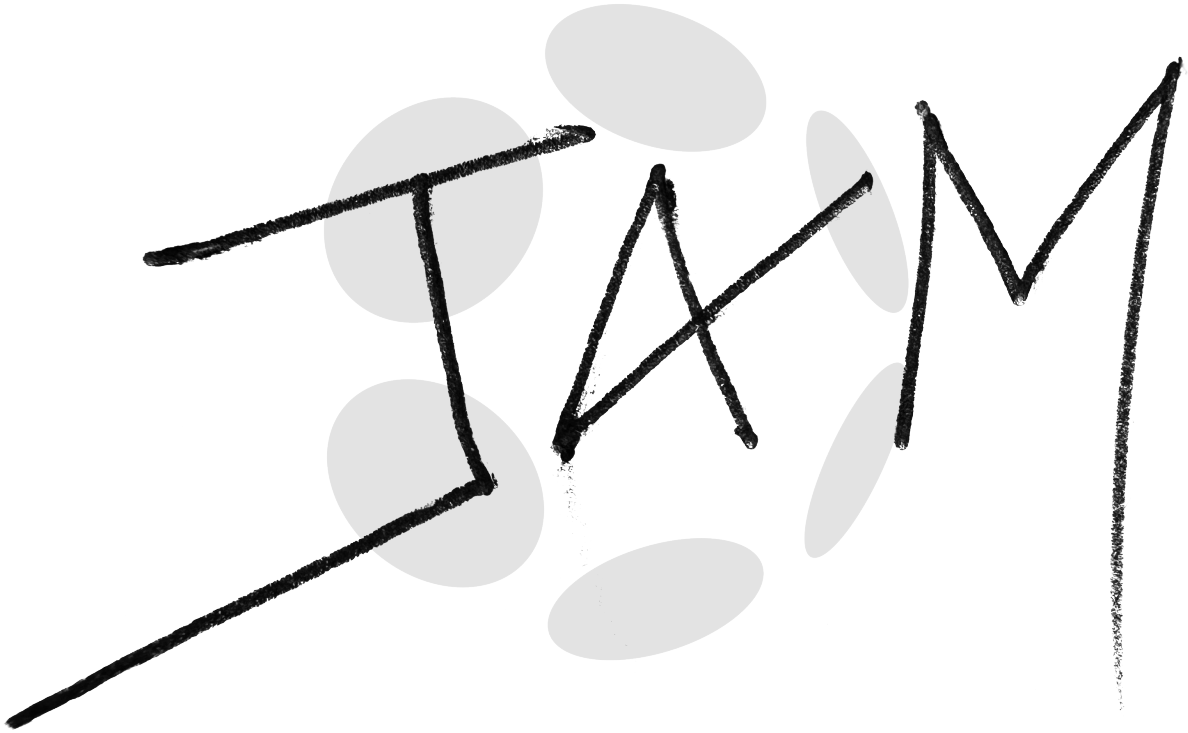

JOIN-ACCUMULATE MACHINE: A SEMI-COHERENT SCALABLE TRUSTLESS VM

DRAFT 0.1.1 - April 19, 2024

DR. GAVIN WOOD
FOUNDER, POLKADOT & ETHEREUM
GAVIN@PARITY.IO

Abstract. We present a comprehensive and formal definition of JAM, a protocol combining elements of both *Polkadot* and *Ethereum*. In a single coherent model, JAM provides a global singleton permissionless object environment—much like the smart-contract environment pioneered by Ethereum—paired with secure sideband computation parallelized over a scalable node network, a proposition pioneered by Polkadot.

JAM introduces a decentralized hybrid system offering smart-contract functionality structured around a secure and scalable in-core/on-chain dualism. While the smart-contract functionality implies some similarities with Ethereum's paradigm, the overall model of the service offered is driven largely by underlying architecture of Polkadot.

JAM is permissionless in nature, allowing anyone to deploy code as a service on it for a fee commensurate with the resources this code utilizes and to induce execution of this code through the procurement and allocation of *core-time*, a metric of resilient and ubiquitous computation, somewhat similar to the purchasing of gas in Ethereum. We already envision a Polkadot-compatible *CoreChains* service.

1. Introduction

1.1. **Nomenclature.** In this paper, we introduce a decentralized, crypto-economic protocol to which the Polkadot Network could conceivably transition itself in a major revision. Following this eventuality (which must not be taken for granted since Polkadot is a decentralized network) this protocol might also become known as *Polkadot* or some derivation thereof. However, at this stage this is not the case, therefore our proposed protocol will for the present be known as JAM.

An early, unrefined, version of this protocol was first proposed in Polkadot Fellowship RFC31, known as *CoreJam*. CoreJam takes its name after the collect/refine/join/accumulate model of computation at the heart of its service proposition. While the CoreJam RFC suggested an incomplete, scope-limited alteration to the Polkadot protocol, JAM refers to a complete and coherent overall blockchain protocol.

1.2. **Driving Factors.** Within the realm of blockchain and the wider Web3, we are driven by the need first and foremost to deliver resilience. A proper Web3 digital system should honor a declared service profile—and ideally meet even perceived expectations—regardless of the desires, wealth or power of any economic actors including individuals, organizations and, indeed, other Web3 systems. Inevitably this is aspirational, and we must be pragmatic over how perfectly this may really be delivered. Nonetheless, a Web3 system should aim to provide such radically strong guarantees that, for practical purposes, the system may be described as *unstoppable*.

While Bitcoin is, perhaps, the first example of such a system within the economic domain, it was not general purpose in terms of the nature of the service it offered. A rules-based service is only as useful as the generality of the rules which may be conceived and placed within it. Bitcoin's rules allowed for an initial use-case, namely a fixed-issuance token, ownership of which is well-approximated and autonomously enforced through knowledge of a secret, as well as some further elaborations on this theme.

Later, Ethereum would provide a categorically more general-purpose rule set, one which was practically Turing complete.[1] In the context of Web3 where we are

[1]The gas mechanism did restrict what programs can execute on it by placing an upper bound on the number of steps which may be executed, but some restriction to avoid infinite-computation must surely be introduced in a permissionless setting.

aiming to deliver a massively multiuser application platform, generality is crucial, and thus we take this as a given.

Beyond resilience and generality, things get more interesting, and we must look a little deeper to understand what our driving factors are. For the present purposes, we identify three additional goals:

(1) Resilience: highly resistant from being stopped, corrupted and censored.

(2) Generality: able to perform Turing-complete computation.

(3) Performance: able to perform computation quickly and at low cost.

(4) Coherency: the causal relationship possible between different elements of state and how thus how well individual applications may be composed.

(5) Accessibility: negligible barriers to innovation; easy, fast, cheap and permissionless.

As a declared Web3 technology, we make an implicit assumption of the first two items. Interestingly, items 3 and 4 are antagonistic according to an information theoretic principle which we are sure must already exist in some form but are nonetheless unaware of a name for it. For argument's sake we shall name it *size-synchrony antagonism*.

1.3. **Scaling under Size-Synchrony Antagonism.** Size-synchrony antagonism is a simple principle implying that as the state-space of information systems grow, then the system necessarily becomes less synchronous. The argument goes:

(1) The more state a system utilizes for its data-processing, the greater the amount of space this state must occupy.

(2) The more space used, then the greater the mean and variance of distances between state-components.

(3) As the mean and variance increase, then interactions become slower and subsystems must manage the possibility that distances between interdependent components of state could be materially different, requiring asynchrony.

This assumes perfect coherency of the system's state. Setting the question of overall security aside for a moment, we can avoid this rule by applying the *divide and conquer* maxim and fragmenting the state of a system, sacrificing its coherency. We might for example create two independent smaller-state systems rather than one large-state system. This pattern applies a step-curve to the principle; intra-system processing has low size and high synchrony, inter-system processing has high size but low synchrony. It is the principle behind meta-networks such as Polkadot, Cosmos and the predominant vision of a scaled Ethereum (all to be discussed in depth shortly).

The present work explores a middle-ground in the antagonism, avoiding the persistent fragmentation of state-space of the system as with existing approaches. We do this by introducing a new model of computation which pipelines a highly scalable element to a highly synchronous element. Asynchrony is not avoided, but we do open the possibility for a greater degree of granularity over how it is traded against size. In particular fragmentation can be made ephemeral rather than persistent, drawing upon a coherent state and fragmenting it only for as long as it takes to execute any given piece of processing on it.

Unlike with SNARK-based L2-blockchain techniques for scaling, this model draws upon crypto-economic mechanisms and inherits their low-cost and high-performance profiles and averts a bias toward centralization.

1.4. **Document Structure.** We begin with a brief overview of present scaling approaches in blockchain technology in section 2. In section 3 we define and clarify the notation from which we will draw for our formalisms.

We follow with a broad overview of the protocol in section 4 outlining the major areas including the Polka Virtual Machine (PVM), the consensus protocols Safrole and GRANDPA, the common clock and build the foundations of the formalism.

We then continue with the full protocol definition split into two parts: firstly the correct on-chain state-transition formula helpful for all nodes wishing to validator the chain state, and secondly, in sections 13 and 15 the honest strategy for the off-chain actions of any actors who wield a validator key.

The main body ends with a discussion over the performance characteristics of the protocol in section 17 and finally conclude in section 18.

The appendix contains various additional material important for the protocol definition including the PVM in appendices A & B, serialization and Merklization

in appendices C & D and cryptography in appendices F, G & H. We finish with an index of terms which includes the values of all simple constant terms used in the work in appendix I, and close with the bibliography.

2. Previous Work and Present Trends

In the years since the initial publication of the Ethereum *YP*, the field of blockchain development has grown immensely. Other than scalability, development has been done around underlying consensus algorithms, smart-contract languages and machines and overall state environments. While interesting, these latter subjects are mostly out scope of the present work since they generally do not impact underlying scalability.

2.1. **Polkadot.**

In order to deliver its service, JAM co-opts much of the same game-theoretic and cryptographic machinery as Polkadot known as ELVES and described by Stewart 2018. However, major differences exist in the actual service offered with JAM, providing an abstraction much closer to the actual computation model generated by the validator nodes its economy incentivizes.

It was a major point of the original Polkadot proposal, a scalable heterogeneous multichain, to deliver high-performance through partition and distribution of the workload over multiple host machines. In doing so it took an explicit position that composability would be lowered. Polkadot's constituent components, parachains are, practically speaking, highly isolated in their nature. Though a message passing system (XCMP) exists it is asynchronous, coarse-grained and practically limited by its reliance on a high-level slowly evolving interaction language XCM.

As such, the composability offered by Polkadot between its constituent chains is lower than that of Ethereum-like smart-contract systems offering a single and universal object environment and allowing for the kind of agile and innovative integration which underpins their success. Polkadot, as it stands, is a collection of independent ecosystems with only limited opportunity for collaboration, very similar in ergonomics to bridged blockchains though with a categorically different security profile. A technical proposal known as SPREE would utilize Polkadot's unique shared-security and improve composability, though blockchains would still remain isolated.

Implementing and launching a blockchain is hard, time-consuming and costly. By its original design, Polkadot limits the clients able to utilize its service to those

who are both able to do this and raise a sufficient deposit to win an auction for a long-term slot, one of around 50 at the present time. While not permissioned per se, accessibility is categorically and substantially lower than for smart-contract systems similar to Ethereum.

Enabling as many innovators to participate and interact, both with each other and each other's user-base, appears to be an important component of success for a Web3 application platform. Accessibility is therefore crucial.

2.2. **Ethereum.** The Ethereum protocol was formally defined in this paper's spiritual predecessor, the *Yellow Paper*, by Wood 2014. This was derived in large part from the initial concept paper by Buterin 2013. In the decade since the *YP* was published, the *de facto* Ethereum protocol and public network instance have gone through a number of evolutions, primarily structured around introducing flexibility via the transaction format and the instruction set and "precompiles" (niche, sophisticated bonus instructions) of its scripting core, the Ethereum virtual machine (EVM).

Almost one million crypto-economic actors take part in the validation for Ethereum.[2] Block extension is done through a randomized leader-rotation method where the physical address of the leader is public in advance of their block production.[3] Ethereum uses Casper-FFG introduced by Buterin and Griffith 2019 to determine finality, which with the large validator base finalizes the chain extension around every 13 minutes.

Ethereum's direct computational performance remains broadly similar to that with which it launched in 2015, with a notable exception that an additional service now allows 1MB of *commitment data* to be hosted per block (all nodes to store it for a limited period). The data cannot be directly utilized by the main state-transition function, but special functions provide proof that the data (or some subsection thereof) is available. According to Ethereum Foundation 2024b, the present design direction is to improve on this over the coming years by splitting

[2]Practical matters do limit the level of real decentralization. Validator software expressly provides functionality to allow a single instance to be configured with multiple key sets, systematically facilitating a much lower level of actual decentralization than the apparent number of actors, both in terms of individual operators and hardware. Using data collated by Dune and hildobby 2024 on Ethereum 2, one can see one major node operator, Lido, has steadily accounted for almost one-third of the almost one million crypto-economic participants.

[3]Ethereum's developers hope to change this to something more secure, but no timeline is fixed.

responsibility for its storage amongst the validator base in a protocol known as *Dank-sharding*.

According to Ethereum Foundation 2024a, the scaling strategy of Ethereum would be to couple this data availability with a private market of *roll-ups*, sideband computation facilities of various design, with ZK-SNARK-based roll-ups being a stated preference. Each vendor's roll-up design, execution and operation comes with its own implications.

One might reasonably assume that a diversified market-based approach for scaling via multivendor roll-ups will allow well-designed solutions to thrive. However, there are potential issues facing the strategy. A research report by Sharma 2023 on the level of decentralization in the various roll-ups found a broad pattern of centralization, but notes that work is underway to attempt to mitigate this. It remains to be seen how decentralized they can yet be made.

Heterogeneous communication properties (such as datagram latency and semantic range), security properties (such as the costs for reversion, corruption, stalling and censorship) and economic properties (the cost of accepting and processing some incoming message or transaction) may differ, potentially quite dramatically, between major areas of some grand patchwork of roll-ups by various competing vendors. While the overall Ethereum network may eventually provide some or even most of the underlying machinery needed to do the sideband computation it is far from clear that there would be a "grand consolidation" of the various properties should such a thing happen. We have not found any good discussion of the negative ramifications of such a fragmented approach.[4]

2.2.1. *SNARK Roll-ups.* While the protocol's foundation makes no great presuppositions on the nature of roll-ups, Ethereum's strategy for sideband computation does centre around SNARK-based rollups and as such the protocol is being evolved into a design that makes sense for this. SNARKs are the product of an area of exotic cryptography which allow proofs to be constructed to demonstrate to a neutral observer that the purported result of performing some predefined computation is correct. The complexity of the verification of these proofs tends to be sub-linear in their size of computation to be proven and will not give away any of the internals of said computation, nor any dependent witness data on which it may rely.

[4]Some initial thoughts on the matter resulted in a proposal by Sadana 2024 to utilize Polkadot technology as a means of helping create a modicum of compatibility between roll-up ecosystems!

ZK-SNARKs come with constraints. There is a trade-off between the proof's size, verification complexity and the computational complexity of generating it. Non-trivial computation, and especially the sort of general-purpose computation laden with binary manipulation which makes smart-contracts so appealing, is hard to fit into the model of SNARKs.

To give a practical example, RISC-zero (as assessed by Bögli 2024) is a leading project and provides a platform for producing SNARKs of computation done by a RISC-V virtual machine, an open-source and succinct RISC machine architecture well-supported by tooling. A recent benchmarking report by PolkaVM Project 2024 showed that compared to RISC-zero's own benchmark, proof generation alone takes over 61,000 times as long as simply recompiling and executing even when executing on 32 times as many cores, using 20,000 times as much RAM and an additional state-of-the-art GPU. According to hardware rental agents `https://cloud-gpus.com/`, the cost multiplier of proving using RISC-zero is 66,000,000x of the cost[5] to execute using our RISC-V recompiler.

Many cryptographic primitives become too expensive to be practical to use and specialized algorithms and structures must be substituted. Often times they are otherwise suboptimal. In expectation of the use of SNARKs (such as PLONK as proposed by Gabizon, Williamson, and Ciobotaru 2019), the prevailing design of the Ethereum project's Dank-sharding availability system uses a form of erasure coding centered around polynomial commitments over a large prime field in order to allow SNARKs to get acceptably performant access to subsections of data. Compared to alternatives, such as a binary field and Merklization in the present work, it leads to a load on the validator nodes orders of magnitude higher in terms of CPU usage.

In addition to their basic cost, SNARKs present no great escape from decentralization and the need for redundancy, leading to further cost multiples. While the need for some benefits of staked decentralization is averted through their verifiable nature, the need to incentivize multiple parties to do much the same work is a requirement to ensure that a single party not form a monopoly (or several not form a cartel). Proving an incorrect state-transition should be impossible, however service integrity may be compromised in other ways; a temporary suspension

[5] In all likelihood actually substantially more as this was using low-tier "spare" hardware in consumer units, and our recompiler was unoptimized.

of proof-generation, even if only for minutes, could amount to major economic ramifications for real-time financial applications.

Real-world examples exist of the pit of centralization giving rise to monopolies. One would be the aforementioned SNARK-based exchange framework; while notionally serving decentralized exchanges, it is in fact centralized with Starkware itself wielding a monopoly over enacting trades through the generation and submission of proofs, leading to a single point of failure—should Starkware's service become compromised, then the liveness of the system would suffer.

It has yet to be demonstrated that SNARK-based strategies for eliminating the trust from computation will ever be able to compete on a cost-basis with a multi-party crypto-economic platform. All as-yet proposed SNARK-based solutions are heavily reliant on crypto-economic systems to frame them and work around their issues. Data availability and sequencing are two areas well understood as requiring a crypto-economic solution.

We would note that SNARK technology is improving and the cryptographers and engineers behind them do expect improvements in the coming years. In a recent article by Thaler 2023 we see some credible speculation that with some recent advancements in cryptographic techniques, slowdowns for proof generation could be as little as 50,000x from regular native execution and much of this could be parallelized. This is substantially better than the present situation, but still several orders of magnitude greater than would be required to compete on a cost-basis with established crypto-economic techniques such as ELVES.

2.3. **Fragmented Meta-Networks.** Directions for general-purpose computation scalability taken by other projects broadly centre around one of two approaches; either what might be termed a *fragmentation* approach or alternatively a *centralization* approach. We argue that neither approach offers a compelling solution.

The fragmentation approach is heralded by projects such as Cosmos (proposed by Kwon and Buchman 2019) and Avalanche (by Tanana 2019). It involves a system fragmented by networks of a homogenous consensus mechanic, yet staffed by separately motivated sets of validators. This is in contrast to Polkadot's single validator set and Ethereum's declared strategy of heterogeneous roll-ups secured partially by the same validator set operating under a coherent incentive framework. The homogeneity of said fragmentation approach allows for reasonably consistent

messaging mechanics, helping to present a fairly unified interface to the multitude of connected networks.

However, the apparent consistency is superficial. The networks are trustless only by assuming correct operation of their validators, who operate under a crypto-economic security framework ultimately conjured and enforced by economic incentives and punishments. To do twice as much work with the same levels of security and no special coordination between validator sets, then such systems essentially prescribe forming a new network with the same overall levels of incentivization.

Several problems arise. Firstly, there is a similar downside as with Polkadot's isolated parachains and Ethereum's isolated roll-up chains: a lack of coherency due to a persistently sharded state preventing synchronous composability.

More problematically, the scaling-by-fragmentation approach, proposed specifically by Cosmos, provides no homogenous security—and therefore trustlessness—guarantees. Validator sets between networks must be assumed to be independently selected and incentivized with no relationship, causal or probabilistic, between the Byzantine actions of a party on one network and potential for appropriate repercussions on another. Essentially, this means that should validators conspire to corrupt or revert the state of one network, the effects may be felt across other networks of the ecosystem.

That this is an issue is broadly accepted, and projects propose for it to be addressed in one of two ways. Firstly, to fix the expected cost-of-attack (and thus level of security) across networks by drawing from the same validator set. The massively redundant way of doing this, as proposed by Cosmos Project 2023 under the name *replicated security*, would be to require each validator to validate on all networks and for the same incentives and punishments. This is economically inefficient in the cost of security provision as each network would need to independently provide the same level of incentives and punishment-requirements as the most secure with which it wanted to interoperate. This is to ensure the economic proposition remain unchanged for validators and the security proposition remained equivalent for all networks. At the present time, replicated security is not a readily available permissionless service. We might speculate that these punishing economics have something to do with it.

The more efficient approach, proposed by the OmniLedger team, Kokoris-Kogias et al. 2017, would be to make the validators non-redundant, partitioning

them between different networks and periodically, securely and randomly repartitioning them. A reduction in the cost to attack over having them all validate on a single network is implied since there is a chance of having a single network accidentally have a compromising number of malicious validators even with less than this proportion overall. This aside it presents an effective means of scaling under a basis of weak-coherency.

Alternatively, as in Elves by Stewart 2018, we may utilize non-redundant partitioning, combine this with a proposal-and-auditing game which validators play to weed out and punish invalid computations, and then require that the finality of one network be contingent on all causally-entangled networks. This is the most secure and economically efficient solution of the three, since there is a mechanism for being highly confident that invalid transitions will be recognized and corrected before their effect is finalized across the ecosystem of networks. However, it requires substantially more sophisticated logic and their causal-entanglement implies some upper limit on the number of networks which may be added.

2.4. **High-Performance Fully Synchronous Networks.** Another trend in the recent years of blockchain development has been to make "tactical" optimizations over data throughput by limiting the validator set size or diversity, focusing on software optimizations, requiring a higher degree of coherency between validators, onerous requirements on the hardware which validators must have, or limiting data availability.

The Solana blockchain is underpinned by technology introduced by Yakovenko 2018 and boasts theoretical figures of over 700,000 transactions per second, though according to Ng 2024 the network is only seen processing a small fraction of this. The underlying throughput is still substantially more than most blockchain networks and is owed to various engineering optimizations in favor of maximizing synchronous performance. The result is a highly-coherent smart-contract environment with an API not unlike that of *YP* Ethereum (albeit using a different underlying VM), but with a near-instant time to inclusion and finality which is taken to be immediate upon inclusion.

Two issues arise with such an approach: firstly, defining the protocol as the outcome of a heavily optimized codebase creates structural centralization and can undermine resilience. Jha 2024 writes "since January 2022, 11 significant outages gave rise to 15 days in which major or partial outages were experienced". This is

an outlier within the major blockchains as the vast majority of major chains have no downtime. There are various causes to this downtime, but they are generally due to bugs found in various subsystems.

Ethereum, at least until recently, provided the most contrasting alternative with its well-reviewed specification, clear research over its crypto-economic foundations and multiple clean-room implementations. It is perhaps no surprise that the network very notably continued largely unabated when a flaw in its most deployed implementation was found and maliciously exploited, as described by Hertig 2016.

The second issue is concerning ultimate scalability of the protocol when it provides no means of distributing workload beyond the hardware of a single machine.

In major usage, both historical transaction data and state would grow impractically. Solana illustrates how much of a problem this can be. Unlike classical blockchains, the Solana protocol offers no solution for the archival and subsequent review of historical data, crucial if the present state is to be proven correct from first principle by a third party. There is little information on how Solana manages this in the literature, but according to Solana Foundation 2023, nodes simply place the data onto a centralized database hosted by Google.[6]

Solana validators are encouraged to install large amounts of RAM to help hold its large state in memory (512 GB is the current recommendation according to Solana Labs 2024). Without a divide-and-conquer approach, Solana shows that the level of hardware which validators can reasonably be expected to provide dictates the upper limit on the performance of a totally synchronous, coherent execution model. Hardware requirements represent barriers to entry for the validator set and cannot grow without sacrificing decentralization and, ultimately, transparency.

3. Notational Conventions

Much as in the Ethereum Yellow Paper, a number of notational conventions are used throughout the present work. We define them here for clarity. The Ethereum Yellow Paper itself may be referred to henceforth as the *YP*.

3.1. **Typography.**

We use a number of different typefaces to denote different kinds of terms. Where a term is used to refer to a value only relevant within

[6]Earlier node versions utilized Arweave network, a decentralized data store, but this was found to be unreliable for the data throughput which Solana required.

some localized section of the document, we use a lower-case roman letter e.g. x, y (typically used for an item of a set or sequence) or e.g. i, j (typically used for numerical indices). Where we refer to a Boolean term or a function in a local context, we tend to use a capitalized roman alphabet letter such as A, F. If particular emphasis is needed on the fact a term is sophisticated or multidimensional, then we may use a bold typeface, especially in the case of sequences and sets.

For items which retain their definition throughout the present work, we use other typographic conventions. Sets are usually referred to with a blackboard typeface, e.g. $\mathbb{N}$ refers to all natural numbers including zero. Sets which may be parameterized may be subscripted or be followed by parenthesized arguments. Imported functions, used by the present work but not specifically introduced by it, are written in calligraphic typeface, e.g. $\mathcal{H}$ the Blake2 cryptographic hashing function. For other non-context dependent functions introduced in the present work, we use upper case Greek letters, e.g. Υ denotes the state transition function.

Values which are not fixed but nonetheless hold some consistent meaning throughout the present work are denoted with lower case Greek letters such as σ, the state identifier. These may be placed in bold typeface to denote that they refer to an abnormally complex value.

3.2. **Functions and Operators.** We define the precedes relation to indicate that one term is defined in terms of another. E.g. $y \prec x$ indicates that y may be defined purely in terms of x:

$$y \prec x \iff \exists f : y = f(x) \tag{1}$$

The substitute-if-nothing function $\mathcal{U}$ is equivalent to the first argument which is not $\varnothing$, or $\varnothing$ if no such argument exists:

$$\mathcal{U}(a_0, \dots) \equiv a_x : (a_x \neq \varnothing \vee x = n), \bigwedge_{i=0}^{x-1} a_i = \varnothing \tag{2}$$

Thus, e.g. $\mathcal{U}(\varnothing, 1, \varnothing, 2) = 1$ and $\mathcal{U}(\varnothing, \varnothing) = \varnothing$.

3.3. **Sets.** We denote the cardinality of some set $\mathbf{s}$, the number of its elements, as the usual $|\mathbf{s}|$. We denote set-disjointness with the relation $\perp\!\!\!\circ$. Formally:

$$A \cap B = \varnothing \iff A \perp\!\!\!\circ B$$

We commonly use $\varnothing$ to indicate that some term is validly left without a specific value. Its cardinality is defined as zero. We define the operation **?** such that $A\mathbf{?} \equiv A \cup \{\varnothing\}$ indicating the same set but with the addition of the $\varnothing$ element.

The term ∇ is utilized to indicate the unexpected failure of an operation or that a value is invalid or unexpected. (We try to avoid the use of the more conventional $\perp$ here to avoid confusion with Boolean false, which may be interpreted as some successful result in some contexts.)

3.4. **Numbers.** $\mathbb{N}$ denotes the set of naturals including zero whereas $\mathbb{N}_n$ denotes the set of naturals less than n. Formally, $\mathbb{N} = \{0, 1, \dots\}$ and $\mathbb{N}_n = \{x \mid x \in \mathbb{N}, x < n\}$.

$\mathbb{Z}$ denotes the set of integers. We denote $\mathbb{Z}_{a\dots b}$ to be the set of integers within the interval $[a, b)$. Formally, $\mathbb{Z}_{a\dots b} = \{x \mid x \in \mathbb{Z}, a \le x < b\}$. E.g. $\mathbb{Z}_{2\dots 5} = \{2, 3, 4\}$. We denote the offset/length form of this set as $\mathbb{Z}_{a\cdots+b}$, a short form of $\mathbb{Z}_{a\dots a+b}$.

It can sometimes be useful to represent lengths of sequences and yet limit their size, especially when dealing with sequences of octets which must be stored practically. Typically, these lengths can be defined as the set $\mathbb{N}_{2^{32}}$. To improve clarity, we denote $\mathbb{N}_L$ as the set of lengths of octet sequences and is equivalent to $\mathbb{N}_{2^{32}}$.

We denote the $\%$ operator as the modulo operator, e.g. $5 \% 3 = 2$. Furthermore, we may occasionally express a division result as a quotient and remainder with the separator R, e.g. $5 \div 3 = 1\,\mathsf{R}\,3$.

3.5. **Dictionaries.** A *dictionary* is a possibly partial mapping from some domain into some co-domain in much the same manner as a regular function. Unlike functions however, with dictionaries the total set of pairings are necessarily enumerable, and we represent them in some data structure as the set of all $(key \mapsto value)$ pairs. (In such data-defined mappings, it is common to name the values within the range a *key* and the values within the domain a *value*, hence the naming.)

Thus, we define the formalism $\mathbb{D}\langle \mathrm{K} \to \mathrm{V} \rangle$ to denote a dictionary which maps from the domain K to the range V. We define a dictionary as a member of the set of all dictionaries $\mathbb{D}$ and a set of pairs $p = (k \mapsto v)$:

$$\mathbb{D} \subset \left\{\{(k \mapsto v)\}\right\} \tag{3}$$

A dictionary's members must associate at most one unique value for any key k:

$$\forall \mathbf{d} \in \mathbb{D} : \forall (k \mapsto v) \in \mathbf{d} : \exists! v' : (k \mapsto v') \in \mathbf{d} \tag{4}$$

This assertion allows us to unambiguously define the subscript and subtraction operator for a dictionary d:

$$\forall \mathbf{d} \in \mathbb{D} : \mathbf{d}[k] \equiv \begin{cases} v & \text{if } \exists k : (k \mapsto v) \in \mathbf{d} \\ \varnothing & \text{otherwise} \end{cases} \tag{5}$$

$$\forall \mathbf{d} \in \mathbb{D}, \mathbf{s} : a \setminus \mathbf{s} \equiv \{(k \mapsto v) : (k \mapsto v) \in \mathbf{d}, k \notin \mathbf{s}\} \tag{6}$$

Note that when using a subscript, it is an implicit assertion that the key exists in the dictionary. Should the key not exist, the result is undefined and any block which relies on it must be considered invalid.

It is typically useful to limit the sets from which the keys and values may be drawn. Formally, we define a typed dictionary $\mathbb{D}\langle K \to V \rangle$ as a set of pairs p of the form $(k \mapsto v)$:

$$\mathbb{D}\langle K \to V \rangle \subset \mathbb{D} \tag{7}$$

$$\mathbb{D}\langle K \to V \rangle \equiv \left\{ \{ (k \mapsto v) \mid k \in K \wedge v \in V \} \right\} \tag{8}$$

To denote the active domain (i.e. set of keys) of a dictionary $\mathbf{d} \in \mathbb{D}\langle K \to V \rangle$, we use $\mathcal{K}(\mathbf{d}) \subset K$ and for the range (i.e. set of values), $\mathcal{V}(\mathbf{d}) \subset V$. Formally:

$$\mathcal{K}(\mathbf{d} \in \mathbb{D}) \equiv \{\ k \mid \exists v : (k \mapsto v) \in \mathbf{d}\ \} \tag{9}$$

$$\mathcal{V}(\mathbf{d} \in \mathbb{D}) \equiv \{\ v \mid \exists k : (k \mapsto v) \in \mathbf{d}\ \} \tag{10}$$

Note that since the domain of $\mathcal{V}$ is a set, should different keys with equal values appear in the dictionary, the set will only contain one such value.

3.6. **Tuples.** Tuples are groups of values where each item typically belongs to a different set. They are denoted with parentheses, e.g. the tuple t of the integers 3 and 5 is denoted $t = (3, 5)$, and it exists in the set of integer pairs sometimes denoted $\mathbb{N} \times \mathbb{N}$, but denoted in the present work as $(\mathbb{N}, \mathbb{N})$.

We have frequent need to refer to a specific item within a tuple value and as such find it convenient to declare a name for each item. E.g. we may denote a tuple with two named integer components a and b as $T = (a \in \mathbb{N}, b \in \mathbb{N})$. We would denote an item $t \in T$ through subscripting its name, thus for some $t = (a\colon 3, b\colon 5)$, $t_a = 3$ and $t_b = 5$.

3.7. **Sequences.** A sequence is a series of elements with particular ordering not dependent on their values. The set of sequences of elements all of which are drawn from some set T is denoted $\llbracket T \rrbracket$, and it defines a partial mapping $\mathbb{N} \to T$. The set of sequences containing exactly n elements each a member of the set T may be denoted $\llbracket T \rrbracket_n$ and accordingly defines a complete mapping $\mathbb{N}_n \to T$. Similarly, sets of sequences of at most n elements and at least n elements may be denoted $\llbracket T \rrbracket_{:n}$ and $\llbracket T \rrbracket_{n:}$ respectively.

Sequences are subscriptable, thus a specific item at index i within a sequence $\mathbf{s}$ may be denoted $\mathbf{s}[i]$, or where unambiguous, $\mathbf{s}_i$. A range may be denoted using an ellipsis for example: $[0,1,2,3]_{\dots 2} == [0,1]$ and $[0,1,2,3]_{1\dots+2} == [1,2]$. The length of such a sequence may be denoted $|\mathbf{s}|$.

We denote modulo subscription as $\mathbf{s}[i]^{\circlearrowleft} \equiv \mathbf{s}[\, i \bmod |\mathbf{s}|\,]$. We denote the final element x of a sequence $\mathbf{s} = [\dots, x]$ through the function $\text{last}(\mathbf{s}) \equiv x$.

3.7.1. *Construction.* We may wish to define a sequence in terms of incremental subscripts of other values: $[\mathbf{x}_0, \mathbf{x}_1, \dots]_n$ denotes a sequence of n values beginning $\mathbf{x}_0$ continuing up to $\mathbf{x}_{n-1}$. Furthermore, we may also wish to define a sequence as elements each of which are a function of their index i; in this case we denote $[f(i) \mid i \lessdot \mathbb{N}_n] \equiv [f(0), f(1), \dots, f(n-1)]$. Thus, when the ordering of elements matters we use $\lessdot$ rather than the unordered notation $\in$. The latter may also be written in short form $[f(i \lessdot \mathbb{N}_n)]$. This applies to any set which has an unambiguous ordering, particularly sequences, thus $[\, i^2 \mid i \lessdot [1,2,3]\,] = [1,4,9]$. Multiple sequences may be combined, thus $[\, i \cdot j \mid i \lessdot [1,2,3], j \lessdot [2,3,4]\,] = [2,6,12]$.

Sequences may be constructed from sets or other sequences whose order should be ignored through sequence ordering notation $[i_k \,\rangle\, i \in X]$, which is defined to result in the set or sequence of its argument except that all elements i are placed in ascending order of the corresponding value i_k.

The key component may be elided in which case it is assumed to be ordered by the elements directly; i.e. $[i \in X] \equiv [i \,\rangle\, i \in X]$. $[i_k \,\rangle\!\rangle\, i \in X]$ does the same, but

excludes any duplicate values of i. E.g. assuming $\mathbf{s} = [1, 3, 2, 3]$, then $[i \,|\!|\, i \in \mathbf{s}] = [1, 2, 3]$ and $[-i \,|\!|\, i \in \mathbf{s}] = [3, 3, 2, 1]$.

Sets may be constructed from sequences with the regular set construction syntax, e.g. assuming $\mathbf{s} = [1, 2, 3, 1]$, then $\{a \mid a \in \mathbf{s}\}$ would be equivalent to $\{1, 2, 3\}$.

Sequences of values which themselves have a defined ordering have an implied ordering akin to a regular dictionary, thus $[1, 2, 3] < [1, 2, 4]$ and $[1, 2, 3] < [1, 2, 3, 1]$.

3.7.2. *Editing.* We define the sequence concatenation operator $\frown$ such that $[\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{y}_0, \mathbf{y}_1$ $\mathbf{x} \frown \mathbf{y}$. Futher, we denote element-concatenation as $x +\!\!+ i \equiv x \frown [i]$. We denote the sequence made up of the first n elements of sequence $\mathbf{s}$ to be $\overrightarrow{\mathbf{s}}^n \equiv [\mathbf{s}_0, \mathbf{s}_1, \ldots, \mathbf{s}_{n-1}]$, and only the final elements as $\overleftarrow{\mathbf{s}}^n$.

We denote sequence subtraction with a slight modification of the set subtraction operator; specifically, some sequence $\mathbf{s}$ excepting the left-most element equal to v would be denoted $\mathbf{s} \setminus \{v\}$.

3.7.3. *Boolean values.* $\mathbb{B}_s$ denotes the set of Boolean strings of length s, thus $\mathbb{B}_s = [\{\bot, \top\}]_s$. When dealing with Boolean values we may assume an implicit equivalence mapping to a bit whereby $\top = 1$ and $\bot = 0$, thus $\mathbb{B}_\square = [\mathbb{N}_2]_\square$. We use the function $\text{bits}(\mathbb{Y}) \in \mathbb{B}$ to denote the sequence of bits, ordered with the least significant first, which represent the octet sequence $\mathbb{Y}$, thus $\text{bits}([5, 0]) = [1, 0, 1, 0, 0, \ldots]$.

3.7.4. *Octets and Blobs.* $\mathbb{Y}$ denotes the set of octet strings ("blobs") of arbitrary length. As might be expected, $\mathbb{Y}_x$ denotes the set of such sequences of length x. $\mathbb{Y}_\$$ denotes the subset of $\mathbb{Y}$ which are ASCII-encoded strings. Note that while an octet has an implicit and obvious bijective relationship with natural numbers less than 256, and we may implicitly coerce between octet form and integer form, we do not treat them as exactly equivalent entities. In particular for the purpose of serialization an octet is always serialized in the sequence containing only itself, whereas an integer may be serialized as a sequence of potentially several octets, depending on its magnitude.

3.7.5. *Shuffling.* We define the sequence-shuffle function $\mathcal{F}$, originally introduced by Fisher and Yates 1938, with an efficient in-place algorithm described by Wikipedia 2024. This accepts a sequence and some entropy and returns a sequence of the same length with the same elements but in an order determined by the entropy.

The entropy may be provided as either an indefinite sequence of integers or a hash. For a full definition see appendix E.

3.8. **Cryptography.**

3.8.1. *Hashing.* $\mathbb{H}$ denotes the set of 256-bit values typically expected to be arrived at through a cryptographic function, equivalent to $\mathbb{Y}_{32}$, with $\mathbb{H}^0$ being equal to $[0]_{32}$. We assume a function $\mathcal{H}(m \in \mathbb{Y}) \in \mathbb{H}$ denoting the Blake2b 256-bit hash introduced by Saarinen and Aumasson 2015 and a function $\mathcal{H}_K(m \in \mathbb{Y}) \in \mathbb{H}$ denoting the Keccak 256-bit hash as proposed by Bertoni et al. 2013 and utilized by Wood 2014.

We may sometimes wish to take only the first x octets of a hash, in which case we denote $\mathcal{H}_x(m) \in \mathbb{Y}_x$ to be the first x octets of $\mathcal{H}(m)$. The inputs of a hash function are generally assumed to be serialized with our codec $\mathcal{E}(x) \in \mathbb{Y}$, however for the purposes of clarity or unambiguity we may also explicitly denote the serialization. Similarly, we may wish to interpret a sequence of octets as some other kind of value with the assumed decoder function $\mathcal{E}^{-1}(x \in \mathbb{Y})$. In both cases, we may subscript the transformation function with the number of octets we expect the octet sequence term to have. Thus, $r = \mathcal{E}_4(x \in \mathbb{N})$ would assert $x \in \mathbb{N}_{2^{32}}$ and $r \in \mathbb{Y}_4$, whereas $s = \mathcal{E}_8^{-1}(y)$ would assert $y \in \mathbb{Y}_8$ and $s \in \mathbb{N}_{2^{64}}$.

3.8.2. *Signing Schemes.* $\mathbb{E}_k\langle m\rangle \subset \mathbb{Y}_{64}$ is the set of valid Ed25519 signatures, defined by Josefsson and Liusvaara 2017, made through knowledge of a secret key whose public key counterpart is $k \in \mathbb{Y}_{32}$ and whose message is m. To aid readability, we denote the set of valid public keys $k \in \mathbb{H}_E$.

We use $\mathbb{Y}_{BLS} \subset \mathbb{Y}_{144}$ to denote the set of public keys for the BLS signature scheme, described by Boneh, Lynn, and Shacham 2004, on curve BLS12-381 defined by Hopwood et al. 2020.

We denote the set of valid Bandersnatch public keys as $\mathbb{H}_B$, defined in appendix G. $\mathbb{F}_{k\in\mathbb{H}_B}^{m\in\mathbb{Y}}\langle x \in \mathbb{Y}\rangle \subset \mathbb{Y}_{96}$ is the set of valid singly-contextualized signatures of utilizing the secret counterpart to the public key k, some context x and message m.

$\bar{\mathbb{F}}_{r\in\mathbb{Y}_R}^{m\in\mathbb{Y}}\langle x \in \mathbb{Y}\rangle \subset \mathbb{Y}_{388}$, meanwhile, is the set of valid Bandersnatch RingVRF deterministic singly-contextualized proofs of knowledge of a secret within some set of secrets identified by some root in the set of valid *roots* $\mathbb{Y}_R \in \mathbb{Y}_{196608}$. We denote $\mathcal{R}(\mathbf{s} \in [\mathbb{H}_B]) \in \mathbb{Y}_R$ to be the root specific to the set of public key counterparts $\mathbf{s}$. A root implies a specific set of Bandersnatch key pairs, knowledge of one of the secrets

would imply being capable of making a unique, valid—and anonymous—proof of knowledge of a unique secret within the set.

Both the Bandersnatch signature and RingVRF proof strictly imply that a member utilized their secret key in combination with both the context x and the message m; the difference is that the member is identified in the former and is anonymous in the latter. Furthermore, both define a VRF *output*, a high entropy hash influenced by x but not by m, formally denoted $\mathcal{Y}(\bar{\mathbb{F}}_r^m\langle x\rangle) \subset \mathbb{H}$ and $\mathcal{Y}(\mathbb{F}_k^m\langle x\rangle) \subset \mathbb{H}$.

We define the function $\mathcal{S}$ as the signature function, such that $\mathcal{S}_k(m) \in \mathbb{F}_k^m\langle[]\rangle \cup \mathbb{E}_k\langle m\rangle$. We assert that the ability to compute a result for this function relies on knowledge of a secret key.

4. Overview

As in the Yellow Paper, we begin our formalisms by recalling that a blockchain may be defined as a pairing of some initial state together with a block-level state-transition function. The latter defines the posterior state given a pairing of some prior state and a block of data applied to it. Formally, we say:

$$\sigma' \equiv \Upsilon(\sigma, \mathbf{B}) \tag{11}$$

Where σ is the prior state, σ' is the posterior state, B is some valid block and Υ is our block-level state-transition function.

Broadly speaking, JAM (and indeed blockchains in general) may be defined simply by specifying Υ and some *genesis state* σ^0.[7] We also make several additional assumptions of agreed knowledge: a universally known clock, and the practical means of sharing data with other systems operating under the same consensus rules. The latter two were both assumptions silently made in the *YP*.

4.1. **The Block.**

To aid comprehension and definition of our protocol, we partition as many of our terms as possible into their functional components. We begin with the block B which may be restated as the header H and some input data

[7] Practically speaking, blockchains sometimes make assumptions of some fraction of participants whose behavior is simply *honest*, and not provably incorrect nor otherwise economically disincentivized. While the assumption may be reasonable, it must nevertheless be stated apart from the rules of state-transition.

external to the system and thus said to be *extrinsic*, $\mathbf{E}$:

$$\mathbf{B} \equiv (\mathbf{H}, \mathbf{E}) \tag{12}$$

$$\mathbf{E} \equiv (\mathbf{E}_T, \mathbf{E}_J, \mathbf{E}_P, \mathbf{E}_A, \mathbf{E}_G) \tag{13}$$

The header is a collection of metadata primarily concerned with cryptographic references to the blockchain ancestors and the operands and result of the present transition. As an immutable known *a priori*, it is assumed to be available throughout the functional components of block transition. The extrinsic data is split into its several portions:

tickets: Tickets, used for the mechanism which manages the selection of validators for the permissioning of block authoring. This component is denoted $\mathbf{E}_T$.

judgements: Votes, by validators, on dispute(s) arising between them presently taking place. This is denoted $\mathbf{E}_J$.

preimages: Static data which is presently being requested to be available for workloads to be able to fetch on demand. This is denoted $\mathbf{E}_P$.

availability: Assurances by each validator concerning which of the input data of workloads they have correctly received and are storing locally. This is denoted $\mathbf{E}_A$.

reports: Reports of newly completed workloads whose accuracy is guaranteed by specific validators. This is denoted $\mathbf{E}_G$.

4.2. **The State.** Our state may be logically partitioned into several largely independent segments which can both help avoid visual clutter within our protocol description and provide formality over elements of computation which may be simultaneously calculated (i.e. parallelized). We therefore pronounce an equivalence between σ (some complete state) and a tuple of partitioned segments of that state:

$$\sigma \equiv (\alpha, \beta, \gamma, \delta, \eta, \iota, \kappa, \lambda, \rho, \tau, \varphi, \chi, \psi) \tag{14}$$

In summary, δ is the portion of state dealing with *services*, analogous in JAM to the Yellow Paper's (smart contract) *accounts*, the only state of the *YP*'s Ethereum. The identities of services which hold some privileged status are tracked in χ.

Validators, who are the set of economic actors uniquely privileged to help build and maintain the JAM chain, are identified within κ, archived in λ and enqueued from ι. All other state concerning the determination of these keys is held within γ. Note this is a departure from the *YP* proof-of-work definitions which were mostly stateless, and this set was not enumerated but rather limited to those with sufficient compute power to find a partial hash-collision in the SHA2-256 cryptographic hash function. An on-chain entropy pool is retained in η.

Our state also tracks two aspects of each core: α, the authorization requirement which work done on that core must satisfy at the time of being reported on-chain, together with the queue which fills this, φ; and ρ, each of the cores' currently assigned *report*, the availability of whose *work-package* must yet be assured by a super-majority of validators.

Finally, details of the most recent blocks and time are tracked in β and τ respectively and ongoing disputes are tracked in $\boldsymbol{\psi}$.

4.2.1. *State Transition Dependency Graph.* Much as in the *YP*, we specify Υ as the implication of formulating all items of posterior state in terms of the prior state and block. To aid the architecting of implementations which parallelize this computation, we minimize the depth of the dependency graph where possible. The overall dependency graph is specified here:

$$\tau' \prec \mathbf{H} \tag{15}$$

$$\beta^\dagger \prec (\mathbf{H}, \beta) \tag{16}$$

$$\beta' \prec (\mathbf{H}, \mathbf{E}_G, \beta^\dagger, \mathbf{C}) \tag{17}$$

$$\gamma' \prec (\mathbf{H}, \tau, \mathbf{E}_T, \gamma, \iota, \eta', \kappa') \tag{18}$$

$$\eta' \prec (\mathbf{H}, \tau, \eta) \tag{19}$$

$$\kappa' \prec (\mathbf{H}, \tau, \kappa, \gamma, \psi') \tag{20}$$

$$\lambda' \prec (\mathbf{H}, \tau, \lambda, \kappa) \tag{21}$$

$$\psi' \prec (\mathbf{E}_J, \psi) \tag{22}$$

$$\delta^\dagger \prec (\mathbf{E}_P, \delta, \tau') \tag{23}$$

$$\rho^{\dagger} \prec (\mathbf{E}_J, \rho) \tag{24}$$

$$\rho^{\ddagger} \prec (\mathbf{E}_A, \rho^{\dagger}) \tag{25}$$

$$\rho' \prec (\mathbf{E}_G, \rho^{\ddagger}, \kappa, \tau') \tag{26}$$

$$\left.\begin{matrix}\delta' \\ \chi' \\ \iota' \\ \varphi' \\ \mathbf{C}\end{matrix}\right\} \prec (\mathbf{E}_A, \rho', \delta^{\dagger}, \chi, \iota, \varphi) \tag{27}$$

$$\alpha' \prec (\mathbf{E}_G, \varphi', \alpha) \tag{28}$$

The only synchronous entangements are visible through the intermediate components superscripted with a dagger and defined in equations 16, 23 and 25. The latter two mark a merge and join in the dependency graph and, concretely, imply that the preimage lookup extrinsic must be folded into state before the availability extrinsic may be fully processed and accumulation of work happen.

4.3. **Which History?** A blockchain is a sequence of blocks, each cryptographically referencing some prior block by including a hash of its header, all the way back to some first block which references the genesis header. We already presume consensus over this genesis header $\mathbf{H}^0$ and the state it represents already defined as σ^0.

By defining a deterministic function for deriving a single posterior state for any (valid) combination of prior state and block, we are able to define a unique *canonical* state for any given block. We generally call the block with the most ancestors the *head* and its state the *head state*.

It is generally possible for two blocks to be valid and yet reference the same prior block in what is known as a *fork*. This implies the possibility of two different heads, each with their own state. While we know of no way to strictly preclude this possibility, for the system to be useful we must nonetheless attempt to minimize it. We therefore strive to ensure that:

(1) It be generally unlikely for two heads to form.

(2) When two heads do form they be quickly resolved into a single head.

(3) It be possible to identify a block not much older than the head which we can be extremely confident will form part of the blockchain's history in perpetuity. When a block becomes identified as such we call it *finalized* and this property naturally extends to all of its ancestor blocks.

These goals are achieved through a combination of two consensus mechanisms: *Safrole*, which governs the (not-necessarily forkless) extension of the blockchain; and *Grandpa*, which governs the finalization of some extension into canonical history. Thus, the former delivers point 1, the latter delivers point 3 and both are important for delivering point 2. We describe these portions of the protocol in detail in sections 6 and 15 respectively.

While Safrole limits forks to a large extent (through cryptography, economics and common-time, below), there may be times when we wish to intentionally fork since we have come to know that a particular chain extension must be reverted. In regular operation this should never happen, however we cannot discount the possibility of malicious or malfunctioning nodes. We therefore define such an extension as any which contains a block in which data is reported which *any other* block's state has tagged as invalid (see section 10 on how this is done). We further require that Grandpa not finalize any extension which contains such a block. See section 15 for more information here.

4.4. **Time.** We presume a pre-existing consensus over time specifically for block production and import. While this was not an assumption of Polkadot, pragmatic and resilient solutions exist including the NTP protocol and network. We utilize this assumption on only one way: we require that blocks be considered temporarily invalid if their timeslot is in the future. This is specified in detail in section 6.

Formally, we define the time in terms of seconds passed since the beginning of the JAM *Common Era*, 1200 UTC on January 1, 2024.[8] Midday CET is selected to ensure that all significant timezones are on the same date at any exact 24-hour multiple from the beginning of the common era. Formally, this value is denoted $\mathcal{T}$.

4.5. **Best block.** Given the recognition of a number of valid blocks, it is necessary to determine which should be treated as the "best" block, by which we mean the most recent block we believe will ultimately be within of all future JAM chains.

[8] 1,704,110,400 seconds after the Unix Epoch.

The simplest and least risky means of doing this would be to inspect the Grandpa finality mechanism which is able to provide a block for which there is a very high degree of confidence it will remain an ancestor to any future chain head.

However, in reducing the risk of the resulting block ultimately not being within the canonical chain, Grandpa will typically return a block some small period older than the most recently authored block. (Existing deployments suggest around 1-2 blocks in the past under regular operation.) There are often circumstances when we may wish to have less latency at the risk of the returned block not ultimately forming a part of the future canonical chain. E.g. we may be in a position of being able to author a block, and we need to decide what its parent should be. Alternatively, we may care to speculate about the most recent state for the purpose of providing information to a downstream application reliant on the state of JAM.

In these cases, we define the best block as the head of the best chain, itself defined in section 15.

4.6. **Economics.** The present work describes a crypto-economic system, i.e. one combining elements of both cryptography and economics and game theory to deliver a self-sovereign digital service. In order to codify and manipulate economic incentives we define a token which is native to the system, which we will simply call *tokens* in the present work.

A value of tokens is generally referred to as a *balance*, and such a value is said to be a member of the set of balances, $\mathbb{N}_B$, which is exactly equivalent to the set of 64-bit unsigned integers:

$$\mathbb{N}_B \equiv \mathbb{N}_{2^{64}} \tag{29}$$

Though unimportant for the present work, we presume that there be a standard named denomination for 10^9 tokens. This is different to both Ethereum (which uses a denomination of 10^{18}), Polkadot (which uses a denomination of 10^{10}) and Polkadot's experimental cousin Kusama (which uses 10^{12}).

The fact that balances are represented as a 64-bit integer implies that there may never be more than around 18×10^9 tokens (each divisible into portions of 10^{-9}) within JAM. We would expect that the total number of tokens ever issued will be a substantially smaller amount than this.

We further presume that a number of constant *prices* stated in terms of tokens are known. However we leave the specific values to be determined in following work:

B_I**:** the additional minimum balance implied for a single item within a mapping.

B_L**:** the additional minimum balance implied for a single octet of data within a mapping.

B_S**:** the minimum balance implied for a service.

4.7. **The Virtual Machine and Gas.** In the present work, we presume the definition of a *Polka Virtual Machine* (PVM). This virtual machine is based around the RISC-V instruction set architecture, specifically the RV32ECM variant, and is the basis for introducing permissionless logic into our state-transition function.

The PVM is comparable to the EVM defined in the Yellow Paper, but somewhat simpler: the complex instructions for cryptographic operations are missing as are those which deal with environmental interactions. Overall it is far less opinionated since it alters a pre-existing general purpose design, RISC-V, and optimizes it for our needs. This gives us excellent pre-existing tooling, since PVM remains essentially compatible with RISC-V, including support from the compiler toolkit LLVM and languages such as Rust and C++. Furthermore, the instruction set simplicity which RISC-V and PVM share, together with the register size (32-bit), active number (13) and endianness (little) make it especially well-suited for creating efficient recompilers on to common hardware architectures.

The PVM is fully defined in appendix A, but for contextualization we will briefly summarize the basic invocation function Ψ which computes the resultant state of a PVM instance initialized with some registers ($\llbracket \mathbb{N}_R \rrbracket_{13}$) and RAM ($\mathbb{M}$) and has executed for up to some amount of gas ($\mathbb{N}_G$), a number of approximately time-proportional computational steps:

$$\Psi\colon \begin{pmatrix} \mathbb{Y}, & \mathbb{N}_R, & \mathbb{N}_G, \\ \llbracket \mathbb{N}_R \rrbracket_{13}, & \mathbb{M} & \end{pmatrix} \to \begin{pmatrix} \{\blacksquare, \lightning, \infty\} \cup \{\lightning, \hbar\} \times \mathbb{N}_R, \\ \mathbb{N}_R, & \mathbb{Z}_G, & \llbracket \mathbb{N}_R \rrbracket_{13}, & \mathbb{M} \end{pmatrix} \tag{30}$$

We refer to the time-proportional computational steps as *gas* (much like in the *YP*) and limit it to a 64-bit quantity. We may use either $\mathbb{N}_G$ or $\mathbb{Z}_G$ to bound

it, the first as a prior argument since it is known to be positive, the latter as a result where a negative value indicates an attempt to execute beyond the gas limit. Within the context of the PVM, $\xi \in \mathbb{N}_G$ is typically used to denote gas.

$$\mathbb{Z}_G \equiv \mathbb{Z}_{-2^{63}:2^{63}} \ , \quad \mathbb{N}_G \equiv \mathbb{N}_{2^{64}} \ , \quad \mathbb{N}_R \equiv \mathbb{N}_{2^{32}} \tag{31}$$

It is left as a rather important implementation detail to ensure that the amount of time taken while computing the function $\Psi(\dots, \xi, \dots)$ has a maximum computation time approximately proportional to the value of ξ regardless of other operands.

The PVM is a very simple RISC *register machine* and as such has 13 registers, each of which is a 32-bit integer, denoted $\mathbb{N}_R$.[9] Within the context of the PVM, $\omega \in \llbracket \mathbb{N}_R \rrbracket_{13}$ is typically used to denote the registers.

$$\mathbb{M} \equiv \left(\mathbf{V} \in \mathbb{Y}_{2^{32}}, \ \mathbf{A} \in \llbracket \{\mathrm{W}, \mathrm{R}, \varnothing\} \rrbracket_{2^{32}}\right) \tag{32}$$

The PVM assumes a simple pageable RAM of 32-bit addressable octets where each octet may be either immutable, mutable or inaccessible. The RAM definition $\mathbb{M}$ includes two components: a value $\mathbf{V}$ and access $\mathbf{A}$. If the component is unspecified while being subscripted then the value component may be assumed. Within the context of the virtual machine, $\mu \in \mathbb{M}$ is typically used to denote RAM.

$$\mathbb{V}_\mu \equiv \{i \mid \mu_\mathbf{A}[i] \neq \varnothing\} \qquad \mathbb{V}^*_\mu \equiv \{i \mid \mu_\mathbf{A}[i] = \mathrm{W}\} \tag{33}$$

We define two sets of indices for the RAM μ: $\mathbb{V}_\mu$ is the set of indices which may be read from; and $\mathbb{V}^*_\mu$ is the set of indices which may be written to.

Invocation of the PVM has an exit-reason as the first item in the resultant tuple. It is either:

- Regular program termination caused by an explicit halt instruction, ∎.
- Irregular program termination caused by some exceptional circumstance, ↯.
- Exhaustion of gas, ∞.

[9] This is three fewer than RISC-V's 16, however the amount that program code output by compilers uses is 13 since two are reserved for operating system use and the third is fixed as zero

- A page fault (attempt to access some address in RAM which is not accessible), $\lightning$. This includes the address at fault.

- An attempt at progressing a host-call, $\hbar$. This allows for the progression and integration of a context-dependent state-machine beyond the regular PVM.

The full definition follows in appendix A.

4.8. **Epochs and Slots.** Unlike the *YP* Ethereum with its proof-of-work consensus system, JAM defines a proof-of-authority consensus mechanism, with the authorized validators presumed to be identified by a set of public keys and decided by a *staking* mechanism residing within some system hosted by JAM. The staking system is out of scope for the present work; instead there is an API which may be utilized to update these keys, and we presume that whatever logic is needed for the staking system will be introduced and utilize this API as needed.

The Safrole mechanism subdivides time following genesis into fixed length *epoch*s with each epoch divided into $\mathsf{E} = 600$ time*slot*s each of uniform length $\mathsf{P} = 6$ seconds, given an epoch period of $\mathsf{E} \cdot \mathsf{P} = 3600$ seconds or one hour.

This six-second slot period represents the minimum time between JAM blocks, and through Safrole we aim to strictly minimize forks arising both due to contention within a slot (where two valid blocks may be produced within the same six-second period) and due to contention over multiple slots (where two valid blocks are produced in different time slots but with the same parent).

Formally when identifying a timeslot index, we use a 32-bit integer indicating the number of six-second timeslots from the JAM Common Era. For use in this context we introduce the set $\mathbb{N}_T$:

$$\mathbb{N}_T \equiv \mathbb{N}_{2^{32}} \tag{34}$$

This implies that the lifespan of the proposed protocol takes us to mid-August of the year 2840, which with the current course that humanity is on should be ample.

4.9. **The Core Model and Services.** Whereas in the Ethereum Yellow Paper when defining the state machine which is held in consensus amongst all network participants, we presume that all machines maintaining the full network state and

contributing to its enlargement—or, at least, hoping to—evaluate all computation. This "everybody does everything" approach might be called the *on-chain consensus model*. It is unfortunately not scalable, since the network can only process as much logic in consensus that it could hope any individual node is capable of doing itself within any given period of time.

4.9.1. *In-core Consensus.* In the present work, we achieve scalability of the work done through introducing a second model for such computation which we call the *in-core consensus model*. In this model, and under normal circumstances, only a subset of the network is responsible for actually executing any given computation and assuring the availability of any input data it relies upon to others. By doing this and assuming a certain amount of computational parallelism within the validator nodes of the network, we are able to scale the amount of computation done in consensus commensurate with the size of the network, and not with the computational power of any single machine. In the present work we expect the network to be able to do upwards of 300 times the amount of computation *in-core* as that which could be performed by a single machine running the virtual machine at full speed.

Since in-core consensus is not evaluated or verified by all nodes on the network, we must find other ways to become adequately confident that the results of the computation are correct, and any data used in determining this is available for a practical period of time. We do this through a crypto-economic game of three stages called *guaranteeing*, *assuring*, *auditing* and, potentially, *judging*. Respectively, these attach a substantial economic cost to the invalidity of some proposed computation; then a sufficient degree of confidence that the inputs of the computation will be available for some period of time; and finally, a sufficient degree of confidence that the validity of the computation (and thus enforcement of the first guarantee) will be checked by some party who we can expect to be honest.

All execution done in-core must be reproducible by any node synchronized to the portion of the chain which has been finalized. Execution done in-core is therefore designed to be as stateless as possible. The requirements for doing it include only the refinement code of the service, the code of the authorizer and any preimage lookups it carried out during its execution.

When a work-report is presented on-chain, a specific block known as the *lookup-anchor* is identified. Correct behavior requires that this must be in the finalized

chain and reasonably recent, both properties which may be proven and thus are acceptable for use within a consensus protocol.

We describe this pipeline in detail in the relevant sections later.

4.9.2. *On Services and Accounts.* In *YP* Ethereum, we have two kinds of accounts: *contract accounts* (whose actions are defined deterministically based on the account's associated code and state) and *simple accounts* which act as gateways for data to arrive into the world state and are controlled by knowledge of some secret key. In JAM, all accounts are *service accounts*. Like Ethereum's contract accounts, they have an associated balance, some code and state. Since they are not controlled by a secret key, they do not need a nonce.

The question then arises: how can external data be fed into the world state of JAM? And, by extension, how does overall payment happen if not by deducting the account balances of those who sign transactions? The answer to the first lies in the fact that our service definition actually includes *multiple* code entry-points, one concerning *refinement* and the other concerning *accumulation*. The former acts as a sort of high-performance stateless processor, able to accept arbitrary input data and distill it into some much smaller amount of output data. The latter code is more stateful, providing access to certain on-chain functionality including the possibility of transferring balance and invoking the execution of code in other services. Being stateful this might be said to more closely correspond to the code of an Ethereum contract account.

To understand how JAM breaks up its service code is to understand JAM's fundamental proposition of generality and scalability. All data extrinsic to JAM is fed into the refinement code of some service. This code is not executed *on-chain* but rather is said to be executed *in-core*. Thus, whereas the accumulator code is subject to the same scalability constraints as Ethereum's contract accounts, refinement code is executed off-chain and subject to no such constraints, enabling JAM services to scale dramatically both in the size of their inputs and in the complexity of their computation.

While refinement and accumulation take place in consensus environments of a different nature, both are executed by the members of the same validator set. The JAM protocol through its rewards and penalties ensures that code executed *in-core* has a comparable level of crypto-economic security to that executed *on-chain*, leaving the primary difference between them one of scalability versus synchroneity.

As for managing payment, JAM introduces a new abstraction mechanism based around Polkadot's Agile Coretime. Within the Ethereum transactive model, the mechanism of account authorization is somewhat combined with the mechanism of purchasing blockspace, both relying on a cryptographic signature to identify a single "transactor" account. In JAM, these are separated and there is no such concept of a "transactor".

In place of Ethereum's gas model for purchasing and measuring blockspace, JAM has the concept of *coretime*, which is prepurchased and assigned to an authorization agent. Coretime is analogous to gas insofar as it is the underlying resource which is being consumed when utilizing JAM. Its procurement is out of scope in the present work and is expected to be managed by a system parachain operating within a parachains service itself blessed with a number of cores for running such system services. The authorization agent allows external actors to provide input to a service without necessarily needing to identify themselves as with Ethereum's transaction signatures. They are discussed in detail in section 8.

5. The Header

We must first define the header in terms of its components. The header comprises a parent hash and prior state root ($\mathbf{H}_p$ and $\mathbf{H}_r$), an extrinsic hash $\mathbf{H}_x$, a time-slot index $\mathbf{H}_t$, the epoch, winning-tickets and judgements markers $\mathbf{H}_e$, $\mathbf{H}_w$ and $\mathbf{H}_j$, a Bandersnatch block author key $\mathbf{K}_k$ and two Bandersnatch signatures; the entropy-yielding VRF signature $\mathbf{H}_v$ and a block seal $\mathbf{H}_s$. Headers may be serialized to an octet sequence with and without the latter seal component using $\mathcal{E}$ and $\mathcal{E}_U$ respectively. Formally:

$$\mathbf{H} \equiv (\mathbf{H}_p, \mathbf{H}_r, \mathbf{H}_x, \mathbf{H}_t, \mathbf{H}_e, \mathbf{H}_w, \mathbf{H}_j, \mathbf{H}_k, \mathbf{H}_v, \mathbf{H}_s) \tag{35}$$

Blocks considered invalid by this rule may become valid as $\mathcal{T}$ advances.

The blockchain is a sequence of blocks, each cryptographically referencing some prior block by including a hash derived from the parent's header, all the way back to some first block which references the genesis header. We already presume consensus over this genesis header $\mathbf{H}^0$ and the state it represents defined as σ^0.

Excepting the Genesis header, all block headers $\mathbf{H}$ have an associated parent header, whose hash is $\mathbf{H}_p$. We denote the parent header $\mathbf{H}^- = P(\mathbf{H})$:

$$\mathbf{H}_p \in \mathbb{H}, \quad \mathbf{H}_p \equiv \mathcal{H}(P(\mathbf{H})) \tag{36}$$

P is thus defined as being the mapping from one block header to its parent block header. With P, we are able to define the set of ancestor headers $\mathbf{A}$:

$$h \in \mathbf{A} \Leftrightarrow h = \mathbf{H} \vee (\exists i \in \mathbf{A} : h = P(i)) \tag{37}$$

We only require implementations to store headers of ancestors which were authored in the previous $\mathsf{L} = 24$ hours of any block $\mathbf{B}$ they wish to validate.

The extrinsic hash is the hash of the block's extrinsic data. Given any block $\mathbf{B} = (\mathbf{H}, \mathbf{E})$, then formally:

$$\mathbf{H}_x \in \mathbb{H}, \quad \mathbf{H}_x \equiv \mathcal{H}(\mathcal{E}(\mathbf{E})) \tag{38}$$

A block may only be regarded as valid once the time-slot index $\mathbf{H}_t$ is in the past. It is always strictly greater than that of its parent. Formally:

$$\mathbf{H}_t \in \mathbb{N}_T, \quad P(\mathbf{H})_t < \mathbf{H}_t \ \wedge \ \mathbf{H}_t \cdot \mathsf{P} \leq \mathcal{T} \tag{39}$$

The parent state root $\mathbf{H}_r$ is the root of a Merkle trie composed by the mapping of the *prior* state's Merkle root, which by definition is also the parent block's posterior state. This is a departure from both Polkadot and the Yellow Paper's Ethereum, in both of which a block's header contains the *posterior* state's Merkle root. We do this to facilitate the pipelining of block computation and in particular of Merklization.

$$\mathbf{H}_r \in \mathbb{H}, \quad \mathbf{H}_r \equiv \mathcal{M}_S(\sigma) \tag{40}$$

We assume the state-Merklization function $\mathcal{M}_S$ is capable of transforming our state σ into a 32-octet commitment. See appendix D for a full definition of these two functions.

All blocks have an associated public key to identify the author of the block. We identify this as an index into the current validator set κ. We denote the Bandersnatch key of the author as $\mathbf{H}_a$ though note that this is merely an equivalence, and is not serialized as part of the header.

$$\mathbf{H}_k \in \mathbb{N}_V\,, \quad \mathbf{H}_a \equiv \kappa[\mathbf{H}_k] \tag{41}$$

5.1. **The Epoch and Winning Tickets Markers.** If not $\varnothing$, then the epoch marker specifies key and entropy relevant to the following epoch in case the ticket contest does not complete adequately (a very much unexpected eventuality). Similarly, the winning-tickets marker, if not $\varnothing$, provides the series of 600 slot sealing "tickets" for the next epoch (see the next section):

$$\mathbf{H}_e \in (\mathbb{H}, \llbracket \mathbb{H}_B \rrbracket_{\mathsf{V}})?\,, \qquad \mathbf{H}_w \in \llbracket \mathbb{C} \rrbracket_{\mathsf{E}}? \tag{42}$$

The terms are fully defined in section 6.6.

6. Block Production and Chain Growth

As mentioned earlier, JAM is architected around a hybrid consensus mechanism, similar in nature to that of Polkadot's BABE/GRANDPA hybrid. JAM's block production mechanism, termed Safrole after the novel Sassafras production mechanism of which it is a simplified variant, is a stateful system rather more complex than the Nakamoto consensus described in the *YP*.

The chief purpose of a block production consensus mechanism is to limit the rate at which new blocks may be authored and, ideally, preclude the possibility of "forks": multiple blocks with equal numbers of ancestors.

To achieve this, Safrole limits the possible author of any block within any given six-second timeslot to a single key-holder from within a prespecified set of *validators*. Furthermore, under normal operation, the identity of the key-holder of any future timeslot will have a very high degree of anonymity. As a side effect of its operation, we can generate a high-quality pool of entropy which may be used by other parts of the protocol and is accessible to services running on it.

Because of its tightly scoped role, the core of Safrole's state, γ, is independent of the rest of the protocol. It interacts with other portions of the protocol through

ι and κ, the prospective and active sets of validator keys respectively; τ, the most recent block's timeslot; and η, the entropy accumulator.

The Safrole protocol generates, once per epoch, a sequence of E *sealing keys*, one for each potential block within a whole epoch. Each block header includes its timeslot index $\mathbf{H}_t$ (the number of six-second periods since the JAM Common Era began) and a valid seal signature $\mathbf{H}_s$, signed by the sealing key corresponding to the timeslot within the aforementioned sequence. Each sealing key is in fact a pseudonym for some validator which was agreed the privilege of authoring a block in the corresponding timeslot.

In order to generate this sequence of sealing keys, and in particular to do so without making public the correspondence relation between them and the validator set, we use a novel cryptographic structure known as a RingVRF, utilizing the Bandersnatch curve. Bandersnatch RingVRF allows for a proof to be provided which simultaneously guarantees the author controlled a key within a set (in our case validators), and secondly provides an output, an unbiasable deterministic hash giving us a secure verifiable random function (VRF) and as a means of determining which validators are able to author in which slots.

6.1. **Timekeeping.** Here, τ defines the most recent block's slot index, which we transition to the slot index as defined in the block's header:

$$\tau \in \mathbb{N}_T \ , \quad \tau' \equiv \mathbf{H}_t \tag{43}$$

We track the slot index in state as τ in order that we are able to easily both identify a new epoch and determine the slot at which the prior block was authored. We denote e as the prior's epoch index and m as the prior's slot phase index within that epoch and e' and m' are the corresponding values for the present block:

$$\text{let} \quad e \,\mathsf{R}\, m = \frac{\tau}{\mathsf{E}}, \quad e' \,\mathsf{R}\, m' = \frac{\tau'}{\mathsf{E}} \tag{44}$$

6.2. **Safrole Basic State.** We restate γ into a number of components:

$$\gamma \equiv (\gamma_{\mathbf{k}}, \gamma_z, \gamma_{\mathbf{s}}, \gamma_{\mathbf{a}}) \tag{45}$$

γ_z is the epoch's root, a Bandersnatch ring root composed with the one Bandersnatch key of each of the next epoch's validators, defined in $\gamma_{\mathbf{k}}$ (itself defined

in the next section).

$$\gamma_z \in \mathbb{Y}_R \tag{46}$$

Finally, $\gamma_{\mathbf{a}}$ is the ticket accumulator, a series of highest-scoring ticket identifiers to be used for the next epoch. $\gamma_{\mathbf{s}}$ is the current epoch's slot-sealer series, which is either a full complement of E tickets or, in the case of a fallback mode, a series of E Bandersnatch keys:

$$\gamma_{\mathbf{a}} \in \llbracket \mathbb{C} \rrbracket_{:\mathsf{E}}, \quad \gamma_{\mathbf{s}} \in \llbracket \mathbb{C} \rrbracket_{\mathsf{E}} \cup \llbracket \mathbb{H}_B \rrbracket_{\mathsf{E}} \tag{47}$$

Here, $\mathbb{C}$ is used to denote the set of *tickets*, a combination of a verifiably random ticket identifier $\mathbf{y}$ and the ticket's entry-index r:

$$\mathbb{C} \equiv (\mathbf{y} \in \mathbb{H}, r \in \mathbb{N}_{\mathsf{N}}) \tag{48}$$

As we state in section 6.4, Safrole requires that every block header $\mathbf{H}$ contain a valid seal $\mathbf{H}_s$, which is a Bandersnatch signature for a public key at the appropriate index m of the current epoch's seal-key series, present in state as $\gamma_{\mathbf{s}}$.

6.3. **Key Rotation.** In addition to the active set of validator keys κ and staging set ι, internal to the Safrole state we retain a pending set $\gamma_{\mathbf{k}}$. The active set is the set of keys identifying the nodes which are currently privileged to author blocks and carry out the validation processes, whereas the pending set $\gamma_{\mathbf{k}}$, which is reset to ι at the beginning of each epoch, is the set of keys which will be active in the next epoch and which determine the Bandersnatch ring root which authorizes tickets into the sealing-key contest for the next epoch.

$$\iota \in \llbracket \mathbb{K} \rrbracket_{\mathsf{V}}, \quad \gamma_{\mathbf{k}} \in \llbracket \mathbb{K} \rrbracket_{\mathsf{V}}, \quad \kappa \in \llbracket \mathbb{K} \rrbracket_{\mathsf{V}}, \quad \lambda \in \llbracket \mathbb{K} \rrbracket_{\mathsf{V}} \tag{49}$$

We must introduce $\mathbb{K}$, the set of validator key tuples. This is a combination of cryptographic public keys for Bandersnatch and Ed25519 cryptography, and a third metadata key which is an opaque octet sequence, but utilized to specify practical identifiers for the validator, not least a hardware address.

The set of validator keys itself is equivalent to the set of 176-octet sequences. However, for clarity, we divide the sequence into four easily denoted components.

For any validator key v, the Bandersnatch key is denoted v_b, and is equivalent to the first 32-octets; the Ed25519 key, v_e, is the second 32 octets; the BLS key denoted v_{BLS} is equivalent to the following 144 octets, and finally the metadata v_m is the last 128 octets. Formally:

$$\mathbb{K} \equiv \mathbb{Y}_{336} \tag{50}$$

$$\forall v \in \mathbb{K} : v_b \in \mathbb{H}_B \equiv v_{0\cdots+32} \tag{51}$$

$$\forall v \in \mathbb{K} : v_e \in \mathbb{H}_E \equiv v_{32\cdots+32} \tag{52}$$

$$\forall v \in \mathbb{K} : v_{BLS} \in \mathbb{Y}_{BLS} \equiv v_{64\cdots+144} \tag{53}$$

$$\forall v \in \mathbb{K} : v_m \in \mathbb{Y}_{208} \equiv v_{208\cdots+128} \tag{54}$$

With a new epoch under regular conditions, validator keys get rotated and the epoch's Bandersnatch key root is updated into γ'_z:

$$\begin{aligned}(\gamma'_{\mathbf{k}}, \kappa', \lambda', \gamma'_z) &\equiv \begin{cases}(\iota, N(\gamma_{\mathbf{k}}), N(\kappa), z) & \text{if } e' > e \wedge \mathbf{H}_J \neq [\,] \\ (\gamma_{\mathbf{k}}, N(\kappa), N(\lambda), \gamma_z) & \text{otherwise}\end{cases} \\ \text{where } z &= \mathcal{R}([k_b \mid k \in \gamma'_{\mathbf{k}}]) \\ \text{and } N(\mathbf{k}) &\equiv \left[\begin{cases}[0,0,\dots] & \text{if } k_e \in \psi'_{\mathbf{p}} \\ k & \text{otherwise}\end{cases} \,\middle|\, k \in \mathbf{k}\right]\end{aligned} \tag{55}$$

Note that the posterior active validator key set κ' is defined such that keys belonging to the historical judgement punish set $\psi'_{\mathbf{p}}$ are replaced with a null key containing only zeroes. The origin of this punish set is explained in section 10.

6.4. **Sealing and Entropy Accumulation.** The header must contain a valid seal and valid VRF output. These are two signatures both using the current slot's seal key; the message data of the former is the header's serialization omitting the seal component $\mathbf{H}_s$, whereas the latter is used as a bias-resistant entropy source and thus its message must already have been fixed: we use the entropy stemming from the VRF of the seal signature. Formally:

$$\text{let } i = \gamma'_{\mathbf{s}}[\mathbf{H}_t]^{\circlearrowleft}:$$

$$\gamma'_{\mathbf{s}} \in [\![\mathbb{C}]\!] \implies \begin{cases} i_{\mathbf{y}} = \mathcal{Y}(\mathbf{H}_s)\,, \\ \mathbf{H}_s \in \mathbb{F}^{\mathcal{E}_U(\mathbf{H})}_{\mathbf{H}_a}\langle \mathsf{X}_S \frown \eta'_3 \mathbin{+\!\!+} i_r \rangle\,, \\ \mathbf{T} = 1 \end{cases} \tag{56}$$

$$\gamma'_{\mathbf{s}} \in [\![\mathbb{H}_B]\!] \implies \begin{cases} i = \mathbf{H}_a\,, \\ \mathbf{H}_s \in \mathbb{F}^{\mathcal{E}_U(\mathbf{H})}_{\mathbf{H}_a}\langle \mathsf{X}_F \frown \eta'_3 \rangle\,, \\ \mathbf{T} = 0 \end{cases} \tag{57}$$

$$\mathbf{H}_v \in \mathbb{F}^{[]}_{\mathbf{H}_a}\langle \mathsf{X}_E \frown \mathcal{Y}(\mathbf{H}_s) \rangle \tag{58}$$

$$\mathsf{X}_E = \texttt{\$jam\_entropy} \tag{59}$$

$$\mathsf{X}_F = \texttt{\$jam\_fallback\_seal} \tag{60}$$

$$\mathsf{X}_S = \texttt{\$jam\_seal} \tag{61}$$

Sealing using the ticket is of greater security, and we utilize this knowledge when determining a candidate block on which to extend the chain, detailed in section 15. We thus note that the block was sealed under the regular security with the boolean marker $\mathbf{T}$. We define this only for the purpose of ease of later specification.

In addition to the entropy accumulator η_0, we retain three additional historical values of the accumulator at the point of each of the three most recently ended epochs, η_1, η_2 and η_3. The second-oldest of these η_2 is utilized to help ensure future entropy is unbiased (see equation 62) and seed the fallback seal-key generation function with randomness (see equation 65). The oldest is used to regenerate this randomness when verifying the seal above.

$$\eta \in [\![\mathbb{H}]\!]_4 \tag{62}$$

η_0 defines the state of the randomness accumulator to which the provably random output of the VRF, the signature over some unbiasable input, is combined each block. η_1 and η_2 meanwhile retain the state of this accumulator at the end of the two most recently ended epochs in order.

$$\eta'_0 \equiv \mathcal{H}(\eta_0 \frown \mathcal{Y}(\mathbf{H}_v)) \tag{63}$$

On an epoch transition (identified as the condition $e' > e$), we therefore rotate the accumulator value into the history η_1, η_2 and η_3:

$$(\eta'_1, \eta'_2, \eta'_3) \equiv \begin{cases} (\eta_0, \eta_1, \eta_2) & \text{if } e' > e \\ (\eta_1, \eta_2, \eta_3) & \text{otherwise} \end{cases} \tag{64}$$

6.5. **The Slot Key Sequence.** The posterior slot key sequence $\gamma'_{\mathbf{s}}$ is one of three expressions depending on the circumstance of the block. If the block is not the first in an epoch, then it remains unchanged from the prior $\gamma_{\mathbf{s}}$. If the block signals the next epoch (by epoch index) and the previous block's slot was within the closing period of the previous epoch, then it takes the value of the prior ticket accumulator $\gamma'_{\mathbf{a}}$. Otherwise, it takes the value of the fallback key sequence. Formally:

$$\gamma'_{\mathbf{s}} \equiv \begin{cases} Z(\gamma_{\mathbf{a}}) & \text{if } e' = e + 1 \wedge m' \geq \mathsf{Y} \wedge |\gamma_{\mathbf{a}}| = \mathsf{E} \\ \gamma_{\mathbf{s}} & \text{if } e' = e \\ F(\eta'_2, \kappa') & \text{otherwise} \end{cases} \tag{65}$$

Here, we use Z as the inside-out sequencer function, defined as follows:

$$Z: \begin{cases} \llbracket \mathbb{C} \rrbracket_{\mathsf{E}} \to \llbracket \mathbb{C} \rrbracket_{\mathsf{E}} \\ \quad \mathbf{s} \mapsto [\mathbf{s}_0, \mathbf{s}_{|\mathbf{s}|-1}, \mathbf{s}_1, \mathbf{s}_{|\mathbf{s}|-2}, \dots] \end{cases} \tag{66}$$

Finally, F is the fallback key sequence function which selects an epoch's worth of validator Bandersnatch keys ($\llbracket \mathbb{H}_B \rrbracket_{\mathsf{E}}$) at random from the validator key set $\mathbf{k}$ using the entropy collected on-chain r:

$$F: \begin{cases} (\mathbb{H}, \llbracket \mathbb{K} \rrbracket) \to \llbracket \mathbb{H}_B \rrbracket_{\mathsf{E}} \\ \quad (r, \mathbf{k}) \mapsto \left[\mathbf{k}[\mathcal{E}^{-1}(\mathcal{H}_4(r \frown \mathcal{E}_4(i)))]_b^{\circlearrowleft} \,\middle|\, i \in \mathbb{N}_{\mathsf{E}} \right] \end{cases} \tag{67}$$

6.6. **The Markers.** The epoch and winning-tickets markers are information placed in the header in order to minimize data transfer necessary to determine the validator keys associated with any given epoch. They are particularly useful to nodes which do not synchronize the entire state for any given block since they facilitate the secure tracking of changes to the validator key sets using only the chain of headers.

As mentioned earlier, the header's epoch marker $\mathbf{H}_e$ is either empty or, if the block is the first in a new epoch, then a tuple of the epoch randomness and a sequence of Bandersnatch keys defining the Bandersnatch validator keys (k_b) beginning in the next epoch. Formally:

$$\mathbf{H}_e \equiv \begin{cases} (\eta'_1, [k_e \mid k \in \gamma'_{\mathbf{k}}]) & \text{if } e' > e \\ \varnothing & \text{otherwise} \end{cases} \tag{68}$$

The winning-tickets marker $\mathbf{H}_w$ is either empty or, if the block is the first after the end of the submission period for tickets and if the ticket accumulator is saturated, then the final sequence of ticket identifiers. Formally:

$$\mathbf{H}_w \equiv \begin{cases} Z(\gamma_{\mathbf{a}}) & \text{if } e' = e \wedge m < \mathsf{Y} \leq m' \wedge |\gamma_{\mathbf{a}}| = \mathsf{E} \\ \varnothing & \text{otherwise} \end{cases} \tag{69}$$

Note that this will not be honored if the next epoch begins with a judgement in its extrinsic.

6.7. **The Extrinsic and Tickets.** The extrinsic $\mathbf{E}_T$ is a sequence of proofs of valid tickets; a ticket implies an entry in our epochal "contest" to determine which validators are privileged to author a block for each timeslot in the following epoch. Tickets specify an ephemeral key and an entry index, both of which are elective, together with a proof of the ticket's validity. The proof implies a ticket identity, a high-entropy unbiasable 32-octet sequence, which is used both as a score in the aforementioned contest and as input to the on-chain VRF.

Towards the end of the epoch (i.e. Y slots from the start) this contest is closed implying successive blocks within the same epoch must have an empty tickets extrinsic. At this point, the following epoch's seal key sequence becomes fixed.

We define the extrinsic as a sequence of proofs of valid tickets, each of which is a tuple of an entry index (a natural number less than N) and a proof of ticket validity. Formally:

$$\mathbf{E}_T \in \left\llbracket \left(r \in \mathbb{N}_{\mathsf{N}},\ p \in \bar{\mathbb{F}}^{[]}_{\gamma_z}\langle \mathsf{X}_T \frown \eta'_2 \mathbin{+\!\!+} r \rangle \right) \right\rrbracket \tag{70}$$

$$|\mathbf{E}_T| \leq \begin{cases} \mathsf{K} & \text{if } m' < \mathsf{Y} \\ 0 & \text{otherwise} \end{cases} \tag{71}$$

$$\mathsf{X}_T = \texttt{\$jam\_ticket} \tag{72}$$

We define $\mathbf{n}$ as the set of new tickets, with the ticket identity, a hash, defined as the output component of the Bandersnatch RingVRF proof:

$$\mathbf{n} \equiv [(\mathbf{y} \colon \mathcal{Y}(i_p),\ r \colon i_r) \mid i \in\!\!\!\!\cdot\ \mathbf{E}_T] \tag{73}$$

The tickets submitted via the extrinsic must already have been placed in order of their implied identity. Duplicate identities are never allowed lest a validator submit the same ticket multiple times:

$$\mathbf{n} = [x_{\mathbf{y}} \,\|\, x \in \mathbf{n}] \tag{74}$$

$$\{x_{\mathbf{y}} \mid x \in \mathbf{n}\} \barwedge \{x_{\mathbf{y}} \mid x \in \gamma_{\mathbf{a}}\} \tag{75}$$

The new ticket accumulator $\gamma'_{\mathbf{a}}$ is constructed by merging new tickets into the previous accumulator value (or the empty sequence if it is a new epoch):

$$\gamma'_{\mathbf{a}} \equiv \overrightarrow{\left[\left\{ x_{\mathbf{y}} \right\} x \in \mathbf{n} \cup \begin{cases} \varnothing & \text{if } e' > e \\ \gamma_{\mathbf{a}} & \text{otherwise} \end{cases} \right]}^{\mathsf{E}} \tag{76}$$

The maximum size of the ticket accumulator is E. On each block, the accumulator becomes the lowest items of the sorted union of tickets from prior accumulator $\gamma_{\mathbf{a}}$ and the submitted tickets. It is invalid to include useless tickets in the extrinsic, so all submitted tickets must exist in their posterior ticket accumulator. Formally:

$$\mathbf{n} \subset \gamma'_{\mathbf{a}} \tag{77}$$

Note that it can be shown that in the case of an empty extrinsic $\mathbf{E}_T = []$, as implied by $m' \geq \mathsf{Y}$, then $\gamma'_{\mathbf{a}} = \gamma_{\mathbf{a}}$.

7. Recent History

We retain in state information on the most recent H blocks. This is used to preclude the possibility of duplicate or out of date work-reports from being

submitted.

$$\beta \in \llbracket (h \in \mathbb{H}, \mathbf{b} \in \llbracket \mathbb{H}? \rrbracket, s \in \mathbb{H}, \mathbf{p} \in \llbracket \mathbb{H} \rrbracket_{:\mathsf{C}}) \rrbracket_{:\mathsf{H}} \tag{78}$$

For each recent block, we retain its header hash, its state root, its accumulation-result MMR and the hash of each work-report made into it which is no more than the total number of cores, $\mathcal{C} = 341$.

During the accumulation stage, a value with the partial transition of this state is provided which contains the update for the newly-known roots of the parent block:

$$\beta^\dagger \equiv \beta \quad \text{except } \beta^\dagger[0]_s = \mathbf{H}_r \tag{79}$$

The final state transition is then:

$$\begin{aligned} \beta' \equiv \beta^\dagger \overleftarrow{+}^{\mathsf{H}} \left(\begin{array}{l} \mathbf{p}{\vdots}\,[((g_w)_s)_p \mid g \lhd \mathbf{E}_G]\,, \\ h{\vdots}\,\mathcal{H}(\mathbf{H})\,, \quad \mathbf{b}\,, \quad s{\vdots}\,\mathbb{H}^0 \end{array} \right) \\ \text{where } \mathbf{b} = \mathcal{A}(\text{last}([[]] \frown [x_{\mathbf{b}} \mid x \lhd \beta]), r) \\ \text{and } r = \mathcal{M}_2([x \rangle \mathcal{E}(x) \mid x \in \mathbf{C}], \mathcal{H}_K) \end{aligned} \tag{80}$$

Thus, we extend the recent history with the new block's header hash, its accumulation-result Merkle tree root and the set of work-reports made into it. Note that the accumulation-result tree root r is derived from $\mathbf{C}$ (defined in section 12) using the basic binary Merklization function $\mathcal{M}_2$ (defined in appendix F) and appending it using the MMR append function $\mathcal{A}$ (defined in appendix F.2) to form a Merkle mountain range.

The state-trie root is as being the zero hash, $\mathbb{H}^0$ which while inaccurate at the end state of the block β', it is nevertheless safe since $\boldsymbol{\beta}'$ is not utilized except to define the next block's $\boldsymbol{\beta}^\dagger$, which contains a corrected value for this.

8. Authorization

We have previously discussed the model of work-packages and services in section 4.9, however we have yet to make a substantial discussion of exactly how some *coretime* resource may be apportioned to some work-package and its associated service. In the *YP* Ethereum model, the underlying resource, gas, is procured

at the point of introduction on-chain and the purchaser is always the same agent who authors the data which describes the work to be done (i.e. the transaction). Conversely, in Polkadot the underlying resource, a parachain slot, is procured with a substantial deposit for typically 24 months at a time and the procurer, generally a parachain team, will often have no direct relation to the author of the work to be done (i.e. a parachain block).

On a principle of flexibility, we would wish JAM capable of supporting a range of interaction patterns both Ethereum-style and Polkadot-style. In an effort to do so, we introduce the *authorization system*, a means of disentangling the intention of usage for some coretime from the specification and submission of a particular workload to be executed on it. We are thus able to disassociate the purchase and assignment of coretime from the specific determination of work to be done with it, and so are able to support both Ethereum-style and Polkadot-style interaction patterns.

8.1. **Authorizers and Authorizations.** The authorization system involves two key concepts: *authorizers* and *authorizations*. An authorization is simply a piece of opaque data to be included with a work-package. An authorizer meanwhile, is a piece of pre-parameterized logic which accepts as an additional parameter an authorization and, when executed within a VM of prespecified computational limits, provides a Boolean output denoting the veracity of said authorization.

Authorizations are identified as the hash of their logic (specified as the VM code) and their pre-parameterization. The process by which work-packages are determined to be authorized (or not) is not the competence of on-chain logic and happens entirely in-core and as such is discussed in section 13.2. However, on-chain logic must identify each set of authorizers assigned to each core in order to verify that a work-package is legitimately able to utilize that resource. It is this subsystem we will now define.

8.2. **Pool and Queue.** We define the set of authorizers allowable for a particular core c as the *authorizer pool* $\alpha[c]$. To maintain this value, a further portion of state is tracked for each core: the core's current *authorizer queue* $\varphi[c]$, from which we draw values to fill the pool. Formally:

$$\alpha \in \left\llbracket \left\llbracket \mathbb{H} \right\rrbracket_{:\mathsf{O}} \right\rrbracket_{\mathsf{C}} , \qquad \varphi \in \left\llbracket \left\llbracket \mathbb{H} \right\rrbracket_{\mathsf{Q}} \right\rrbracket_{\mathsf{C}} \tag{81}$$

Note: The portion of state φ may be altered only through an exogenous call made from the accumulate logic of an appropriately privileged service.

The state transition of a block involves placing a new authorization into the pool from the queue:

$$\forall c \in \mathbb{N}_\mathsf{C} : \alpha'[c] \equiv \overleftarrow{F(c) \pplus \varphi'[c][\mathbf{H}_t]^{\mathsf{O}}}^{\mathsf{O}} \tag{82}$$

$$F(c) \equiv \begin{cases} \alpha[c] \setminus \{g_a\} & \text{if } \exists g \in E_G : g_c = c \\ \alpha[c] & \text{otherwise} \end{cases} \tag{83}$$

Since α' is dependent on φ', practically speaking, this step must be computed after accumulation, the stage in which φ' is defined.

9. Service Accounts

As we already noted, a service in JAM is somewhat analogous to a smart contract in Ethereum in that it includes amongst other items, a code component, a storage component and a balance. Unlike Ethereum, the code is split over two isolated entry-points each with their own environmental conditions; one, *refinement*, is essentially stateless and happens in-core, and the other, *accumulation*, which is stateful and happens on-chain. It is the latter which we will concern ourselves with now.

Service accounts are held in state under δ, a partial mapping from a service identifier $\mathbb{N}_S$ into a tuple of named elements which specify the attributes of the service relevant to the JAM protocol. Formally:

$$\mathbb{N}_S \equiv \mathbb{N}_{2^{32}} \tag{84}$$

$$\delta \in \mathbb{D}\langle \mathbb{N}_S \to \mathbb{A} \rangle \tag{85}$$

The service account is defined as the tuple of storage dictionary $\mathbf{s}$, preimage lookup dictionaries $\mathbf{p}$ and $\mathbf{l}$, code hash c, and balance b as well as the two code gas limits g & m. Formally:

$$\mathbb{A} \equiv \begin{pmatrix} \mathbf{s} \in \mathbb{D}\langle \mathbb{H} \to \mathbb{Y} \rangle \ , & \mathbf{p} \in \mathbb{D}\langle \mathbb{H} \to \mathbb{Y} \rangle \ , \\ \mathbf{l} \in \mathbb{D}\langle (\mathbb{H}, \mathbb{N}_L) \to [\mathbb{N}_T]_{:3} \rangle \ , & \\ c \in \mathbb{H} \ , \quad b \in \mathbb{N}_B \ , & g \in \mathbb{Z}_G \ , \quad m \in \mathbb{Z}_G \end{pmatrix} \tag{86}$$

Thus, the balance of the service of index s would be denoted $\delta[s]_b$ and the storage item of key $k \in \mathbb{H}$ for that service is written $\delta[s]_\mathbf{s}[k]$.

9.1. **Code and Gas.** The code c of a service account is represented by a hash which, if the service is to be functional, must be present within its preimage lookup (see section 9.2). We thus define the actual code $\mathbf{c}$:

$$\forall \mathbf{a} \in \mathbb{A} : \mathbf{a_c} \equiv \begin{cases} \mathbf{a_p}[\mathbf{a}_c] & \text{if } \mathbf{a}_c \in \mathbf{a_p} \\ \varnothing & \text{otherwise} \end{cases} \tag{87}$$

There are three entry-points in the code:

0 `refine`: Refinement, executed in-core and stateless.[10]

1 `accumulate`: Accumulation, executed on-chain and stateful.

2 `on_transfer`: Transfer handler, executed on-chain and stateful.

Whereas the first, executing in-core, is described in more detail in section 13.2, the latter two are defined in the present section.

As stated in appendix A, execution time in the JAM virtual machine is measured deterministically in units of *gas*, represented as a 64-bit integer formally denoted $\mathbb{Z}_G$. There are two limits specified in the account, g, the minimum gas required in order to execute the *Accumulate* entry-point of the service's code, and m, the minimum required for the *On Transfer* entry-point.

9.2. **Preimage Lookups.** In addition to storing data in arbitrary key/value pairs available only on-chain, an account may also solicit data to be made available also in-core, and thus available to the Refine logic of the service's code. State concerning this facility is held under the service's $\mathbf{p}$ and $\mathbf{l}$ components.

There are several differences between preimage-lookups and storage. Firstly, preimage-lookups act as a mapping from a hash to its preimage, whereas general storage maps arbitrary keys to values. Secondly, preimage data is supplied extrinsically, whereas storage data originates as part of the service's accumulation. Thirdly preimage data, once supplied, may not be removed freely; instead it goes through a process of being marked as unavailable, and only after a period of time

[10] Technically there is some small assumption of state, namely that some modestly recent instance of each service's preimages. The specifics of this are discussed in section 13.2.

may it be removed from state. This ensures that historical information on its existence is retained. The final point especially is important since preimage data is designed to be queried in-core, under the Refine logic of the service's code, and thus it is important that the historical availability of the preimage is known.

We begin by reformulating the portion of state concerning our data-lookup system. The purpose of this system is to provide a means of storing static data on-chain such that it may later be made available within the execution of any service code as a function accepting only the hash of the data and its length in octets.

During the on-chain execution of the *Accumulate* function, this is trivial to achieve since there is inherently a state which all validators verifying the block necessarily have complete knowledge of, i.e. σ. However, for the in-core execution of *Refine*, there is no such state inherently available to all validators; we thus name a historical state, the *lookup anchor* which must be considered recently finalized before the work result may be accumulated hence providing this guarantee.

By retaining historical information on its availability, we become confident that any validator with a recently finalized view of the chain is able to determine whether any given preimage was available at any time within the period where auditing may occur. This ensures confidence that judgements will be deterministic even without consensus on chain state.

Restated, we must be able to define some *historical* lookup function Λ which determines whether the preimage of some hash h was available for lookup by some service account $\mathbf{a}$ at some timeslot t, and if so, provide its preimage:

$$\Lambda: \begin{cases} (\mathbb{A}, \mathbb{N}_{\mathbf{H}_t - C_D \ldots \mathbf{H}_t}, \mathbb{H}) \to \mathbb{Y}? \\ \qquad (\mathbf{a}, t, \mathcal{H}(\mathbf{p})) \mapsto v : v \in \{\mathbf{p}, \varnothing\} \end{cases} \tag{88}$$

This function is defined shortly below in equation 90.

The preimage lookup for some service of index s is denoted $\delta[s]_{\mathbf{p}}$ is a dictionary mapping a hash to its corresponding preimage. Additionally, there is metadata associated with the lookup denoted $\delta[s]_{\mathbf{l}}$ which is a dictionary mapping some hash and presupposed length into historical information.

9.2.1. *Invariants.* The state of the lookup system naturally satisfies a number of invariants. Firstly, any preimage value must correspond to its hash, equation 89.

Secondly, a preimage value being in state implies that its hash and length pair has some associated status, also in equation 89. Formally:

$$\forall a \in \mathbb{A}, (h \mapsto \mathbf{p}) \in a_{\mathbf{p}} \Rightarrow h = \mathcal{H}(\mathbf{p}) \wedge (h, |\mathbf{p}|) \in \mathcal{K}(a_{\mathbf{l}}) \tag{89}$$

9.2.2. *Semantics.* The historical status component $h \in [\mathbb{N}_T]_{:3}$ is a sequence of up to three time slots and the cardinality of this sequence implies one of four modes:

- $h = []$: The preimage is *requested*, but has not yet been supplied.
- $h \in [\![\mathbb{N}_T]\!]_1$: The preimage is *available* and has been from time h_0.
- $h \in [\![\mathbb{N}_T]\!]_2$: The previously available preimage is now *unavailable* since time h_1. It had been available from time h_0.
- $h \in [\![\mathbb{N}_T]\!]_3$: The preimage is *available* and has been from time h_2. It had previously been available from time h_0 until time h_1.

The historical lookup function Λ may now be defined as:

$$\begin{aligned}
&\Lambda\colon (\mathbb{A}, \mathbb{N}_T, \mathbb{H}) \to \mathbb{Y}? \\
&\Lambda(\mathbf{a}, t, h) \equiv \begin{cases} \mathbf{a}_{\mathbf{p}}[h] & \text{if } h \in \mathcal{K}(\mathbf{a}_{\mathbf{p}}) \wedge I(\mathbf{a}_{\mathbf{l}}[h, |\mathbf{a}_{\mathbf{p}}[h]|], t) \\ \varnothing & \text{otherwise} \end{cases} \\
&\text{where } I(\mathbf{l}, t) = \begin{cases} \bot & \text{if } [] = \mathbf{l} \\ x \le t & \text{if } [x] = \mathbf{l} \\ x \le t < y & \text{if } [x, y] = \mathbf{l} \\ x \le t < y \vee z \le t & \text{if } [x, y, z] = \mathbf{l} \end{cases}
\end{aligned} \tag{90}$$

9.3. **Account Footprint and Threshold Balance.** We define the dependent values i and l as the storage footprint of the service, specifically the number of items in storage and the total number of octets used in storage. They are defined purely in terms of the storage map of a service, and it must be assumed that whenever a service's storage is changed, these change also.

Furthermore, as we will see in the account serialization function in section C, these are expected to be found explicitly within the Merklized state data. Because of this we make explicit their set.

We may then define a second dependent term t, the minimum, or *threshold*, balance needed for any given service account in terms of its storage footprint.

$$\forall a \in \mathcal{V}(\delta) : \begin{cases} a_i \in \mathbb{N}_{2^{32}} & \equiv 2 \cdot |a_{\mathbf{l}}| + |a_{\mathbf{s}}| \\ a_l \in \mathbb{N}_{2^{64}} & \equiv \sum_{(h,z) \in \mathcal{K}(a_{\mathbf{l}})} 81 + z \\ & \quad + \sum_{x \in \mathcal{V}(a_{\mathbf{s}})} 32 + |x| \\ a_t \in \mathbb{N}_B & \equiv \mathsf{B}_S + \mathsf{B}_I \cdot a_i + \mathsf{B}_L \cdot a_l \end{cases} \tag{91}$$

9.4. **Service Privileges.** Up to three services may be recognized as privileged. The portion of state in which this is held is denoted χ and has three components, each a service index. m is the index of the *manager* service, the service able to effect an alteration of χ from block to block. a and v are each the indices of services able to alter φ and ι from block to block. Formally:

$$\chi \equiv (\chi_m \in \mathbb{N}_S, \chi_a \in \mathbb{N}_S, \chi_v \in \mathbb{N}_S) \tag{92}$$

10. Judgements

Jam provides a means of recording a vote amongst all validators over the validity of a *work-report*, a unit of work done within Jam (for greater detail on the nature of a work-report, see section 11). Such a vote is not expected to happen very often in practice (if at all), however it is an important security backstop, allowing a convenient manner of removing troublesome keys from the validator set at short notice where there is consensus over their malfunction. It also helps coordinate the ability of unfinalized chain-extensions to be reverted and replaced with an extension which does not contain some invalid work-report.

Generally speaking, judgement data will come about as a result of a dispute between validators, an off-chain process described in section 10. A judgement against a report will imply that the chain will have been reverted to immediately prior to the accumulation of that report. Placing the judgement on-chain has the effect of cancelling its accumulation. The specific strategy for chain selection is described fully in section 15.

In the case that a sufficient number of validator nodes *do* make some judgement in $\mathbf{E}_J$, then an indexed record of that judgement is placed on-chain (in ψ, the portion of state handling dispute judgements).

Having a persistent on-chain record is helpful in a number of ways. Firstly it provides a very simple means of recognizing the circumstances under which action against a validator must be taken by any higher-level validator-selection logic. Should JAM be used for a public network such as *Polkadot*, this would imply the slashing of the offending validator's stake on the staking parachain.

As mentioned, recording reports found to have a high confidence of invalidity is important to ensure that said reports are not allowed to be resubmitted. Conversely, recording reports found to be valid ensures that additional disputes cannot be raised in the future of the chain.

10.1. **State.** The judgements state includes three items, an allow-set ($\psi_{\mathbf{a}}$), a ban-set ($\psi_{\mathbf{b}}$) and a punish-set ($\psi_{\mathbf{p}}$). The allow-set contains the hashes of all work-reports which were disputed and judged to be accurate. The ban-set contains the hashes of all work-reports which were disputed and whose accuracy could not be confidently confirmed. The punish-set is a set of keys of Bandersnatch keys which were found to have guaranteed a report which was confidently found to be invalid.

$$\psi \equiv (\psi_{\mathbf{a}}, \psi_{\mathbf{b}}, \psi_{\mathbf{p}}, \psi_{\mathbf{k}}) \tag{93}$$

We store the last epoch's validator set in $\psi_{\mathbf{k}}$:

$$\psi'_{\mathbf{k}} = \begin{cases} \kappa & \text{if } \left\lfloor \frac{\tau'}{\mathsf{E}} \right\rfloor \neq \left\lfloor \frac{\tau}{\mathsf{E}} \right\rfloor \\ \psi_{\mathbf{k}} & \text{otherwise} \end{cases} \tag{94}$$

10.2. **Extrinsic.** The judgements extrinsic, $\mathbf{E}_J$ may contain one or more judgements as a compilation of signatures coming from exactly two-thirds plus one of either the active validator set (i.e. the Ed25519 keys of κ) or the previous epoch's validator set (i.e. the keys of $\psi_{\mathbf{k}}$):

$$\mathbf{E}_J \in \left[\!\left[\left(\mathbb{H}, \left[\!\left[(\{\top, \bot\}, \mathbb{N}_{\mathsf{V}}, \mathbb{F}) \right]\!\right]_{\lfloor 2/3\mathsf{V} \rfloor + 1} \right) \right]\!\right] \tag{95}$$

All signatures must be valid in terms of one of the two allowed validator key-sets. Note that the two epoch's key-sets may not be mixed! Formally:

$$\begin{aligned}&\forall (r, \mathbf{v}) \in \mathbf{E}_J, \forall (v, i, s) \in \mathbf{v} : s \in \mathbb{E}_{\mathbf{k}[i]_e}\langle \mathsf{X}_v \frown r \rangle\\ &\quad \text{where } \mathsf{X}_\top = \texttt{\$jam\_valid}\\ &\quad \text{and } \mathsf{X}_\bot = \texttt{\$jam\_invalid}\\ &\quad \mathbf{k} \in \{\kappa, \psi_{\mathbf{k}}\}\end{aligned} \tag{96}$$

Judgements must be ordered by report hash and there may be no duplicate report hashes within the extrinsic, nor amongst any past reported hashes. Formally:

$$\mathbf{E}_J = [r \,\big|\big|\, (r, \mathbf{v}) \in \mathbf{E}_J] \tag{97}$$

$$\{r \mid (r, \mathbf{v})\} \downarrow \psi_{\mathbf{a}} \cup \psi_{\mathbf{b}} \tag{98}$$

The votes of all judgements must be ordered by validator index and there may be no duplicate such indices. Formally:

$$\forall (r, \mathbf{v}) \in \mathbf{E}_J : \mathbf{v} = [i \,\big|\big|\, (v, i, s) \in \mathbf{v}] \tag{99}$$

We define $\mathbf{J}$ as the sequence of judgements introduced in the block's extrinsic (and ordered respectively), with the sequence of signatures substituted with the sum of votes over the signatures. We require this total to be exactly zero, two-thirds-plus-one or one-third-plus-one of the validator set indicating, respectively, that we are confident of the report's validity, confident of its invalidity, or lacking confidence in either. This requirement may seem somewhat arbitrary, but these happen to be the decision thresholds for our three possible actions and are acceptable since the security assumptions include the requirement that at least two-thirds-plus-one validators are live (Stewart 2018 discusses the security implications in depth).

Formally:

$$\mathbf{J} \in \llbracket (\mathbb{H}, \llbracket \mathbb{H}_B \rrbracket_{2:3}, \mathbb{N}) \rrbracket \tag{100}$$

$$\mathbf{J} = \left[\left(r, \sum_{(v,i,s) \in \mathbf{v}} v \right) \,\middle|\, (r, \mathbf{v}) \lessdot \mathbf{E}_J \right] \tag{101}$$

$$\forall (r,t) \in \mathbf{J} : t \in \{0, \lfloor 1/3 \mathsf{V} \rfloor, \lfloor 2/3 \mathsf{V} \rfloor + 1\} \tag{102}$$

Note that t is the threshold of judgements that the report is *valid*, calculated by summing Boolean values in their implicit equivalence to binary digits of the set $\mathbb{N}_2$.

We clear any work-reports judged to be non-valid from their core:

$$\forall c \in \mathbb{N}_\mathsf{C} : \rho^\dagger[c] = \begin{cases} \varnothing & \text{if } \{(\rho[c]_r, t) \in \mathbf{J}, t < \lfloor 2/3 \mathsf{V} \rfloor\} \\ \rho_c & \text{otherwise} \end{cases} \tag{103}$$

The allow-set assimilates the hashes of any reports we judge to be valid. The ban-set assimilates any other judged report-hashes. Finally, the punish-set accumulates the guarantor keys of any report judged to be invalid:

$$\psi'_\mathbf{a} \equiv \psi_\mathbf{a} \cup \{r \mid (r, \lfloor 2/3 \mathsf{V} \rfloor + 1) \in \mathbf{J}\} \tag{104}$$

$$\psi'_\mathbf{b} \equiv \psi_\mathbf{b} \cup \{r \mid (r,t) \in \mathbf{J}, t \neq \lfloor 2/3 \mathsf{V} \rfloor + 1\} \tag{105}$$

$$\psi'_\mathbf{p} \equiv \psi_\mathbf{p} \cup \{\rho[c]_g \mid (\rho[c]_r, 0) \in \mathbf{J}\} \tag{106}$$

Note that the augmented punish-set is utilized when determining κ' to nullify any validator keys which appear in the punish-list.

10.3. **Header.** The judgement marker must contain exactly the sequence of report hashes judged not as confidently valid (i.e. either controversial or invalid). Formally:

$$\mathbf{H}_j \equiv [r \mid (r,t) \lessdot \mathbf{J}, t \neq 0] \tag{107}$$

11. Reporting and Assurance

Reporting and assurance are the two on-chain processes we do to allow the results of in-core computation to make its way into the service state singleton, δ. A *work-package*, which comprises several *work items*, is transformed by validators acting as *guarantors* into its corresponding *work-report*, which similarly comprises several *work outputs* and then presented on-chain within the *guarantees* extrinsic. At this point, the work-package is erasure coded into a multitude of segments and each segment distributed to the associated validator who then attests to its

availability through an *assurance* placed on-chain. After either enough assurances or a time-out (whichever happens first), the work-report is considered *available*, and the work outputs transform the state of their associated service by virtue of accumulation, covered in section 12.

From the perspective of the work-report, therefore, the guarantee happens first and the assurance afterwards. However, from the perspective of a block's state-transition, the assurances are best processed first since each core may only have a single work-report pending its package becoming available at a time. Thus, we will first cover the transition arising from processing the availability assurances followed by the work-report guarantees. This synchroneity can be seen formally through the requirement of an intermediate state $\rho^\ddagger$, utilized later in equation 134.

11.1. **State.** The state of the reporting and availability portion of the protocol is largely contained within ρ, which tracks the work-reports which have been reported but not yet accumulated and the identities of the guarantors who reported them and the time at which it was reported. As mentioned earlier, at only one report may be assigned to a core at any given time. Formally:

$$\rho \in \left\llbracket (w \in \mathbb{W}, g \in \left\llbracket \mathbb{H}_E \right\rrbracket_{2:3}, t \in \mathbb{N}_T)? \right\rrbracket_{\mathsf{C}} \tag{108}$$

As usual, intermediate and posterior values ($\rho^\dagger$, $\rho^\ddagger$, ρ') are held under the same constraints as the prior value.

11.1.1. *Work Report.* A work-report, of the set $\mathbb{W}$, is defined as a tuple of authorizer hash and output, the refinement context, the package specification and the results of the evaluation of each of the items in the package, which is always at least one item and may be no more than I items. Formally:

$$\mathbb{W} \equiv \left(a \in \mathbb{H}, o \in \mathbb{Y}, x \in \mathbb{X}, s \in \mathbb{S}, r \in \left\llbracket \mathbb{L} \right\rrbracket_{1:\mathsf{I}}\right) \tag{109}$$

The total serialized size of a work-report may be no greater than W_R bytes:

$$\forall w \in \mathbb{W} : |\mathcal{E}(w)| \le \mathsf{W}_R \tag{110}$$

11.1.2. *Refinement Context.* A *refinement context*, denoted by the set $\mathbb{X}$, describes the context of the chain at the point that the report's corresponding work-package was evaluated. It identifies two historical blocks, the *anchor*, header hash a along

with its associated posterior state-root s and posterior BEEFY root b; and the *lookup-anchor*, header hash l and of timeslot t. Finally, it identifies the hash of an optional prerequisite work-package p. Formally:

$$\mathbb{X} \equiv \begin{pmatrix} a \in \mathbb{H}, & s \in \mathbb{H}, & b \in \mathbb{H}, \\ l \in \mathbb{H}, & t \in \mathbb{N}_T, & p \in \mathbb{H}? \end{pmatrix} \tag{111}$$

11.1.3. *Work Package Specification.* We define the set of *work-package specifications*, $\mathbb{S}$, as the tuple of the work-package's hash and serialized length together with an erasure root. Formally:

$$\mathbb{S} \equiv (h \in \mathbb{H}, l \in \mathbb{N}_L, u \in \mathbb{H}) \tag{112}$$

11.1.4. *Work Result.* We finally come to define a *work result*, $\mathbb{L}$, which is the data conduit by which services' states may be altered through the computation done within a work-package.

$$\mathbb{L} \equiv (s \in \mathbb{N}_S, c \in \mathbb{H}, l \in \mathbb{H}, g \in \mathbb{Z}_G, o \in \mathbb{Y} \cup \mathbb{J}) \tag{113}$$

Work results are a tuple comprising several items. Firstly s, the index of the service whose state is to be altered and thus whose refine code was already executed. We include the hash of the code of the service at the time of being reported c, which must be accurately predicted within the work-report according to equation 144;

Next, the hash of the payload (l) within the work item which was executed in the refine stage to give this result. This has no immediate relevance, but is something provided to the accumulation logic of the service. We follow with the gas prioritization ratio g used when determining how much gas should be allocated to execute of this item's accumulate.

Finally, there is the output or error of the execution of the code o, which may be either an octet sequence in case it was successful, or a member of the set $\mathbb{J}$, if not. This latter set is defined as the set of possible errors, formally:

$$\mathbb{J} \in \{\infty, \lightning, \texttt{BAD}, \texttt{BIG}\} \tag{114}$$

The first two are special values concerning execution of the virtual machine, ∞ denoting an out-of-gas error and $\lightning$ denoting an unexpected program termination. Of the remaining two, the first indicates that the service's code was not available for lookup in state at the posterior state of the lookup-anchor block. The second indicates that the code was available but was beyond the maximum size allowed S.

11.2. **Package Availability Assurances.** We first define $\rho^\ddagger$, the intermediate state to be utilized next in section 11.4 as well as $\mathbf{R}$, the set of available work-reports, which will we utilize later in section 12. Both require the integration of information from the assurances extrinsic $\mathbf{E}_A$.

11.2.1. *The Assurances Extrinsic.* The assurances extrinsic is a sequence of *assurance* values, at most one per validator. Each assurance is a sequence of binary values (i.e. a bitstring), one per core, together with a signature and the index of the validator who is assuring. A value of 1 (or $\top$, if interpreted as a Boolean) at any given index implies that the validator assures they are contributing to its availability.[11] Formally:

$$\mathbf{E}_A \in \llbracket (a \in \mathbb{H},\ f \in \mathbb{B}_{\mathsf{C}},\ v \in \mathbb{N}_{\mathsf{V}},\ s \in \mathbb{E}) \rrbracket_{:\mathsf{V}} \tag{115}$$

The assurances must all be anchored on the parent and ordered by validator index:

$$\forall a \in \mathbf{E}_A : a_a = \mathbf{H}_p \tag{116}$$

$$\forall i \in \{1 \dots |\mathbf{E}_A|\} : \mathbf{E}_A[i-1]_v < \mathbf{E}_A[i]_v \tag{117}$$

The signature must be one whose public key is that of the validator assuring and whose message is the serialization of the parent hash $\mathbf{H}_p$ and the aforementioned bitstring:

$$\forall a \in \mathbf{E}_A : a_s \in \mathbb{E}_{\kappa[a_v]_e}\langle \mathsf{X}_A \frown \mathcal{H}(\mathbf{H}_p, a_f) \rangle \tag{118}$$

$$\mathsf{X}_A = \texttt{\$jam\_available} \tag{119}$$

[11] This is a "soft" implication since there is no consequence on-chain if dishonestly reported. For more information on this implication see section 13.4.

A bit may only be set if the corresponding core has a report pending availability on it:

$$\forall a \in \mathbf{E}_A : \forall c \in \mathbb{N}_\mathsf{C} : a_f[c] \implies \rho^\dagger[c] \neq \varnothing \tag{120}$$

11.2.2. *Available Reports.* A work-report is said to become *available* if and only if there are a clear $2/3$ super-majority of validators who have marked its core as set within the block's assurance extrinsic. Formally, we define the series of available work-reports $\mathbf{R}$ as:

$$\mathbf{R} \equiv \left[\rho^\dagger[c]_w \;\middle|\; c \lessdot \mathbb{N}_\mathsf{C},\ \sum_{a \in \mathbf{E}_A} a_v[c] > 2/3\,\mathsf{V} \right] \tag{121}$$

This value is utilized in the definition of both δ' and $\rho^\ddagger$ which we will define presently as equivalent to $\rho^\dagger$ except for the removal of items which are now available:

$$\forall c \in \mathbb{N}_C : \rho^\ddagger[c] \equiv \begin{cases} \varnothing & \text{if } \rho[c]_w \in \mathbf{R} \\ \rho^\dagger[c] & \text{otherwise} \end{cases} \tag{122}$$

11.3. **Guarantor Assignments.** Every block, each core has some particular number of validators uniquely assigned to it assigned to guarantee work reports for it. With $\mathsf{V} = 1,023$ validators and $\mathsf{C} = 341$ cores, this results in exactly $\mathsf{V}/\mathsf{C} = 3$ validators per core. The Ed25519 keys of these validators are denoted by $\mathbf{G}$:

$$\mathbf{G} \in \left[\left[\mathbb{H}_E \right]_{\mathsf{V}/\mathsf{C}}\right]_\mathsf{C} \tag{123}$$

We determine the core to which any given validator is assigned through a shuffle using epochal entropy and a periodic rotation to help guard the security and liveness of the network. We use η_2 for the epochal entropy rather than η_1 to avoid the possibility of fork-magnification where uncertainty about chain state at the end of an epoch could give rise to two established forks before it naturally resolves.

We define the permute function P, the rotation function R and finally the guarantor assignments $\mathbf{G}$ as follows:

$$P(e, t) \equiv R\left(\mathcal{F}\left(\left[\left\lfloor \frac{\mathsf{V} \cdot i}{\mathsf{C}} \right\rfloor \middle| i \in \mathbb{N}_{\mathsf{V}}\right], e\right), \left\lfloor \frac{t \bmod \mathsf{E}}{\mathsf{R}} \right\rfloor\right) \tag{124}$$

$$R(\mathbf{c}, n) \equiv [(x + n) \bmod \mathsf{C} \mid x \in \mathbf{c}] \tag{125}$$

$$\forall c \in \mathbb{N}_C : \mathbf{G} \equiv [\kappa'_i \mid i \in \mathbb{N}_V, P(\eta'_2, \tau')_i = c] \tag{126}$$

We also define $\mathbf{G}^*$, which is equivalent to the value $\mathbf{G}$ as it would have been under the previous rotation:

$$\forall c \in \mathbb{N}_C : \mathbf{G}^* \equiv [\mathbf{k}_i \mid i \in \mathbb{N}_V, P(e, \tau' - \mathsf{R})_i = c] \tag{127}$$

$$\text{where } e = \begin{cases} (\eta'_2, \kappa) & \text{if } \left\lfloor \frac{\tau' - \mathsf{R}}{\mathsf{E}} \right\rfloor = \left\lfloor \frac{\tau'}{\mathsf{E}} \right\rfloor \\ (\eta'_3, \lambda) & \text{otherwise} \end{cases} \tag{128}$$

11.4. **Work Report Guarantees.** We begin by defining the guarantees extrinsic, $\mathbf{E}_G$, a series of *guarantees*, at most one for each core, each of which is a tuple of a core index, *work-report*, a credential a and its corresponding timeslot t. Formally:

$$\mathbf{E}_G \in \llbracket (c \in \mathbb{N}_{\mathsf{C}}, w \in \mathbb{W}, t \in \mathbb{N}_T, a \in \llbracket \mathbb{E}? \rrbracket_3) \rrbracket_{:\mathsf{C}} \tag{129}$$

The credential is itself a sequence of either two or three tuples of a signature and a validator index. The core index of each guarantee must be in ascending order:

$$\mathbf{E}_G = [i_c \setminus i \in \mathbf{E}_G] \tag{130}$$

Credentials may only have one missing signature:

$$\forall g \in \mathbf{E}_G : |\{x \in g_a : x \neq \varnothing\}| \geq 2 \tag{131}$$

The signature must be one whose public key is that of the validator identified in the credential, and whose message is the serialization of the core index and the work-report. The signing validators must be assigned to the core in question in either this block $\mathbf{G}$ if the timeslot for the guarantee is in the same rotation as this

block's timeslot, or in the most recent previous set of assignments, $\mathbf{G}^*$:

$$\forall g \in \mathbf{E}_G : \forall i \in \mathbb{N}_3, g_a[i] \neq \varnothing : \begin{cases} a_s \in \mathbb{E}_{\mathbf{k}[g_c]_i}\langle \mathsf{X}_G \frown \mathcal{H}(g_c, g_r)\rangle \\ \text{where } \mathbf{k} = \begin{cases} \mathbf{G} & \text{if } \left\lfloor \frac{\tau'}{\mathsf{R}} \right\rfloor = \left\lfloor \frac{g_t}{\mathsf{R}} \right\rfloor \\ \mathbf{G}^* & \text{otherwise} \end{cases} \end{cases} \tag{132}$$

$$\mathsf{X}_G = \texttt{\$jam\_guarantee} \tag{133}$$

No reports may be placed on cores with a report pending availability on it unless it has timed out. In the latter case, $\mathsf{U} = 5$ slots must have elapsed after the report was made. A report is invalid if the authorizer hash is not present in the authorizer pool of the core on which the work is reported. Formally:

$$\forall g \in \mathbf{E}_G : \begin{cases} \rho^\ddagger[g_c] = \varnothing \vee \mathbf{H}_t \geq \rho^\ddagger[g_c]_t + \mathsf{U} \ , \\ g_a \in \alpha[g_c] \end{cases} \tag{134}$$

We denote $\mathbf{w}$ to be the set of work-reports in the present extrinsic $\mathbf{E}$:

$$\text{let } \mathbf{w} = \{g_w \mid g \in \mathbf{E}_G\} \tag{135}$$

We specify the maximum total accumulation gas requirement a work-report may imply as G_A, and we require the sum of all services' minimum gas requirements to be no greater than this:

$$\forall w \in \mathbf{w} : \sum_{s \in (w_r)_s} \delta[s]_m \ \leq \ \mathsf{G}_A \tag{136}$$

11.4.1. *Contextual Validity of Reports.* For convenience, we define two equivalences $\mathbf{x}$ and $\mathbf{p}$ to be, respectively, the set of all contexts and work-package hashes within the extrinsic:

$$\text{let } \mathbf{x} \equiv \{w_x \mid w \in \mathbf{w}\} \ , \quad \mathbf{p} \equiv \{(w_s)_h \mid w \in \mathbf{w}\} \tag{137}$$

There must be no duplicate work-package hashes (i.e. two work-reports of the same package). Therefore, we require the cardinality of $\mathbf{p}$ to be the length of the

work-report sequence $\mathbf{w}$:

$$|\mathbf{p}| = |\mathbf{w}| \tag{138}$$

We require that the anchor block be within the last H blocks and that its details be correct by ensuring that it appears within our most recent blocks β:

$$\forall x \in \mathbf{x} : \exists y \in \beta : x_a = y_h \wedge x_s = y_s \wedge x_b = \mathcal{H}_K(\mathcal{E}_M(y_\mathbf{b})) \tag{139}$$

We require that each lookup-anchor block be within the last L timeslots:

$$\forall x \in \mathbf{x} : \; x_t \geq \mathbf{H}_t - \mathsf{L} \tag{140}$$

We also require that we have a record of it; this is one of the few conditions which cannot be checked purely with on-chain state and must be checked by virtue of retaining the series of the last L headers as the ancestor set $\mathbf{A}$. Since it is determined through the header chain, it is still deterministic and calculable. Formally:

$$\forall x \in \mathbf{x} : \; \exists h \in \mathbf{A} : h_t = x_t \wedge \mathcal{H}(h) = x_h \tag{141}$$

We require that the work-package of the report not be the work-package of some other report made in the past. Since the work-package implies the anchor block, and the anchor block is limited to the most recent blocks, we need only ensure that the work-package not appear in our recent history:

$$\forall p \in \mathbf{p}, \forall x \in \beta : p \notin x_\mathbf{p} \tag{142}$$

We require that the prerequisite work-package, if present, be either in the extrinsic or in our recent history:

$$\begin{aligned} &\forall w \in \mathbf{w}, (w_x)_p \neq \varnothing : \\ &\quad (w_x)_p \in \mathbf{p} \cup \{x \mid x \in b_\mathbf{p}, b \in \beta\} \end{aligned} \tag{143}$$

We require that all work results within the extrinsic predicted the correct code hash for their corresponding service:

$$\forall w \in \mathbf{w}, \forall r \in w_r : r_c = \delta[r_s]_c \tag{144}$$

11.5. **Transitioning for Reports.** We define ρ' as being equivalent to $\rho^\ddagger$, except where the extrinsic replaced an entry. In the case an entry is replaced, the new value includes the present time τ' allowing for the value may be replaced without respect to its availability once sufficient time has elapsed (see equation 134).

$$\forall c \in \mathbb{N}_\mathsf{C} : \rho'[c] \equiv \begin{cases} (w, g \colon G(a), t \colon \tau') & \text{if } \exists (c, w, a) \in \mathbf{E}_G \\ \rho^\ddagger[c] & \text{otherwise} \end{cases} \tag{145}$$
$$\text{where } G(a) \equiv \{\kappa[v]_e \mid (s, v) \in a\}$$

This concludes the section on reporting and assurance. We now have a complete definition of ρ' together with $\mathbf{R}$ to be utilized in section 12, describing the portion of the state transition happening once a work-report is guaranteed and made available.

12. Accumulation

Accumulation may be defined as some function whose arguments are $\mathbf{R}$ and δ together with selected portions of (at times partially transitioned) state and which yields the posterior service state δ' together with additional state elements ι', φ' and χ'.

The proposition of accumulation is in fact quite simple: we merely wish to execute the *Accumulate* logic of the service code of each of the services which has at least one work output, passing to it the work outputs and useful contextual information. However, there are three main complications. Firstly, we must define the execution environment of this logic and in particular the host functions available to it. Secondly, we must define the amount of gas to be allowed for each service's execution. Finally, we must determine the nature of transfers within Accumulate which, as we will see, leads to the need for a second entry-point, *on-transfer*.

12.1. **Preimage Integration.** Prior to accumulation, we must first integrate all preimages provided in the lookup extrinsic. The lookup extrinsic is a sequence of pairs of service indices and data. These pairs must be ordered and without duplicates (equation 147 requires this). The data must have been solicited by a service but not yet be provided. Formally:

$$\mathbf{E}_P \in \llbracket (\mathbb{N}_S, \mathbb{Y}) \rrbracket \tag{146}$$

$$\mathbf{E}_P = [i \,\{\!\{\, i \in \mathbf{E}_P]\tag{147}$$

$$\forall (s, \mathbf{p}) \in \mathbf{E}_P : \begin{cases} \mathcal{K}(\delta[s]_{\mathbf{p}}) \neq \mathcal{H}(\mathbf{p}) \ , \\ \delta[s]_{\mathbf{l}}[(\mathcal{H}(\mathbf{p}), |\mathbf{p}|)] = [] \end{cases}\tag{148}$$

We define $\delta^\dagger$ as the state after the integration of the preimages:

$$\delta^\dagger = \delta \text{ ex. } \forall (s, \mathbf{p}) \in \mathbf{E}_P : \begin{cases} \delta^\dagger[s]_{\mathbf{p}}[\mathcal{H}(\mathbf{p})] = \mathbf{p} \\ \delta^\dagger[s]_{\mathbf{l}}[\mathcal{H}(\mathbf{p}), |\mathbf{p}|] = [\tau'] \end{cases}\tag{149}$$

12.2. **Gas Accounting.** We define $\mathbf{S}$, the set of all services which will be accumulated in this block; this is all services which have at least one work output within $\mathbf{R}$, together with all privileged services, χ. Formally:

$$\mathbf{S} \equiv \{r_s \mid w \in \mathbf{R}, r \in w_r\} \cup \{\chi_m, \chi_a, \chi_v\}\tag{150}$$

We calculate the gas attributable for each service as the sum of each of the service's work outputs' share of their report's elective accumulation gas together with the subtotal of minimum gas requirements:

$$G: \begin{cases} \mathbb{N}_S \to \mathbb{Z}_G \\ \quad s \mapsto \sum_{w \in \mathbf{R}} \sum_{r \in w_r, r_s = s} \delta^\dagger[s]_g + \left\lfloor r_g \cdot \frac{\mathsf{G}_A - \sum_{r \in w_r} \delta^\dagger[r_s]_g}{\sum_{r \in w_r} r_g} \right\rfloor \end{cases}\tag{151}$$

12.3. **Wrangling.** We finally define the results which will be given as an operand into the accumulate function for each service in $\mathbf{S}$. This is a sequence of operand tuples $\mathbb{O}$, one sequence for each service in $\mathbf{S}$. Each sequence contains one element per work output (or error) to be accumulated for that service, together with said work output's payload hash, package hash and authorization output. The tuples are sequenced in the same order as they appear in $\mathbf{R}$. Formally:

$$\mathbb{O} \equiv (o \in \mathbb{Y} \cup \mathbb{J}, l \in \mathbb{H}, k \in \mathbb{H}, a \in \mathbb{Y})\tag{152}$$

$$M \colon \begin{cases} \mathbb{N}_S \to [\![\mathbb{O}]\!] \\ \quad s \mapsto \left[\begin{pmatrix} o \colon r_o, & l \colon r_p, \\ a \colon w_o, & k \colon (w_s)_h \end{pmatrix} \middle| \begin{array}{l} w \in \mathbf{R}, \\ r \in w_r, \\ r_s = s \end{array} \right] \end{cases} \tag{153}$$

12.4. **Invocation.** Within this section, we define A, the function which conducts the accumulation of a single service. Formally speaking, A assumes omnipresence of timeslot $\mathbf{H}_t$ and some prior state components $\delta^\dagger$, ν, $\mathbf{R_d}$, and takes as specific arguments the service index $s \in \mathbf{S}$ (from which it may derive the wrangled results $M(s)$ and gas limit $G(s)$) and yields values for $\boldsymbol{\delta}^\dagger[s]$ and staging assignments into φ, ι together with a series of lookup solicitations/forgets, a series of deferred transfers and $\mathbf{C}$ mapping from service index to Beefy commitment hashes.

We first denote the set of deferred transfers as $\mathbb{T}$, noting that a transfer includes a memo component m of 64 octets, together with the service index of the sender s, the service index of the receiver d, the amount of tokens to be transferred a and the gas limit g for the transfer. Formally:

$$\mathbb{T} \equiv \left(s \in \mathbb{N}_S,\, d \in \mathbb{N}_S,\, a \in \mathbb{N}_B,\, m \in \mathbb{Y}_{\mathsf{M}},\, g \in \mathbb{Z}_G\right) \tag{154}$$

We may then define A, the mapping from the index of accumulated services to the various components in terms of which we will be imminently defining our posterior state:

$$A \colon \begin{cases} \mathbb{N}_S \to \left(\begin{array}{l} \mathbf{s} \in \mathbb{A}^{?}, \quad \mathbf{v} \in \mathbb{K}_{\mathsf{V}}, \quad \mathbf{t} \in [\![\mathbb{T}]\!], \quad r \in \mathbb{H}, \\ \mathbf{c} \in \left[\!\left[[\![\mathbb{H}]\!]_{\mathsf{Q}} \right]\!\right]_{\mathsf{C}}, \qquad \mathbf{n} \in \mathbb{D}\langle \mathbb{N}_S \to \mathbb{A} \rangle, \\ p \in \left(m \in \mathbb{N}_S,\, a \in \mathbb{N}_S,\, v \in \mathbb{N}_S\right) \end{array} \right) \\ \quad s \mapsto \Psi_A(\delta^\dagger, s, M(s), G(s)) \end{cases} \tag{155}$$

As can be seen plainly, our accumulation mapping A combines portions of the prior state into arguments for a virtual-machine invocation. Specifically the service accounts $\delta^\dagger$ together with the index of the service in question s and its wrangled refine-results $M(s)$ and gas limit $G(s)$ are arranged to create the arguments for Ψ_A, itself using a virtual-machine invocation as defined in appendix B.4.

The Beefy commitment map is a function mapping all accumulated services to their accumulation result (the r component of the result of A). This is utilized in determining the accumulation-result tree root for the present block, useful for the Beefy protocol:

$$\mathbf{C} \equiv \{(s, A(s)_r) \mid s \in \mathbf{S}, A(s)_r \neq \varnothing\} \tag{156}$$

Given our mapping A, which may be calculated exhaustively from the vm invocations of each accumulated service $\mathbf{S}$, we may define the posterior state δ', χ', φ' and ι' as the result of integrating A into our state.

12.4.1. *Privileged Transitions.* The staging core assignments, and validator keys and privileged service set are each altered based on the effects of the accumulation of each of the three privileged services:

$$\chi' \equiv A(\chi_m)_p \ , \quad \varphi' \equiv A(\chi_a)_{\mathbf{c}} \ , \quad \iota' \equiv A(\chi_v)_{\mathbf{v}} \tag{157}$$

12.4.2. *Service Account Transitions.* Finally, we integrate all changes to the service accounts into state.

We note that all newly added service indices, defined as $\mathcal{K}(A(s)_{\mathbf{n}})$ for any accumulated service s, must not conflict with the indices of existing services or newly added services. This should never happen, since new indices are explicitly selected to avoid such conflicts, but in the unlikely event it happens, the block would be invalid. Formally:

$$\begin{aligned} &\forall s \in \mathbf{S} : \mathcal{K}(A(s)_{\mathbf{n}}) \cap \mathcal{K}(\delta^\dagger) = \varnothing, \\ &\forall t \in \mathbf{S} \setminus \{s\} : \mathcal{K}(A(s)_{\mathbf{n}}) \cap \mathcal{K}(A(t)_{\mathbf{n}}) = \varnothing \end{aligned} \tag{158}$$

We first define $\delta^\ddagger$, an intermediate state after main accumulation but before the transfers have been credited and handled:

$$\mathcal{K}(\delta^\ddagger) \equiv \left(\mathcal{K}(\delta^\dagger) \cup \bigcup_{s \in \mathbf{S}} \mathcal{K}(A(s)_\mathbf{n})\right) \setminus \left\{ s \;\middle|\; \begin{array}{r} s \in \mathbf{S}, \\ s_\mathbf{s} = \varnothing \end{array} \right\}$$

$$\delta^\ddagger[s] \equiv \begin{cases} A(s)_\mathbf{s} & \text{if } s \in \mathbf{S} \\ A(t)_\mathbf{n}[s] & \text{if } \exists! t : t \in \mathbf{S}, s \in \mathcal{K}(A(t)_\mathbf{n}) \\ \delta^\dagger[s] & \text{otherwise} \end{cases} \tag{159}$$

We denote $R(s)$ the sequence of transfers received by a given service of index s, in order of them being sent from services of ascending index. (If some service s received no transfers or simply does not exist then $R(s)$ would be validly defined as the empty sequence.) Formally:

$$R: \begin{cases} \mathbb{N}_S \to \llbracket \mathbb{T} \rrbracket \\ \quad d \mapsto [\, t \mid s \lhd \mathbf{S},\ t \lhd A(s)_\mathbf{t},\ t_d = d \,] \end{cases} \tag{160}$$

The posterior state δ' may then be defined as the intermediate state with all the deferred effects of the transfers applied:

$$\delta' = \{ s \mapsto \Psi_T(\delta^\ddagger, a, R(a)) \mid (s \mapsto a) \in \delta^\ddagger \} \tag{161}$$

Note that Ψ_T is defined in appendix B.5 such that it results in $\delta^\ddagger[d]$, i.e. no difference to the account's intermediate state, if $R(d) = []$, i.e. said account received no transfers.

13. Work Packages and Work Reports

13.1. **Honest Behavior.** We have so far specified how to recognize blocks for a correctly transitioning JAM blockchain. Through defining the state transition function and a state Merklization function, we have also defined how to recognize a valid header. While it is not especially difficult to understand how a new block may be authored for any node which controls a key which would allow the creation of the two signatures in the header, nor indeed to fill in the other header fields, readers will note that the contents of the extrinsic remain unclear.

We define not only correct behavior through the creation of correct blocks but also *honest behavior*, which involves the node taking part in several *off-chain* activities. This does have analogous aspects within *YP* Ethereum, though it is not mentioned so explicitly in said document: the creation of blocks along with the gossiping and inclusion of transactions within those blocks would all count as off-chain activities for which honest behavior is helpful. In JAM's case, honest behavior is well-defined and expected of at least $2/3$ of validators.

Beyond the production of blocks, incentivized honest behavior includes:

- the guaranteeing and reporting of work-packages, along with chunking and distribution of both the chunks and the work-package itself, discussed in section 13.3;
- assuring the availability of work-packages after being in receipt of their data;
- making and submitting judgements on the correctness of work-reports;
- determining which work-reports to audit, fetching and auditing them, and creating and distributing an adverse judgement to other nodes based on the outcome of the audit;
- submitting the correct amount of work seen being done by other validators, discussed in section 16.

We begin with the first of these, the guaranteeing of work-packages.

13.2. **Packages and Items.** We begin by defining a *work-package*, of set $\mathbb{P}$, and its constituent *work item*s, of set $\mathbb{I}$. A work-package includes a simple blob acting as an authorization token $\mathbf{j}$, a service identifier for where authorization code is hosted h, an authorization code hash c and a parameterization blob $\mathbf{p}$, a context $\mathbf{x}$ and a sequence of work items limited in size $\mathbf{i}$:

$$\mathbb{P} \in (\mathbf{j} \in \mathbb{Y}, h \in \mathbb{N}_S, c \in \mathbb{H}, \mathbf{p} \in \mathbb{Y}, \mathbf{x} \in \mathbb{X}, \mathbf{i} \in \mathbb{I}_{1:I}) \tag{162}$$

We limit the encoded size of work-packages to a little over 6MB in order to allow for 1MB/s/core data throughput:

$$|\mathcal{E}(\mathbb{P})| \le \mathsf{W}_P \tag{163}$$

$$\mathsf{W}_P = 6 \cdot 2^{20} + 2^{16} \tag{164}$$

A work item includes the identifier of the service to which it relates s, the code hash of the service at the time of reporting, and whose preimage must be available from the perspective of the lookup anchor block c, a payload blob y, and a gas limit g:

$$\mathbb{I} \in (s \in \mathbb{N}_\mathsf{S}, c \in \mathbb{H}, \mathbf{y} \in \mathbb{Y}, g \in \mathbb{N}_G) \tag{165}$$

We define the work-package's implied authorizer as $\mathbf{p}_a$, the hash of the concatenation of the authorization code and the parameterization. We define the authorization code as $\mathbf{p_c}$ and require that it be available at the time of the lookup anchor block from the historical lookup of service h. Formally:

$$\forall \mathbf{p} \in \mathbb{P} : \begin{cases} \mathbf{p}_a \equiv \mathcal{H}(\mathbf{p_c} \frown \mathbf{p_p}) \\ \mathbf{p_c} \equiv H(\delta[\mathbf{p}_h], (\mathbf{p}_x)_t, \mathbf{p}_c) \\ \mathbf{p_c} \in \mathbb{Y} \end{cases} \tag{166}$$

(Λ is the historical lookup function defined in equation 90.)

We now come to the work result computation function Ξ. This forms the basis for all utilization of cores on JAM. It operates on some work-package $\mathbf{p}$ for some nominated core c and results in either an error ∇ or the work result, which is deterministic and, thanks to the historical lookup functionality, can be evaluated by any node which has a recently finalized chain for up to 24 epochs after the lookup-anchor block.

Formally:

$$\Xi \colon \begin{cases} (\mathbb{P}, \mathbb{N}_C) \to \mathbb{W} \\ \quad (\mathbf{p}, c) \mapsto \begin{cases} \nabla & \text{if } \mathbf{o} \notin \mathbb{Y} \\ (a \colon \mathbf{p}_a, \mathbf{o}, x \colon \mathbf{p}_x, s, r) & \text{otherwise} \end{cases} \end{cases} \tag{167}$$

where:

$$\begin{aligned} \mathbf{o} &= \Psi_I(\mathbf{p}, c) \\ s &= (h, l \colon |\mathcal{E}(\mathbf{p})|, u \colon \mathcal{M}_2([\mathcal{H}(x) \mid x \lessdot \mathcal{C}(\mathcal{P}_{2^{11}}(\mathcal{E}(\mathbf{p})))])) \end{aligned}$$

$$r = [\Psi_R(c, g, s, h, \mathbf{y}, \mathbf{p_x}, \mathbf{p}_a, \mathbf{o}) \mid (s, c, \mathbf{y}, g) \in \mathbf{p_i}]$$
$$h = \mathcal{H}(\mathbf{p})$$

And $\mathcal{P}$ is the zero-padding function to take an octet array to some multiple of n in length:

$$\mathcal{P}_{n \in \mathbb{N}_{1:}} : \begin{cases} \mathbb{Y} \to \mathbb{Y}_{k \cdot n} \\ \mathbf{x} \mapsto \mathbf{x} \frown [0, 0, \ldots]_{((|x|+n-1) \bmod n)+1 \ldots n} \end{cases} \tag{168}$$

We define the binary Merklization function $\mathcal{M}_2$ in equation 280. Note that $\mathcal{C}$ represents the erasure-coding function for the chunks and is defined in appendix H.

Validators are incentivized to distribute each work-package chunk to each other validator, since they are not paid for guaranteeing unless a work-report is considered to be *available*. Given our work-package $\mathbf{p}$, we should therefore send chunk $\mathcal{C}(\mathcal{E}(\mathbf{p}))_v$ to each validator whose keys are κ_v. In the case of a coming epoch change, they may also maximize expected reward by distributing to the new validator set (and thus also send the chunk to $(\gamma_\mathbf{k})_v$).

We will see this function utilized in the next sections, for guaranteeing, auditing and judging.

13.3. **Guaranteeing.** Guaranteeing work-packages involves the creation and distribution of a corresponding *work-report* which requires certain conditions to be met. Along with the report, a signature demonstrating the validator's commitment to its correctness is needed. With two guarantor signatures, the work-report may be distributed to the forthcoming JAM chain block author in order to be used in the $\mathbf{E}_G$, which leads to a reward for the guarantors.

We presume that in a public system, validators will be punished severely if they malfunction and commit to a report which does not faithfully represent the result of Ξ applied on a work-package. Overall, the process is:

(1) Evaluation of the work-package's authorization, and cross-referencing against the authorization pool in the most recent JAM chain state.

(2) Chunking of the work-package report according to the erasure codec.

(3) Creation and publication of a work-package report.

(4) Distributing the chunks and package as needed to other nodes.

For any work-package $\mathbf{p}$ we are in receipt of, we may determine the work result, if any, it corresponds to for the core c that we are assigned to. When JAM chain state is needed, we always utilize the chain state of the most recent block.

For any guarantor of index v assigned to core c and a work-package $\mathbf{p}$, we define the work result $\mathbf{r}$ simply as:

$$\mathbf{r} = \Xi(\mathbf{p}, c) \tag{169}$$

Such guarantors may safely create and distribute the payload (s, v). The component s may be created according to equation 132; specifically it is a signature using the validator's registered Ed25519 key on a payload l:

$$l = \mathcal{H}(c, \mathbf{r}) \tag{170}$$

To maximize profit, the guarantor should require the work result meets all expectations which are in place during the guarantee extrinsic described in section 11.4. This includes contextual validity, inclusion of the authorization in the authorization pool, and ensuring total gas is at most G_A. No doing so does not result in punishment, but will prevent the block author from including the package and so reduces rewards.

Advanced nodes may maximize the likelihood that their reports will be includable on-chain by attempting to predict the state of the chain at the time that the report will get to the block author. Naive nodes may simply use the current chain head when verifying the work-report. To minimize work done, nodes should make all such evaluations *prior* to evaluating the Ψ_R function to calculate the report's work results.

Once evaluated as a reasonable work-package to guarantee, guarantors should maximize the chance that their work is not wasted by attempting to form consensus over the core. To achieve this they should send the work-package to any other guarantors on the same core which they do not believe already know of it.

In order to minimize the work for block authors and thus maximize expected profits, guarantors should attempt to construct their core's next guarantee extrinsic from the work-report, core index and set of attestations including their own and as many others as possible.

In order to minimize the chance of any block authors disregarding the guarantor for anti-spam measures, guarantors should sign an average of no more than two work-reports per timeslot.

13.4. **Availability Assurance.** Validators should issue signed statements, called *assurances*, when they are in possession of their corresponding erasure-coding chunk of the work-package for any corresponding work-reports which are currently pending availability.

The correct erasure-coding chunk can be determined through a proof using the commitment to the work-package chunks Merkle root specified in the work-report.

13.5. **Auditing and Judging.** The auditing and judging system is theoretically equivalent to that in ELVES, introduced by Stewart 2018. For a full security analysis of the mechanism, see this work. The main differences are in terminology, whereby the terms *backing* and *approval* there refer to our guaranteeing and auditing, respectively.

13.5.1. *Overview.* The auditing process involves each node requiring themselves to fetch, evaluate and issue judgement on a random but deterministic set of work-reports from each JAM chain block in which the work-report becomes available (i.e. from $\mathbf{R}$). Prior to any evaluation, a node declares and proves its requirement. At specific common junctures in time thereafter the set of work-reports which a node requires itself to evaluate from each block's $\mathbf{R}$ may be enlarged if any declared intentions are not matched by a positive judgement in a reasonable time or in the event of a negative judgement being seen. These enlargement events are called tranches.

If all declared intentions for a work-report are matched by a positive judgement at any given juncture, then the work-report is considered *audited*. Once all of any given block's newly available work-reports are audited, then we consider the block to be *audited*. One prerequisite of a node finalizing a block is for it to view the block as audited. Note that while there will be eventual consensus on whether a block is audited, there may not be consensus at the time that the block gets finalized. This does not affect the crypto-economic guarantees of this system.

In regular operation, no negative judgements will ultimately be found for a work-report, and there will be no direct consequences of the auditing stage. In the unlikely event that a negative judgement is found, then one of several things

happens; if there are still more than $^2/_3\mathsf{V}$ positive judgements, then validators issuing negative judgements may receive a punishment for time-wasting. If there are greater than $^1/_3\mathsf{V}$ negative judgements, then the block which includes the work-report is ban-listed. It and all its descendants are disregarded and may not be built on. In all cases, once there are enough votes, a judgement extrinsic can be constructed by a block author and placed on-chain to denote the outcome. See section 10 for details on this.

All announcements and judgements are published to all validators along with metadata describing the signed material. On receipt of sure data, validators are expected to update their perspective accordingly (later defined as J and A).

13.5.2. *Auditing Specifics.* Each validator shall perform auditing duties on each valid block received. Since we are entering off-chain logic, and we cannot assume consensus, we henceforth now consider ourselves a specific validator of index v and assume ourselves focused on some block $\mathbf{B}$ with other terms corresponding, so σ' is said block's posterior state, $\mathbf{H}$ is its header &c. Practically, all considerations must be replicated for all blocks and multiple blocks' considerations may be underway simultaneously.

We define the sequence of work-reports which we may be required to audit as $\mathbf{Q}$, a sequence of length equal to the number of cores, which functions as a mapping of core index to a work-report pending which has just become available, or $\varnothing$ if no report became available on the core. Formally:

$$\mathbf{Q} \in \llbracket \mathbb{W}? \rrbracket_{\mathsf{C}} \tag{171}$$

$$\mathbf{Q} \equiv \left[\begin{cases} \rho[c]_w & \text{if } \rho[c]_w \in \mathbf{R} \\ \varnothing & \text{otherwise} \end{cases} \middle| \, c \in \mathbb{N}_{\mathsf{C}} \right] \tag{172}$$

We define our initial audit tranche in terms of a verifiable random quantity s_0 created specifically for it:

$$s_0 \in \mathbb{F}^{[]}_{\kappa[v]_b}\langle \mathsf{X}_U \frown \mathcal{Y}(\mathbf{H}_v) \rangle \tag{173}$$

$$\mathsf{X}_U = \texttt{\$jam\_audit} \tag{174}$$

We may then define $\mathbf{a}_0$ as the non-empty items to audit through a verifiably random selection of ten cores:

$$\mathbf{a}_0 = \{(c, w) \mid (c, w) \in \mathbf{p}_{\dots+10}, w \neq \varnothing\} \tag{175}$$

$$\text{where } \mathbf{p} = \mathcal{F}([[(c, \mathbf{Q}_c) \mid c \in \mathbb{N}_\mathsf{C}], r) \tag{176}$$

$$\text{and } r = \mathcal{Y}(s_0) \tag{177}$$

Every $\mathsf{A} = 8$ seconds following a new time slot, a new tranche begins, and we may determine that additional cores warrant an audit from us. Such items are defined as $\mathbf{a}_n$ where n is the current tranche. Formally:

$$\text{let } n = \left\lfloor \frac{\mathcal{T} - \mathsf{P} \cdot \tau'}{\mathsf{A}} \right\rfloor \tag{178}$$

New tranches may contain items from $\mathbf{Q}$ stemming from one of two reasons: either a negative judgement has been received; or the number of judgements from the previous tranche is less than the number of announcements from said tranche. In the first case, the validator is always required to issue a judgement on the work-report. In the second case, a new special-purpose VRF must be constructed to determine if an audit and judgement is warranted from us.

In all cases, we publish a signed statement of which of the cores we believe we are required to audit (an *announcement*) together with evidence of the VRF signature to select them and the other validators' announcements from the previous tranche unmatched with a judgement in order that all other validators are capable of verifying the announcement. *Publication of an announcement should be taken as a contract to complete the audit regardless of any future information.*

Formally, for each tranche n we ensure the announcement statement is published and distributed to all other validators along with our validator index v, evidence s_n and all signed data. Validator's announcement statements must be in the set:

$$\mathbb{E}_{\kappa[v]_e}\langle \mathsf{X}_I \mathbin{+\!\!+} n \frown \mathcal{E}([\mathcal{E}_2(c) \frown \mathbf{H}(w) \mid (c, w) \in \mathbf{a}_0])\rangle \tag{179}$$

$$\mathsf{X}_I = \texttt{\$jam\_announce} \tag{180}$$

We define A_n as our perception of which validator is required to audit each of the work-reports (identified by their associated core) at tranche n. This comes

from each other validators' announcements (defined above). It cannot be correctly evaluated until n is current. We have absolute knowledge about our own audit requirements.

$$A_n : \mathbb{W} \to \wp(\mathbb{N}_\mathbb{V}) \tag{181}$$

$$\forall (c, w) \in \mathbf{a}_0 : v \in q_0(w) \tag{182}$$

We further define $J_\top$ and $J_\bot$ to be the validator indices who we know to have made respectively, positive and negative, judgements mapped from each work-report's core. We don't care from which tranche a judgement is made.

$$J_{\{\bot,\top\}} : \mathbb{W} \to \wp(\mathbb{N}_\mathbb{V}) \tag{183}$$

We are able to define $\mathbf{a}_n$ for tranches beyond the first on the basis of the number of validators who we know are required to conduct an audit yet from whom we have not yet seen a judgement. It is possible that the late arrival of information alters $\mathbf{a}_n$ and nodes should reevaluate and act accordingly should this happen.

We can thus define $\mathbf{a}_n$ beyond the initial tranche through a new VRF which acts upon the set of *no-show* validators.

$$\forall n > 0 :$$

$$s_n(w) \in \mathbb{F}^{[]}_{\kappa[v]_b}\langle \mathsf{X}_U \frown \mathcal{Y}(\mathbf{H}_v) \frown \mathcal{H}(w) +\!\!+ n \rangle \tag{184}$$

$$\mathbf{a}_n \equiv \{ w \in \mathbf{Q} \mid \tfrac{\mathsf{F}}{256\mathbb{V}} \mathcal{Y}(s_n(w))_0 < |A_{n-1}(w) \setminus J_\top(w)| \} \tag{185}$$

We define our bias factor $\mathsf{F} = 2$, which is the expected number of validators which will be required to issue a judgement for a work-report given a single no-show in the tranche before. Modeling by Stewart 2018 shows that this is optimal.

Later audits must be announced in a similar fashion to the first. If audit requirements lesson on the receipt of new information (i.e. a positive judgement being returned for a previous *no-show*), then any audits already announced are completed and judgements published. If audit requirements raise on the receipt of new information (i.e. an additional announcement being found without an accompanying judgement), then we announce the additional audit(s) we will undertake.

As n increases with the passage of time $\mathbf{a}_n$ becomes known and defines our auditing responsibilities. We must attempt to reconstruct all work-packages corresponding to each work-report we must audit. This may be done through requesting erasure-coded chunks from one-third of the validators. It may also be short-cutted through asking a third-party (e.g. an original guarantor) for a reverse-hash lookup using the work-package hash in the work-report's package specification.

Thus, for any such work-report w we are assured we will be able to fetch some candidate work-package encoding $F(w)$ which comes either from reconstructing erasure-coded chunks verified through the erasure coding's Merkle root, or alternatively from the preimage of the work-package hash. We decode this candidate blob into a work-package and attempt to reproduce the report on the core to give e_n, a mapping from cores to evaluations:

$$\forall (c, w) \in \mathbf{a}_n : e_n(w) \Leftrightarrow \begin{cases} w = \Xi(p, c) & \text{if } \exists p \in \mathbb{P} : \mathcal{E}(p) = F(w) \\ \bot & \text{otherwise} \end{cases} \tag{186}$$

Note that a failure to decode implies an invalid work-report.

From this mapping the validator issues a set of judgements $\mathbf{j}_n$:

$$\mathbf{j}_n = \{\mathcal{S}_{\kappa[v]_e}(\mathsf{X}_{e(w)} \frown \mathcal{H}(w)) \mid (c, w) \in \mathbf{a}_n\} \tag{187}$$

All judgements $\mathbf{j}_*$ should be published to other validators in order that they build their view of J and in the case of a negative judgement arising, can form an extrinsic for $\mathbf{E}_J$.

We consider a work-report as audited under two circumstances. Either, when it has no negative judgements and there exists some tranche in which we see a positive judgement from all validators who we believe are required to audit it; or when we see positive judgements for it from greater than two-thirds of the validator set.

$$U(w) \Leftrightarrow \bigvee \begin{cases} J_\bot(w) = \varnothing \wedge \exists n : A_n(w) \subset J_\top(w) \\ |J_\top(w)| > 2/3\mathsf{V} \end{cases} \tag{188}$$

Our block $\mathbf{B}$ may be considered audited, a condition denoted $\mathbf{U}$, when all the work-reports which were made available are considered audited. Formally:

$$\mathbf{U} \Leftrightarrow \forall w \in \mathbf{R} : U(w) \tag{189}$$

For any block we must judge it to be audited (i.e. $\mathbf{U} = \top$) before we vote for the block to be finalized in Grandpa. See section 15 for more information here.

Furthermore, we pointedly disregard chains which include the accumulation of a report which we know at least 1/3 of validators judge as being invalid. Any chains including such a block are not eligible for authoring on. The *best block*, i.e. that on which we build new blocks, is defined as the chain with the most regular Safrole blocks which does *not* contain any such disregarded block. Implementation-wise, this may require reversion to an earlier head or alternative fork.

As a block author, we include a judgement extrinsic which collects judgement signatures together and reports them on-chain. In the case of a non-valid judgement (i.e. one which is not two-thirds-plus-one of judgements confirming validity) then this extrinsic will be introduced in a block in which accumulation of the non-valid work-report is about to take place. The non-valid judgement extrinsic removes it from the pending work-reports, ρ. Refer to section 10 for more details on this.

14. Beefy Distribution

For each finalized block $\mathbf{B}$ which a validator imports, said validator shall make a bls signature on the bls12-381 curve, as defined by Hopwood et al. 2020, affirming the Keccak hash of the block's most recent Beefy mmr. This should be published and distributed freely, along with the signed material. These signatures may be aggregated in order to provide concise proofs of finality to third-party systems. The signing and aggregation mechanism is defined fully by Jeff Burdges et al. 2022.

Formally, let $\mathbf{F}_v$ be the signed commitment of validator index v which will be published:

$$\mathbf{F}_v \equiv \mathcal{S}_{\kappa_v}(\mathsf{X}_B \frown \mathcal{H}_K(\mathcal{E}_M(\text{last}(\beta)_\mathbf{b}))) \tag{190}$$

$$\mathsf{X}_B = \texttt{\$jam\_beefy} \tag{191}$$

15. Grandpa and the Best Chain

Nodes take part in the Grandpa protocol as defined by Stewart and Kokoris-Kogia 2020.

We define the latest finalized block as $\mathbf{B}^{\natural}$. All associated terms concerning block and state are similarly superscripted. We consider the *best block*, $\mathbf{B}^{\flat}$ to be that which is drawn from the set of acceptable blocks of the following criteria:

- Has the finalized block as an ancestor.
- Contains no unfinalized blocks where we see an equivocation (two valid blocks at the same timeslot).
- Is considered audited.

Formally:

$$\mathbf{A}(\mathbf{H}^{\flat}) \ni \mathbf{H}^{\natural} \tag{192}$$

$$\mathbf{U}^{\flat} \equiv \top \tag{193}$$

$$\nexists\, \mathbf{H}^A, \mathbf{H}^B : \bigwedge \begin{cases} \mathbf{H}^A \neq \mathbf{H}^B \\ \mathbf{H}^A_T = \mathbf{H}^B_T \\ \mathbf{H}^A \in \mathbf{A}(\mathbf{H}^{\flat}) \\ \mathbf{H}^A \notin \mathbf{A}(\mathbf{H}^{\natural}) \end{cases} \tag{194}$$

Of these acceptable blocks, that which contains the most ancestor blocks whose author used a seal-key ticket, rather than a fallback key should be selected as the best head, and thus the chain on which the participant should make Grandpa votes.

Formally, we aim to select $\mathbf{B}^{\flat}$ to maximize the value m where:

$$m = \sum_{\mathbf{H}^A \in \mathbf{A}^{\flat}} \mathbf{T}^A \tag{195}$$

16. Ratings and Rewards

The Jam chain does not explicitly issue rewards—we leave this as a job to be done by the staking subsystem (a system parachain—hosted without fees—in the current imagining of a public Jam network). However, much as with validator

punishment information, it is important for the JAM chain to facilitate the arrival of performance information in to the staking subsystem so that it may be acted upon.

Such performance information cannot directly cover all aspects of validator activity; whereas block production, guarantor reports and availability assurance can easily be tracked on-chain, GRANDPA, BEEFY and auditing activity cannot. In the latter case, this is instead tracked with validator voting activity: validators vote on their impression of each other's efforts and a median may be accepted as the truth for any given validator. With an assumption of 50% honest validators, this gives an adequate means of oraclizing this information.

17. DISCUSSION

17.1. **Technical Characteristics.** In total, with our stated target of 1,023 validators and three validators per core, along with requiring a mean of ten audits per validator per timeslot, and thus 30 audits per work-report, JAM is capable of trustlessly processing and integrating 341 work-packages per timeslot.

We assume node hardware is a modern 16 core CPU with 64GB RAM, 1TB secondary storage and 0.5Gbe networking.

Our performance models assume a rough split of CPU time as follows:

	Proportion
Audits	$10/16$
Merklization	$1/16$
Block execution	$2/16$
GRANDPA and BEEFY	$1/16$
Erasure coding	$1/16$
Networking & misc	$1/16$

Estimates for network bandwidth requirements are as follows:

	Upload Mb/s	*Download* Mb/s
Guaranteeing	30	40
Assuring	12	8
Auditing	200	200
Block publication	42	42
Grandpa and Beefy	4	4
Total	**288**	**294**

Thus, a connection able to sustain 500Mb/s should leave a sufficient margin of error and headroom to serve other validators as well as some public connections, though the burstiness of block publication would imply validators are best to ensure that peak bandwidth is higher.

Under these conditions, we would expect an overall network-provided data availability capacity of 2PB, with each node dedicating at most 6TB to availability storage.

Estimates for memory usage are as follows:

	GB	
Auditing	20	2 × 10 PVM instances
Block execution	2	1 PVM instance
State cache	40	
Misc	2	
Total	**64**	

As a rough guide, each parachain has an average footprint of around 2MB in the Polkadot Relay chain; a 40GB state would allow 20,000 parachains' information to be retained in state.

What might be called the "virtual hardware" of a JAM core is essentially a regular CPU core executing at somewhere between 25% and 50% of regular speed for the whole six-second portion and which may draw and provide 2.5MB/s average in general-purpose I/O and utilize up to 2GB in RAM. The I/O includes any trustless reads from the JAM chain state, albeit in the recent past. This virtual hardware also provides unlimited reads from a semi-static preimage-lookup database.

Each work-package may occupy this hardware and execute arbitrary code on it in six-second segments to create some result of at most 90KB. This work result is then entitled to 10ms on the same machine, this time with no "external" I/O beyond said result, but instead with full and immediate access to the JAM chain state and may alter the service(s) to which the results belong.

17.2. **Illustrating Performance.** In terms of pure processing power, the JAM machine architecture can deliver extremely high levels of homogeneous trustless computation. However, the core model of JAM is a classic parallelized compute architecture, and for solutions to be able to utilize the architecture well they must be designed with it in mind to some extent. Accordingly, until such use-cases appear on JAM with similar semantics to existing ones, it is very difficult to make direct comparisons to existing systems. That said, if we indulge ourselves with some assumptions then we can make some crude comparisons.

17.2.1. *Comparison to Polkadot.* Pre-asynchronous backing, Polkadot validates around 50 parachains, each one utilizing approximately 250ms of native computation (i.e. half a second of Wasm execution time at around a 50% overhead) and 5MB of I/O for every twelve seconds of real time which passes. This corresponds to an aggregate compute performance of around parity with a native CPU core and a total 24-hour distributed availability of around 20MB/s. Accumulation is beyond Polkadot's capabilities and so not comparable.

Post asynchronous-backing and estimating that Polkadot is at present capable of validating at most 80 parachains each doing one second of native computation in every six, then the aggregate performance is increased to around 13x native CPU and the distributed availability increased to around 67MB/s.

For comparison, in our basic models, JAM should be capable of attaining around 85x the computation load of a single native CPU core and a distributed availability of 852MB/s.

17.2.2. *Simple Transfers.* We might also attempt to model a simple transactions-per-second amount, with each transaction requiring a signature verification and the modification of two account balances. Once again, until there are clear designs for precisely how this would work we must make some assumptions. Our most naive model would be to use the JAM cores (i.e. refinement) simply for transaction verification and account lookups. The JAM chain would then hold and alter the

balances in its state. This is unlikely to give great performance since almost all the needed I/O would be synchronous, but it can serve as a basis.

A 15MB work-package can hold around 125k transactions at 128 bytes per transaction. However, a 90KB work-result could only encode around 11k account updates when each update is given as a pair of a 4 byte account index and 4 byte balance, resulting in a limit of 5.5k transactions per package, or 312k TPS in total. It is possible that the eight bytes could typically be compressed by a byte or two, increasing maximum throughput a little. Our expectations are that state updates, with highly parallelized Merklization, can be done at between 500k and 1 million reads/write per second, implying around 250k-350k TPS, depending on which turns out to be the bottleneck.

A more sophisticated model would be to use the JAM cores for balance updates as well as transaction verification. We would have to assume that state and the transactions which operate on them can be partitioned between work-packages with some degree of efficiency, and that the 15MB of the work-package would be split between transaction data and state witness data. Our basic models predict that a 4bn 32-bit account system paginated into 2^{10} accounts/page and 128 bytes per transaction could, assuming only around 1% of oraclized accounts were useful, average upwards of 1.7mTPS depending on partitioning and usage characteristics. Partitioning could be done with a fixed fragmentation (essentially sharding state), a rotating partition pattern or a dynamic partitioning (which would require specialized sequencing).

Interestingly, we expect neither model to be bottlenecked in computation, meaning that transactions could be substantially more sophisticated, perhaps with more flexible cryptography or smart contract functionality, without a significant impact on performance.

17.2.3. *Computation Throughput.* The TPS metric does not lend itself well to measuring distributed systems' computational performance, so we now turn to another slightly more compute-focussed benchmark: the EVM. The basic *YP* Ethereum network, now approaching a decade old, is probably the best known example of general purpose decentralized computation and makes for a reasonable yardstick. It is able to sustain a computation and I/O rate of 1.25M gas/sec, with a peak throughput of twice that. The EVM gas metric was designed to be a

time-proportional metric for predicting and constraining program execution. Attempting to determine a concrete comparison to PVM throughput is non-trivial and necessarily opinionated owing to the disparity between the two platforms including word size, endianness and stack/register architecture and memory model. However, we will attempt to determine a reasonable range of values.

EVM gas does not directly translate into native execution as it also combines state reads and writes as well as transaction input data, implying it is able to process some combination of up to 595 storage reads, 57 storage writes and 1.25M gas as well as 78KB input data in each second, trading one against the other.[12] We cannot find any analysis of the typical breakdown between storage I/O and pure computation, so to make a very conservative estimate, we assume it does all four. In reality, we would expect it to be able to do on average $1/4$ of each.

Our experiments[13] show that on modern, high-end consumer hardware with a modern EVM implementation, we can expect somewhere between 180 and 500 gas/µs in throughput on pure-compute workloads (we specifically utilized Odd-Product, Triangle-Number and several implementations of the Fibonacci calculation). To make a conservative comparison to PVM, we propose transcompilation of the EVM code into PVM code and then re-execution of it under the PolkaVM prototype.[14]

To help estimate a reasonable lower-bound of EVM gas/µs, e.g. for workloads which are more memory and I/O intensive, we look toward real-world permissionless deployments of the EVM and see that the Moonbeam network, after correcting for the slowdown of executing within the recompiled WebAssembly platform on the somewhat conservative Polkadot hardware platform, implies a throughput of

[12]The latest "proto-danksharding" changes allow it to accept 87.3KB/s in committed-to data though this is not directly available within state, so we exclude it from this illustration, though including it with the input data would change the results little.

[13]This is detailed at `https://hackmd.io/@XXX9CM1uSSCWVNFRYaSB5g/HJarTUhJA` and intended to be updated as we get more information.

[14]It is conservative since we don't take into account that the source code was originally compiled into EVM code and thus the PVM machine code will replicate architectural artifacts and thus is very likely to be pessimistic. As an example, all arithmetic operations in EVM are 256-bit and 32-bit native PVM is being forced to honor this even if the source code only actually required 32-bit values.

around 100 gas/µs. We therefore assert that in terms of computation, 1µs EVM gas approximates to around 100-500 gas on modern high-end consumer hardware.[15]

Benchmarking and regression tests show that for the iterative Fibonacci calculation, the prototype PVM engine has a fixed preprocessing overhead of around 5ns/byte of program code and a marginal factor of 1.6%, implying an asymptotic speedup of around 63x. For machine code 1MB in size expected to take of the order of a second to compute, the compilation cost becomes only 0.5% of the overall time. The pattern is reproducible with other benchmarks.[16] For code not inherently suited to the 256-bit EVM ISA, we would expect substantially improved relative execution times on PVM.

Even if we allow for preprocessing to take up to the same component within execution as the marginal cost (owing to, for example, an extremely large but short-running program) and for the PVM metering to imply a safety overhead of 2x to execution speeds, then we can expect a JAM core to be able to process the equivalent of around 1,500 EVM gas/µs. We might reason a typical case, targeted to PVM ratehr than via an EVM representation, to be in excess of ten times that.

JAM cores are each capable of 2.5MB/s bandwidth, which must include any state I/O and data which must be newly introduced (e.g. transactions). While writes come at comparatively little cost to the core, only requiring hashing to determine an eventual updated Merkle root, reads must be witnessed, with each one costing around 640 bytes of witness conservatively assuming a one-million entry binary Merkle trie. This would result in a maximum of a little under 4k reads/second/core, with the exact amount dependent upon how much of the bandwidth is used for newly introduced input data.

Aggregating everything across JAM, excepting accumulation which could add further throughput, numbers can be multiplied by 341 (with the caveat that each one's computation cannot interfere with any of the others' except through state oraclization and accumulation). Unlike for *roll-up chain* designs such as Polkadot and Ethereum, there is no need to have persistently fragmented state. Smart-contract state may be held in a coherent format on the JAM chain so long as any

[15]We speculate that the substantial range could possibly be caused in part by the major architectural differences between the EVM ISA typical modern hardware.

[16]As an example, the odd-product benchmark, another pure-compute arithmetic task, execution takes 58s on EVM, and 1.04s within our PVM prototype, including all preprocessing.

updates are made through the 15KB/core/sec work results, which would need to contain only the hashes of the altered contracts' state roots.

Under our modelling assumptions, we can therefore summarize:

	Eth. L1	JAM Core	JAM
Compute (EVM gas/µs)	1.25†	1.5-15K	0.5-5M
State writes (s^{-1})	57†	n/a	n/a
State reads (s^{-1})	595†	4K‡	1.4M‡
Input data (s^{-1})	78KB†	2.5MB‡	852MB‡

What we can see is that JAM's overall predicted performance profile implies it could be comparable to thousands or even millions of that of the basic Ethereum L1 chain. The large factor here is essentially due to three things: spacial parallelism, as JAM can host several hundred cores under its security apparatus; temporal parallelism, as JAM targets continuous execution for its cores and pipelines much of the computation between blocks to ensure a constant, optimal workload; and platform optimization by using a VM and gas model which closely fits modern hardware architectures.

It must however be understood that this is a provisional and crude estimation only. It is included for only the purpose of expressing JAM's performance in tangible terms and is not intended as a means of comparing to a "full-blown" Ethereum/L2-ecosystem combination. Specifically, it does not take into account:

- that these numbers are based on real performance of Ethereum and performance modelling of JAM (though our models are based on real-world performance of the components);
- any L2 scaling which may be possible with either JAM or Ethereum;
- the state partitioning which uses of JAM would imply;
- the as-yet unfixed gas model for the PVM;
- that PVM/EVM comparisons are necessarily imprecise;
- (†) all figures for Ethereum L1 are drawn from the same resource: on average each figure will be only ¼ of this maximum.

- (‡) the state reads and input data figures for JAM are drawn from the same resource: on average each figure will be only ½ of this maximum.

We leave it as further work for an empirical analysis of performance and an analysis and comparison between JAM and the aggregate of a hypothetical Ethereum ecosystem which included some maximal amount of L2 deployments together with full Dank-sharding and any other additional consensus elements which they would require. This, however, is out of scope for the present work.

18. Conclusion

We have introduced a novel computation model which is able to make use of pre-existing crypto-economic mechanisms in order to deliver major improvements in scalability without causing persistent state-fragmentation and thus sacrificing overall cohesion. We call this overall pattern collect-refine-join-accumulate. Furthermore, we have formally defined the on-chain portion of this logic, essentially the join-accumulate portion. We call this protocol the JAM chain.

We argue that the model of JAM provides a novel "sweet spot", allowing for massive amounts of computation to be done in secure, resilient consensus compared to fully-synchronous models, and yet still have strict guarantees about both timing and integration of the computation into some singleton state machine unlike persistently fragmented models.

18.1. **Further Work.**

While we are able to estimate theoretical computation possible given some basic assumptions and even make broad comparisons to existing systems, practical numbers are invaluable. We believe the model warrants further empirical research in order to better understand how these theoretical limits translate into real-world performance. We feel a proper cost analysis and comparison to pre-existing protocols would also be an excellent topic for further work.

We can be reasonably confident that the design of JAM allows it to host a service under which Polkadot *parachains* could be validated, however further prototyping work is needed to understand the possible throughput which a PVM-powered metering system could support. We leave such a report as further work. Likewise, we have also intentionally omitted details of higher-level protocol elements including cryptocurrency, coretime sales, staking and regular smart-contract functionality.

A number of potential alterations to the protocol described here are being considered in order to make practical utilization of the protocol easier. These include:

- Synchronous calls between services in accumulate.
- Restrictions on the `transfer` function in order to allow for substantial parallelism over accumulation.
- The possibility of reserving substantial additional computation capacity during accumulate under certain conditions.
- Introducing Merklization into the Work Package format in order to obviate the need to have the whole package downloaded in order to evaluate its authorization.

The networking protocol is also left intentionally undefined at this stage and its description must be done in a follow-up proposal.

Validator performance is not presently tracked on-chain. We do expect this to be tracked on-chain in the final revision of the JAM protocol, but its specific format is not yet certain and it is therefore omitted at present.

19. Acknowledgements

Much of this present work is based in large part on the work of others. The Web3 Foundation research team and in particular Alistair Stewart and Jeff Burdges are responsible for ELVES, the security apparatus of Polkadot which enables the possibility of in-core computation for JAM. The same team is responsible for Sassafras, GRANDPA and BEEFY.

Safrole is a mild simplification of Sassafras and was made under the careful review of Davide Gallosi and Alistair Stewart.

The original CoreJam RFC was refined under the review of Bastian Köcher and Robert Habermeier and most of the key elements of that proposal have made their way into the present work.

The PVM is a formalization of a partially simplified *PolkaVM* software prototype, developed by Jan Bujak. Cyrill Leutwiler contributed to the empirical analysis of the PVM reported in the present work.

The *PolkaJam* team and in particular Arkadiy Paronyan, Emeric Chevalier and Dave Emmet have been instrumental in the design of the lower-level aspects of the JAM protocol, especially concerning Merklization and I/O.

And, of course, thanks to the awesome Lemon Jelly, a.k.a. Fred Deakin and Nick Franglen, for three of the most beautiful albums ever produced, the cover art of the first of which was inspiration for this paper's background art.

Appendix A. Polka Virtual Machine

A.1. **Basic Definition.** We declare the general PVM function Ψ. We assume a single-step invocation function define Ψ_1 and define the full PVM recursively as a sequence of such mutations up until the single-step mutation results in a halting condition.

(196)
$$\Psi: \begin{cases} (\mathbb{Y}, \mathbb{N}_R, \mathbb{N}_G, [\![\mathbb{N}_R]\!]_{13}, \mathbb{M}) \to (\{\blacksquare, \lightning, \infty\} \cup \{\text{⨅}\} \times \mathbb{N}_R \cup \{\hbar\} \times \mathbb{N}_R, \mathbb{N}_R, \mathbb{Z}_G, [\![\mathbb{N}_R]\!]_{13}, \mathbb{M}) \\ \qquad (\mathbf{p}, \imath, \xi, \omega, \mu) \mapsto \begin{cases} \Psi(\mathbf{p}, \imath', \xi', \omega', \mu') & \text{if } \zeta = \blacktriangleright \\ (\infty, \imath', \xi', \omega', \mu') & \text{if } \xi' < 0 \\ (\zeta, \imath', \xi', \omega', \mu') & \text{otherwise} \end{cases} \\ \quad \text{where } (\zeta, \imath', \xi', \omega', \mu') = \Psi_1(\mathbf{c}, \mathbf{j}, \imath, \xi, \omega, \mu) \\ \qquad\qquad \text{and } \mathbf{p} = \mathcal{E}(\updownarrow\mathbf{j}, \mathbf{c}) \end{cases}$$

If the latter condition cannot be satisfied, then $(\lightning, \imath, \xi, \omega, \mu)$ is the result.

The PVM exit reason $r \in \{\blacksquare, \lightning, \infty\} \cup \{\text{⨅}, \hbar\} \times \mathbb{N}_R$ may be one of regular halt $\blacksquare$, panic $\lightning$ or out-of-gas ∞, or alternatively a host-call $\hbar$, in which the host-call identifier is associated, or page-fault ⨅ in which case the address into RAM is associated.

A.2. **Instructions, Basic-Blocks and the Jump Table.** The program blob $\mathbf{p}$ is split into a series of octets which make up the instruction data $\mathbf{c}$ and the *dynamic jump table*, $\mathbf{j}$. The former implies an instruction sequence, and by extension a *basic-block sequence*, itself a sequence of indices of the instructions which follow a *block-termination* instruction.

The dynamic jump table builds on this and is a sequence containing, in order, every basic-block index to which the instruction counter may be altered if computed dynamically.

Most instructions are composed of multiple octets. Since the PVM counts instructions in unit terms (rather than octet terms) it is convenient to define the sequence of instruction blobs which these octets represent. Given our instructions blob $\mathbf{c}$, we make an equivalence for our instruction-blob sequence ε. To define this we presume a function $\ell(\mathbb{N}_{2^8})$ which provides the length of the instruction whose opcode is given as parameter according to the ℓ value in the tables of section A.4.

Formally:

$$\varepsilon \equiv [I(0), I(1), \dots I(m-1)] \tag{197}$$

$$\text{where } I(n) \equiv \mathbf{c}_{o \dots + \ell(\mathbf{c}_o)} \ , \quad m = \min(n \in \mathbb{N} : o = |\mathbf{c}|) \ , \quad o = \sum_{m \in \mathbb{N}_n} |I(m, \mathbf{c})| \tag{198}$$

A.2.1. *Basic Blocks and Termination Instructions.* Instructions of the following opcodes are considered basic-block termination instructions; other than `trap` & `fallthrough`, they corresond to instructions which may define the instruction-counter to be something other than one more than its prior value:

- Trap and fallthrough: `trap` , `fallthrough`
- Jumps: `jump` , `jump_ind`
- Calls: `call` , `call_ind`
- Branches: `branch_eq` , `branch_ne` , `branch_ge_u` , `branch_ge_s` , `branch_lt_u` , `branch_lt_s` , `branch_eq_imm` , `branch_ne_imm`
- Immediate branches: `branch_lt_u_imm` , `branch_lt_s_imm` , `branch_le_u_imm` , `branch_le_s_imm` , `branch_ge_u_imm` , `branch_ge_s_imm` , `branch_gt_u_imm` , `branch_gt_s_imm`

We denote this set, as opcode indices rather than names, as T. We define the instruction indices denoting the beginning of basic-blocks as ϖ:

$$\varpi \equiv [0] \frown [n + 1 \mid n \lhd \mathbb{N}_{|\varepsilon|} \wedge \varepsilon_n \in T] \tag{199}$$

We require that the dynamic jump-table only contain the indices of basic-blocks and that they include the indices for all basic-blocks immediately following a `call` (6) or `call_ind` (42) instruction:

$$\mathbf{j} \subset \mathbb{N}_{|\varpi|} \tag{200}$$

$$\forall n \in \mathbb{N}_{|\varepsilon|} : (\varepsilon_n)_0 \in \{6, 42\} \implies b \in \mathbf{j} \text{ where } \varpi_b = n + 1 \tag{201}$$

As with other hard requirements, if the former is not satisfied, then posterior machine state is equivalent to prior and the exit reason of the PVM function is ↯.

A.3. **Single-Step State Transition.** We must now define the single-step PVM state-transition function Ψ_1:

(202)
$$\Psi_1 \colon \begin{cases} (\mathbb{Y}, \llbracket \mathbb{N}_R \rrbracket, \mathbb{N}_R, \mathbb{N}_G, \llbracket \mathbb{N}_R \rrbracket_{13}, \mathbb{M}) \to (\{\lightning, \blacksquare, \blacktriangleright\} \cup \{\sqsubtract, \hbar\} \times \mathbb{N}_R, \mathbb{Z}_G, \llbracket \mathbb{N}_R \rrbracket_{13}, \mathbb{M}) \\ \qquad (\mathbf{c}, \mathbf{j}, \imath, \xi, \omega, \mu) \mapsto (\zeta, \imath', \xi', \omega', \mu') \end{cases}$$

We define ζ together with the posterior values (denoted as prime) of each of the items of the machine state as being in accordance with the table below.

In general, when transitioning machine state for an instruction a number of conditions hold true and instructions are defined essentially by their exceptions to these rules. Specifically, the machine does not halt, the instruction counter increments by one, the gas remaining is reduced by the amount corresponding to the instruction type and RAM & registers are unchanged. Formally:

(203) $$\zeta = \blacktriangleright, \quad \imath' = \imath + 1, \quad \xi' = \xi - \xi_\Delta, \quad \omega' = \omega, \quad \mu' = \mu \text{ except as indicated}$$

Where RAM must be inspected and yet access is not possible, then machine state is unchanged, and the exit reason is a fault with the lowest address to be read which is inaccessible. More formally, let $\mathbf{a}$ be the set of indices in to which μ must be subscripted in order to calculate the result of Ψ_1. If $\mathbf{a} \notin \mathbb{V}_\mu$ then let $\zeta = \sqsubtract \times \min(\mathbf{a} \setminus \mathbb{V}_\mu)$.

Similarly, where RAM must be mutated and yet mutable access is not possible, then machine state is unchanged, and the exit reason is a fault with the lowest address to be read which is inaccessible. More formally, let $\mathbf{a}$ be the set of indices in to which μ' must be subscripted in order to calculate the result of Ψ_1. If $\mathbf{a} \notin \mathbb{V}^*_\mu$ then let $\zeta = \sqsubtract \times \min(\mathbf{a} \setminus \mathbb{V}^*_\mu)$.

We define signed/unsigned transitions for various octet widths:

(204) $$\mathcal{Z}_{n \in \mathbb{N}} \colon \begin{cases} \mathbb{N}_{2^{8n}} \to \mathbb{Z}_{-2^{8n-1}:2^{8n-1}} \\ \qquad a \mapsto \begin{cases} a & \text{if } a < 2^{8n-1} \\ a - 2^{8n} & \text{otherwise} \end{cases} \end{cases}$$

(205) $$\mathcal{Z}^{-1}_{n \in \mathbb{N}} \colon \begin{cases} \mathbb{Z}_{-2^{8n-1}:2^{8n-1}} \to \mathbb{N}_{2^{8n}} \\ \qquad a \mapsto (2^{8n} + a) \bmod 2^{8n} \end{cases}$$

$$\mathcal{B}_{n\in\mathbb{N}}: \begin{cases} \mathbb{N}_{2^{8n}} \to \mathbb{B}_{8n} \\ \quad x \mapsto \mathbf{y} : \forall i \in \mathbb{N}_{2^{8n}} : \mathbf{y}[i] \Leftrightarrow \left\lfloor \frac{x}{2^i} \right\rfloor \bmod 2 \end{cases} \tag{206}$$

$$\mathcal{B}^{-1}_{n\in\mathbb{N}}: \begin{cases} \mathbb{B}_{8n} \to \mathbb{N}_{2^{8n}} \\ \quad \mathbf{x} \mapsto y : \sum_{i\in\mathbb{N}_{2^{8n}}} \mathbf{x}_i \cdot 2^i \end{cases} \tag{207}$$

Static jumps, calls and branches must be to valid items in the jump table and if they are not, then a panic occurs:

$$\texttt{branch}(b, C) \implies (\zeta, \imath') = \begin{cases} (\lightning, \imath) & \text{if } b \geq |\varpi| \\ (\blacktriangleright, \imath) & \text{otherwise if } \neg C \\ (\blacktriangleright, \varpi_b) & \text{otherwise} \end{cases} \tag{208}$$

Calls and jumps whose next instruction is dynamically computed meanwhile must use an address which may be indexed into the jump-table $\mathbf{j}$. Through a quirk of tooling[17], we define the dynamic address required by the instructions as the jump table index incremented by one and then multiplied by our jump alignment factor $\mathsf{Z}_A = 4$:

(209)

$$\texttt{djump}(a) \implies (\zeta, \imath') = \begin{cases} (\blacksquare, \imath) & \text{if } a = 2^{32} - 2^{16} \\ (\lightning, \imath) & \text{otherwise if } a = 0 \vee a > |\mathbf{j}| \cdot \mathsf{Z}_A \vee a \bmod \mathsf{Z}_A \neq 0 \\ (\blacktriangleright, \varpi_{\mathbf{j}_{(a/\mathsf{Z}_A)-1}}) & \text{otherwise} \end{cases}$$

A.4. **Instruction Tables.**

A.4.1. *Instructions without Arguments.*

$$\ell = 1 \tag{210}$$

[17]The popular code generation backend LLVM requires and assumes in its code generation that dynamically computed jump destinations always have a certain memory alignment. Since at present we depend on this for our tooling, we must acquiesce to its assumptions.

$(\varepsilon_\imath)_0$	**Name**	ξ_Δ	**Mutations**
0	`trap`	0	$\zeta = \blacksquare$
17	`fallthrough`	0	

A.4.2. *Instructions with Arguments of One Register & One Immediate.*

$$\ell = 2 + x, \quad \omega_A \equiv \omega_{(\varepsilon_\imath)_1}, \quad \omega'_A \equiv \omega'_{(\varepsilon_\imath)_1}, \quad \exists x, \nu_X \in \mathbb{N}_R : (\varepsilon_\imath)_{2\cdots+x} = \mathcal{E}(\nu_X) \tag{211}$$

In the case that the above condition is not met, then the instruction is considered invalid, and it results in a panic; $\zeta = \lightning$.

We denote the next jump index as j_N, as always equivalent to the lowest value in the jump table which is greater than the instruction counter. In the case of an instruction where this value is utilized when undefined then the instruction causes no mutations to machine state but results in an exit reason of panic $\lightning$.

$$R(\imath) \equiv \mathsf{Z}_A(1 + j) \text{ where } \varpi_{\mathbf{j}_j} = \imath + 1 \tag{212}$$

$(\varepsilon_\imath)_0$	**Name**	ξ_Δ	**Mutations**
6	`call`	0	$\texttt{branch}(\nu_X, \top)$, $\omega'_A = R(\imath)$
19	`jump_ind`	0	$\texttt{djump}((\omega_A + \nu_X) \bmod 2^{32})$
4	`load_imm`	0	$\omega'_A = \nu_X$
60	`load_u8`	0	$\omega'_A = \mu_{\nu_X}$
74	`load_i8`	0	$\omega'_A = \mathcal{Z}_4^{-1}(\mathcal{Z}_1(\mu_{\nu_X}))$
76	`load_u16`	0	$\omega'_A = \mathcal{E}_2^{-1}(\mu^{\circlearrowleft}_{\nu_X\cdots+2})$
66	`load_i16`	0	$\omega'_A = \mathcal{Z}_4^{-1}(\mathcal{Z}_2(\mathcal{E}_2^{-1}(\mu^{\circlearrowleft}_{\nu_X\cdots+2})))$

$(\varepsilon_\imath)_0$	**Name**	ξ_Δ	**Mutations**
10	`load_u32`	2	$\omega'_A = \mathcal{E}_4^{-1}(\mu^{\circlearrowleft}_{\nu_X \cdots +4})$
71	`store_u8`	0	$\mu'^{\circlearrowleft}_{\nu_X} = \omega_A \bmod 2^8$
69	`store_u16`	0	$\mu'^{\circlearrowleft}_{\nu_X \cdots +2} = \mathcal{E}_2(\omega_A \bmod 2^{16})$
22	`store_u32`	2	$\mu'^{\circlearrowleft}_{\nu_X \cdots +4} = \mathcal{E}_4(\omega_A)$

A.4.3. *Instructions with Arguments of a Register & Two Immediates.* (213)

$\ell = 2{+}x{+}y, \quad \omega_A \equiv \omega_{(\varepsilon_\imath)_1}, \quad \omega'_A \equiv \omega'_{(\varepsilon_\imath)_1}, \quad \exists x, y, \nu_X, \nu_Y \in \mathbb{N}_R : (\varepsilon_\imath)_{2\cdots+x} = \mathcal{E}(\nu_X) \wedge (\varepsilon_\imath)_{2+x\cdots+y} = \mathcal{E}$

In the case that the above condition is not met, then the instruction is considered invalid, and it results in a panic; $\zeta = \lightning$.

$(\varepsilon_\imath)_0$	**Name**	ξ_Δ	**Mutations**
7	`branch_eq_imm`	0	$\texttt{branch}(\nu_Y, \omega_A = \nu_X)$
15	`branch_ne_imm`	0	$\texttt{branch}(\nu_Y, \omega_A \neq \nu_X)$
44	`branch_lt_u_imm`	0	$\texttt{branch}(\nu_Y, \omega_A < \nu_X)$
59	`branch_le_u_imm`	0	$\texttt{branch}(\nu_Y, \omega_A \leq \nu_X)$
52	`branch_ge_u_imm`	0	$\texttt{branch}(\nu_Y, \omega_A \geq \nu_X)$
50	`branch_gt_u_imm`	0	$\texttt{branch}(\nu_Y, \omega_A > \nu_X)$
32	`branch_lt_s_imm`	0	$\texttt{branch}(\nu_Y, \mathcal{Z}_4(\omega_A) < \mathcal{Z}_4(\nu_X))$
46	`branch_le_s_imm`	0	$\texttt{branch}(\nu_Y, \mathcal{Z}_4(\omega_A) \leq \mathcal{Z}_4(\nu_X))$
45	`branch_ge_s_imm`	0	$\texttt{branch}(\nu_Y, \mathcal{Z}_4(\omega_A) \geq \mathcal{Z}_4(\nu_X))$
53	`branch_gt_s_imm`	0	$\texttt{branch}(\nu_Y, \mathcal{Z}_4(\omega_A) > \mathcal{Z}_4(\nu_X))$

	Name	ξ_Δ	Mutations
$(\varepsilon_\imath)_0$			
26	`store_imm_ind_u8`	0	$\mu'^{\circlearrowleft}_{\omega_A+\nu_X} = \nu_Y \bmod 2^8$
54	`store_imm_ind_u16`	0	$\mu'^{\circlearrowleft}_{\omega_A+\nu_X} = \mathcal{E}_2(\nu_Y \bmod 2^{16})$
13	`store_imm_ind_u32`	2	$\mu'^{\circlearrowleft}_{\omega_A+\nu_X} = \mathcal{E}_4(\nu_Y)$

A.4.4. *Instructions with Arguments of Two Registers & One Immediate.* (214)

$\ell = 2 + x, \quad \omega_A \equiv \omega_a, \quad \omega'_A \equiv \omega'_a, \quad \omega_B \equiv \omega_b, \quad \omega'_B \equiv \omega'_b, \quad a \,\mathsf{R}\, b = {}^{(\varepsilon_\imath)_1}/_{16}, \quad \exists x, \nu_X \in \mathbb{N}_R : (\varepsilon_\imath)_{2\cdots+}$

In the case that the above condition is not met, then the instruction is considered invalid, and it results in a panic; $\zeta = \lightning$.

	Name	ξ_Δ	Mutations
$(\varepsilon_\imath)_0$			
16	`store_ind_u8`	0	$\mu'^{\circlearrowleft}_{\omega_B+\nu_X} = \omega_A \bmod 2^8$
29	`store_ind_u16`	0	$\mu'^{\circlearrowleft}_{\omega_B+\nu_X} = \mathcal{E}_2(\omega_A \bmod 2^{16})$
3	`store_ind_u32`	0	$\mu'^{\circlearrowleft}_{\omega_B+\nu_X} = \mathcal{E}_4(\omega_A)$
11	`load_ind_u8`	0	$\omega'_A = \mu_{\omega_B+\nu_X}$
21	`load_ind_i8`	0	$\omega'_A = \mathcal{Z}_4^{-1}(\mathcal{Z}_1(\mu_{\omega_B+\nu_X}))$
37	`load_ind_u16`	0	$\omega'_A = \mathcal{E}_2^{-1}(\mu^{\circlearrowleft}_{\omega_B+\nu_X\cdots+2})$
33	`load_ind_i16`	0	$\omega'_A = \mathcal{Z}_4^{-1}(\mathcal{Z}_2(\mathcal{E}_2^{-1}(\mu^{\circlearrowleft}_{\omega_B+\nu_X\cdots+2})))$
1	`load_ind_u32`	0	$\omega'_A = \mathcal{E}_4^{-1}(\mu^{\circlearrowleft}_{\omega_B+\nu_X\cdots+4})$
42	`call_ind`	0	$\texttt{djump}((\omega_B + \nu_X) \bmod 2^{32}) \;, \quad \omega'_A = R(\imath)$
2	`add_imm`	0	$\omega'_B = (\omega_A + \nu_X) \bmod 2^{32}$
18	`and_imm`	0	$\forall i \in \mathbb{N}_{32} : \mathcal{B}_4(\omega'_B)_i = \mathcal{B}_4(\omega_A)_i \wedge \mathcal{B}_4(\nu_X)_i$

$(\varepsilon_i)_0$	**Name**	ξ_Δ	**Mutations**
31	`xor_imm`	0	$\forall i \in \mathbb{N}_{32} : \mathcal{B}_4(\omega'_B)_i = \mathcal{B}_4(\omega_A)_i \oplus \mathcal{B}_4(\nu_X)_i$
49	`or_imm`	0	$\forall i \in \mathbb{N}_{32} : \mathcal{B}_4(\omega'_B)_i = \mathcal{B}_4(\omega_A)_i \vee \mathcal{B}_4(\nu_X)_i$
35	`mul_imm`	0	$\omega'_B = (\omega_A \cdot \nu_X) \bmod 2^{32}$
65	`mul_upper_s_s_imm`	0	$\omega'_B = \mathcal{Z}_4^{-1}(\lfloor (\mathcal{Z}_4(\omega_A) \cdot \mathcal{Z}_4(\nu_X)) \div 2^{32} \rfloor)$
63	`mul_upper_u_u_imm`	0	$\omega'_B = \lfloor (\omega_A \cdot \nu_X) \div 2^{32} \rfloor$
27	`set_lt_u_imm`	0	$\omega'_B = \omega_A < \nu_X$
56	`set_lt_s_imm`	0	$\omega'_B = \mathcal{Z}_4(\omega_A) < \mathcal{Z}_4(\nu_X)$
9	`shlo_l_imm`	0	$\omega'_B = (\omega_A \cdot 2^{\nu_X \bmod 32}) \bmod 2^{32}$
14	`shlo_r_imm`	0	$\omega'_B = \lfloor \omega_A \div 2^{\nu_X \bmod 32} \rfloor$
25	`shar_r_imm`	0	$\omega'_B = \mathcal{Z}_4^{-1}(\lfloor \mathcal{Z}_4(\omega_A) \div 2^{\nu_X \bmod 32} \rfloor)$
40	`neg_add_imm`	0	$\omega'_B = (\nu_X + 2^{32} - \omega_A) \bmod 2^{32}$
39	`set_gt_u_imm`	0	$\omega'_B = \omega_A > \nu_X$
61	`set_gt_s_imm`	0	$\omega'_B = \mathcal{Z}_4(\omega_A) > \mathcal{Z}_4(\nu_X)$
75	`shlo_l_imm_alt`	1	$\omega'_B = (\nu_X \cdot 2^{\omega_A \bmod 32}) \bmod 2^{32}$
72	`shlo_r_imm_alt`	1	$\omega'_B = \lfloor \nu_X \div 2^{\omega_A \bmod 32} \rfloor$
80	`shar_r_imm_alt`	1	$\omega'_B = \mathcal{Z}_4^{-1}(\lfloor \mathcal{Z}_4(\nu_X) \div 2^{\omega_A \bmod 32} \rfloor)$
24	`branch_eq`	0	$\texttt{branch}(\nu_X, \omega_A = \omega_B)$
30	`branch_ne`	0	$\texttt{branch}(\nu_X, \omega_A \neq \omega_B)$
47	`branch_lt_u`	0	$\texttt{branch}(\nu_X, \omega_A < \omega_B)$
48	`branch_lt_s`	0	$\texttt{branch}(\nu_X, \mathcal{Z}_4(\omega_A) < \mathcal{Z}_4(\omega_B))$
41	`branch_ge_u`	0	$\texttt{branch}(\nu_X, \omega_A \geq \omega_B)$
43	`branch_ge_s`	0	$\texttt{branch}(\nu_X, \mathcal{Z}_4(\omega_A) \geq \mathcal{Z}_4(\omega_B))$

A.4.5. *Instructions with Arguments of Three Registers.*

(215)

$$\ell = 3, \quad \omega_D \equiv \omega_d, \quad \omega'_D \equiv \omega'_d, \quad \omega_A \equiv \omega_a, \quad \omega'_A \equiv \omega'_a, \quad a \,\mathsf{R}\, d = {}^{(\varepsilon_\imath)_1}\!/_{16}, \quad \omega_B \equiv \omega_{(\varepsilon_\imath)_2}, \quad \omega'_B \equiv \omega'_{(\varepsilon_\imath}$$

In the case that the above condition is not met, then the instruction is considered invalid, and it results in a panic; $\zeta = \lightning$.

$(\varepsilon_\imath)_0$	Name	ξ_Δ	Mutations
8	`add`	0	$\omega'_D = (\omega_A + \omega_B) \bmod 2^{32}$
20	`sub`	0	$\omega'_D = (\omega_A + 2^{32} - \omega_B) \bmod 2^{32}$
23	`and`	0	$\forall i \in \mathbb{N}_{32} : \mathcal{B}_4(\omega'_D)_i = \mathcal{B}_4(\omega_A)_i \wedge \mathcal{B}_4(\omega_B)_i$
28	`xor`	0	$\forall i \in \mathbb{N}_{32} : \mathcal{B}_4(\omega'_D)_i = \mathcal{B}_4(\omega_A)_i \oplus \mathcal{B}_4(\omega_B)_i$
12	`or`	0	$\forall i \in \mathbb{N}_{32} : \mathcal{B}_4(\omega'_D)_i = \mathcal{B}_4(\omega_A)_i \vee \mathcal{B}_4(\omega_B)_i$
34	`mul`	0	$\omega'_D = (\omega_A \cdot \omega_B) \bmod 2^{32}$
67	`mul_upper_s_s`	0	$\omega'_D = \mathcal{Z}_4^{-1}(\lfloor (\mathcal{Z}_4(\omega_A) \cdot \mathcal{Z}_4(\omega_B)) \div 2^{32} \rfloor)$
57	`mul_upper_u_u`	0	$\omega'_D = \lfloor (\omega_A \cdot \omega_B) \div 2^{32} \rfloor$
81	`mul_upper_s_u`	0	$\omega'_D = \mathcal{Z}_4^{-1}(\lfloor (\mathcal{Z}_4(\omega_A) \cdot \omega_B) \div 2^{32} \rfloor)$
68	`div_u`	0	$\omega'_D = \begin{cases} 2^{32} - 1 & \text{if } \omega_B = 0 \\ \lfloor \omega_A \div \omega_B \rfloor & \text{otherwise} \end{cases}$
64	`div_s`	0	$\omega'_D = \begin{cases} 2^{32} - 1 & \text{if } \omega_B = 0 \\ \omega_A & \text{if } \mathcal{Z}_4(\omega_A) = -2^{31} \wedge \mathcal{Z}_4(\omega_B) = - \\ \mathcal{Z}_4^{-1}(\lfloor \mathcal{Z}_4(\omega_A) \div \mathcal{Z}_4(\omega_B) \rfloor) & \text{otherwise} \end{cases}$
73	`rem_u`	0	$\omega'_D = \begin{cases} \omega_A & \text{if } \omega_B = 0 \\ \lfloor \omega_A \bmod \omega_B \rfloor & \text{otherwise} \end{cases}$

$(\varepsilon_\imath)_0$	**Name**	ξ_Δ	**Mutations**
70	`rem_s`	0	$\omega'_D = \begin{cases} \omega_A & \text{if } \omega_B = 0 \\ 0 & \text{if } \mathcal{Z}_4(\omega_A) = -2^{31} \wedge \mathcal{Z}_4(\omega_B) = -1 \\ \mathcal{Z}_4^{-1}(\lfloor \mathcal{Z}_4(\omega_A) \bmod \mathcal{Z}_4(\omega_B) \rfloor) & \text{otherwise} \end{cases}$
36	`set_lt_u`	0	$\omega'_D = \omega_A < \omega_B$
58	`set_lt_s`	0	$\omega'_D = \mathcal{Z}_4(\omega_A) < \mathcal{Z}_4(\omega_B)$
55	`shlo_l`	0	$\omega'_D = (\omega_A \cdot 2^{\omega_B \bmod 32}) \bmod 2^{32}$
51	`shlo_r`	0	$\omega'_D = \lfloor \omega_A \div 2^{\omega_B \bmod 32} \rfloor$
77	`shar_r`	0	$\omega'_D = \mathcal{Z}_4^{-1}(\lfloor \mathcal{Z}_4(\omega_A) \div 2^{\omega_B \bmod 32} \rfloor)$
83	`cmov_iz`	0	$\omega'_D = \begin{cases} 0 & \text{if } \omega_B = 0 \\ \omega_A & \text{otherwise} \end{cases}$
84	`cmov_nz`	0	$\omega'_D = \begin{cases} \omega_A & \text{if } \omega_B = 0 \\ 0 & \text{otherwise} \end{cases}$

A.4.6. *Instructions with Arguments of One Immediate.*

$$\ell = 1 + x, \quad \exists x, \nu_X \in \mathbb{N}_R : (\varepsilon_\imath)_{1 \dots + x} = \mathcal{E}(\nu_X) \tag{216}$$

In the case that the above condition is not met, then the instruction is considered invalid, and it results in a panic; $\zeta = \lightning$.

$(\varepsilon_\imath)_0$	**Name**	ξ_Δ	**Mutations**
5	`jump`	0	$\texttt{branch}(\nu_X, \top)$
78	`ecalli`	0	$\zeta = \hbar \times \nu_X$

A.4.7. *Instructions with Arguments of Two Immediates.*

$$\ell = 3 + x + y, \quad \exists x, y, \nu_X, \nu_Y \in \mathbb{N}_R : (\varepsilon_\imath)_{1\cdots+x} = \mathcal{E}(\nu_X) \wedge (\varepsilon_\imath)_{2+x\cdots+y} = \mathcal{E}(\nu_Y) \tag{217}$$

In the case that the above condition is not met, then the instruction is considered invalid, and it results in a panic; $\zeta = \lightning$.

$(\varepsilon_\imath)_0$	Name	ξ_Δ	Mutations
62	`store_imm_u8`	0	$\mu'^{\circlearrowleft}_{\nu_Y} = \nu_X \bmod 2^8$
79	`store_imm_u16`	0	$\mu'^{\circlearrowleft}_{\nu_Y\cdots+2} = \mathcal{E}_2(\nu_X \bmod 2^{16})$
38	`store_imm_u32`	0	$\mu'^{\circlearrowleft}_{\nu_Y\cdots+4} = \mathcal{E}_4(\nu_X)$

A.4.8. *Instructions with Arguments of Two Registers.*

$$\ell = 2, \quad \omega_D \equiv \omega_d, \quad \omega'_D \equiv \omega'_d, \quad \omega_A \equiv \omega_a, \quad \omega'_A \equiv \omega'_a, \quad a \,\mathsf{R}\, d = {}^{(\varepsilon_\imath)_1}/_{16} \tag{218}$$

In the case that the above condition is not met, then the instruction is considered invalid, and it results in a panic; $\zeta = \lightning$.

$(\varepsilon_\imath)_0$	Name	ξ_Δ	Mutations
82	`move_reg`	0	$\omega'_D = \omega_A$
87	`sbrk`	0	$\mathbb{N}_{\omega'_D\cdots+\omega_A} \in \mathbb{V}^*_\mu$ All ranges are guaranteed unique for all invocations of this ins

A.5. **Host Call Definition.** An extended version of the PVM invocation which is able to progress an inner *host-call* state-machine in the case of a host-call halt

condition is defined as Ψ_H:

(219)
$$\Psi_H\colon \begin{cases} (\mathbb{Y}, \mathbb{N}_R, \mathbb{N}_G, [\![\mathbb{N}_R]\!]_{13}, \mathbb{M}, \Omega_X, X) \to (\{\lightning, \infty, \blacksquare\} \cup \{\exists\} \times \mathbb{N}_R, \mathbb{Z}_G, [\![\mathbb{N}_R]\!]_{13}, \mathbb{M}, X) \\ (\mathbf{c}, \imath, \xi, \omega, \mu, f, \mathbf{x}) \mapsto \begin{cases} (\exists \times a, \imath', \xi', \omega', \mu', \mathbf{x}') & \text{if } \bigwedge \begin{cases} \zeta = \hbar \times h \\ \exists \times a = f(h, \xi' \end{cases} \\ \Psi_H(\mathbf{c}, \imath' + 1, \xi'', \omega'', \mu'', f, \mathbf{x}'') & \text{if } \bigwedge \begin{cases} \zeta = \hbar \times (h, i) \\ (\xi'', \omega'', \mu'', \mathbf{x}'' \end{cases} \\ (\zeta, \imath', \xi', \omega', \mu', \mathbf{x}) & \text{otherwise} \end{cases} \\ \quad \text{where } (\zeta, \imath', \xi', \omega', \mu') = \Psi(\mathbf{c}, \imath, \xi, \omega, \mu) \end{cases}$$

On exit, the instruction counter $\imath'$ references the instruction *which caused the exit*. Should the machine be invoked again using this instruction counter and code, then the same instruction which caused the exit would be executed. This is sensible when the instruction is one which necessarily needs re-executing such as in the case of an out-of-gas or page fault reason.

However, when the exit reason to Ψ is a host-call $\hbar$, then the resultant instruction-counter has a value of the host-call instruction and resuming with this state would immediately exit with the same result. Re-invoking would therefore require both the post-host-call machine state *and* the instruction counter value for the instruction following the one which resulted in the host-call exit reason. This is always one greater. Resuming the machine with this instruction counter will continue beyond the host-call instruction.

We use both values of instruction-counter for the definition of Ψ_H since if the host-call results in a page fault we need to allow the outer environment to resolve the fault and re-try the host-call. Conversely, if we successfully transition state according to the host-call, then on resumption we wish to begin with the instruction directly following the host-call.

A.6. **Standard Program Initialization.** The software programs which will run in each of the four instances where the PVM is utilized in the main document have a very typical setup pattern characteristic of an output of a compiler and linker. This means section for program-specific read-only data, read-write (heap) data and the stack. An adjunct to this, very typical of our usage patterns is an extra read-only segment via which invocation-specific data may be passed (i.e. arguments). It thus makes sense to define this properly in a single initializer function.

We thus define the standard program code format $\mathbf{p}$, which includes not only the instructions and jump table (previously represented by the term $\mathbf{c}$), but also information on the state of the RAM and registers at program start. Given some $\mathbf{p}$ which is appropriately encoded together with some argument data $\mathbf{a}$, we can define program code $\mathbf{c}$, registers ω and RAM μ through the standard initialization decoder function Y:

$$Y: \begin{cases} \mathbb{Y} \to (\mathbb{Y}, [\![\mathbb{N}_R]\!]_{13}, \mathbb{M})? \\ \mathbf{p} \mapsto x \end{cases} \tag{220}$$

Where:

$$\text{let } \mathcal{E}_3(|\mathbf{o}|) \frown \mathcal{E}_3(|\mathbf{w}|) \frown \mathcal{E}_2(z) \frown \mathcal{E}_3(s) \frown \mathbf{o} \frown \mathbf{w} \frown \mathcal{E}_4(|\mathbf{c}|) \frown \mathbf{c} = \mathbf{p} \tag{221}$$

$$\mathsf{Z}_P = 2^{14} \quad , \quad \mathsf{Z}_Q = 2^{16} \quad , \quad \mathsf{Z}_I = 2^{24} \tag{222}$$

$$\text{let } P(x \in \mathbb{N}) \equiv \mathsf{Z}_P \left\lceil \frac{x}{\mathsf{Z}_P} \right\rceil \quad , \qquad Q(x \in \mathbb{N}) \equiv \mathsf{Z}_Q \left\lceil \frac{x}{\mathsf{Z}_Q} \right\rceil \tag{223}$$

$$5\mathsf{Z}_Q + Q(|o|) + Q(|w| + z\mathsf{Z}_P) + Q(s) + \mathsf{Z}_I \le 2^{32} \tag{224}$$

If the above cannot be satisfied, then $x = \varnothing$, otherwise $x = (\mathbf{c}, \omega, \mu)$ with $\mathbf{c}$ as above and ω, μ where:

(225)

$$\forall i \in \mathbb{N}_R : \mu_i = \begin{cases} \left(\mathbf{V}\colon \mathbf{o}_{i-\mathsf{Z}_Q},\ \mathbf{A}\colon R\right) & \text{if } \mathsf{Z}_Q \le i < \mathsf{Z}_Q + |\mathbf{o}| \\ (0, R) & \text{if } \mathsf{Z}_Q + |\mathbf{o}| \le i < \mathsf{Z}_Q + P(|\mathbf{o}| \\ (\mathbf{w}_{i-(2\mathsf{Z}_Q+Q(|\mathbf{o}|))}, W) & \text{if } 2\mathsf{Z}_Q + Q(|\mathbf{o}|) \le i < 2\mathsf{Z}_Q + Q(|\mathbf{o}|) + |w \\ (0, W) & \text{if } 2\mathsf{Z}_Q + Q(|\mathbf{o}|) + |\mathbf{w}| \le i < 2\mathsf{Z}_Q + Q(|\mathbf{o}|) + P(|\mathbf{w}|) + z\mathsf{Z} \\ (0, W) & \text{if } 2^{32} - 2\mathsf{Z}_Q - \mathsf{Z}_I - P(s) \le i < 2^{32} - 2\mathsf{Z}_Q - \mathsf{Z} \\ (\mathbf{a}_{i-(2^{32}-\mathsf{Z}_Q-\mathsf{Z}_I)}, R) & \text{if } 2^{32} - \mathsf{Z}_Q - \mathsf{Z}_I \le i < 2^{32} - \mathsf{Z}_Q - \mathsf{Z}_I + |\mathbf{a} \\ (0, R) & \text{if } 2^{32} - \mathsf{Z}_Q - \mathsf{Z}_I + |\mathbf{a}| \le i < 2^{32} - \mathsf{Z}_Q - \mathsf{Z}_I + P(|\mathbf{a} \\ (0, \varnothing) & \text{otherwise} \end{cases}$$

$$\forall i \in \mathbb{N}_{16} : \omega_i = \begin{cases} 2^{32} - 2^{16} & \text{if } i = 1 \\ 2^{32} - 2\mathsf{Z}_Q - \mathsf{Z}_I & \text{if } i = 2 \\ 2^{32} - \mathsf{Z}_Q - \mathsf{Z}_I & \text{if } i = 10 \\ |\mathbf{a}| & \text{if } i = 11 \\ 0 & \text{otherwise} \end{cases} \tag{226}$$

A.7. **Argument Invocation Definition.** The four instances where the PVM is utilized each expect to be able to pass argument data in and receive some return data back. We thus define the common PVM program-argument invocation function Ψ_M:

(227)

$$\Psi_M \colon \begin{cases} (\mathbb{Y}, \mathbb{N}, \mathbb{N}_G, \mathbb{Y}_{:\mathsf{Z}_I}, \Omega_X, X) \to ((\mathbb{N}_G, \mathbb{Y}) \cup \{\lightning, \infty\}, X) \\ \qquad (\mathbf{p}, \imath, \xi, \mathbf{a}, f, \mathbf{x}) \mapsto \begin{cases} (\lightning, \mathbf{x}) & \text{if } Y(\mathbf{p}) = \varnothing \\ R(\Psi_H(\mathbf{c}, \imath, \xi, \omega, \mu, f, \mathbf{x})) & \text{if } Y(\mathbf{p}) = (\mathbf{c}, \omega, \mu) \end{cases} \end{cases}$$

(228)

$$\text{where } R \colon (\zeta, \imath', \xi', \omega', \mu', \mathbf{x}) \mapsto \begin{cases} (\zeta, \mathbf{x}) & \text{if } \zeta = \infty \\ (\xi', \boldsymbol{\mu}'_{\omega'_{10} \cdots + \omega'_{11}}) & \text{if } \zeta = \blacksquare \wedge \mathbb{Z}_{\omega'_{10} \cdots + \omega'_{11}} \subset \mathbb{V}_{\mu'} \\ (\lightning, \mathbf{x}) & \text{otherwise} \end{cases}$$

Appendix B. Virtual Machine Invocations

B.1. **Host-Call Result Constants.**

`NONE` = $2^{32} - 1$**:** The return value indicating an item does not exist.

`OOB` = $2^{32} - 2$**:** The return value for when a memory index is provided for reading/writing which is not accessible.

`WHO` = $2^{32} - 3$**:** Index unknown.

`FULL` = $2^{32} - 4$**:** Storage full.

`CORE` = $2^{32} - 5$**:** Core index unknown.

`CASH` = $2^{32} - 6$**:** Insufficient funds.

`LOW` = $2^{32} - 7$**:** Gas limit too low.

`HIGH` = $2^{32} - 8$**:** Gas limit too high.

`WHAT` = $2^{32} - 9$**:** Name unknown.

`HUH` = $2^{32} - 10$**:** The item is already solicited or cannot be forgotten.

`OK` = 0**:** The return value indicating general success.

Inner PVM invocations have their own set of result codes:

`HALT` = 0**:** The invocation completed and halted normally.

`PANIC` = $2^{32} - 12$**:** The invocation completed with a panic.

`FAULT` = $2^{32} - 13$**:** The invocation completed with a page fault.

`HOST` = $2^{32} - 14$**:** The invocation completed with a host-call fault.

Note return codes for a host-call-request exit are any non-zero value less than $2^{32} - 13$.

B.2. **Is-Authorized Invocation.** The Is-Authorized invocation is the first and simplest of the four, being totally stateless. It provides only a single host-call function, Ω_G for determining the amount of gas remaining. It accepts as arguments the work-package as a whole, $\mathbf{p}$ and the core on which it should be executed, c. Formally, it is defined as Ψ_I:

$$\Psi_I \colon \begin{cases} (\mathbb{P}, \mathbb{N}_C) \to \mathbb{Y} \cup \mathbb{J} \\ \quad (\mathbf{p}, c) \mapsto \begin{cases} \mathbf{r} & \text{otherwise if } \mathbf{r} \in \{\infty, \lightning\} \\ \mathbf{o} & \text{otherwise if } \mathbf{r} = (g, \mathbf{o}) \\ & \text{where } (\mathbf{r}, \varnothing) = \Psi_M(\mathbf{p}_\mathbf{c}, 0, \mathsf{G}_I, \mathcal{E}(p, c), F, \varnothing) \end{cases} \end{cases} \tag{229}$$

$$F \colon \begin{cases} (\mathbb{Y}_\$, \mathbb{N}_G, [\![\mathbb{N}_R]\!]_6, \mathbb{M}) \to (\mathbb{Z}_G, [\![\mathbb{N}_R]\!]_2, \mathbb{M}) \\ \quad (\mathbf{n}, \xi, \omega, \mu) \mapsto \begin{cases} \Omega_G(\xi, \omega, \mu) & \text{if } \mathbf{n} = \$\mathsf{gas} \\ (\xi - 10, [\mathtt{WHAT}, \omega_1, \dots], \mu) & \text{otherwise} \end{cases} \end{cases} \tag{230}$$

Note for the Is-Authorized host-call dispatch function F in equation 230, we elide the host-call context since, being essentially stateless, it is always $\varnothing$.

B.3. **Refine Invocation.** We define the Refine service-account invocation function as Ψ_R. It is stateless, with the only exceptions being the ability to make a historical lookup and the ability to create inner instances of the PVM.

The historical-lookup host-call function, Ω_H, is designed to give the same result regardless of the state of the chain for any time when auditing may occur (which we bound to be less than two epochs from being accumulated). However, the lookup anchor may be up to L timeslots before the recent history and therefore adds to the potential age at the time of audit. We therefore set $\mathsf{D} = 28,800$, a safe amount of 48 hours.

The inner PVM invocation host-calls, meanwhile, depend on an integrated PVM type, which we shall denote *pvm*. It holds some program code, instruction counter and RAM:

$$\mathbf{M} \equiv (\mathbf{p} \in \mathbb{Y}, \mathbf{u} \in \mathbb{M}, i \in \mathbb{N}_R) \tag{231}$$

The Refine invocation function implicitly draws upon some recent head state δ and explicitly accepts the work payload, $\mathbf{y}$ together with the service index which is the subject of refinement s, the prediction of the hash of that service's code c at the time of reporting, the hash of the containing work-package p, the refinement context $\mathbf{x}$ and the authorizer hash a together with its output $\mathbf{o}$. It results in either some error $\mathbb{J}$ or some refinement output blob. Formally:

(232)

$$\Psi_R \colon \begin{cases} (\mathbb{H}, \mathbb{N}_G, \mathbb{N}_S, \mathbb{H}, \mathbb{Y}, \mathbb{X}, \mathbb{H}, \mathbb{Y}) \to \mathbb{Y} \cup \mathbb{J} \\ \qquad (c, g, s, p, \mathbf{y}, \mathbf{x}, a, \mathbf{o}) \mapsto \begin{cases} \texttt{BAD} & \text{if } s \notin \mathcal{K}(\delta) \vee H(\delta[s], \mathbf{x}_t, c) = \varnothing \\ \texttt{BIG} & \text{otherwise if } |H(\delta[s], \mathbf{x}_t, c)| > \mathsf{S} \\ & \text{otherwise let } (\mathbf{r}, \varnothing) = \Psi_M(H(\delta[s], \mathbf{x}_t, c), 1, g, \mathcal{E}(s, p \\ \mathbf{r} & \quad \text{if } \mathbf{r} \in \{\infty, \lightning\} \\ \mathbf{u} & \quad \text{if } \mathbf{r} = (g, \mathbf{u}) \end{cases} \end{cases}$$

(233)
$$F: \begin{cases} (\mathbb{Y}_{\$}, \mathbb{N}_G, [\![\mathbb{N}_R]\!]_6, \mathbb{M}, \mathbb{D}\langle\mathbb{N} \to \mathbf{M}\rangle) \to (\mathbb{N}_G, [\![\mathbb{N}_R]\!]_2, \mathbb{M}, \mathbb{D}\langle\mathbb{N} \to \mathbf{M}\rangle) \\ (\mathbf{n}, \xi, \omega, \mu, \mathbf{m}) \mapsto \begin{cases} \Omega_H(\xi, \omega, \mu, \mathbf{m}, s, \delta, \mathbf{x}_t) & \text{if } \mathbf{n} = \$\texttt{lookup} \\ \Omega_G(\xi, \omega, \mu, \mathbf{m}) & \text{if } \mathbf{n} = \$\texttt{gas} \\ \Omega_M(\xi, \omega, \mu, \mathbf{m}) & \text{if } \mathbf{n} = \$\texttt{machine} \\ \Omega_P(\xi, \omega, \mu, \mathbf{m}) & \text{if } \mathbf{n} = \$\texttt{peek} \\ \Omega_O(\xi, \omega, \mu, \mathbf{m}) & \text{if } \mathbf{n} = \$\texttt{poke} \\ \Omega_K(\xi, \omega, \mu, \mathbf{m}) & \text{if } \mathbf{n} = \$\texttt{invoke} \\ \Omega_X(\xi, \omega, \mu, \mathbf{m}) & \text{if } \mathbf{n} = \$\texttt{expunge} \\ (\xi - 10, [\texttt{WHAT}, \omega_1, \dots], \mu) & \text{otherwise} \end{cases} \end{cases}$$

Note for the Refine host-call dispatcher F in equation 233, we elide the host-call context since, being essentially stateless, it is always $\varnothing$.

B.4. **Accumulate Invocation.** Since this is a transition which can directly affect a substantial amount of on-chain state, our invocation context is accordingly complex. It is a tuple with elements for each of the aspects of state which can be altered through this invocation and beyond the account of the service itself includes the deferred transfer list and several dictionaries for alterations to preimage lookup state, core assignments, validator key assignments, newly created accounts and alterations to account privilege levels.

Formally, we define our result context to be $\mathbf{X}$, and our invocation context to be a pair of these contexts, $\mathbf{X}\times\mathbf{X}$, with one dimension being the regular dimension and generally named $\mathbf{x}$ and the other being the exceptional dimension and being named $\mathbf{y}$. The only function which actually alters this second dimension is `checkpoint`, Ω_C and so it is rarely seen.

We track both regular and exceptional dimensions within our context mutator, but collapse the result of the invocation to one or the other depending on whether the termination was regular or exceptional (i.e. out-of-gas or panic).

(234)
$$\mathbf{X} \equiv \begin{pmatrix} \mathbf{s} \in \mathbb{A}^{?}, & \mathbf{c} \in [[\mathbb{H}]_{\mathsf{Q}}]_{\mathsf{C}}, & \mathbf{v} \in \mathbb{K}_{\mathsf{V}}, & i \in \mathbb{N}_S, \\ \mathbf{t} \in [\mathbb{T}], & \mathbf{n} \in \mathbb{D}\langle\mathbb{N}_S \to \mathbb{A}\rangle, & p \in (m \in \mathbb{N}_S, a \in \mathbb{N}_S, v \in \mathbb{N}_S) & \end{pmatrix}$$

We define Ψ_A, the Accumulation invocation function as:

(235)
$$\Psi_A : \begin{cases} (\mathbb{D}\langle \mathbb{N}_S \to \mathbb{A} \rangle, \mathbb{N}_S, \mathbb{N}_G, [\![\mathbb{O}]\!]) & \to \mathbf{X} \times (r \in \mathbb{H}?) \\ (\delta^\dagger, s, g, \mathbf{o}) & \mapsto \begin{cases} \delta^\dagger[s] & \text{if } \delta^\dagger[s]_{\mathbf{c}} = \varnothing \\ C(\Psi_M(\delta^\dagger[s]_{\mathbf{c}}, 2, g, \mathcal{E}(\mathbf{o}), F, I(\delta^\dagger[s], s))) & \text{otherwise} \end{cases} \end{cases}$$

(236)
$$I(\mathbf{a} \in \mathbb{A}, s \in \mathbb{N}_S) \equiv (\mathbf{x}, \mathbf{y}) \text{ where } \begin{cases} \mathbf{x} = \mathbf{y} = (\mathbf{s} \colon \mathbf{a},\ \mathbf{t} \colon [\,],\ i,\ p \colon \chi,\ \mathbf{c} \colon \varphi,\ \mathbf{v} \colon \iota,\ \mathbf{n} \colon \varnothing \\ i = \text{check}((\mathcal{E}_4^{-1}(\mathcal{H}(s, \boldsymbol{\eta}'_0, \mathbf{H}_t)) \bmod (2^{32} - \end{cases}$$

(237)

$$F(\mathbf{n}, \xi, \omega, \mu, (\mathbf{x}, \mathbf{y})) \equiv \begin{cases} G(\Omega_R(\xi, \omega, \mu, \mathbf{x}_\mathbf{s}, s, \delta^\dagger), (\mathbf{x}, \mathbf{y})) & \text{if } \mathbf{n} = \$\texttt{read} \\ G(\Omega_W(\xi, \omega, \mu, \mathbf{x}_\mathbf{s}), (\mathbf{x}, \mathbf{y})) & \text{if } \mathbf{n} = \$\texttt{write} \\ G(\Omega_L(\xi, \omega, \mu, s, \delta^\dagger), (\mathbf{x}, \mathbf{y})) & \text{if } \mathbf{n} = \$\texttt{lookup} \\ G(\Omega_G(\xi, \omega, \mu), (\mathbf{x}, \mathbf{y})) & \text{if } \mathbf{n} = \$\texttt{gas} \\ G(\Omega_I(\xi, \omega, \mu, \mathbf{x}_\mathbf{s}, s, \delta^\dagger), (\mathbf{x}, \mathbf{y})) & \text{if } \mathbf{n} = \$\texttt{info} \\ \Omega_E(\xi, \omega, \mu, (\mathbf{x}, \mathbf{y})) & \text{if } \mathbf{n} = \$\texttt{empower} \\ \Omega_A(\xi, \omega, \mu, (\mathbf{x}, \mathbf{y})) & \text{if } \mathbf{n} = \$\texttt{assign} \\ \Omega_D(\xi, \omega, \mu, (\mathbf{x}, \mathbf{y})) & \text{if } \mathbf{n} = \$\texttt{designate} \\ \Omega_C(\xi, \omega, \mu, (\mathbf{x}, \mathbf{y})) & \text{if } \mathbf{n} = \$\texttt{checkpoint} \\ \Omega_N(\xi, \omega, \mu, (\mathbf{x}, \mathbf{y})) & \text{if } \mathbf{n} = \$\texttt{new} \\ \Omega_U(\xi, \omega, \mu, (\mathbf{x}, \mathbf{y}), s) & \text{if } \mathbf{n} = \$\texttt{upgrade} \\ \Omega_T(\xi, \omega, \mu, (\mathbf{x}, \mathbf{y}), s, \delta^\dagger) & \text{if } \mathbf{n} = \$\texttt{transfer} \\ \Omega_Q(\xi, \omega, \mu, (\mathbf{x}, \mathbf{y}), s) & \text{if } \mathbf{n} = \$\texttt{quit} \\ \Omega_S(\xi, \omega, \mu, (\mathbf{x}, \mathbf{y}), \mathbf{H}_t) & \text{if } \mathbf{n} = \$\texttt{solicit} \\ \Omega_F(\xi, \omega, \mu, (\mathbf{x}, \mathbf{y}), \mathbf{H}_t) & \text{if } \mathbf{n} = \$\texttt{forget} \\ (\xi - 10, [\texttt{WHAT}, \omega_1, \dots], \mu, \mathbf{x}) & \text{otherwise} \end{cases}$$

(238)

$$G((\xi', \omega', \mu', \mathbf{s}'), (\mathbf{x}, \mathbf{y})) \equiv (\xi', \omega', \mu', (\mathbf{x}, \mathbf{y})) \text{ where } \mathbf{x} = \mathbf{x}' \text{ except } \mathbf{x}_\mathbf{s} = \mathbf{s}'$$

(239)

$$C((\mathbf{x} \in \mathbf{X}, \mathbf{y} \in \mathbf{X}), \mathbf{o} \in \mathbb{Y} \cup \{\infty, \lightning\}) \equiv \begin{cases} \mathbf{x} \times (r \vdots \mathbf{o}) & \text{if } \mathbf{o} \in \mathbb{H} \\ \mathbf{x} \times (r \vdots \varnothing) & \text{if } \mathbf{o} \in \mathbb{Y} \setminus \mathbb{H} \\ \mathbf{y} \times (r \vdots \varnothing) & \text{if } \mathbf{o} \in \{\infty, \lightning\} \end{cases}$$

The mutator F governs how this context will alter for any given parameterization, and the collapse function C selects one of the two dimensions of context depending on whether the virtual machine's halt was regular or exceptional.

The initializer function I maps some service account $\mathbf{s}$ along with its index s to yield a mutator context such that no alterations to state are implied (beyond those already inherent in $\mathbf{s}$) in either exit scenario. Note that the component a utilizes

the random accumulator η_0 and the block's timeslot $\mathbf{H}_t$ to create a deterministic sequence of identifiers which are extremely likely to be unique.

Concretely, we create the identifier from the Blake2 hash of the identifier of the creating service, the current random accumulator η_0 and the block's timeslot. Thus, within a service's accumulation it is almost certainly unique, but it is not necessarily unique across all services, nor at all times in the past. We utilize a *check* function to find the first such index in this sequence which does not already represent a service:

$$\text{check}(i \in \mathbb{N}_S) \equiv \begin{cases} i & \text{if } i \notin \mathcal{K}(\delta^\dagger) \\ \text{check}((i - 2^8 + 1) \bmod (2^{32} - 2^9) + 2^8) & \text{otherwise} \end{cases} \tag{240}$$

NB In the highly unlikely event that a block executes to find that a single service index has inadvertently been attached to two different services, then the block is considered invalid. Since no service can predict the identifier sequence ahead of time, they cannot intentionally disadvantage the block author.

B.5. **On-Transfer Invocation.** We define the On-Transfer service-account invocation function as Ψ_T; it is somewhat similar to the Accumulation Invocation except that the only state alteration it facilitates are basic alteration to the storage of the subject account. No further transfers may be made, no privileged operations are possible, no new accounts may be created nor other operations done on the subject account itself. The function is defined as:

(241)

$$\Psi_T : \begin{cases} (\mathbb{D}\langle \mathbb{N}_S \to \mathbb{A} \rangle, \mathbb{N}_S, [\mathbb{T}]) & \to \mathbb{A} \\ (\delta^\ddagger, s, \mathbf{t}) & \mapsto \begin{cases} \mathbf{s} & \text{if } \mathbf{s_c} = \varnothing \vee \mathbf{t} = \\ \Psi_M(\mathbf{s_c}, 3, \sum_{r \in \mathbf{t}}(r_g), \mathcal{E}(\mathbf{t}), F, \mathbf{s}) & \text{otherwise} \end{cases} \end{cases}$$

(242)

$$\text{where } \mathbf{s} = \delta^\ddagger[s] \text{ except } \mathbf{s}_b = \delta^\ddagger[s]_b + \sum_{r \in \mathbf{t}} r_a$$

(243)
$$F(\mathbf{n}, \xi, \omega, \mu, \mathbf{s}) \equiv \begin{cases} \Omega_L(\xi, \omega, \mu, \mathbf{s}, s, \delta^\ddagger) & \text{if } \mathbf{n} = \$\texttt{lookup} \\ \Omega_R(\xi, \omega, \mu, \mathbf{s}, s, \delta^\ddagger) & \text{if } \mathbf{n} = \$\texttt{read} \\ \Omega_W(\xi, \omega, \mu, \mathbf{s}) & \text{if } \mathbf{n} = \$\texttt{write} \\ \Omega_G(\xi, \omega, \mu) & \text{if } \mathbf{n} = \$\texttt{gas} \\ \Omega_I(\xi, \omega, \mu, \mathbf{s}, s, \delta^\ddagger) & \text{if } \mathbf{n} = \$\texttt{info} \\ (\xi - 10, [\texttt{WHAT}, \omega_1, \dots], \mu, \mathbf{s}) & \text{otherwise} \end{cases}$$

NB When considering the mutator functions Ω_R and Ω_I, the final arguments passed are both the post-accumulation accounts state, $\delta^\ddagger$. Within the function, this parameter however is denoted simply $\mathbf{d}$. This is intentional and avoids potential confusion since the functions are also utilized for the Accumulation Invocation where the argument is $\delta^\dagger$.

B.6. **General Functions.** This defines a number of functions broadly of the form $(\xi' \in \mathbb{Z}_G, \omega' \in [\mathbb{N}_R]_2, \mu', \mathbf{s}') = \Omega_\square(\xi \in \mathbb{N}_G, \omega \in [\mathbb{N}_R]_6, \mu \in \mathbb{M}, \mathbf{s} \in \mathbf{A}, \dots)$. Functions which have a result component which is equivalent to the corresponding argument may have said components elided in the description. Functions may also depend upon particular additional parameters.

Unlike the Accumulate functions in appendix B.7, these do not mutate an accumulation context, but merely a service account $\mathbf{s}$.

The `gas` function, Ω_G has a parameter list suffixed with an ellipsis to denote that any additional parameters may be taken and are provided transparently into its result. This allows it to be easily utilized in multiple PVM invocations.

(244)
$$\xi' \equiv \xi - g$$

(245)
$$(\omega', \mu', \mathbf{s}') \equiv \begin{cases} (\omega, \mu, \mathbf{s}) & \text{if } \xi < g \\ (\omega, \mu, \mathbf{s}) \text{ except as indicated below} & \text{otherwise} \end{cases}$$

Function Identifier Gas usage	**Mutations**
$\Omega_L(\xi, \omega, \mu, \mathbf{s}, s, \mathbf{d})$ `lookup` $g = 10$	$\text{let } \mathbf{a} = \begin{cases} \mathbf{s} & \text{if } \omega_0 \in \{s, 2^{32}-1\} \\ \mathbf{d}[\omega_0] & \text{otherwise} \end{cases}$ $\text{let } [h_o, b_o, b_z] = \omega_{1..4}$ $\text{let } h = \begin{cases} \mathcal{H}(\mu_{h_o \cdots +32}) & \text{if } \mathbb{Z}_{h_o \cdots +32} \subset \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $\text{let } \mathbf{v} = \begin{cases} \mathbf{a_p}[h] & \text{if } \mathbf{a} \neq \varnothing \wedge h \in \mathcal{K}(\mathbf{a_p}) \\ \varnothing & \text{otherwise} \end{cases}$ $\forall i \in \mathbb{N}_{\min(b_z, \lvert\mathbf{v}\rvert)} : \mu'_{b_o+i} \equiv \begin{cases} \mathbf{v}_i & \text{if } \mathbf{v} \neq \varnothing \wedge \mathbb{Z}_{b_o \cdots +b_z} \subset \mathbb{V}^*_\mu \\ \mu_{b_o+i} & \text{otherwise} \end{cases}$ $\omega'_0 \equiv \begin{cases} \texttt{NONE} & \text{if } \mathbf{v} = \varnothing \\ \lvert\mathbf{v}\rvert & \text{otherwise} \end{cases} \quad \text{if } k \neq \nabla \wedge \mathbb{Z}_{b_o \cdots +b_z} \subset \mathbb{V}^*_\mu$ $\texttt{OOB} \quad \text{otherwise}$

Function Identifier Gas usage	**Mutations**
$\Omega_R(\xi, \omega, \mu, \mathbf{s}, s, \mathbf{d})$ `read` $g = 10$	$\text{let } \mathbf{a} = \begin{cases} \mathbf{s} & \text{if } \omega_0 \in \{s, 2^{32}-1\} \\ \mathbf{d}[\omega_0] & \text{otherwise if } \omega_0 \in \mathcal{K}(\mathbf{d}) \\ \varnothing & \text{otherwise} \end{cases}$ $\text{let } [k_o, k_z, b_o, b_z] = \omega_{1..5}$ $\text{let } k = \begin{cases} \mathcal{H}(\mathcal{E}_4(s) \frown \mu_{k_o \cdots + k_z}) & \text{if } \mathbb{Z}_{k_o \cdots + k_z} \subset \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $\text{let } \mathbf{v} = \begin{cases} \mathbf{a}_\mathbf{s}[k] & \text{if } \mathbf{a} \neq \varnothing \wedge k \in \mathcal{K}(\mathbf{a}_\mathbf{s}) \\ \varnothing & \text{otherwise} \end{cases}$ $\forall i \in \mathbb{N}_{\min(b_z, \lvert\mathbf{v}\rvert)} : \mu'_{b_o+i} \equiv \begin{cases} \mathbf{v}_i & \text{if } \mathbf{v} \neq \varnothing \wedge \mathbb{Z}_{b_o \cdots + b_z} \subset \mathbb{V}^*_\mu \\ \mu_{b_o+i} & \text{otherwise} \end{cases}$ $\omega'_0 \equiv \begin{cases} \begin{cases} \texttt{NONE} & \text{if } \mathbf{v} = \varnothing \\ \lvert\mathbf{v}\rvert & \text{otherwise} \end{cases} & \text{if } k \neq \nabla \wedge \mathbb{Z}_{b_o \cdots + b_z} \subset \mathbb{V}^*_\mu \\ \texttt{OOB} & \text{otherwise} \end{cases}$
$\Omega_W(\xi, \omega, \mu, \mathbf{s})$ `write` $g = 10$	$\text{let } [k_o, k_z, v_o, v_z] = \omega_{0..4}$ $\text{let } k = \begin{cases} \mathcal{H}(\mathcal{E}_4(s) \frown \mu_{k_o \cdots + k_z}) & \text{if } \mathbb{Z}_{k_o \cdots + k_z} \subset \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $\text{let } \mathbf{a} = \begin{cases} \mathbf{s} \text{ except } \begin{cases} \mathcal{K}(\mathbf{a}_\mathbf{s}) = \mathcal{K}(\mathbf{a}_\mathbf{s}) \setminus \{k\} & \text{if } v_z = 0 \\ \mathbf{a}_\mathbf{s}[k] = \mu_{v_o \cdots + v_z} & \text{otherwise} \end{cases} & \text{if } \mathbb{Z}_{v_o} \\ \nabla & \text{other} \end{cases}$ $\text{let } l = \begin{cases} \lvert\mathbf{s}_\mathbf{s}[k]\rvert & \text{if } k \in \mathcal{K}(\mathbf{s}_\mathbf{s}) \\ \texttt{NONE} & \text{otherwise} \end{cases}$ $(\omega'_0, \mathbf{s}') \equiv \begin{cases} (l, \mathbf{a}) & \text{if } k \neq \nabla \wedge \mathbf{a} \neq \nabla \wedge \mathbf{a}_t \leq \mathbf{a}_b \\ (\texttt{FULL}, \mathbf{s}) & \text{if } \mathbf{a}_t > \mathbf{a}_b \\ (\texttt{OOB}, \mathbf{s}) & \text{otherwise} \end{cases}$

Function Identifier Gas usage	**Mutations**
$\Omega_G(\xi, \omega, \dots)$ `gas` $g = 10$	$\omega'_0 \equiv \xi' \bmod 2^{32}$ $\omega'_1 \equiv \lfloor \xi' \div 2^{32} \rfloor$
$\Omega_I(\xi, \omega, \mu, \mathbf{s}, s, \mathbf{d})$ `info` $g = 10$	$\text{let } \mathbf{t} = \begin{cases} \mathbf{s} & \text{if } \omega_0 \in \{s, 2^{32}-1\} \\ (\mathbf{d} \cup \mathbf{x_n})[\omega_0] & \text{otherwise} \end{cases}$ $\text{let } o = \omega_1$ $\text{let } \mathbf{m} = \begin{cases} \mathcal{E}(\mathbf{t}_c, \mathbf{t}_b, \mathbf{t}_t, \mathbf{t}_g, \mathbf{t}_m, \mathbf{t}_l, \mathbf{t}_i) & \text{if } \mathbf{t} \neq \varnothing \\ \varnothing & \text{otherwise} \end{cases}$ $\forall i \in \mathbb{N}_{\lvert\mathbf{m}\rvert} : \mu'_{o+i} \equiv \begin{cases} \mathbf{m}_i & \text{if } \mathbf{m} \neq \varnothing \wedge \mathbb{Z}_{o \cdots + \lvert\mathbf{m}\rvert} \subset \mathbb{V}^*_\mu \\ \mu_{b_o+i} & \text{otherwise} \end{cases}$ $\omega'_0 \equiv \begin{cases} \texttt{OK} & \text{if } \mathbf{m} \neq \varnothing \wedge \mathbb{Z}_{o \cdots + \lvert\mathbf{m}\rvert} \subset \mathbb{V}^*_\mu \\ \texttt{NONE} & \text{if } \mathbf{m} = \varnothing \\ \texttt{OOB} & \text{otherwise} \end{cases}$

B.7. **Accumulate Functions.** This defines a number of functions broadly of the form $(\xi' \in \mathbb{Z}_G, \omega' \in [\mathbb{N}_R]_2, \mu', (\mathbf{x}', \mathbf{y}')) = \Omega_\square(\xi \in \mathbb{N}_G, \omega \in [\mathbb{N}_R]_6, \mu \in \mathbb{M}, (\mathbf{x} \in \mathbf{X}, \mathbf{y} \in \mathbf{X}), \dots)$. Functions which have a result component which is equivalent to the corresponding argument may have said components elided in the description. Functions may also depend upon particular additional parameters.

$$\xi' \equiv \xi - g \tag{246}$$

$$(\omega', \mu', \mathbf{x}', \mathbf{y}') \equiv \begin{cases} (\omega, \mu, \mathbf{x}, \mathbf{y}) & \text{if } \xi < g \\ (\omega, \mu, \mathbf{x}, \mathbf{y}) \text{ except as indicated below} & \text{otherwise} \end{cases} \tag{247}$$

Function Identifier Gas usage	**Mutations**
$\Omega_E(\xi, \omega, \mu, (\mathbf{x}, \mathbf{y}))$ `empower` $g = 10$	$(\mathbf{x}'_p)_m = \omega_0$ $(\mathbf{x}'_p)_a = \omega_1$ $(\mathbf{x}'_p)_v = \omega_2$
$\Omega_A(\xi, \omega, \mu, (\mathbf{x}, \mathbf{y}))$ `assign` $g = 10$	let $o = \omega_1$ let $\mathbf{c} = \begin{cases} [\mu_{o+32i\cdots+32} \mid i \in \mathbb{N}_{\mathsf{Q}}] & \text{if } \mathbb{Z}_{o\cdots+32\mathsf{Q}} \subset \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $(\omega'_0, \mathbf{x}') = \begin{cases} (\mathtt{OK}, \mathbf{x} \text{ except } \mathbf{x}'_{\mathbf{c}}[\omega_0] = \mathbf{c}) & \text{if } \omega_0 < \mathsf{C} \wedge \mathbf{c} \neq \nabla \\ (\mathtt{OOB}, \mathbf{x}) & \text{if } \mathbf{c} = \nabla \\ (\mathtt{CORE}, \mathbf{x}) & \text{otherwise} \end{cases}$
$\Omega_D(\xi, \omega, \mu, (\mathbf{x}, \mathbf{y}))$ `designate` $g = 10$	let $o = \omega_0$ let $\mathbf{v} = \begin{cases} [\mu_{o+176i\cdots+176} \mid i \in \mathbb{N}_{\mathsf{V}}] & \text{if } \mathbb{Z}_{o\cdots+176\mathsf{V}} \subset \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $(\omega'_0, \mathbf{x}') = \begin{cases} (\mathtt{OK}, \mathbf{x} \text{ except } \mathbf{x}'_{\mathbf{v}} = \mathbf{v}) & \text{if } \mathbf{v} \neq \nabla \\ (\mathtt{OOB}, \mathbf{x}) & \text{otherwise} \end{cases}$
$\Omega_C(\xi, \omega, \mu, (\mathbf{x}, \mathbf{y}))$ `checkpoint` $g = 10$	$\mathbf{y}' \equiv \mathbf{x}$ $\omega'_0 \equiv \xi' \bmod 2^{32}$ $\omega'_1 \equiv \lfloor \xi' \div 2^{32} \rfloor$

Function Identifier Gas usage	**Mutations**
$\Omega_N(\xi, \omega, \mu, (\mathbf{x}, \mathbf{y}))$ `new` $g = 10$	$\text{let } [o, l, g_l, g_h, m_l, m_h] = \omega_{0..6}$ $\text{let } c = \begin{cases} \mu_{o\cdots+32} & \text{if } \mathbb{N}_{o\cdots+32} \subset \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $\text{let } g = 2^{32} \cdot g_h + g_l$ $\text{let } m = 2^{32} \cdot m_h + m_l$ $\text{let } \mathbf{a} \in \mathbb{A} \cup \{\nabla\} = \begin{cases} (c, \mathbf{s} : \{\}, \mathbf{l} : \{(c, l) \mapsto []\}, b : \mathbf{a}_t, g, m) & \text{if } c \neq \nabla \\ \nabla & \text{otherwi} \end{cases}$ $\text{let } b = (\mathbf{x_s})_b - \mathbf{a}_t$ $(\omega'_0, \mathbf{x}'_i, \mathbf{x}'_\mathbf{n}, (\mathbf{x}'_\mathbf{s})_b) \equiv \begin{cases} (\mathbf{x}_i, \text{check}(\text{bump}(\mathbf{x}_i)), \mathbf{x_n} \cup \{\mathbf{x}_i \mapsto \mathbf{a}\}, b) & \text{if } \mathbf{a} \\ (\texttt{OOB}, \mathbf{x}_T) & \text{if } c \\ (\texttt{CASH}, \mathbf{x}_T) & \text{oth} \end{cases}$ $\text{where bump}(i \in \mathbb{N}_S) = 2^8 + (i - 2^8 + 42) \bmod (2^{32} - 2^9)$
$\Omega_U(\xi, \omega, \mu, (\mathbf{x}, \mathbf{y}), s)$ `upgrade` $g = 10$	$\text{let } [o, g_h, g_l, m_h, m_l] = \omega_{0..5}$ $\text{let } c = \begin{cases} \mu_{o\cdots+32} & \text{if } \mathbb{N}_{o\cdots+32} \subset \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $\text{let } g = 2^{32} \cdot g_h + g_l$ $\text{let } m = 2^{32} \cdot m_h + m_l$ $(\omega'_0, \mathbf{x}'[s]_c, \mathbf{x}'[s]_g, \mathbf{x}'[s]_m) \equiv \begin{cases} (\texttt{OK}, c, g, m) & \text{if } c \neq \nabla \\ (\texttt{OOB}, \mathbf{x}[s]_c, \mathbf{x}[s]_g, \mathbf{x}[s]_m) & \text{otherwise} \end{cases}$

Function Identifier Gas usage	**Mutations**
$\Omega_T(\xi, \omega, \mu, (\mathbf{x}, \mathbf{y}), s, \delta)$ `transfer` $g = 10 + \omega_1 + 2^{32} \cdot \omega_2$	$\text{let } (d, a_l, a_h, g_l, g_h, o) = \omega_{0..6},$ $\text{let } a = 2^{32} \cdot a_h + a_l$ $\text{let } g = 2^{32} \cdot g_h + g_l$ $\text{let } \mathbf{t} \in \mathbb{T} \cup \{\nabla\} = \begin{cases} (s, d, a, m, g) : m = \mathcal{E}^{-1}(\mu_{o\cdots+\mathsf{M}}) & \text{if } \mathbb{N}_{o\cdots+\mathsf{M}} \subset \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $\text{let } b = (\mathbf{x_s})_b - a$ $(\omega'_0, \mathbf{x'_t}, (\mathbf{x'_s})_b) \equiv \begin{cases} (\texttt{OOB}, \mathbf{x_t}, (\mathbf{x_s})_b) & \text{if } t = \nabla \\ (\texttt{WHO}, \mathbf{x_t}, (\mathbf{x_s})_b) & \text{otherwise if } d \notin \mathcal{K}(\delta \cup \mathbf{x_n}) \\ (\texttt{LOW}, \mathbf{x_t}, (\mathbf{x_s})_b) & \text{otherwise if } g < (\delta \cup \mathbf{x_n})[d]_g \\ (\texttt{HIGH}, \mathbf{x_t}, (\mathbf{x_s})_b) & \text{otherwise if } \xi < g \\ (\texttt{CASH}, \mathbf{x_t}, (\mathbf{x_s})_b) & \text{otherwise if } b < (\mathbf{x_s})_t \\ (\texttt{OK}, \mathbf{x_t} \pitchfork \mathbf{t}, b) & \text{otherwise} \end{cases}$
$\Omega_X(\xi, \omega, \mu, (\mathbf{x}, \mathbf{y}), s)$ `quit` $g = 10 + \omega_1 + 2^{32} \cdot \omega_2$	$\text{let } [d, o] = \omega_{0,1}$ $\text{let } a = (\mathbf{x_s})_b - (\mathbf{x_s})_t + \mathsf{B}_S$ $\text{let } g = \xi$ $\text{let } \mathbf{t} \in \mathbb{T} \cup \{\nabla, \varnothing\} = \begin{cases} \varnothing & \text{if } d \in \{s, 2^{32} - 1\} \\ (s, d, a, m, g) : m = \mathcal{E}^{-1}(\mu_{o\cdots+\mathsf{M}}) & \text{otherwise if } \mathbb{N}_{o\cdots+\mathsf{M}} \\ \nabla & \text{otherwise} \end{cases}$ $(\omega'_0, \mathbf{x'_s}, \mathbf{x'_t}) \equiv \begin{cases} (\texttt{OK}, \varnothing, \mathbf{x_t}),\ \textbf{virtual machine halts} & \text{if } \mathbf{t} = \varnothing \\ (\texttt{OOB}, \mathbf{x_t}, (\mathbf{x_s})_b) & \text{otherwi} \\ (\texttt{WHO}, \mathbf{x_t}, (\mathbf{x_s})_b) & \text{otherwi} \\ (\texttt{LOW}, \mathbf{x_t}, (\mathbf{x_s})_b) & \text{otherwi} \\ (\texttt{OK}, \varnothing, \mathbf{x_t} \pitchfork \mathbf{t}),\ \textbf{virtual machine halts} & \text{otherwi} \end{cases}$

Function Identifier Gas usage	**Mutations**
$\Omega_S(\xi, \omega, \mu, (\mathbf{x}, \mathbf{y}))$ `solicit` $g = 10$	let $[o, z] = \omega_{0,1}$ let $h = \begin{cases} \mu_{o\cdots+32} & \text{if } \mathbb{Z}_{o\cdots+32} \subset \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ let $\mathbf{a} = \begin{cases} \mathbf{x_s} \text{ except:} & \\ \quad \mathbf{a_l}[(h, z)] = [] & \text{if } h \neq \nabla \wedge (h, z) \notin (\mathbf{x_s})_\mathbf{l} \\ \quad \mathbf{a_l}[(h, z)] = (\mathbf{x_s})_\mathbf{l}[(h, z)] \mathbin{+\!\!+} t & \text{if } (\mathbf{x_s})_\mathbf{l}[(h, z)] = [x, y] \\ \nabla & \text{otherwise} \end{cases}$ $(\omega'_0, \mathbf{x}'_\mathbf{s}) \equiv \begin{cases} (\texttt{OOB}, \mathbf{x_s}) & \text{if } h = \nabla \\ (\texttt{HUH}, \mathbf{x_s}) & \text{otherwise if } \mathbf{a} = \nabla \\ (\texttt{FULL}, \mathbf{x_s}) & \text{otherwise if } \mathbf{a}_b < \mathbf{a}_t \\ (\texttt{OK}, \mathbf{a}) & \text{otherwise} \end{cases}$
$\Omega_F(\xi, \omega, \mu, (\mathbf{x}, \mathbf{y}), t)$ `forget` $g = 10$	let $[o, z] = \omega_{0,1}$ let $h = \begin{cases} \mu_{o\cdots+32} & \text{if } \mathbb{Z}_{o\cdots+32} \subset \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ let $\mathbf{a} = \begin{cases} \mathbf{x_s} \text{ except:} & \\ \quad \left.\begin{array}{l} \mathcal{K}(\mathbf{a_l}) = \mathcal{K}((\mathbf{x_s})_\mathbf{l}) \setminus \{(h, z)\} \ , \\ \mathcal{K}(\mathbf{a_p}) = \mathcal{K}((\mathbf{x_s})_\mathbf{p}) \setminus \{h\} \end{array}\right\} & \text{if } (\mathbf{x_s})_\mathbf{l}[h, z] \in \{[], [x, y\\ \quad \mathbf{a_l}[h, z] = (\mathbf{x_s})_\mathbf{l}[h, z] \mathbin{+\!\!+} t & \text{if } \lvert(\mathbf{x_s})_\mathbf{l}[h, z]\rvert = 1 \\ \quad \mathbf{a_l}[h, z] = [(\mathbf{x_s})_\mathbf{l}[h, z]_2, t] & \text{if } (\mathbf{x_s})_\mathbf{l}[h, z] = [x, y, w\\ \nabla & \text{otherwise} \end{cases}$ $(\omega'_0, \mathbf{x}'_\mathbf{s}) \equiv \begin{cases} (\texttt{OOB}, \mathbf{x_s}) & \text{if } h = \nabla \\ (\texttt{HUH}, \mathbf{x_s}) & \text{otherwise if } \mathbf{a} = \nabla \\ (\texttt{OK}, \mathbf{a}) & \text{otherwise} \end{cases}$

Function Identifier Gas usage	**Mutations**
$\Omega_V(\xi, \omega, \mu, (\mathbf{x}, \mathbf{y}))$ `invoke` $g = 10$	let $[o, z] = \omega_{0,1}$ let $h = \begin{cases} \mu_{o\cdots+32} & \text{if } \mathbb{Z}_{o\cdots+32} \subset \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ let $\mathbf{a} = \begin{cases} \mathbf{x_s} \text{ except:} & \\ \quad \mathbf{a_l}[(h, z)] = [] & \text{if } h \neq \nabla \wedge (h, z) \notin (\mathbf{x_s})_\mathbf{l} \\ \quad \mathbf{a_l}[(h, z)] = (\mathbf{x_s})_\mathbf{l}[(h, z)] \pplus t & \text{if } (\mathbf{x_s})_\mathbf{l}[(h, z)] = [x, y] \\ \nabla & \text{otherwise} \end{cases}$ $(\omega'_0, \mathbf{x}'_\mathbf{s}) \equiv \begin{cases} (\texttt{OOB}, \mathbf{x_s}) & \text{if } h = \nabla \\ (\texttt{HUH}, \mathbf{x_s}) & \text{otherwise if } \mathbf{a} = \nabla \\ (\texttt{FULL}, \mathbf{x_s}) & \text{otherwise if } \mathbf{a}_b < \mathbf{a}_t \\ (\texttt{OK}, \mathbf{a}) & \text{otherwise} \end{cases}$

B.8. **Refine Function.** These assume some refine context $\mathbf{M} \in \mathbb{D}\langle\mathbb{N}_R \to (\mathbb{Y}, \mathbb{M}, [\![\mathbb{N}_R]\!]_{13}, \mathbb{N}_R)\rangle$ initially empty and which tends to be stated as $\mathbf{m}$ (posterior $\mathbf{m}'$).

Function Identifier Gas usage	**Mutations**
$\Omega_H(\xi, \omega, \mu, \mathbf{m}, s, \delta, t)$ `historical_lookup` $g = 10$	$\text{let } \mathbf{a} = \begin{cases} \delta[s] & \text{if } \omega_0 = 2^{32}-1 \wedge s \in \mathcal{K}(\delta) \\ \delta[\omega_0] & \text{if } \omega_0 \in \mathcal{K}(\delta) \\ \varnothing & \text{otherwise} \end{cases}$ $\text{let } [h_o, b_o, b_z] = \omega_{1..4}$ $\text{let } h = \begin{cases} \mathcal{H}(\mu_{h_o \cdots +32}) & \text{if } \mathbb{Z}_{h_o \cdots +32} \subset \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $\text{let } \mathbf{v} = H(\mathbf{a}, t, h)$ $\forall i \in \mathbb{N}_{\min(b_z, \lvert\mathbf{v}\rvert)} : \mu'_{b_o+i} \equiv \begin{cases} \mathbf{v}_i & \text{if } \mathbf{v} \neq \varnothing \wedge \mathbb{Z}_{b_o \cdots +b_z} \subset \mathbb{V}^*_\mu \\ \mu_{b_o+i} & \text{otherwise} \end{cases}$ $\omega'_0 \equiv \begin{cases} \begin{cases} \lvert\mathbf{v}\rvert & \text{if } \mathbf{v} \neq \varnothing \\ \texttt{NONE} & \text{otherwise} \end{cases} & \text{if } k \neq \nabla \wedge \mathbb{Z}_{b_o \cdots +b_z} \subset \mathbb{V}^*_\mu \\ \texttt{OOB} & \text{otherwise} \end{cases}$
$\Omega_M(\xi, \omega, \mu, \mathbf{m})$ `machine` $g = 10$	$\text{let } [p_o, p_z, i] = \omega_{0...2}$ $\text{let } \mathbf{p} = \begin{cases} \mu_{p_o \cdots +p_z} & \text{if } \mathbb{Z}_{p_o \cdots +p_z} \subset \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $\text{let } n = \min(n \in \mathbb{N}, n \notin \mathcal{K}(\mathbf{m}))$ $\text{let } \mathbf{u} = (\mathbf{V} \vdots [0, 0, \dots], \mathbf{A} \vdots [\varnothing, \varnothing, \dots])$ $(\omega'_0, \mathbf{m}) \equiv \begin{cases} (\texttt{OOB}, \mathbf{m}) & \text{if } \mathbf{p} = \nabla \\ (n, \mathbf{m} \cup \{n \mapsto (\mathbf{p}, \mathbf{u}, i)\}) & \text{otherwise} \end{cases}$

Function Identifier Gas usage	**Mutations**
$\Omega_P(\xi, \omega, \mu, \mathbf{m})$ `peek` $g = 10$	$\text{let } [n, a, b, l] = \omega_{0\ldots4}$ $\text{let } \mathbf{s} = \begin{cases} \varnothing & \text{if } n \notin \mathcal{K}(\mathbf{m}) \\ \nabla & \text{if } \mathbb{N}_{b\cdots+i} \notin \mathbb{V}_{\mathbf{m}[n]_\mathbf{u}} \\ \mathbf{m}[n]_{\mathbf{u}_{b\cdots+i}} & \text{otherwise} \end{cases}$ $(\omega'_0, \mu') \equiv \begin{cases} (\texttt{OOB}, \mu) & \text{if } \mathbf{s} = \nabla \\ (\texttt{WHO}, \mu) & \text{if } \mathbf{s} = \varnothing \\ (\texttt{OK}, \mu') \text{ where } \mu' = \mu \text{ except } \mu'_{a\cdots+l} = \mathbf{s} & \text{otherwise} \end{cases}$
$\Omega_O(\xi, \omega, \mu, \mathbf{m})$ `poke` $g = 10$	$\text{let } [n, a, b, l] = \omega_{0\ldots4}$ $\text{let } \mathbf{u} = \begin{cases} \mathbf{m}[n]_\mathbf{u} & \text{if } n \in \mathcal{K}(\mathbf{m}) \\ \nabla & \text{otherwise} \end{cases}$ $\text{let } \mathbf{s} = \begin{cases} \mu_{a\cdots+i} & \text{if } \mathbb{N}_{a\cdots+i} \in \mathbb{V}_\mathbf{u} \\ \nabla & \text{otherwise} \end{cases}$ $\text{let } \mathbf{u}' = \mathbf{u} \text{ except } \begin{cases} (\mathbf{u}'_\mathbf{V})_{b\cdots+l} = \mathbf{s} \\ (\mathbf{u}'_\mathbf{A})_{b\cdots+l} = [\mathrm{W}, \mathrm{W}, \ldots] \end{cases}$ $(\omega'_0, \mathbf{m}') \equiv \begin{cases} (\texttt{OOB}, \mathbf{m}) & \text{if } \mathbf{s} = \nabla \\ (\texttt{WHO}, \mathbf{m}) & \text{otherwi} \\ (\texttt{OK}, \mathbf{m}'), \text{ where } \mathbf{m}' = \mathbf{m} \text{ except } \mathbf{m}'[n]_\mathbf{u} = \mathbf{u}' & \text{otherwi} \end{cases}$

Function Identifier Gas usage	**Mutations**
$\Omega_K(\xi, \omega, \mu, \mathbf{m})$ `invoke` $g = 10$	$\text{let } [n, o] = \omega_{0\ldots2}$ $\text{let } (g, \mathbf{w}) = \begin{cases} (\mathcal{E}_8^{-1}(\mu_{o\cdots+8}), [\mathcal{E}_4^{-1}(\mu_{o+8+4x\cdots+4}) \mid x \in \mathbb{N}_{13}]) & \text{if } \mathbb{N}_{o\cdots+} \\ (\nabla, \nabla) & \text{otherw} \end{cases}$ $\text{let } (c, i', g', \mathbf{w}', \mathbf{u}') = \Psi(\mathbf{m}[n]_{\mathbf{p}}, \mathbf{m}[n]_i, g, \mathbf{w}, \mathbf{m}[n]_{\mathbf{u}})$ $\text{let } \mu^* = \mu \text{ except } \mu^*_{o\cdots+60} = \mathcal{E}_8(g') \frown \mathcal{E}([\mathcal{E}_4(x) \mid x \in \mathbf{w}'])$ $\text{let } \mathbf{m}^* = \mathbf{m} \text{ except } \begin{cases} \mathbf{m}^*[n]_{\mathbf{u}} = \mathbf{u}' \\ \mathbf{m}^*[n]_i = \begin{cases} i' + 1 & \text{if } c \in \{\hbar\} \times \mathbb{N}_R \\ i' & \text{otherwise} \end{cases} \end{cases}$ $(\omega'_0, \omega'_1, \mu', \mathbf{m}') \equiv \begin{cases} (\texttt{OOB}, \omega_1, \mu, \mathbf{m}) & \text{if } g = \nabla \\ (\texttt{WHO}, \omega_1, \mu, \mathbf{m}) & \text{otherwise if } n \notin \mathbf{m} \\ (\texttt{HOST}, h, \mu^*, \mathbf{m}^*) & \text{otherwise if } c = \hbar \times h \\ (\texttt{FAULT}, x, \mu^*, \mathbf{m}^*) & \text{otherwise if } c = \lightning \times x \\ (\texttt{PANIC}, \omega_1, \mu^*, \mathbf{m}^*) & \text{otherwise if } c = \lightning \\ (\texttt{HALT}, \omega_1, \mu^*, \mathbf{m}^*) & \text{otherwise if } c = \blacksquare \end{cases}$
$\Omega_X(\xi, \omega, \mu, \mathbf{m})$ `expunge` $g = 10$	$\text{let } n = \omega_0$ $(\omega'_0, \mathbf{m}') \equiv \begin{cases} (\texttt{WHO}, \mathbf{m}) & \text{if } n \neq \mathcal{K}(\mathbf{m}) \\ (\mathbf{m}[n]_i, \mathbf{m} \setminus n) & \text{otherwise} \end{cases}$

Appendix C. Serialization Codec

C.1. **Common Terms.** Our codec function $\mathcal{E}$ is used to serialize some term into a sequence of octets. We define the deserialization function $\mathcal{E}^{-1} = \mathcal{E}^{-1}$ as the inverse of $\mathcal{E}$ and able to decode some sequence into the original value. The codec is designed such that exactly one value is encoded into any given sequence of octets, and in cases where this is not desirable then we use special codec functions.

C.1.1. *Trivial Encodings.* We define the serialization of $\varnothing$ as the empty sequence:

$$\mathcal{E}(\varnothing) \equiv [] \tag{248}$$

We also define the serialization of an octet-sequence as itself:

$$\mathcal{E}(x \in \mathbb{Y}) \equiv x \tag{249}$$

We define anonymous tuples to be encoded as the concatenation of their encoded elements:

$$\mathcal{E}((a, b, \dots)) \equiv \mathcal{E}(a) \frown \mathcal{E}(b) \frown \dots \tag{250}$$

Passing multiple arguments to the serialization functions is equivalent to passing a tuple of those arguments. Formally:

$$\mathcal{E}(a, b, c, \dots) \equiv \mathcal{E}((a, b, c, \dots)) \tag{251}$$

C.1.2. *Integer Encoding.* We first define the trivial integer serialization functions which are subscripted by the number of octets of the final sequence. Values are encoded in a regular little-endian fashion. Formally:

$$\mathcal{E}_{l \in \mathbb{N}}: \begin{cases} \mathbb{N}_{2^{8l}} \to \mathbb{Y}_l \\ \quad x \mapsto \begin{cases} [] & \text{if } l = 0 \\ [x \bmod 256] \frown \mathcal{E}_{l-1}\left(\left\lfloor \frac{x}{256} \right\rfloor\right) & \text{otherwise} \end{cases} \end{cases} \tag{252}$$

We also define the variable-size prefix 29-bit integer serialization function $\mathcal{E}_{4*}$:

$$\mathcal{E}_{4*}: \begin{cases} \mathbb{N}_{2^{29}} \to \mathbb{Y}_{1:4} \\ \quad x \mapsto \begin{cases} \left[2^8 - 2^{8-l} + \left\lfloor \frac{x}{2^{8l}} \right\rfloor\right] \frown \mathcal{E}_l(x \bmod 2^{8l}) & \text{if } \exists l \in \mathbb{N}_3 : 2^{7l} \le x < 2^{7(l+1)} \\ \left[2^8 - 2^5 + \left\lfloor \frac{x}{2^{24}} \right\rfloor\right] \frown \mathcal{E}_3(x \bmod 2^{24}) & \text{if } 2^{21} \le x < 2^{29} \end{cases} \end{cases} \tag{253}$$

We define general integer serialization, able to encode integers of up to 2^{64}, as:

$$\mathcal{E}: \begin{cases} \mathbb{N}_{2^{64}} \to \mathbb{Y}_{1:9} \\ \quad x \mapsto \begin{cases} [2^8 - 2^{8-l} + \lfloor \frac{x}{2^{8l}} \rfloor] \frown \mathcal{E}_l(x \bmod 2^{8l}) & \text{if } \exists l \in \mathbb{N}_8 : 2^{7l} \le x < 2^{7(l+1)} \\ [2^8 - 1] \frown \mathcal{E}_8(x) & \text{if } x < 2^{64} \end{cases} \end{cases} \tag{254}$$

C.1.3. *Sequence Encoding.* We define the sequence serialization function $\mathcal{E}(\llbracket T \rrbracket)$ for any T which is itself a subset of the domain of $\mathcal{E}$. We simply concatenate the serializations of each element in the sequence in turn:

$$\mathcal{E}([i_0, i_1, ...]) \equiv \mathcal{E}(i_0) \frown \mathcal{E}(i_1) \frown \dots \tag{255}$$

Thus, conveniently, fixed length octet sequences (e.g. hashes $\mathbb{H}$ and its variants) have an identity serialization.

C.1.4. *Discriminator Encoding.* When we have sets of heterogeneous items such as a union of different kinds of tuples or sequences of different length, we require a discriminator to determine the nature of the encoded item for successful deserialization. Discriminators are encoded as a general integer and are encoded immediately prior to the item.

We generally use a *length discriminator* which serializing sequence terms which have variable length (e.g. general blobs $\mathbb{Y}$ or unbound numeric sequences $\llbracket \mathbb{N} \rrbracket$) (though this is omitted in the case of fixed-length terms such as hashes $\mathbb{H}$).[18] In this case, we simply prefix the term its length prior to encoding. Thus, for some term $y \in (x \in \mathbb{Y}, \dots)$, we would generally define its serialized form to be $\mathcal{E}(|x|) \frown \mathcal{E}(x) \frown \dots$. To avoid repetition of the term in such cases, we define the notation $\updownarrow x$ to mean that the term of value x is variable in size and requires a length discriminator. Formally:

$$\updownarrow x \equiv (|x|, x) \text{ thus } \mathcal{E}(\updownarrow x) \equiv \mathcal{E}(|x|) \frown \mathcal{E}(x) \tag{256}$$

[18] Note that since specific values may belong to both sets which would need a discriminator and those that would not then we are sadly unable to introduce a function capable of serializing corresponding to the *term*'s limitation. A more sophisticated formalism than basic set-theory would be needed, capable of taking into account not simply the value but the term from which or to which it belongs in order to do this succinctly.

We also define a convenient discriminator operator $¿x$ specifically for terms defined by some serializable set in union with $\varnothing$ (generally denoted for some set S as $S?$):

$$¿x \equiv \begin{cases} 0 & \text{if } x = \varnothing \\ (1, x) & \text{otherwise} \end{cases} \tag{257}$$

C.1.5. *Bit Sequence Encoding.* A sequence of bits $b \in \mathbb{B}$ is a special case since encoding each individual bit as an octet would be very wasteful. We instead pack the bits into octets in order of least significant to most, and arrange into an octet stream. In the case of a variable length sequence, then the length is prefixed as in the general case.

$$\mathcal{E}(b \in \mathbb{B}) \equiv \begin{cases} [] & \text{if } b = [] \\ \left[\sum_{i=0}^{\min(8,|b|)} b_i \cdot 2^i \right] \frown \mathcal{E}(b_{8\ldots}) & \text{otherwise} \end{cases} \tag{258}$$

C.1.6. *Dictionary Encoding.* Dictionaries whose key and value domains are encodable and whose key domains are ordered are themselves encodable in one of two manners. The first is as a sequence of encoded key/value pairs ordered by the key. This is generally used when the expected number of entries is somewhat less than the number of possible keys (i.e. the size of the active domain is somewhat less than the domain).

$$\forall K, V : \mathcal{E}(d \in \mathbb{D}\langle K \to V \rangle) \equiv \mathcal{E}(\updownarrow[k \backslash (\mathcal{E}(k), \mathcal{E}(d[k])) \mid k \in \mathcal{K}(d)]) \tag{259}$$

The second is as a sequence of encoded values only, with each entry having an implicit key according to an enumeration over its domain. This is typically used when the domain is small in magnitude and typically similar in magnitude to the active domain. In order to account for the possibility of a key not existing in the dictionary (i.e. the domain and the active domain not being equal) we prefix each value with an octet of one to indicate that the key is indeed present in the dictionary and place an octet of zero if the implied key is not.

We provide a function $\mathcal{S}$ to deliver this from a regular dictionary:

$$\forall K, V : \mathcal{S}(d \in \mathbb{D}\langle K \to V\rangle) \equiv \updownarrow\left[k \begin{cases} [1] \frown \mathcal{E}(d[k]) & \text{if } k \in \mathcal{K}(d) \\ [0] & \text{otherwise} \end{cases} \middle| k \in K \right] \tag{260}$$

C.2. **Block Serialization.** A block $\mathbf{B}$ is serialized as a tuple of its elements in regular order, as implied in equations 12, 13 and 35. For the header, we define both the regular serialization and the unsigned serialization $\mathcal{E}_U$ (the latter has no inverse). Formally:

$$\mathcal{E}(\mathbf{B}) = \mathcal{E}\begin{pmatrix} \mathbf{H},\ \updownarrow\mathbf{E}_T,\ \updownarrow[(r, [(v, \mathcal{E}_2(i), s) \mid (v, i, s) \in \mathbf{v}]) \mid (r, \mathbf{v}) \in \mathbf{E}_J], \\ \updownarrow[(s, \updownarrow p) \mid (s, p) \in \mathbf{E}_P],\ \updownarrow\mathbf{E}_A,\ \updownarrow[(c, w, \updownarrow a) \mid (c, w, a) \in \mathbf{E}_G] \end{pmatrix} \tag{261}$$

$$\mathcal{E}(\mathbf{H}) = \mathcal{E}_U(\mathbf{H}) \frown \mathcal{E}(\mathbf{H}_s) \tag{262}$$

$$\mathcal{E}_U(\mathbf{H}) = \mathcal{E}(\mathbf{H}_p, \mathbf{H}_r, \mathbf{H}_x) \frown \mathcal{E}_4(\mathbf{H}_t) \frown \mathcal{E}(¿\mathbf{H}_e, ¿\mathbf{H}_w, \mathcal{E}_4(\mathbf{H}_k), \mathbf{H}_v) \tag{263}$$

$$\mathcal{E}(x \in \mathbb{X}) \equiv \mathcal{E}(x_a, x_s, x_b, x_l) \frown \mathcal{E}_4(x_t) \frown \mathcal{E}(¿x_p) \tag{264}$$

$$\mathcal{E}(x \in \mathbb{S}) \equiv \mathcal{E}(x_h) \frown \mathcal{E}_4(x_l) \frown \mathcal{E}(x_u) \tag{265}$$

$$\mathcal{E}(x \in \mathbb{L}) \equiv \mathcal{E}_4(x_s) \frown \mathcal{E}(x_c, x_l) \frown \mathcal{E}_8(x_g) \frown \mathcal{E}(O(x_o)) \tag{266}$$

$$\mathcal{E}(x \in \mathbb{W}) \equiv \mathcal{E}(x_a, \updownarrow x_o, x_x, x_s, \updownarrow x_r) \tag{267}$$

$$\mathcal{E}(x \in \mathbb{P}) \equiv \mathcal{E}(\updownarrow x_{\mathbf{j}}, \mathcal{E}_4(x_h), x_c, \updownarrow x_{\mathbf{p}}, x_{\mathbf{x}}, \updownarrow x_{\mathbf{i}}) \tag{268}$$

$$\mathcal{E}(x \in \mathbb{C}) \equiv \mathcal{E}(x_{\mathbf{y}}, x_r) \tag{269}$$

(270)

$$O(o \in \mathbb{J} \cup \mathbb{Y}) \equiv \begin{cases} (0, \updownarrow o) & \text{if } o \in \mathbb{Y} \\ 1 & \text{if } o = \infty \\ 2 & \text{if } o = \lightning \\ 3 & \text{if } o = \texttt{BAD} \\ 4 & \text{if } o = \texttt{BIG} \end{cases}$$

Note the use of O above to succinctly encode the result of a work item and the slight transformations of $\mathbf{E}_G$ and $\mathbf{E}_P$ to take account of the fact their inner tuples contain variable-length sequence terms a and p which need length discriminators.

Appendix D. State Serialization and Merklization

The Merklization process defines a cryptographic commitment from which arbitrary information within state may be provided as being authentic in a concise and swift fashion. We describe this in two stages; the first defines a mapping from 32-octet sequences to (unlimited) octet sequences in a process called *state serialization*. The second forms a 32-octet commitment from this mapping in a process called *Merklization*.

D.1. **State Serialization.** The serialization of state primarily involves placing all the various components of σ into a single mapping from 32-octet sequence *state-keys* to octet sequences of indefinite length. The state-key is constructed from a hash component and a chapter component, equivalent to either the index of a state component or, in the case of the inner dictionaries of δ, a service index.

We define the state-key constructor functions C as:

(271)
$$C: \begin{cases} \mathbb{N}_{2^8} \cup (\mathbb{N}_{2^8}, \mathbb{N}_S) \cup (\mathbb{N}_S, \mathbb{Y}) \to \mathbb{H} \\ \qquad i \in \mathbb{N}_{2^8} \mapsto [i, 0, 0, \dots] \\ \qquad (i, s \in \mathbb{N}_S) \mapsto [i, n_0, n_1, n_2, n_3, 0, 0, \dots] \text{ where } n = \mathcal{E}_4(s) \\ \qquad (s, h) \mapsto [n_0, h_0, n_1, h_1, n_2, h_2, n_3, h_3, h_4, h_5, \dots, h_{27}] \text{ where } n = \mathcal{E}_4(s) \end{cases}$$

The state serialization is then defined as the dictionary built from the amalgamation of each of the components. Cryptographic hashing ensures that there will be no duplicate state-keys given that there are no duplicate inputs to C. Formally,

we define T which transforms some state σ into its serialized form:

(272)

$$T(\sigma) \equiv \begin{cases} & C(1) \mapsto \mathcal{E}([\updownarrow x \mid x \lessdot \alpha]) \ , \\ & C(2) \mapsto \mathcal{E}(\varphi) \ , \\ & C(3) \mapsto \mathcal{E}(\updownarrow[(h, \mathcal{E}_M(\mathbf{b}), s, \updownarrow\mathbf{p}) \mid (h, \mathbf{b}, \\ & C(4) \mapsto \mathcal{E}\left(\left(\gamma_{\mathbf{k}}, \gamma_z, \begin{cases} 0 \text{ if } \gamma_{\mathbf{s}} \in [\![\mathbb{C}]\!]_{\mathsf{E}} \\ 1 \text{ if } \gamma_{\mathbf{s}} \in [\![\mathbb{H}_B]\!]_{\mathsf{E}} \end{cases}\right.\right. \\ & C(5) \mapsto \mathcal{E}(\updownarrow[x \setminus x \in \psi_{\mathbf{a}}], \updownarrow[x \setminus x \in \psi_{\mathbf{b}}], \\ & C(6) \mapsto \mathcal{E}(\eta) \ , \\ & C(7) \mapsto \mathcal{E}(\iota) \ , \\ & C(8) \mapsto \mathcal{E}(\kappa) \ , \\ & C(9) \mapsto \mathcal{E}(\lambda) \ , \\ & C(10) \mapsto \mathcal{E}([¿(w, \updownarrow\mathbf{g}, \mathcal{E}_4(t)) \mid (w, t, \mathbf{g}) \\ & C(11) \mapsto \mathcal{E}_4(\tau) \ , \\ & C(12) \mapsto \mathcal{E}_4(\chi) \ , \\ \forall (s \mapsto \mathbf{a}) \in \delta : & C(255, s) \mapsto \mathbf{a}_c \frown \mathcal{E}_8(\mathbf{a}_b, \mathbf{a}_g, \mathbf{a}_m, \mathbf{a}_l) \frown \mathcal{E}_4(\mathbf{a}_i \\ \forall (s \mapsto \mathbf{a}) \in \delta, (h \mapsto \mathbf{v}) \in \mathbf{a}_{\mathbf{s}} : & C(s, h) \mapsto \mathbf{v} \ , \\ \forall (s \mapsto \mathbf{a}) \in \delta, (h \mapsto \mathbf{p}) \in \mathbf{a}_{\mathbf{p}} : & C(s, h) \mapsto \mathbf{p} \ , \\ \forall (s \mapsto \mathbf{a}) \in \delta, ((h, l) \mapsto \mathbf{t}) \in \mathbf{a}_{\mathbf{l}} : & C(s, \mathcal{E}_4(l) \frown (\neg h_{4:})) \mapsto \mathcal{E}(\updownarrow[\mathcal{E}_4(x) \mid x \lessdot \mathbf{t}]) \end{cases}$$

Note that most rows describe a single mapping between an integer-derived key and the serialization of a state component. However, the final four rows each define sets of mappings since these items act over all service accounts and in the case of the final three rows, the keys of a nested dictionary with the service.

Also note that all non-discriminator integer serialization in state is done in fixed-length according to the size of the term.

D.2. **Merklization.** With T defined, we now define the rest of $\mathcal{M}_S$ which primarily involves transforming the serialized mapping into a cryptographic commitment. We define this commitment as the root of the binary Patricia Merkle Trie with a format optimized for modern compute hardware, primarily by optimizing sizes to fit succinctly into typical memory layouts and reducing the need for unpredictable branching.

D.2.1. *Node Encoding and Trie Identification.* We identify (sub-)tries as the hash of their root node, with one exception: empty (sub-)tries are identified as the zero-hash, $\mathbb{H}^0$.

Nodes are fixed in size at 486 bit (60 bytes and six bits) in order to fit 63 into a 4,096-byte memory page, typical for modern hardware. (We actually make the nodes a round 488 bits, or 61 bytes, but we fix the first two bits as zero, so they need not be stored explicitly on disk.) Each node is either a branch, a leaf or an embedded-value leaf. The first two bits discriminate between these.

In the case of an embedded-value leaf, then the value is stored directly in the node and 5 bits following the first may be used to determine the length of the embedded data. Of the remaining 480 bits, the first 224 bits are dedicated to the last 224 bits of the key and 256 are defined as the value, filling with zeroes if its length is less than 32 bytes.

In the case of a branch, the remaining 484 bits are split between the two child node hashes, using the first 242 bits of each of the sub-trie identities.

In the case of a regular leaf, the remaining 484 bits are dedicated to the last 228 bits of the key and the 256 bits of the hash of the value.

Formally, we define the encoding functions B and L:

(273)

$$B: \begin{cases} (\mathbb{H}, \mathbb{H}) \to \mathbb{B}_{488} \\ \quad (l, r) \mapsto [0,0,0,0] \frown l_{13:16} \frown r_{13:16} \frown l_{16:} \frown r_{16:} \end{cases}$$

(274)

$$L: \begin{cases} (\mathbb{H}, \mathbb{Y}) \to \mathbb{B}_{488} \\ \quad (k, v) \mapsto \begin{cases} [0,0,1] \frown \text{bits}(\mathcal{E}_1(|v|-1))_{:5} \frown \text{bits}(k)_{32:} \frown \text{bits}(v) \frown [0,0,\dots] & \text{if } 0 < |v| \le 32 \\ [0,0,0,1] \frown \text{bits}(k)_{28:} \frown \text{bits}(\mathcal{H}(v)) & \text{otherwise} \end{cases} \end{cases}$$

We may then define, the Merklization function $\mathcal{M}_S$ is as:

(275)

$$\mathcal{M}_S(\sigma) \equiv M(\{(\text{bits}(k) \mapsto (k, v)) \mid (k \mapsto v) \in T(\sigma)\})$$

(276)

$$M(d : \mathbb{D}\langle\mathbb{B} \to (\mathbb{H}, \mathbb{Y})\rangle) \equiv \begin{cases} \mathbb{H}^0 & \text{if } |d| = 0 \\ \mathcal{H}(\text{bits}^{-1}(L(k, v))) & \text{if } \mathcal{V}(d) = \{(k, v)\} \\ \mathcal{H}(\text{bits}^{-1}(B(l, r))) & \text{otherwise, where } \forall b, p : (b \mapsto p) \in d \Leftrightarrow (b_{1:} \mapsto \end{cases}$$

Appendix E. Shuffling

The Fisher-Yates shuffle function is defined formally as:

(277)

$$\forall T, l \in \mathbb{N} : \mathcal{F}: \begin{cases} ([\![T]\!]_l, [\![\mathbb{N}]\!]_{l:}) \to [\![T]\!]_l \\ \qquad (\mathbf{s}, \mathbf{r}) \mapsto \begin{cases} [\mathbf{s}_{\mathbf{r}_0 \bmod l}] \frown \mathcal{F}([\mathbf{s}_i \mid i \in \mathbb{N}_l \setminus \{\mathbf{r}_0 \bmod l\}], [\mathbf{r}_1, \mathbf{r}_2, \dots]) & \text{if } \mathbf{s} \neq [\\ [] & \text{otherw} \end{cases} \end{cases}$$

Since it is often useful to shuffle a sequence based on some random seed in the form of a hash, we provide a secondary form of the shuffle function $\mathcal{F}$ which accepts a 32-byte hash instead of the numeric sequence. We define $\mathcal{Q}$, the numeric-sequence-from-hash function, thus:

(278)
$$\forall l \in \mathbb{N} : \ \mathcal{Q}_l: \begin{cases} \mathbb{H} \to [\![\mathbb{N}_l]\!] \\ h \mapsto [\mathcal{E}_4^{-1}(\mathcal{H}(h \frown \mathcal{E}_4(\lfloor i/8 \rfloor))_{4i \bmod 32\dots}) \mid i \in \mathbb{N}_l] \end{cases}$$

(279)
$$\forall T, l \in \mathbb{N} : \ \mathcal{F}: \begin{cases} ([\![T]\!]_l, \mathbb{H}) \to [\![T]\!]_l \\ \quad (\mathbf{s}, h) \mapsto \mathcal{F}(\mathbf{s}, \mathcal{Q}_l(h)) \end{cases}$$

Appendix F. General Merklization

F.1. **Binary Merkle Tree.** The Merkle tree is a cryptographic data structure yielding a hash commitment to a specific sequence of values. It provides $O(N)$ computation and $O(logN)$ proof size for inclusion. This *well-balanced* formulation ensures that the maximum depth of any leaf is minimal and that the number of leaves at that depth is also minimal.

We define the well-balanced binary Merkle function as $\mathcal{M}_2$:

$$\mathcal{M}_2: \begin{cases} ([[\mathbb{Y}_{n>32}]], \mathbb{Y} \to \mathbb{H}) \to \mathbb{H} \\ \qquad (\mathbf{v}, H) \mapsto \begin{cases} \mathbb{H}_0 & \text{if } |\mathbf{v}| = 0 \\ H(\mathbf{v}_0) & \text{if } |\mathbf{v}| = 1 \\ N(\mathbf{v}, H) & \text{otherwise} \end{cases} \end{cases} \tag{280}$$

$$\text{where } N: \begin{cases} ([[\mathbb{Y}_{n>32}]]_{1:}, \mathbb{Y} \to \mathbb{H}) \to \mathbb{Y}_{n>32} \cup \mathbb{H} \\ \qquad (\mathbf{v}, H) \mapsto \begin{cases} \mathbf{v}_0 & \text{if } |\mathbf{v}| = 1 \\ H(N(\mathbf{v}_{...\lceil |\mathbf{v}|/2 \rceil}) \frown N(\mathbf{v}_{\lceil |\mathbf{v}|/2 \rceil ...})) & \text{otherwise} \end{cases} \end{cases}$$

F.2. **Merkle Mountain Ranges.** The Merkle mountain range (MMR) is an append-only cryptographic data structure which yields a commitment to a sequence of values. Appending to an MMR and proof of inclusion of some item within it are both $O(logN)$ in time and space for the size of the set.

We define a Merkle mountain range as being within the set $[[\mathbb{H}?]]$, a sequence of peaks, each peak the root of a Merkle tree containing 2^i items where i is the index in the sequence. Since we support set sizes which are not always powers-of-two-minus-one, some peaks may be empty, $\varnothing$ rather than a Merkle root.

Since the sequence of hashes is somewhat unwieldy as a commitment, Merkle mountain ranges are themselves generally hashed before being published. Hashing them removes the possibility of further appending so the range itself is kept on the system which needs to generate future proofs.

We define the append function $\mathcal{A}$ as:

$$\mathcal{A}: \begin{cases} ([[\mathbb{H}?]], \mathbb{H}, \mathbb{Y} \to \mathbb{H}) \to [[\mathbb{H}?]] \\ \qquad (\mathbf{r}, l, H) \mapsto P(\mathbf{r}, l, 0, H) \end{cases} \tag{281}$$

$$\text{where } P: \begin{cases} ([[\mathbb{H}?]], \mathbb{H}, \mathbb{N}, \mathbb{Y} \to \mathbb{H}) \to [[\mathbb{H}?]] \\ \qquad (\mathbf{r}, l, n, H) \mapsto \begin{cases} \mathbf{r} \nplus l & \text{if } n \geq |\mathbf{r}| \\ R(\mathbf{r}, n, l) & \text{if } n < |\mathbf{r}| \wedge \mathbf{r}_n = \varnothing \\ P(R(\mathbf{r}, n, \varnothing), H(\mathbf{r}_n \frown l), n+1, H) & \text{otherwise} \end{cases} \end{cases}$$

$$\text{and } R: \begin{cases} ([[T]], \mathbb{N}, T) \to [[T]] \\ \qquad (\mathbf{s}, i, v) \mapsto \mathbf{s}' \text{ where } \mathbf{s}' = \mathbf{s} \text{ except } \mathbf{s}'_i = v \end{cases}$$

We define the MMR encoding function as $\mathcal{E}_M$:

$$\mathcal{E}_M: \begin{cases} [\![\mathbb{H}?]\!] \to \mathbb{H} \\ \quad \mathbf{b} \mapsto \mathcal{E}(\updownarrow[¿x \mid x \lessdot \mathbf{b}]) \end{cases} \tag{282}$$

Appendix G. Bandersnatch Ring VRF

The Bandersnatch curve is defined by Masson, Sanso, and Zhang 2021.

The singly-contextualized Bandersnatch Schnorr-like signatures $\mathbb{F}^m_k\langle c\rangle$ are defined as a formulation under the *ietf* VRF template specified by Hosseini and Galassi 2024 (as IETF VRF) and further detailed by Goldberg et al. 2023.

(283)
$$\mathbb{F}^{m\in\mathbb{Y}}_{k\in\mathbb{H}_B}\langle c\in\mathbb{H}\rangle \subset \mathbb{Y}_{96} \equiv \{x \mid x\in Y_{96}, \text{verify}(k,c,m,\text{decode}(x_{:32}),\text{decode}(x_{32:})) = \top\}$$

(284)
$$\mathcal{Y}(s\in\mathbb{F}^m_k\langle c\rangle)\in\mathbb{H} \equiv \text{hashed\_output}(\text{decode}(x_{:32}) \mid x\in\mathbb{F}^m_k\langle c\rangle)$$

The singly-contextualized Bandersnatch RingVRF proofs $\bar{\mathbb{F}}^m_r\langle c\rangle$ are a zk-SNARK-enabled analogue utilizing the Pedersen VRF, also defined by Hosseini and Galassi 2024 and further detailed by Jeffrey Burdges et al. 2023.

(285) $\mathcal{R}([\![\mathbb{H}_B]\!])\in\mathbb{Y}_R \equiv ...$

(286)
$$\bar{\mathbb{F}}^{m\in\mathbb{Y}}_{r\in\mathbb{Y}_R}\langle c\in\mathbb{H}\rangle \subset \mathbb{Y}_{784} \equiv \{x \mid x\in\mathbb{Y}_{784}, \text{verify}(r,c,m,\text{decode}(x_{:32}),\text{decode}(x_{32:})) = \top\}$$

(287)
$$\mathcal{Y}(p\in\bar{\mathbb{F}}^m_r\langle c\rangle)\in\mathbb{H} \equiv \text{hashed\_output}(\text{decode}(x_{:32}) \mid x\in\bar{\mathbb{F}}^m_r\langle c\rangle)$$

Appendix H. Erasure Coding

We assume a piece-encode function $E_{piece} : \mathbb{Y}_{2048} \to [\![\mathbb{Y}_6]\!]_{1024}$ and decode function $E^{-1}_{piece} : [\![\mathbb{Y}_6]\!]_{342} \to \mathbb{Y}_{2048}$. This is a Reed-Solomon erasure codec with a rate of 1:3 and basis as defined by Lin, Chung, and Han 2014. For our 1,023 validators, we use a message size of $n = 1024$ to take advantage of a suitable FFT for fast encoding (i.e. 2^{10}).

We assume some data blob $\mathbf{d} \in \mathbb{Y}_{2048 \cdot k}, k \in \mathbb{N}$. We are able to express this as a whole number of k pieces each of a sequence of 2,048 octets. We denote these pieces $\mathbf{p} \in [\![\mathbb{Y}_{2048}]\!] = \text{split}_{2048}(\mathbf{p})$. Each piece is then split into 1,024 chunks each a six octet sequence. The resulting matrix of chunks is grouped by its index in its piece and concatenated to form 1,024 super-chunks, made up of many six octet subsequences one from each piece. Any 342 of these super-chunks may then be used to reconstruct the original data $\mathbf{d}$.

Formally we begin by defining two utility functions for splitting some large sequence into a number of equal-sized sub-sequences and for joining subsequences back into a single large sequence:

$$\forall n, k \in \mathbb{N} : \text{split}_n(\mathbf{d} \in \mathbb{Y}_{k \cdot n}) \in [\![\mathbb{Y}_n]\!]_k \equiv \left[\mathbf{d}_{0 \cdots + n}, \mathbf{d}_{n \cdots + n}, \cdots, \mathbf{d}_{(k-1)n \cdots + n}\right] \tag{288}$$

$$\forall n, k \in \mathbb{N} : \text{join}(\mathbf{c} \in [\![\mathbb{Y}_n]\!]_k) \in \mathbb{Y}_{k \cdot n} \equiv \mathbf{c}_0 \frown \mathbf{c}_1 \frown \ldots \tag{289}$$

We define ${}^{\mathrm{T}}\mathbf{x}$ as the transposition of the sequence-of-sequences $\mathbf{x}$:

$${}^{\mathrm{T}}[[\mathbf{x}_{0,0}, \mathbf{x}_{0,1}, \mathbf{x}_{0,2}, \ldots], [\mathbf{x}_{1,0}, \mathbf{x}_{1,1}, \ldots], \ldots] \equiv [[\mathbf{x}_{0,0}, \mathbf{x}_{1,0}, \mathbf{x}_{2,0}, \ldots], [\mathbf{x}_{0,1}, \mathbf{x}_{1,1}, \ldots], \ldots] \tag{290}$$

We may then define our erasure-code chunking function which accepts an arbitrary sized data blob whose length divides wholly into 2,048 octets and results in 1,024 sequences of sequences each of smaller blobs:

$$\forall k \in \mathbb{N} : \mathcal{C} : \begin{cases} \mathbb{Y}_{2048 \cdot k} \to [\![\mathbb{Y}_{6k}]\!]_{1024} \\ \quad \mathbf{d} \mapsto [\text{join}(\mathbf{c}) \mid \mathbf{c} \vartriangleleft {}^{\mathrm{T}}[E_{piece}(\mathbf{p}) \mid \mathbf{p} \vartriangleleft \text{split}_{2048}(\mathbf{d})]] \end{cases} \tag{291}$$

It may be inverted with only one-third-plus-one of the items of its result (i.e. 342) needed to rebuild the original blob:

$$\forall k \in \mathbb{N} : \mathcal{C}^{-1} : \begin{cases} [\![\mathbb{Y}_{6k}]\!]_{342} \to \mathbb{Y}_{2048 \cdot k} \\ \quad \mathbf{c} \mapsto \text{join}([E^{-1}_{piece}(x) \mid \mathbf{x} \vartriangleleft {}^{\mathrm{T}}[\text{split}_6(\mathbf{x}) \mid \mathbf{x} \vartriangleleft \mathbf{c}]]) \end{cases} \tag{292}$$

Appendix I. Index of Notation

I.1. **Sets.**

I.1.1. *Regular Notation.*

$\mathbb{N}$**:** The set of positive integers including zero. Subscript denotes one greater than the maximum. See section 3.4.

$\mathbb{N}^+$**:** The set of positive integers not including zero.

$\mathbb{N}_B$**:** The set of balance values. Equivalent to $\mathbb{N}_{2^{64}}$. See equation 29.

$\mathbb{N}_G$**:** The set of unsigned gas values. Equivalent to $\mathbb{N}_{2^{64}}$. See equation 31.

$\mathbb{N}_L$**:** The set of blob length values. Equivalent to $\mathbb{N}_{2^{32}}$. See section 3.4.

$\mathbb{N}_S$**:** The index of a service. Equivalent to $\mathbb{N}_{2^{32}}$. See section 84.

$\mathbb{N}_T$**:** The set of timeslot values. Equivalent to $\mathbb{N}_{2^{32}}$. See equation 34.

$\mathbb{Q}$**:** The set of rational numbers. Unused.

$\mathbb{R}$**:** The set of real numbers. Unused.

$\mathbb{Z}$**:** The set of integers. Subscript denotes range. See section 3.4.

$\mathbb{Z}_G$**:** The set of signed gas values. Equivalent to $\mathbb{Z}_{-2^{63}:2^{63}}$. See equation 31.

I.1.2. *Custom Notation.*

$\mathbb{A}$**:** The set of service accounts. See equation 86.

$\mathbb{B}$**:** The set of Boolean sequences/bitstrings. Subscript denotes length. See section 3.7.

$\mathbb{C}$**:** The set of seal-key tickets. See equation 48. *Not used as the set of complex numbers.*

$\mathbb{D}$**:** The set of dictionaries. See section 3.5.

$\mathbb{D}\langle K \to V\rangle$**:** The set of dictionaries making a partial bijection of domain K to range V. See section 3.5.

$\mathbb{E}$**:** The set of valid Ed25519 signatures. A subset of $\mathbb{Y}_{64}$. See section 3.8.

$\mathbb{E}_K\langle M\rangle$**:** The set of valid Ed25519 signatures of the key K and message M. A subset of $\mathbb{E}$. See section 3.8.

$\mathbb{F}$**:** The set of Bandersnatch signatures. A subset of $\mathbb{Y}_{64}$. See section 3.8. *NOTE: Not used as finite fields.*

$\mathbb{F}_K^M\langle C\rangle$**:** The set of Bandersnatch signatures of the public key K, context C and message M. A subset of $\mathbb{F}$. See section 3.8.

$\bar{\mathbb{F}}$**:** The set of Bandersnatch RingVRF proofs. See section 3.8.

$\bar{\mathbb{F}}_R^M\langle C\rangle$**:** The set of Bandersnatch RingVRF proofs of the root R, context C and message M. A subset of $\bar{\mathbb{F}}$. See section 3.8.

$\mathbb{G}$**:** Unused.

$\mathbb{H}$**:** The set of 32-octet cryptographic values. A subset of $\mathbb{Y}_{32}$. $\mathbb{H}$ without a subscript generally implies a hash function result. See section 3.8. *NOTE: Not used as quaternions.*

$\mathbb{H}_B$**:** The set of Bandersnatch public keys. A subset of $\mathbb{Y}_{32}$. See section 3.8 and appendix G.

$\mathbb{H}_E$**:** The set of Ed25519 public keys. A subset of $\mathbb{Y}_{32}$. See section 3.8.2.

$\mathbb{Y}_R$**:** The set of Bandersnatch ring roots. A subset of $\mathbb{Y}_{32}$. See section 3.8 and appendix G.

$\mathbb{I}$**:** The set of work items. See equation 165.

$\mathbb{J}$**:** The set of work execution errors.

$\mathbb{K}$**:** The set of validator key-sets. See equation 49.

$\mathbb{L}$**:** The set of work results.

$\mathbb{M}$**:** The set of PVM RAM states. A superset of $\mathbb{Y}_{2^{32}}$. See appendix A.

$\mathbb{O}$**:** The accumulation operand element, corresponding to a single work result.

$\mathbb{P}$**:** The set of work-packages. See equation 162.

$\mathbb{S}$**:** The set of work-package specifications.

$\mathbb{T}$**:** The set of deferred transfers.

$\mathbb{U}$**:** Unused.

$\mathbb{V}_\mu$**:** The set of validly readable indices for PVM RAM μ. See appendix A.

$\mathbb{V}^*_\mu$**:** The set of validly writable indices for PVM RAM μ. See appendix A.

$\mathbb{W}$**:** The set of work-reports.

$\mathbb{X}$**:** The set of refinement contexts.

$\mathbb{Y}$**:** The set of octet strings/"blobs". Subscript denotes length. See section 3.7.

- $\mathbb{Y}_{BLS}$**:** The set of BLS public keys. A subset of $\mathbb{Y}_{144}$. See section 3.8.2.
- $\mathbb{Y}_R$**:** The set of Bandersnatch ring roots. A subset of $\mathbb{Y}_{196608}$. See section 3.8 and appendix G.

I.2. **Functions.**

Λ**:** The historical lookup function. See equation 90.

Ξ**:** The work result computation function. See equation 167.

Υ**:** The general state transition function. See equations 11, 15.

Ψ**:** The whole-program PVM machine state-transition function. See equation A.

- Ψ_1**:** The single-step (PVM) machine state-transition function. See appendix A.
- Ψ_A**:** The Accumulate PVM invocation function. See appendix B.
- Ψ_H**:** The host-function invocation (PVM) with host-function marshalling. See appendix A.
- Ψ_I**:** The Is-Authorized PVM invocation function. See appendix B.
- Ψ_M**:** The marshalling whole-program PVM machine state-transition function. See appendix A.

Ψ_R: The Refine PVM invocation function. See appendix B.

Ψ_T: The On-Transfer PVM invocation function. See appendix B.

Ω: Virtual machine host-call functions. See appendix B.

Ω_A: Assign-core host-call.

Ω_C: Checkpoint host-call.

Ω_D: Designate-validators host-call.

Ω_E: Empower-service host-call.

Ω_F: Forget-preimage host-call.

Ω_G: Gas-remaining host-call.

Ω_H: Historical-lookup-preimage host-call.

Ω_I: Information-on-service host-call.

Ω_K: Kickoff-PVM host-call.

Ω_L: Lookup-preimage host-call.

Ω_M: Make-PVM host-call.

Ω_N: New-service host-call.

Ω_O: Poke-PVM host-call.

Ω_P: Peek-PVM host-call.

Ω_Q: Quit-service host-call.

Ω_S: Solicit-preimage host-call.

Ω_R: Read-storage host-call.

Ω_T: Transfer host-call.

Ω_U: Upgrade-service host-call.

Ω_W: Write-storage host-call.

Ω_X: Expunge-PVM host-call.

I.3. **Utilities, Externalities and Standard Functions.**

$\mathcal{A}(\dots)$**:** The Merkle mountain range append function. See equation 281.

$\mathcal{B}_n(\dots)$**:** The octets-to-bits function for n octets. Superscripted $^{-1}$ to denote the inverse. See equation 206.

$\mathcal{C}_n(\dots)$**:** The erasure coding function for n chunks. Superscripted $^{-1}$ to denote the inverse. See equation 291.

$\mathcal{E}(\dots)$**:** The SCALE encode function. Superscripted $^{-1}$ to denote the inverse. See appendix C.

$\mathcal{F}(\dots)$**:** The Fisher-Yates shuffle function. See equation 277.

$\mathcal{H}(\dots)$**:** The Blake 2b 256-bit hash function. See section 3.8.

$\mathcal{H}_K(\dots)$**:** The Keccak 256-bit hash function. See section 3.8.

$\mathcal{K}(\dots)$**:** The domain, or set of keys, of a dictionary. See section 3.5.

$\mathcal{M}_2(\dots)$**:** The binary Merklization function. See appendix F.

$\mathcal{M}_S(\dots)$**:** The state Merklization function. See appendix D.

$\mathcal{N}(\dots)$**:** The erasure-coding chunks function. See appendix H.

$\mathcal{P}_n(\dots)$**:** The octet-array zero-padding function. See equation 168.

$\mathcal{Q}(\dots)$**:** The numeric-sequence-from-hash function. See equation 279.

$\mathcal{R}(\dots)$**:** The Bandersnatch ring root function. See section 3.8 and appendix G.

$\mathcal{S}_k(\dots)$**:** The general signature function. See section 3.8.

$\mathcal{T}$**:** The current time expressed in seconds after the start of the JAM Common Era. See section 4.4.

$\mathcal{U}(\dots)$**:** The substitute-if-nothing function. See equation 2.

$\mathcal{V}(\dots)$**:** The range, or set of values, of a dictionary or sequence. See section 3.5.

$\mathcal{Y}(\dots)$**:** The alias/output/entropy function of a Bandersnatch VRF signature/proof. See section 3.8 and appendix G.

$\mathcal{Z}_n(\dots)$**:** The into-signed function for a value in $\mathbb{N}_{2^{8n}}$. Superscripted with $^{-1}$ to denote the inverse. See equation 204.

$\wp(\dots)$**:** Power set function.

I.4. **Values.**

I.4.1. *Block-context Terms.* These terms are all contextualized to a single block. They may be superscripted with some other term to alter the context and reference some other block.

A: The ancestor set of the block. See equation 37.

B: The block. See equation 12.

C: The service accumulation-commitment, used to form the BEEFY root. See equation 156.

E: The block extrinsic. See equation 13.

$\mathbf{F}_v$**:** The BEEFY signed commitment of validator v. See equation 190.

H: The block header. See equation 35.

G: The mapping from cores to guarantor keys. See section 11.3.

$\mathbf{G}^*$**:** The mapping from cores to guarantor keys for the previous rotation. See section 11.3.

Q: The selection of ready work-reports which a validator determined they must audit. See equation 171.

R: The set of work-reports which have now become available and ready for accumulation. See equation 121.

S: The set of indices of services which have been accumulated ("progressed") in the block. See equation 150.

T: The ticketed condition, true if the block was sealed with a ticket signature rather than a fallback. See equations 56 and 57.

U: The audit condition, equal to $\top$ once the block is audited. See section 13.5.

Without any superscript, the block is assumed to the block being imported or, if no block is being imported, the head of the best chain (see section 15). Explicit block-contextualizing superscripts include:

$\mathbf{B}^{\natural}$**:** The latest finalized block. See equation 15.

$\mathbf{B}^{\flat}$**:** The block at the head of the best chain. See equation 15.

I.4.2. *State components.* Here, the prime annotation indicates posterior state. Individual components may be identified with a letter subscript.

α**:** The core αuthorizations pool. See equation 81.

β**:** Information on the most recent βlocks.

γ**:** State concerning Safrole. See equation 45.

$\gamma_{\mathbf{a}}$**:** The sealing-key contest ticket accumulator.

$\gamma_{\mathbf{k}}$**:** The validator keys for the following epoch.

$\gamma_{\mathbf{s}}$**:** The sealing-key series of the current epoch.

γ_z**:** The current epoch's Bandersnatch root.

δ**:** The (prior) state of the service accounts.

$\delta^{\dagger}$**:** The post-preimage integration, pre-accumulation intermediate state.

$\delta^{\ddagger}$**:** The post-accumulation, pre-transfer intermediate state.

η**:** The eηtropy accumulator and epochal raηdomness.

ι**:** The validator keys and metadata to be drawn from next.

κ**:** The validator κeys and metadata currently active.

λ**:** The validator keys and metadata which were active in the prior epoch.

ρ**:** The ρending reports, per core, which are being made available prior to accumulation.

$\rho^\dagger$**:** The post-judgement, pre-guarantees-extrinsic intermediate state.

$\rho^\ddagger$**:** The post-guarantees-extrinsic, pre-assurances-extrinsic, intermediate state.

σ**:** The σverall state of the system. See equations 11, 14.

τ**:** The most recent block's τimeslot.

φ**:** The authorization queue.

ψ**:** Votes regarding any ongoing disputes.

χ**:** The privileged service indices.

χ_m**:** The index of the empower service.

χ_v**:** The index of the designate service.

χ_a**:** The index of the assign service.

I.4.3. *Virtual Machine components.*

ε**:** The instruction sequence.

ν**:** The immediate values of an instruction.

μ**:** The memory sequence; a member of the set $\mathbb{M}$.

ξ**:** The gas counter.

ω**:** The registers.

ζ**:** The exit-reason resulting from all machine state transitions.

ϖ**:** The sequence of basic blocks of the program.

$\imath$**:** The instruction counter.

I.4.4. *Constants.*

$\mathsf{A} = 8$**:** The period, in seconds, between audit tranches.

$\mathsf{B}_I = 10$**:** The additional minimum balance required per item of elective service state.

$\mathsf{B}_L = 1$**:** The additional minimum balance required per octet of elective service state.

$\mathsf{B}_S = 100$**:** The basic minimum balance which all services require.

$\mathsf{C} = 341$**:** The total number of cores.

$\mathsf{D} = 28,800$**:** The period in timeslots after which an unreferenced preimage may be expunged.

$\mathsf{E} = 600$**:** The length of an epoch in timeslots.

$\mathsf{F} = 2$**:** The audit bias factor, the expected number of additional validators who will audit a work-report in the following tranche for each no-show in the previous.

G_A**:** The total gas allocated to a core for Accumulation.

G_I**:** The gas allocated to invoke a work-package's Is-Authorized logic.

G_R**:** The total gas allocated for a work-package's Refine logic.

$\mathsf{H} = 8$**:** The size of recent history, in blocks.

$\mathsf{I} = 4$**:** The maximum amount of work items in a package.

$\mathsf{K} = 16$**:** The maximum number of tickets which may be submitted in a single extrinsic.

$\mathsf{L} = 14,400$**:** The maximum age in timeslots of the lookup anchor.

$\mathsf{M} = 128$**:** The size of a transfer memo in octets.

$\mathsf{N} = 2$**:** The number of ticket entries per validator.

$\mathsf{O} = 8$**:** The maximum number of items in the authorizations pool.

$\mathsf{P} = 6$**:** The slot period, in seconds.

$\mathsf{Q} = 80$**:** The maximum number of items in the authorizations queue.

$\mathsf{R} = 10$**:** The rotation period of validator-core assignments, in timeslots.

$\mathsf{S} = 4,000,000$**:** The maximum size of service code in octets.

$\mathsf{U} = 5$**:** The period in timeslots after which reported but unavailable work may be replaced.

$\mathsf{V} = 1023$**:** The total number of validators.

$\mathsf{W}_P = 6 \cdot 2^{20} + 2^{16}$**:** The maximum size of an encoded work-package in octets.

$\mathsf{W}_R = 96 \cdot 2^{10}$**:** The maximum size of an encoded work-report in octets.

X**:** Context strings, see below.

$\mathsf{Y} = 500$**:** The number of slots into an epoch at which ticket-submission ends.

$\mathsf{Z}_A = 4$**:** The PVM dynamic address alignment factor. See equation 209.

$\mathsf{Z}_I = 2^{24}$**:** The standard PVM program initialization input data size. See equation A.6.

$\mathsf{Z}_P = 2^{14}$**:** The standard PVM program initialization page size. See section A.6.

$\mathsf{Z}_Q = 2^{16}$**:** The standard PVM program initialization segment size. See section A.6.

I.4.5. *Signing Contexts.*

X_A = **`$jam_available`:** *Ed25519* Availability assurances.

X_B = **`$jam_beefy`:** *BLS* Accumulate-result-root-MMR commitment.

X_F = **`$jam_fallback_seal`:** *Bandersnatch* Fallback block seal.

X_G = **`$jam_guarantee`:** *Ed25519* Guarantee statements.

X_I = **`$jam_announce`:** *Ed25519* Audit announcement statements.

X_S = **`$jam_seal:`** *Bandersnatch* Regular block seal.

X_T = **`$jam_ticket:`** *Bandersnatch RingVRF* Ticket generation.

X_U = **`$jam_audit:`** *Bandersnatch* Audit selection entropy.

$\mathsf{X}_\top$ = **`$jam_valid:`** *Ed25519* Judgements for valid work-reports.

$\mathsf{X}_\bot$ = **`$jam_invalid:`** *Ed25519* Judgements for invalid work-reports.

References

Bertoni, Guido et al. (2013). "Keccak". In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer, pp. 313–314.

Bögli, Roman (2024). "Assessing RISC Zero using ZKit: An Extensible Testing and Benchmarking Suite for ZKP Frameworks". PhD thesis. OST Ostschweizer Fachhochschule.

Boneh, Dan, Ben Lynn, and Hovav Shacham (2004). "Short Signatures from the Weil Pairing". In: *J. Cryptology* 17, pp. 297–319. DOI: 10.1007/s00145-004-0314-9.

Burdges, Jeff et al. (2022). *Efficient Aggregatable BLS Signatures with Chaum-Pedersen Proofs*. Cryptology ePrint Archive, Paper 2022/1611. https://eprint.iacr.org/2022/1611. URL: https://eprint.iacr.org/2022/1611.

Burdges, Jeffrey et al. (2023). *Ring Verifiable Random Functions and Zero-Knowledge Continuations*. Cryptology ePrint Archive, Paper 2023/002. URL: https://eprint.iacr.org/2023/002.

Buterin, Vitalik (2013). *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. URL: https://github.com/ethereum/wiki/wiki/White-Paper.

Buterin, Vitalik and Virgil Griffith (2019). *Casper the Friendly Finality Gadget*. arXiv: 1710.09437 [cs.CR].

Cosmos Project (2023). *Interchain Security Begins a New Era for Cosmos*. Fetched 18th March, 2024. URL: https://blog.cosmos.network/interchain-security-begins-a-new-era-for-cosmos-a2dc3c0be63.

Dune and hildobby (2024). *Ethereum Staking*. Fetched 18th March, 2024. URL: https://dune.com/hildobby/eth2-staking.

Ethereum Foundation (2024a). "A digital future on a global scale". In: Fetched 4th April, 2024. URL: https://ethereum.org/en/roadmap/vision/.

— (2024b). *Danksharding*. Fetched 18th March, 2024. URL: https://ethereum.org/en/roadmap/danksharding/.

Fisher, Ronald Aylmer and Frank Yates (1938). *Statistical tables for biological, agricultural and medical research*. Oliver and Boyd.

Gabizon, Ariel, Zachary J. Williamson, and Oana Ciobotaru (2019). *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments*

of Knowledge. Cryptology ePrint Archive, Paper 2019/953. URL: https://eprint.iacr.org/2019/953.

Goldberg, Sharon et al. (Aug. 2023). *Verifiable Random Functions (VRFs)*. RFC 9381. DOI: 10.17487/RFC9381. URL: https://www.rfc-editor.org/info/rfc9381.

Hertig, Alyssa (2016). *So, Ethereum's Blockchain is Still Under Attack...* Fetched 18th March, 2024. URL: https://www.coindesk.com/markets/2016/10/06/so-ethereums-blockchain-is-still-under-attack/.

Hopwood, Daira et al. (2020). *BLS12-381*. URL: https://z.cash/technology/jubjub/.

Hosseini, Seyed and Davide Galassi (2024). "Bandersnatch VRF-AD Specification". In: Fetched 4th April, 2024. URL: https://github.com/davxy/bandersnatch-vrfs-spec/blob/main/specification.pdf.

Jha, Prashant (2024). *Solana outage raises questions about client diversity and beta status*. Fetched 18th March, 2024. URL: https://cointelegraph.com/news/solana-outage-client-diversity-beta.

Josefsson, Simon and Ilari Liusvaara (Jan. 2017). *Edwards-Curve Digital Signature Algorithm (EdDSA)*. RFC 8032. DOI: 10.17487/RFC8032. URL: https://www.rfc-editor.org/info/rfc8032.

Kokoris-Kogias, Eleftherios et al. (2017). *OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding*. Cryptology ePrint Archive, Paper 2017/406. https://eprint.iacr.org/2017/406. URL: https://eprint.iacr.org/2017/406.

Kwon, Jae and Ethan Buchman (2019). "Cosmos whitepaper". In: *A Netw. Distrib. Ledgers* 27, pp. 1–32.

Lin, Sian-Jheng, Wei-Ho Chung, and Yunghsiang S. Han (2014). "Novel Polynomial Basis and Its Application to Reed-Solomon Erasure Codes". In: *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pp. 316–325. DOI: 10.1109/FOCS.2014.41.

Masson, Simon, Antonio Sanso, and Zhenfei Zhang (2021). *Bandersnatch: a fast elliptic curve built over the BLS12-381 scalar field*. Cryptology ePrint Archive, Paper 2021/1152. URL: https://eprint.iacr.org/2021/1152.

Ng, Felix (2024). *Is measuring blockchain transactions per second stupid in 2024?* Fetched 18th March, 2024. URL: https://cointelegraph.com/magazine/blockchain-transactions-per-second-tps-stupid-big-questions/.

PolkaVM Project (2024). "PolkaVM/RISC0 Benchmark Results". In: Fetched 3rd April, 2024. URL: https://github.com/koute/risc0-benchmark/blob/master/README.md.

Saarinen, Markku-Juhani O. and Jean-Philippe Aumasson (Nov. 2015). *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)*. RFC 7693. DOI: 10.17487/RFC7693. URL: https://www.rfc-editor.org/info/rfc7693.

Sadana, Apoorv (2024). *Bringing Polkadot tech to Ethereum*. Fetched 18th March, 2024. URL: https://ethresear.ch/t/bringing-polkadot-tech-to-ethereum/17104.

Sharma, Shivam (2023). *Ethereum's Rollups are Centralized*. URL: https://public.bnbstatic.com/static/files/research/ethereums-rollups-are-centralized-a-look-into-decentralized-sequencers.pdf.

Solana Foundation (2023). *Solana data goes live on Google Cloud BigQuery*. Fetched 18th March, 2024. URL: https://solana.com/news/solana-data-live-on-google-cloud-bigquery.

Solana Labs (2024). *Solana Validator Requirements*. Fetched 18th March, 2024. URL: https://docs.solanalabs.com/operations/requirements.

Stewart, Alistair (2018). "Efficient Block-Auditing for Blockchains". In.

Stewart, Alistair and Eleftherios Kokoris-Kogia (2020). "Grandpa: a byzantine finality gadget". In: *arXiv preprint arXiv:2007.01560*.

Tanana, Dmitry (2019). "Avalanche blockchain protocol for distributed computing security". In: *2019 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)*. IEEE, pp. 1–3.

Thaler, Justin (2023). "A technical FAQ on Lasso, Jolt, and recent advancements in SNARK design". In: Fetched 3rd April, 2024. URL: https://a16zcrypto.com/posts/article/a-technical-faq-on-lasso-jolt-and-recent-advancements-in-snark-design/.

Wikipedia (2024). *Fisher-Yates shuffle: The modern algorithm*. URL: https://en.wikipedia.org/wiki/Fisher%5C%E2%5C%80%5C%93Yates_shuffle%5C#The_modern_algorithm.

Wood, Gavin (2014). “Ethereum: A secure decentralised generalised transaction ledger”. In: *Ethereum project yellow paper* 151, pp. 1–32.

Yakovenko, Anatoly (2018). “Solana: A new architecture for a high performance blockchain v0. 8.13”. In.