JOIN-ACCUMULATE MACHINE: A MOSTLY-COHERENT TRUSTLESS SUPERCOMPUTER

DRAFT 0.7.1$^{\Delta}$ - November 1, 2025

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 will transition itself in a major revision on the basis of approval by its governance apparatus.

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 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.

[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.

(3) Performance: able to perform computation quickly and at low cost.
(4) Coherency: the causal relationship possible between different elements of state and 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-coherency antagonism*.

1.3. **Scaling under Size-Coherency Antagonism.** Size-coherency antagonism is a simple principle implying that as the state-space of information systems grow, then the system necessarily becomes less coherent. It is a direct implication of principle that causality is limited by speed. The maximum speed allowed by physics is C the speed of light in a vacuum, however other information systems may have lower bounds: In biological system this is largely determined by various chemical processes whereas in electronic systems is it determined by the speed of electrons in various substances. Distributed software systems will tend to have much lower bounds still, being dependent on a substrate of software, hardware and packet-switched networks of varying reliability.

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 time for causal resolution (i.e. all correct implications of an event to be felt) becomes divergent across the system, causing incoherence.

Setting the question of overall security aside for a moment, we can manage incoherence by fragmenting the system into causally-independent subsystems, each of which is small enough to be coherent. In a resource-rich environment, a bacterium may split into two rather than growing to double its size. This pattern is rather a crude means of dealing with incoherency under growth: intra-system processing has low size and total coherence, inter-system processing supports higher overall sizes but without coherence. 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). Such systems typically rely on asynchronous and simplistic communication with "settlement areas" which provide a small-scoped coherent state-space to manage specific interactions such as a token transfer.

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, *mostly coherent* element to a synchronous, fully coherent element. Asynchrony is not avoided, but we bound it to the length of the pipeline and substitute the crude partitioning we see in scalable systems so far with a form of "cache affinity" as it typically seen in multi-CPU systems with a shared RAM.

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 Polkadot 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 validate the chain state, and secondly, in sections 14 and 19 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 20 and finally conclude in section 21.

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 E, 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 **cryptoeprint:2024/961**. 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 **wood2014ethereum**. This was derived in large part from the initial concept paper by **buterin2013ethereum**. 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 **buterin2019casper** 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 **ethereum2024danksharding**, the present design direction is to improve on this over the coming years by splitting responsibility for its storage amongst the validator base in a protocol known as *Dank-sharding*.

According to **ethereum2024sigital**, 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 **sharma2024ethereums** 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.

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 **bogli2024assessing**) 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 **koute2024risc0** 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

[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 **hildobby2024eth2** 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.

[4]Some initial thoughts on the matter resulted in a proposal by **sadana2024bringing** to utilize Polkadot technology as a means of helping create a modicum of compatibility between roll-up ecosystems!

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 the PolkaVM 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 **cryptoeprint:2019/953**), 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 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 **thaler2023technical** 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 **kwon2019cosmos**) and Avalanche (by **tanana2019avalanche**). 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 **cosmos2024interchain** 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, **cryptoeprint:2017/406**, would be to

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

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 **cryptoeprint:2024/961**, 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 **yakovenko2018solana** and boasts theoretical figures of over 700,000 transactions per second, though according to **ng2024is** 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. **jha2024solana** 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 **hertig2016so**.

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 **solana2023solana**, 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 **solana2024solana**). 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 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.

[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.

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:

$$y \prec x \iff \exists f : y = f(x) \tag{3.1}$$

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

$$\mathcal{U}(a_0, \dots a_n) \equiv a_x : (a_x \neq \varnothing \vee x = n), \bigwedge_{i=0}^{x-1} a_i = \varnothing \tag{3.2}$$

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

3.3. **Sets.** Given some set $\mathbf{s}$, its power set and cardinality are denoted as $\{\!\{ s \}\!\}$ and $|\mathbf{s}|$. When forming a power set, we may use a numeric subscript in order to restrict the resultant expansion to a particular cardinality. E.g. $\{\!\{ \{1,2,3\} \}\!\}_2 = \{\{1,2\},\{1,3\},\{2,3\}\}$.

Sets may be operated on with scalars, in which case the result is a set with the operation applied to each element, e.g. $\{1,2,3\} + 3 = \{4,5,6\}$. Functions may also be applied to all members of a set to yield a new set, but for clarity we denote this with a $\#$ superscript, e.g. $f^{\#}(\{1,2\}) \equiv \{f(1), f(2)\}$.

We denote set-disjointness with the relation $\downharpoonright$. Formally:

$$A \cap B = \varnothing \iff A \downharpoonright B$$

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 $\mathbf{?}$ 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 $\mathbb{N}_n$ implies a restriction on that set to values 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. $5 \div 3 = 1 \mathsf{R} 2$.

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 domain a *key* and the values within the co-domain a *value*, hence the naming.)

Thus, we define the formalism $\langle \mathrm{K} \to \mathrm{V} \rangle$ to denote a dictionary which maps from the domain K to the range V. It is a subset of the power set of pairs (K, V):

$$\langle \mathrm{K} \to \mathrm{V} \rangle \subset \{\!\{ (\mathrm{K}, \mathrm{V}) \}\!\} \tag{3.3}$$

The subset is caused by a constraint that a dictionary's members must associate at most one unique value for any given key k:

$$\forall \mathrm{K}, \mathrm{V}, \mathbf{d} \in \langle \mathrm{K} \to \mathrm{V} \rangle : \forall (k, v) \in \mathbf{d} : \exists! v' : (k, v') \in \mathbf{d} \tag{3.4}$$

In the context of a dictionary we denote the pairs with a mapping notation:

$$\langle \mathrm{K} \to \mathrm{V} \rangle \equiv \{\!\{ (\mathrm{K} \to \mathrm{V}) \}\!\} \tag{3.5}$$

$$\mathbf{p} \in (\mathrm{K} \to \mathrm{V}) \Leftrightarrow \exists k \in \mathrm{K}, v \in \mathrm{V}, \mathbf{p} \equiv (k \mapsto v) \tag{3.6}$$

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

$$\forall \mathrm{K}, \mathrm{V}, \mathbf{d} \in \langle \mathrm{K} \to \mathrm{V} \rangle : \mathbf{d}[k] \equiv \begin{cases} v & \text{if } \exists k : (k \mapsto v) \in \mathbf{d} \\ \varnothing & \text{otherwise} \end{cases} \tag{3.7}$$

$$\forall \mathrm{K}, \mathrm{V}, \mathbf{d} \in \langle \mathrm{K} \to \mathrm{V} \rangle, \mathbf{s} \subseteq K : \mathbf{d} \setminus \mathbf{s} \equiv \{ (k \mapsto v) : (k \mapsto v) \in \mathbf{d}, k \notin \mathbf{s} \} \tag{3.8}$$

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.

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

$$\forall \mathrm{K}, \mathrm{V}, \mathbf{d} \in \langle \mathrm{K} \to \mathrm{V} \rangle : \mathcal{K}(\mathbf{d}) \equiv \{ k \mid \exists v : (k \mapsto v) \in \mathbf{d} \} \tag{3.9}$$

$$\forall \mathrm{K}, \mathrm{V}, \mathbf{d} \in \langle \mathrm{K} \to \mathrm{V} \rangle : \mathcal{V}(\mathbf{d}) \equiv \{ v \mid \exists k : (k \mapsto v) \in \mathbf{d} \} \tag{3.10}$$

Note that since the co-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.

Dictionaries may be combined through the union operator $\cup$, which priorities the right-side operand in the case of a key-collision:

$$\forall \mathbf{d} \in \mathrm{K}, \mathrm{V}, (\mathbf{d}, \mathbf{e}) \in \langle \mathrm{K} \to \mathrm{V} \rangle^2 : \mathbf{d} \cup \mathbf{e} \equiv (\mathbf{d} \setminus \mathcal{K}(\mathbf{e})) \cup \mathbf{e} \tag{3.11}$$

3.6. **Tuples.** Tuples are groups of values where each item may belong to a different set. They are denoted with parentheses, e.g. the tuple t of the naturals 3 and 5 is denoted $t = (3, 5)$, and it exists in the set of natural 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 natural 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 $\mathbf{s}[i]^{\circlearrowleft} \equiv \mathbf{s}[\, i \,\%\, |\mathbf{s}|\,]$. 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: $[\mathbf{x}_0, \mathbf{x}_1, \dots]_{\dots n}$ 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 $[f(i) \mid i \in \mathbb{N}_n] \equiv [f(0), f(1), \dots, f(n-1)]$. Thus, when the ordering of elements matters we use $\vartriangleleft$ rather than the unordered notation $\in$. The latter may also be written in short form $[f(i \vartriangleleft \mathbb{N}_n)]$. This applies to any set which has an unambiguous ordering, particularly sequences, thus $\left[i^2 \,\middle|\, i \vartriangleleft [1,2,3]\right] = [1,4,9]$. Multiple sequences may be combined, thus $[i \cdot j \mid i \vartriangleleft [1,2,3], j \vartriangleleft [2,3,4]] = [2,6,12]$.

As with sets, we use explicit notation $f^{\#}$ to denote a function mapping over all items of a sequence.

Sequences may be constructed from sets or other sequences whose order should be ignored through sequence ordering notation $[i \in X \,\rangle\, f(i)]$, 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 $f(i)$.

The key component may be elided in which case it is assumed to be ordered by the elements directly; i.e. $[i \in X] \equiv [i \in X \,\rangle\, i]$. $[i \in X \,\rangle\!\rangle\, i]$ does the same, but excludes any duplicate values of i. E.g. assuming $\mathbf{s} = [1,3,2,3]$, then $[i \in \mathbf{s} \,\rangle\!\rangle\, i] = [1,2,3]$ and $[i \in \mathbf{s} \,\rangle\, -i] = [3,3,2,1]$.

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 $[\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{y}_0, \mathbf{y}_1, \dots] \equiv \mathbf{x} \frown \mathbf{y}$. For sequences of sequences, we define a unary concatenate-all operator: $\overset{\frown}{\mathbf{x}} \equiv \mathbf{x}_0 \frown \mathbf{x}_1 \frown \dots$. Further, we denote element concatenation as $x \mathbin{+\!\!+} i \equiv x \frown [i]$. We denote the sequence made up of the first n elements of sequence $\mathbf{s}$ to be $\overrightarrow{\mathbf{s}}^n \equiv [\mathbf{s}_0, \mathbf{s}_1, \dots, \mathbf{s}_{n-1}]$, and only the final elements as $\overleftarrow{\mathbf{s}}^n$.

We define ${}^{\mathsf{T}}\mathbf{x}$ as the transposition of the sequence-of-sequences $\mathbf{x}$, fully defined in equation H.3. We may also apply this to sequences-of-tuples to yield a tuple of sequences.

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 $\mathbb{b}_s = \llbracket \{\bot, \top\} \rrbracket_s$. When dealing with Boolean values we may assume an implicit equivalence mapping to a bit whereby $\top = 1$ and $\bot = 0$, thus $\mathbb{b}_\square = \llbracket \mathbb{N}_2 \rrbracket_\square$. We use the function $\text{bits}(\mathbb{B}) \in \mathbb{b}$ to denote the sequence of bits, ordered with the most significant first, which represent the octet sequence $\mathbb{B}$, thus $\text{bits}([160, 0]) = [1,0,1,0,0,\dots]$.

The unary-not operator applies to both boolean values and sequences of boolean values, thus $\neg\top = \bot$ and $\neg[\top, \bot] = [\bot, \top]$.

3.7.4. *Octets and Blobs.* $\mathbb{B}$ denotes the set of octet strings ("blobs") of arbitrary length. As might be expected, $\mathbb{B}_x$ denotes the set of such sequences of length x. $\mathbb{B}_\$$ denotes the subset of $\mathbb{B}$ 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 natural number form, we do not treat them as exactly equivalent entities. In particular for the purpose of serialization, an octet is always serialized to itself, whereas a natural number may be serialized as a sequence of potentially several octets, depending on its magnitude and the encoding variant.

3.7.5. *Shuffling.* We define the sequence-shuffle function $\mathcal{F}$, originally introduced by **fisheryates1938statistical**, with an efficient in-place algorithm described by **wikipedia2024fisheryates**. 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 naturals or a hash. For a full definition see appendix F.

3.8. **Cryptography.**

3.8.1. *Hashing.* $\mathbb{H}$ denotes the set of 256-bit values equivalent to $\mathbb{B}_{32}$. All hash functions in the present work output to this type and $\mathbb{H}_0$ is the value equal to $[0]_{32}$. We assume a function $\mathcal{H}(m \in \mathbb{B}) \in \mathbb{H}$ denoting the Blake2b 256-bit hash introduced by **rfc7693** and a function $\mathcal{H}_K(m \in \mathbb{B}) \in \mathbb{H}$ denoting the Keccak 256-bit hash as proposed by **bertoni2013keccak** and utilized by **wood2014ethereum**.

The inputs of a hash function should be expected to be passed through our serialization codec $\mathcal{E}$ to yield an octet sequence to which the cryptography may be applied. (Note that an octet sequence conveniently yields an identity transform.) We may wish to interpret a sequence of octets as some other kind of value with the assumed decoder function $\mathcal{E}^{-1}(x \in \mathbb{B})$. 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 $r \in \mathbb{B}_4$, whereas $s = \mathcal{E}_8^{-1}(y)$ would assert $y \in \mathbb{B}_8$ and $s \in \mathbb{N}_{2^{64}}$.

3.8.2. *Signing Schemes.* $\bar{\mathbb{V}}_k\langle m\rangle \subset \mathbb{B}_{64}$ is the set of valid Ed25519 signatures, defined by **rfc8032**, made through knowledge of a secret key whose public key counterpart is $k \in \mathbb{H}$ and whose message is m. To aid readability, we denote the set of valid public keys $\bar{\mathbb{H}}$.

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

$\mathring{\mathbb{V}}^{m\in\mathbb{B}}_{r\in\mathring{\mathbb{B}}}\langle x \in \mathbb{B}\rangle \subset \mathbb{B}_{784}$, 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* $\mathring{\mathbb{B}} \subset \mathbb{B}_{144}$. We denote $\mathcal{O}\Big(\mathbf{s} \in \big[\!\big[\check{\mathbb{H}}\big]\!\big]\Big) \in \mathring{\mathbb{B}}$ 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 $\mathcal{Y}\Big(\mathring{\mathbb{V}}^m_r\langle x\rangle\Big) \subset \mathbb{H}$ and $\mathcal{Y}\Big(\check{\mathbb{V}}^m_k\langle x\rangle\Big) \subset \mathbb{H}$.

We use $\overset{\text{BLS}}{\mathbb{B}} \subset \mathbb{B}_{144}$ to denote the set of public keys for the BLS signature scheme, described by **jofc-2004-14130**, on curve BLS12-381 defined by **bls12-381**. We correspondingly use the notation $\overset{\text{BLS}}{\mathbb{V}}_k\langle m\rangle$ to denote the set of valid BLS signatures for public key $k \in \overset{\text{BLS}}{\mathbb{B}}$ and message $m \in \mathbb{B}$.

We define the signature functions for creating valid signatures; $\bar{\mathcal{S}}(m) \in \bar{\mathbb{V}}_k\langle m\rangle$, $\overset{\text{BLS}}{\mathcal{S}}(m) \in \overset{\text{BLS}}{\mathbb{V}}_k\langle m\rangle$. 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:

$$\sigma' \equiv \Upsilon(\sigma, \mathbf{B}) \tag{4.1}$$

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 $\mathbf{B}$ which may be restated as the header $\mathbf{H}$ and some input data external to the system and thus said to be *extrinsic*, $\mathbf{E}$:

$$\mathbf{B} \equiv (\mathbf{H}, \mathbf{E}) \tag{4.2}$$

$$\mathbf{E} \equiv (\mathbf{E}_T, \mathbf{E}_D, \mathbf{E}_P, \mathbf{E}_A, \mathbf{E}_G) \tag{4.3}$$

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 $\mathbf{E}_T$.
- **preimages:** Static data which is presently being requested to be available for workloads to be able to fetch on demand. This is denoted $\mathbf{E}_P$.
- **reports:** Reports of newly completed workloads whose accuracy is guaranteed by specific validators. This is denoted $\mathbf{E}_G$.
- **availability:** Assurances by each validator concerning which of the input data of workloads they have correctly received and are storing locally. This is denoted $\mathbf{E}_A$.
- **disputes:** Information relating to disputes between validators over the validity of reports. This is denoted $\mathbf{E}_D$.

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:

$$\sigma \equiv (\alpha, \beta, \theta, \gamma, \delta, \eta, \iota, \kappa, \lambda, \rho, \tau, \phi, \chi, \psi, \pi, \omega, \xi) \tag{4.4}$$

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

[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.

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 timeslot index are tracked in β_H and τ respectively, work-reports which are ready to be accumulated and work-packages which were recently accumulated are tracked in ω and ξ respectively and, judgments are tracked in ψ and validator statistics are tracked in π.

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:

$$\tau' \prec \mathbf{H} \tag{4.5}$$

$$\beta^\dagger_H \prec (\mathbf{H}, \beta_H) \tag{4.6}$$

$$\gamma' \prec \left(\mathbf{H}, \tau, \mathbf{E}_T, \gamma, \iota, \eta', \kappa', \psi'\right) \tag{4.7}$$

$$\eta' \prec (\mathbf{H}, \tau, \eta) \tag{4.8}$$

$$\kappa' \prec (\mathbf{H}, \tau, \kappa, \gamma) \tag{4.9}$$

$$\lambda' \prec (\mathbf{H}, \tau, \lambda, \kappa) \tag{4.10}$$

$$\psi' \prec (\mathbf{E}_D, \psi) \tag{4.11}$$

$$\rho^\dagger \prec (\mathbf{E}_D, \rho) \tag{4.12}$$

$$\rho^\ddagger \prec \left(\mathbf{E}_A, \rho^\dagger\right) \tag{4.13}$$

$$\rho' \prec \left(\mathbf{E}_G, \rho^\ddagger, \kappa, \tau'\right) \tag{4.14}$$

$$\mathbf{R}^* \prec \left(\mathbf{E}_A, \rho^\dagger\right) \tag{4.15}$$

$$\left(\omega', \xi', \delta^\ddagger, \chi', \iota', \phi', \theta', \mathbf{S}\right) \prec \left(\mathbf{R}^*, \omega, \xi, \delta, \chi, \iota, \phi, \tau, \tau'\right) \tag{4.16}$$

$$\beta'_H \prec \left(\mathbf{H}, \mathbf{E}_G, \beta^\dagger_H, \theta'\right) \tag{4.17}$$

$$\delta' \prec \left(\mathbf{E}_P, \delta^\ddagger, \tau'\right) \tag{4.18}$$

$$\alpha' \prec \left(\mathbf{H}, \mathbf{E}_G, \phi', \alpha\right) \tag{4.19}$$

$$\pi' \prec \left(\mathbf{E}_G, \mathbf{E}_P, \mathbf{E}_A, \mathbf{E}_T, \tau, \kappa', \pi, \mathbf{H}, \mathbf{S}\right) \tag{4.20}$$

The only synchronous entanglements are visible through the intermediate components superscripted with a dagger and defined in equations 4.6, 4.12, 4.13, 4.14, 4.16, 4.17 and 4.18. The latter two mark a merge and join in the dependency graph and, concretely, imply that the availability extrinsic may be fully processed and accumulation of work happen before the preimage lookup extrinsic is folded into state.

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 19 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 19 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 in 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, 2025.[8] Midday UTC is selected to ensure that all major 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. 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

[8] 1,735,732,800 seconds after the Unix Epoch.

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 19.

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 naturals less than 2^{64} (i.e. 64-bit unsigned integers in coding parlance). Formally:

$$\mathbb{N}_B \equiv \mathbb{N}_{2^{64}} \tag{4.21}$$

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 constrained to being less than 2^{64} implies that there may never be more than around $18{\times}10^9$ 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 *Polkadot Virtual Machine* (PVM). This virtual machine is based around the RISC-V instruction set architecture, specifically the RV64EM 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 (64-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 ($[\![\mathbb{N}_R]\!]_{13}$) 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:

$$\Psi\colon \begin{pmatrix} \mathbb{B}, & \mathbb{N}_R, & \mathbb{N}_G, \\ [\![\mathbb{N}_R]\!]_{13}, & \mathbb{M} \end{pmatrix} \to \begin{pmatrix} \{\blacksquare, \lightning, \infty\} \cup \{\exists, \hbar\} \times \mathbb{N}_R, \\ \mathbb{N}_R, \quad \mathbb{Z}_G, \quad [\![\mathbb{N}_R]\!]_{13}, \quad \mathbb{M} \end{pmatrix} \tag{4.22}$$

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, $\varrho \in \mathbb{N}_G$ is typically used to denote gas.

$$\mathbb{Z}_G \equiv \mathbb{Z}_{-2^{63}\ldots 2^{63}}\ , \quad \mathbb{N}_G \equiv \mathbb{N}_{2^{64}}\ , \quad \mathbb{N}_R \equiv \mathbb{N}_{2^{64}} \tag{4.23}$$

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

The PVM is a very simple RISC *register machine* and as such has 13 registers, each of which is a 64-bit quantity, denoted as $\mathbb{N}_R$, a natural less than 2^{64}.[9] Within the context of the PVM, $\varphi \in [\![\mathbb{N}_R]\!]_{13}$ is typically used to denote the registers.

$$\mathbb{M} \equiv \left(\mathbf{v} \in \mathbb{B}_{2^{32}}, \mathbf{a} \in [\![\{\mathrm{W}, \mathrm{R}, \varnothing\}]\!]_p\right),\ p = \frac{2^{32}}{\mathsf{Z}_P} \tag{4.24}$$

$$\mathsf{Z}_P = 2^{12} \tag{4.25}$$

The PVM assumes a simple pageable RAM of 32-bit addressable octets situated in pages of $\mathsf{Z}_P = 4096$ octets where each page 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.

$$\mathbb{V}_\mu \equiv \{\, i \mid \mu_{\mathbf{a}}[\lfloor i/\mathsf{Z}_P \rfloor] \neq \varnothing \,\} \tag{4.26}$$

$$\mathbb{V}^*_\mu \equiv \{\, i \mid \mu_{\mathbf{a}}[\lfloor i/\mathsf{Z}_P \rfloor] = \mathrm{W} \,\} \tag{4.27}$$

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:

[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

- Regular program termination caused by an explicit halt instruction, $\blacksquare$.
- Irregular program termination caused by some exceptional circumstance, $\lightning$.
- Exhaustion of gas, ∞.
- A page fault (attempt to access some address in RAM which is not accessible), $\exists$. This includes the address of the page 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 natural less than 2^{32} (in compute parlance, a 32-bit unsigned integer) indicating the number of six-second timeslots from the JAM Common Era. For use in this context we introduce the set $\mathbb{N}_T$:

$$\mathbb{N}_T \equiv \mathbb{N}_{2^{32}} \tag{4.28}$$

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, which together with some metadata is known as a *digest*. 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 offenders markers $\mathbf{H}_E$, $\mathbf{H}_W$ and $\mathbf{H}_O$, a block author index $\mathbf{H}_I$ 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:

(5.1)
$$\mathbf{H} \equiv (\mathbf{H}_P, \mathbf{H}_R, \mathbf{H}_X, \mathbf{H}_T, \mathbf{H}_E, \mathbf{H}_W, \mathbf{H}_O, \mathbf{H}_I, \mathbf{H}_V, \mathbf{H}_S)$$

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 $\mathbf{H}^- = P(\mathbf{H})$:

$$\mathbf{H}_P \in \mathbb{H}\,, \quad \mathbf{H}_P \equiv \mathcal{H}(\mathcal{E}(P(\mathbf{H}))) \tag{5.2}$$

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 $\mathbf{A}$:

$$h \in \mathbf{A} \Leftrightarrow h = \mathbf{H} \vee (\exists i \in \mathbf{A} : h = P(i)) \tag{5.3}$$

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 a Merkle commitment to the block's extrinsic data, taking care to allow for the possibility of reports to individually have their inclusion proven. Given any block $\mathbf{B} = (\mathbf{H}, \mathbf{E})$, then formally:

$$\mathbf{H}_X \in \mathbb{H}\,, \quad \mathbf{H}_X \equiv \mathcal{H}\Big(\mathcal{E}\Big(\mathcal{H}^{\#}(\mathbf{a})\Big)\Big) \tag{5.4}$$

$$\text{where } \mathbf{a} = [\mathcal{E}_T(\mathbf{E}_T), \mathcal{E}_P(\mathbf{E}_P), \mathbf{g}, \mathcal{E}_A(\mathbf{E}_A), \mathcal{E}_D(\mathbf{E}_D)] \tag{5.5}$$

$$\text{and } \mathbf{g} = \mathcal{E}(\updownarrow[(\mathcal{H}(\mathbf{r}), \mathcal{E}_4(t), \updownarrow a) \mid (\mathbf{r}, t, a) \lessdot \mathbf{E}_G]) \tag{5.6}$$

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:

$$\mathbf{H}_T \in \mathbb{N}_T\,, \quad P(\mathbf{H})_T < \mathbf{H}_T \ \wedge\ \mathbf{H}_T \cdot \mathsf{P} \le \mathcal{T} \tag{5.7}$$

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

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.

$$\mathbf{H}_R \in \mathbb{H}\,, \quad \mathbf{H}_R \equiv \mathcal{M}_\sigma(\sigma) \tag{5.8}$$

We assume the state-Merklization function $\mathcal{M}_\sigma$ 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 posterior 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.

$$\mathbf{H}_I \in \mathbb{N}_\mathsf{V}\,, \quad \mathbf{H}_A \equiv \kappa'[\mathbf{H}_I]_b \tag{5.9}$$

5.1. The 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). Finally, the offenders marker is the sequence of Ed25519 keys of newly misbehaving validators, to be fully explained in section 10. Formally:

(5.10)
$$\mathbf{H}_E \in \Big(\mathbb{H}, \mathbb{H}, \Big[\Big(\tilde{\mathbb{H}}, \bar{\mathbb{H}}\Big)\Big]_\mathsf{V}\Big)?\,, \quad \mathbf{H}_W \in [\![\mathbb{T}]\!]_\mathsf{E}?\,, \quad \mathbf{H}_O \in [\![\bar{\mathbb{H}}]\!]$$

The terms are fully defined in sections 6.6 and 10.

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 in regular operation, 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). This anonymous and secure random output is a *ticket* and validators' tickets with the best score define the new sealing keys allowing the chosen validators to exercise their privilege and create a new block at the appropriate time.

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:

$$\tau \in \mathbb{N}_T \;, \quad \tau' \equiv \mathbf{H}_T \tag{6.1}$$

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:

$$\text{let} \quad e \,\mathsf{R}\, m = \frac{\tau}{\mathsf{E}}\,, \quad e' \,\mathsf{R}\, m' = \frac{\tau'}{\mathsf{E}} \tag{6.2}$$

6.2. Safrole Basic State.

We restate γ into a number of components:

$$\gamma \equiv (\gamma_P, \gamma_Z, \gamma_S, \gamma_A) \tag{6.3}$$

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 γ_P (itself defined in the next section).

$$\gamma_Z \in \mathring{\mathbb{B}} \tag{6.4}$$

Finally, γ_A is the ticket accumulator, a series of highest-scoring ticket identifiers to be used for the next epoch. γ_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:

$$\gamma_A \in \llbracket \mathbb{T} \rrbracket_{:\mathsf{E}}\,, \quad \gamma_S \in \llbracket \mathbb{T} \rrbracket_{\mathsf{E}} \cup \left\llbracket \mathring{\mathbb{H}} \right\rrbracket_{\mathsf{E}} \tag{6.5}$$

Here, $\mathbb{T}$ is used to denote the set of *tickets*, a combination of a verifiably random ticket identifier y and the ticket's entry-index e:

$$\mathbb{T} \equiv (y \in \mathbb{H}, e \in \mathbb{N}_{\mathsf{N}}) \tag{6.6}$$

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 γ_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 γ_P. 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 γ_P, 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.

$$\iota \in \llbracket \mathbb{K} \rrbracket_{\mathsf{V}}\,, \quad \gamma_P \in \llbracket \mathbb{K} \rrbracket_{\mathsf{V}}\,, \quad \kappa \in \llbracket \mathbb{K} \rrbracket_{\mathsf{V}}\,, \quad \lambda \in \llbracket \mathbb{K} \rrbracket_{\mathsf{V}} \tag{6.7}$$

We must introduce $\mathbb{K}$, the set of validator key tuples. This is a combination of a set of cryptographic public keys and metadata 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 336-octet sequences. However, for clarity, we divide the sequence into four easily denoted components. For any validator key k, the Bandersnatch key is denoted k_b, and is equivalent to the first 32-octets; the Ed25519 key, k_e, is the second 32 octets; the BLS key denoted k_l is equivalent to the following 144 octets, and finally the metadata k_m is the last 128 octets. Formally:

$$\mathbb{K} \equiv \mathbb{B}_{336} \tag{6.8}$$

$$\forall k \in \mathbb{K} : k_b \in \mathring{\mathbb{H}} \equiv k_{0\cdots+32} \tag{6.9}$$

$$\forall k \in \mathbb{K} : k_e \in \bar{\mathbb{H}} \equiv k_{32\cdots+32} \tag{6.10}$$

$$\forall k \in \mathbb{K} : k_l \in \overset{\text{BLS}}{\mathbb{B}} \equiv k_{64\cdots+144} \tag{6.11}$$

$$\forall k \in \mathbb{K} : k_m \in \mathbb{B}_{128} \equiv k_{208\cdots+128} \tag{6.12}$$

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

$$\left(\gamma'_P, \kappa', \lambda', \gamma'_Z\right) \equiv \begin{cases} (\Phi(\iota), \gamma_P, \kappa, z) & \text{if } e' > e \\ (\gamma_P, \kappa, \lambda, \gamma_Z) & \text{otherwise} \end{cases} \tag{6.13}$$

$$\text{where } z = \mathcal{O}\left(\left[k_b \mid k \in \gamma'_P\right]\right)$$

$$\Phi(\mathbf{k}) \equiv \left[\begin{cases} [0,0,\dots] & \text{if } k_e \in \psi'_O \\ k & \text{otherwise} \end{cases} \;\middle|\; k \in \mathbf{k} \right] \tag{6.14}$$

Note that on epoch changes the posterior queued validator key set γ'_P is defined such that incoming keys belonging to the offenders ψ'_O are replaced with a null key containing only zeroes. The origin of the offenders 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:

let $i = \gamma'_S[\mathbf{H}_T]^{\circlearrowleft}$:

$$\gamma'_S \in [\![\mathbb{T}]\!] \implies \begin{cases} i_y = \mathcal{Y}(\mathbf{H}_S), \\ \mathbf{H}_S \in \check{\mathbb{V}}^{\mathcal{E}_U(\mathbf{H})}_{\mathbf{H}_A}\left\langle \mathsf{X}_T \frown \eta'_3 \mathbin{+\!\!+} i_e \right\rangle, \\ \mathbf{T} = 1 \end{cases} \tag{6.15}$$

$$\gamma'_S \in \left[\!\!\left[\check{\mathbb{H}}\right]\!\!\right] \implies \begin{cases} i = \mathbf{H}_A, \\ \mathbf{H}_S \in \check{\mathbb{V}}^{\mathcal{E}_U(\mathbf{H})}_{\mathbf{H}_A}\left\langle \mathsf{X}_F \frown \eta'_3 \right\rangle, \\ \mathbf{T} = 0 \end{cases} \tag{6.16}$$

$$\mathbf{H}_V \in \check{\mathbb{V}}^{[]}_{\mathbf{H}_A}\langle \mathsf{X}_E \frown \mathcal{Y}(\mathbf{H}_S)\rangle \tag{6.17}$$

$$\mathsf{X}_E = \texttt{\$jam\_entropy} \tag{6.18}$$

$$\mathsf{X}_F = \texttt{\$jam\_fallback\_seal} \tag{6.19}$$

$$\mathsf{X}_T = \texttt{\$jam\_ticket\_seal} \tag{6.20}$$

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 19. 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 6.29) and seed the fallback seal-key generation function with randomness (see equation 6.24). The oldest is used to regenerate this randomness when verifying the seal above (see equations 6.16 and 6.15).

$$\eta \in [\![\mathbb{H}]\!]_4 \tag{6.21}$$

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, η_2 and η_3 meanwhile retain the state of this accumulator at the end of the three most recently ended epochs in order.

$$\eta'_0 \equiv \mathcal{H}(\eta_0 \frown \mathcal{Y}(\mathbf{H}_V)) \tag{6.22}$$

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:

$$\left(\eta'_1, \eta'_2, \eta'_3\right) \equiv \begin{cases} (\eta_0, \eta_1, \eta_2) & \text{if } e' > e \\ (\eta_1, \eta_2, \eta_3) & \text{otherwise} \end{cases} \tag{6.23}$$

6.5. **The Slot Key Sequence.** The posterior slot key sequence $\mathbb{T}$ 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 γ_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 γ_A. Otherwise, it takes the value of the fallback key sequence. Formally:

$$\gamma'_S \equiv \begin{cases} Z(\gamma_A) & \text{if } e' = e+1 \wedge m \geq \mathsf{Y} \wedge |\gamma_A| = \mathsf{E} \\ \gamma_S & \text{if } e' = e \\ F(\eta'_2, \kappa') & \text{otherwise} \end{cases} \tag{6.24}$$

Here, we use Z as the outside-in sequencer function, defined as follows:

$$Z: \begin{cases} [\![\mathbb{T}]\!]_{\mathsf{E}} \to [\![\mathbb{T}]\!]_{\mathsf{E}} \\ \quad \mathbf{s} \mapsto \left[\mathbf{s}_0, \mathbf{s}_{|\mathbf{s}|-1}, \mathbf{s}_1, \mathbf{s}_{|\mathbf{s}|-2}, \dots\right] \end{cases} \tag{6.25}$$

Finally, F is the fallback key sequence function which selects an epoch's worth of validator Bandersnatch keys ($\left[\!\!\left[\check{\mathbb{H}}\right]\!\!\right]_{\mathsf{E}}$) from the validator key set $\mathbf{k}$ using the entropy collected on-chain r:

$$F: \begin{cases} (\mathbb{H}, [\![\mathbb{K}]\!]) \to \left[\!\!\left[\check{\mathbb{H}}\right]\!\!\right]_{\mathsf{E}} \\ \quad (r, \mathbf{k}) \mapsto \left[\mathbf{k}_{\mathcal{E}_4^{-1}(\mathcal{H}(r \frown \mathcal{E}_4(i))_{\dots 4})}{}^{\circlearrowleft}_{b} \;\middle|\; i \in \mathbb{N}_{\mathsf{E}} \right] \end{cases} \tag{6.26}$$

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 next and current epoch randomness, along with a sequence of tuples containing both Bandersnatch keys and Ed25519 keys for each validator defining the validator keys beginning in the next epoch. Formally:

$$\mathbf{H}_E \equiv \begin{cases} (\eta_0, \eta_1, [(k_b, k_e) \mid k \in \gamma'_P]) & \text{if } e' > e \\ \varnothing & \text{otherwise} \end{cases} \tag{6.27}$$

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:

$$\mathbf{H}_W \equiv \begin{cases} Z(\gamma_A) & \text{if } e' = e \wedge m < \mathsf{Y} \leq m' \wedge |\gamma_A| = \mathsf{E} \\ \varnothing & \text{otherwise} \end{cases} \tag{6.28}$$

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 entry index together with a proof of ticket's validity. The proof implies a ticket identifier, 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:

$$\mathbf{E}_T \in \left[\left(e \in \mathbb{N}_\mathsf{N},\, p \in \mathring{\mathbb{V}}^{[]}_{\gamma'_Z}\left\langle \mathsf{X}_T \frown \eta'_2 \mathbin{+\!\!+} e\right\rangle\right)\right] \tag{6.29}$$

$$|\mathbf{E}_T| \le \begin{cases} \mathsf{K} & \text{if } m' < \mathsf{Y} \\ 0 & \text{otherwise} \end{cases} \tag{6.30}$$

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

$$\mathbf{n} \equiv [(y\colon \mathcal{Y}(i_p),\, e\colon i_e) \mid i \in \mathbf{E}_T] \tag{6.31}$$

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

$$\mathbf{n} = [x \in \mathbf{n} \,\rangle\!\rangle\, x_y] \tag{6.32}$$

$$\{\, x_y \mid x \in \mathbf{n} \,\} \subseteq_{\circ} \{\, x_y \mid x \in \gamma_A \,\} \tag{6.33}$$

The new ticket accumulator γ'_A is constructed by merging new tickets into the previous accumulator value (or the empty sequence if it is a new epoch):

$$\gamma'_A \equiv \overrightarrow{\left[x \in \mathbf{n} \cup \left\{\begin{matrix} \varnothing & \text{if } e' > e \\ \gamma_A & \text{otherwise} \end{matrix}\right\} x_y\right]}^{\mathsf{E}} \tag{6.34}$$

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 γ_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:

$$\mathbf{n} \subseteq \gamma'_A \tag{6.35}$$

Note that it can be shown that in the case of an empty extrinsic $\mathbf{E}_T = []$, as implied by $m' \ge \mathsf{Y}$, and unchanged epoch ($e' = e$), then $\gamma'_A = \gamma_A$.

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.

$$\beta \equiv (\beta_H, \beta_B) \tag{7.1}$$

$$\beta_H \in \llbracket (h \in \mathbb{H}, s \in \mathbb{H}, b \in \mathbb{H}, \mathbf{p} \in \langle \mathbb{H} \to \mathbb{H} \rangle) \rrbracket_{:\mathsf{H}} \tag{7.2}$$

$$\beta_B \in \llbracket \mathbb{H}? \rrbracket \tag{7.3}$$

$$\theta \in \llbracket (\mathbb{N}_S, \mathbb{H}) \rrbracket \tag{7.4}$$

For each recent block, we retain its header hash, its state root, its accumulation-result MMB and the corresponding work-package hashes of each item reported (which is no more than the total number of cores, $\mathsf{C} = 341$).

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

$$\beta^\dagger_H \equiv \beta_H \quad \text{except} \quad \beta^\dagger_H[|\beta_H| - 1]_s = \mathbf{H}_R \tag{7.5}$$

We define the new Accumulation Output Log β_B. This is formed from the block's accumulation-output sequence θ' (defined in section 12), taking its root using the basic binary Merklization function ($\mathcal{M}_B$, defined in appendix E) and appending it to the previous log value with the MMB append function (defined in appendix E.2). Throughout, the Keccak hash function is used to maximize compatibility with legacy systems:

$$\text{let } \mathbf{s} = \left[\mathcal{E}_4(s) \frown \mathcal{E}(h) \;\middle|\; (s,h) \lhd \theta'\right] \tag{7.6}$$

$$\beta'_B \equiv \mathcal{A}(\beta_B, \mathcal{M}_B(\mathbf{s}, \mathcal{H}_K), \mathcal{H}_K) \tag{7.7}$$

The final state transition for β_H appends a new item including the new block's header hash, a Merkle commitment to the block's Accumulation Output Log and the set of work-reports made into it (for which we use the guarantees extrinsic, $\mathbf{E}_G$). Formally:

$$\beta'_H \equiv \overleftarrow{\beta^\dagger_H \mathbin{+\!\!+} \left(\mathbf{p}, h\colon \mathcal{H}(\mathbf{H}), s\colon \mathbb{H}_0, b\colon \mathcal{M}_R(\beta'_B)\right)}^{\mathsf{H}} \quad \text{where } \mathbf{p} = \{\, (((g_\mathbf{r})_\mathbf{s})_p \mapsto ((g_\mathbf{r})_\mathbf{s})_e) \mid g \in \mathbf{E}_G \,\} \tag{7.8}$$

The new state-trie root is the zero hash, $\mathbb{H}_0$, which is inaccurate but safe since β' is not utilized except to define the next block's $\beta^\dagger$, which contains a corrected value for this, as per equation 7.5.

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 *core-time* 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 three key concepts: *Authorizers*, *Tokens* and *Traces*. A Token is simply a piece of opaque data to be included with a work-package to help make an argument that the work-package should be authorized. Similarly, a Trace is a piece of opaque data which helps characterize or describe some successful authorization. An Authorizer meanwhile, is a piece of logic which executes within some pre-specified and well-known computational limits and determines whether a work-package—including its Token—is authorized for execution on some particular core and yields a Trace on success.

Authorizers are identified as the hash of their PVM code concatenated with their Configuration blob, the latter being, like Tokens and Traces, opaque data meaningful to the PVM code. 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 14.3. 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* $\phi[c]$, from which we draw values to fill the pool. Formally:

$$\alpha \in \left[\left[\mathbb{H}\right]_{:\mathsf{O}}\right]_{\mathsf{C}}, \qquad \phi \in \left[\left[\mathbb{H}\right]_{\mathsf{Q}}\right]_{\mathsf{C}} \tag{8.1}$$

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:

$$\forall c \in \mathbb{N}_{\mathsf{C}} : \alpha'[c] \equiv \overleftarrow{F(c) \pplus \phi'[c][\mathbf{H}_T]}^{\mathsf{O}} \tag{8.2}$$

$$F(c) \equiv \begin{cases} \alpha[c] \circledwedge \{\,(g_{\mathbf{r}})_a\,\} & \text{if } \exists g \in \mathbf{E}_G : (g_{\mathbf{r}})_c = c \\ \alpha[c] & \text{otherwise} \end{cases} \tag{8.3}$$

Since α' is dependent on ϕ', practically speaking, this step must be computed after accumulation, the stage in which ϕ' is defined. Note that we utilize the guarantees extrinsic $\mathbf{E}_G$ to remove the oldest authorizer which has been used to justify a guaranteed work-package in the current block. This is further defined in equation 11.23.

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:

$$\mathbb{N}_S \equiv \mathbb{N}_{2^{32}} \tag{9.1}$$

$$\delta \in \left\langle \mathbb{N}_S \to \mathbb{A} \right\rangle \tag{9.2}$$

The service account is defined as the tuple of storage dictionary $\mathbf{s}$, preimage lookup dictionaries $\mathbf{p}$ and $\mathbf{l}$, code hash c, balance b and gratis storage offset f, as well as the two code gas limits g & m. We also record certain usage characteristics concerning the account: the time slot at creation r, the time slot at the most recent accumulation a and the parent service p. Formally:

$$\mathbb{A} \equiv \begin{pmatrix} \mathbf{s} \in \left\langle \mathbb{B} \to \mathbb{B} \right\rangle, \ \mathbf{p} \in \left\langle \mathbb{H} \to \mathbb{B} \right\rangle, \\ \mathbf{l} \in \left\langle (\mathbb{H}, \mathbb{N}_L) \to \left[\mathbb{N}_T\right]_{:3} \right\rangle, \\ f \in \mathbb{N}_B, \ c \in \mathbb{H}, \ b \in \mathbb{N}_B, \ g \in \mathbb{N}_G, \\ m \in \mathbb{N}_G, \ r \in \mathbb{N}_T, \ a \in \mathbb{N}_T, \ p \in \mathbb{N}_S \end{pmatrix} \tag{9.3}$$

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

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

$$\forall \mathbf{a} \in \mathbb{A} : (\mathbf{a}_{\mathbf{m}}, \mathbf{a}_{\mathbf{c}}) \equiv \begin{cases} (\mathbf{m}, \mathbf{c}) & \text{if } \mathcal{E}(\updownarrow \mathbf{m}, \mathbf{c}) = \mathbf{a}_{\mathbf{p}}[\mathbf{a}_c] \\ (\varnothing, \varnothing) & \text{otherwise} \end{cases} \tag{9.4}$$

There are two entry-points in the code:

- **0 `refine`:** Refinement, executed in-core and stateless.[10]
- **1 `accumulate`:** Accumulation, executed on-chain and stateful.

Refinement and accumulation are described in more detail in sections 14.4 and 12.2 respectively.

As stated in appendix A, execution time in the JAM virtual machine is measured deterministically in units of *gas*, represented as a natural number less than 2^{64} and formally denoted $\mathbb{N}_G$. We may also use $\mathbb{Z}_G$ to denote the set $\mathbb{Z}_{-2^{63}\ldots 2^{63}}$ if the quantity may be negative. There are two limits specified in the account, which determine the minimum gas required in order to execute the *Accumulate* entry-point of the service's code. g is the minimum gas required per work-item, while m is the minimum gas required per deferred-transfer.

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

[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 14.3.

it goes through a process of being marked as unavailable, and only after a period of time 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's implications 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 judgments 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 was available for lookup by some service account at some timeslot, and if so, provide it:

$$\Lambda: \begin{cases} (\mathbb{A}, \mathbb{N}_{(\mathbf{H}_T-\mathsf{D})\ldots\mathbf{H}_T}, \mathbb{H}) \to \mathbb{B}? \\ \qquad (\mathbf{a}, t, \mathcal{H}(\mathbf{p})) \mapsto v : v \in \{\mathbf{p}, \varnothing\} \end{cases} \tag{9.5}$$

This function is defined shortly below in equation 9.7.

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 9.6. Secondly, a preimage value being in state implies that its hash and length pair has some associated status, also in equation 9.6. Formally:

$$\forall \mathbf{a} \in \mathbb{A}, (h \mapsto \mathbf{d}) \in \mathbf{a}_{\mathbf{p}} \Rightarrow h = \mathcal{H}(\mathbf{d}) \wedge (h, |\mathbf{d}|) \in \mathcal{K}(\mathbf{a}_{\mathbf{l}}) \tag{9.6}$$

9.2.2. *Semantics.* The historical status component $h \in \lbrace\!\lbrace \mathbb{N}_T \rbrace\!\rbrace_{:3}$ 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.
- $h \in \lbrace\!\lbrace \mathbb{N}_T \rbrace\!\rbrace_1$: The preimage is *available* and has been from time h_0.
- $h \in \lbrace\!\lbrace \mathbb{N}_T \rbrace\!\rbrace_2$: The previously available preimage is now *unavailable* since time h_1. It had been available from time h_0.
- $h \in \lbrace\!\lbrace \mathbb{N}_T \rbrace\!\rbrace_3$: 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:

$$\Lambda: (\mathbb{A}, \mathbb{N}_T, \mathbb{H}) \to \mathbb{B}?$$

$$\Lambda(\mathbf{a}, t, h) \equiv \begin{cases} \mathbf{a}_{\mathbf{p}}[h] \text{ if } h \in \mathcal{K}(\mathbf{a}_{\mathbf{p}}) \wedge I(\mathbf{a}_{\mathbf{l}}[h, |\mathbf{a}_{\mathbf{p}}[h]|], t) \\ \varnothing \qquad \text{otherwise} \end{cases}$$

$$\text{where } I(\mathbf{l}, t) = \begin{cases} \bot & \text{if } [\,] = \mathbf{l} \\ x \le t & \text{if } [x] = \mathbf{l} \\ x \le t < y & \text{if } [x, y] = \mathbf{l} \\ x \le t < y \vee z \le t & \text{if } [x, y, z] = \mathbf{l} \end{cases} \tag{9.7}$$

9.3. **Account Footprint and Threshold Balance.** We define the dependent values i and o 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 third dependent term t, the minimum, or *threshold*, balance needed for any given service account in terms of its storage footprint.

$$\forall \mathbf{a} \in \mathcal{V}(\delta): \begin{cases} \mathbf{a}_i \in \mathbb{N}_{2^{32}} \equiv 2 \cdot |\,\mathbf{a}_{\mathbf{l}}\,| + |\,\mathbf{a}_{\mathbf{s}}\,| \\ \mathbf{a}_o \in \mathbb{N}_{2^{64}} \equiv \sum_{(h,z) \in \mathcal{K}(\mathbf{a}_{\mathbf{l}})} 81 + z \\ \qquad\qquad + \sum_{(x,y) \in \mathbf{a}_{\mathbf{s}}} 34 + |y| + |x| \\ \mathbf{a}_t \in \mathbb{N}_B \equiv \max(0, \mathsf{B}_S + \mathsf{B}_I \cdot \mathbf{a}_i + \mathsf{B}_L \cdot \mathbf{a}_o - \mathbf{a}_f) \end{cases} \tag{9.8}$$

9.4. **Service Privileges.** JAM includes the ability to bestow privileges on a number of services. The portion of state in which this is held is denoted χ and includes five kinds of privilege. The first, χ_M, is the index of the *manager* service which is the service able to effect an alteration of χ from block to block as well as bestow services with storage deposit credits. The next, χ_V, is able to set ι. Then χ_R alone is able to create new service accounts with indices in the protected range. The following, χ_A, are the service indices capable of altering the authorizer queue ϕ, one for each core.

Finally, χ_Z is a small dictionary containing the indices of services which automatically accumulate in each block together with a basic amount of gas with which each accumulates. Formally:

$$\chi \equiv (\chi_M, \chi_V, \chi_R, \chi_A, \chi_Z) \tag{9.9}$$

$$\chi_M \in \mathbb{N}_S\,, \qquad \chi_V \in \mathbb{N}_S\,, \qquad \chi_R \in \mathbb{N}_S \tag{9.10}$$

$$\chi_A \in \lbrace\!\lbrace \mathbb{N}_S \rbrace\!\rbrace_{\mathsf{C}}\,, \qquad \chi_Z \in \langle \mathbb{N}_S \to \mathbb{N}_G \rangle \tag{9.11}$$

10. Disputes, Verdicts and Judgments

JAM provides a means of recording *judgments*: consequential votes amongst most of the validators over the validity of a *work-report* (a unit of work done within JAM, see section 11). Such collections of judgments are known

as *verdicts*. JAM also provides a means of registering *offenses*, judgments and guarantees which dissent with an established *verdict*. Together these form the *disputes* system.

The registration of a verdict is not expected to happen very often in practice, however it is an important security backstop for removing and banning invalid work-reports from the processing pipeline as well as removing troublesome keys from the validator set where there is consensus over their malfunction. It also helps coordinate nodes to revert chain-extensions containing invalid work-reports and provides a convenient means of aggregating all offending validators for punishment in a higher-level system.

Judgement statements come about naturally as part of the auditing process and are expected to be positive, further affirming the guarantors' assertion that the work-report is valid. In the event of a negative judgment, then all validators audit said work-report and we assume a verdict will be reached. Auditing and guaranteeing are off-chain processes properly described in sections 14 and 17.

A judgment against a report implies that the chain is already reverted to some point prior to the accumulation of said report, usually forking at the block immediately prior to that at which accumulation happened. The specific strategy for chain selection is described fully in section 19. Authoring a block with a non-positive verdict has the effect of cancelling its imminent accumulation, as can be seen in equation 10.15.

Registering a verdict also has the effect of placing a permanent record of the event on-chain and allowing any offending keys to be placed on-chain both immediately or in forthcoming blocks, again for permanent record.

Having a persistent on-chain record of misbehavior is helpful in a number of ways. 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. **The State.** The *disputes* state includes four items, three of which concern verdicts: a good-set (ψ_G), a bad-set (ψ_B) and a wonky-set (ψ_W) containing the hashes of all work-reports which were respectively judged to be correct, incorrect or that it appears impossible to judge. The fourth item, the punish-set (ψ_O), is a set of Ed25519 keys representing validators which were found to have misjudged a work-report.

$$\psi \equiv (\psi_G, \psi_B, \psi_W, \psi_O) \tag{10.1}$$

10.2. **Extrinsic.** The disputes extrinsic $\mathbf{E}_D$ is functional grouping of three otherwise independent extrinsics. It comprises *verdicts* $\mathbf{E}_V$, *culprits* $\mathbf{E}_C$, and *faults* $\mathbf{E}_F$. Verdicts are a compilation of judgments coming from exactly two-thirds plus one of either the active validator set or the previous epoch's validator set, i.e. the Ed25519 keys of κ or λ. Culprits and faults are proofs of the misbehavior of one or more validators, respectively either by guaranteeing a work-report found to be invalid, or by signing a judgment found to be contradiction to a work-report's validity. Both of these are considered a kind of *offense*. Formally:

$$\mathbf{E}_D \equiv (\mathbf{E}_V, \mathbf{E}_C, \mathbf{E}_F) \tag{10.2}$$

$$\text{where } \mathbf{E}_V \in \left\llbracket \left(\mathbb{H}, \left\lfloor \frac{\tau}{\mathsf{E}} \right\rfloor - \mathbb{N}_2, \left\llbracket (\{\top, \bot\}, \mathbb{N}_\mathsf{V}, \bar{\mathbb{V}}) \right\rrbracket_{\lfloor 2/3\mathsf{V} \rfloor + 1} \right) \right\rrbracket$$

$$\text{and } \mathbf{E}_C \in \left\llbracket (\mathbb{H}, \bar{\mathbb{H}}, \bar{\mathbb{V}}) \right\rrbracket, \quad \mathbf{E}_F \in \left\llbracket (\mathbb{H}, \{\top, \bot\}, \bar{\mathbb{H}}, \bar{\mathbb{V}}) \right\rrbracket$$

The signatures of all judgments must be valid in terms of one of the two allowed validator key-sets, identified by the verdict's second term which must be either the epoch index of the prior state or one less. Formally:

$$\forall (r, a, \mathbf{j}) \in \mathbf{E}_V, \forall (v, i, s) \in \mathbf{j} : s \in \bar{\mathbb{V}}_{\mathbf{k}[i]_e} \langle \mathsf{X}_v \frown r \rangle \tag{10.3}$$

$$\text{where } \mathbf{k} = \begin{cases} \kappa & \text{if } a = \left\lfloor \frac{\tau}{\mathsf{E}} \right\rfloor \\ \lambda & \text{otherwise} \end{cases}$$

$$\mathsf{X}_\top \equiv \texttt{\$jam\_valid}, \ \mathsf{X}_\bot \equiv \texttt{\$jam\_invalid} \tag{10.4}$$

Offender signatures must be similarly valid and reference work-reports with judgments and may not report keys which are already in the punish-set:

$$\forall (r, f, s) \in \mathbf{E}_C : \bigwedge \begin{cases} r \in \psi'_B, \\ f \in \mathbf{k}, \\ s \in \bar{\mathbb{V}}_f \langle \mathsf{X}_G \frown r \rangle \end{cases} \tag{10.5}$$

$$\forall (r, v, f, s) \in \mathbf{E}_F : \bigwedge \begin{cases} r \in \psi'_B \Leftrightarrow r \notin \psi'_G \Leftrightarrow v, \\ k \in \mathbf{k}, \\ s \in \bar{\mathbb{V}}_f \langle \mathsf{X}_v \frown r \rangle \end{cases} \tag{10.6}$$

$$\text{where } \mathbf{k} = \{ i_e \mid i \in \lambda \cup \kappa \} \setminus \psi_O$$

Verdicts $\mathbf{E}_V$ must be ordered by report hash. Offender signatures $\mathbf{E}_C$ and $\mathbf{E}_F$ must each be ordered by the validator's Ed25519 key. There may be no duplicate report hashes within the extrinsic, nor amongst any past reported hashes. Formally:

$$\mathbf{E}_V = [(r, a, \mathbf{j}) \in \mathbf{E}_V \,\backslash\!\backslash\, r] \tag{10.7}$$

$$\mathbf{E}_C = [(r, f, s) \in \mathbf{E}_C \,\backslash\!\backslash\, f], \ \mathbf{E}_F = [(r, v, f, s) \in \mathbf{E}_F \,\backslash\!\backslash\, f] \tag{10.8}$$

$$\{ r \mid (r, a, \mathbf{j}) \in \mathbf{E}_V \} \eqslantless \psi_G \cup \psi_B \cup \psi_W \tag{10.9}$$

The judgments of all verdicts must be ordered by validator index and there may be no duplicates:

$$\forall (r, a, \mathbf{j}) \in \mathbf{E}_V : \mathbf{j} = [(v, i, s) \in \mathbf{j} \,\backslash\!\backslash\, i] \tag{10.10}$$

We define $\mathbf{v}$ to derive from the sequence of verdicts introduced in the block's extrinsic, containing only the report hash and the sum of positive judgments. We require this total to be either exactly two-thirds-plus-one, zero or one-third of the validator set indicating, respectively, that the report is good, that it's bad, or that it's wonky.[11] Formally:

$$\mathbf{v} \in \llbracket (\mathbb{H}, \{ 0, \lfloor 1/3\mathsf{V} \rfloor, \lfloor 2/3\mathsf{V} \rfloor + 1 \}) \rrbracket \tag{10.11}$$

$$\mathbf{v} = \left[\left(r, \sum_{(v,i,s) \in \mathbf{j}} v \right) \,\middle|\, (r, a, \mathbf{j}) \lessdot \mathbf{E}_V \right] \tag{10.12}$$

[11] 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 (**cryptoeprint:2024/961** discusses the security implications in depth).

There are some constraints placed on the composition of this extrinsic: any verdict containing solely valid judgments implies the same report having at least one valid entry in the faults sequence $\mathbf{E}_F$. Any verdict containing solely invalid judgments implies the same report having at least two valid entries in the culprits sequence $\mathbf{E}_C$. Formally:

$$\forall (r, \lfloor 2/3 \mathsf{V} \rfloor + 1) \in \mathbf{v} : \exists (r, \dots) \in \mathbf{E}_F \tag{10.13}$$

$$\forall (r, 0) \in \mathbf{v} : |\{ (r, \dots) \in \mathbf{E}_C \}| \geq 2 \tag{10.14}$$

We clear any work-reports which we judged as uncertain or invalid from their core:

$$\forall c \in \mathbb{N}_\mathsf{C} : \rho^\dagger[c] = \begin{cases} \varnothing & \text{if } \left(\mathcal{H}\left(\rho[c]_\mathbf{r}\right), t\right) \in \mathbf{v}, t < \lfloor 2/3 \mathsf{V} \rfloor \\ \rho[c] & \text{otherwise} \end{cases} \tag{10.15}$$

The state's good-set, bad-set and wonky-set assimilate the hashes of the reports from each verdict. Finally, the punish-set accumulates the keys of any validators who have been found guilty of offending. Formally:

$$\psi'_G \equiv \psi_G \cup \{ r \mid (r, \lfloor 2/3 \mathsf{V} \rfloor + 1) \in \mathbf{v} \} \tag{10.16}$$

$$\psi'_B \equiv \psi_B \cup \{ r \mid (r, 0) \in \mathbf{v} \} \tag{10.17}$$

$$\psi'_W \equiv \psi_W \cup \{ r \mid (r, \lfloor 1/3 \mathsf{V} \rfloor) \in \mathbf{v} \} \tag{10.18}$$

$$\psi'_O \equiv \psi_O \cup \{ f \mid (f, \dots) \in \mathbf{E}_C \} \cup \{ f \mid (f, \dots) \in \mathbf{E}_F \} \tag{10.19}$$

10.3. **Header.** The offenders markers must contain exactly the keys of all new offenders, respectively. Formally:

$$\mathbf{H}_O \equiv [f \mid (f, \dots) \in \mathbf{E}_C] \frown [f \mid (f, \dots) \in \mathbf{E}_F] \tag{10.20}$$

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 their way into the state of service accounts, 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-digests* 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 enough assurances the work-report is considered *available*, and the work-digests transform the state of their associated service by virtue of accumulation, covered in section 12. The report may also be *timed-out*, implying it may be replaced by another report without accumulation.

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 11.29.

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 are not yet known to be available to a super-majority of validators, together with the time at which each was reported. As mentioned earlier, only one report may be assigned to a core at any given time. Formally:

$$\rho \in \left[\!\left[(\mathbf{r} \in \mathbb{R}, t \in \mathbb{N}_T)? \right]\!\right]_\mathsf{C} \tag{11.1}$$

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 $\mathbb{R}$, is defined as a tuple of the work-package specification, $\mathbf{s}$; the refinement context, $\mathbf{c}$; the core-index (i.e. on which the work is done), c; as well as the authorizer hash a and trace $\mathbf{t}$; a segment-root lookup dictionary $\mathbf{l}$; the gas consumed during the Is-Authorized invocation, g; and finally the work-digests $\mathbf{d}$ which comprise the results of the evaluation of each of the items in the package together with some associated data. Formally:

$$\mathbb{R} \equiv \begin{pmatrix} \mathbf{s} \in \mathbb{Y},\ \mathbf{c} \in \mathbb{C},\ c \in \mathbb{N}_\mathsf{C},\ a \in \mathbb{H},\ \mathbf{t} \in \mathbb{B}, \\ \mathbf{l} \in \langle \mathbb{H} \to \mathbb{H} \rangle,\ \mathbf{d} \in \left[\!\left[\mathbb{D} \right]\!\right]_{1:\mathsf{I}},\ g \in \mathbb{N}_G \end{pmatrix} \tag{11.2}$$

We limit the sum of the number of items in the segment-root lookup dictionary and the number of prerequisites to $\mathsf{J} = 8$:

$$\forall \mathbf{r} \in \mathbb{R} : |\mathbf{r}_\mathbf{l}| + |(\mathbf{r}_\mathbf{c})_\mathbf{p}| \leq \mathsf{J} \tag{11.3}$$

11.1.2. *Refinement Context.* A *refinement context*, denoted by the set $\mathbb{C}$, 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 accumulation output log super-peak b; and the *lookup-anchor*, header hash l and of timeslot t. Finally, it identifies the hash of any prerequisite work-packages $\mathbf{p}$. Formally:

$$\mathbb{C} \equiv \begin{pmatrix} a \in \mathbb{H}\,,\ s \in \mathbb{H}\,,\quad b \in \mathbb{H}\,, \\ l \in \mathbb{H}\,,\ t \in \mathbb{N}_T\,,\ \mathbf{p} \in \left\{\!\left\{ \mathbb{H} \right\}\!\right\} \end{pmatrix} \tag{11.4}$$

11.1.3. *Availability.* We define the set of *availability specifications*, $\mathbb{Y}$, as the tuple of the work-package's hash p, an auditable work bundle length l (see section 14.4.1 for more clarity on what this is), together with an erasure-root u, a segment-root e and segment-count n. Work-results include this availability specification in order to ensure they are able to correctly reconstruct and audit the purported ramifications of any reported work-package. Formally:

$$\mathbb{Y} \equiv (p \in \mathbb{H}\,,\ l \in \mathbb{N}_L\,,\ u \in \mathbb{H}\,,\ e \in \mathbb{H}\,,\ n \in \mathbb{N}) \tag{11.5}$$

The *erasure-root* (u) is the root of a binary Merkle tree which functions as a commitment to all data required for the auditing of the report and for use by later work-packages should they need to retrieve any data yielded. It is thus used by assurers to verify the correctness of data they have been sent by guarantors, and it is later verified as correct by auditors. It is discussed fully in section 14.

The *segment-root* (e) is the root of a constant-depth, left-biased and zero-hash-padded binary Merkle tree committing to the hashes of each of the exported segments of each work-item. These are used by guarantors to verify the correctness of any reconstructed segments they are

called upon to import for evaluation of some later work-package. It is also discussed in section 14.

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

$$\mathbb{D} \equiv \begin{pmatrix} s \in \mathbb{N}_S\,,\ c \in \mathbb{H}\,,\ y \in \mathbb{H}\,,\ g \in \mathbb{N}_G\,,\ \mathbf{l} \in \mathbb{B} \cup \mathbb{E}\,, \\ u \in \mathbb{N}_G\,,\ i \in \mathbb{N}\,,\ x \in \mathbb{N}\,,\ z \in \mathbb{N}\,,\quad e \in \mathbb{N} \end{pmatrix} \tag{11.6}$$

Work-digests 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 11.42.

Next, the hash of the payload (y) 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 limit g for executing this item's accumulate.

There is the work *result*, the output blob or error of the execution of the code, $\mathbf{l}$, which may be either an octet sequence in case it was successful, or a member of the set $\mathbb{E}$, if not. This latter set is defined as the set of possible errors, formally:

$$\mathbb{E} \in \left\{ \infty, \lightning, \circledcirc, \ominus, \texttt{BAD}, \texttt{BIG} \right\} \tag{11.7}$$

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 four, the first indicates that the number of exports made was invalidly reported, the second that the size of the digest (refinement output) would cross the acceptable limit, the third indicates that the service's code was not available for lookup in state at the posterior state of the lookup-anchor block. The fourth indicates that the code was available but was beyond the maximum size allowed W_C.

Finally, we have five fields describing the level of activity which this workload imposed on the core in bringing the result to bear. We include u the actual amount of gas used during refinement; i and e the number of segments imported from, and exported into, the $\mathrm{D}^3\mathrm{L}$ respectively; and x and z the number of, and total size in octets of, the extrinsics used in computing the workload. See section 14 for more information on the meaning of these values.

In order to ensure fair use of a block's extrinsic space, work-reports are limited in the maximum total size of the successful refinement output blobs together with the authorizer trace, effectively limiting their overall size:

$$\forall \mathbf{r} \in \mathbb{R} : |\mathbf{r_t}| + \sum_{\mathbf{d} \in \mathbf{r_d} \cap \mathbb{B}} |\mathbf{d_l}| \le \mathsf{W}_R \tag{11.8}$$

$$\mathsf{W}_R \equiv 48 \cdot 2^{10} \tag{11.9}$$

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.[12] Formally:

$$\mathbf{E}_A \in \left\llbracket \left(a \in \mathbb{H},\ f \in \mathbb{b}_\mathsf{C},\ v \in \mathbb{N}_\mathsf{V},\ s \in \bar{\mathbb{V}} \right) \right\rrbracket_{:\mathsf{V}} \tag{11.10}$$

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

$$\forall a \in \mathbf{E}_A : a_a = \mathbf{H}_P \tag{11.11}$$

$$\forall i \in \{\,1 \ldots |\mathbf{E}_A|\,\} : \mathbf{E}_A[i-1]_v < \mathbf{E}_A[i]_v \tag{11.12}$$

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:

$$\forall a \in \mathbf{E}_A : a_s \in \bar{\mathbb{V}}_{\kappa[a_v]_e} \langle \mathsf{X}_A \frown \mathcal{H}(\mathcal{E}(\mathbf{H}_P, a_f)) \rangle \tag{11.13}$$

$$\mathsf{X}_A \equiv \texttt{\$jam\_available} \tag{11.14}$$

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

$$\forall a \in \mathbf{E}_A, c \in \mathbb{N}_\mathsf{C} : \quad a_f[c] \Rightarrow \rho^\dagger[c] \neq \varnothing \tag{11.15}$$

11.2.2. *Available Reports.* A work-report is said to become *available* if and only if there are a clear $^2/_3$supermajority of validators who have marked its core as set within the block's assurance extrinsic. Formally, we define the sequence of newly available work-reports $\mathbf{R}$ as:

$$\mathbf{R} \equiv \left[\rho^\dagger[c]_\mathbf{r} \;\middle|\; c \in \mathbb{N}_\mathsf{C},\ \sum_{a \in \mathbf{E}_A} a_f[c] > {}^2/_3\mathsf{V} \right] \tag{11.16}$$

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 either now available or have timed out:

$$\forall c \in \mathbb{N}_\mathsf{C} : \rho^\ddagger[c] \equiv \begin{cases} \varnothing & \text{if } \rho[c]_\mathbf{r} \in \mathbf{R} \vee \mathbf{H}_T \ge \rho^\dagger[c]_t + \mathsf{U} \\ \rho^\dagger[c] & \text{otherwise} \end{cases} \tag{11.17}$$

11.3. **Guarantor Assignments.** Every block, each core has three validators uniquely assigned to guarantee work-reports for it. This is borne out with $\mathsf{V} = 1{,}023$ validators and $\mathsf{C} = 341$ cores, since $\mathsf{V}/\mathsf{C} = 3$. The core index assigned to each of the validators, as well as the validators' keys are denoted by $\mathbf{M}$:

$$\mathbf{M} \in (\llbracket \mathbb{N}_\mathsf{C} \rrbracket_\mathsf{V}, \llbracket \mathbb{K} \rrbracket_\mathsf{V}) \tag{11.18}$$

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 $\mathbf{M}$ as follows:

$$R(\mathbf{c}, n) \equiv [(x + n) \bmod \mathsf{C} \mid x \in \mathbf{c}] \tag{11.19}$$

[12] This is a "soft" implication since there is no consequence on-chain if dishonestly reported. For more information on this implication see section 16.

$$P(e,t) \equiv R\Big(\mathcal{F}\Big(\Big[\Big\lfloor \frac{\mathsf{C}\cdot i}{\mathsf{V}} \Big\rfloor \Big| \, i \in \mathbb{N}_{\mathsf{V}}\Big], e\Big), \Big\lfloor \frac{t \bmod \mathsf{E}}{\mathsf{R}} \Big\rfloor\Big) \tag{11.20}$$

$$\mathbf{M} \equiv \big(P(\eta'_2, \tau'), \Phi(\kappa')\big) \tag{11.21}$$

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

$$\text{let } (e, \mathbf{k}) = \begin{cases} (\eta'_2, \kappa') & \text{if } \left\lfloor \frac{\tau' - \mathsf{R}}{\mathsf{E}} \right\rfloor = \left\lfloor \frac{\tau'}{\mathsf{E}} \right\rfloor \\ (\eta'_3, \lambda') & \text{otherwise} \end{cases} \tag{11.22}$$

$$\mathbf{M}^* \equiv \big(P(e, \tau' - \mathsf{R}), \Phi(\mathbf{k})\big)$$

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 *work-report*, a credential a and its corresponding timeslot t. The core index of each guarantee must be unique and guarantees must be in ascending order of this. Formally:

$$\mathbf{E}_G \in \big\llbracket \big(\mathbf{r} \in \mathbb{R},\, t \in \mathbb{N}_T,\, a \in \big\llbracket (\mathbb{N}_{\mathsf{V}}, \bar{\mathbb{V}}) \big\rrbracket_{2:3}\big) \big\rrbracket_{:\mathsf{C}} \tag{11.23}$$

$$\mathbf{E}_G = [g \in \mathbf{E}_G \,\big\backslash\!\big\backslash\, (g_{\mathbf{r}})_c] \tag{11.24}$$

The credential is a sequence of two or three tuples of a unique validator index and a signature. Credentials must be ordered by their validator index:

$$\forall g \in \mathbf{E}_G : g_a = [(v,s) \in g_a \,\big\backslash\!\big\backslash\, v] \tag{11.25}$$

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 hash of the work-report. The signing validators must be assigned to the core in question in either this block $\mathbf{M}$ 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, $\mathbf{M}^*$:

$$\begin{aligned} &\forall (\mathbf{r}, t, a) \in \mathbf{E}_G, \\ &\quad \forall (v,s) \in a \end{aligned} : \begin{cases} s \in \bar{\mathbb{V}}_{(\mathbf{k}_v)_e}\langle \mathsf{X}_G \frown \mathcal{H}(\mathbf{r}) \rangle \\ \mathbf{c}_v = \mathbf{r}_c \wedge \mathsf{R}(\lfloor \tau'/\mathsf{R} \rfloor - 1) \le t \le \tau' \end{cases}$$

$$k \in \mathbf{G} \Leftrightarrow \exists (\mathbf{r}, t, a) \in \mathbf{E}_G, \exists (v,s) \in a : k = (\mathbf{k}_v)_e \tag{11.26}$$

$$\text{where } (\mathbf{c}, \mathbf{k}) = \begin{cases} \mathbf{M} & \text{if } \left\lfloor \frac{\tau'}{\mathsf{R}} \right\rfloor = \left\lfloor \frac{t}{\mathsf{R}} \right\rfloor \\ \mathbf{M}^* & \text{otherwise} \end{cases}$$

$$\mathsf{X}_G \equiv \texttt{\$jam\_guarantee} \tag{11.27}$$

We note that the Ed25519 key of each validator whose signature is in a credential is placed in the *reporters* set $\mathbf{G}$. This is utilized by the validator activity statistics bookkeeping system section 13.

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

$$\text{let } \mathbf{I} = \{\, g_{\mathbf{r}} \mid g \in \mathbf{E}_G \,\} \tag{11.28}$$

No reports may be placed on cores with a report pending availability on it. A report is valid only if the authorizer hash is present in the authorizer pool of the core on which the work is reported. Formally:

$$\forall \mathbf{r} \in \mathbf{I} : \rho^{\ddagger}[\mathbf{r}_c] = \varnothing \wedge \mathbf{r}_a \in \alpha[\mathbf{r}_c] \tag{11.29}$$

We require that the gas allotted for accumulation of each work-digest in each work-report respects its service's minimum gas requirements. We also require that all work-reports' total allotted accumulation gas is no greater than the overall gas limit G_A:

$$\forall \mathbf{r} \in \mathbf{I} : \sum_{\mathbf{d} \in \mathbf{r_d}} (\mathbf{d}_g) \le \mathsf{G}_A \;\wedge\; \forall \mathbf{d} \in \mathbf{r_d} : \mathbf{d}_g \ge \delta[\mathbf{d}_s]_g \tag{11.30}$$

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:

$$\text{let } \mathbf{x} \equiv \{\, \mathbf{r_c} \mid \mathbf{r} \in \mathbf{I} \,\}, \quad \mathbf{p} \equiv \{\, (\mathbf{r_s})_p \mid \mathbf{r} \in \mathbf{I} \,\} \tag{11.31}$$

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 $\mathbf{I}$:

$$|\mathbf{p}| = |\mathbf{I}| \tag{11.32}$$

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 $\beta^{\dagger}_H$:

$$\forall x \in \mathbf{x} : \exists y \in \beta^{\dagger}_H : x_a = y_h \wedge x_s = y_s \wedge x_b = y_b \tag{11.33}$$

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

$$\forall x \in \mathbf{x} : \; x_t \ge \mathbf{H}_T - \mathsf{L} \tag{11.34}$$

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:

$$\forall x \in \mathbf{x} : \; \exists h \in \mathbf{A} : h_T = x_t \wedge \mathcal{H}(h) = x_l \tag{11.35}$$

We require that the work-package of the report not be the work-package of some other report made in the past. We ensure that the work-package not appear anywhere within our pipeline. Formally:

$$\text{let } \mathbf{q} = \{\, (\mathbf{r_s})_p \mid (\mathbf{r}, \mathbf{d}) \in \widehat{\omega} \,\} \tag{11.36}$$

$$\text{let } \mathbf{a} = \{\, ((\mathbf{r_r})_\mathbf{s})_p \mid \mathbf{r} \in \rho, \mathbf{r} \ne \varnothing \,\} \tag{11.37}$$

$$\forall p \in \mathbf{p}, p \notin \bigcup_{x \in \beta_H} \mathcal{K}(x_\mathbf{p}) \cup \bigcup_{x \in \xi} x \cup \mathbf{q} \cup \mathbf{a} \tag{11.38}$$

We require that the prerequisite work-packages, if present, and any work-packages mentioned in the segment-root lookup, be either in the extrinsic or in our recent history.

$$\begin{aligned} &\forall \mathbf{r} \in \mathbf{I}, \forall p \in (\mathbf{r_c})_\mathbf{p} \cup \mathcal{K}(\mathbf{r_l}) : \\ &\quad p \in \mathbf{p} \cup \{\, x \mid x \in \mathcal{K}(b_\mathbf{p}),\, b \in \beta_H \,\} \end{aligned} \tag{11.39}$$

We require that any segment roots mentioned in the segment-root lookup be verified as correct based on our recent work-package history and the present block:

$$\text{let } \mathbf{p} = \{\, (((g_\mathbf{r})_\mathbf{s})_p \mapsto ((g_\mathbf{r})_\mathbf{s})_e) \mid g \in \mathbf{E}_G \,\} \tag{11.40}$$

$$\forall \mathbf{r} \in \mathbf{I} : \mathbf{r_l} \subseteq \mathbf{p} \cup \bigcup_{b \in \beta_H} b_\mathbf{p} \tag{11.41}$$

(Note that these checks leave open the possibility of accepting work-reports in apparent dependency loops. We do not consider this a problem: the pre-accumulation stage effectively guarantees that accumulation never happens in these cases and the reports are simply ignored.)

Finally, we require that all work-digests within the extrinsic predicted the correct code hash for their corresponding service:

$$\forall \mathbf{r} \in \mathbf{I}, \forall \mathbf{d} \in \mathbf{r_d} : \mathbf{d}_c = \delta[\mathbf{d}_s]_c \tag{11.42}$$

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 to be replaced without respect to its availability once sufficient time has elapsed (see equation 11.29).

(11.43)
$$\forall c \in \mathbb{N}_{\mathsf{C}} : \rho'[c] \equiv \begin{cases} (\mathbf{r},\, t \colon \tau') & \text{if } \exists (\mathbf{r},\, t,\, a) \in \mathbf{E}_G, \mathbf{r}_c = c \\ \rho^{\ddagger}[c] & \text{otherwise} \end{cases}$$

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-digest, passing to it relevant data from said digests together with 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.

12.1. **History and Queuing.** Accumulation of a work-report is deferred in the case that it has a not-yet-fulfilled dependency and is cancelled entirely in the case of an invalid dependency. Dependencies are specified as work-package hashes and in order to know which work-packages have been accumulated already, we maintain a history of what has been accumulated. This history, ξ, is sufficiently large for an epoch worth of work-reports. Formally:

(12.1)
$$\xi \in \left[\{\!\{ \mathbb{H} \}\!\}\right]_{\mathsf{E}}$$

(12.2)
$$\widetilde{\xi} \equiv \bigcup_{x \in \xi} (x)$$

We also maintain knowledge of ready (i.e. available and/or audited) but not-yet-accumulated work-reports in the state item ω. Each of these were made available at most one epoch ago but have or had unfulfilled dependencies. Alongside the work-report itself, we retain its unaccumulated dependencies, a set of work-package hashes. Formally:

(12.3)
$$\omega \in \left[\left[(\mathbb{R}, \{\!\{ \mathbb{H} \}\!\})\right]\right]_{\mathsf{E}}$$

The newly available work-reports, $\mathbf{R}$, are partitioned into two sequences based on the condition of having zero prerequisite work-reports. Those meeting the condition, $\mathbf{R}^!$, are accumulated immediately. Those not, $\mathbf{R}^Q$, are for queued execution. Formally:

(12.4)
$$\mathbf{R}^! \equiv [r \mid r \lessdot \mathbf{R}, |(r_{\mathbf{c}})_{\mathbf{p}}| = 0 \wedge r_{\mathbf{l}} = \{\}]$$

(12.5)
$$\mathbf{R}^Q \equiv E([D(r) \mid r \lessdot \mathbf{R}, |(r_{\mathbf{c}})_{\mathbf{p}}| > 0 \vee r_{\mathbf{l}} \neq \{\}],\, \widetilde{\xi}\,)$$

(12.6)
$$D(r) \equiv (r, \{\, (r_{\mathbf{c}})_{\mathbf{p}} \,\} \cup \mathcal{K}(r_{\mathbf{l}}))$$

We define the queue-editing function E, which is essentially a mutator function for items such as those of ω, parameterized by sets of now-accumulated work-package hashes (those in ξ). It is used to update queues of work-reports when some of them are accumulated. Functionally, it removes all entries whose work-report's hash is in the set provided as a parameter, and removes any dependencies which appear in said set. Formally:

(12.7)
$$E \colon \begin{cases} \left(\left[\left(\mathbb{R}, \{\!\{ \mathbb{H} \}\!\}\right)\right], \{\!\{ \mathbb{H} \}\!\}\right) \to \left[\left(\mathbb{R}, \{\!\{ \mathbb{H} \}\!\}\right)\right] \\ (\mathbf{r}, \mathbf{x}) \mapsto \left[(r, \mathbf{d} \setminus \mathbf{x}) \;\middle|\; \begin{array}{l} (r, \mathbf{d}) \lessdot \mathbf{r}, \\ (r_{\mathbf{s}})_p \notin \mathbf{x} \end{array} \right] \end{cases}$$

We further define the accumulation priority queue function Q, which provides the sequence of work-reports which are able to be accumulated given a set of not-yet-accumulated work-reports and their dependencies.

(12.8)
$$Q \colon \begin{cases} \left[\left(\mathbb{R}, \{\!\{ \mathbb{H} \}\!\}\right)\right] \to \left[\mathbb{R}\right] \\ \mathbf{r} \mapsto \begin{cases} [] & \text{if } \mathbf{g} = [] \\ \mathbf{g} \frown Q(E(\mathbf{r}, P(\mathbf{g}))) & \text{otherwise} \\ \quad \text{where } \mathbf{g} = [r \mid (r, \{\}) \lessdot \mathbf{r}] \end{cases} \end{cases}$$

Finally, we define the mapping function P which extracts the corresponding work-package hashes from a set of work-reports:

(12.9)
$$P \colon \begin{cases} \{\!\{ \mathbb{R} \}\!\} \to \{\!\{ \mathbb{H} \}\!\} \\ \quad \mathbf{r} \mapsto \{\, (r_{\mathbf{s}})_p \mid r \in \mathbf{r} \,\} \end{cases}$$

We may now define the sequence of accumulatable work-reports in this block as $\mathbf{R}^*$:

(12.10)
$$\text{let } m = \mathbf{H}_T \bmod \mathsf{E}$$

(12.11)
$$\mathbf{R}^* \equiv \mathbf{R}^! \frown Q(\mathbf{q})$$

(12.12)
$$\text{where } \mathbf{q} = E(\widetilde{\omega_{m\ldots}} \frown \widetilde{\omega_{\ldots m}} \frown \mathbf{R}^Q, P(\mathbf{R}^!))$$

12.2. **Execution.** We work with a limited amount of gas per block and therefore may not be able to process all items in $\mathbf{R}^*$ in a single block. There are two slightly antagonistic factors allowing us to optimize the amount of work-items, and thus work-reports, accumulated in a single block:

Firstly, while we have a well-known gas-limit for each work-item to be accumulated, accumulation may still result in a lower amount of gas used. Only after a work-item is accumulated can it be known if it uses less gas than the advertised limit. This implies a sequential execution pattern.

Secondly, since PVM setup cannot be expected to be zero-cost, we wish to amortize this cost over as many work-items as possible. This can be done by aggregating work-items associated with the same service into the same PVM invocation. This implies a non-sequential execution pattern.

We resolve this by defining a function Δ_+ which accumulates work-reports sequentially, and which itself utilizes a function Δ_* which accumulates work-reports in a non-sequential, service-aggregated manner. In all but the first invocation of Δ_+, we also integrate the effects of any *deferred-transfers* implied by the previous round of accumulation, thus the accumulation function must accept both the information contained in work-digests and that of deferred-transfers.

Rather than passing whole work-digests into accumulate, we extract the salient information from them and combine with information implied by their work-reports. We call this kind of combined value an *operand tuple*, $\mathbb{U}$. Likewise, we denote the set characterizing a *deferred transfer* as $\mathbb{X}$, noting that a transfer includes a memo component m of $\mathsf{W}_T = 128$ octets, together with the service index of the sender s, the service index of the receiver d, the balance to be transferred a and the gas limit g for the transfer. Formally:

$$\mathbb{U} \equiv \begin{pmatrix} p \in \mathbb{H},\ \ e \in \mathbb{H}, a \in \mathbb{H},\ y \in \mathbb{H}, \\ g \in \mathbb{N}_G, \mathbf{t} \in \mathbb{B},\ \mathbf{l} \in \mathbb{B} \cup \mathbb{E} \end{pmatrix} \tag{12.13}$$

$$\mathbb{X} \equiv (s \in \mathbb{N}_S, d \in \mathbb{N}_S, a \in \mathbb{N}_B, m \in \mathbb{B}_{\mathsf{W}_T}, g \in \mathbb{N}_G) \tag{12.14}$$

$$\mathbb{I} \equiv \mathbb{U} \cup \mathbb{X} \tag{12.15}$$

Note that the union of the two is the *accumulation input*, $\mathbb{I}$.

Our formalisms continue by defining $\mathbb{S}$ as a characterization of (i.e. values capable of representing) state components which are both needed and mutable by the accumulation process. This comprises the service accounts state (as in δ), the upcoming validator keys ι, the queue of authorizers ϕ and the privileges state χ. Formally:

$$\mathbb{S} \equiv \begin{pmatrix} \mathbf{d} \in \langle \mathbb{N}_S \to \mathbb{A} \rangle,\ \mathbf{i} \in \llbracket \mathbb{K} \rrbracket_{\mathsf{V}},\ \mathbf{q} \in \llbracket \llbracket \mathbb{H} \rrbracket_{\mathsf{Q}} \rrbracket_{\mathsf{C}},\ m \in \mathbb{N}_S, \\ \mathbf{a} \in \llbracket \mathbb{N}_S \rrbracket_{\mathsf{C}},\ v \in \mathbb{N}_S,\ r \in \mathbb{N}_S,\ \mathbf{z} \in \langle \mathbb{N}_S \to \mathbb{N}_G \rangle \end{pmatrix} \tag{12.16}$$

Finally, we define B and U, the sets characterizing service-indexed commitments to accumulation output and service-indexed gas usage respectively:

$$B \equiv \{\!\{ (\mathbb{N}_S, \mathbb{H}) \}\!\} \qquad U \equiv \llbracket (\mathbb{N}_S, \mathbb{N}_G) \rrbracket \tag{12.17}$$

We define the outer accumulation function Δ_+ which transforms a gas-limit, a sequence of deferred transfers, a sequence of work-reports, an initial partial-state and a dictionary of services enjoying free accumulation, into a tuple of the number of work-reports accumulated, a posterior state-context, the resultant accumulation-output pairings and the service-indexed gas usage:

$$\Delta_+ \colon \begin{cases} (\mathbb{N}_G, \llbracket \mathbb{X} \rrbracket, \llbracket \mathbb{R} \rrbracket, \mathbb{S}, \langle \mathbb{N}_S \to \mathbb{N}_G \rangle) \to (\mathbb{N}, \mathbb{S}, B, U) \\ (g, \mathbf{t}, \mathbf{r}, \mathbf{e}, \mathbf{f}) \mapsto \begin{cases} (0, \mathbf{e}, \{\}, []) & \text{if } n = 0 \\ (i + j, \mathbf{e}', \mathbf{b}^* \cup \mathbf{b}, \mathbf{u}^* \frown \mathbf{u}) & \text{o/w} \end{cases} \\ \quad \text{where } i = \max(\mathbb{N}_{|\mathbf{r}|+1} : \sum_{r \in \mathbf{r}_{\ldots i}, d \in r_{\mathbf{d}}} (d_g) \le g \\ \quad \text{and } n = |\mathbf{t}| + i + |\mathbf{f}| \\ \quad \text{and } (\mathbf{e}^*, \mathbf{t}^*, \mathbf{b}^*, \mathbf{u}^*) = \Delta_*(\mathbf{e}, \mathbf{t}, \mathbf{r}_{\ldots i}, \mathbf{f}) \\ \quad \text{and } (j, \mathbf{e}', \mathbf{b}, \mathbf{u}) = \Delta_+(g^* - \sum_{(s,u) \in \mathbf{u}^*}(u), \mathbf{t}^*, \mathbf{r}_{i \ldots}, \mathbf{e}^*, \{\}) \\ \quad \text{and } g^* = g + \sum_{t \in \mathbf{t}}(t_g) \end{cases} \tag{12.18}$$

We come to define the parallelized accumulation function Δ_* which, with the help of the single-service accumulation function Δ_1, transforms an initial state-context, together with a sequence of deferred transfers, a sequence of work-reports and a dictionary of privileged always-accumulate services, into a tuple of the posterior state-context, the resultant deferred-transfers and accumulation-output pairings, and the service-indexed gas usage. Note that for the privileges we employ a function R which selects the service to which the manager service changed, or if no change was made, then that which the service itself changed to. This allows privileges to be 'owned' and facilitates the removal of the manager service which we see as a helpful possibility. Formally:

$$\Delta_* \colon \begin{cases} (\mathbb{S}, \llbracket \mathbb{X} \rrbracket, \llbracket \mathbb{R} \rrbracket, \langle \mathbb{N}_S \to \mathbb{N}_G \rangle) \to (\mathbb{S}, \llbracket \mathbb{X} \rrbracket, B, U) \\ (\mathbf{e}, \mathbf{t}, \mathbf{r}, \mathbf{f}) \mapsto \left((\mathbf{d}', \mathbf{i}', \mathbf{q}', m', \mathbf{a}', v', r', \mathbf{z}'), \widehat{\mathbf{t}'}, \mathbf{b}, \mathbf{u} \right) \\ \text{where:} \\ \qquad \text{let } \mathbf{s} = \{ d_s \mid r \in \mathbf{r}, d \in r_{\mathbf{d}} \} \cup \mathcal{K}(\mathbf{f}) \cup \{ t_d \mid t \in \mathbf{t} \} \\ \qquad \Delta(s) \equiv \Delta_1(\mathbf{e}, \mathbf{t}, \mathbf{r}, \mathbf{f}, s) \\ \qquad\quad \mathbf{u} = [(s, \Delta(s)_u) \mid s \lhd \mathbf{s}] \\ \qquad\quad \mathbf{b} = \{ (s, b) \mid s \in \mathbf{s}, b = \Delta(s)_y, b \neq \varnothing \} \\ \qquad\quad \mathbf{t}' = [\Delta(s)_{\mathbf{t}} \mid s \lhd \mathbf{s}] \\ \qquad\quad \mathbf{d}' = I((\mathbf{d} \cup \mathbf{n}) \setminus \mathbf{m}, \bigcup_{s \in \mathbf{s}} \Delta(s)_{\mathbf{p}}) \\ \qquad\quad (\mathbf{d}, \mathbf{i}, \mathbf{q}, m, \mathbf{a}, v, r, \mathbf{z}) = \mathbf{e} \\ \qquad\quad \mathbf{e}^* = \Delta(m)_{\mathbf{e}} \\ \qquad (m', \mathbf{z}') = \mathbf{e}^*_{(m, \mathbf{z})} \\ \forall c \in \mathbb{N}_{\mathsf{C}} : \mathbf{a}'_c = R(\mathbf{a}_c, (\mathbf{e}^*_{\mathbf{a}})_c, ((\Delta(\mathbf{a}_c)_{\mathbf{e}})_{\mathbf{a}})_c) \\ \qquad\quad v' = R(v, \mathbf{e}^*_v, (\Delta(v)_{\mathbf{e}})_v) \\ \qquad\quad r' = R(r, \mathbf{e}^*_r, (\Delta(r)_{\mathbf{e}})_r) \\ \qquad\quad \mathbf{i}' = (\Delta(v)_{\mathbf{e}})_{\mathbf{i}} \\ \forall c \in \mathbb{N}_{\mathsf{C}} : \mathbf{q}'_c = ((\Delta(\mathbf{a}_c)_{\mathbf{e}})_{\mathbf{q}})_c \\ \qquad\quad \mathbf{n} = \bigcup_{s \in \mathbf{s}} ((\Delta(s)_{\mathbf{e}})_{\mathbf{d}} \setminus \mathcal{K}(\mathbf{d} \setminus \{ s \})) \\ \qquad\quad \mathbf{m} = \bigcup_{s \in \mathbf{s}} (\mathcal{K}(\mathbf{d}) \setminus \mathcal{K}((\Delta(s)_{\mathbf{e}})_{\mathbf{d}})) \end{cases} \tag{12.19}$$

$$R(o, a, b) \equiv \begin{cases} b & \text{if } a = o \\ a & \text{otherwise} \end{cases} \tag{12.20}$$

And I is the preimage integration function, which transforms a dictionary of service states and a set of service/blob pairs into a new dictionary of service states. Preimage provisions into services which no longer exist or whose relevant request is dropped are disregarded:

$$I \colon \begin{cases} (\langle \mathbb{N}_S \to \mathbb{A} \rangle, \{\!\{ (\mathbb{N}_S, \mathbb{B}) \}\!\}) \to \langle \mathbb{N}_S \to \mathbb{A} \rangle \\ (\mathbf{d}, \mathbf{p}) \mapsto \mathbf{d}' \text{ where } \mathbf{d}' = \mathbf{d} \text{ except:} \\ \quad \forall (s, \mathbf{i}) \in \mathbf{p},\ Y(\mathbf{d}, s, \mathbf{i}) : \\ \qquad \mathbf{d}'[s]_{\mathbf{l}}[(\mathcal{H}(\mathbf{i}), |\mathbf{i}|)] = [\tau'] \\ \qquad \mathbf{d}'[s]_{\mathbf{p}}[\mathcal{H}(\mathbf{i})] = \mathbf{i} \end{cases} \tag{12.21}$$

$$Y \colon \begin{cases} (\langle \mathbb{N}_S \to \mathbb{A} \rangle, \mathbb{N}_S, \mathbb{B}) \to \{ \bot, \top \} \\ (\mathbf{d}, s, \mathbf{i}) \mapsto \begin{cases} \mathbf{d}[s]_{\mathbf{l}}[(\mathcal{H}(\mathbf{i}), |\mathbf{i}|)] = [] & \text{if } s \in \mathcal{K}(\mathbf{d}) \\ \bot & \text{otherwise} \end{cases} \end{cases} \tag{12.22}$$

We note that while forming the union of all altered, newly added service and newly removed indices, defined in the above context as $\mathcal{K}(\mathbf{n}) \cup \mathbf{m}$, different services may not each contribute the same index for a new, altered or

removed service. This cannot happen for the set of removed and altered services since the code hash of removable services has no known preimage and thus cannot execute itself to make an alteration. For new services this should also never happen since new indices are explicitly selected to avoid such conflicts. In the unlikely event it does happen, the block must be considered invalid.

The single-service accumulation function, Δ_1, transforms an initial state-context, a sequence of deferred-transfers, a sequence of work-reports, a dictionary of services enjoying free accumulation (with the values indicating the amount of free gas) and a service index into an alterations state-context, a sequence of *transfers*, a possible accumulation-output, the actual PVM gas used and a set of preimage provisions. This function wrangles the work-digests of a particular service from a set of work-reports and invokes PVM execution with said data:

$$\mathbb{O} \equiv \begin{pmatrix} \mathbf{e} \in \mathbb{S}, & \mathbf{t} \in \llbracket \mathbb{X} \rrbracket,\ y \in \mathbb{H}? , \\ u \in \mathbb{N}_G,\ \mathbf{p} \in \{\!\{ (\mathbb{N}_S, \mathbb{B}) \}\!\} \end{pmatrix} \tag{12.23}$$

(12.24)

$$\Delta_1 \colon \begin{cases} \begin{pmatrix} \mathbb{S}, \llbracket \mathbb{X} \rrbracket, \llbracket \mathbb{R} \rrbracket, \\ \langle \mathbb{N}_S \to \mathbb{N}_G \rangle, \mathbb{N}_S \end{pmatrix} \to \mathbb{O} \\ \qquad (\mathbf{e}, \mathbf{t}, \mathbf{r}, \mathbf{f}, s) \mapsto \Psi_A(\mathbf{e}, \tau', s, g, \mathbf{i}^{T} \frown \mathbf{i}^{U}) \\ \text{where:} \\ \quad g = \mathcal{U}(\mathbf{f}_s, 0) + \sum_{t \in \mathbf{t}, t_d = s} (t_g) + \sum_{r \in \mathbf{r}, d \in r_{\mathbf{d}}, d_s = s} (d_g) \\ \quad \mathbf{i}^{T} = [t \mid t \lhd \mathbf{t}, t_d = s] \\ \quad \mathbf{i}^{U} = \left[\begin{pmatrix} \mathbf{l} \colon d_{\mathbf{l}},\ g \colon d_g,\ y \colon d_y, \mathbf{t} : r_{\mathbf{t}}, \\ e \colon (r_{\mathbf{s}})_e,\ p \colon (r_{\mathbf{s}})_p, a \colon r_a \end{pmatrix} \middle| \begin{matrix} r \lhd \mathbf{r}, \\ d \lhd r_{\mathbf{d}},\ d_s = s \end{matrix} \right] \end{cases}$$

This draws upon g, the gas limit implied by the selected deferred-transfers, work-reports and gas-privileges.

12.3. **Final State Integration.** Given the result of the top-level Δ_+, we may define the posterior state χ', ϕ' and ι' as well as the first intermediate state of the service-accounts $\delta^\dagger$ and the Accumulation Output Log θ':

$$\text{let } g = \max\left(\mathsf{G}_T, \mathsf{G}_A \cdot \mathsf{C} + \textstyle\sum_{x \in \mathcal{V}(\chi_Z)}(x)\right)$$

$$\text{and } \mathbf{e} = (\mathbf{d} \colon \delta, \mathbf{i} \colon \iota, \mathbf{q} \colon \phi, m \colon \chi_M, \mathbf{a} \colon \chi_A, v \colon \chi_V, r \colon \chi_R, \mathbf{z} \colon \chi_Z)$$

(12.25)

$$\left(n, \mathbf{e}', \mathbf{b}, \mathbf{u}\right) \equiv \Delta_+(g, [], \mathbf{R}^*, \mathbf{e}, \chi_Z)$$

(12.26)

$$\theta' \equiv [(s, h) \in \mathbf{b}]$$

(12.27)

$$\left(\mathbf{d} \colon \delta^\dagger, \mathbf{i} \colon \iota', \mathbf{q} \colon \phi', m \colon \chi'_M, \mathbf{a} \colon \chi'_A, v \colon \chi'_V, r \colon \chi'_R, \mathbf{z} \colon \chi'_Z\right) \equiv \mathbf{e}'$$

From this formulation, we also receive n, the total number of work-reports accumulated and $\mathbf{u}$, the gas used in the accumulation process for each service. We compose $\mathbf{S}$, our accumulation statistics, which is a mapping from the service indices which were accumulated to the amount of gas used throughout accumulation and the number of work-items accumulated. Formally:

$$\mathbf{S} \in \langle \mathbb{N}_S \to (\mathbb{N}_G, \mathbb{N}) \rangle \tag{12.28}$$

$$\mathbf{S} \equiv \{\, (s \mapsto (G(s), N(s))) \mid G(s) + N(s) \neq 0 \,\} \tag{12.29}$$

$$\text{where } G(s) \equiv \sum_{(s,u) \in \mathbf{u}} (u)$$

$$\text{and } N(s) \equiv \left|\left[d \,\middle|\, r \lhd \mathbf{R}^*_{\cdots n}, d \lhd r_{\mathbf{d}}, d_s = s \right]\right|$$

The second intermediate state $\delta^\ddagger$ may then be defined with the last-accumulation record being updated for all accumulated services:

$$\delta^\ddagger \equiv \left\{ \left(s \mapsto a'\right) \,\middle|\, (s \mapsto a) \in \delta^\dagger \right\} \tag{12.30}$$

$$\text{where } a' = \begin{cases} a \text{ except } a'_a = \tau' & \text{if } s \in \mathcal{K}(\mathbf{S}) \\ a & \text{otherwise} \end{cases} \tag{12.31}$$

We define the final state of the ready queue and the accumulated map by integrating those work-reports which were accumulated in this block and shifting any from the prior state with the oldest such items being dropped entirely:

$$\xi'_{\mathsf{E}-1} = P(\mathbf{R}^*_{\cdots n}) \tag{12.32}$$

$$\forall i \in \mathbb{N}_{\mathsf{E}-1} : \xi'_i \equiv \xi_{i+1} \tag{12.33}$$

$$\forall i \in \mathbb{N}_{\mathsf{E}} : \omega'^{\circlearrowleft}_{m-i} \equiv \begin{cases} E(\mathbf{R}^Q, \xi'_{\mathsf{E}-1}) & \text{if } i = 0 \\ [] & \text{if } 1 \leq i < \tau' - \tau \\ E(\omega^{\circlearrowleft}_{m-i}, \xi'_{\mathsf{E}-1}) & \text{if } i \geq \tau' - \tau \end{cases} \tag{12.34}$$

12.4. **Preimage Integration.** After accumulation, we must integrate all preimages provided in the lookup extrinsic to arrive at the posterior account state. The lookup extrinsic is a sequence of pairs of service indices and data. These pairs must be ordered and without duplicates (equation 12.36 requires this). The data must have been solicited by a service but not yet provided in the *prior* state. Formally:

$$\mathbf{E}_P \in \llbracket (\mathbb{N}_S, \mathbb{B}) \rrbracket \tag{12.35}$$

$$\mathbf{E}_P = [i \in \mathbf{E}_P \,\rangle\!\rangle\, i] \tag{12.36}$$

$$\forall (s, \mathbf{d}) \in \mathbf{E}_P : Y(\delta, s, \mathbf{d}) \tag{12.37}$$

We disregard, without prejudice, any preimages which due to the effects of accumulation are no longer useful. We define δ' as the state after the integration of the still-relevant preimages:

$$\delta' = I(\delta^\ddagger, \mathbf{E}_P) \tag{12.38}$$

13. Statistics

13.1. **Validator Activity.** The JAM chain does not explicitly issue rewards—we leave this as a job to be done by the staking subsystem (in Polkadot's case envisioned as 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 information on validator activity 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.

The validator statistics are made on a per-epoch basis and we retain one record of completed statistics together with one record which serves as an accumulator for the present epoch. Both are tracked in π, which is thus a

sequence of two elements, with the first being the accumulator and the second the previous epoch's statistics. For each epoch we track a performance record for each validator:

$$\pi \equiv (\pi_V, \pi_L, \pi_C, \pi_S) \tag{13.1}$$

$$(\pi_V, \pi_L) \in \left[\!\left[(b \in \mathbb{N}, t \in \mathbb{N}, p \in \mathbb{N}, d \in \mathbb{N}, g \in \mathbb{N}, a \in \mathbb{N}) \right]\!\right]^2_{\mathsf{V}} \tag{13.2}$$

The six validator statistics we track are:

- b: The number of blocks produced by the validator.
- t: The number of tickets introduced by the validator.
- p: The number of preimages introduced by the validator.
- d: The total number of octets across all preimages introduced by the validator.
- g: The number of reports guaranteed by the validator.
- a: The number of availability assurances made by the validator.

The objective statistics are updated in line with their description, formally:

$$\text{let } e = \left\lfloor \frac{\tau}{\mathsf{E}} \right\rfloor, \quad e' = \left\lfloor \frac{\tau'}{\mathsf{E}} \right\rfloor \tag{13.3}$$

$$\left(\mathbf{a}, \pi'_L\right) \equiv \begin{cases} (\pi_V, \pi_L) & \text{if } e' = e \\ ([[0, \dots, [0, \dots]), \dots], \pi_V) & \text{otherwise} \end{cases} \tag{13.4}$$

$$\forall v \in \mathbb{N}_{\mathsf{V}} : \begin{cases} \pi'_V[v]_b \equiv \mathbf{a}[v]_b + (v = \mathbf{H}_I) \\ \pi'_V[v]_t \equiv \mathbf{a}[v]_t + \begin{cases} |\mathbf{E}_T| & \text{if } v = \mathbf{H}_I \\ 0 & \text{otherwise} \end{cases} \\ \pi'_V[v]_p \equiv \mathbf{a}[v]_p + \begin{cases} |\mathbf{E}_P| & \text{if } v = \mathbf{H}_I \\ 0 & \text{otherwise} \end{cases} \\ \pi'_V[v]_d \equiv \mathbf{a}[v]_d + \begin{cases} \sum_{d \in \mathbf{E}_P} |d| & \text{if } v = \mathbf{H}_I \\ 0 & \text{otherwise} \end{cases} \\ \pi'_V[v]_g \equiv \mathbf{a}[v]_g + (\kappa'_v \in \mathbf{G}) \\ \pi'_V[v]_a \equiv \mathbf{a}[v]_a + (\exists a \in \mathbf{E}_A : a_v = v) \end{cases} \tag{13.5}$$

Note that $\mathbf{G}$ is the *Reporters* set, as defined in equation 11.26.

13.2. **Cores and Services.** The other two components of statistics are the core and service activity statistics. These are tracked only on a per-block basis unlike the validator statistics which are tracked over the whole epoch.

$$\pi_C \in \left[\!\left[\begin{pmatrix} d \in \mathbb{N}, p \in \mathbb{N}, i \in \mathbb{N}, x \in \mathbb{N}, \\ z \in \mathbb{N}, e \in \mathbb{N}, l \in \mathbb{N}, u \in \mathbb{N}_G \end{pmatrix} \right]\!\right]_{\mathsf{C}} \tag{13.6}$$

$$\pi_S \in \left\langle \mathbb{N}_S \to \begin{pmatrix} p \in (\mathbb{N}, \mathbb{N}), \quad r \in (\mathbb{N}, \mathbb{N}_G), \\ i \in \mathbb{N}, x \in \mathbb{N}, z \in \mathbb{N}, e \in \mathbb{N}, \\ a \in (\mathbb{N}, \mathbb{N}_G) \end{pmatrix} \right\rangle \tag{13.7}$$

The core statistics are updated using several intermediate values from across the overall state-transition function; $\mathbf{I}$, the incoming work-reports, as defined in 11.28 and $\mathbf{R}$, the newly available work-reports, as defined in 11.16. We define the statistics as follows:

$$\forall c \in \mathbb{N}_{\mathsf{C}} : \pi'_C[c] \equiv \begin{pmatrix} i\colon R(c)_i, \ x\colon R(c)_x, \ z\colon R(c)_z, \\ e\colon R(c)_e, \ u\colon R(c)_u, \ l\colon L(c), \\ d\colon D(c), \ \ p\colon \sum_{a \in \mathbf{E}_A} a_f[c] \end{pmatrix} \tag{13.8}$$

$$\text{where } R(c \in \mathbb{N}_{\mathsf{C}}) \equiv \sum_{\mathbf{d} \in \mathbf{r_d}, \mathbf{r} \in \mathbf{I}, \mathbf{r}_c = c} (\mathbf{d}_i, \mathbf{d}_x, \mathbf{d}_z, \mathbf{d}_e, \mathbf{d}_u,) \tag{13.9}$$

$$\text{and } L(c \in \mathbb{N}_{\mathsf{C}}) \equiv \sum_{\mathbf{r} \in \mathbf{I}, \mathbf{r}_c = c} (\mathbf{r_s})_l \tag{13.10}$$

$$\text{and } D(c \in \mathbb{N}_{\mathsf{C}}) \equiv \sum_{\mathbf{r} \in \mathbf{R}, \mathbf{r}_c = c} (\mathbf{r_s})_l + \mathsf{W}_G \lceil (\mathbf{r_s})_n {}^{65}\!/_{64} \rceil \tag{13.11}$$

Finally, the service statistics are updated using the same intermediate values as the core statistics, but with a different set of calculations:

$$\forall s \in \mathbf{s} : \pi'_S[s] \equiv \begin{pmatrix} i\colon R(s)_i, \ x\colon R(s)_x, \ z\colon R(s)_z, \\ e\colon R(s)_e, \ r\colon (R(s)_n, R(s)_u), \\ p\colon \sum_{(s, \mathbf{d}) \in \mathbf{E}_P} (1, |\mathbf{d}|), \\ a\colon \quad \mathcal{U}(\mathbf{S}[s], (0, 0)) \end{pmatrix} \tag{13.12}$$

$$\text{where } \mathbf{s} = \mathbf{s}^R \cup \mathbf{s}^P \cup \mathcal{K}(\mathbf{S}) \tag{13.13}$$

$$\text{and } \mathbf{s}^R = \{\, \mathbf{d}_s \mid \mathbf{d} \in \mathbf{r_d}, \mathbf{r} \in \mathbf{I} \,\} \tag{13.14}$$

$$\text{and } \mathbf{s}^P = \{\, s \mid \exists x : (s, x) \in \mathbf{E}_P \,\} \tag{13.15}$$

$$\text{and } R(s \in \mathbb{N}_S) \equiv \sum_{\mathbf{d} \in \mathbf{r_d}, \mathbf{r} \in \mathbf{I}, \mathbf{d}_s = s} (n\colon 1, \mathbf{d}_u, \mathbf{d}_i, \mathbf{d}_x, \mathbf{d}_z, \mathbf{d}_e) \tag{13.16}$$

14. Work Packages and Work Reports

14.1. **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 15;
- assuring the availability of work-packages after being in receipt of their data;
- determining which work-reports to audit, fetching and auditing them, and creating and distributing judgments appropriately based on the outcome of the audit;
- submitting the correct amount of auditing work seen being done by other validators, discussed in section 13.

14.2. **Segments and the Manifest.** Our basic erasure-coding segment size is $\mathsf{W}_E = 684$ octets, derived from the fact we wish to be able to reconstruct even should almost two-thirds of our 1023 participants be malicious or incapacitated, the 16-bit Galois field on which the erasure-code

is based and the desire to efficiently support encoding data of close to, but no less than, 4KB.

Work-packages are generally small to ensure guarantors need not invest a lot of bandwidth in order to discover whether they can get paid for their evaluation into a work-report. Rather than having much data inline, they instead *reference* data through commitments. The simplest commitments are extrinsic data.

Extrinsic data are blobs which are being introduced into the system alongside the work-package itself generally by the work-package builder. They are exposed to the Refine logic as an argument. We commit to them through including each of their hashes in the work-package.

Work-packages have two other types of external data associated with them: A cryptographic commitment to each *imported* segment and finally the number of segments which are *exported*.

14.2.1. *Segments, Imports and Exports.* The ability to communicate large amounts of data from one work-package to some subsequent work-package is a key feature of the JAM availability system. An export segment, defined as the set $\mathbb{J}$, is an octet sequence of fixed length $\mathsf{W}_G = 4104$. It is the smallest datum which may individually be imported from—or exported to—the long-term D^{3}L during the Refine function of a work-package. Being an exact multiple of the erasure-coding piece size ensures that the data segments of work-package can be efficiently placed in the D^{3}L system.

$$\mathbb{J} \equiv \mathbb{B}_{\mathsf{W}_G} \tag{14.1}$$

Exported segments are data which are *generated* through the execution of the Refine logic and thus are a side effect of transforming the work-package into a work-report. Since their data is deterministic based on the execution of the Refine logic, we do not require any particular commitment to them in the work-package beyond knowing how many are associated with each Refine invocation in order that we can supply an exact index.

On the other hand, imported segments are segments which were exported by previous work-packages. In order for them to be easily fetched and verified they are referenced not by hash but rather the root of a Merkle tree which includes any other segments introduced at the time, together with an index into this sequence. This allows for justifications of correctness to be generated, stored, included alongside the fetched data and verified. This is described in depth in the next section.

14.2.2. *Data Collection and Justification.* It is the task of a guarantor to reconstitute all imported segments through fetching said segments' erasure-coded chunks from enough unique validators. Reconstitution alone is not enough since corruption of the data would occur if one or more validators provided an incorrect chunk. For this reason we ensure that the import segment specification (a Merkle root and an index into the tree) be a kind of cryptographic commitment capable of having a justification applied to demonstrate that any particular segment is indeed correct.

Justification data must be available to any node over the course of its segment's potential requirement. At around 350 bytes to justify a single segment, justification data is too voluminous to have all validators store all data. We therefore use the same overall availability framework for hosting justification metadata as the data itself.

The guarantor is able to use this proof to justify to themselves that they are not wasting their time on incorrect behavior. We do not force auditors to go through the same process. Instead, guarantors build an *Auditable Work Package*, and place this in the Audit DA system. This is the original work-package, its extrinsic data, its imported data and a concise proof of correctness of that imported data. This tactic routinely duplicates data between the D^{3}L and the Audit DA, however it is acceptable in order to reduce the bandwidth cost for auditors who must justify the correctness as cheaply as possible as auditing happens on average 30 times for each work-package whereas guaranteeing happens only twice or thrice.

14.3. **Packages and Items.** We begin by defining a *work-package*, of set $\mathbb{P}$, and its constituent *work-items*, of set $\mathbb{W}$. A work-package includes a simple blob acting as an authorization token $\mathbf{j}$, the index of the service which hosts the authorization code h, an authorization code hash u and a configuration blob $\mathbf{f}$, a context $\mathbf{c}$ and a sequence of work items $\mathbf{w}$:

$$\mathbb{P} \equiv (\mathbf{j} \in \mathbb{B},\ h \in \mathbb{N}_S,\ u \in \mathbb{H},\ \mathbf{f} \in \mathbb{B},\ \mathbf{c} \in \mathbb{C},\ \mathbf{w} \in \llbracket \mathbb{W} \rrbracket_{1:\mathbb{I}}) \tag{14.2}$$

A work item includes: s the identifier of the service to which it relates, the code hash of the service at the time of reporting c (whose preimage must be available from the perspective of the lookup anchor block), a payload blob $\mathbf{y}$, gas limits for Refinement and Accumulation g & a, and the three elements of its manifest, a sequence of imported data segments $\mathbf{i}$ which identify a prior exported segment through an index and the identity of an exporting work-package, $\mathbf{x}$, a sequence of blob hashes and lengths to be introduced in this block (and which we assume the validator knows) and e the number of data segments exported by this work item.

$$\mathbb{W} \equiv \begin{pmatrix} s \in \mathbb{N}_S, c \in \mathbb{H}, \mathbf{y} \in \mathbb{B}, g \in \mathbb{N}_G, a \in \mathbb{N}_G, e \in \mathbb{N}, \\ \mathbf{i} \in \llbracket \left(\mathbb{H} \cup (\mathbb{H}^{\boxplus}), \mathbb{N}\right) \rrbracket, \mathbf{x} \in \llbracket (\mathbb{H}, \mathbb{N}) \rrbracket \end{pmatrix} \tag{14.3}$$

Note that an imported data segment's work-package is identified through the union of sets $\mathbb{H}$ and a tagged variant $\mathbb{H}^{\boxplus}$. A value drawn from the regular $\mathbb{H}$ implies the hash value is of the segment-root containing the export, whereas a value drawn from $\mathbb{H}^{\boxplus}$ implies the hash value is the hash of the exporting work-package. In the latter case it must be converted into a segment-root by the guarantor and this conversion reported in the work-report for on-chain validation.

We limit the total number of exported items to $\mathsf{W}_X = 3072$, the total number of imported items to $\mathsf{W}_M = 3072$, and the total number of extrinsics to $\mathsf{T} = 128$:

$$\forall \mathbf{p} \in \mathbb{P} : \sum_{\mathbf{w} \in \mathbf{p_w}} \mathbf{w}_e \le \mathsf{W}_X \ \wedge \sum_{\mathbf{w} \in \mathbf{p_w}} |\mathbf{w_i}| \le \mathsf{W}_M \ \wedge \sum_{\mathbf{w} \in \mathbf{p_w}} |\mathbf{w_x}| \le \mathsf{T} \tag{14.4}$$

We make an assumption that the preimage to each extrinsic hash in each work-item is known by the guarantor. In general this data will be passed to the guarantor alongside the work-package.

We limit the total size of the auditable *work-bundle*, containing the work-package, import and extrinsic items, together with all payloads, the authorizer configuration

and the authorization token to around 13.6MB. This limit allows 2MB/s/core D^{3}L imports, and thus a full complement of 3,072 imports, assuming no extrinsics, 64 bytes for each of the authorization token and trace, and a work-item payload of 4KB:

$$\forall \mathbf{p} \in \mathbb{P} : \left(|\mathbf{p_j}| + |\mathbf{p_f}| + \sum_{\mathbf{w} \in \mathbf{p_w}} S(\mathbf{w}) \right) \le \mathsf{W}_B \quad \text{where } S(\mathbf{w} \in \mathbb{W}) \equiv |\mathbf{w_y}| + |\mathbf{w_i}| \cdot \mathsf{W}_F + \sum_{(h,l) \in \mathbf{w_x}} l \tag{14.5}$$

$$\mathsf{W}_F \equiv \mathsf{W}_G + 32\lceil \log_2(\mathsf{W}_M) \rceil \tag{14.6}$$

$$\mathsf{W}_B \equiv \mathsf{W}_M \cdot \mathsf{W}_F + 4096 + 64 + 64 = 13,791,360 \tag{14.7}$$

We limit the sums of each of the two gas limits to be at most the maximum gas allocated to a core for the corresponding operation:

$$\forall \mathbf{p} \in \mathbb{P} : \sum_{\mathbf{w} \in \mathbf{p_w}} (\mathbf{w}_a) < \mathsf{G}_A \quad \wedge \quad \sum_{\mathbf{w} \in \mathbf{p_w}} (\mathbf{w}_g) < \mathsf{G}_R \tag{14.8}$$

Given the result $\mathbf{l}$ and gas used u of some work-item, we define the item-to-digest function C as:

$$C \colon \begin{cases} (\mathbb{W}, \mathbb{B} \cup \mathbb{E}, \mathbb{N}_G) \to \mathbb{D} \\ \left(\begin{pmatrix} s, c, \mathbf{y}, \\ a, e, \mathbf{i}, \mathbf{x} \end{pmatrix}, \mathbf{l}, u \right) \mapsto \begin{pmatrix} s,\ c,\ y \colon \mathcal{H}(\mathbf{y}),\ g \colon a,\ \mathbf{l},\ u, \\ i \colon |\mathbf{i}|,\ e,\ x \colon |\mathbf{x}|,\ z \colon \sum_{(h,z) \in \mathbf{x}} z \end{pmatrix} \end{cases} \tag{14.9}$$

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

$$\forall \mathbf{p} \in \mathbb{P} : \begin{cases} \mathbf{p}_a \equiv \mathcal{H}(\mathbf{p}_u \frown \mathbf{p_f}) \\ \mathcal{E}(\updownarrow \mathbf{p_m}, \mathbf{p_u}) \equiv \Lambda(\delta[\mathbf{p}_h], (\mathbf{p_c})_t, \mathbf{p}_u) \\ (\mathbf{p_m}, \mathbf{p_u}) \in (\mathbb{B}, \mathbb{B}) \end{cases} \tag{14.10}$$

(The historical lookup function, Λ, is defined in equation 9.7.)

14.3.1. *Exporting.* Any of a work-package's work-items may *export* segments and a *segments-root* is placed in the work-report committing to these, ordered according to the work-item which is exporting. It is formed as the root of a constant-depth binary Merkle tree as defined in equation E.4.

Guarantors are required to erasure-code and distribute two data sets: one blob, the auditable *bundle* containing the encoded work-package, extrinsic data and self-justifying imported segments which is placed in the short-term Audit DA store; and a second set of exported-segments data together with the *Paged-Proofs* metadata. Items in the first store are short-lived; assurers are expected to keep them only until finality of the block in which the availability of the work-result's work-package is assured. Items in the second, meanwhile, are long-lived and expected to be kept for a minimum of 28 days (672 complete epochs) following the reporting of the work-report. This latter store is referred to as the *Distributed, Decentralized, Data Lake* or D^{3}L owing to its large size.

We define the paged-proofs function P which accepts a series of exported segments $\mathbf{s}$ and defines some series of additional segments placed into the D^{3}L via erasure-coding and distribution. The function evaluates to pages of hashes, together with subtree proofs, such that justifications of correctness based on a segments-root may be made from it:

$$P \colon \begin{cases} [\mathbb{J}] \to [\mathbb{J}] \\ \quad \mathbf{s} \mapsto \left[\mathcal{P}_l(\mathcal{E}(\updownarrow \mathcal{J}_6(\mathbf{s}, i), \updownarrow \mathcal{L}_6(\mathbf{s}, i))) \;\middle|\; i \in \mathbb{N}_{\lceil |\mathbf{s}|/64 \rceil} \right] \\ \qquad \text{where } l = \mathsf{W}_G \end{cases} \tag{14.11}$$

14.4. **Computation of Work-Report.** We now come to the work-report computation function Ξ. This forms the basis for all utilization of cores on JAM. It accepts some work-package $\mathbf{p}$ for some nominated core c and results in either an error ∇ or the work-report and series of exported segments. This function is deterministic and requires only that it be evaluated within eight epochs of a recently finalized block thanks to the historical lookup functionality. It can thus comfortably be evaluated by any node within the auditing period, even allowing for practicalities of imperfect synchronization. Formally:

$$\Xi \colon \begin{cases} (\mathbb{P}, \mathbb{N}_\mathsf{C}) \to \mathbb{R} \\ \quad (\mathbf{p}, c) \mapsto \begin{cases} \nabla & \text{if } \mathbf{t} \notin \mathbb{B}_{\mathsf{W}_R} \\ (\mathbf{s}, \mathbf{c} \colon \mathbf{p_c}, c, a \colon \mathbf{p}_a, \mathbf{t}, \mathbf{l}, \mathbf{d}, g) & \text{otherwise} \end{cases} \end{cases} \tag{14.12}$$

Where:

$$\mathcal{K}(\mathbf{l}) \equiv \left\{ h \;\middle|\; \mathbf{w} \in \mathbf{p_w}, \left(h^{\boxplus}, n\right) \in \mathbf{w_i} \right\}, \quad |\mathbf{l}| \le 8$$

$$(\mathbf{t}, g) = \Psi_I(\mathbf{p}, c)$$

$$(\mathbf{d}, \overline{\mathbf{e}}) = {}^{\mathrm{T}}\left[(C(\mathbf{p_w}[j], r, u), \mathbf{e}) \;\middle|\; (r, u, \mathbf{e}) = I(\mathbf{p}, j),\ j \in \mathbb{N}_{|\mathbf{p_w}|} \right]$$

$$I(\mathbf{p}, j) \equiv \begin{cases} \left(\ominus, u, [\mathbb{J}_0, \mathbb{J}_0, \dots]_{\dots m_e}\right) & \text{if } |r| + z > \mathsf{W}_R \\ \left(\circledcirc, u, [\mathbb{J}_0, \mathbb{J}_0, \dots]_{\dots m_e}\right) & \text{otherwise if } |\mathbf{e}| \neq m_e \\ \left(r, u, [\mathbb{J}_0, \mathbb{J}_0, \dots]_{\dots m_e}\right) & \text{otherwise if } r \notin \mathbb{B} \\ (r, u, \mathbf{e}) & \text{otherwise} \\ \quad \text{where } (r, \mathbf{e}, u) = \Psi_R(c, j, \mathbf{p}, \mathbf{o}, S^{\#}(\mathbf{p_w}), \ell) \\ \quad \text{and } h = \mathcal{H}(\mathbf{p}),\ m = \mathbf{p_w}[j],\ \ell = \sum_{k<j} \mathbf{p_w}[k]_e \\ \quad \text{and } z = |\mathbf{o}| + \sum_{k<j, (r \in \mathbb{B}, \dots) = I(\mathbf{p}, k)} |r| \end{cases}$$

Note that we gracefully handle both the case where the output size of the work output would take the work-report beyond its acceptable size and where number of segments exported by a work-item's Refinement execution is incorrectly reported in the work-item's export segment count. In both cases, the work-package continues to be valid as a whole, but the work-item's exported segments are replaced by a sequence of zero-segments equal in size to the export segment count and its output is replaced by an error.

Initially we constrain the segment-root dictionary $\mathbf{l}$: It should contain entries for all unique work-package hashes of imported segments not identified directly via a segment-root but rather through a work-package hash.

We immediately define the segment-root lookup function L, dependent on this dictionary, which collapses a union of segment-roots and work-package hashes into segment-roots using the dictionary:

$$L(r \in \mathbb{H} \cup \mathbb{H}^{\boxplus}) \equiv \begin{cases} r & \text{if } r \in \mathbb{H} \\ \mathbf{l}[h] & \text{if } \exists h \in \mathbb{H} : r = h^{\boxplus} \end{cases} \tag{14.13}$$

In order to expect to be compensated for a work-report they are building, guarantors must compose a value for $\mathbf{l}$ to ensure not only the above but also a further constraint

that all pairs of work-package hashes and segment-roots do properly correspond:

(14.14)
$$\forall (h \mapsto e) \in \mathbf{l} : \exists \mathbf{p}, c \in \mathbb{P}, \mathbb{N}_{\mathsf{C}} : \mathcal{H}(\mathbf{p}) = h \wedge (\Xi(\mathbf{p}, c)_{\mathbf{s}})_e = e$$

As long as the guarantor is unable to satisfy the above constraints, then it should consider the work-package unable to be guaranteed. Auditors are not expected to populate this but rather to reuse the value in the work-report they are auditing.

The next term to be introduced, $(\mathbf{t}, g)$, is the authorization trace, the result of the Is-Authorized function together with the amount of gas it used. The second term, $(\mathbf{d}, \overline{\mathbf{e}})$ is the sequence of results for each of the work-items in the work-package together with all segments exported by each work-item. The third definition I performs an ordered accumulation (i.e. counter) in order to ensure that the Refine function has access to the total number of exports made from the work-package up to the current work-item.

The above relies on two functions, S and X which, respectively, define the import segment data and the extrinsic data for some work-item argument $\mathbf{w}$. We also define J, which compiles justifications of segment data:

(14.15)
$$X(\mathbf{w} \in \mathbb{W}) \equiv [\mathbf{d} \mid (\mathcal{H}(\mathbf{d}), |\mathbf{d}|) \lessdot \mathbf{w}_{\mathbf{x}}]$$
$$S(\mathbf{w} \in \mathbb{W}) \equiv [\mathbf{b}[n] \mid \mathcal{M}(\mathbf{b}) = L(r), (r, n) \lessdot \mathbf{w}_{\mathbf{i}}]$$
$$J(\mathbf{w} \in \mathbb{W}) \equiv [\updownarrow \mathcal{J}_0(\mathbf{b}, n) \mid \mathcal{M}(\mathbf{b}) = L(r), (r, n) \lessdot \mathbf{w}_{\mathbf{i}}]$$

We may then define $\mathbf{s}$ as the data availability specification of the package using these three functions together with the yet to be defined *Availability Specifier* function A (see section 14.4.1):

$$\mathbf{s} = A(\mathcal{H}(\mathbf{p}), \mathcal{E}\Big(\mathbf{p}, X^{\#}(\mathbf{p}_{\mathbf{w}}), S^{\#}(\mathbf{p}_{\mathbf{w}}), J^{\#}(\mathbf{p}_{\mathbf{w}})\Big), \widehat{\mathbf{e}}) \tag{14.16}$$

Note that while the formulations of S and J seem to require (due to the inner term $\mathbf{b}$) all segments exported by all work-packages exporting a segment to be imported, such a vast amount of data is not generally needed. In particular, each justification can be derived through a single paged-proof. This reduces the worst case data fetching for a guarantor to two segments for every one to be imported. In the case that contiguously exported segments are imported (which we might assume is a fairly common situation), then a single proof-page should be sufficient to justify many imported segments.

Also of note is the lack of length prefixes: only the Merkle paths for the justifications have a length prefix. All other sequence lengths are determinable through the work package itself.

The Is-Authorized logic it references must be executed first in order to ensure that the work-package warrants the needed core-time. Next, the guarantor should ensure that all segment-tree roots which form imported segment commitments are known and have not expired. Finally, the guarantor should ensure that they can fetch all preimage data referenced as the commitments of extrinsic segments.

Once done, then imported segments must be reconstructed. This process may in fact be lazy as the Refine function makes no usage of the data until the *fetch* host-call is made. Fetching generally implies that, for each imported segment, erasure-coded chunks are retrieved from enough unique validators (342, including the guarantor) and is described in more depth in appendix H. (Since we specify systematic erasure-coding, its reconstruction is trivial in the case that the correct 342 validators are responsive.) Chunks must be fetched for both the data itself and for justification metadata which allows us to ensure that the data is correct.

Validators, in their role as availability assurers, should index such chunks according to the index of the segments-tree whose reconstruction they facilitate. Since the data for segment chunks is so small at 12 octets, fixed communications costs should be kept to a bare minimum. A good network protocol (out of scope at present) will allow guarantors to specify only the segments-tree root and index together with a Boolean to indicate whether the proof chunk need be supplied. Since we assume at least 341 other validators are online and benevolent, we can assume that the guarantor can compute S and J above with confidence, based on the general availability of data committed to with $\mathbf{s}^{\clubsuit}$, which is specified below.

14.4.1. *Availability Specifier.* We define the availability specifier function A, which creates an availability specifier from the package hash, an octet sequence of the audit-friendly work-package bundle (comprising the work-package itself, the extrinsic data and the concatenated import segments along with their proofs of correctness), and the sequence of exported segments:

$$A: \begin{cases} (\mathbb{H}, \mathbb{B}, [\![\mathbb{J}]\!]) \to \mathbb{Y} \\ \quad (p, \mathbf{b}, \mathbf{s}) \mapsto (p,\ l: |\mathbf{b}|,\ u,\ e: \mathcal{M}(\mathbf{s}),\ n: |\mathbf{s}|) \end{cases} \tag{14.17}$$

$$\text{where } u = \mathcal{M}_B\big([\widehat{\mathbf{x}} \mid \mathbf{x} \lessdot {}^{\mathrm{T}}[\mathbf{b}^{\clubsuit}, \mathbf{s}^{\clubsuit}]]\big)$$
$$\text{and } \mathbf{b}^{\clubsuit} = \mathcal{H}^{\#}\big(\mathcal{C}_{\lceil |\mathbf{b}|/\mathsf{W}_E \rceil}(\mathcal{P}_{\mathsf{W}_E}(\mathbf{b}))\big)$$
$$\text{and } \mathbf{s}^{\clubsuit} = \mathcal{M}_B^{\#}\Big({}^{\mathrm{T}}\mathcal{C}_6^{\#}(\mathbf{s} \frown P(\mathbf{s}))\Big)$$

The paged-proofs function P, defined earlier in equation 14.11, accepts a sequence of segments and returns a sequence of paged-proofs sufficient to justify the correctness of every segment. There are exactly $\lceil {}^{1}/_{64} \rceil$ paged-proof segments as the number of yielded segments, each composed of a page of 64 hashes of segments, together with a Merkle proof from the root to the subtree-root which includes those 64 segments.

The functions $\mathcal{M}$ and $\mathcal{M}_B$ are the fixed-depth and simple binary Merkle root functions, defined in equations E.4 and E.3. The function $\mathcal{C}$ is the erasure-coding function, defined in appendix H.

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

$$\mathcal{P}_{n \in \mathbb{N}_{1\ldots}} : \begin{cases} \mathbb{B} \to \mathbb{B}_{k \cdot n} \\ \mathbf{x} \mapsto \mathbf{x} \frown [0, 0, \ldots]_{((|x|+n-1) \bmod n)+1 \ldots n} \end{cases} \tag{14.18}$$

Validators are incentivized to distribute each newly erasure-coded data chunk to the relevant validator, since they are not paid for guaranteeing unless a work-report is considered to be *available* by a super-majority of validators. Given our work-package $\mathbf{p}$, we should therefore send the corresponding work-package bundle chunk and exported segments chunks to each validator whose keys are together with similarly corresponding chunks for imported, extrinsic and exported segments data, such that each validator can justify completeness according to the work-report's *erasure-root*. In the case of a coming epoch change, they may also maximize expected reward by distributing to the new validator set.

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

15. 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) Creation and publication of a work-package report.
(3) Chunking of the work-package and each of its extrinsic and exported data, according to the erasure codec.
(4) Distributing the aforementioned chunks across the validator set.
(5) Providing the work-package, extrinsic and exported data to other validators on request is also helpful for optimal network performance.

For any work-package p we are in receipt of, we may determine the work-report, 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 p, we define the work-report r simply as:

$$r = \Xi(p, c) \tag{15.1}$$

Such guarantors may safely create and distribute the payload (s, v). The component s may be created according to equation 11.26; specifically it is a signature using the validator's registered Ed25519 key on a payload l:

$$l = \mathcal{H}(\mathcal{E}(r)) \tag{15.2}$$

To maximize profit, the guarantor should require the work-digest meets all expectations which are in place during the guarantee extrinsic described in section 11.4. This includes contextual validity and inclusion of the authorization in the authorization pool. 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.

16. Availability Assurance

Validators should issue a signed statement, called an *assurance*, when they are in possession of all of their corresponding erasure-coded chunks for a given work-report which is currently pending availability. For any work-report to gain an assurance, there are two classes of data a validator must have:

Firstly, their erasure-coded chunk for this report's bundle. The validity of this chunk can be trivially proven through the work-report's work-package erasure-root and a Merkle-proof of inclusion in the correct location. The proof should be included from the guarantor. This chunk is needed to verify the work-report's validity and completeness and need not be retained after the work-report is considered audited. Until then, it should be provided on request to validators.

Secondly, the validator should have in hand the corresponding erasure-coded chunk for each of the exported segments referenced by the *segments root*. These should be retained for 28 days and provided to any validator on request.

17. Auditing and Judging

The auditing and judging system is theoretically equivalent to that in Elves, introduced by **cryptoeprint:2024/961**. For a full security analysis of the mechanism, see this work. There is a difference in terminology, where the terms *backing*, *approval* and *inclusion* there refer to our guaranteeing, auditing and accumulation, respectively.

17.1. **Overview.** The auditing process involves each node requiring themselves to fetch, evaluate and issue judgment 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 judgment in a reasonable time or in the event of a negative judgment being seen. These enlargement events are called tranches.

If all declared intentions for a work-report are matched by a positive judgment 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 judgments 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 judgment is found, then one of several things happens; if there are still more than $^2/_3\mathsf{V}$ positive judgments, then validators issuing negative judgments may receive a punishment for time-wasting. If there are greater than $^1/_3\mathsf{V}$ negative judgments, 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 judgment 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 judgments 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).

17.2. **Data Fetching.** For each work-report to be audited, we use its erasure-root to request erasure-coded chunks from enough assurers. From each assurer we fetch three items (which with a good network protocol should be done under a single request) corresponding to the work-package super-chunks, the self-justifying imports super-chunks and the extrinsic segments super-chunks.

We may validate the work-package reconstruction by ensuring its hash is equivalent to the hash includes as part of the work-package specification in the work-report. We may validate the extrinsic segments through ensuring their hashes are each equivalent to those found in the relevant work-item.

Finally, we may validate each imported segment as a justification must follow the concatenated segments which allows verification that each segment's hash is included in the referencing Merkle root and index of the corresponding work-item.

Exported segments need not be reconstructed in the same way, but rather should be determined in the same manner as with guaranteeing, i.e. through the execution of the Refine logic.

All items in the work-package specification field of the work-report should be recalculated from this now known-good data and verified, essentially retracing the guarantors steps and ensuring correctness.

17.3. **Selection of Reports.** 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 consider ourselves a specific validator of index v and assume ourselves focused on some recent block $\mathbf{B}$ with other terms corresponding to the state-transition implied by that block, so ρ is said block's prior core-allocation, κ is its prior validator set, $\mathbf{H}$ 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:

$$\mathbf{q} \in \llbracket \mathbb{R}? \rrbracket_{\mathsf{C}} \tag{17.1}$$

$$\mathbf{q} \equiv \left[\begin{cases} \rho[c]_{\mathbf{r}} & \text{if } \rho[c]_{\mathbf{r}} \in \mathbf{R} \\ \varnothing & \text{otherwise} \end{cases} \;\middle|\; c \in \mathbb{N}_{\mathsf{C}} \right] \tag{17.2}$$

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

$$s_0 \in \tilde{\mathbb{V}}^{[]}_{\kappa[v]_b}\langle \mathsf{X}_U \frown \mathcal{Y}(\mathbf{H}_V) \rangle \tag{17.3}$$

$$\mathsf{X}_U = \texttt{\$jam\_audit} \tag{17.4}$$

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

$$\mathbf{a}_0 = \{ (\mathbf{r}, c) \mid (\mathbf{r}, c) \in \mathbf{p}_{\dots+10}, \mathbf{r} \neq \varnothing \} \tag{17.5}$$

$$\text{where } \mathbf{p} = \mathcal{F}([(c, \mathbf{q}_c) \mid c \in \mathbb{N}_{\mathsf{C}}], \mathcal{Y}(s_0)) \tag{17.6}$$

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:

$$\text{let } n = \left\lfloor \frac{\mathcal{T} - \mathsf{P} \cdot \mathbf{H}_T}{\mathsf{A}} \right\rfloor \tag{17.7}$$

New tranches may contain items from $\mathbf{q}$ stemming from one of two reasons: either a negative judgment has been received; or the number of judgments 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 judgment on the work-report. In the second case, a new special-purpose VRF must be constructed to determine if an audit and judgment 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 judgment 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 S:

$$S \equiv \bar{\mathbb{V}}_{\kappa[v]_e}\langle \mathsf{X}_I \plusplus n \frown \mathbf{x}_n \frown \mathcal{H}(\mathbf{H}) \rangle \tag{17.8}$$

$$\text{where } \mathbf{x}_n = \mathcal{E}(\{ \mathcal{E}_2(c) \frown \mathcal{H}(\mathbf{r}) \mid (\mathbf{r}, c) \in \mathbf{a}_n \}) \tag{17.9}$$

$$\mathsf{X}_I = \texttt{\$jam\_announce} \tag{17.10}$$

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.

$$A_n : \mathbb{R} \to \{\!\{ \mathbb{N}_{\mathsf{V}} \}\!\} \tag{17.11}$$

(17.12)

We further define $J_\top$ and $J_\bot$ to be the validator indices who we know to have made respectively, positive and negative, judgments mapped from each work-report's

core. We don't care from which tranche a judgment is made.

$$J_{\{\perp,\top\}} : \mathbb{R} \to \{\!\{ \mathbb{N}_V \}\!\} \tag{17.13}$$

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 judgment. 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.

$$\forall n > 0 :$$

$$s_n(\mathbf{r}) \in \tilde{\mathbb{V}}^{[]}_{\kappa[v]_b}\langle \mathsf{X}_U \frown \mathcal{Y}(\mathbf{H}_V) \frown \mathcal{H}(\mathbf{r}) \+ n \rangle \tag{17.14}$$

$$\mathbf{a}_n \equiv \left\{ \mathbf{r} \;\middle|\; \tfrac{V}{256\mathsf{F}}\mathcal{Y}(s_n(\mathbf{r}))_0 < m_n, \mathbf{r} \in \mathbf{q}, \mathbf{r} \neq \varnothing \right\} \tag{17.15}$$

$$\text{where } m_n = |A_{n-1}(\mathbf{r}) \setminus J_\top(\mathbf{r})|$$

We define our bias factor $\mathsf{F} = 2$, which is the expected number of validators which will be required to issue a judgment for a work-report given a single no-show in the tranche before. Modeling by **cryptoeprint:2024/961** shows that this is optimal.

Later audits must be announced in a similar fashion to the first. If audit requirements lessen on the receipt of new information (i.e. a positive judgment being returned for a previous *no-show*), then any audits already announced are completed and judgments published. If audit requirements raise on the receipt of new information (i.e. an additional announcement being found without an accompanying judgment), 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 and their requisite data 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 by asking a cooperative third party (e.g. an original guarantor) for the preimages.

Thus, for any such work-report $\mathbf{r}$ we are assured we will be able to fetch some candidate work-package encoding $F(\mathbf{r})$ 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.

In addition to the work-package, we also assume we are able to fetch all manifest data associated with it through requesting and reconstructing erasure-coded chunks from one-third of validators in the same way as above.

We then attempt to reproduce the report on the core to give e_n, a mapping from cores to evaluations:

$$\forall (c, \mathbf{r}) \in \mathbf{a}_n : \quad e_n(c) \Leftrightarrow \begin{cases} \mathbf{r} = \Xi(p, c) & \text{if } \exists p \in \mathbb{P} : \mathcal{E}(p) = F(\mathbf{r}) \\ \perp & \text{otherwise} \end{cases} \tag{17.16}$$

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

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

$$\mathbf{j}_n = \left\{ \bar{\mathcal{S}}_{\kappa[]}\left(\mathsf{X}_{e_n(c)} \frown \mathcal{H}(\mathbf{r})\right) \;\middle|\; (c, \mathbf{r}) \in \mathbf{a}_n \right\} \tag{17.17}$$

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

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

$$U(\mathbf{r}) \Leftrightarrow \bigvee \begin{cases} J_\perp(\mathbf{r}) = \varnothing \wedge \exists n : A_n(\mathbf{r}) \subset J_\top(\mathbf{r}) \\ |J_\top(\mathbf{r})| > 2/3 V \end{cases} \tag{17.18}$$

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:

$$\mathbf{U} \Leftrightarrow \forall \mathbf{r} \in \mathbf{R} : U(\mathbf{r}) \tag{17.19}$$

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 19 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 judgment extrinsic which collects judgment signatures together and reports them on-chain. In the case of a non-valid judgment (i.e. one which is not two-thirds-plus-one of judgments 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 judgment extrinsic removes it from the pending work-reports, ρ. Refer to section 10 for more details on this.

18. 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 **bls12-381**, 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 **cryptoeprint:2022/1611**.

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

$$\mathbf{F}_v \equiv \overset{\text{BLS}}{\mathcal{S}}_{\kappa'}(\mathsf{X}_B \frown \text{last}(\beta_H)_b) \tag{18.1}$$

$$\mathsf{X}_B = \texttt{\$jam\_beefy} \tag{18.2}$$

19. GRANDPA AND THE BEST CHAIN

Nodes take part in the GRANDPA protocol as defined by **stewart2020grandpa**.

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:

$$\mathbf{A}(\mathbf{H}^\flat) \ni \mathbf{H}^\natural \tag{19.1}$$

$$\mathbf{U}^\flat \equiv \top \tag{19.2}$$

$$\nexists\, \mathbf{H}^A, \mathbf{H}^B : \bigwedge \begin{cases} \mathbf{H}^A \neq \mathbf{H}^B \\ \mathbf{H}^A_T = \mathbf{H}^B_T \\ \mathbf{H}^A \in \mathbf{A}(\mathbf{H}^\flat) \\ \mathbf{H}^A \notin \mathbf{A}(\mathbf{H}^\natural) \end{cases} \tag{19.3}$$

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:

$$m = \sum_{\mathbf{H}^A \in \mathbf{A}^\flat} \mathbf{T}^A \tag{19.4}$$

The specific data to be voted on in Grandpa shall be the block header of the best block, $\mathbf{B}^\flat$ together with its *posterior* state root, $\mathcal{M}_\sigma(\sigma')$. The state root has no direct relevance to the Grandpa protocol, but is included alongside the header during voting/signing into order to ensure that systems utilizing the output of Grandpa are able to verify the most recent chain state as possible.

This implies that the posterior state must be known at the time that Grandpa voting occurs in order to finalize the block. However, since Grandpa is relied on primarily for state-root verification it makes little sense to finalize a block without an associated commitment to the posterior state.

The posterior state only affects Grandpa voting in so much as votes for the same block hash but with different associated posterior state roots are considered votes for different blocks. This would only happen in the case of a misbehaving node or an ambiguity in the present document.

20. Discussion

20.1. **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, 8TB 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:

Throughput, MB/slot	*Tx*	*Rx*
Guaranteeing	106	48
Assuring	144	13
Auditing	0	133
Authoring	53	87
Grandpa and Beefy	4	4
Total	**304**	**281**
Implied bandwidth, Mb/s	**387**	**357**

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 2MB/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 48KB. This work-result is then entitled to 10ms on the same machine, this time with no "external" I/O, but instead with full and immediate access to the Jam chain state and may alter the service(s) to which the results belong.

20.2. **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.

20.2.1. *Comparison to Polkadot.* Polkadot is at present capable of validating at most 80 parachains each doing one second of native computation and 5MB of I/O in every six. This corresponds to an aggregate compute performance of around 13x native CPU and a total 24-hour distributed availability of around 67MB/s. Accumulation is beyond Polkadot's capabilities and so not comparable.

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 682MB/s.

20.2.2. *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 12MB work-package can hold around 96k transactions at 128 bytes per transaction. However, a 48KB work-result could only encode around 6k 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 3k transactions per package, or 171k 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 12MB of the work-package would be split between transaction data and state witness data. Our basic models predict that a 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 1.4mTPS 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.

20.2.3. *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, 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 computation-gas as well as 78KB input data in each second, trading one against the other.[13] 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[14] show that on modern, high-end consumer hardware with a high-quality EVM implementation, we can expect somewhere between 100 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 transpilation of the EVM code into PVM code and then re-execution of it under the PolkaVM prototype.[15]

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 around 100 gas/µs. We therefore assert that in terms of computation, 1µs approximates to around 100-500 EVM gas on modern high-end consumer hardware.[16]

[13] 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.

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

[15] 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 64-bit native PVM is being forced to honor this even if the source code only actually required 64-bit values.

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

Benchmarking and regression tests show that the prototype PVM engine has a fixed preprocessing overhead of around 5ns/byte of program code and, for arithmetic-heavy tasks at least, a marginal factor of 1.6-2% compared to EVM execution, implying an asymptotic speedup of around 50-60x. 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. [17] For code not inherently suited to the 256-bit EVM ISA, we would expect substantially improved relative execution times on PVM, though more work must be done in order to gain confidence that these speed-ups are broadly applicable.

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. Owing to the crudeness of our analysis we might reasonably predict it to be somewhere within a factor of three either way—i.e. 500-5,000 EVM gas/µs.

JAM cores are each capable of 2MB/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 over 3k 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 updates are made through the 8KB/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†	500-5,000	0.15-1.5M
State writes (s^{-1})	57†	n/a	n/a
State reads (s^{-1})	595†	4K‡	1.4M‡
Input data (s^{-1})	78KB†	2MB‡	682MB‡

What we can see is that JAM's overall predicted performance profile implies it could be comparable to many thousands 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 only for the purpose of expressing JAM's performance in tangible terms. 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 1/4 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 1/2 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.

21. 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.

21.1. 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*

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

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.

22. 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 Galassi 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 Emett have been instrumental in the design of the lower-level aspects of the JAM protocol, especially concerning Merklization and I/O.

Numerous contributors to the repository since publication have helped correct errors. Thank you to all.

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. Polkadot 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. We additionally define the function deblob which extracts the instruction data, opcode bitmask and dynamic jump table from a program blob:

$$\Psi: \begin{cases} (\mathbb{B}, \mathbb{N}_R, \mathbb{N}_G, [\mathbb{N}_R]_{13}, \mathbb{M}) \to \left(\{\blacksquare, \lightning, \infty\} \cup \{\exclamation, \hbar\} \times \mathbb{N}_R, \mathbb{N}_R, \mathbb{Z}_G, [\mathbb{N}_R]_{13}, \mathbb{M}\right) \\ \quad (\mathbf{p}, \imath, \varrho, \varphi, \mu) \mapsto \begin{cases} \Psi(\mathbf{p}, \imath', \varrho', \varphi', \mu') & \text{if } \varepsilon = \blacktriangleright \\ (\infty, \imath, \varrho', \varphi, \mu) & \text{if } \varrho' < 0 \\ (\varepsilon, 0, \varrho', \varphi', \mu') & \text{if } \varepsilon \in \{\lightning, \blacksquare\} \\ (\varepsilon, \imath, \varrho', \varphi, \mu) & \text{otherwise} \end{cases} \\ \quad \text{where } (\varepsilon, \imath', \varrho', \varphi', \mu') = \begin{cases} \Psi_1(\mathbf{c}, \mathbf{k}, \mathbf{j}, \imath, \varrho, \varphi, \mu) & \text{if } (\mathbf{c}, \mathbf{k}, \mathbf{j}) = \text{deblob}(\mathbf{p}) \\ (\lightning, \imath, \varrho, \varphi, \mu) & \text{otherwise} \end{cases} \end{cases} \tag{A.1}$$

$$\text{deblob}: \begin{cases} \mathbb{B} \to (\mathbb{B}, \mathbb{b}, [\mathbb{N}_R]) \cup \nabla \\ \mathbf{p} \mapsto \begin{cases} (\mathbf{c}, \mathbf{k}, \mathbf{j}) & \text{if } \exists! \, \mathbf{c}, \mathbf{k}, \mathbf{j} : \mathbf{p} = \mathcal{E}(|\mathbf{j}|) \frown \mathcal{E}_1(z) \frown \mathcal{E}(|\mathbf{c}|) \frown \mathcal{E}_z(\mathbf{j}) \frown \mathcal{E}(\mathbf{c}) \frown \mathcal{E}(\mathbf{k}), \ |\mathbf{k}| = |\mathbf{c}| \\ \nabla & \text{otherwise} \end{cases} \end{cases} \tag{A.2}$$

The PVM exit reason $\varepsilon \in \{\blacksquare, \lightning, \infty\} \cup \{\exclamation, \hbar\} \times \mathbb{N}_R$ 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 $\exclamation$ in which case the address into RAM is associated.

Assuming the program blob is valid (which can be validated statically), some gas is always charged whenever execution is attempted. This is the case even if no instruction is effectively executed and machine state is unchanged (i.e. the result state is equal to the parameter).

In the case of a final halt, either through panic or success, the instruction counter returned is zero. In all other cases, the return value of the instruction counter indexes the one *which caused the exit to happen* and the machine state represents the prior state of said instruction, thus ensuring *de facto* consistency. In order to continue beyond these exit cases, some environmental factor must be adjusted; for a page-fault, RAM must be changed, for a gas-underflow, more gas must be supplied and for a host-call, the instruction-counter must be incremented and the relevant host-call state-transition performed.

A.2. **Instructions, Opcodes and Skip-distance.** The program blob $\mathbf{p}$ is split into a series of octets which make up the *instruction data* $\mathbf{c}$ and the *opcode bitmask* $\mathbf{k}$ as well as the *dynamic jump table*, $\mathbf{j}$. The former two imply 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 latter, dynamic jump table, is a sequence of indices into the instruction data blob and is indexed into when dynamically-computed jumps are taken. It is encoded as a sequence of natural numbers (i.e. non-negative integers) each encoded with the same length in octets. This length, term z above, is itself encoded prior.

The PVM counts instructions in octet terms (rather than in terms of instructions) and it is thus necessary to define which octets represent the beginning of an instruction, i.e. the opcode octet, and which do not. This is the purpose of $\mathbf{k}$, the instruction-opcode bitmask. We assert that the length of the bitmask is equal to the length of the instruction blob.

We define the Skip function skip which provides the number of octets, minus one, to the next instruction's opcode, given the index of instruction's opcode index into $\mathbf{c}$ (and by extension $\mathbf{k}$):

$$\text{skip}: \begin{cases} \mathbb{N} \to \mathbb{N} \\ i \mapsto \min(24, \ j \in \mathbb{N} : (\mathbf{k} \frown [1, 1, \dots])_{i+1+j} = 1) \end{cases} \tag{A.3}$$

The Skip function appends $\mathbf{k}$ with a sequence of set bits in order to ensure a well-defined result for the final instruction $\text{skip}(|\mathbf{c}| - 1)$.

Given some instruction-index i, its opcode is readily expressed as $\mathbf{c}_i$ and the distance in octets to move forward to the next instruction is $1 + \text{skip}(i)$. However, each instruction's "length" (defined as the number of contiguous octets starting with the opcode which are needed to fully define the instruction's semantics) is left implicit though limited to being at most 16.

We define ζ as being equivalent to the instructions $\mathbf{c}$ except with an indefinite sequence of zeroes suffixed to ensure that no out-of-bounds access is possible. This effectively defines any otherwise-undefined arguments to the final instruction and ensures that a trap will occur if the program counter passes beyond the program code. Formally:

$$\zeta \equiv \mathbf{c} \frown [0, 0, \dots] \tag{A.4}$$

A.3. **Basic Blocks and Termination Instructions.** Instructions of the following opcodes are considered basic-block termination instructions; other than `trap` & `fallthrough`, they correspond to instructions which may define the instruction-counter to be something other than its prior value plus the instruction's skip amount:

- Trap and fallthrough: `trap` , `fallthrough`
- Jumps: `jump` , `jump_ind`
- Load-and-Jumps: `load_imm_jump` , `load_imm_jump_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, which is a subset of all valid opcode indices U. We define the instruction opcode indices denoting the beginning of basic-blocks as ϖ:

$$\varpi \equiv \left(\{\,0\,\} \cup \left\{\, n + 1 + \text{skip}(n) \;\middle|\; n \in \mathbb{N}_{|\mathbf{c}|} \wedge \mathbf{k}_n = 1 \wedge \mathbf{c}_n \in T \,\right\}\right) \cap \{\, n \mid \mathbf{k}_n = 1 \wedge \mathbf{c}_n \in U \,\} \tag{A.5}$$

A.4. **Single-Step State Transition.** We must now define the single-step PVM state-transition function Ψ_1:

$$\Psi_1 \colon \begin{cases} (\mathbb{B}, \mathbb{b}, [\![\mathbb{N}_R]\!], \mathbb{N}_R, \mathbb{N}_G, [\![\mathbb{N}_R]\!]_{13}, \mathbb{M}) \to \left(\{\, \lightning, \blacksquare, \blacktriangleright \,\} \cup \{\, \exists, \hbar \,\} \times \mathbb{N}_R, \mathbb{N}_R, \mathbb{Z}_G, [\![\mathbb{N}_R]\!]_{13}, \mathbb{M}\right) \\ \qquad (\mathbf{c}, \mathbf{k}, \mathbf{j}, \imath, \varrho, \varphi, \mu) \mapsto \left(\varepsilon^*, \imath^*, \varrho^*, \varphi^*, \mu^*\right) \end{cases} \tag{A.6}$$

During the course of executing instructions RAM may be accessed. When an index of RAM below 2^{16} is required, the machine always panics immediately without further changes to its state regardless of the apparent (in)accessibility of the value. Otherwise, should the given index of RAM not be accessible then machine state remains unchanged and the exit reason is a fault with the lowest inaccessible *page address* to be read. 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 page address to be written which is inaccessible.

Formally, let $\mathbf{r}$ and $\mathbf{w}$ be the set of indices by which μ must be subscripted for inspection and mutation respectively in order to calculate the result of Ψ_1. We define the memory-access exceptional execution state ε^μ which shall, if not $\blacktriangleright$, singly effect the returned return of Ψ_1 as following:

$$\text{let } \mathbf{x} = \left\{\, x \;\middle|\; x \in \mathbf{r} \wedge x \bmod 2^{32} \notin \mathbb{V}_\mu \;\vee\; x \in \mathbf{w} \wedge x \bmod 2^{32} \notin \mathbb{V}^*_\mu \,\right\} \tag{A.7}$$

$$\left(\varepsilon^*, \imath^*, \varrho^*, \varphi^*, \mu^*\right) = \begin{cases} (\varepsilon, \imath', \varrho', \varphi', \mu') & \text{if } \mathbf{x} = \{\} \\ \left(\lightning, \imath, \varrho, \varphi, \mu\right) & \text{if } \min(\mathbf{x}) \bmod 2^{32} < 2^{16} \\ \left(\exists \times \mathsf{Z}_P \middle\lfloor \min(\mathbf{x}) \bmod 2^{32} \div \mathsf{Z}_P \middle\rfloor, \imath, \varrho, \varphi, \mu\right) & \text{otherwise} \end{cases} \tag{A.8}$$

We define ε together with the posterior values of regular execution (denoted as prime) of each of the items of the machine state as being in accordance with the table below. When transitioning machine state for an instruction, a number of conditions typically 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:

$$\varepsilon = \blacktriangleright, \quad \imath' = \imath + 1 + \text{skip}(\imath), \quad \varrho' = \varrho - \varrho_\Delta, \quad \varphi' = \varphi, \quad \mu' = \mu \text{ except as indicated} \tag{A.9}$$

In the case that Ψ_1 takes the ε^μ

We define signed/unsigned transitions for various octet widths:

$$\mathcal{Z}_{n \in \mathbb{N}} \colon \begin{cases} \mathbb{N}_{2^{8n}} \to \mathbb{Z}_{-2^{8n-1} \ldots 2^{8n-1}} \\ \quad a \mapsto \begin{cases} a & \text{if } a < 2^{8n-1} \\ a - 2^{8n} & \text{otherwise} \end{cases} \end{cases} \tag{A.10}$$

$$\mathcal{Z}^{-1}_{n \in \mathbb{N}} \colon \begin{cases} \mathbb{Z}_{-2^{8n-1} \ldots 2^{8n-1}} \to \mathbb{N}_{2^{8n}} \\ \quad a \mapsto (2^{8n} + a) \bmod 2^{8n} \end{cases} \tag{A.11}$$

$$\mathcal{B}_{n \in \mathbb{N}} \colon \begin{cases} \mathbb{N}_{2^{8n}} \to \mathbb{b}_{8n} \\ \quad x \mapsto \mathbf{y} : \forall i \in \mathbb{N}_{8n} : \mathbf{y}[i] \Leftrightarrow \left\lfloor \frac{x}{2^i} \right\rfloor \bmod 2 \end{cases} \tag{A.12}$$

$$\mathcal{B}^{-1}_{n \in \mathbb{N}} \colon \begin{cases} \mathbb{b}_{8n} \to \mathbb{N}_{2^{8n}} \\ \quad \mathbf{x} \mapsto y : \sum_{i \in \mathbb{N}_{8n}} \mathbf{x}_i \cdot 2^i \end{cases} \tag{A.13}$$

$$\overleftarrow{\mathcal{B}}_{n \in \mathbb{N}} \colon \begin{cases} \mathbb{N}_{2^{8n}} \to \mathbb{b}_{8n} \\ \quad x \mapsto \mathbf{y} : \forall i \in \mathbb{N}_{8n} : \mathbf{y}[8n - 1 - i] \Leftrightarrow \left\lfloor \frac{x}{2^i} \right\rfloor \bmod 2 \end{cases} \tag{A.14}$$

$$\overleftarrow{\mathcal{B}}^{-1}_{n \in \mathbb{N}} \colon \begin{cases} \mathbb{b}_{8n} \to \mathbb{N}_{2^{8n}} \\ \quad \mathbf{x} \mapsto y : \sum_{i \in \mathbb{N}_{8n}} \mathbf{x}_{8n-1-i} \cdot 2^i \end{cases} \tag{A.15}$$

Immediate arguments are encoded in little-endian format with the most-significant bit being the sign bit. They may be compactly encoded by eliding more significant octets. Elided octets are assumed to be zero if the MSB of the value is zero, and 255 otherwise. This allows for compact representation of both positive and negative encoded values. We thus define the signed extension function operating on an input of n octets as $\mathcal{X}_n$:

$$\mathcal{X}_{n \in \{0,1,2,3,4,8\}} \colon \begin{cases} \mathbb{N}_{2^{8n}} \to \mathbb{N}_R \\ \quad x \mapsto x + \left\lfloor \frac{x}{2^{8n-1}} \right\rfloor (2^{64} - 2^{8n}) \end{cases} \tag{A.16}$$

Any alterations of the program counter stemming from a static jump, call or branch must be to the start of a basic block or else a panic occurs. Hypotheticals are not considered. Formally:

$$\texttt{branch}(b, C) \implies (\varepsilon, \imath') = \begin{cases} (\blacktriangleright, \imath) & \text{if } \neg C \\ (\lightning, \imath) & \text{otherwise if } b \notin \varpi \\ (\blacktriangleright, b) & \text{otherwise} \end{cases} \tag{A.17}$$

Jumps whose next instruction is dynamically computed must use an address which may be indexed into the jump-table $\mathbf{j}$. Through a quirk of tooling[18], 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 $\mathsf{Z}_A = 2$.

As with other irregular alterations to the program counter, target code index must be the start of a basic block or else a panic occurs. Formally:

$$\texttt{djump}(a) \implies (\varepsilon, \imath') = \begin{cases} (\blacksquare, \imath) & \text{if } a = 2^{32} - 2^{16} \\ (\lightning, \imath) & \text{otherwise if } a = 0 \vee a > |\mathbf{j}| \cdot \mathsf{Z}_A \vee a \bmod \mathsf{Z}_A \neq 0 \vee \mathbf{j}_{(a/\mathsf{Z}_A)-1} \notin \varpi \\ (\blacktriangleright, \mathbf{j}_{(a/\mathsf{Z}_A)-1}) & \text{otherwise} \end{cases} \tag{A.18}$$

A.5. **Instruction Tables.** Only instructions which are defined in the following tables and whose opcode has its corresponding bit set in the bitmask are considered valid, otherwise the instruction behaves as-if its opcode was equal to zero. Assuming U denotes all valid opcode indices, formally:

$$\text{opcode}: \begin{cases} \mathbb{N} \to \mathbb{N} \\ n \mapsto \begin{cases} \mathbf{c}_n & \text{if } \mathbf{k}_n = 1 \wedge \mathbf{c}_n \in U \\ 0 & \text{otherwise} \end{cases} \end{cases} \tag{A.19}$$

We assume the skip length ℓ is well-defined:

$$\ell \equiv \text{skip}(\imath) \tag{A.20}$$

A.5.1. *Instructions without Arguments.*

$\zeta_\imath$	Name	Mutations
0	`trap`	$\varepsilon = \lightning$
1	`fallthrough`	
3	`unlikely`	

A.5.2. *Instructions with Arguments of One Immediate.*

$$\text{let } l_X = \min(4, \ell), \quad \nu_X \equiv \mathcal{X}_{l_X}\left(\mathcal{E}^{-1}_{l_X}(\zeta_{\imath+1\cdots+l_X})\right) \tag{A.21}$$

$\zeta_\imath$	Name	Mutations
10	`ecalli`	$\varepsilon = \hbar \times \nu_X$

A.5.3. *Instructions with Arguments of One Register and One Extended Width Immediate.*

$$\text{let } r_A = \min(12, \zeta_{\imath+1} \bmod 16), \quad \varphi'_A \equiv \varphi'_{r_A}, \quad \nu_X \equiv \mathcal{E}^{-1}_8(\zeta_{\imath+2\cdots+8}) \tag{A.22}$$

$\zeta_\imath$	Name	Mutations
20	`load_imm_64`	$\varphi'_A = \nu_X$

[18] 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.

$\zeta_\imath$	Name	Mutations

A.5.4. *Instructions with Arguments of Two Immediates.*

$$\begin{aligned} &\text{let } l_X = \min(4, \zeta_{\imath+1} \bmod 8), && \nu_X \equiv \mathcal{X}_{l_X}\left(\mathcal{E}^{-1}_{l_X}(\zeta_{\imath+2\cdots+l_X})\right) \\ &\text{let } l_Y = \min(4, \max(0, \ell - l_X - 1)), && \nu_Y \equiv \mathcal{X}_{l_Y}\left(\mathcal{E}^{-1}_{l_Y}(\zeta_{\imath+2+l_X\cdots+l_Y})\right) \end{aligned} \tag{A.23}$$

$\zeta_\imath$	Name	Mutations
30	`store_imm_u8`	$\mu'^{\circlearrowleft}_{\nu_X} = \nu_Y \bmod 2^8$
31	`store_imm_u16`	$\mu'^{\circlearrowleft}_{\nu_X\cdots+2} = \mathcal{E}_2\left(\nu_Y \bmod 2^{16}\right)$
32	`store_imm_u32`	$\mu'^{\circlearrowleft}_{\nu_X\cdots+4} = \mathcal{E}_4\left(\nu_Y \bmod 2^{32}\right)$
33	`store_imm_u64`	$\mu'^{\circlearrowleft}_{\nu_X\cdots+8} = \mathcal{E}_8(\nu_Y)$

A.5.5. *Instructions with Arguments of One Offset.*

$$\text{let } l_X = \min(4, \ell), \quad \nu_X \equiv \imath + \mathcal{Z}_{l_X}(\mathcal{E}^{-1}_{l_X}(\zeta_{\imath+1\cdots+l_X})) \tag{A.24}$$

$\zeta_\imath$	Name	Mutations
40	`jump`	$\texttt{branch}(\nu_X, \top)$

A.5.6. *Instructions with Arguments of One Register & One Immediate.*

$$\begin{aligned} &\text{let } r_A = \min(12, \zeta_{\imath+1} \bmod 16), && \varphi_A \equiv \varphi_{r_A}, \quad \varphi'_A \equiv \varphi'_{r_A} \\ &\text{let } l_X = \min(4, \max(0, \ell - 1)), && \nu_X \equiv \mathcal{X}_{l_X}\left(\mathcal{E}^{-1}_{l_X}(\zeta_{\imath+2\cdots+l_X})\right) \end{aligned} \tag{A.25}$$

$\zeta_\imath$	Name	Mutations
50	`jump_ind`	$\texttt{djump}((\varphi_A + \nu_X) \bmod 2^{32})$
51	`load_imm`	$\varphi'_A = \nu_X$
52	`load_u8`	$\varphi'_A = \mu^{\circlearrowleft}_{\nu_X}$
53	`load_i8`	$\varphi'_A = \mathcal{X}_1\left(\mu^{\circlearrowleft}_{\nu_X}\right)$
54	`load_u16`	$\varphi'_A = \mathcal{E}_2^{-1}\left(\mu^{\circlearrowleft}_{\nu_X\cdots+2}\right)$
55	`load_i16`	$\varphi'_A = \mathcal{X}_2\left(\mathcal{E}_2^{-1}\left(\mu^{\circlearrowleft}_{\nu_X\cdots+2}\right)\right)$
56	`load_u32`	$\varphi'_A = \mathcal{E}_4^{-1}\left(\mu^{\circlearrowleft}_{\nu_X\cdots+4}\right)$
57	`load_i32`	$\varphi'_A = \mathcal{X}_4\left(\mathcal{E}_4^{-1}\left(\mu^{\circlearrowleft}_{\nu_X\cdots+4}\right)\right)$
58	`load_u64`	$\varphi'_A = \mathcal{E}_8^{-1}\left(\mu^{\circlearrowleft}_{\nu_X\cdots+8}\right)$
59	`store_u8`	$\mu'^{\circlearrowleft}_{\nu_X} = \varphi_A \bmod 2^8$
60	`store_u16`	$\mu'^{\circlearrowleft}_{\nu_X\cdots+2} = \mathcal{E}_2\left(\varphi_A \bmod 2^{16}\right)$
61	`store_u32`	$\mu'^{\circlearrowleft}_{\nu_X\cdots+4} = \mathcal{E}_4\left(\varphi_A \bmod 2^{32}\right)$
62	`store_u64`	$\mu'^{\circlearrowleft}_{\nu_X\cdots+8} = \mathcal{E}_8(\varphi_A)$

A.5.7. *Instructions with Arguments of One Register & Two Immediates.*

$$\begin{aligned} &\text{let } r_A = \min(12, \zeta_{\imath+1} \bmod 16), && \varphi_A \equiv \varphi_{r_A}, \quad \varphi'_A \equiv \varphi'_{r_A} \\ &\text{let } l_X = \min(4, \left\lfloor \frac{\zeta_{\imath+1}}{16} \right\rfloor \bmod 8), && \nu_X = \mathcal{X}_{l_X}\left(\mathcal{E}^{-1}_{l_X}(\zeta_{\imath+2\cdots+l_X})\right) \\ &\text{let } l_Y = \min(4, \max(0, \ell - l_X - 1)), && \nu_Y = \mathcal{X}_{l_Y}\left(\mathcal{E}^{-1}_{l_Y}(\zeta_{\imath+2+l_X\cdots+l_Y})\right) \end{aligned} \tag{A.26}$$

$\zeta_\imath$	Name	Mutations
70	store_imm_ind_u8	$\mu'^{\circlearrowleft}_{\varphi_A+\nu_X} = \nu_Y \bmod 2^8$
71	store_imm_ind_u16	$\mu'^{\circlearrowleft}_{\varphi_A+\nu_X\cdots+2} = \mathcal{E}_2\big(\nu_Y \bmod 2^{16}\big)$
72	store_imm_ind_u32	$\mu'^{\circlearrowleft}_{\varphi_A+\nu_X\cdots+4} = \mathcal{E}_4\big(\nu_Y \bmod 2^{32}\big)$
73	store_imm_ind_u64	$\mu'^{\circlearrowleft}_{\varphi_A+\nu_X\cdots+8} = \mathcal{E}_8(\nu_Y)$

A.5.8. *Instructions with Arguments of One Register, One Immediate and One Offset.*

$$\begin{aligned}
&\text{let } r_A = \min(12, \zeta_{\imath+1} \bmod 16)\,, && \varphi_A \equiv \varphi_{r_A}\,, \quad \varphi'_A \equiv \varphi'_{r_A}\\
&\text{let } l_X = \min(4, \left\lfloor \frac{\zeta_{\imath+1}}{16} \right\rfloor \bmod 8)\,, && \nu_X = \mathcal{X}_{l_X}\big(\mathcal{E}^{-1}_{l_X}(\zeta_{\imath+2\cdots+l_X})\big)\\
&\text{let } l_Y = \min(4, \max(0, \ell - l_X - 1))\,, && \nu_Y = \imath + \mathcal{Z}_{l_Y}(\mathcal{E}^{-1}_{l_Y}(\zeta_{\imath+2+l_X\cdots+l_Y}))
\end{aligned} \tag{A.27}$$

$\zeta_\imath$	Name	Mutations
80	load_imm_jump	$\texttt{branch}(\nu_Y, \top)\,, \quad \varphi'_A = \nu_X$
81	branch_eq_imm	$\texttt{branch}(\nu_Y, \varphi_A = \nu_X)$
82	branch_ne_imm	$\texttt{branch}(\nu_Y, \varphi_A \neq \nu_X)$
83	branch_lt_u_imm	$\texttt{branch}(\nu_Y, \varphi_A < \nu_X)$
84	branch_le_u_imm	$\texttt{branch}(\nu_Y, \varphi_A \leq \nu_X)$
85	branch_ge_u_imm	$\texttt{branch}(\nu_Y, \varphi_A \geq \nu_X)$
86	branch_gt_u_imm	$\texttt{branch}(\nu_Y, \varphi_A > \nu_X)$
87	branch_lt_s_imm	$\texttt{branch}(\nu_Y, \mathcal{Z}_8(\varphi_A) < \mathcal{Z}_8(\nu_X))$
88	branch_le_s_imm	$\texttt{branch}(\nu_Y, \mathcal{Z}_8(\varphi_A) \leq \mathcal{Z}_8(\nu_X))$
89	branch_ge_s_imm	$\texttt{branch}(\nu_Y, \mathcal{Z}_8(\varphi_A) \geq \mathcal{Z}_8(\nu_X))$
90	branch_gt_s_imm	$\texttt{branch}(\nu_Y, \mathcal{Z}_8(\varphi_A) > \mathcal{Z}_8(\nu_X))$

A.5.9. *Instructions with Arguments of Two Registers.*

$$\begin{aligned}
&\text{let } r_D = \min(12, (\zeta_{\imath+1}) \bmod 16)\,, && \varphi_D \equiv \varphi_{r_D}\,, \quad \varphi'_D \equiv \varphi'_{r_D}\\
&\text{let } r_A = \min(12, \left\lfloor \frac{\zeta_{\imath+1}}{16} \right\rfloor)\,, && \varphi_A \equiv \varphi_{r_A}\,, \quad \varphi'_A \equiv \varphi'_{r_A}
\end{aligned} \tag{A.28}$$

$\zeta_\imath$	Name	Mutations
100	move_reg	$\varphi'_D = \varphi_A$
101	sbrk	$\varphi'_D \equiv \min(x \in \mathbb{N}_R)$: $x \geq h$, $\mathbb{N}_{x\cdots+\varphi_A} \notin \mathbb{V}_\mu$, $\mathbb{N}_{x\cdots+\varphi_A} \subseteq \mathbb{V}^*_{\mu'}$
102	count_set_bits_64	$\varphi'_D = \sum_{i=0}^{63} \mathcal{B}_8(\varphi_A)_i$
103	count_set_bits_32	$\varphi'_D = \sum_{i=0}^{31} \mathcal{B}_4(\varphi_A \bmod 2^{32})_i$
104	leading_zero_bits_64	$\varphi'_D = \max(n \in \mathbb{N}_{65})$ where $\sum_{i=0}^{i<n} \overleftarrow{\mathcal{B}}_8(\varphi_A)_i = 0$
105	leading_zero_bits_32	$\varphi'_D = \max(n \in \mathbb{N}_{33})$ where $\sum_{i=0}^{i<n} \overleftarrow{\mathcal{B}}_4(\varphi_A \bmod 2^{32})_i = 0$
106	trailing_zero_bits_64	$\varphi'_D = \max(n \in \mathbb{N}_{65})$ where $\sum_{i=0}^{i<n} \mathcal{B}_8(\varphi_A)_i = 0$
107	trailing_zero_bits_32	$\varphi'_D = \max(n \in \mathbb{N}_{33})$ where $\sum_{i=0}^{i<n} \mathcal{B}_4(\varphi_A \bmod 2^{32})_i = 0$

$\zeta_\imath$	Name	Mutations
108	`sign_extend_8`	$\varphi'_D = \mathcal{Z}_8^{-1}(\mathcal{Z}_1(\varphi_A \bmod 2^8))$
109	`sign_extend_16`	$\varphi'_D = \mathcal{Z}_8^{-1}(\mathcal{Z}_2(\varphi_A \bmod 2^{16}))$
110	`zero_extend_16`	$\varphi'_D = \varphi_A \bmod 2^{16}$
111	`reverse_bytes`	$\forall i \in \mathbb{N}_8 : \mathcal{E}_8(\varphi'_D)_i = \mathcal{E}_8(\varphi_A)_{7-i}$

Note, the term h above refers to the beginning of the heap, the second major section of memory as defined in equation A.42 as $2Z_Z + Z(|\mathbf{o}|)$. If `sbrk` instruction is invoked on a PVM instance which does not have such a memory layout, then $h = 0$.

A.5.10. *Instructions with Arguments of Two Registers & One Immediate.*

$$\begin{aligned} &\text{let } r_A = \min(12, (\zeta_{\imath+1}) \bmod 16), && \varphi_A \equiv \varphi_{r_A}, \quad \varphi'_A \equiv \varphi'_{r_A} \\ &\text{let } r_B = \min(12, \left\lfloor \frac{\zeta_{\imath+1}}{16} \right\rfloor), && \varphi_B \equiv \varphi_{r_B}, \quad \varphi'_B \equiv \varphi'_{r_B} \\ &\text{let } l_X = \min(4, \max(0, \ell - 1)), && \nu_X \equiv \mathcal{X}_{l_X}\left(\mathcal{E}_{l_X}^{-1}(\zeta_{\imath+2\cdots+l_X})\right) \end{aligned} \tag{A.29}$$

$\zeta_\imath$	Name	Mutations
120	`store_ind_u8`	$\mu'^{\circlearrowleft}_{\varphi_B+\nu_X} = \varphi_A \bmod 2^8$
121	`store_ind_u16`	$\mu'^{\circlearrowleft}_{\varphi_B+\nu_X\cdots+2} = \mathcal{E}_2\left(\varphi_A \bmod 2^{16}\right)$
122	`store_ind_u32`	$\mu'^{\circlearrowleft}_{\varphi_B+\nu_X\cdots+4} = \mathcal{E}_4\left(\varphi_A \bmod 2^{32}\right)$
123	`store_ind_u64`	$\mu'^{\circlearrowleft}_{\varphi_B+\nu_X\cdots+8} = \mathcal{E}_8(\varphi_A)$
124	`load_ind_u8`	$\varphi'_A = \mu^{\circlearrowleft}_{\varphi_B+\nu_X}$
125	`load_ind_i8`	$\varphi'_A = \mathcal{Z}_8^{-1}(\mathcal{Z}_1(\mu^{\circlearrowleft}_{\varphi_B+\nu_X}))$
126	`load_ind_u16`	$\varphi'_A = \mathcal{E}_2^{-1}\left(\mu^{\circlearrowleft}_{\varphi_B+\nu_X\cdots+2}\right)$
127	`load_ind_i16`	$\varphi'_A = \mathcal{Z}_8^{-1}(\mathcal{Z}_2(\mathcal{E}_2^{-1}\left(\mu^{\circlearrowleft}_{\varphi_B+\nu_X\cdots+2}\right)))$
128	`load_ind_u32`	$\varphi'_A = \mathcal{E}_4^{-1}\left(\mu^{\circlearrowleft}_{\varphi_B+\nu_X\cdots+4}\right)$
129	`load_ind_i32`	$\varphi'_A = \mathcal{Z}_8^{-1}(\mathcal{Z}_4(\mathcal{E}_4^{-1}\left(\mu^{\circlearrowleft}_{\varphi_B+\nu_X\cdots+4}\right)))$
130	`load_ind_u64`	$\varphi'_A = \mathcal{E}_8^{-1}\left(\mu^{\circlearrowleft}_{\varphi_B+\nu_X\cdots+8}\right)$
131	`add_imm_32`	$\varphi'_A = \mathcal{X}_4\left((\varphi_B + \nu_X) \bmod 2^{32}\right)$
132	`and_imm`	$\forall i \in \mathbb{N}_{64} : \mathcal{B}_8(\varphi'_A)_i = \mathcal{B}_8(\varphi_B)_i \wedge \mathcal{B}_8(\nu_X)_i$
133	`xor_imm`	$\forall i \in \mathbb{N}_{64} : \mathcal{B}_8(\varphi'_A)_i = \mathcal{B}_8(\varphi_B)_i \oplus \mathcal{B}_8(\nu_X)_i$
134	`or_imm`	$\forall i \in \mathbb{N}_{64} : \mathcal{B}_8(\varphi'_A)_i = \mathcal{B}_8(\varphi_B)_i \vee \mathcal{B}_8(\nu_X)_i$
135	`mul_imm_32`	$\varphi'_A = \mathcal{X}_4\left((\varphi_B \cdot \nu_X) \bmod 2^{32}\right)$
136	`set_lt_u_imm`	$\varphi'_A = \varphi_B < \nu_X$
137	`set_lt_s_imm`	$\varphi'_A = \mathcal{Z}_8(\varphi_B) < \mathcal{Z}_8(\nu_X)$
138	`shlo_l_imm_32`	$\varphi'_A = \mathcal{X}_4\left((\varphi_B \cdot 2^{\nu_X \bmod 32}) \bmod 2^{32}\right)$
139	`shlo_r_imm_32`	$\varphi'_A = \mathcal{X}_4\left(\left\lfloor \varphi_B \bmod 2^{32} \div 2^{\nu_X \bmod 32} \right\rfloor\right)$
140	`shar_r_imm_32`	$\varphi'_A = \mathcal{Z}_8^{-1}(\left\lfloor \mathcal{Z}_4(\varphi_B \bmod 2^{32}) \div 2^{\nu_X \bmod 32} \right\rfloor)$
141	`neg_add_imm_32`	$\varphi'_A = \mathcal{X}_4\left((\nu_X + 2^{32} - \varphi_B) \bmod 2^{32}\right)$
142	`set_gt_u_imm`	$\varphi'_A = \varphi_B > \nu_X$
143	`set_gt_s_imm`	$\varphi'_A = \mathcal{Z}_8(\varphi_B) > \mathcal{Z}_8(\nu_X)$
144	`shlo_l_imm_alt_32`	$\varphi'_A = \mathcal{X}_4\left((\nu_X \cdot 2^{\varphi_B \bmod 32}) \bmod 2^{32}\right)$
145	`shlo_r_imm_alt_32`	$\varphi'_A = \mathcal{X}_4\left(\left\lfloor \nu_X \bmod 2^{32} \div 2^{\varphi_B \bmod 32} \right\rfloor\right)$
146	`shar_r_imm_alt_32`	$\varphi'_A = \mathcal{Z}_8^{-1}(\left\lfloor \mathcal{Z}_4(\nu_X \bmod 2^{32}) \div 2^{\varphi_B \bmod 32} \right\rfloor)$

$\zeta_\imath$	Name	Mutations
147	cmov_iz_imm	$\varphi'_A = \begin{cases}\nu_X & \text{if } \varphi_B = 0\\ \varphi_A & \text{otherwise}\end{cases}$
148	cmov_nz_imm	$\varphi'_A = \begin{cases}\nu_X & \text{if } \varphi_B \neq 0\\ \varphi_A & \text{otherwise}\end{cases}$
149	add_imm_64	$\varphi'_A = (\varphi_B + \nu_X) \bmod 2^{64}$
150	mul_imm_64	$\varphi'_A = (\varphi_B \cdot \nu_X) \bmod 2^{64}$
151	shlo_l_imm_64	$\varphi'_A = \mathcal{X}_8\big((\varphi_B \cdot 2^{\nu_X \bmod 64}) \bmod 2^{64}\big)$
152	shlo_r_imm_64	$\varphi'_A = \mathcal{X}_8\big(\lfloor \varphi_B \div 2^{\nu_X \bmod 64} \rfloor\big)$
153	shar_r_imm_64	$\varphi'_A = \mathcal{Z}_8^{-1}(\lfloor \mathcal{Z}_8(\varphi_B) \div 2^{\nu_X \bmod 64} \rfloor)$
154	neg_add_imm_64	$\varphi'_A = (\nu_X + 2^{64} - \varphi_B) \bmod 2^{64}$
155	shlo_l_imm_alt_64	$\varphi'_A = (\nu_X \cdot 2^{\varphi_B \bmod 64}) \bmod 2^{64}$
156	shlo_r_imm_alt_64	$\varphi'_A = \lfloor \nu_X \div 2^{\varphi_B \bmod 64} \rfloor$
157	shar_r_imm_alt_64	$\varphi'_A = \mathcal{Z}_8^{-1}(\lfloor \mathcal{Z}_8(\nu_X) \div 2^{\varphi_B \bmod 64} \rfloor)$
158	rot_r_64_imm	$\forall i \in \mathbb{N}_{64} : \mathcal{B}_8(\varphi'_A)_i = \mathcal{B}_8(\varphi_B)_{(i+\nu_X) \bmod 64}$
159	rot_r_64_imm_alt	$\forall i \in \mathbb{N}_{64} : \mathcal{B}_8(\varphi'_A)_i = \mathcal{B}_8(\nu_X)_{(i+\varphi_B) \bmod 64}$
160	rot_r_32_imm	$\varphi'_A = \mathcal{X}_4(x)$ where $x \in \mathbb{N}_{2^{32}}, \forall i \in \mathbb{N}_{32} : \mathcal{B}_4(x)_i = \mathcal{B}_4(\varphi_B)_{(i+\nu_X) \bmod 32}$
161	rot_r_32_imm_alt	$\varphi'_A = \mathcal{X}_4(x)$ where $x \in \mathbb{N}_{2^{32}}, \forall i \in \mathbb{N}_{32} : \mathcal{B}_4(x)_i = \mathcal{B}_4(\nu_X)_{(i+\varphi_B) \bmod 32}$

A.5.11. *Instructions with Arguments of Two Registers & One Offset.*

$$\begin{aligned}
&\text{let } r_A = \min(12, (\zeta_{\imath+1}) \bmod 16)\,, && \varphi_A \equiv \varphi_{r_A}\,, \quad \varphi'_A \equiv \varphi'_{r_A}\\
&\text{let } r_B = \min(12, \left\lfloor \frac{\zeta_{\imath+1}}{16} \right\rfloor)\,, && \varphi_B \equiv \varphi_{r_B}\,, \quad \varphi'_B \equiv \varphi'_{r_B}\\
&\text{let } l_X = \min(4, \max(0, \ell - 1))\,, && \nu_X \equiv \imath + \mathcal{Z}_{l_X}(\mathcal{E}^{-1}_{l_X}(\zeta_{\imath+2\cdots+l_X}))
\end{aligned} \tag{A.30}$$

$\zeta_\imath$	Name	Mutations
170	branch_eq	$\texttt{branch}(\nu_X, \varphi_A = \varphi_B)$
171	branch_ne	$\texttt{branch}(\nu_X, \varphi_A \neq \varphi_B)$
172	branch_lt_u	$\texttt{branch}(\nu_X, \varphi_A < \varphi_B)$
173	branch_lt_s	$\texttt{branch}(\nu_X, \mathcal{Z}_8(\varphi_A) < \mathcal{Z}_8(\varphi_B))$
174	branch_ge_u	$\texttt{branch}(\nu_X, \varphi_A \geq \varphi_B)$
175	branch_ge_s	$\texttt{branch}(\nu_X, \mathcal{Z}_8(\varphi_A) \geq \mathcal{Z}_8(\varphi_B))$

A.5.12. *Instruction with Arguments of Two Registers and Two Immediates.*

$$\begin{aligned}
&\text{let } r_A = \min(12, (\zeta_{\imath+1}) \bmod 16)\,, && \varphi_A \equiv \varphi_{r_A}\,, \quad \varphi'_A \equiv \varphi'_{r_A}\\
&\text{let } r_B = \min(12, \left\lfloor \frac{\zeta_{\imath+1}}{16} \right\rfloor)\,, && \varphi_B \equiv \varphi_{r_B}\,, \quad \varphi'_B \equiv \varphi'_{r_B}\\
&\text{let } l_X = \min(4, \zeta_{\imath+2} \bmod 8)\,, && \nu_X = \mathcal{X}_{l_X}\big(\mathcal{E}^{-1}_{l_X}(\zeta_{\imath+3\cdots+l_X})\big)\\
&\text{let } l_Y = \min(4, \max(0, \ell - l_X - 2))\,, && \nu_Y = \mathcal{X}_{l_Y}\big(\mathcal{E}^{-1}_{l_Y}(\zeta_{\imath+3+l_X\cdots+l_Y})\big)
\end{aligned} \tag{A.31}$$

$\zeta_\imath$	Name	Mutations
180	load_imm_jump_ind	$\texttt{djump}((\varphi_B + \nu_Y) \bmod 2^{32})\,, \quad \varphi'_A = \nu_X$

A.5.13. *Instructions with Arguments of Three Registers.*

$$\begin{aligned}
&\text{let } r_A = \min(12, (\zeta_{\imath+1}) \bmod 16)\,, && \varphi_A \equiv \varphi_{r_A}\,, \quad \varphi'_A \equiv \varphi'_{r_A}\\
&\text{let } r_B = \min(12, \left\lfloor \frac{\zeta_{\imath+1}}{16} \right\rfloor)\,, && \varphi_B \equiv \varphi_{r_B}\,, \quad \varphi'_B \equiv \varphi'_{r_B}\\
&\text{let } r_D = \min(12, \zeta_{\imath+2})\,, && \varphi_D \equiv \varphi_{r_D}\,, \quad \varphi'_D \equiv \varphi'_{r_D}
\end{aligned} \tag{A.32}$$

$\zeta_\imath$	Name	Mutations
190	add_32	$\varphi'_D = \mathcal{X}_4\big((\varphi_A + \varphi_B) \bmod 2^{32}\big)$
191	sub_32	$\varphi'_D = \mathcal{X}_4\big((\varphi_A + 2^{32} - (\varphi_B \bmod 2^{32})) \bmod 2^{32}\big)$
192	mul_32	$\varphi'_D = \mathcal{X}_4\big((\varphi_A \cdot \varphi_B) \bmod 2^{32}\big)$
193	div_u_32	$\varphi'_D = \begin{cases} 2^{64}-1 & \text{if } \varphi_B \bmod 2^{32} = 0 \\ \mathcal{X}_4\big(\lfloor (\varphi_A \bmod 2^{32}) \div (\varphi_B \bmod 2^{32}) \rfloor\big) & \text{otherwise} \end{cases}$
194	div_s_32	$\varphi'_D = \begin{cases} 2^{64}-1 & \text{if } b = 0 \\ \mathcal{Z}_8^{-1}(a) & \text{if } a = -2^{31} \wedge b = -1 \\ \mathcal{Z}_8^{-1}(\text{rtz}(a \div b)) & \text{otherwise} \\ \quad \text{where } a = \mathcal{Z}_4(\varphi_A \bmod 2^{32}),\ b = \mathcal{Z}_4(\varphi_B \bmod 2^{32}) & \end{cases}$
195	rem_u_32	$\varphi'_D = \begin{cases} \mathcal{X}_4\big(\varphi_A \bmod 2^{32}\big) & \text{if } \varphi_B \bmod 2^{32} = 0 \\ \mathcal{X}_4\big((\varphi_A \bmod 2^{32}) \bmod (\varphi_B \bmod 2^{32})\big) & \text{otherwise} \end{cases}$
196	rem_s_32	$\varphi'_D = \begin{cases} 0 & \text{if } a = -2^{31} \wedge b = -1 \\ \mathcal{Z}_8^{-1}(\text{smod}(a,b)) & \text{otherwise} \\ \quad \text{where } a = \mathcal{Z}_4(\varphi_A \bmod 2^{32}),\ b = \mathcal{Z}_4(\varphi_B \bmod 2^{32}) & \end{cases}$
197	shlo_l_32	$\varphi'_D = \mathcal{X}_4\big((\varphi_A \cdot 2^{\varphi_B \bmod 32}) \bmod 2^{32}\big)$
198	shlo_r_32	$\varphi'_D = \mathcal{X}_4\big(\lfloor (\varphi_A \bmod 2^{32}) \div 2^{\varphi_B \bmod 32} \rfloor\big)$
199	shar_r_32	$\varphi'_D = \mathcal{Z}_8^{-1}\big(\lfloor \mathcal{Z}_4(\varphi_A \bmod 2^{32}) \div 2^{\varphi_B \bmod 32} \rfloor\big)$
200	add_64	$\varphi'_D = (\varphi_A + \varphi_B) \bmod 2^{64}$
201	sub_64	$\varphi'_D = (\varphi_A + 2^{64} - \varphi_B) \bmod 2^{64}$
202	mul_64	$\varphi'_D = (\varphi_A \cdot \varphi_B) \bmod 2^{64}$
203	div_u_64	$\varphi'_D = \begin{cases} 2^{64}-1 & \text{if } \varphi_B = 0 \\ \lfloor \varphi_A \div \varphi_B \rfloor & \text{otherwise} \end{cases}$
204	div_s_64	$\varphi'_D = \begin{cases} 2^{64}-1 & \text{if } \varphi_B = 0 \\ \varphi_A & \text{if } \mathcal{Z}_8(\varphi_A) = -2^{63} \wedge \mathcal{Z}_8(\varphi_B) = -1 \\ \mathcal{Z}_8^{-1}(\text{rtz}(\mathcal{Z}_8(\varphi_A) \div \mathcal{Z}_8(\varphi_B))) & \text{otherwise} \end{cases}$
205	rem_u_64	$\varphi'_D = \begin{cases} \varphi_A & \text{if } \varphi_B = 0 \\ \varphi_A \bmod \varphi_B & \text{otherwise} \end{cases}$
206	rem_s_64	$\varphi'_D = \begin{cases} 0 & \text{if } \mathcal{Z}_8(\varphi_A) = -2^{63} \wedge \mathcal{Z}_8(\varphi_B) = -1 \\ \mathcal{Z}_8^{-1}(\text{smod}(\mathcal{Z}_8(\varphi_A), \mathcal{Z}_8(\varphi_B))) & \text{otherwise} \end{cases}$
207	shlo_l_64	$\varphi'_D = (\varphi_A \cdot 2^{\varphi_B \bmod 64}) \bmod 2^{64}$
208	shlo_r_64	$\varphi'_D = \lfloor \varphi_A \div 2^{\varphi_B \bmod 64} \rfloor$
209	shar_r_64	$\varphi'_D = \mathcal{Z}_8^{-1}\big(\lfloor \mathcal{Z}_8(\varphi_A) \div 2^{\varphi_B \bmod 64} \rfloor\big)$
210	and	$\forall i \in \mathbb{N}_{64} : \mathcal{B}_8(\varphi'_D)_i = \mathcal{B}_8(\varphi_A)_i \wedge \mathcal{B}_8(\varphi_B)_i$
211	xor	$\forall i \in \mathbb{N}_{64} : \mathcal{B}_8(\varphi'_D)_i = \mathcal{B}_8(\varphi_A)_i \oplus \mathcal{B}_8(\varphi_B)_i$
212	or	$\forall i \in \mathbb{N}_{64} : \mathcal{B}_8(\varphi'_D)_i = \mathcal{B}_8(\varphi_A)_i \vee \mathcal{B}_8(\varphi_B)_i$
213	mul_upper_s_s	$\varphi'_D = \mathcal{Z}_8^{-1}\big(\lfloor (\mathcal{Z}_8(\varphi_A) \cdot \mathcal{Z}_8(\varphi_B)) \div 2^{64} \rfloor\big)$
214	mul_upper_u_u	$\varphi'_D = \lfloor (\varphi_A \cdot \varphi_B) \div 2^{64} \rfloor$
215	mul_upper_s_u	$\varphi'_D = \mathcal{Z}_8^{-1}\big(\lfloor (\mathcal{Z}_8(\varphi_A) \cdot \varphi_B) \div 2^{64} \rfloor\big)$
216	set_lt_u	$\varphi'_D = \varphi_A < \varphi_B$
217	set_lt_s	$\varphi'_D = \mathcal{Z}_8(\varphi_A) < \mathcal{Z}_8(\varphi_B)$
218	cmov_iz	$\varphi'_D = \begin{cases} \varphi_A & \text{if } \varphi_B = 0 \\ \varphi_D & \text{otherwise} \end{cases}$

$\zeta_\imath$	Name	Mutations
219	`cmov_nz`	$\varphi'_D = \begin{cases}\varphi_A & \text{if } \varphi_B \neq 0\\ \varphi_D & \text{otherwise}\end{cases}$
220	`rot_l_64`	$\forall i \in \mathbb{N}_{64} : \mathcal{B}_8(\varphi'_D)_{(i+\varphi_B) \bmod 64} = \mathcal{B}_8(\varphi_A)_i$
221	`rot_l_32`	$\varphi'_D = \mathcal{X}_4(x)$ where $x \in \mathbb{N}_{2^{32}}, \forall i \in \mathbb{N}_{32} : \mathcal{B}_4(x)_{(i+\varphi_B) \bmod 32} = \mathcal{B}_4(\varphi_A)_i$
222	`rot_r_64`	$\forall i \in \mathbb{N}_{64} : \mathcal{B}_8(\varphi'_D)_i = \mathcal{B}_8(\varphi_A)_{(i+\varphi_B) \bmod 64}$
223	`rot_r_32`	$\varphi'_D = \mathcal{X}_4(x)$ where $x \in \mathbb{N}_{2^{32}}, \forall i \in \mathbb{N}_{32} : \mathcal{B}_4(x)_i = \mathcal{B}_4(\varphi_A)_{(i+\varphi_B) \bmod 32}$
224	`and_inv`	$\forall i \in \mathbb{N}_{64} : \mathcal{B}_8(\varphi'_D)_i = \mathcal{B}_8(\varphi_A)_i \wedge \neg\mathcal{B}_8(\varphi_B)_i$
225	`or_inv`	$\forall i \in \mathbb{N}_{64} : \mathcal{B}_8(\varphi'_D)_i = \mathcal{B}_8(\varphi_A)_i \vee \neg\mathcal{B}_8(\varphi_B)_i$
226	`xnor`	$\forall i \in \mathbb{N}_{64} : \mathcal{B}_8(\varphi'_D)_i = \neg(\mathcal{B}_8(\varphi_A)_i \oplus \mathcal{B}_8(\varphi_B)_i)$
227	`max`	$\varphi'_D = \mathcal{Z}_8^{-1}(\max(\mathcal{Z}_8(\varphi_A), \mathcal{Z}_8(\varphi_B)))$
228	`max_u`	$\varphi'_D = \max(\varphi_A, \varphi_B)$
229	`min`	$\varphi'_D = \mathcal{Z}_8^{-1}(\min(\mathcal{Z}_8(\varphi_A), \mathcal{Z}_8(\varphi_B)))$
230	`min_u`	$\varphi'_D = \min(\varphi_A, \varphi_B)$

Note that the two signed modulo operations have an idiosyncratic definition, operating as the modulo of the absolute values, but with the sign of the numerator. Formally:

$$\text{smod}: \begin{cases}(\mathbb{Z},\mathbb{Z}) \to \mathbb{Z}\\ (a,b) \mapsto \begin{cases}a & \text{if } b = 0\\ \text{sgn}(a)\cdot(|a| \bmod |b|) & \text{otherwise}\end{cases}\end{cases} \tag{A.33}$$

Division operations always round their result towards zero. Formally:

$$\text{rtz}: \begin{cases}\mathbb{Z} \to \mathbb{Z}\\ x \mapsto \begin{cases}\lceil x \rceil & \text{if } x < 0\\ \lfloor x \rfloor & \text{otherwise}\end{cases}\end{cases} \tag{A.34}$$

A.6. **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:

$$\Psi_H : \begin{cases}\begin{pmatrix}\mathbb{B},\mathbb{N}_R,\mathbb{N}_G,[\mathbb{N}_R]_{13},\\ \mathbb{M},\Omega\langle X\rangle,X\end{pmatrix} \to \left(\{\lightning,\infty,\blacksquare\}\cup\{\boxminus\}\times\mathbb{N}_R,\mathbb{N}_R,\mathbb{Z}_G,[\mathbb{N}_R]_{13},\mathbb{M},X\right)\\ (\mathbf{c},\imath,\varrho,\varphi,\mu,f,\mathbf{x}) \mapsto \begin{cases}\text{let } (\varepsilon',\imath',\varrho',\varphi',\mu') = \Psi(\mathbf{c},\imath,\varrho,\varphi,\mu):\\ (\varepsilon',\imath',\varrho',\varphi',\mu',\mathbf{x}) & \text{if } \varepsilon' \in \{\blacksquare,\lightning,\infty\}\cup\{\boxminus\}\times\mathbb{N}_R\\ \Psi_H(\mathbf{c},\imath'',\varrho'',\varphi'',\mu'',f,\mathbf{x}'') & \text{if } \wedge\begin{cases}\varepsilon' = \hbar\times h\\ (\blacktriangleright,\varrho'',\varphi'',\mu'',\mathbf{x}'') = f(h,\varrho',\varphi',\mu',\mathbf{x})\end{cases}\\ \quad\text{where } \imath'' = \imath' + 1 + \text{skip}(\imath')\\ (\varepsilon'',\imath',\varrho'',\varphi'',\mu'',\mathbf{x}'') & \text{if } \wedge\begin{cases}\varepsilon' = \hbar\times h\\ (\varepsilon'',\varrho'',\varphi'',\mu'',\mathbf{x}'') = f(h,\varrho',\varphi',\mu',\mathbf{x})\\ \varepsilon'' \in \{\lightning,\blacksquare,\infty\}\end{cases}\end{cases}\end{cases} \tag{A.35}$$

$$\Omega\langle X\rangle \equiv (\mathbb{N},\mathbb{N}_G,[\mathbb{N}_R]_{13},\mathbb{M},X) \to \left(\{\blacktriangleright,\blacksquare,\lightning,\infty\},\mathbb{N}_G,[\mathbb{N}_R]_{13},\mathbb{M},X\right) \tag{A.36}$$

As with Φ, on exit the instruction counter references the instruction *which caused the exit* and the machine state is that prior to this instruction. Should the machine be invoked again using this instruction counter and code, then the same instruction which caused the exit would be executed on the proper (prior) machine state.

With Φ_H, host-calls (i.e. `ecalli` instructions) are in effect handled internally with the state-mutator function provided as an argument, preventing the possibility of the result being a host-call fault. Note that in the case of a successful host-call transition, we must provide the new instruction counter value $\imath''$ explicitly alongside the fresh posterior state for said instruction.

A.7. **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 that RAM has sections 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 section via which invocation-specific data may be passed (i.e. arguments). It thus makes sense to define this properly in a single initializer function. These sections are

quantized into *major zones*, and one major zone is always left unallocated between sections in order to reduce accidental overrun. Sections are padded with zeroes to the nearest PVM memory page boundary.

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 at program start. Given program blob $\mathbf{p}$ and argument data $\mathbf{a}$, we can decode the program code $\mathbf{c}$, registers φ, and RAM μ by invoking the standard initialization function $Y(\mathbf{p}, \mathbf{a})$:

$$Y: \begin{cases} (\mathbb{B}, \mathbb{B}_{:\mathsf{Z}_I}) \to (\mathbb{B}, [\![\mathbb{N}_R]\!]_{13}, \mathbb{M})? \\ \quad (\mathbf{p}, \mathbf{a}) \mapsto \begin{cases} (\mathbf{c}, \varphi, \mu) & \text{if } \exists!(\mathbf{c}, \mathbf{o}, \mathbf{w}, z, s) \text{ which satisfy equation A.38} \\ \varnothing & \text{otherwise} \end{cases} \end{cases} \tag{A.37}$$

With conditions:

$$\text{let } \mathcal{E}_3(|\mathbf{o}|) \frown \mathcal{E}_3(|\mathbf{w}|) \frown \mathcal{E}_2(z) \frown \mathcal{E}_3(s) \frown \mathbf{o} \frown \mathbf{w} \frown \mathcal{E}_4(|\mathbf{c}|) \frown \mathbf{c} = \mathbf{p} \tag{A.38}$$

$$\mathsf{Z}_Z = 2^{16}, \quad \mathsf{Z}_I = 2^{24} \tag{A.39}$$

$$\text{let } P(x \in \mathbb{N}) \equiv \mathsf{Z}_P \left\lceil \frac{x}{\mathsf{Z}_P} \right\rceil, \qquad Z(x \in \mathbb{N}) \equiv \mathsf{Z}_Z \left\lceil \frac{x}{\mathsf{Z}_Z} \right\rceil \tag{A.40}$$

$$5\mathsf{Z}_Z + Z(|\mathbf{o}|) + Z(|\mathbf{w}| + z\mathsf{Z}_P) + Z(s) + \mathsf{Z}_I \le 2^{32} \tag{A.41}$$

Thus, if the above conditions cannot be satisfied with unique values, then the result is $\varnothing$, otherwise it is a tuple of $\mathbf{c}$ as above and μ, φ such that:

$$\forall i \in \mathbb{N}_{2^{32}} : ((\mu_{\mathbf{v}})_i, (\mu_{\mathbf{a}})_{\lfloor i/\mathsf{Z}_P \rfloor}) = \begin{cases} (\mathbf{v} \vdots \mathbf{o}_{i-\mathsf{Z}_Z}, \mathbf{a} \vdots R) & \text{if } \mathsf{Z}_Z & \le i < & \mathsf{Z}_Z + |\mathbf{o}| \\ (0, R) & \text{if } \mathsf{Z}_Z + |\mathbf{o}| & \le i < & \mathsf{Z}_Z + P(|\mathbf{o}|) \\ (\mathbf{w}_{i-(2\mathsf{Z}_Z + Z(|\mathbf{o}|))}, W) & \text{if } 2\mathsf{Z}_Z + Z(|\mathbf{o}|) & \le i < & 2\mathsf{Z}_Z + Z(|\mathbf{o}|) + |\mathbf{w}| \\ (0, W) & \text{if } 2\mathsf{Z}_Z + Z(|\mathbf{o}|) + |\mathbf{w}| & \le i < & 2\mathsf{Z}_Z + Z(|\mathbf{o}|) + P(|\mathbf{w}|) + z\mathsf{Z}_P \\ (0, W) & \text{if } 2^{32} - 2\mathsf{Z}_Z - \mathsf{Z}_I - P(s) & \le i < & 2^{32} - 2\mathsf{Z}_Z - \mathsf{Z}_I \\ (\mathbf{a}_{i-(2^{32}-\mathsf{Z}_Z-\mathsf{Z}_I)}, R) & \text{if } 2^{32} - \mathsf{Z}_Z - \mathsf{Z}_I & \le i < & 2^{32} - \mathsf{Z}_Z - \mathsf{Z}_I + |\mathbf{a}| \\ (0, R) & \text{if } 2^{32} - \mathsf{Z}_Z - \mathsf{Z}_I + |\mathbf{a}| & \le i < & 2^{32} - \mathsf{Z}_Z - \mathsf{Z}_I + P(|\mathbf{a}|) \\ (0, \varnothing) & \text{otherwise} \end{cases} \tag{A.42}$$

$$\forall i \in \mathbb{N}_{13} : \varphi_i = \begin{cases} 2^{32} - 2^{16} & \text{if } i = 0 \\ 2^{32} - 2\mathsf{Z}_Z - \mathsf{Z}_I & \text{if } i = 1 \\ 2^{32} - \mathsf{Z}_Z - \mathsf{Z}_I & \text{if } i = 7 \\ |\mathbf{a}| & \text{if } i = 8 \\ 0 & \text{otherwise} \end{cases} \tag{A.43}$$

A.8. **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:

$$\Psi_M: \begin{cases} (\mathbb{B}, \mathbb{N}_R, \mathbb{N}_G, \mathbb{B}_{:\mathsf{Z}_I}, \Omega\langle X\rangle, X) \to \left(\mathbb{N}_G, \mathbb{B} \cup \left\{ \lightning, \infty \right\}, X\right) \\ \quad (\mathbf{p}, \imath, \varrho, \mathbf{a}, f, \mathbf{x}) \mapsto \begin{cases} (0, \lightning, \mathbf{x}) & \text{if } Y(\mathbf{p}, \mathbf{a}) = \varnothing \\ R(\varrho, \Psi_H(\mathbf{c}, \imath, \varrho, \varphi, \mu, f, \mathbf{x})) & \text{if } Y(\mathbf{p}, \mathbf{a}) = (\mathbf{c}, \varphi, \mu) \end{cases} \\ \qquad \text{where } R: \left(\varrho, \begin{pmatrix} \varepsilon, & \imath', & \varrho', \\ \varphi', & \mu', & \mathbf{x}' \end{pmatrix}\right) \mapsto \begin{cases} (u, \infty, \mathbf{x}') & \text{if } \varepsilon = \infty \\ \left(u, \mu'_{\varphi'_7 \cdots + \varphi'_8}, \mathbf{x}'\right) & \text{if } \varepsilon = \blacksquare \wedge \mathbb{N}_{\varphi'_7 \cdots + \varphi'_8} \subseteq \mathbb{V}_{\mu'} \\ (u, [], \mathbf{x}') & \text{if } \varepsilon = \blacksquare \wedge \mathbb{N}_{\varphi'_7 \cdots + \varphi'_8} \notin \mathbb{V}_{\mu'} \\ (u, \lightning, \mathbf{x}') & \text{otherwise} \end{cases} \\ \qquad\qquad \text{where } u = \varrho - \max(\varrho', 0) \end{cases} \tag{A.44}$$

Note that the first tuple item is the amount of gas consumed by the operation, but never greater than the amount of gas provided for the operation.

A.9. **Gas Cost Model.** The gas cost model for the PVM is a simplified model of a pipelined, out-of-order CPU microarchitecture, similar to the ones used by modern production-grade compilers to predict how much time a given piece of code will take. For each basic block in the program the model simulates its execution flow and computes the required number of virtual CPU cycles that would be needed to execute it.

The gas cost $\varrho_\imath$ for a given basic block starting at instruction opcode index $\imath \in \varpi$ is defined by the number of virtual CPU cycles as determined by the the gas cost model transition function, up until every instruction of the basic block it has ingested has been retired and the simulation has converged.

The initial state Ξ_0 of the simulation is defined as follows:

$$\Xi_0(\imath) = \begin{cases} \imath^{(0)} = \imath \\ \dot{c}^{(0)} = 0, \dot{n}^{(0)} = 0, \dot{d}^{(0)} = 4, \dot{e}^{(0)} = 5 \\ \vec{s}^{(0)} = \langle\,\rangle, \vec{c}^{(0)} = \langle\,\rangle, \vec{p}^{(0)} = \langle\,\rangle, \vec{r}^{(0)} = \langle\,\rangle, \vec{x}^{(0)} = \langle\,\rangle \\ \dot{x}^{(0)} = \{\mathrm{A} = 4, \mathrm{L} = 4, \mathrm{S} = 4, \mathrm{M} = 1, \mathrm{D} = 1\} \end{cases} \tag{A.45}$$

The state transition function of the simulation from step n to step $n+1$ is defined as follows:

$$\Xi_{n+1}(\Xi_n, \mathbf{c}, \mathbf{k}) = \begin{cases} \Xi'_{n+1}(\Xi_n, \mathbf{c}, \mathbf{k}) & \text{if } n = 0 \vee (\imath^{(n)} \neq \varnothing \wedge \check{d}(\mathbf{c}, \mathbf{k}, \imath^{(n)}) \leq \dot{d}^{(n)} \wedge |\vec{s}^{(n)}| < 32) \\ \Xi''_{n+1}(\Xi_n) & \text{if } \mathfrak{S}(\Xi_n) \neq \varnothing \wedge \dot{e}^{(n)} > 0 \\ \varnothing & \text{if } n \neq 0 \wedge |\vec{s}^{(n)}| = 0 \\ \Xi'''_{n+1}(\Xi_n) & \text{otherwise} \end{cases} \tag{A.46}$$

Assuming m such that $\Xi_{m+1} = \varnothing$ the final gas cost of a basic block starting at $\imath$ is defined as:

$$\varrho_\imath = \max(\dot{c}^{(m)} - 3, 1) \tag{A.47}$$

The state transition function Ξ'_{n+1} which decodes the instructions into a reorder buffer without triggering the virtual CPU pipeline simulation is defined as follows (those pieces of state which are not explicitly mentioned by the equations are assumed to be unchanged and omitted for clarity):

$$\Xi'_{n+1}(\Xi_n, \mathbf{c}, \mathbf{k}) = \begin{cases} \Xi^{\mathrm{mov}}_{n+1}(\Xi_n, \mathbf{c}, \mathbf{k}) & \text{if opcode}(\mathbf{c}, \mathbf{k}, \imath^{(n)}) = \texttt{move\_reg} \\ \Xi^{\mathrm{decode}}_{n+1}(\Xi_n, \mathbf{c}, \mathbf{k}) & \text{otherwise} \end{cases} \tag{A.48}$$

The `move_reg` instruction is special-cased to be handled by the frontend of our virtual CPU, without being added to the reorder buffer:

$$\Xi^{\mathrm{mov}}_{n+1}(\Xi_n, \mathbf{c}, \mathbf{k}) = \begin{cases} \imath^{(n+1)} = \imath^{(n)} + 1 + \mathrm{skip}(\imath^{(n)}) \\ \dot{d}^{(n+1)} = \dot{d}^{(n)} - 1 \\ j \in \mathbb{Z} \Rightarrow \vec{r}^{(n+1)}_j = \begin{cases} \vec{r}^{(n)}_j \cup \check{r}(\mathbf{c}, \mathbf{k}, \imath^{(n)}) & \text{if } \vec{r}^{(n)}_j \cap \check{s}(\mathbf{c}, \mathbf{k}, \imath^{(n)}) \neq \varnothing \\ \vec{r}^{(n)}_j \setminus \check{r}(\mathbf{c}, \mathbf{k}, \imath^{(n)}) & \text{otherwise} \end{cases} \end{cases} \tag{A.49}$$

Every other instruction is fully decoded and added to the reorder buffer as follows:

$$\Xi^{\mathrm{decode}}_{n+1}(\Xi_n, \mathbf{c}, \mathbf{k}) = \begin{cases} \imath^{(n+1)} = \begin{cases} \varnothing & \text{if opcode}(\mathbf{c}, \mathbf{k}, \imath^{(n)}) \in T \\ \imath^{(n)} + 1 + \mathrm{skip}(\imath^{(n)}) & \text{otherwise} \end{cases} \\ \dot{d}^{(n+1)} = \dot{d}^{(n)} - \check{d}(\mathbf{c}, \mathbf{k}, \imath^{(n)}) \\ \dot{n}^{(n+1)} = \dot{n}^{(n)} + 1 \\ \vec{s}^{(n+1)}_{\dot{n}^{(n)}} = 1 \\ \vec{c}^{(n+1)}_{\dot{n}^{(n)}} = \check{c}(\mathbf{c}, \mathbf{k}, \imath^{(n)}) \\ \vec{x}^{(n+1)}_{\dot{n}^{(n)}} = \check{x}(\mathbf{c}, \mathbf{k}, \imath^{(n)}) \\ \vec{p}^{(n+1)}_{\dot{n}^{(n)}} = \left\{ j \;\middle|\; \check{s}(\mathbf{c}, \mathbf{k}, \imath^{(n)}) \cap \vec{r}^{(n)}_j \neq \varnothing \right\} \\ j \in \mathbb{Z} \Rightarrow \vec{r}^{(n+1)}_j = \begin{cases} \check{r}(\mathbf{c}, \mathbf{k}, \imath^{(n)}) & \text{if } j = \dot{n}^{(n)} \\ \vec{r}^{(n)}_j \setminus \check{r}(\mathbf{c}, \mathbf{k}, \imath^{(n)}) & \text{otherwise} \end{cases} \end{cases} \tag{A.50}$$

The state transition function Ξ''_{n+1} which starts the execution of the next pending instruction is defined as follows:

$$\Xi''_{n+1}(\Xi_n) = \begin{cases} \vec{s}^{(n+1)}_{\mathfrak{S}(\Xi_n)} = 3 \\ \dot{x}^{(n+1)} = \dot{x}^{(n)} - \vec{x}^{(n)}_{\mathfrak{S}(\Xi_n)} \\ \dot{e}^{(n+1)} = \dot{e}^{(n)} - 1 \end{cases} \tag{A.51}$$

The function $\mathfrak{S}(\Xi_n)$ which checks which instruction inside of the reorder buffer is ready to start executing (and whether such an instruction even exists) is defined as follows:

$$\mathfrak{S}(\Xi_n) = \min(j \in \mathbb{Z} \mid \vec{s}^{(n)}_j = 2 \wedge \vec{x}^{(n)}_j \leq \dot{x}^{(n)} \wedge (\forall k \in \vec{p}^{(n)}_j \Rightarrow \vec{c}^{(n)}_k \leq 0)) \tag{A.52}$$

The state transition function Ξ'''_{n+1} which simulates the rest of the virtual CPU pipeline is defined as follows:

$$\Xi'''_{n+1}(\Xi_n) = \begin{cases} j \in \mathbb{Z} \Rightarrow \vec{s}^{(n+1)}_j = \begin{cases} \varnothing & \text{if } \forall k \in \mathbb{Z}, \quad 0 \le k \le j \Rightarrow \vec{s}^{(n)}_k = 4 \\ 2 & \text{if } \vec{s}^{(n)}_j = 1 \\ 4 & \text{if } \vec{s}^{(n)}_j = 3 \wedge \vec{c}^{(n)}_j = 0 \\ \vec{s}^{(n)}_j & \text{otherwise} \end{cases} \\ j \in \mathbb{Z} \Rightarrow \vec{c}^{(n+1)}_j = \begin{cases} \vec{c}^{(n)}_j - 1 & \text{if } \vec{s}^{(n)}_j = 3 \\ \vec{c}^{(n)}_j & \text{otherwise} \end{cases} \\ j \in \mathbb{Z} \Rightarrow \vec{r}^{(n+1)}_j = \begin{cases} \varnothing & \text{if } \vec{s}^{(n)}_j = 3 \wedge \vec{c}^{(n)}_j = 1 \\ \vec{r}^{(n)}_j & \text{otherwise} \end{cases} \\ \dot{x}^{(n+1)} = \dot{x}^{(n)} + \sum_{j \in \mathbb{Z} \Rightarrow \vec{s}^{(n)}_j = 3 \wedge \vec{c}^{(n)}_j = 1} \vec{x}^{(n)}_j \\ \dot{c}^{(n+1)} = \dot{c}^{(n)} + 1 \\ \dot{d}^{(n+1)} = 4 \\ \dot{e}^{(n+1)} = 5 \end{cases} \tag{A.53}$$

The function $\check{s}(\mathbf{c}, \mathbf{k}, \imath)$ returns a set of source registers read by a given instruction, and the function $\check{r}(\mathbf{c}, \mathbf{k}, \imath)$ returns a set of destination registers which are written by a given instruction, regardless of whether those registers would actually have been modified by that instruction when executed at runtime. `ecalli` is assumed to not read nor write to any registers in this model.

$\check{c}$ is the number of cycles a given instruction needs to finish execution, $\check{d}$ is the number of decoding slots necessary to decode it, and $\check{x}$ is the number of virtual CPU execution units required to start its execution.

A.10. **Gas Cost Tables.** For some of the instructions their cost depends on whether the destination register overlaps with any of the source registers:

$$\mathfrak{P}(a, b, \mathbf{c}, \mathbf{k}, \imath) = \begin{cases} a & \text{if } \check{s}(\mathbf{c}, \mathbf{k}, \imath) \cap \check{r}(\mathbf{c}, \mathbf{k}, \imath) \neq \varnothing \\ b & \text{otherwise} \end{cases} \tag{A.54}$$

For non-immediate shift and rotate instructions only the first source register matters:

$$\mathfrak{P}_S(a, b, \mathbf{c}, \mathbf{k}, \imath) = \begin{cases} a & \text{if } \varphi_A = \varphi_D \\ b & \text{otherwise} \end{cases} \tag{A.55}$$

The cost of memory accesses depends on the memory cache model used by the program:

$$\mathfrak{m} = \begin{cases} 25 & \text{if memory model is L2HIT} \\ 37 & \text{if memory model is L3HIT} \end{cases} \tag{A.56}$$

The cost of a branch depends on whether any of its targets (either the jump target or the implicit fallthrough) points to an instruction byte which is equal to the opcode for the `unlikely` or the `trap` instruction; formally:

$$\mathfrak{b}(\mathbf{c}, \mathbf{k}, \imath) = \begin{cases} 1 & \text{if } \{\zeta_{\imath+1+\text{skip}(\imath)}, \zeta_{\imath_{\text{target}}}\} \cap \{\texttt{unlikely}, \texttt{trap}\} \neq \varnothing \\ 20 & \text{otherwise} \end{cases} \tag{A.57}$$

Instruction	$\check{c}$	$\check{d}$	$\check{x}_\text{A}$	$\check{x}_\text{L}$	$\check{x}_\text{S}$	$\check{x}_\text{M}$	$\check{x}_\text{D}$
`move_reg`	0	1	0	0	0	0	0
`and`	1	$\mathfrak{P}(1,2)$	1	0	0	0	0
`xor`	1	$\mathfrak{P}(1,2)$	1	0	0	0	0
`or`	1	$\mathfrak{P}(1,2)$	1	0	0	0	0
`add_64`	1	$\mathfrak{P}(1,2)$	1	0	0	0	0
`sub_64`	1	$\mathfrak{P}(1,2)$	1	0	0	0	0
`add_32`	2	$\mathfrak{P}(2,3)$	1	0	0	0	0
`sub_32`	2	$\mathfrak{P}(2,3)$	1	0	0	0	0
`and_imm`	1	$\mathfrak{P}(1,2)$	1	0	0	0	0

Instruction	$\check{c}$	$\check{d}$	$\check{x}_{\mathrm{A}}$	$\check{x}_{\mathrm{L}}$	$\check{x}_{\mathrm{S}}$	$\check{x}_{\mathrm{M}}$	$\check{x}_{\mathrm{D}}$
xor_imm	1	$\mathfrak{P}(1,2)$	1	0	0	0	0
or_imm	1	$\mathfrak{P}(1,2)$	1	0	0	0	0
add_imm_64	1	$\mathfrak{P}(1,2)$	1	0	0	0	0
shlo_r_imm_64	1	$\mathfrak{P}(1,2)$	1	0	0	0	0
shar_r_imm_64	1	$\mathfrak{P}(1,2)$	1	0	0	0	0
shlo_l_imm_64	1	$\mathfrak{P}(1,2)$	1	0	0	0	0
rot_r_64_imm	1	$\mathfrak{P}(1,2)$	1	0	0	0	0
reverse_bytes	1	$\mathfrak{P}(1,2)$	1	0	0	0	0
add_imm_32	2	$\mathfrak{P}(2,3)$	1	0	0	0	0
shlo_r_imm_32	2	$\mathfrak{P}(2,3)$	1	0	0	0	0
shar_r_imm_32	2	$\mathfrak{P}(2,3)$	1	0	0	0	0
shlo_l_imm_32	2	$\mathfrak{P}(2,3)$	1	0	0	0	0
rot_r_32_imm	2	$\mathfrak{P}(2,3)$	1	0	0	0	0
count_set_bits_64	1	1	1	0	0	0	0
count_set_bits_32	1	1	1	0	0	0	0
leading_zero_bits_64	1	1	1	0	0	0	0
leading_zero_bits_32	1	1	1	0	0	0	0
sign_extend_8	1	1	1	0	0	0	0
sign_extend_16	1	1	1	0	0	0	0
zero_extend_16	1	1	1	0	0	0	0
trailing_zero_bits_64	2	1	2	0	0	0	0
trailing_zero_bits_32	2	1	2	0	0	0	0
shlo_l_64	1	$\mathfrak{P}_S(2,3)$	1	0	0	0	0
shlo_r_64	1	$\mathfrak{P}_S(2,3)$	1	0	0	0	0
shar_r_64	1	$\mathfrak{P}_S(2,3)$	1	0	0	0	0
rot_l_64	1	$\mathfrak{P}_S(2,3)$	1	0	0	0	0
rot_r_64	1	$\mathfrak{P}_S(2,3)$	1	0	0	0	0
shlo_l_32	2	$\mathfrak{P}_S(3,4)$	1	0	0	0	0
shlo_r_32	2	$\mathfrak{P}_S(3,4)$	1	0	0	0	0
shar_r_32	2	$\mathfrak{P}_S(3,4)$	1	0	0	0	0
rot_l_32	2	$\mathfrak{P}_S(3,4)$	1	0	0	0	0
rot_r_32	2	$\mathfrak{P}_S(3,4)$	1	0	0	0	0
shlo_l_imm_alt_64	1	3	1	0	0	0	0
shlo_r_imm_alt_64	1	3	1	0	0	0	0
shar_r_imm_alt_64	1	3	1	0	0	0	0
rot_r_64_imm_alt	1	3	1	0	0	0	0
shlo_l_imm_alt_32	2	4	1	0	0	0	0
shlo_r_imm_alt_32	2	4	1	0	0	0	0
shar_r_imm_alt_32	2	4	1	0	0	0	0
rot_r_32_imm_alt	2	4	1	0	0	0	0
set_lt_u	3	3	1	0	0	0	0
set_lt_s	3	3	1	0	0	0	0
set_lt_u_imm	3	3	1	0	0	0	0
set_lt_s_imm	3	3	1	0	0	0	0

Instruction	$\check{c}$	$\check{d}$	$\check{x}_{\mathrm{A}}$	$\check{x}_{\mathrm{L}}$	$\check{x}_{\mathrm{S}}$	$\check{x}_{\mathrm{M}}$	$\check{x}_{\mathrm{D}}$
set_gt_u_imm	3	3	1	0	0	0	0
set_gt_s_imm	3	3	1	0	0	0	0
cmov_iz	2	2	1	0	0	0	0
cmov_nz	2	2	1	0	0	0	0
cmov_iz_imm	2	3	1	0	0	0	0
cmov_nz_imm	2	3	1	0	0	0	0
max	3	$\mathfrak{P}(2,3)$	1	0	0	0	0
max_u	3	$\mathfrak{P}(2,3)$	1	0	0	0	0
min	3	$\mathfrak{P}(2,3)$	1	0	0	0	0
min_u	3	$\mathfrak{P}(2,3)$	1	0	0	0	0
load_ind_u8	$\mathfrak{m}$	1	1	1	0	0	0
load_ind_i8	$\mathfrak{m}$	1	1	1	0	0	0
load_ind_u16	$\mathfrak{m}$	1	1	1	0	0	0
load_ind_i16	$\mathfrak{m}$	1	1	1	0	0	0
load_ind_u32	$\mathfrak{m}$	1	1	1	0	0	0
load_ind_i32	$\mathfrak{m}$	1	1	1	0	0	0
load_ind_u64	$\mathfrak{m}$	1	1	1	0	0	0
load_u8	$\mathfrak{m}$	1	1	1	0	0	0
load_i8	$\mathfrak{m}$	1	1	1	0	0	0
load_u16	$\mathfrak{m}$	1	1	1	0	0	0
load_i16	$\mathfrak{m}$	1	1	1	0	0	0
load_u32	$\mathfrak{m}$	1	1	1	0	0	0
load_i32	$\mathfrak{m}$	1	1	1	0	0	0
load_u64	$\mathfrak{m}$	1	1	1	0	0	0
store_imm_ind_u8	25	1	1	0	1	0	0
store_imm_ind_u16	25	1	1	0	1	0	0
store_imm_ind_u32	25	1	1	0	1	0	0
store_imm_ind_u64	25	1	1	0	1	0	0
store_ind_u8	25	1	1	0	1	0	0
store_ind_u16	25	1	1	0	1	0	0
store_ind_u32	25	1	1	0	1	0	0
store_ind_u64	25	1	1	0	1	0	0
store_imm_u8	25	1	1	0	1	0	0
store_imm_u16	25	1	1	0	1	0	0
store_imm_u32	25	1	1	0	1	0	0
store_imm_u64	25	1	1	0	1	0	0
store_u8	25	1	1	0	1	0	0
store_u16	25	1	1	0	1	0	0
store_u32	25	1	1	0	1	0	0
store_u64	25	1	1	0	1	0	0
branch_eq	$\mathfrak{b}$	1	1	0	0	0	0
branch_ne	$\mathfrak{b}$	1	1	0	0	0	0
branch_lt_u	$\mathfrak{b}$	1	1	0	0	0	0
branch_lt_s	$\mathfrak{b}$	1	1	0	0	0	0

Instruction	$\check{c}$	$\check{d}$	$\check{x}_{\mathrm{A}}$	$\check{x}_{\mathrm{L}}$	$\check{x}_{\mathrm{S}}$	$\check{x}_{\mathrm{M}}$	$\check{x}_{\mathrm{D}}$
branch_ge_u	$\mathfrak{b}$	1	1	0	0	0	0
branch_ge_s	$\mathfrak{b}$	1	1	0	0	0	0
branch_eq_imm	$\mathfrak{b}$	1	1	0	0	0	0
branch_ne_imm	$\mathfrak{b}$	1	1	0	0	0	0
branch_lt_u_imm	$\mathfrak{b}$	1	1	0	0	0	0
branch_le_u_imm	$\mathfrak{b}$	1	1	0	0	0	0
branch_ge_u_imm	$\mathfrak{b}$	1	1	0	0	0	0
branch_gt_u_imm	$\mathfrak{b}$	1	1	0	0	0	0
branch_lt_s_imm	$\mathfrak{b}$	1	1	0	0	0	0
branch_le_s_imm	$\mathfrak{b}$	1	1	0	0	0	0
branch_ge_s_imm	$\mathfrak{b}$	1	1	0	0	0	0
branch_gt_s_imm	$\mathfrak{b}$	1	1	0	0	0	0
div_u_32	60	4	1	0	0	0	1
div_s_32	60	4	1	0	0	0	1
rem_u_32	60	4	1	0	0	0	1
rem_s_32	60	4	1	0	0	0	1
div_u_64	60	4	1	0	0	0	1
div_s_64	60	4	1	0	0	0	1
rem_u_64	60	4	1	0	0	0	1
rem_s_64	60	4	1	0	0	0	1
and_inv	2	3	1	0	0	0	0
or_inv	2	3	1	0	0	0	0
xnor	2	$\mathfrak{P}(2,3)$	1	0	0	0	0
neg_add_imm_64	2	3	1	0	0	0	0
neg_add_imm_32	3	4	1	0	0	0	0
load_imm	1	1	0	0	0	0	0
load_imm_64	1	2	0	0	0	0	0
mul_64	3	$\mathfrak{P}(1,2)$	1	0	0	1	0
mul_32	4	$\mathfrak{P}(2,3)$	1	0	0	1	0
mul_imm_64	3	$\mathfrak{P}(1,2)$	1	0	0	1	0
mul_imm_32	4	$\mathfrak{P}(2,3)$	1	0	0	1	0
mul_upper_s_s	4	4	1	0	0	1	0
mul_upper_u_u	4	4	1	0	0	1	0
mul_upper_s_u	6	4	1	0	0	1	0
trap	2	1	0	0	0	0	0
fallthrough	2	1	0	0	0	0	0
unlikely	40	1	0	0	0	0	0
jump	15	1	0	0	0	0	0
load_imm_jump	15	1	0	0	0	0	0
jump_ind	22	1	0	0	0	0	0
load_imm_jump_ind	22	1	0	0	0	0	0
ecalli	100	4	1	0	0	0	0
sbrk	100	4	1	0	0	0	0

Appendix B. Virtual Machine Invocations

We now define the three practical instances where we wish to invoke a PVM instance as part of the protocol. In general, we avoid introducing unbounded data as part of the basic invocation arguments in order to minimize the chance of an unexpectedly large RAM allocation, which could lead to gas inflation and unavoidable underflow. This makes for a more cumbersome interface, but one which is more predictable and easier to reason about.

B.1. Host-Call Result Constants.

$\texttt{NONE} = 2^{64} - 1$: The return value indicating an item does not exist.
$\texttt{WHAT} = 2^{64} - 2$: Name unknown.
$\texttt{OOB} = 2^{64} - 3$: The inner PVM memory index provided for reading/writing is not accessible.
$\texttt{WHO} = 2^{64} - 4$: Index unknown.
$\texttt{FULL} = 2^{64} - 5$: Storage full or resource already allocated.
$\texttt{CORE} = 2^{64} - 6$: Core index unknown.
$\texttt{CASH} = 2^{64} - 7$: Insufficient funds.
$\texttt{LOW} = 2^{64} - 8$: Gas limit too low.
$\texttt{HUH} = 2^{64} - 9$: The item is already solicited, cannot be forgotten or the operation is invalid due to privilege level.
$\texttt{OK} = 0$: The return value indicating general success.

Inner PVM invocations have their own set of result codes:

$\texttt{HALT} = 0$: The invocation completed and halted normally.
$\texttt{PANIC} = 1$: The invocation completed with a panic.
$\texttt{FAULT} = 2$: The invocation completed with a page fault.
$\texttt{HOST} = 3$: The invocation completed with a host-call fault.
$\texttt{OOG} = 4$: The invocation completed by running out of gas.

Note return codes for a host-call-request exit are any non-zero value less than $2^{64} - 13$.

B.2. **Is-Authorized Invocation.** The Is-Authorized invocation is the first and simplest of the four, being totally stateless. It provides only host-call functions for inspecting its environment and parameters. It accepts as arguments only the core on which it should be executed, c. Formally, it is defined as Ψ_I:

$$\Psi_I \colon \begin{cases} (\mathbb{P}, \mathbb{N}_C) \to (\mathbb{B} \cup \mathbb{E}, \mathbb{N}_G) \\ (\mathbf{p}, c) \mapsto \begin{cases} (\texttt{BAD}, 0) & \text{if } \mathbf{p_u} = \varnothing \\ (\texttt{BIG}, 0) & \text{otherwise if } |\mathbf{p_u}| > \mathsf{W}_A \\ (\mathbf{r}, u) & \text{otherwise} \\ \quad \text{where } (u, \mathbf{r}, \varnothing) = \Psi_M(\mathbf{p_u}, 0, \mathsf{G}_I, \mathcal{E}_2(c), F, \varnothing) \end{cases} \end{cases} \tag{B.1}$$

$$F \in \Omega\langle\{\}\rangle \colon (n, \varrho, \varphi, \mu) \mapsto \begin{cases} \Omega_G(\varrho, \varphi, \mu) & \text{if } n = \texttt{gas} \\ \Omega_Y(\varrho, \varphi, \mu, \mathbf{p}, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing) & \text{if } n = \texttt{fetch} \\ (\infty, \varrho', \varphi', \mu) & \text{otherwise if } \varrho' < 0 \\ (\blacktriangleright, \varrho', \varphi', \mu) & \text{otherwise} \\ \quad \text{where } \varphi' = \varphi \text{ except } \varphi'_7 = \texttt{WHAT} & \\ \quad \text{and } \varrho' = \varrho - 10 & \end{cases} \tag{B.2}$$

Note for the Is-Authorized host-call dispatch function F in equation B.2, 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 has no general access to the state of the JAM chain, with the slight exception being the ability to make a historical lookup. Beyond this it is able to create inner instances of the PVM and dictate pieces of data to export.

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). 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 D to have a safety margin of eight hours:

$$\mathsf{D} \equiv \mathsf{L} + 4,800 = 19,200 \tag{B.3}$$

The inner PVM invocation host-calls, meanwhile, depend on an integrated PVM type, which we shall denote $\mathbb{G}$. It holds some program code, instruction counter and RAM:

$$\mathbb{G} \equiv (\mathbf{p} \in \mathbb{B}, \mathbf{u} \in \mathbb{M}, i \in \mathbb{N}_R) \tag{B.4}$$

The Export host-call depends on two pieces of context; one sequence of segments (blobs of length W_G) to which it may append, and the other an argument passed to the invocation function to dictate the number of segments prior which may assumed to have already been appended. The latter value ensures that an accurate segment index can be provided to the caller.

Unlike the other invocation functions, the Refine invocation function implicitly draws upon some recent service account state item δ. The specific block from which this comes is not important, as long as it is no earlier than its work-package's

lookup-anchor block. It explicitly accepts the work-package p and the index of the work item to be refined, i together with the core which is doing the refining c. Additionally, the authorizer trace $\mathbf{r}$ is provided together with all work items' import segments $\bar{\mathbf{i}}$ and an export segment offset ς. It results in a tuple of some error $\mathbb{E}$ or the refinement output blob (signalling success), the export sequence in the case of success and the gas used in evaluation. Formally:

$$\Psi_R\colon \begin{cases} \left(\mathbb{N}_C, \mathbb{N}, \mathbb{P}, \mathbb{B}, \llbracket \llbracket \mathbb{J} \rrbracket \rrbracket, \mathbb{N}\right) \to (\mathbb{B} \cup \mathbb{E}, \llbracket \mathbb{J} \rrbracket, \mathbb{N}_G) \\ \left(c, i, p, \mathbf{r}, \bar{\mathbf{i}}, \varsigma\right) \mapsto \begin{cases} (\texttt{BAD}, [], 0) & \text{if } w_s \notin \mathcal{K}(\delta) \vee \Lambda(\delta[w_s], (p_{\mathbf{c}})_t, w_c) = \varnothing \\ (\texttt{BIG}, [], 0) & \text{otherwise if } |\Lambda(\delta[w_s], (p_{\mathbf{c}})_t, w_c)| > \mathsf{W}_C \\ & \text{otherwise:} \\ & \quad \text{let } \mathbf{a} = \mathcal{E}(c, i, w_s, \updownarrow w_{\mathbf{y}}, \mathcal{H}(p)),\ \mathcal{E}(\updownarrow \mathbf{z}, \mathbf{c}) = \Lambda(\delta[w_s], (p_{\mathbf{c}})_t, w_c) \\ & \quad \text{and } (u, \mathbf{o}, (\mathbf{m}, \mathbf{e})) = \Psi_M(\mathbf{c}, 0, w_g, \mathbf{a}, F, (\varnothing, [])) : \\ (\mathbf{o}, [], u) & \quad \text{if } \mathbf{o} \in \left\{\infty, \lightning\right\} \\ (\mathbf{o}, \mathbf{e}, u) & \quad \text{otherwise} \\ & \text{where } w = p_{\mathbf{w}}[i] \end{cases} \end{cases} \tag{B.5}$$

$$F \in \Omega\left\langle\left(\left\langle\mathbb{N} \to \mathbb{G}\right\rangle, \llbracket \mathbb{J} \rrbracket\right)\right\rangle\colon (n, \varrho, \varphi, \mu, (\mathbf{m}, \mathbf{e})) \mapsto \begin{cases} \Omega_G(\varrho, \varphi, \mu, (\mathbf{m}, \mathbf{e})) & \text{if } n = \texttt{gas} \\ \Omega_Y(\varrho, \varphi, \mu, p, \mathbb{H}_0, \mathbf{r}, i, \bar{\mathbf{i}}, \overline{\mathbf{x}}, \varnothing, (\mathbf{m}, \mathbf{e})) & \text{if } n = \texttt{fetch} \\ \Omega_H(\varrho, \varphi, \mu, (\mathbf{m}, \mathbf{e}), w_s, \delta, (p_{\mathbf{c}})_t) & \text{if } n = \texttt{historical\_lookup} \\ \Omega_E(\varrho, \varphi, \mu, (\mathbf{m}, \mathbf{e}), \varsigma) & \text{if } n = \texttt{export} \\ \Omega_M(\varrho, \varphi, \mu, (\mathbf{m}, \mathbf{e})) & \text{if } n = \texttt{machine} \\ \Omega_P(\varrho, \varphi, \mu, (\mathbf{m}, \mathbf{e})) & \text{if } n = \texttt{peek} \\ \Omega_O(\varrho, \varphi, \mu, (\mathbf{m}, \mathbf{e})) & \text{if } n = \texttt{poke} \\ \Omega_Z(\varrho, \varphi, \mu, (\mathbf{m}, \mathbf{e})) & \text{if } n = \texttt{pages} \\ \Omega_K(\varrho, \varphi, \mu, (\mathbf{m}, \mathbf{e})) & \text{if } n = \texttt{invoke} \\ \Omega_X(\varrho, \varphi, \mu, (\mathbf{m}, \mathbf{e})) & \text{if } n = \texttt{expunge} \\ (\infty, \varrho', \varphi', \mu) & \text{otherwise if } \varrho' < 0 \\ (\blacktriangleright, \varrho', \varphi', \mu) & \text{otherwise} \\ \quad \text{where } \varphi' = \varphi \text{ except } \varphi'_7 = \texttt{WHAT} & \\ \quad \text{and } \varrho' = \varrho - 10 & \\ \quad \text{and } \overline{\mathbf{x}} = [[\mathbf{x} \mid (\mathcal{H}(\mathbf{x}), |\mathbf{x}|) \lessdot \mathbf{w}_{\mathbf{x}}] \mid \mathbf{w} \lessdot p_{\mathbf{w}}] & \end{cases} \tag{B.6}$$

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 $\mathbb{L}$, and our invocation context to be a pair of these contexts, $\mathbb{L} \times \mathbb{L}$ (and thus for any value $\mathbf{x} \in \mathbb{L}$ there exists $\mathbf{x}^2 \in \mathbb{L} \times \mathbb{L}$), 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.

$$\mathbb{L} \equiv \left(s \in \mathbb{N}_S, \mathbf{e} \in \mathbb{S}, i \in \mathbb{N}_S, \mathbf{t} \in \llbracket \mathbb{X} \rrbracket, y \in \mathbb{H}?, \mathbf{p} \in \{\!\{ \left(\mathbb{N}_S, \mathbb{B}\right) \}\!\}\right) \tag{B.7}$$

$$\forall \mathbf{x} \in \mathbb{L} : \mathbf{x}_{\mathbf{s}} \equiv (\mathbf{x}_{\mathbf{e}})_{\mathbf{d}}[\mathbf{x}_s] \tag{B.8}$$

We define a convenience equivalence $\mathbf{x}_{\mathbf{s}}$ to easily denote the accumulating service account.

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).

We define Ψ_A, the Accumulation invocation function as:

$$\Psi_A\colon \begin{cases} \left(\mathbb{S}, \mathbb{N}_T, \mathbb{N}_S, \mathbb{N}_G, \llbracket \mathbb{I} \rrbracket\right) \to \mathbb{O} \\ (\mathbf{e}, t, s, g, \mathbf{i}) \mapsto \begin{cases} (\mathbf{e}\colon \mathbf{s}, \mathbf{t}\colon [], y\colon \varnothing, u\colon 0, \mathbf{p}\colon []) & \text{if } \mathbf{c} = \varnothing \vee |\mathbf{c}| > \mathsf{W}_C \\ C(\Psi_M(\mathbf{c}, 5, g, \mathcal{E}(t, s, |\mathbf{i}|), F, I(\mathbf{s}, s)^2)) & \text{otherwise} \\ \quad \text{where } \mathbf{c} = \mathbf{e}_{\mathbf{d}}[s]_{\mathbf{c}} & \\ \quad \text{and } \mathbf{s} = \mathbf{e} \text{ except } \mathbf{s}_{\mathbf{d}}[s]_b = \mathbf{e}_{\mathbf{d}}[s]_b + \sum_{r \in \mathbf{x}} r_a & \\ \quad \text{and } \mathbf{x} = [i \mid i \lessdot \mathbf{i}, i \in \mathbb{X}] & \end{cases} \end{cases} \tag{B.9}$$

$$I\colon \begin{cases} \left(\mathbb{S}, \mathbb{N}_S\right) \to \mathbb{L} \\ \quad (\mathbf{e}, s) \mapsto (s, \mathbf{e}, i, \mathbf{t}\colon [], y\colon \varnothing, \mathbf{p}\colon []) \\ \qquad \text{where } i = \text{check}((\mathcal{E}_4^{-1}\left(\mathcal{H}\left(\mathcal{E}\left(s, \eta'_0, \mathbf{H}_T\right)\right)\right)) \bmod (2^{32} - \mathsf{S} - 2^8)) + \mathsf{S}) \end{cases} \tag{B.10}$$

$$F \in \Omega\langle(\mathbb{L},\mathbb{L})\rangle\colon (n,\varrho,\varphi,\mu,(\mathbf{x},\mathbf{y})) \mapsto \begin{cases} \Omega_G(\varrho,\varphi,\mu,(\mathbf{x},\mathbf{y})) & \text{if } n = \texttt{gas} \\ \Omega_Y(\varrho,\varphi,\mu,\varnothing,\eta'_0,\varnothing,\varnothing,\varnothing,\varnothing,\mathbf{i},(\mathbf{x},\mathbf{y})) & \text{if } n = \texttt{fetch} \\ G(\Omega_R(\varrho,\varphi,\mu,\mathbf{x}_\mathbf{s},\mathbf{x}_s,(\mathbf{x}_\mathbf{e})_\mathbf{d}),(\mathbf{x},\mathbf{y})) & \text{if } n = \texttt{read} \\ G(\Omega_W(\varrho,\varphi,\mu,\mathbf{x}_\mathbf{s},\mathbf{x}_s),(\mathbf{x},\mathbf{y})) & \text{if } n = \texttt{write} \\ G(\Omega_L(\varrho,\varphi,\mu,\mathbf{x}_\mathbf{s},\mathbf{x}_s,(\mathbf{x}_\mathbf{e})_\mathbf{d}),(\mathbf{x},\mathbf{y})) & \text{if } n = \texttt{lookup} \\ G(\Omega_I(\varrho,\varphi,\mu,\mathbf{x}_s,(\mathbf{x}_\mathbf{e})_\mathbf{d}),(\mathbf{x},\mathbf{y})) & \text{if } n = \texttt{info} \\ \Omega_B(\varrho,\varphi,\mu,(\mathbf{x},\mathbf{y})) & \text{if } n = \texttt{bless} \\ \Omega_A(\varrho,\varphi,\mu,(\mathbf{x},\mathbf{y})) & \text{if } n = \texttt{assign} \\ \Omega_D(\varrho,\varphi,\mu,(\mathbf{x},\mathbf{y})) & \text{if } n = \texttt{designate} \\ \Omega_C(\varrho,\varphi,\mu,(\mathbf{x},\mathbf{y})) & \text{if } n = \texttt{checkpoint} \\ \Omega_N(\varrho,\varphi,\mu,(\mathbf{x},\mathbf{y}),\mathbf{H}_T) & \text{if } n = \texttt{new} \\ \Omega_U(\varrho,\varphi,\mu,(\mathbf{x},\mathbf{y})) & \text{if } n = \texttt{upgrade} \\ \Omega_T(\varrho,\varphi,\mu,(\mathbf{x},\mathbf{y})) & \text{if } n = \texttt{transfer} \\ \Omega_J(\varrho,\varphi,\mu,(\mathbf{x},\mathbf{y}),\mathbf{H}_T) & \text{if } n = \texttt{eject} \\ \Omega_Q(\varrho,\varphi,\mu,(\mathbf{x},\mathbf{y})) & \text{if } n = \texttt{query} \\ \Omega_S(\varrho,\varphi,\mu,(\mathbf{x},\mathbf{y}),\mathbf{H}_T) & \text{if } n = \texttt{solicit} \\ \Omega_F(\varrho,\varphi,\mu,(\mathbf{x},\mathbf{y}),\mathbf{H}_T) & \text{if } n = \texttt{forget} \\ \Omega_\aleph(\varrho,\varphi,\mu,(\mathbf{x},\mathbf{y})) & \text{if } n = \texttt{yield} \\ \Omega_\Upsilon(\varrho,\varphi,\mu,(\mathbf{x},\mathbf{y})) & \text{if } n = \texttt{provide} \\ (\infty,\varrho',\varphi',\mu,(\mathbf{x},\mathbf{y})) & \text{otherwise if } \varrho' < 0 \\ (\blacktriangleright,\varrho',\varphi',\mu,(\mathbf{x},\mathbf{y})) & \text{otherwise} \\ \quad \text{where } \varphi' = \varphi \text{ except } \varphi'_7 = \texttt{WHAT} & \\ \quad \text{and } \varrho' = \varrho - 10 & \end{cases} \tag{B.11}$$

$$G\colon \begin{cases} \big(\{\blacktriangleright,\blacksquare,\lightning,\infty\},\mathbb{N}_G,[\mathbb{N}_R]_{13},\mathbb{M},\mathbb{A}\big),(\mathbb{L},\mathbb{L})\big) \to \big(\{\blacktriangleright,\blacksquare,\lightning,\infty\},\mathbb{N}_G,[\mathbb{N}_R]_{13},\mathbb{M},(\mathbb{L},\mathbb{L})\big) \\ \qquad ((\varepsilon,\varrho,\varphi,\mu,\mathbf{s}),(\mathbf{x},\mathbf{y})) \mapsto \big(\varepsilon,\varrho,\varphi,\mu,\big(\mathbf{x}^*,\mathbf{y}\big)\big) \\ \qquad\qquad \text{where } \mathbf{x}^* = \mathbf{x} \text{ except } \mathbf{x}^*_\mathbf{s} = \mathbf{s} \end{cases} \tag{B.12}$$

$$C\colon \begin{cases} \big(\mathbb{N}_G,\mathbb{B}\cup\{\infty,\lightning\},(\mathbb{L},\mathbb{L})\big) \to \mathbb{O} \\ \qquad (u,\mathbf{o},(\mathbf{x},\mathbf{y})) \mapsto \begin{cases} (\mathbf{e}\colon \mathbf{y}_\mathbf{e},\mathbf{t}\colon \mathbf{y}_\mathbf{t},y\colon \mathbf{y}_y,u,\mathbf{p}\colon \mathbf{y}_\mathbf{p}) & \text{if } \mathbf{o} \in \{\infty,\lightning\} \\ \big(\mathbf{e}\colon \mathbf{x}_\mathbf{e},\mathbf{t}\colon \mathbf{x}_\mathbf{t},y\colon \mathbf{o},u,\mathbf{p}\colon (\mathbf{x},\mathbf{y})_\mathbf{p}\big) & \text{otherwise if } \mathbf{o} \in \mathbb{H} \\ (\mathbf{e}\colon \mathbf{x}_\mathbf{e},\mathbf{t}\colon \mathbf{x}_\mathbf{t},y\colon \mathbf{x}_y,u,\mathbf{p}\colon \mathbf{x}_\mathbf{p}) & \text{otherwise} \end{cases} \end{cases} \tag{B.13}$$

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 partial state along with a service account index to yield a mutator context such that no alterations to the given state are implied 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:

$$\text{check}(i \in \mathbb{N}_S) \equiv \begin{cases} i & \text{if } i \notin \mathcal{K}(\mathbf{e}_\mathbf{d}) \\ \text{check}((i - \mathsf{S} + 1) \bmod (2^{32} - 2^8 - \mathsf{S}) + \mathsf{S}) & \text{otherwise} \end{cases} \tag{B.14}$$

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. **General Functions.** We come now to defining the host functions which are utilized by the PVM invocations. Generally, these map some PVM state, including invocation context, possibly together with some additional parameters, to a new PVM state.

The general functions are all broadly of the form $\big(\varrho' \in \mathbb{Z}_G, \varphi' \in [\mathbb{N}_R]_{13}, \mu' \in \mathbb{M}\big) = \Omega_\square(\varrho \in \mathbb{N}_G, \varphi \in [\mathbb{N}_R]_{13}, \mu \in \mathbb{M})$. 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. Some, such as `write` mutate a service account and both accept and return some $\mathbf{s} \in \mathbb{A}$. Others are more general functions, such as `fetch` and

do not assume any context but have a parameter list suffixed with an ellipsis to denote that the context parameter may be taken and is provided transparently into its result. This allows it to be easily utilized in multiple PVM invocations.

Other than the gas-counter which is explicitly defined, elements of PVM state are each assumed to remain unchanged by the host-call unless explicitly specified.

(B.15) $$\varrho' \equiv \varrho - g$$

(B.16) $$\left(\varepsilon', \varphi', \mu', \mathbf{s}'\right) \equiv \begin{cases} (\infty, \varphi, \mu, \mathbf{s}) & \text{if } \varrho < g \\ (\blacktriangleright, \varphi, \mu, \mathbf{s}) \text{ except as indicated below} & \text{otherwise} \end{cases}$$

Function Identifier Gas usage	Mutations
$\Omega_G(\varrho, \varphi, \dots)$ $\mathtt{gas} = 0$ $g = 10$	$\varphi'_7 \equiv \varrho'$
$\Omega_Y(\varrho, \varphi, \mu, p, n, \mathbf{r}, i, \bar{\mathbf{i}}, \overline{\mathbf{x}}, \mathbf{i}, \dots)$ $\mathtt{fetch} = 1$ $g = 10$	$\text{let } \mathbf{v} = \begin{cases} \mathbf{c} & \text{if } \varphi_{10} = 0 \\ \quad \text{where } \mathbf{c} = \mathcal{E}\begin{pmatrix} \mathcal{E}_8(\mathsf{B}_I), \mathcal{E}_8(\mathsf{B}_L), \mathcal{E}_8(\mathsf{B}_S), \mathcal{E}_2(\mathsf{C}), \mathcal{E}_4(\mathsf{D}), \mathcal{E}_4(\mathsf{E}), \mathcal{E}_8(\mathsf{G}_A), \\ \mathcal{E}_8(\mathsf{G}_I), \mathcal{E}_8(\mathsf{G}_R), \mathcal{E}_8(\mathsf{G}_T), \mathcal{E}_2(\mathsf{H}), \mathcal{E}_2(\mathsf{I}), \mathcal{E}_2(\mathsf{J}), \mathcal{E}_2(\mathsf{K}), \\ \mathcal{E}_4(\mathsf{L}), \mathcal{E}_2(\mathsf{N}), \mathcal{E}_2(\mathsf{O}), \mathcal{E}_2(\mathsf{P}), \mathcal{E}_2(\mathsf{Q}), \mathcal{E}_2(\mathsf{R}), \mathcal{E}_2(\mathsf{T}), \mathcal{E}_2(\mathsf{U}), \\ \mathcal{E}_2(\mathsf{V}), \mathcal{E}_4(\mathsf{W}_A), \mathcal{E}_4(\mathsf{W}_B), \mathcal{E}_4(\mathsf{W}_C), \mathcal{E}_4(\mathsf{W}_E), \mathcal{E}_4(\mathsf{W}_M), \\ \mathcal{E}_4(\mathsf{W}_P), \mathcal{E}_4(\mathsf{W}_R), \mathcal{E}_4(\mathsf{W}_T), \mathcal{E}_4(\mathsf{W}_X), \mathcal{E}_4(\mathsf{Y}) \end{pmatrix} \\ n & \text{if } n \neq \varnothing \wedge \varphi_{10} = 1 \\ \mathbf{r} & \text{if } \mathbf{r} \neq \varnothing \wedge \varphi_{10} = 2 \\ \overline{\mathbf{x}}[\varphi_{11}]_{\varphi_{12}} & \text{if } \overline{\mathbf{x}} \neq \varnothing \wedge \varphi_{10} = 3 \wedge \varphi_{11} < |\overline{\mathbf{x}}| \wedge \varphi_{12} < |\overline{\mathbf{x}}[\varphi_{11}]| \\ \overline{\mathbf{x}}[i]_{\varphi_{11}} & \text{if } \overline{\mathbf{x}} \neq \varnothing \wedge i \neq \varnothing \wedge \varphi_{10} = 4 \wedge \varphi_{11} < |\overline{\mathbf{x}}[i]| \\ \bar{\mathbf{i}}[\varphi_{11}]_{\varphi_{12}} & \text{if } \bar{\mathbf{i}} \neq \varnothing \wedge \varphi_{10} = 5 \wedge \varphi_{11} < \left|\bar{\mathbf{i}}\right| \wedge \varphi_{12} < \left|\bar{\mathbf{i}}[\varphi_{11}]\right| \\ \bar{\mathbf{i}}[i]_{\varphi_{11}} & \text{if } \bar{\mathbf{i}} \neq \varnothing \wedge i \neq \varnothing \wedge \varphi_{10} = 6 \wedge \varphi_{11} < \left|\bar{\mathbf{i}}[i]\right| \\ \mathcal{E}(p) & \text{if } p \neq \varnothing \wedge \varphi_{10} = 7 \\ p_{\mathbf{f}} & \text{if } p \neq \varnothing \wedge \varphi_{10} = 8 \\ p_{\mathbf{j}} & \text{if } p \neq \varnothing \wedge \varphi_{10} = 9 \\ \mathcal{E}(p_{\mathbf{c}}) & \text{if } p \neq \varnothing \wedge \varphi_{10} = 10 \\ \mathcal{E}(\updownarrow[S(w) \mid w \lessdot p_{\mathbf{w}}]) & \text{if } p \neq \varnothing \wedge \varphi_{10} = 11 \\ S(p_{\mathbf{w}}[\varphi_{11}]) & \text{if } p \neq \varnothing \wedge \varphi_{10} = 12 \wedge \varphi_{11} < |p_{\mathbf{w}}| \\ \quad \text{where } S(w) \equiv \mathcal{E}(\mathcal{E}_4(w_s), w_c, \mathcal{E}_8(w_g, w_a), \mathcal{E}_2(w_e, |w_{\mathbf{i}}|, |w_{\mathbf{x}}|), \mathcal{E}_4(|w_{\mathbf{y}}|)) \\ p_{\mathbf{w}}[\varphi_{11}]_{\mathbf{y}} & \text{if } p \neq \varnothing \wedge \varphi_{10} = 13 \wedge \varphi_{11} < |p_{\mathbf{w}}| \\ \mathcal{E}(\updownarrow \mathbf{i}) & \text{if } \mathbf{i} \neq \varnothing \wedge \varphi_{10} = 14 \\ \mathcal{E}(\mathbf{i}[\varphi_{11}]) & \text{if } \mathbf{i} \neq \varnothing \wedge \varphi_{10} = 15 \wedge \varphi_{11} < |\mathbf{i}| \\ \varnothing & \text{otherwise} \end{cases}$ $\text{let } o = \varphi_7$ $\text{let } f = \min(\varphi_8, |\mathbf{v}|)$ $\text{let } l = \min(\varphi_9, |\mathbf{v}| - f)$ $\left(\varepsilon', \varphi'_7, \mu'_{o\dots+l}\right) \equiv \begin{cases} \left(\lightning, \varphi_7, \mu_{o\dots+l}\right) & \text{if } \mathbb{N}_{o\dots+l} \notin \mathbb{V}^*_\mu \\ (\blacktriangleright, \mathtt{NONE}, \mu_{o\dots+l}) & \text{otherwise if } \mathbf{v} = \varnothing \\ (\blacktriangleright, |\mathbf{v}|, \mathbf{v}_{f\dots+l}) & \text{otherwise} \end{cases}$

Function Identifier Gas usage	Mutations
$\Omega_L(\varrho, \varphi, \mu, \mathbf{s}, s, \mathbf{d})$ $\texttt{lookup} = 2$ $g = 10$	$\text{let } \mathbf{a} = \begin{cases} \mathbf{s} & \text{if } \varphi_7 \in \{ s, 2^{64}-1 \} \\ \mathbf{d}[\varphi_7] & \text{otherwise if } \varphi_7 \in \mathcal{K}(\mathbf{d}) \\ \varnothing & \text{otherwise} \end{cases}$ $\text{let } [h, o] = \varphi_{8\cdots+2}$ $\text{let } \mathbf{v} = \begin{cases} \nabla & \text{if } \mathbb{N}_{h\cdots+32} \notin \mathbb{V}_\mu \\ \varnothing & \text{otherwise if } \mathbf{a} = \varnothing \vee \mu_{h\cdots+32} \notin \mathcal{K}(\mathbf{a_p}) \\ \mathbf{a_p}[\mu_{h\cdots+32}] & \text{otherwise} \end{cases}$ $\text{let } f = \min(\varphi_{10}, \lvert\mathbf{v}\rvert)$ $\text{let } l = \min(\varphi_{11}, \lvert\mathbf{v}\rvert - f)$ $(\varepsilon', \varphi_7', \mu'_{o\cdots+l}) \equiv \begin{cases} (\lightning, \varphi_7, \mu_{o\cdots+l}) & \text{if } \mathbf{v} = \nabla \vee \mathbb{N}_{o\cdots+l} \notin \mathbb{V}^*_\mu \\ (\blacktriangleright, \texttt{NONE}, \mu_{o\cdots+l}) & \text{otherwise if } \mathbf{v} = \varnothing \\ (\blacktriangleright, \lvert\mathbf{v}\rvert, \mathbf{v}_{f\cdots+l}) & \text{otherwise} \end{cases}$
$\Omega_R(\varrho, \varphi, \mu, \mathbf{s}, s, \mathbf{d})$ $\texttt{read} = 3$ $g = 10$	$\text{let } s^* = \begin{cases} s & \text{if } \varphi_7 = 2^{64}-1 \\ \varphi_7 & \text{otherwise} \end{cases}$ $\text{let } \mathbf{a} = \begin{cases} \mathbf{s} & \text{if } s^* = s \\ \mathbf{d}[s^*] & \text{otherwise if } s^* \in \mathcal{K}(\mathbf{d}) \\ \varnothing & \text{otherwise} \end{cases}$ $\text{let } [k_O, k_Z, o] = \varphi_{8\cdots+3}$ $\text{let } \mathbf{v} = \begin{cases} \nabla & \text{if } \mathbb{N}_{k_O\cdots+k_Z} \notin \mathbb{V}_\mu \\ \mathbf{a_s}[\mathbf{k}] & \text{otherwise if } \mathbf{a} \neq \varnothing \wedge \mathbf{k} \in \mathcal{K}(\mathbf{a_s}), \text{ where } \mathbf{k} = \mu_{k_O\cdots+k_Z} \\ \varnothing & \text{otherwise} \end{cases}$ $\text{let } f = \min(\varphi_{11}, \lvert\mathbf{v}\rvert)$ $\text{let } l = \min(\varphi_{12}, \lvert\mathbf{v}\rvert - f)$ $(\varepsilon', \varphi_7', \mu'_{o\cdots+l}) \equiv \begin{cases} (\lightning, \varphi_7, \mu_{o\cdots+l}) & \text{if } \mathbf{v} = \nabla \vee \mathbb{N}_{o\cdots+l} \notin \mathbb{V}^*_\mu \\ (\blacktriangleright, \texttt{NONE}, \mu_{o\cdots+l}) & \text{otherwise if } \mathbf{v} = \varnothing \\ (\blacktriangleright, \lvert\mathbf{v}\rvert, \mathbf{v}_{f\cdots+l}) & \text{otherwise} \end{cases}$
$\Omega_W(\varrho, \varphi, \mu, \mathbf{s}, s)$ $\texttt{write} = 4$ $g = 10$	$\text{let } [k_O, k_Z, v_O, v_Z] = \varphi_{7\cdots+4}$ $\text{let } \mathbf{k} = \begin{cases} \mu_{k_O\cdots+k_Z} & \text{if } \mathbb{N}_{k_O\cdots+k_Z} \subseteq \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $\text{let } \mathbf{a} = \begin{cases} \mathbf{s}, \text{ except } \mathcal{K}(\mathbf{a_s}) = \mathcal{K}(\mathbf{a_s}) \setminus \{ k \} & \text{if } v_Z = 0 \\ \mathbf{s}, \text{ except } \mathbf{a_s}[\mathbf{k}] = \mu_{v_O\cdots+v_Z} & \text{otherwise if } \mathbb{N}_{v_O\cdots+v_Z} \subseteq \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $\text{let } l = \begin{cases} \lvert\mathbf{s_s}[k]\rvert & \text{if } \mathbf{k} \in \mathcal{K}(\mathbf{s_s}) \\ \texttt{NONE} & \text{otherwise} \end{cases}$ $(\varepsilon', \varphi_7', \mathbf{s}') \equiv \begin{cases} (\lightning, \varphi_7, \mathbf{s}) & \text{if } \mathbf{k} = \nabla \vee \mathbf{a} = \nabla \\ (\blacktriangleright, \texttt{FULL}, \mathbf{s}) & \text{otherwise if } \mathbf{a}_t > \mathbf{a}_b \\ (\blacktriangleright, l, \mathbf{a}) & \text{otherwise} \end{cases}$

Function Identifier Gas usage	Mutations
$\Omega_I(\varrho, \varphi, \mu, s, \mathbf{d})$ $\texttt{info} = 5$ $g = 10$	$\text{let } \mathbf{a} = \begin{cases} \mathbf{d}[s] & \text{if } \varphi_7 = 2^{64} - 1 \\ \mathbf{d}[\varphi_7] & \text{otherwise} \end{cases}$ $\text{let } o = \varphi_8$ $\text{let } \mathbf{v} = \begin{cases} \mathcal{E}(\mathbf{a}_c, \mathcal{E}_8(\mathbf{a}_b, \mathbf{a}_t, \mathbf{a}_g, \mathbf{a}_m, \mathbf{a}_o), \mathcal{E}_4(\mathbf{a}_i), \mathcal{E}_8(\mathbf{a}_f), \mathcal{E}_4(\mathbf{a}_r, \mathbf{a}_a, \mathbf{a}_p)) & \text{if } \mathbf{a} \neq \varnothing \\ \varnothing & \text{otherwise} \end{cases}$ $\text{let } f = \min(\varphi_9, \lvert\mathbf{v}\rvert)$ $\text{let } l = \min(\varphi_{10}, \lvert\mathbf{v}\rvert - f)$ $(\varepsilon', \varphi_7', \mu'_{o\cdots+l}) \equiv \begin{cases} (\lightning, \varphi_7, \mu_{o\cdots+l}) & \text{if } \mathbf{v} = \nabla \vee \mathbb{N}_{o\cdots+l} \notin \mathbb{V}^*_\mu \\ (\blacktriangleright, \texttt{NONE}, \mu_{o\cdots+l}) & \text{otherwise if } \mathbf{v} = \varnothing \\ (\blacktriangleright, \lvert\mathbf{v}\rvert, \mathbf{v}_{f\cdots+l}) & \text{otherwise} \end{cases}$

B.6. **Refine Functions.** These assume some refine context pair $(\mathbf{m}, \mathbf{e}) \in (\langle\mathbb{N} \to \mathbb{G}\rangle, [\![\mathbb{J}]\!])$, which are both initially empty. Other than the gas-counter which is explicitly defined, elements of PVM state are each assumed to remain unchanged by the host-call unless explicitly specified.

(B.17) $$\varrho' \equiv \varrho - g$$

(B.18) $$(\varepsilon', \varphi', \mu') \equiv \begin{cases} (\infty, \varphi, \mu) & \text{if } \varrho < g \\ (\blacktriangleright, \varphi, \mu) \text{ except as indicated below} & \text{otherwise} \end{cases}$$

Function Identifier Gas usage	Mutations
$\Omega_H(\varrho, \varphi, \mu, (\mathbf{m}, \mathbf{e}), s, \mathbf{d}, t)$ $\texttt{historical\_lookup} = 6$ $g = 10$	$\text{let } \mathbf{a} = \begin{cases} \mathbf{d}[s] & \text{if } \varphi_7 = 2^{64} - 1 \wedge s \in \mathcal{K}(\mathbf{d}) \\ \mathbf{d}[\varphi_7] & \text{if } \varphi_7 \in \mathcal{K}(\mathbf{d}) \\ \varnothing & \text{otherwise} \end{cases}$ $\text{let } [h, o] = \varphi_{8\cdots+2}$ $\text{let } \mathbf{v} = \begin{cases} \nabla & \text{if } \mathbb{N}_{h\cdots+32} \notin \mathbb{V}_\mu \\ \varnothing & \text{otherwise if } \mathbf{a} = \varnothing \\ \Lambda(\mathbf{a}, t, \mu_{h\cdots+32}) & \text{otherwise} \end{cases}$ $\text{let } f = \min(\varphi_{10}, \lvert\mathbf{v}\rvert)$ $\text{let } l = \min(\varphi_{11}, \lvert\mathbf{v}\rvert - f)$ $(\varepsilon', \varphi_7', \mu'_{o\cdots+l}) \equiv \begin{cases} (\lightning, \varphi_7, \mu_{o\cdots+l}) & \text{if } \mathbf{v} = \nabla \vee \mathbb{N}_{o\cdots+l} \notin \mathbb{V}^*_\mu \\ (\blacktriangleright, \texttt{NONE}, \mu_{o\cdots+l}) & \text{otherwise if } \mathbf{v} = \varnothing \\ (\blacktriangleright, \lvert\mathbf{v}\rvert, \mathbf{v}_{f\cdots+l}) & \text{otherwise} \end{cases}$
$\Omega_E(\varrho, \varphi, \mu, (\mathbf{m}, \mathbf{e}), \varsigma)$ $\texttt{export} = 7$ $g = 10$	$\text{let } p = \varphi_7$ $\text{let } z = \min(\varphi_8, \mathsf{W}_G)$ $\text{let } \mathbf{x} = \begin{cases} \mathcal{P}_{\mathsf{W}_G}(\mu_{p\cdots+z}) & \text{if } \mathbb{N}_{p\cdots+z} \subseteq \mathbb{V}_{[}\mu] \\ \nabla & \text{otherwise} \end{cases}$ $(\varepsilon', \varphi_7', \mathbf{e}') \equiv \begin{cases} (\lightning, \varphi_7, \mathbf{e}) & \text{if } \mathbf{x} = \nabla \\ (\blacktriangleright, \texttt{FULL}, \mathbf{e}) & \text{otherwise if } \varsigma + \lvert\mathbf{e}\rvert \geq \mathsf{W}_X \\ (\blacktriangleright, \varsigma + \lvert\mathbf{e}\rvert, \mathbf{e} \mathbin{+\!\!+} \mathbf{x}) & \text{otherwise} \end{cases}$

Function Identifier Gas usage	Mutations
$\Omega_M(\varrho, \varphi, \mu, (\mathbf{m}, \mathbf{e}))$ machine $= 8$ $g = 10$	let $[p_O, p_Z, i] = \varphi_{7\cdots+3}$ let $\mathbf{p} = \begin{cases} \mu_{p_O \cdots + p_Z} & \text{if } \mathbb{N}_{p_O \cdots + p_Z} \subseteq \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ let $n = \min(n \in \mathbb{N}, n \notin \mathcal{K}(\mathbf{m}))$ let $\mathbf{u} = (\mathbf{v}\colon [0, 0, \dots], \mathbf{a}\colon [\varnothing, \varnothing, \dots])$ $(\varepsilon', \varphi_7', \mathbf{m}) \equiv \begin{cases} (\lightning, \varphi_7, \mathbf{m}) & \text{if } \mathbf{p} = \nabla \\ (\blacktriangleright, \text{HUH}, \mathbf{m}) & \text{otherwise if deblob}(\mathbf{p}) = \nabla \\ (\blacktriangleright, n, \mathbf{m} \cup \{ (n \mapsto (\mathbf{p}, \mathbf{u}, i)) \}) & \text{otherwise} \end{cases}$
$\Omega_P(\varrho, \varphi, \mu, (\mathbf{m}, \mathbf{e}))$ peek $= 9$ $g = 10$	let $[n, o, s, z] = \varphi_{7\cdots+4}$ $(\varepsilon', \varphi_7', \mu') \equiv \begin{cases} (\lightning, \varphi_7, \mu) & \text{if } \mathbb{N}_{o\cdots+z} \notin \mathbb{V}^*_{[\mu]} \\ (\blacktriangleright, \text{WHO}, \mu) & \text{otherwise if } n \notin \mathcal{K}(\mathbf{m}) \\ (\blacktriangleright, \text{OOB}, \mu) & \text{otherwise if } \mathbb{N}_{s\cdots+z} \notin \mathbb{V}_{\mathbf{m}[n]_\mathbf{u}} \\ (\blacktriangleright, \text{OK}, \mu') & \text{otherwise} \\ \quad \text{where } \mu' = \mu \text{ except } \mu_{o\cdots+z} = (\mathbf{m}[n]_\mathbf{u})_{s\cdots+z} \end{cases}$
$\Omega_O(\varrho, \varphi, \mu, (\mathbf{m}, \mathbf{e}))$ poke $= 10$ $g = 10$	let $[n, s, o, z] = \varphi_{7\cdots+4}$ $(\varepsilon', \varphi_7', \mathbf{m}') \equiv \begin{cases} (\lightning, \varphi_7, \mathbf{m}) & \text{if } \mathbb{N}_{s\cdots+z} \notin \mathbb{V}_{[\mu]} \\ (\blacktriangleright, \text{WHO}, \mathbf{m}) & \text{otherwise if } n \notin \mathcal{K}(\mathbf{m}) \\ (\blacktriangleright, \text{OOB}, \mathbf{m}) & \text{otherwise if } \mathbb{N}_{o\cdots+z} \notin \mathbb{V}^*_{\mathbf{m}[n]_\mathbf{u}} \\ (\blacktriangleright, \text{OK}, \mathbf{m}') & \text{otherwise} \\ \quad \text{where } \mathbf{m}' = \mathbf{m} \text{ except } (\mathbf{m}'[n]_\mathbf{u})_{o\cdots+z} = \mu_{s\cdots+z} \end{cases}$
$\Omega_Z(\varrho, \varphi, \mu, (\mathbf{m}, \mathbf{e}))$ pages $= 11$ $g = 10$	let $[n, p, c, r] = \varphi_{7\cdots+4}$ let $\mathbf{u} = \begin{cases} \mathbf{m}[n]_\mathbf{u} & \text{if } n \in \mathcal{K}(\mathbf{m}) \\ \nabla & \text{otherwise} \end{cases}$ let $\mathbf{u}' = \mathbf{u}$ except $\begin{cases} (\mathbf{u}'_\mathbf{v})_{p\mathsf{Z}_P \cdots + c\mathsf{Z}_P} = \begin{cases} [0, 0, \dots] & \text{if } r < 3 \\ (\mathbf{u}_\mathbf{v})_{p\mathsf{Z}_P \cdots + c\mathsf{Z}_P} & \text{otherwise} \end{cases} \\ (\mathbf{u}'_\mathbf{a})_{p\cdots+c} = \begin{cases} [\varnothing, \varnothing, \dots] & \text{if } r = 0 \\ [\text{R}, \text{R}, \dots] & \text{if } r = 1 \vee r = 3 \\ [\text{W}, \text{W}, \dots] & \text{if } r = 2 \vee r = 4 \end{cases} \end{cases}$ $(\varphi_7', \mathbf{m}') \equiv \begin{cases} (\text{WHO}, \mathbf{m}) & \text{if } \mathbf{u} = \nabla \\ (\text{HUH}, \mathbf{m}) & \text{otherwise if } r > 4 \vee p < 16 \vee p + c \geq {2^{32}}/{\mathsf{Z}_P} \\ (\text{HUH}, \mathbf{m}) & \text{otherwise if } r > 2 \wedge (\mathbf{u}_\mathbf{a})_{p\cdots+c} \ni \varnothing \\ (\text{OK}, \mathbf{m}') & \text{otherwise, where } \mathbf{m}' = \mathbf{m} \text{ except } \mathbf{m}'[n]_\mathbf{u} = \mathbf{u}' \end{cases}$

Function Identifier Gas usage	Mutations
$\Omega_K(\varrho, \varphi, \mu, (\mathbf{m}, \mathbf{e}))$ `invoke` $= 12$ $g = 10$	$\text{let } [n, o] = \varphi_{7,8}$ $\text{let } (g, \mathbf{w}) = \begin{cases} (g, \mathbf{w}) : \mathcal{E}_8(g) \frown \mathcal{E}_8(\mathbf{w}) = \mu_{o\cdots+112} & \text{if } \mathbb{N}_{o\cdots+112} \subseteq \mathbb{V}^*_\mu \\ (\nabla, \nabla) & \text{otherwise} \end{cases}$ $\text{let } \left(c, i', g', \mathbf{w}', \mathbf{u}'\right) = \Psi(\mathbf{m}[n]_{\mathbf{p}}, \mathbf{m}[n]_i, g, \mathbf{w}, \mathbf{m}[n]_{\mathbf{u}})$ $\text{let } \mu^* = \mu \text{ except } \mu^*_{o\cdots+112} = \mathcal{E}_8(g') \frown \mathcal{E}_8(\mathbf{w}')$ $\text{let } \mathbf{m}^* = \mathbf{m} \text{ except } \begin{cases} \mathbf{m}^*[n]_{\mathbf{u}} = \mathbf{u}' \\ \mathbf{m}^*[n]_i = \begin{cases} i' + \text{skip}(i') + 1 & \text{if } c \in \{\hbar\} \times \mathbb{N}_R \\ i' & \text{otherwise} \end{cases} \end{cases}$ $\left(\varepsilon', \varphi_7', \varphi_8', \mu', \mathbf{m}'\right) \equiv \begin{cases} \left(\lightning, \varphi_7, \varphi_8, \mu, \mathbf{m}\right) & \text{if } g = \nabla \\ (\blacktriangleright, \texttt{WHO}, \varphi_8, \mu, \mathbf{m}) & \text{otherwise if } n \notin \mathbf{m} \\ (\blacktriangleright, \texttt{HOST}, h, \mu^*, \mathbf{m}^*) & \text{otherwise if } c = \hbar \times h \\ (\blacktriangleright, \texttt{FAULT}, x, \mu^*, \mathbf{m}^*) & \text{otherwise if } c = \text{∃} \times x \\ (\blacktriangleright, \texttt{OOG}, \varphi_8, \mu^*, \mathbf{m}^*) & \text{otherwise if } c = \infty \\ (\blacktriangleright, \texttt{PANIC}, \varphi_8, \mu^*, \mathbf{m}^*) & \text{otherwise if } c = \lightning \\ (\blacktriangleright, \texttt{HALT}, \varphi_8, \mu^*, \mathbf{m}^*) & \text{otherwise if } c = \blacksquare \end{cases}$
$\Omega_X(\varrho, \varphi, \mu, (\mathbf{m}, \mathbf{e}))$ `expunge` $= 13$ $g = 10$	$\text{let } n = \varphi_7$ $\left(\varphi_7', \mathbf{m}'\right) \equiv \begin{cases} (\texttt{WHO}, \mathbf{m}) & \text{if } n \notin \mathcal{K}(\mathbf{m}) \\ \left(\mathbf{m}[n]_i, \mathbf{m} \setminus n\right) & \text{otherwise} \end{cases}$

B.7. **Accumulate Functions.** This defines a number of functions broadly of the form $(\varrho' \in \mathbb{Z}_G, \varphi' \in \llbracket \mathbb{N}_R \rrbracket_{13}, \mu', (\mathbf{x}', \mathbf{y}')) = \Omega_\square(\varrho \in \mathbb{N}_G, \varphi \in \llbracket \mathbb{N}_R \rrbracket_{13}, \mu \in \mathbb{M}, (\mathbf{x}, \mathbf{y}) \in \mathbb{L}^2, \ldots)$. 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.

Other than the gas-counter which is explicitly defined, elements of PVM state are each assumed to remain unchanged by the host-call unless explicitly specified.

$$\varrho' \equiv \varrho - g \tag{B.19}$$

$$\left(\varepsilon', \varphi', \mu', \mathbf{x}', \mathbf{y}'\right) \equiv \begin{cases} (\infty, \varphi, \mu, \mathbf{x}, \mathbf{y}) & \text{if } \varrho < g \\ (\blacktriangleright, \varphi, \mu, \mathbf{x}, \mathbf{y}) \text{ except as indicated below} & \text{otherwise} \end{cases} \tag{B.20}$$

Function Identifier Gas usage	Mutations
$\Omega_B(\varrho, \varphi, \mu, (\mathbf{x}, \mathbf{y}))$ `bless` $= 14$ $g = 10$	$\text{let } [m, a, v, r, o, n] = \varphi_{7\cdots+6}$ $\text{let } \mathbf{a} = \begin{cases} \mathcal{E}_4^{-1}(\mu_{a\cdots+4\mathsf{C}}) & \text{if } \mathbb{N}_{a\cdots+4\mathsf{C}} \subseteq \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $\text{let } \mathbf{z} = \begin{cases} \{\, (s \mapsto g) \text{ where } \mathcal{E}_4(s) \frown \mathcal{E}_8(g) = \mu_{o+12i\cdots+12} \mid i \in \mathbb{N}_n \,\} & \text{if } \mathbb{N}_{o\cdots+12n} \subseteq \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $\left(\varepsilon', \varphi_7', (\mathbf{x}'_{\mathbf{e}})_{(m,\mathbf{a},v,r,\mathbf{z})}\right) = \begin{cases} \left(\lightning, \varphi_7, (\mathbf{x}_{\mathbf{e}})_{(m,\mathbf{a},v,r,\mathbf{z})}\right) & \text{if } \{\mathbf{z}, \mathbf{a}\} \ni \nabla \\ \left(\blacktriangleright, \texttt{WHO}, (\mathbf{x}_{\mathbf{e}})_{(m,\mathbf{a},v,r,\mathbf{z})}\right) & \text{otherwise if } (m, v, r) \notin \mathbb{N}_S^3 \\ (\blacktriangleright, \texttt{OK}, (m, \mathbf{a}, v, r, \mathbf{z})) & \text{otherwise} \end{cases}$

Function Identifier Gas usage	Mutations
$\Omega_A(\varrho, \varphi, \mu, (\mathbf{x}, \mathbf{y}))$ $\texttt{assign} = 15$ $g = 10$	$\text{let } [c, o, a] = \varphi_{7\cdots+3}$ $\text{let } \mathbf{q} = \begin{cases} [\mu_{o+32i\cdots+32} \mid i \in \mathbb{N}_{\mathsf{Q}}] & \text{if } \mathbb{N}_{o\cdots+32\mathsf{Q}} \subseteq \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $\left(\varepsilon', \varphi_7', (\mathbf{x}'_\mathbf{e})_\mathbf{q}[c], (\mathbf{x}'_\mathbf{e})_\mathbf{a}[c]\right) = \begin{cases} (\lightning, \varphi_7, (\mathbf{x}_\mathbf{e})_\mathbf{q}[c], (\mathbf{x}_\mathbf{e})_\mathbf{a}[c]) & \text{if } \mathbf{q} = \nabla \\ (\blacktriangleright, \texttt{CORE}, (\mathbf{x}_\mathbf{e})_\mathbf{q}[c], (\mathbf{x}_\mathbf{e})_\mathbf{a}[c]) & \text{otherwise if } c \geq \mathsf{C} \\ (\blacktriangleright, \texttt{HUH}, (\mathbf{x}_\mathbf{e})_\mathbf{q}[c], (\mathbf{x}_\mathbf{e})_\mathbf{a}[c]) & \text{otherwise if } \mathbf{x}_s \neq (\mathbf{x}_\mathbf{e})_\mathbf{a}[c] \\ (\blacktriangleright, \texttt{WHO}, (\mathbf{x}_\mathbf{e})_\mathbf{q}[c], (\mathbf{x}_\mathbf{e})_\mathbf{a}[c]) & \text{otherwise if } a \notin \mathbb{N}_S \\ (\blacktriangleright, \texttt{OK}, \mathbf{q}, a) & \text{otherwise} \end{cases}$
$\Omega_D(\varrho, \varphi, \mu, (\mathbf{x}, \mathbf{y}))$ $\texttt{designate} = 16$ $g = 10$	$\text{let } o = \varphi_7$ $\text{let } \mathbf{v} = \begin{cases} [\mu_{o+336i\cdots+336} \mid i \in \mathbb{N}_{\mathsf{V}}] & \text{if } \mathbb{N}_{o\cdots+336\mathsf{V}} \subseteq \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $\left(\varepsilon', \varphi_7', (\mathbf{x}'_\mathbf{e})_\mathbf{i}\right) = \begin{cases} (\lightning, \varphi_7, (\mathbf{x}_\mathbf{e})_\mathbf{i}) & \text{if } \mathbf{v} = \nabla \\ (\blacktriangleright, \texttt{HUH}, (\mathbf{x}_\mathbf{e})_\mathbf{i}) & \text{otherwise if } \mathbf{x}_s \neq (\mathbf{x}_\mathbf{e})_v \\ (\blacktriangleright, \texttt{OK}, \mathbf{v}) & \text{otherwise} \end{cases}$
$\Omega_C(\varrho, \varphi, \mu, (\mathbf{x}, \mathbf{y}))$ $\texttt{checkpoint} = 17$ $g = 10$	$\mathbf{y}' \equiv \mathbf{x}$ $\varphi_7' \equiv \varrho'$
$\Omega_N(\varrho, \varphi, \mu, (\mathbf{x}, \mathbf{y}), t)$ $\texttt{new} = 18$ $g = 10$	$\text{let } [o, l, g, m, f, i] = \varphi_{7\cdots+6}$ $\text{let } c = \begin{cases} \mu_{o\cdots+32} & \text{if } \mathbb{N}_{o\cdots+32} \subseteq \mathbb{V}_\mu \wedge l \in \mathbb{N}_{2^{32}} \\ \nabla & \text{otherwise} \end{cases}$ $\text{let } \mathbf{a} \in \mathbb{A} \cup \{\nabla\} = \begin{cases} (c, \mathbf{s}\colon\{\}, \mathbf{l}\colon\{((c, l) \mapsto [])\}, b\colon \mathbf{a}_t, g, m, \mathbf{p}\colon\{\}, r\colon t, f, a\colon 0, p\colon \mathbf{x}_s) & \text{if } c \neq \nabla \\ \nabla & \text{otherwise} \end{cases}$ $\text{let } \mathbf{s} = \mathbf{x}_\mathbf{s} \text{ except } \mathbf{s}_b = (\mathbf{x}_\mathbf{s})_b - \mathbf{a}_t$ $\left(\varepsilon', \varphi_7', \mathbf{x}'_i, (\mathbf{x}'_\mathbf{e})_\mathbf{d}\right) \equiv \begin{cases} (\lightning, \varphi_7, \mathbf{x}_i, (\mathbf{x}_\mathbf{e})_\mathbf{d}) & \text{if } c = \nabla \\ (\blacktriangleright, \texttt{HUH}, \mathbf{x}_i, (\mathbf{x}_\mathbf{e})_\mathbf{d}) & \text{otherwise if } f \neq 0 \wedge \mathbf{x}_s \neq (\mathbf{x}_\mathbf{e})_m \\ (\blacktriangleright, \texttt{CASH}, \mathbf{x}_i, (\mathbf{x}_\mathbf{e})_\mathbf{d}) & \text{otherwise if } \mathbf{s}_b < (\mathbf{x}_\mathbf{s})_t \\ (\blacktriangleright, \texttt{FULL}, \mathbf{x}_i, (\mathbf{x}_\mathbf{e})_\mathbf{d}) & \text{otherwise if } \mathbf{x}_s = (\mathbf{x}_\mathbf{e})_r \wedge i < \mathsf{S} \wedge i \in \mathcal{K}((\mathbf{x}_\mathbf{e})_\mathbf{d}) \\ (\blacktriangleright, i, \mathbf{x}_i, (\mathbf{x}_\mathbf{e})_\mathbf{d} \cup \mathbf{d}) & \text{otherwise if } \mathbf{x}_s = (\mathbf{x}_\mathbf{e})_r \wedge i < \mathsf{S} \\ \quad \text{where } \mathbf{d} = \{(i \mapsto \mathbf{a}), (\mathbf{x}_s \mapsto \mathbf{s})\} & \\ (\blacktriangleright, \mathbf{x}_i, i^*, (\mathbf{x}_\mathbf{e})_\mathbf{d} \cup \mathbf{d}) & \text{otherwise} \\ \quad \text{where } i^* = \text{check}(\mathsf{S} + (\mathbf{x}_i - \mathsf{S} + 42) \bmod (2^{32} - \mathsf{S} - 2^8)) & \\ \quad \text{and } \mathbf{d} = \{(\mathbf{x}_i \mapsto \mathbf{a}), (\mathbf{x}_s \mapsto \mathbf{s})\} & \end{cases}$
$\Omega_U(\varrho, \varphi, \mu, (\mathbf{x}, \mathbf{y}))$ $\texttt{upgrade} = 19$ $g = 10$	$\text{let } [o, g, m] = \varphi_{7\cdots+3}$ $\text{let } c = \begin{cases} \mu_{o\cdots+32} & \text{if } \mathbb{N}_{o\cdots+32} \subseteq \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $\left(\varepsilon', \varphi_7', (\mathbf{x}'_\mathbf{s})_c, (\mathbf{x}'_\mathbf{s})_g, (\mathbf{x}'_\mathbf{s})_m\right) \equiv \begin{cases} (\lightning, \varphi_7, (\mathbf{x}_\mathbf{s})_c, (\mathbf{x}_\mathbf{s})_g, (\mathbf{x}_\mathbf{s})_m) & \text{if } c = \nabla \\ (\blacktriangleright, \texttt{OK}, c, g, m) & \text{otherwise} \end{cases}$

Function Identifier Gas usage	Mutations
$\Omega_T(\varrho, \varphi, \mu, (\mathbf{x}, \mathbf{y}))$ $\texttt{transfer} = 20$ $g = 10 + t$	$\text{let } [d, a, l, o] = \varphi_{7\cdots+4},$ $\text{let } \mathbf{d} = (\mathbf{x_e})_\mathbf{d}$ $\text{let } \mathbf{t} \in \mathbb{X} \cup \{\nabla\} = \begin{cases} (s\colon \mathbf{x}_s, d, a, m\colon \mu_{o\cdots+\mathsf{W}_T}, g\colon l) & \text{if } \mathbb{N}_{o\cdots+\mathsf{W}_T} \subseteq \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $\text{let } b = (\mathbf{x_s})_b - a$ $\text{let } (c, t) = \begin{cases} (\lightning, 0) & \text{if } \mathbf{t} = \nabla \\ (\texttt{WHO}, 0) & \text{otherwise if } d \notin \mathcal{K}(\mathbf{d}) \\ (\texttt{LOW}, 0) & \text{otherwise if } l < \mathbf{d}[d]_m \\ (\texttt{CASH}, 0) & \text{otherwise if } b < (\mathbf{x_s})_t \\ (\texttt{OK}, l) & \text{otherwise} \end{cases}$ $\left(\varepsilon', \varphi_7', \mathbf{x}_\mathbf{t}', (\mathbf{x}_\mathbf{s}')_b\right) \equiv \begin{cases} (\lightning, \varphi_7, \mathbf{x_t}, (\mathbf{x_s})_b) & \text{if } c = \lightning \\ (\blacktriangleright, c, \mathbf{x_t}, (\mathbf{x_s})_b) & \text{otherwise if } c \neq \texttt{OK} \\ (\blacktriangleright, \texttt{OK}, \mathbf{x_t} +\!\!+ \mathbf{t}, b) & \text{otherwise} \end{cases}$
$\Omega_J(\varrho, \varphi, \mu, (\mathbf{x}, \mathbf{y}), t)$ $\texttt{eject} = 21$ $g = 10$	$\text{let } [d, o] = \varphi_{7,8}$ $\text{let } h = \begin{cases} \mu_{o\cdots+32} & \text{if } \mathbb{N}_{o\cdots+32} \subseteq \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $\text{let } \mathbf{d} = \begin{cases} (\mathbf{x_e})_\mathbf{d}[d] & \text{if } d \neq \mathbf{x}_s \wedge d \in \mathcal{K}((\mathbf{x_e})_\mathbf{d}) \\ \nabla & \text{otherwise} \end{cases}$ $\text{let } l = \max(81, \mathbf{d}_o) - 81$ $\text{let } \mathbf{s}' = \mathbf{x_s} \text{ except } \mathbf{s}'_b = (\mathbf{x_s})_b + \mathbf{d}_b$ $\left(\varepsilon', \varphi_7', (\mathbf{x}_\mathbf{e}')_\mathbf{d}\right) \equiv \begin{cases} (\lightning, \varphi_7, (\mathbf{x_e})_\mathbf{d}) & \text{if } h = \nabla \\ (\blacktriangleright, \texttt{WHO}, (\mathbf{x_e})_\mathbf{d}) & \text{otherwise if } \mathbf{d} = \nabla \vee \mathbf{d}_c \neq \mathcal{E}_{32}(\mathbf{x}_s) \\ (\blacktriangleright, \texttt{HUH}, (\mathbf{x_e})_\mathbf{d}) & \text{otherwise if } \mathbf{d}_i \neq 2 \vee (h, l) \notin \mathbf{d_l} \\ (\blacktriangleright, \texttt{OK}, (\mathbf{x_e})_\mathbf{d} \setminus \{d\} \cup \{(\mathbf{x}_s \mapsto \mathbf{s}')\}) & \text{otherwise if } \mathbf{d_l}[h, l] = [x, y], y < t - \mathsf{D} \\ (\blacktriangleright, \texttt{HUH}, (\mathbf{x_e})_\mathbf{d}) & \text{otherwise} \end{cases}$
$\Omega_Q(\varrho, \varphi, \mu, (\mathbf{x}, \mathbf{y}))$ $\texttt{query} = 22$ $g = 10$	$\text{let } [o, z] = \varphi_{7,8}$ $\text{let } h = \begin{cases} \mu_{o\cdots+32} & \text{if } \mathbb{N}_{o\cdots+32} \subseteq \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $\text{let } \mathbf{a} = \begin{cases} (\mathbf{x_s})_\mathbf{l}[h, z] & \text{if } (h, z) \in \mathcal{K}((\mathbf{x_s})_\mathbf{l}) \\ \nabla & \text{otherwise} \end{cases}$ $\left(\varepsilon', \varphi_7', \varphi_8'\right) \equiv \begin{cases} (\lightning, \varphi_7, \varphi_8) & \text{if } h = \nabla \\ (\blacktriangleright, \texttt{NONE}, 0) & \text{otherwise if } \mathbf{a} = \nabla \\ (\blacktriangleright, 0, 0) & \text{otherwise if } \mathbf{a} = [] \\ (\blacktriangleright, 1 + 2^{32}x, 0) & \text{otherwise if } \mathbf{a} = [x] \\ (\blacktriangleright, 2 + 2^{32}x, y) & \text{otherwise if } \mathbf{a} = [x, y] \\ (\blacktriangleright, 3 + 2^{32}x, y + 2^{32}z) & \text{otherwise if } \mathbf{a} = [x, y, z] \end{cases}$

Function Identifier Gas usage	Mutations
$\Omega_S(\varrho, \varphi, \mu, (\mathbf{x}, \mathbf{y}), t)$ $\texttt{solicit} = 23$ $g = 10$	$\text{let } [o, z] = \varphi_{7,8}$ $\text{let } h = \begin{cases} \mu_{o\cdots+32} & \text{if } \mathbb{N}_{o\cdots+32} \subseteq \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $\text{let } \mathbf{a} = \begin{cases} \mathbf{x_s} \text{ except:} & \\ \quad \mathbf{a_l}[(h,z)] = [] & \text{if } h \neq \nabla \wedge (h,z) \notin \mathcal{K}((\mathbf{x_s})_\mathbf{l}) \\ \quad \mathbf{a_l}[(h,z)] = (\mathbf{x_s})_\mathbf{l}[(h,z)] \multimap t & \text{if } (\mathbf{x_s})_\mathbf{l}[(h,z)] = [x,y] \\ \nabla & \text{otherwise} \end{cases}$ $(\varepsilon', \varphi_7', \mathbf{x_s}') \equiv \begin{cases} (\lightning, \varphi_7, \mathbf{x_s}) & \text{if } h = \nabla \\ (\blacktriangleright, \texttt{HUH}, \mathbf{x_s}) & \text{otherwise if } \mathbf{a} = \nabla \\ (\blacktriangleright, \texttt{FULL}, \mathbf{x_s}) & \text{otherwise if } \mathbf{a}_b < \mathbf{a}_t \\ (\blacktriangleright, \texttt{OK}, \mathbf{a}) & \text{otherwise} \end{cases}$
$\Omega_F(\varrho, \varphi, \mu, (\mathbf{x}, \mathbf{y}), t)$ $\texttt{forget} = 24$ $g = 10$	$\text{let } [o, z] = \varphi_{7,8}$ $\text{let } h = \begin{cases} \mu_{o\cdots+32} & \text{if } \mathbb{N}_{o\cdots+32} \subseteq \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $\text{let } \mathbf{a} = \begin{cases} \mathbf{x_s} \text{ except:} & \\ \left.\begin{array}{l} \mathcal{K}(\mathbf{a_l}) = \mathcal{K}((\mathbf{x_s})_\mathbf{l}) \setminus \{(h,z)\}, \\ \mathcal{K}(\mathbf{a_p}) = \mathcal{K}((\mathbf{x_s})_\mathbf{p}) \setminus \{h\} \end{array}\right\} & \text{if } (\mathbf{x_s})_\mathbf{l}[h,z] \in \{[], [x,y]\},\ y < t - \mathsf{D} \\ \quad \mathbf{a_l}[h,z] = [x,t] & \text{if } (\mathbf{x_s})_\mathbf{l}[h,z] = [x] \\ \quad \mathbf{a_l}[h,z] = [w,t] & \text{if } (\mathbf{x_s})_\mathbf{l}[h,z] = [x,y,w],\ y < t - \mathsf{D} \\ \nabla & \text{otherwise} \end{cases}$ $(\varepsilon', \varphi_7', \mathbf{x_s}') \equiv \begin{cases} (\lightning, \varphi_7, \mathbf{x_s}) & \text{if } h = \nabla \\ (\blacktriangleright, \texttt{HUH}, \mathbf{x_s}) & \text{otherwise if } \mathbf{a} = \nabla \\ (\blacktriangleright, \texttt{OK}, \mathbf{a}) & \text{otherwise} \end{cases}$
$\Omega_\text{Y}(\varrho, \varphi, \mu, (\mathbf{x}, \mathbf{y}))$ $\texttt{yield} = 25$ $g = 10$	$\text{let } o = \varphi_7$ $\text{let } h = \begin{cases} \mu_{o\cdots+32} & \text{if } \mathbb{N}_{o\cdots+32} \subseteq \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $(\varepsilon', \varphi_7', \mathbf{x}_y') \equiv \begin{cases} (\lightning, \varphi_7, \mathbf{x}_y) & \text{if } h = \nabla \\ (\blacktriangleright, \texttt{OK}, h) & \text{otherwise} \end{cases}$
$\Omega_\text{P}(\varrho, \varphi, \mu, (\mathbf{x}, \mathbf{y}))$ $\texttt{provide} = 26$ $g = 10$	$\text{let } [o, z] = \varphi_{8,9}$ $\text{let } \mathbf{d} = (\mathbf{x_e})_\mathbf{d}$ $\text{let } s = \begin{cases} \mathbf{x}_s & \text{if } \varphi_7 = 2^{64} - 1 \\ \varphi_7 & \text{otherwise} \end{cases}$ $\text{let } \mathbf{i} = \begin{cases} \mu_{o\cdots+z} & \text{if } \mathbb{N}_{o\cdots+z} \subseteq \mathbb{V}_\mu \\ \nabla & \text{otherwise} \end{cases}$ $\text{let } \mathbf{a} = \begin{cases} \mathbf{d}[s] & \text{if } s \in \mathcal{K}(\mathbf{d}) \\ \varnothing & \text{otherwise} \end{cases}$ $(\varepsilon', \varphi_7', \mathbf{x_p}') \equiv \begin{cases} (\lightning, \varphi_7, \mathbf{x_p}) & \text{if } \mathbf{i} = \nabla \\ (\blacktriangleright, \texttt{WHO}, \mathbf{x_p}) & \text{otherwise if } \mathbf{a} = \varnothing \\ (\blacktriangleright, \texttt{HUH}, \mathbf{x_p}) & \text{otherwise if } \mathbf{a_l}[(\mathcal{H}(\mathbf{i}), z)] \neq [] \\ (\blacktriangleright, \texttt{HUH}, \mathbf{x_p}) & \text{otherwise if } (s, \mathbf{i}) \in \mathbf{x_p} \\ (\blacktriangleright, \texttt{OK}, \mathbf{x_p} \cup \{(s, \mathbf{i})\}) & \text{otherwise} \end{cases}$

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 $\mathcal{E}^{-1}$ 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:

$$\mathcal{E}(\varnothing) \equiv [] \tag{C.1}$$

We also define the serialization of an octet-sequence as itself:

$$\mathcal{E}(x \in \mathbb{B}) \equiv x \tag{C.2}$$

We define anonymous tuples to be encoded as the concatenation of their encoded elements:

$$\mathcal{E}((a, b, \dots)) \equiv \mathcal{E}(a) \frown \mathcal{E}(b) \frown \dots \tag{C.3}$$

Passing multiple arguments to the serialization functions is equivalent to passing a tuple of those arguments. Formally:

$$\mathcal{E}(a, b, \dots) \equiv \mathcal{E}((a, b, \dots)) \tag{C.4}$$

We define general natural number serialization, able to encode naturals of up to 2^{64}, as:

$$\mathcal{E}: \begin{cases} \mathbb{N}_{2^{64}} \to \mathbb{B}_{1:9} \\ \quad x \mapsto \begin{cases} [0] & \text{if } x = 0 \\ \left[2^8 - 2^{8-l} + \left\lfloor \frac{x}{2^{8l}} \right\rfloor\right] \frown \mathcal{E}_l\left(x \bmod 2^{8l}\right) & \text{if } \exists l \in \mathbb{N}_8 : 2^{7l} \le x < 2^{7(l+1)} \\ \left[2^8 - 1\right] \frown \mathcal{E}_8(x) & \text{otherwise if } x < 2^{64} \end{cases} \end{cases} \tag{C.5}$$

C.1.2. *Sequence Encoding.* We define the sequence serialization function $\mathcal{E}\left(\llbracket T \rrbracket\right)$ 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:

$$\mathcal{E}([\mathbf{i}_0, \mathbf{i}_1, ...]) \equiv \mathcal{E}(\mathbf{i}_0) \frown \mathcal{E}(\mathbf{i}_1) \frown \dots \tag{C.6}$$

Thus, conveniently, fixed length octet sequences (e.g. hashes $\mathbb{H}$ and its variants) have an identity serialization.

C.1.3. *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 natural and are encoded immediately prior to the item.

We generally use a *length discriminator* when serializing sequence terms which have variable length (e.g. general blobs $\mathbb{B}$ or unbound numeric sequences $\llbracket \mathbb{N} \rrbracket$) (though this is omitted in the case of fixed-length terms such as hashes $\mathbb{H}$).[19] In this case, we simply prefix the term its length prior to encoding. Thus, for some term $y \in (x \in \mathbb{B}, \dots)$, 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:

$$\updownarrow x \equiv (|x|, x) \text{ thus } \mathcal{E}(\updownarrow x) \equiv \mathcal{E}(|x|) \frown \mathcal{E}(x) \tag{C.7}$$

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 $S?$):

$$¿x \equiv \begin{cases} 0 & \text{if } x = \varnothing \\ (1, x) & \text{otherwise} \end{cases} \tag{C.8}$$

C.1.4. *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.

$$\mathcal{E}(b \in \mathbb{b}) \equiv \begin{cases} [] & \text{if } b = [] \\ \left[\sum_{i=0}^{i<\min(8,|b|)} b_i \cdot 2^i\right] \frown \mathcal{E}(b_{8\dots}) & \text{otherwise} \end{cases} \tag{C.9}$$

C.1.5. *Dictionary Encoding.* In general, dictionaries are placed in the Merkle trie directly (see appendix E for details). However, small dictionaries may reasonably be encoded as a sequence of pairs ordered by the key. Formally:

$$\forall K, V : \mathcal{E}(d \in \langle K \to V \rangle) \equiv \mathcal{E}(\updownarrow[(\mathcal{E}(k), \mathcal{E}(d[k])) \mid k \in \mathcal{K}(d) \setminus k]) \tag{C.10}$$

C.1.6. *Set Encoding.* For any values which are sets and don't already have a defined encoding above, we define the serialization of a set as the serialization of the set's elements in proper order. Formally:

$$\mathcal{E}(\{ a, b, c, \dots \}) \equiv \mathcal{E}(a) \frown \mathcal{E}(b) \frown \mathcal{E}(c) \frown \dots \text{ where } a < b < c < \dots \tag{C.11}$$

[19] 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.

C.1.7. *Fixed-length Integer Encoding.* We first define the trivial natural number serialization functions which are subscripted by the number of octets of the final sequence. Values are encoded in a regular little-endian fashion. This is utilized for almost all integer encoding across the protocol. Formally:

$$\mathcal{E}_{l\in\mathbb{N}}: \begin{cases} \mathbb{N}_{2^{8l}} \to \mathbb{B}_l \\ \quad x \mapsto \begin{cases} [] & \text{if } l = 0 \\ [x \bmod 256] \frown \mathcal{E}_{l-1}\left(\left\lfloor \frac{x}{256} \right\rfloor\right) & \text{otherwise} \end{cases} \end{cases} \tag{C.12}$$

For non-natural arguments, $\mathcal{E}_{l\in\mathbb{N}}$ corresponds to the definitions of $\mathcal{E}$, except that recursive elements are made as $\mathcal{E}_l$ rather than $\mathcal{E}$. Thus:

$$\mathcal{E}_{l\in\mathbb{N}}(a, b, \dots) \equiv \mathcal{E}_l((a, b, \dots)) \tag{C.13}$$

$$\mathcal{E}_{l\in\mathbb{N}}((a, b, \dots)) \equiv \mathcal{E}_l(a) \frown \mathcal{E}_l(b) \frown \dots \tag{C.14}$$

$$\mathcal{E}_{l\in\mathbb{N}}([\mathbf{i}_0, \mathbf{i}_1, \dots]) \equiv \mathcal{E}_l(\mathbf{i}_0) \frown \mathcal{E}_l(\mathbf{i}_1) \frown \dots \tag{C.15}$$

And so on.

C.2. **Block Serialization.** A block $\mathbf{B}$ is serialized as a tuple of its elements in regular order, as implied in equations 4.2, 4.3 and 5.1. For the header, we define both the regular serialization and the unsigned serialization $\mathcal{E}_U$. Formally:

$$\mathcal{E}(\mathbf{B}) = \mathcal{E}(\mathbf{H}, \mathcal{E}_T(\mathbf{E}_T), \mathcal{E}_P(\mathbf{E}_P), \mathcal{E}_G(\mathbf{E}_G), \mathcal{E}_A(\mathbf{E}_A), \mathcal{E}_D(\mathbf{E}_D)) \tag{C.16}$$

$$\mathcal{E}_T(\mathbf{E}_T) = \mathcal{E}(\updownarrow\mathbf{E}_T) \tag{C.17}$$

$$\mathcal{E}_P(\mathbf{E}_P) = \mathcal{E}(\updownarrow[(\mathcal{E}_4(s), \updownarrow\mathbf{d}) \mid (s, \mathbf{d}) \in \mathbf{E}_P]) \tag{C.18}$$

$$\mathcal{E}_G(\mathbf{E}_G) = \mathcal{E}(\updownarrow[(\mathbf{r}, \mathcal{E}_4(t), \updownarrow[(\mathcal{E}_2(v), s) \mid (v, s) \in a]) \mid (\mathbf{r}, t, a) \in \mathbf{E}_G]) \tag{C.19}$$

$$\mathcal{E}_A(\mathbf{E}_A) = \mathcal{E}(\updownarrow[(a, f, \mathcal{E}_2(v), s) \mid (a, f, v, s) \in \mathbf{E}_A]) \tag{C.20}$$

$$\mathcal{E}_D((\mathbf{v}, \mathbf{c}, \mathbf{f})) = \mathcal{E}(\updownarrow[(r, \mathcal{E}_4(a), [(v, \mathcal{E}_2(i), s) \mid (v, i, s) \in \mathbf{j}]) \mid (r, a, \mathbf{j}) \in \mathbf{v}], \updownarrow\mathbf{c}, \updownarrow\mathbf{f}) \tag{C.21}$$

$$\mathcal{E}(\mathbf{H}) = \mathcal{E}(\mathcal{E}_U(\mathbf{H}), \mathbf{H}_S) \tag{C.22}$$

$$\mathcal{E}_U(\mathbf{H}) = \mathcal{E}(\mathbf{H}_P, \mathbf{H}_R, \mathbf{H}_X, \mathcal{E}_4(\mathbf{H}_T), ¿\mathbf{H}_E, ¿\mathbf{H}_W, \mathcal{E}_2(\mathbf{H}_I), \mathbf{H}_V, \updownarrow\mathbf{H}_O) \tag{C.23}$$

$$\mathcal{E}(\mathbf{x} \in \mathbb{C}) \equiv \mathcal{E}(\mathbf{x}_a, \mathbf{x}_s, \mathbf{x}_b, \mathbf{x}_l, \mathcal{E}_4(\mathbf{x}_t), \updownarrow\mathbf{x_p}) \tag{C.24}$$

$$\mathcal{E}(\mathbf{x} \in \mathbb{Y}) \equiv \mathcal{E}(\mathbf{x}_p, \mathcal{E}_4(\mathbf{x}_l), \mathbf{x}_u, \mathbf{x}_e, \mathcal{E}_2(\mathbf{x}_n)) \tag{C.25}$$

$$\mathcal{E}(\mathbf{d} \in \mathbb{D}) \equiv \mathcal{E}(\mathcal{E}_4(\mathbf{d}_s), \mathbf{d}_c, \mathbf{d}_y, \mathcal{E}_8(\mathbf{d}_g), O(\mathbf{d_l}), \mathbf{d}_u, \mathbf{d}_i, \mathbf{d}_x, \mathbf{d}_z, \mathbf{d}_e) \tag{C.26}$$

$$\mathcal{E}(\mathbf{r} \in \mathbb{R}) \equiv \mathcal{E}(\mathbf{r_s}, \mathbf{r_c}, \mathbf{r}_c, \mathbf{r}_a, \mathbf{r}_g, \updownarrow\mathbf{r_t}, \updownarrow\mathbf{r_l}, \updownarrow\mathbf{r_d}) \tag{C.27}$$

$$\mathcal{E}(\mathbf{p} \in \mathbb{P}) \equiv \mathcal{E}(\mathcal{E}_4(\mathbf{p}_h), \mathbf{p}_u, \mathbf{p_c}, \updownarrow\mathbf{p_j}, \updownarrow\mathbf{p_f}, \updownarrow\mathbf{p_w}) \tag{C.28}$$

$$\mathcal{E}(\mathbf{w} \in \mathbb{W}) \equiv \mathcal{E}\Big(\mathcal{E}_4(\mathbf{w}_s), \mathbf{w}_c, \mathcal{E}_8(\mathbf{w}_g), \mathcal{E}_8(\mathbf{w}_a), \mathcal{E}_2(\mathbf{w}_e), \updownarrow\mathbf{w_y}, \updownarrow I^{\#}(\mathbf{w_i}), \updownarrow[(h, \mathcal{E}_4(i)) \mid (h, i) \in \mathbf{w_x}]\Big) \tag{C.29}$$

$$\mathcal{E}(x \in \mathbb{T}) \equiv \mathcal{E}(x_y, x_e) \tag{C.30}$$

$$\mathcal{E}_X(x \in \mathbb{X}) \equiv \mathcal{E}(\mathcal{E}_4(x_s), \mathcal{E}_4(x_d), \mathcal{E}_8(x_a), x_m, \mathcal{E}_8(x_g)) \tag{C.31}$$

$$\mathcal{E}_U(\mathbf{x} \in \mathbb{U}) \equiv \mathcal{E}(\mathbf{x}_p, \mathbf{x}_e, \mathbf{x}_a, \mathbf{x}_y, \mathbf{x}_g, O(\mathbf{x_l}), \updownarrow\mathbf{x_t}) \tag{C.32}$$

$$\mathcal{E}(\mathbf{x} \in \mathbb{I}) \equiv \begin{cases} \mathcal{E}(0, \mathcal{E}_U(o)) & \text{if } \mathbf{x} \in \mathbb{U} \\ \mathcal{E}(1, \mathcal{E}_X(o)) & \text{if } \mathbf{x} \in \mathbb{X} \end{cases} \tag{C.33}$$

$$O(o \in \mathbb{E} \cup \mathbb{B}) \equiv \begin{cases} (0, \updownarrow o) & \text{if } o \in \mathbb{B} \\ 1 & \text{if } o = \infty \\ 2 & \text{if } o = \lightning \\ 3 & \text{if } o = \circledcirc \\ 4 & \text{if } o = \ominus \\ 5 & \text{if } o = \texttt{BAD} \\ 6 & \text{if } o = \texttt{BIG} \end{cases} \tag{C.34}$$

$$I\big((h \in \mathbb{H} \cup \mathbb{H}^{\boxplus}, i \in \mathbb{N}_{2^{15}})\big) \equiv \begin{cases} (h, \mathcal{E}_2(i)) & \text{if } h \in \mathbb{H} \\ \big(r, \mathcal{E}_2\big(i + 2^{15}\big)\big) & \text{if } \exists r \in \mathbb{H}, h = r^{\boxplus} \end{cases} \tag{C.35}$$

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 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 31-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. **Serialization.** The serialization of state primarily involves placing all the various components of σ into a single mapping from 31-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:

$$C: \begin{cases} \mathbb{N}_{2^8} \cup (\mathbb{N}_{2^8}, \mathbb{N}_S) \cup (\mathbb{N}_S, \mathbb{B}) \to \mathbb{B}_{31} \\ \quad i \in \mathbb{N}_{2^8} \mapsto [i, 0, 0, \dots] \\ \quad (i, s \in \mathbb{N}_S) \mapsto [i, n_0, 0, n_1, 0, n_2, 0, n_3, 0, 0, \dots] \text{ where } n = \mathcal{E}_4(s) \\ \quad (s, h) \mapsto [n_0, a_0, n_1, a_1, n_2, a_2, n_3, a_3, a_4, a_5, \dots, a_{26}] \text{ where } n = \mathcal{E}_4(s), a = \mathcal{H}(h) \end{cases} \tag{D.1}$$

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:

(D.2)

$$T(\sigma) \equiv \begin{cases} C(1) \mapsto \mathcal{E}([\updownarrow x \mid x \lessdot \alpha]) \,, \\ C(2) \mapsto \mathcal{E}(\phi) \,, \\ C(3) \mapsto \mathcal{E}(\updownarrow[(h, b, s, \updownarrow \mathbf{p}) \mid (h, b, s, \mathbf{p}) \lessdot \beta_H], \mathcal{E}_M(\beta_B)) \,, \\ C(4) \mapsto \mathcal{E}\left(\gamma_P, \gamma_Z, \begin{cases} 0 \text{ if } \gamma_S \in [\![\mathbb{T}]\!]_{\mathsf{E}} \\ 1 \text{ if } \gamma_S \in \left[\!\left[\tilde{\mathbb{H}}\right]\!\right]_{\mathsf{E}} \end{cases}, \gamma_S, \updownarrow \gamma_A\right) \,, \\ C(5) \mapsto \mathcal{E}(\updownarrow[x \in \psi_G \setminus x], \updownarrow[x \in \psi_B \setminus x], \updownarrow[x \in \psi_W \setminus x], \updownarrow[x \in \psi_O \setminus x]) \,, \\ 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}([\natural(\mathbf{r}, \mathcal{E}_4(t)) \mid (\mathbf{r}, t) \lessdot \rho]) \,, \\ C(11) \mapsto \mathcal{E}_4(\tau) \,, \\ C(12) \mapsto \mathcal{E}(\mathcal{E}_4(\chi_M, \chi_A, \chi_V, \chi_R), \chi_Z) \,, \\ C(13) \mapsto \mathcal{E}(\mathcal{E}_4(\pi_V, \pi_L), \pi_C, \pi_S) \,, \\ C(14) \mapsto \mathcal{E}([\updownarrow[(\mathbf{r}, \updownarrow \mathbf{d}) \mid (\mathbf{r}, \mathbf{d}) \lessdot \mathbf{i}] \mid \mathbf{i} \lessdot \omega]) \,, \\ C(15) \mapsto \mathcal{E}([\updownarrow \mathbf{i} \mid \mathbf{i} \lessdot \xi]) \,, \\ C(16) \mapsto \mathcal{E}(\updownarrow[(\mathcal{E}_4(s), \mathcal{E}(h)) \mid (s, h) \lessdot \theta]) \,, \\ \forall (s \mapsto \mathbf{a}) \in \delta : \quad C(255, s) \mapsto \mathcal{E}(0, \mathbf{a}_c, \mathcal{E}_8(\mathbf{a}_b, \mathbf{a}_g, \mathbf{a}_m, \mathbf{a}_o, \mathbf{a}_f), \mathcal{E}_4(\mathbf{a}_i, \mathbf{a}_r, \mathbf{a}_a, \mathbf{a}_p)) \,, \\ \forall (s \mapsto \mathbf{a}) \in \delta, (\mathbf{k} \mapsto \mathbf{v}) \in \mathbf{a_s} : \quad C(s, \mathcal{E}_4(2^{32} - 1) \frown \mathbf{k}) \mapsto \mathbf{v} \,, \\ \forall (s \mapsto \mathbf{a}) \in \delta, (h \mapsto \mathbf{p}) \in \mathbf{a_p} : \quad C(s, \mathcal{E}_4(2^{32} - 2) \frown h) \mapsto \mathbf{p} \,, \\ \forall (s \mapsto \mathbf{a}) \in \delta, ((h, l) \mapsto \mathbf{t}) \in \mathbf{a_l} : \quad C(s, \mathcal{E}_4(l) \frown h) \mapsto \mathcal{E}(\updownarrow[\mathcal{E}_4(x) \mid x \lessdot \mathbf{t}]) \end{cases}$$

Note that most rows describe a single mapping between a key derived from a natural 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 numeric serialization in state is done in fixed-length according to the size of the term.

Finally, be aware that JAM does not allow service storage keys to be directly inspected or enumerated. Thus the key values themselves are not required to be known by implementations, and only the Merklisation-ready serialisation is important, which is a fixed-size hash (alongside the service index and item marker). Implementations are free to use this fact in order to avoid storing the keys themselves.

D.2. **Merklization.** With T defined, we now define the rest of $\mathcal{M}_\sigma$ 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 512 bit (64 bytes). Each node is either a branch or a leaf. The first bit discriminate between these two types.

In the case of a branch, the remaining 511 bits are split between the two child node hashes, using the last 255 bits of the 0-bit (left) sub-trie identity and the full 256 bits of the 1-bit (right) sub-trie identity.

Leaf nodes are further subdivided into embedded-value leaves and regular leaves. The second bit of the node discriminates between these.

In the case of an embedded-value leaf, the remaining 6 bits of the first byte are used to store the embedded value size. The following 31 bytes are dedicated to the state key. The last 32 bytes are defined as the value, filling with zeroes if its length is less than 32 bytes.

In the case of a regular leaf, the remaining 6 bits of the first byte are zeroed. The following 31 bytes store the state key. The last 32 bytes store the hash of the value.

Formally, we define the encoding functions B and L:

$$B: \begin{cases} (\mathbb{H}, \mathbb{H}) \to \mathbb{b}_{512} \\ \quad (l, r) \mapsto [0] \frown \text{bits}(l)_{1\ldots} \frown \text{bits}(r) \end{cases} \tag{D.3}$$

$$L: \begin{cases} (\mathbb{B}_{31}, \mathbb{B}) \to \mathbb{b}_{512} \\ \quad (k, v) \mapsto \begin{cases} [1,0] \frown \text{bits}(\mathcal{E}_1(|v|))_{2\ldots} \frown \text{bits}(k) \frown \text{bits}(v) \frown [0,0,\ldots] & \text{if } |v| \le 32 \\ [1,1,0,0,0,0,0,0] \frown \text{bits}(k) \frown \text{bits}(\mathcal{H}(v)) & \text{otherwise} \end{cases} \end{cases} \tag{D.4}$$

We may then define the basic Merklization function $\mathcal{M}_\sigma$ as:

$$\mathcal{M}_\sigma(\sigma) \equiv M(\{\, (\text{bits}(k) \mapsto (k, v)) \mid (k \mapsto v) \in T(\sigma) \,\}) \tag{D.5}$$

$$M(d : \{\mathbb{b} \to (\mathbb{B}_{31}, \mathbb{B})\}) \equiv \begin{cases} \mathbb{H}_0 & \text{if } |d| = 0 \\ \mathcal{H}\big(\text{bits}^{-1}(L(k, v))\big) & \text{if } \mathcal{V}(d) = \{\, (k, v) \,\} \\ \mathcal{H}\big(\text{bits}^{-1}(B(M(l), M(r)))\big) & \text{otherwise} \\ \quad \text{where } \forall b, p : (b \mapsto p) \in d \Leftrightarrow (b_{1\ldots} \mapsto p) \in \begin{cases} l & \text{if } b_0 = 0 \\ r & \text{if } b_0 = 1 \end{cases} \end{cases} \tag{D.6}$$

Appendix E. General Merklization

E.1. **Binary Merkle Trees.** The Merkle tree is a cryptographic data structure yielding a hash commitment to a specific sequence of values. It provides $O(N)$ computation and $O(\log(N))$ 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.

The underlying function for our Merkle trees is the *node* function N, which accepts some sequence of blobs of some length n and provides either such a blob back or a hash:

$$N: \begin{cases} (\llbracket \mathbb{B}_n \rrbracket, \mathbb{B} \to \mathbb{H}) \to \mathbb{B}_n \cup \mathbb{H} \\ \quad (\mathbf{v}, H) \mapsto \begin{cases} \mathbb{H}_0 & \text{if } |\mathbf{v}| = 0 \\ \mathbf{v}_0 & \text{if } |\mathbf{v}| = 1 \\ H(\texttt{\$node} \frown N(\mathbf{v}_{\ldots\lceil |\mathbf{v}|/2 \rceil}, H) \frown N(\mathbf{v}_{\lceil |\mathbf{v}|/2 \rceil\ldots}, H)) & \text{otherwise} \end{cases} \end{cases} \tag{E.1}$$

The astute reader will realize that if our $\mathbb{B}_n$ happens to be equivalent $\mathbb{H}$ then this function will always evaluate into $\mathbb{H}$. That said, for it to be secure care must be taken to ensure there is no possibility of preimage collision. For this purpose we include the hash prefix `$node` to minimize the chance of this; simply ensure any items are hashed with a different prefix and the system can be considered secure.

We also define the *trace* function T, which returns each opposite node from top to bottom as the tree is navigated to arrive at some leaf corresponding to the item of a given index into the sequence. It is useful in creating justifications of data inclusion.

$$T: \begin{cases} (\llbracket \mathbb{B}_n \rrbracket, \mathbb{N}_{|\mathbf{v}|}, \mathbb{B} \to \mathbb{H}) \to \llbracket \mathbb{B}_n \cup \mathbb{H} \rrbracket \\ \quad (\mathbf{v}, i, H) \mapsto \begin{cases} \big[N(P^\perp(\mathbf{v}, i), H)\big] \frown T(P^\top(\mathbf{v}, i), i - P_I(\mathbf{v}, i), H) & \text{if } |\mathbf{v}| > 1 \\ [\,] & \text{otherwise} \end{cases} \\ \quad\quad \text{where } P^s(\mathbf{v}, i) \equiv \begin{cases} \mathbf{v}_{\ldots\lceil |\mathbf{v}|/2 \rceil} & \text{if } (i < \lceil |\mathbf{v}|/2 \rceil) = s \\ \mathbf{v}_{\lceil |\mathbf{v}|/2 \rceil\ldots} & \text{otherwise} \end{cases} \\ \quad\quad\quad \text{and } P_I(\mathbf{v}, i) \equiv \begin{cases} 0 & \text{if } i < \lceil |\mathbf{v}|/2 \rceil \\ \lceil |\mathbf{v}|/2 \rceil & \text{otherwise} \end{cases} \end{cases} \tag{E.2}$$

From this we define our other Merklization functions.

E.1.1. *Well-Balanced Tree.* We define the well-balanced binary Merkle function as $\mathcal{M}_B$:

$$\mathcal{M}_B: \begin{cases} (\llbracket \mathbb{B} \rrbracket, \mathbb{B} \to \mathbb{H}) \to \mathbb{H} \\ \quad (\mathbf{v}, H) \mapsto \begin{cases} H(\mathbf{v}_0) & \text{if } |\mathbf{v}| = 1 \\ N(\mathbf{v}, H) & \text{otherwise} \end{cases} \end{cases} \tag{E.3}$$

This is suitable for creating proofs on data which is not much greater than 32 octets in length since it avoids hashing each item in the sequence. For sequences with larger data items, it is better to hash them beforehand to ensure proof-size is minimal since each proof will generally contain a data item.

Note: In the case that no hash function argument H is supplied, we may assume Blake 2b.

E.1.2. *Constant-Depth Tree.* We define the constant-depth binary Merkle function as $\mathcal{M}$. We define two corresponding functions for working with subtree pages, $\mathcal{J}_x$ and $\mathcal{L}_x$. The latter provides a single page of leaves, themselves hashed, prefixed data. The former provides the Merkle path to a single page. Both assume size-aligned pages of size 2^x and accept page indices.

$$\mathcal{M}: \begin{cases} ([\![\mathbb{B}]\!], \mathbb{B} \to \mathbb{H}) \to \mathbb{H} \\ \qquad (\mathbf{v}, H) \mapsto N(C(\mathbf{v}, H), H) \end{cases} \tag{E.4}$$

$$\mathcal{J}_x: \begin{cases} ([\![\mathbb{B}]\!], \mathbb{N}_{|\mathbf{v}|}, \mathbb{B} \to \mathbb{H}) \to [\![\mathbb{H}]\!] \\ \qquad (\mathbf{v}, i, H) \mapsto T(C(\mathbf{v}, H), 2^x i, H)_{\dots \max(0, \lceil \log_2(\max(1, |\mathbf{v}|)) - x \rceil)} \end{cases} \tag{E.5}$$

$$\mathcal{L}_x: \begin{cases} ([\![\mathbb{B}]\!], \mathbb{N}_{|\mathbf{v}|}, \mathbb{B} \to \mathbb{H}) \to [\![\mathbb{H}]\!] \\ \qquad (\mathbf{v}, i, H) \mapsto \left[H(\texttt{\$leaf} \frown l) \;\middle|\; l \in \mathbf{v}_{2^x i \dots \min(2^x i + 2^x, |\mathbf{v}|)} \right] \end{cases} \tag{E.6}$$

For the latter justification $\mathcal{J}_x$ to be acceptable, we must assume the target observer also knows not merely the value of the item at the given index, but also all other leaves within its 2^x size subtree, given by $\mathcal{L}_x$.

As above, we may assume a default value for H of Blake 2b.

For justifications and Merkle root calculations, a constancy preprocessor function C is applied which hashes all data items with a fixed prefix "leaf" and then pads the overall size to the next power of two with the zero hash $\mathbb{H}_0$:

$$C: \begin{cases} ([\![\mathbb{B}]\!], \mathbb{B} \to \mathbb{H}) \to [\![\mathbb{H}]\!] \\ \qquad (\mathbf{v}, H) \mapsto \mathbf{v}' \text{ where } \begin{cases} |\mathbf{v}'| = 2^{\lceil \log_2(\max(1, |\mathbf{v}|)) \rceil} \\ \mathbf{v}'_i = \begin{cases} H(\texttt{\$leaf} \frown \mathbf{v}_i) & \text{if } i < |\mathbf{v}| \\ \mathbb{H}_0 & \text{otherwise} \end{cases} \end{cases} \end{cases} \tag{E.7}$$

E.2. **Merkle Mountain Ranges and Belts.** 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 $O(\log(N))$ 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 MMB append function $\mathcal{A}$ as:

$$\mathcal{A}: \begin{cases} ([\![\mathbb{H}?]\!], \mathbb{H}, \mathbb{B} \to \mathbb{H}) \to [\![\mathbb{H}?]\!] \\ \qquad (\mathbf{r}, l, H) \mapsto P(\mathbf{r}, l, 0, H) \end{cases}$$

$$\text{where } P: \begin{cases} ([\![\mathbb{H}?]\!], \mathbb{H}, \mathbb{N}, \mathbb{B} \to \mathbb{H}) \to [\![\mathbb{H}?]\!] \\ \qquad (\mathbf{r}, l, n, H) \mapsto \begin{cases} \mathbf{r} +\!\!+ l & \text{if } n \geq |\mathbf{r}| \\ R(\mathbf{r}, n, l) & \text{if } n < |\mathbf{r}| \wedge \mathbf{r}_n = \varnothing \\ P(R(\mathbf{r}, n, \varnothing), H(\mathbf{r}_n \frown l), n + 1, H) & \text{otherwise} \end{cases} \end{cases} \tag{E.8}$$

$$\text{and } R: \begin{cases} ([\![T]\!], \mathbb{N}, T) \to [\![T]\!] \\ \qquad (\mathbf{s}, i, v) \mapsto \mathbf{s}' \text{ where } \mathbf{s}' = \mathbf{s} \text{ except } \mathbf{s}'_i = v \end{cases}$$

We define the MMR encoding function as $\mathcal{E}_M$:

$$\mathcal{E}_M: \begin{cases} [\![\mathbb{H}?]\!] \to \mathbb{B} \\ \qquad \mathbf{b} \mapsto \mathcal{E}(\updownarrow[\wr x \mid x \in \mathbf{b}]) \end{cases} \tag{E.9}$$

We define the MMR super-peak function as $\mathcal{M}_R$:

$$\mathcal{M}_R: \begin{cases} [\![\mathbb{H}?]\!] \to \mathbb{H} \\ \qquad \mathbf{b} \mapsto \begin{cases} \mathbb{H}_0 & \text{if } |\mathbf{h}| = 0 \\ \mathbf{h}_0 & \text{if } |\mathbf{h}| = 1 \\ \mathcal{H}_K\left(\texttt{\$peak} \frown \mathcal{M}_R\left(\mathbf{h}_{\dots |\mathbf{h}|-1}\right) \frown \mathbf{h}_{|\mathbf{h}|-1}\right) & \text{otherwise} \\ \quad \text{where } \mathbf{h} = [h \mid h \in \mathbf{b}, h \neq \varnothing] \end{cases} \end{cases} \tag{E.10}$$

Appendix F. Shuffling

The Fisher-Yates shuffle function is defined formally as:

$$\forall T, l \in \mathbb{N} : \mathcal{F}: \begin{cases} (\llbracket T \rrbracket_l, \llbracket \mathbb{N} \rrbracket_{l:}) \to \llbracket T \rrbracket_l \\ (\mathbf{s}, \mathbf{r}) \mapsto \begin{cases} [\mathbf{s}_{\mathbf{r}_0 \bmod l}] \frown \mathcal{F}(\mathbf{s}'_{\dots l-1}, \mathbf{r}_{1\dots}) \text{ where } \mathbf{s}' = \mathbf{s} \text{ except } \mathbf{s}'_{\mathbf{r}_0 \bmod l} = \mathbf{s}_{l-1} & \text{if } \mathbf{s} \neq [] \\ [] & \text{otherwise} \end{cases} \end{cases} \tag{F.1}$$

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:

$$\forall l \in \mathbb{N} : \ \mathcal{Q}_l: \begin{cases} \mathbb{H} \to \llbracket \mathbb{N}_{2^{32}} \rrbracket_l \\ h \mapsto \left[\mathcal{E}_4^{-1}\left(\mathcal{H}(h \frown \mathcal{E}_4(\lfloor i/8 \rfloor))_{4i \bmod 32 \dots +4} \right) \mid i \in \mathbb{N}_l \right] \end{cases} \tag{F.2}$$

$$\forall T, l \in \mathbb{N} : \ \mathcal{F}: \begin{cases} (\llbracket T \rrbracket_l, \mathbb{H}) \to \llbracket T \rrbracket_l \\ (\mathbf{s}, h) \mapsto \mathcal{F}(\mathbf{s}, \mathcal{Q}_l(h)) \end{cases} \tag{F.3}$$

Appendix G. Bandersnatch VRF

The Bandersnatch curve is defined by **cryptoeprint:2021/1152**.

The singly-contextualized Bandersnatch Schnorr-like signatures $\tilde{\mathbb{V}}^m_k\langle c \rangle$ are defined as a formulation under the *IETF* VRF template specified by **hosseini2024bandersnatch** (as IETF VRF) and further detailed by **rfc9381**.

$$\tilde{\mathbb{V}}^{m \in \mathbb{B}}_{k \in \tilde{\mathbb{H}}}\langle c \in \mathbb{H} \rangle \subset \mathbb{B}_{96} \equiv \{ \, x \mid x \in \mathbb{B}_{96}, \text{verify}(k, c, m, x) = \top \, \} \tag{G.1}$$

$$\mathcal{Y}\Big(s \in \tilde{\mathbb{V}}^m_k\langle c \rangle \Big) \in \mathbb{H} \equiv \text{output}(x \mid x \in \tilde{\mathbb{V}}^m_k\langle c \rangle)_{\dots 32} \tag{G.2}$$

The singly-contextualized Bandersnatch RingVRF proofs $\mathring{\mathbb{V}}^m_r\langle c \rangle$ are a zk-SNARK-enabled analogue utilizing the Pedersen VRF, also defined by **hosseini2024bandersnatch** and further detailed by **cryptoeprint:2023/002**.

$$\mathcal{O}\Big(\big[\!\big[\tilde{\mathbb{H}} \big]\!\big] \Big) \in \mathring{\mathbb{B}} \equiv \text{commit}(\big[\!\big[\tilde{\mathbb{H}} \big]\!\big]) \tag{G.3}$$

$$\mathring{\mathbb{V}}^{m \in \mathbb{B}}_{r \in \mathring{\mathbb{B}}}\langle c \in \mathbb{H} \rangle \subset \mathbb{B}_{784} \equiv \{ \, x \mid x \in \mathbb{B}_{784}, \text{verify}(r, c, m, x) = \top \, \} \tag{G.4}$$

$$\mathcal{Y}\Big(p \in \mathring{\mathbb{V}}^m_r\langle c \rangle \Big) \in \mathbb{H} \equiv \text{output}(x \mid x \in \mathring{\mathbb{V}}^m_r\langle c \rangle)_{\dots 32} \tag{G.5}$$

Note that in the case a key $\tilde{\mathbb{H}}$ has no corresponding Bandersnatch point when constructing the ring, then the Bandersnatch *padding point* as stated by **hosseini2024bandersnatch** should be substituted.

Appendix H. Erasure Coding

The foundation of the data-availability and distribution system of JAM is a systematic Reed-Solomon erasure coding function in GF(2^{16}) of rate 342:1023, the same transform as done by the algorithm of **lin2014novel**. We use a little-endian $\mathbb{B}_2$ form of the 16-bit GF points with a functional equivalence given by $\mathcal{E}_2$. From this we may assume the encoding function $\mathcal{C} : \llbracket \mathbb{B}_2 \rrbracket_{342} \to \llbracket \mathbb{B}_2 \rrbracket_{1023}$ and the recovery function $\mathcal{R} : \{\!\{ (\mathbb{B}_2, \mathbb{N}_{1023}) \}\!\}_{342} \to \llbracket \mathbb{B}_2 \rrbracket_{342}$. Encoding is done by extrapolating a data blob of size 684 octets (provided in $\mathcal{C}$ here as 342 octet pairs) into 1,023 octet pairs. Recovery is done by collecting together any distinct 342 octet pairs, together with their indices, and transforming this into the original sequence of 342 octet pairs.

Practically speaking, this allows for the efficient encoding and recovery of data whose size is a multiple of 684 octets. Data whose length is not divisible by 684 must be padded (we pad with zeroes). We use this erasure-coding in two contexts within the JAM protocol; one where we encode variable sized (but typically very large) data blobs for the Audit DA and block-distribution system, and the other where we encode much smaller fixed-size data *segments* for the Import DA system.

For the Import DA system, we deal with an input size of 4,104 octets resulting in data-parallelism of order six. We may attain a greater degree of data parallelism if encoding or recovering more than one segment at a time though for recovery, we may be restricted to requiring each segment to be formed from the same set of indices (depending on the specific algorithm).

H.1. Blob Encoding and Recovery.

We assume some data blob $\mathbf{d} \in \mathbb{B}_{684k}, k \in \mathbb{N}$. This blob is split into a whole number of k pieces, each a sequence of 342 octet pairs. Each piece is erasure-coded using $\mathcal{C}$ as above to give 1,023 octet pairs per piece.

The resulting matrix is grouped by its pair-index and concatenated to form 1,023 *chunks*, each of k octet-pairs. Any 342 of these 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 reconstituting such subsequences back into a single large sequence:

$$\forall n \in \mathbb{N}, k \in \mathbb{N} : \text{split}_n(\mathbf{d} \in \mathbb{B}_{kn}) \in \llbracket \mathbb{B}_n \rrbracket_k \equiv \left[\mathbf{d}_{0 \dots +n}, \mathbf{d}_{n \dots +n}, \dots, \mathbf{d}_{(k-1)n \dots +n} \right] \tag{H.1}$$

$$\forall n \in \mathbb{N}, k \in \mathbb{N} : \text{join}(\mathbf{c} \in [\![\mathbb{B}_n]\!]_k) \in \mathbb{B}_{kn} \equiv \mathbf{c}_0 \frown \mathbf{c}_1 \frown \dots \tag{H.2}$$

We define the transposition operator hence:

$$^{\mathrm{T}}[[\mathbf{x}_{0,0}, \mathbf{x}_{0,1}, \mathbf{x}_{0,2}, \dots], [\mathbf{x}_{1,0}, \mathbf{x}_{1,1}, \dots], \dots] \equiv [[\mathbf{x}_{0,0}, \mathbf{x}_{1,0}, \mathbf{x}_{2,0}, \dots], [\mathbf{x}_{0,1}, \mathbf{x}_{1,1}, \dots], \dots] \tag{H.3}$$

We may then define our erasure-code chunking function which accepts an arbitrary sized data blob whose length divides wholly into 684 octets and results in a sequence of 1,023 smaller blobs:

$$\mathcal{C}_{k \in \mathbb{N}} \colon \begin{cases} \mathbb{B}_{684k} \to [\![\mathbb{B}_{2k}]\!]_{1023} \\ \quad \mathbf{d} \mapsto \text{join}^{\#}(^{\mathrm{T}}\left[\mathcal{C}(\mathbf{p}) \;\middle|\; \mathbf{p} \lessdot {}^{\mathrm{T}}\text{split}_2^{\#}(\text{split}_{2k}(\mathbf{d}))\right]) \end{cases} \tag{H.4}$$

The original data may be reconstructed with any 342 of the 1,023 resultant items (along with their indices). If the original 342 items are known then reconstruction is just their concatenation.

$$\mathcal{R}_{k \in \mathbb{N}} \colon \begin{cases} \{\!\{ (\mathbb{B}_{2k}, \mathbb{N}_{1023}) \}\!\}_{342} \to \mathbb{B}_{684k} \\ \quad \mathbf{c} \mapsto \begin{cases} \mathcal{E}([\mathbf{x} \mid (\mathbf{x}, i) \lessdot [(\mathbf{x}, i) \in \mathbf{c} \setminus i]]) & \text{if } \{\, i \mid (\mathbf{x}, i) \in \mathbf{c} \,\} = \mathbb{N}_{342} \\ \text{join}(\text{join}^{\#}(^{\mathrm{T}}[\mathcal{R}(\{\, (\text{split}_2(\mathbf{x})_p, i) \mid (\mathbf{x}, i) \in \mathbf{c} \,\}) \mid p \in \mathbb{N}_k])) & \text{always} \end{cases} \end{cases} \tag{H.5}$$

Segment encoding/decoding may be done using the same functions albeit with a constant $k = 6$.

H.2. **Code Word representation.** For the sake of brevity we call each octet pair a *word*. The code words (including the message words) are treated as element of $\mathbb{F}_{2^{16}}$ finite field. The field is generated as an extension of $\mathbb{F}_2$ using the irreducible polynomial:

$$x^{16} + x^5 + x^3 + x^2 + 1 \tag{H.6}$$

Hence:

$$\mathbb{F}_{2^{16}} \equiv \frac{\mathbb{F}_2[x]}{x^{16} + x^5 + x^3 + x^2 + 1} \tag{H.7}$$

We name the generator of $\frac{\mathbb{F}_{2^{16}}}{\mathbb{F}_2}$, the root of the above polynomial, α as such: $\mathbb{F}_{2^{16}} = \mathbb{F}_2(\alpha)$.

Instead of using the standard basis $\{1, \alpha, \alpha^2, \dots, \alpha^{15}\}$, we opt for a representation of $\mathbb{F}_{2^{16}}$ which performs more efficiently for the encoding and the decoding process. To that aim, we name this specific representation of $\mathbb{F}_{2^{16}}$ as $\tilde{\mathbb{F}}_{2^{16}}$ and define it as a vector space generated by the following Cantor basis:

v_0	1
v_1	$\alpha^{15} + \alpha^{13} + \alpha^{11} + \alpha^{10} + \alpha^7 + \alpha^6 + \alpha^3 + \alpha$
v_2	$\alpha^{13} + \alpha^{12} + \alpha^{11} + \alpha^{10} + \alpha^3 + \alpha^2 + \alpha$
v_3	$\alpha^{12} + \alpha^{10} + \alpha^9 + \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 + \alpha$
v_4	$\alpha^{15} + \alpha^{14} + \alpha^{10} + \alpha^8 + \alpha^7 + \alpha$
v_5	$\alpha^{15} + \alpha^{14} + \alpha^{13} + \alpha^{11} + \alpha^{10} + \alpha^8 + \alpha^5 + \alpha^3 + \alpha^2 + \alpha$
v_6	$\alpha^{15} + \alpha^{12} + \alpha^8 + \alpha^6 + \alpha^3 + \alpha^2$
v_7	$\alpha^{14} + \alpha^4 + \alpha$
v_8	$\alpha^{14} + \alpha^{13} + \alpha^{11} + \alpha^{10} + \alpha^7 + \alpha^4 + \alpha^3$
v_9	$\alpha^{12} + \alpha^7 + \alpha^6 + \alpha^4 + \alpha^3$
v_{10}	$\alpha^{14} + \alpha^{13} + \alpha^{11} + \alpha^9 + \alpha^6 + \alpha^5 + \alpha^4 + \alpha$
v_{11}	$\alpha^{15} + \alpha^{13} + \alpha^{12} + \alpha^{11} + \alpha^8$
v_{12}	$\alpha^{15} + \alpha^{14} + \alpha^{13} + \alpha^{12} + \alpha^{11} + \alpha^{10} + \alpha^8 + \alpha^7 + \alpha^5 + \alpha^4 + \alpha^3$
v_{13}	$\alpha^{15} + \alpha^{14} + \alpha^{13} + \alpha^{12} + \alpha^{11} + \alpha^9 + \alpha^8 + \alpha^5 + \alpha^4 + \alpha^2$
v_{14}	$\alpha^{15} + \alpha^{14} + \alpha^{13} + \alpha^{12} + \alpha^{11} + \alpha^{10} + \alpha^9 + \alpha^8 + \alpha^5 + \alpha^4 + \alpha^3$
v_{15}	$\alpha^{15} + \alpha^{12} + \alpha^{11} + \alpha^8 + \alpha^4 + \alpha^3 + \alpha^2 + \alpha$

Every message word $m_i = m_{i,15} \dots m_{i,0}$ consists of 16 bits. As such it could be regarded as binary vector of length 16:

$$m_i = (m_{i,0} \dots m_{i,15}) \tag{H.8}$$

Where $m_{i,0}$ is the least significant bit of message word m_i. Accordingly we consider the field element $\tilde{m}_i = \sum_{j=0}^{15} m_{i,j} v_j$ to represent that message word.

Similarly, we assign a unique index to each validator between 0 and 1,022 and we represent validator i with the field element:

$$\tilde{i} = \sum_{j=0}^{15} i_j v_j \tag{H.9}$$

where $i = i_{15} \dots i_0$ is the binary representation of i.

H.3. **The Generator Polynomial.** To erasure code a message of 342 words into 1023 code words, we represent each message as a field element as described in previous section and we interpolate the polynomial $p(y)$ of maximum 341 degree which satisfies the following equalities:

$$\begin{aligned} p(\tilde{0}) &= \widetilde{m_0} \\ p(\tilde{1}) &= \widetilde{m_1} \\ &\vdots \\ p(\widetilde{341}) &= \widetilde{m_{341}} \end{aligned} \tag{H.10}$$

After finding $p(y)$ with such properties, we evaluate p at the following points:

$$\begin{aligned} \widetilde{r_{342}} &:= p(\widetilde{342}) \\ \widetilde{r_{343}} &:= p(\widetilde{343}) \\ &\vdots \\ \widetilde{r_{1022}} &:= p(\widetilde{1022}) \end{aligned} \tag{H.11}$$

We then distribute the message words and the extra code words among the validators according to their corresponding indices.

Appendix I. Index of Notation

I.1. Sets.

I.1.1. *Regular Notation.*

$\mathbb{F}$: The set of finite fields.

$\mathbb{N}$: The set of non-negative integers. 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 4.21.
- $\mathbb{N}_G$: The set of unsigned gas values. Equivalent to $\mathbb{N}_{2^{64}}$. See equation 4.23.
- $\mathbb{N}_L$: The set of blob length values. Equivalent to $\mathbb{N}_{2^{32}}$. See section 3.4.
- $\mathbb{N}_R$: The set of register values. Equivalent to $\mathbb{N}_{2^{64}}$. See equation 4.23.
- $\mathbb{N}_S$: The set from which service indices are drawn. Equivalent to $\mathbb{N}_{2^{32}}$. See section 9.1.
- $\mathbb{N}_T$: The set of timeslot values. Equivalent to $\mathbb{N}_{2^{32}}$. See equation 4.28.

$\mathbb{Q}$: The set of rational 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 $\mathbb{Z}_{-2^{63}\ldots 2^{63}}$. See equation 4.23.

I.1.2. *Custom Notation.*

$\langle K \to V \rangle$: The set of dictionaries making a partial bijection of domain k to range v. See section 3.5.

$\mathbb{A}$: The set of service $\mathbb{A}$ccounts. See equation 9.3.

$\mathbb{b}$: The set of $\mathbb{b}$itstrings (Boolean sequences). Subscript denotes length. See section 3.7.

$\mathbb{B}$: The set of $\mathbb{B}$lobs (octet sequences). Subscript denotes length. See section 3.7.

- $\overset{\text{BLS}}{\mathbb{B}}$: The set of BLS public keys. A subset of $\mathbb{B}_{144}$. See section 3.8.2.
- $\mathring{\mathbb{B}}$: The set of Bandersnatch ring roots. A subset of $\mathbb{B}_{144}$. See section 3.8 and appendix G.

$\mathbb{C}$: The set of work-$\mathbb{C}$ontexts. See equation 11.4. *Not used as the set of complex numbers.*

$\mathbb{D}$: The set of work-$\mathbb{D}$igests. See equation 11.6.

$\mathbb{E}$: The set of work execution $\mathbb{E}$rrors. See equation 11.7.

$\mathbb{G}$: The set representing the state of a $\mathbb{G}$uest PVM instance. See equation B.4.

$\mathbb{H}$: The set of 32-octet cryptographic values, equivalent to $\mathbb{B}_{32}$. Often a $\mathbb{H}$ash function's result. See section 3.8.

- $\mathbb{H}$: The set of Ed25519 public keys. A subset of $\mathbb{B}_{32}$. See section 3.8.2.
- $\mathring{\mathbb{H}}$: The set of Bandersnatch public keys. A subset of $\mathbb{B}_{32}$. See section 3.8 and appendix G.

$\mathbb{U}$: The $\mathbb{I}$nformation concerning a single work-item once prepared as an operand for the accumulation function. See equation 12.13.

$\mathbb{J}$: The set of data segments, equivalent to $\mathbb{B}_{\mathsf{W}_G}$. See equation 14.1.

$\mathbb{K}$: The set of validator $\mathbb{K}$ey-sets. See equation 6.7.

$\mathbb{L}$: The set representing implications of accumulation. See equation B.7.

$\mathbb{M}$: The set of PVM $\mathbb{M}$emory (RAM) states. See equation 4.24.

$\mathbb{P}$: The set of work-$\mathbb{P}$ackages. See equation 14.2.

$\mathbb{R}$: The set of work-$\mathbb{R}$eports. See equation 11.2. *Note used for the set of real numbers.*

$\mathbb{S}$: The set representating a portion of overall $\mathbb{S}$tate, used during accumulation. See equation 12.16.

$\mathbb{T}$: The set of seal-key $\mathbb{T}$ickets. See equation 6.6.

$\mathbb{V}_\mu$: The set of $\mathbb{V}$alidly readable indices for PVM RAM μ. See appendix A.

$\mathbb{V}^*_\mu$: The set of $\mathbb{V}$alidly writable indices for PVM RAM μ. See appendix A.

$\bar{\mathbb{V}}_k\langle m \rangle$: The set of $\mathbb{V}$alid Ed25519 signatures of the key k and message m. A subset of $\mathbb{B}_{64}$. See section 3.8.

$\tilde{\mathbb{V}}^m_k\langle c \rangle$: The set of $\mathbb{V}$alid Bandersnatch signatures of the public key k, context c and message m. A subset of $\mathbb{B}_{96}$. See section 3.8.

$\mathring{\mathbb{V}}^m_r\langle c \rangle$: The set of $\mathbb{V}$alid Bandersnatch RingVRF proofs of the root r, context c and message m. A subset of $\mathbb{B}_{784}$. See section 3.8.

$\mathbb{W}$: The set of $\mathbb{W}$ork items. See equation 14.3.

$\mathbb{X}$: The set of deferred transfers. See equation 12.14.

$\mathbb{Y}$: The set of availability specifications. See equation 11.5.

I.2. Functions.

Δ: The accumulation functions (see section 12.2):

- Δ_1: The single-step accumulation function. See equation 12.24.
- Δ_*: The parallel accumulation function. See equation 12.19.
- Δ_+: The full sequential accumulation function. See equation 12.18.

Λ: The historical lookup function. See equation 9.7.

Ξ: The work-report computation function. See equation 14.12.

Υ: The general state transition function. See equations 4.1, 4.5.

Φ: The key-nullifier function. See equation 6.14.

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.
Ω: Virtual machine host-call functions. See appendix B.
Ω_A: Assign-core host-call.
Ω_B: Empower-service host-call.
Ω_C: Checkpoint host-call.
Ω_D: Designate-validators host-call.
Ω_E: Export segment 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_J: Eject-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: Query-preimage host-call.
Ω_R: Read-storage host-call.
Ω_S: Solicit-preimage 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.
Ω_Y: Fetch data host-call.
Ω_Z: Pages inner-PVM memory host-call.
$\Omega_\aleph$: Yield accumulation trie result host-call.
$\Omega_\wp$: Provide preimage host-call.

I.3. **Utilities, Externalities and Standard Functions.**

$\mathcal{A}(\dots)$: The Merkle mountain range append function. See equation E.8.
$\mathcal{B}_n(\dots)$: The octets-to-bits function for n octets. Superscripted $^{-1}$ to denote the inverse. See equation A.12.
$\mathcal{C}_n(\dots)$: The erasure-coding functions for n chunks. See equation H.4.
$\mathcal{E}(\dots)$: The octet-sequence encode function. Superscripted $^{-1}$ to denote the inverse. See appendix C.
$\mathcal{F}(\dots)$: The Fisher-Yates shuffle function. See equation F.1.
$\mathcal{H}(\dots)$: 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.
$\mathcal{J}_x$: The justification path to a specific 2^x size page of a constant-depth Merkle tree. See equation E.5.
$\mathcal{K}(\dots)$: The domain, or set of keys, of a dictionary. See section 3.5.
$\mathcal{L}_x$: The 2^x size page function for a constant-depth Merkle tree. See equation E.6.
$\mathcal{M}(\dots)$: The constant-depth binary Merklization function. See appendix E.
$\mathcal{M}_B(\dots)$: The well-balanced binary Merklization function. See appendix E.
$\mathcal{M}_\sigma(\dots)$: The state Merklization function. See appendix D.
$\mathcal{O}(\dots)$: The Bandersnatch ring root function. See section 3.8 and appendix G.
$\mathcal{P}_n(\dots)$: The octet-array zero-padding function. See equation 14.18.
$\mathcal{Q}(\dots)$: The numeric-sequence-from-hash function. See equation F.3.
$\mathcal{R}(\dots)$: The group of erasure-coding piece-recovery functions. See equation H.5.
$\bar{\mathcal{S}}(\dots)$: The Ed25519 signing function. See section 3.8.
$\overset{\text{BLS}}{\mathcal{S}}(\dots)$: The BLS signing 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 3.2.
$\mathcal{V}(\dots)$: The range, or set of values, of a dictionary or sequence. See section 3.5.
$\mathcal{X}_n(\dots)$: The signed-extension function for a value in $\mathbb{N}_{2^{8n}}$. See equation A.16.
$\mathcal{Y}(\dots)$: 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 A.10.

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 5.3.
B: The block. See equation 4.2.
E: The block extrinsic. See equation 4.3.
F$_v$**:** The BEEFY signed commitment of validator v. See equation 18.1.
G: The set of Ed25519 guarantor keys who made a work-report. See equation 11.26.
H: The block header. See equation 5.1.
S: The sequence of work-reports which were accumulated this in this block. See equations 12.28 and 12.29.
M: The mapping from cores to guarantor keys. See section 11.3.
M***:** The mapping from cores to guarantor keys for the previous rotation. See section 11.3.
R: The sequence of work-reports which have now become available and ready for accumulation. See equation 11.16.
T: The ticketed condition, true if the block was sealed with a ticket signature rather than a fallback. See equations 6.15 and 6.16.
U: The audit condition, equal to $\top$ once the block is audited. See section 17.

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 19). Explicit block-contextualizing superscripts include:

B$^{\natural}$**:** The latest finalized block. See equation 19.
B$^{\flat}$**:** The block at the head of the best chain. See equation 19.

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 8.1.
β**:** Log of recent activity. See equation 7.1.
  β_H**:** Information on the most recent blocks. See equation 7.2.
  β_B**:** The Merkle mountain belt for accumulating Accumulation outputs. See equations 7.3 and 7.7.
γ**:** State concerning Safrole. See equation 6.3.
  γ_A**:** The sealing lottery ticket accumulator. See equation 6.5.
  γ_P**:** The keys for the validators of the next epoch, equivalent to those keys which constitute γ_Z. See equation 6.7.
  γ_S**:** The sealing-key sequence of the current epoch. See equation 6.5.
  γ_Z**:** The Bandersnatch root for the current epoch's ticket submissions. See equation 6.4.
δ**:** The (prior) state of the service accounts. See equation 9.1.
  $\delta^\dagger$**:** The post-accumulation, pre-preimage integration intermediate state. See equation 12.27.
η**:** The entropy accumulator and epochal randomness. See equation 6.21.
ι**:** The validator keys and metadata to be drawn from next. See equation 6.7.
κ**:** The validator keys and metadata currently active. See equation 6.7.
λ**:** The validator keys and metadata which were active in the prior epoch. See equation 6.7.
ρ**:** The pending reports, per core, which are being made available prior to accumulation. See equation 11.1.
  $\rho^\dagger$**:** The post-judgment, pre-guarantees-extrinsic intermediate state. See equation 10.15.
  $\rho^\ddagger$**:** The post-guarantees-extrinsic, pre-assurances-extrinsic, intermediate state. See equation 11.17.
σ**:** The overall state of the system. See equations 4.1, 4.4.
τ**:** The most recent block's timeslot. See equation 6.1.
ϕ**:** The authorization queue. See equation 8.1.
ψ**:** Past judgments on work-reports and validators. See equation 10.1.
  ψ_B**:** Work-reports judged to be incorrect. See equation 10.17.
  ψ_G**:** Work-reports judged to be correct. See equation 10.16.
  ψ_W**:** Work-reports whose validity is judged to be unknowable. See equation 10.18.
  ψ_O**:** Validators who made a judgment found to be incorrect. See equation 10.19.
χ**:** The privileged service indices. See equation 9.9.
  χ_M**:** The index of the blessed service. See equation 12.27.
  χ_A**:** The indices of the services able to assign each core's authorizer queue. See equation 12.27.
  χ_V**:** The index of the designate service. See equation 12.27.
  χ_R**:** The index of the registrar service. See equation 12.27.
  χ_Z**:** The always-accumulate service indices and their basic gas allowance. See equation 12.27.
π**:** The activity statistics for the validators. See equation 13.1.
ω**:** The accumulation queue. See equation 12.3.
ξ**:** The accumulation history. See equation 12.1.
θ**:** The most recent Accumulation outputs. See equations 7.4 and 12.25.

I.4.3. *Virtual Machine components.*

ε**:** The exit-reason resulting from all machine state transitions.
ν**:** The immediate values of an instruction.

μ: The memory sequence; a member of the set $\mathbb{M}$.
ϱ: The gas counter.
φ: The registers.
ζ: The instruction sequence.
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. See section 17.3.
$\mathsf{B}_I = 10$: The additional minimum balance required per item of elective service state. See equation 9.8.
$\mathsf{B}_L = 1$: The additional minimum balance required per octet of elective service state. See equation 9.8.
$\mathsf{B}_S = 100$: The basic minimum balance which all services require. See equation 9.8.
$\mathsf{C} = 341$: The total number of cores.
$\mathsf{D} = 19,200$: The period in timeslots after which an unreferenced preimage may be expunged. See `eject` definition in section B.7.
$\mathsf{E} = 600$: The length of an epoch in timeslots. See section 4.8.
$\mathsf{F} = 2$: 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. See equation 17.14.
$\mathsf{G}_A = 10,000,000$: The gas allocated to invoke a work-report's Accumulation logic.
$\mathsf{G}_I = 50,000,000$: The gas allocated to invoke a work-package's Is-Authorized logic.
$\mathsf{G}_R = 5,000,000,000$: The gas allocated to invoke a work-package's Refine logic.
$\mathsf{G}_T = 3,500,000,000$: The total gas allocated across for all Accumulation. Should be no smaller than $\mathsf{G}_A \cdot \mathsf{C} + \sum_{g \in \mathcal{V}(\chi_Z)}(g)$.
$\mathsf{H} = 8$: The size of recent history, in blocks. See equation 7.8.
$\mathsf{I} = 16$: The maximum amount of work items in a package. See equations 11.2 and 14.2.
$\mathsf{J} = 8$: The maximum sum of dependency items in a work-report. See equation 11.3.
$\mathsf{K} = 16$: The maximum number of tickets which may be submitted in a single extrinsic. See equation 6.30.
$\mathsf{L} = 14,400$: The maximum age in timeslots of the lookup anchor. See equation 11.34.
$\mathsf{N} = 2$: The number of ticket entries per validator. See equation 6.29.
$\mathsf{O} = 8$: The maximum number of items in the authorizations pool. See equation 8.1.
$\mathsf{P} = 6$: The slot period, in seconds. See equation 4.8.
$\mathsf{Q} = 80$: The number of items in the authorizations queue. See equation 8.1.
$\mathsf{R} = 10$: The rotation period of validator-core assignments, in timeslots. See sections 11.3 and 11.4.
$\mathsf{S} = 2^{16}$: The minimum public service index. Services of indices below these may only be created by the Registrar. See equation B.14.
$\mathsf{T} = 128$: The maximum number of extrinsics in a work-package. See equation 14.4.
$\mathsf{U} = 5$: The period in timeslots after which reported but unavailable work may be replaced. See equation 11.17.
$\mathsf{V} = 1023$: The total number of validators.
$\mathsf{W}_A = 64,000$: The maximum size of is-authorized code in octets. See equation B.1.
$\mathsf{W}_B = 13,791,360$: The maximum size of the concatenated variable-size blobs, extrinsics and imported segments of a work-package, in octets. See equation 14.5.
$\mathsf{W}_C = 4,000,000$: The maximum size of service code in octets. See equations B.5, B.9 & **??**.
$\mathsf{W}_E = 684$: The basic size of erasure-coded pieces in octets. See equation H.4.
$\mathsf{W}_G = \mathsf{W}_P\mathsf{W}_E = 4104$: The size of a segment in octets. See section 14.2.1.
$\mathsf{W}_F = \mathsf{W}_G + 32\lceil\log_2(\mathsf{W}_M)\rceil = 4488$: The additional footprint in the Audits DA of a single imported segment. See equation 14.6.
$\mathsf{W}_M = 3,072$: The maximum number of imports in a work-package. See equation 14.4.
$\mathsf{W}_P = 6$: The number of erasure-coded pieces in a segment.
$\mathsf{W}_R = 48 \cdot 2^{10}$: The maximum total size of all unbounded blobs in a work-report, in octets. See equation 11.8.
$\mathsf{W}_T = 128$: The size of a transfer memo in octets. See equation 12.14.
$\mathsf{W}_X = 3,072$: The maximum number of exports in a work-package. See equation 14.4.
X: Context strings, see below.
$\mathsf{Y} = 500$: The number of slots into an epoch at which ticket-submission ends. See sections 6.5, 6.6 and 6.7.
$\mathsf{Z}_A = 2$: The PVM dynamic address alignment factor. See equation A.18.
$\mathsf{Z}_I = 2^{24}$: The standard PVM program initialization input data size. See equation A.7.
$\mathsf{Z}_P = 2^{12}$: The PVM memory page size. See equation 4.24.
$\mathsf{Z}_Z = 2^{16}$: The standard PVM program initialization zone size. See section A.7.

I.4.5. *Signing Contexts.*

X_A = `$jam_available`: *Ed25519* Availability assurances. See equation 11.13.
X_B = `$jam_beefy`: *BLS* Accumulate-result-root-MMR commitment. See equation 18.1.
X_E = `$jam_entropy`: On-chain entropy generation. See equation 6.17.
X_F = `$jam_fallback_seal`: *Bandersnatch* Fallback block seal. See equation 6.16.
X_G = `$jam_guarantee`: *Ed25519* Guarantee statements. See equation 11.26.

X_I = `$jam_announce:` *Ed25519* Audit announcement statements. See equation 17.8.
X_T = `$jam_ticket_seal:` *Bandersnatch RingVRF* Ticket generation and regular block seal. See equation 6.15.
X_U = `$jam_audit:` *Bandersnatch* Audit selection entropy. See equations 17.3 and 17.14.
$\mathsf{X}_\top$ = `$jam_valid:` *Ed25519* Judgments for valid work-reports. See equation 17.17.
$\mathsf{X}_\bot$ = `$jam_invalid:` *Ed25519* Judgments for invalid work-reports. See equation 17.17.