

Truebit Unchained

a technical report on transparency

Jason Teutsch Federico Kattan Blane Sims

March 27, 2024

Abstract

Data interactions frequently lack provenance due to external dependencies. Indeed, a chasm exists between the visibility of what one computes locally and the more opaque work of others. We investigate this issue within the context of transparent computation wherein a machine which deviates from its prescribed protocol triggers a smoking gun witnessing its erratic behavior. Finally, we implement a protocol which operates efficiently under this model.

Contents

1	Invariants as a basis for trust	2
2	The need for transcripts	3
3	Aspects of transparent computation	3
3.1	Blockchain as auditor	4
3.2	Cloud coordination	4
3.3	A taste of secret sauce	4
4	Task lifecycle	5
4.1	Protocol overview	5
4.1.1	Code distribution	5
4.1.2	Task issuance	6
4.1.3	Node registration	6
4.1.4	Node selection	7
4.1.5	Task broadcast	7
4.1.6	Solution commitment	7
4.1.7	Transcript generation	7

4.1.8	Node remuneration	8
4.2	Task Requester endpoints	8
4.2.1	Metatasks	9
4.2.2	Hub-attested transcript	9
4.2.3	Transcript ensemble	9
4.2.4	External data integration	9
4.2.5	Snap response	10
4.3	Economics don't lie	10
5	Things that can't go wrong	10
5.1	Solution censorship	10
5.2	Conflicting solutions	11
5.3	Collusion on Node selection	11
5.4	Are spoofed transcripts possible?	12
5.5	Lazy Dispatcher versus red balloons	14
5.6	A filesystem broadcast protocol	16
5.6.1	Code files and large inputs	17
5.6.2	Large solutions	17
5.7	Deregistering unresponsive Nodes	18
5.8	On slightly misplaced timestamps	18
5.9	Premature reveals	18
5.10	Bifurcated invoices	19
6	Platform architecture	19
6.1	Audit ledgers	19
6.2	Preparing a task	21
6.2.1	Instrumentation	21
6.2.2	Pre-deployment	22
6.3	Coordination services	23
6.3.1	Deployment	23
6.3.2	Dispatcher	23
6.3.3	Hub	24
6.3.4	Transcripts	24
6.3.5	Accounting	25
6.4	Inside a Node	25

1 Invariants as a basis for trust

An *invariant* is something that doesn't change regardless of environmental circumstances or operations. Barring a miraculous resurfacing of the net-

work founder’s private keys, “Satoshi Nakamoto’s wallets hold one million bitcoin” is an invariant [12]. Invariants foster trust in day-to-day society. In real estate, a buyer and seller contractually agree on transaction invariants, including price, timeline, and documentation requirements, while they work to remove contingencies. A word to the wise: when someone asks for trust, demand to see invariants.

Let us zoom in on the custom, real estate engagement. Once two parties have agreed on contract terms in the form of a Microsoft Word .docx document, one party is supposed to convert that .docx to .pdf, sign it, and email the signed .pdf to the other side for countersignature. How does the countersigning party determine whether the .pdf conversion process actually preserved the .docx text? Traditionally, the receiver performs a careful, manual comparison of the two files. Wouldn’t it be nice if instead the sender included machine-readable proof that the document transformation was text-invariant?

2 The need for transcripts

A *certificate* is definitive proof that something happened. Network execution of a *task* results in an augmented certificate called a *transcript*, a descriptive object relative to the traditional realm of *proof-based verifiable computing* [31]. Transcripts chronicle code execution, including what was executed and when, as well as inputs, outputs or results, identifiers for each party that touches data or code, methodology including virtual machine parameters, as well as annotations for economic incentives, runtime errors, consensus issues, data accessibility, and automated resolutions.

Transcripts enable universal consensus with respect to data origin and processing, or *data provenance* [3]. They indicate the *event provenance* of when the processing took place, can trigger downstream actions, and conversely afford an iterative view into historical, upstream processing. For analogy, an email documents that a message was sent at a particular time, while a transcript does the same for arbitrary code.

In short, a transcript provides a portable, invariant witness of process “unchained” from any particular network.

3 Aspects of transparent computation

A *transparent* system produces transcripts through a specialized server, called a *Hub*, whose operation combines elements of blockchain and cloud.

In a nutshell, if either a compromised Hub or its related services were to deviate from prescribed protocol, the corresponding *transcript ensemble* (Section 4.2.3) would witness their collective, erratic behavior as a *smoking gun* [26]. Absent a smoking gun, a truly transparent system rules out improper action as infeasible. Meanwhile, the Hub holds independent, *Byzantine* [20] solver *Nodes* accountable for task execution.

Why should one rely on an errant Hub to report its own errors in a transcript which it itself creates? Before dipping into secret sauce, let us first inspect a few properties of the transparent computation model.

3.1 Blockchain as auditor

Blockchain *ledgers* globally and indisputably time-stamp data [30]. Since the present system writes to ledger only during auditing and registration, and not routine execution, its operation avoids typical scalability issues associated with blockchains [19]. Indeed, ledger capacity bounds neither task throughput nor the quantity of active Nodes available to execute tasks.

3.2 Cloud coordination

Transparent computation can realize blockchain-like auditability without sacrificing the familiar performance, versatility, cost-effectiveness, and convenience of mainstream, cloud applications. Unlike *smart contracts* [33], whose data processing requires heavy, universal consensus, the lightweight, auditable Hub operates like a typical cloud server. Not only does the Hub “stay awake” between interactions, but it can access internet resources and message natively via standard protocols like `https`.

3.3 A taste of secret sauce

While the Hub and Nodes message each other entirely via standard protocols under normal operation, a Node can at any moment use a ledger to time-stamp, or *time-channel*, its task execution message. This keeps the Hub on its toes and prevents censorship (Section 5.1). In addition, economic incentives exist to ensure that Nodes complain about errant transcripts and record such audits to ledger (Section 4.1.8). Finally, anyone inspecting a task transcript *ex post facto* can both confirm that the protocol selected Nodes randomly (Section 5.3) and that the selected Nodes actually performed the task (Section 4.1.6).

4 Task lifecycle

The Hub outsources computation tasks to multiple, economically motivated Nodes, and the network scales as additional Nodes join. While we assume finances bound Node behavior, the system in fact tolerates a Byzantine majority with respect to task execution.

4.1 Protocol overview

Basic, *happy path* task execution follows eight phases. All messages are cryptographically signed by their respective senders and hence cannot be forged. Nodes receive signed *acknowledgements* for each message sent to the Hub, and the Hub signs each transcript.

4.1.1 Code distribution

Prior to issuing compiled tasks for the first time, the Hub must issue a special *instrumentation task* [15] which authoritatively supplements the *Task Devel-*

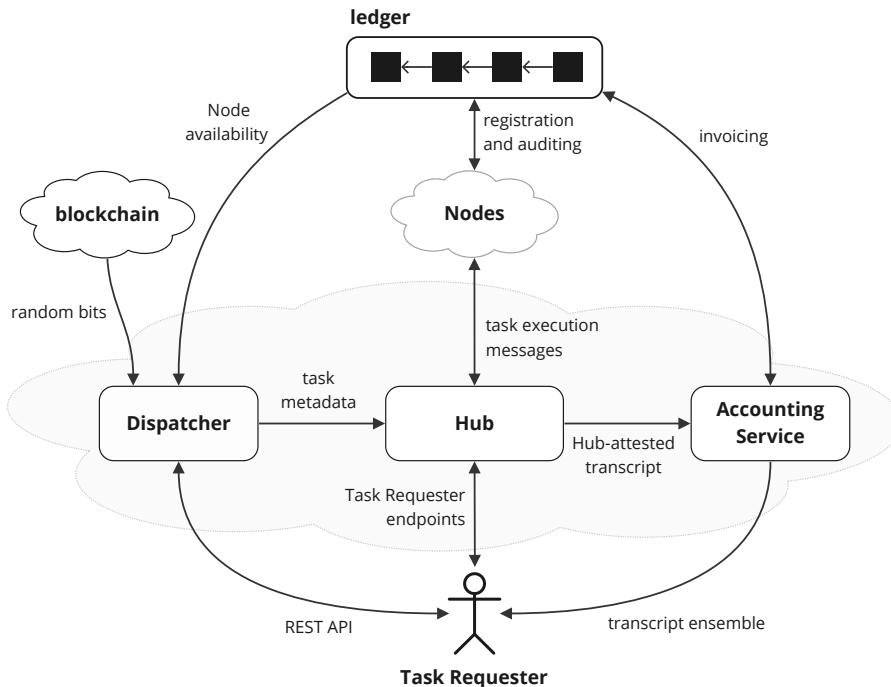


Figure 1: System schematic with information flows.

oper's code file with low-level verification hooks including checkpoints, metering, determinism, and, if not done already, filesystem calls (Section 6.2.1). The Hub then deploys the well-formed (instrumented) code file to the Nodes (Section 5.6.1).

4.1.2 Task issuance

Once instrumented code has been distributed, a *Task Requester* proceeds to broadcast the task's (optionally signed) inputs and metadata to the *Dispatcher* (Section 5.5). The Task Requester then awaits a response. Task *metadata* includes *content addressed* [11] links to large inputs.

4.1.3 Node registration

Nodes must register via ledger in order to be selected for participation in tasks. In order to disambiguate which Nodes were online at the moment of task issuance, the ledger timestamps each Node registration. Secondly, the ledger provides basic database operations, essentially smart contracts, which guarantee tamper-proof registration and *ex post facto* registration lookups with respect to historical blocks.

In more detail, the ledger maintains a one-to-one correspondence from registered Node *IDs* to integers so that a random set of integers indicates an independent set of Nodes. Two invariant, constant-time operations "registration" and "deregistration," detailed below, preserve the mappings' range as a set of consecutive integers from one through the number of registered Nodes and maintain the bijection between IDs and integers. The smart contract synchronously updates an array and a set of mappings such that, given an ID, x , and integer, n , the n^{th} array position contains x , and the mapping with input x returns n . In the descriptions below, the array mirrors all changes to the mappings.

Initialization. Initially, there are no mappings, and we define the maximum integer of the empty set to be zero.

Registration. When a Node registers, the smart contract creates a new mapping from the new Node's ID to the maximum integer plus one.

Deregistration. When a Node deregisters, the mapping from its ID to, say, integer n is destroyed, as is the mapping from some ID x to the maximum integer. Now a new mapping is created from x to n .

Lookup. During transcript verification, one makes simple queries to the ledger of the following form, “Give me the IDs corresponding to random bits 456, 232, 656, and 971 in historical registration epoch 80.” The ledger can now easily read off the corresponding IDs, and then a transcript observer can confirm that the task was executed by Nodes corresponding to those IDs.

4.1.4 Node selection

The Dispatcher selects a random subset (or union of subsets) of Nodes, optionally with predesignated restrictions or operators, from among those ledger-registered as being online. It bases its selection on random bits generated from the concatenation of three seeds: the Task Requester’s timestamp for message send, the Dispatcher’s timestamp for message receive, and finally a recent, unpredictable *block hash* [1] from a blockchain. These immutable, unpredictable, random bits, which are verifiable by anyone with access to the task transcript, also serve as optional inputs to the task itself.

4.1.5 Task broadcast

The Dispatcher relays the task metadata to each selected Node and can use *peer-to-peer* [25] methodology to confirm proper broadcast (Section 5.5).

4.1.6 Solution commitment

Each selected Node commits a signed, salted, and encrypted *solution* to the coordinating Hub before the task’s timeout. Nodes cannot see each others’ solutions, and the Hub acknowledges each solution received. Solutions which are too large to include in task metadata require a separate message from the Node which begets a further acknowledgement from the Hub (Section 5.6.2).

4.1.7 Transcript generation

The Hub decrypts each of the Nodes’ solutions and then generates a transcript consisting of task metadata, selected Node set, and the Nodes’ plain-text solutions. All solutions should be identical because instrumentation makes task execution deterministic. In the unlikely event that two Nodes disagree on a solution, they must play a *verification game* [29] (Section 5.2) to pinpoint the first divergent step in their computations. The Hub records the interactive dispute and outcome in the task’s transcript.

4.1.8 Node remuneration

The Hub submits its completed transcript to the *Accounting Service* where three fixed-time, ledger-based stages encompass the protocol’s final verification.

In the candidate *Invoicing Stage*, the Accounting Service first prescriptively annotates the most recent transcripts with appropriate economic parameters for Nodes and Task Requesters. It then batches these annotated transcripts, or *invoices*, together and *Merkle tree* [22] commits them to ledger along with updated account balances for Nodes. The Accounting Service provides participating Nodes with access to the newly batched invoices.

During the subsequent *Rebuttal Stage*, a Node may express *grievance(s)* regarding invoicing. If the Accounting Service fails to Merkle commit an invoice to ledger, fails to grant full invoice access to a participating Node, provides a transcript missing the Node’s solution or verification game records, or invoices incorrectly, then the Node can post a *rebuttal* to ledger in the form of a signed Hub acknowledgement or *time channel* (Section 5.1).

In the *Finalization Stage*, the Accounting Service responds to rebuttals by posting *Merkle proofs* [21], account balance updates, an invoice hash, or proof of message propagation (Sections 5.5–5.6) to ledger. Rebuttal and Finalization Stages may iterate within time bounds until all four grievance categories have been sequentially exhausted. If no Nodes presented objections during the Rebuttal Stage, as expected in the normal course of affairs, then the Accounting Service’s commitment from the Invoicing Stage automatically carries over to the Finalization Stage without additional writes to ledger.

At any moment, a Node or Task Requester can settle its outstanding balance, as represented in its most recent invoice participation, for standard *ERC-20* [14] tokens on Ethereum. All remaining balances reside on the Accounting Service’s queryable, internal accounting system. Any deviation from prescribed accounting principles yields a smoking gun due to ledger traceability.

4.2 Task Requester endpoints

Task execution returns different objects to the Task Requester depending on the task’s metadata specifications as detailed below.

4.2.1 Metatasks

A *metatask* occurs when outputs from a set of tasks feed into an input for a downstream task. Metatasks which link temporally contiguous tasks ensure consistent, data availability and enhanced performance by persisting Node participation across the entire execution sequence. Task Requesters can specify such execution dependencies through *directed acyclic graphs* [13] wherein the system pinpoints execution discrepancies by identifying the left-most branch of disagreement followed by the earliest, disputed task execution along that branch [35, Figure 6]. Non-contemporaneous executions, on the other hand, rely on prior transcripts' input and output commitments to verify downstream data authenticity *ex post facto*.

4.2.2 Hub-attested transcript

The Task Requester obtains a *Hub-attested* transcript describing the solutions provided by several Nodes. Either all solutions agree, or the transcript must include a valid description and outcome for a verification game [29] (Section 5.2).

4.2.3 Transcript ensemble

Following a Hub-attested transcript, the Accounting Service creates transparency by returning a finalized transcript ensemble. A complete transcript ensemble consists of a Hub-attested transcript wrapped inside a final invoice which includes pointers to invoice commitment(s) on ledger as well as any Node rebuttals and Accounting Service responses.

4.2.4 External data integration

A Task Requester can get data into or out of the system through certified interactions wherein selected Nodes together form a *joint session key* [17] which reads or writes data via an internet-facing *application programming interface* [2], or *api*. This method resembles Linux *curl*, and Task Requesters can pipe results into downstream tasks.

When conjoined with a transcript ensemble via metatask, *api adapter* transcripts constitute self-contained, succinct, sustainable proof that, even without persisting access to the original input, data served from an authenticated *api* exhibits certain, user-defined characteristics. Similarly, one can definitively memorialize the result of feeding one *api*'s response into another *api* at a particular moment in time.

4.2.5 Snap response

In *snap response* mode, the Task Requester obtains a synchronous solution from a single, random Node along with a corresponding transcript.

4.3 Economics don't lie

To summarize, after each task execution, the Accounting Service publicly invoices Nodes for their participation (Section 4.1.8). Rational Nodes will dispute errant invoices, and public invoice disputes become part of the final transcript ensemble. In this way, economics error-correct transcripts and deliver finality to tasks. Forms of meritorious evidence include, as detailed in the next section, signed acknowledgements from the Hub, time channels, proof of broadcast censorship, and evidence of prematurely revealed information.

5 Things that can't go wrong

We observe system resilience against critical attack vectors. In all cases, the economic reward for correctly reporting a smoking gun substantially dominates other task rewards and slashes.

5.1 Solution censorship

The Hub cannot censor solutions. Indeed, if the Hub were to fail to acknowledge a Node's solution, that Node could simply time-channel its overlooked message. Were the Hub to acknowledge a Node's solution but omit it from a task transcript or garble the transcript itself, a rational Node would then present the Hub acknowledgement to the Accounting Service as smoking gun evidence of transcript tampering. Economic incentives thus prevent undetected solution censorship. Under normal operation, however, such errors do not arise because the Hub acknowledges and records all valid messages.

The solution recording process illustrates two quintessential methods for achieving transparency. First, a transcript omitting a Hub-acknowledged solution points clearly to a smoking gun. Second, time channels make censorship of short messages impossible. Note that a time channel in isolation can't distinguish a negligent Node from an unresponsive Hub.

5.2 Conflicting solutions

When Nodes present two or more inconsistent solutions, the Hub must append to the transcript verification game [29] component(s) each consisting of an interactive proof pinpointing the source of disagreement between disparate execution pairs. By design, each pair agrees on the initial, virtual machine state but disagrees on the final one. The initial state depends on code but not runtime inputs as the latter are read in through the course of execution.

Each verification game proceeds via *binary search* [6] At first, one *player* presents a solution while the other declares whether it agrees at the midpoint computation step, $n/2$, thereby slicing the area of contention in half and bringing the next step in question to either $n/4$ or $3n/4$, etc. After $\log n$ iterations, only a tiny, single step remains. The winner of each verification game receives a reward, and the loser gets slashed. When a task requires more than one verification game, each game is played in sequence, and each Node gets penalized for a flawed solution at most once.

Absent Hub acknowledgement, time channels can still definitively prove that a participating verification game player responded in a timely manner by the method of Section 5.1. In case the Hub were to fail to relay a message, the underinformed player would replay its last confirmed move via time channel, the opponent would respond on ledger, whereafter the game would resume on Hub. In either scenario, time channels prevent censorship.

5.3 Collusion on Node selection

Suppose that the Task Requester were to collude with the Dispatcher on Node selection with the goal of selecting a friendly Node set which mutually agreed on a bogus solution for a task. While the colluding pair could choose a pair of favorable timestamps for random bit seeds, they have limited time to make the selection because the third seed consists of a recent and unpredictable block hash (Section 4.1.4). A successful collusion between Task Requester and Dispatcher might go undetected in the transcript ensemble and therefore violate transparency. Thus we are left to argue for infeasibility of such collusion. We remark that neither the Task Requester nor the Dispatcher can easily cheat independently.

Let us calculate the probability of guessing a suitable pair of timestamps

over a fixed time interval. Let

- t = time required to check one guess,
- b = length of time interval for guessing,
- k = number of Nodes assigned to the task, and
- a = fraction of Nodes influenced by the Task Requester.

For concreteness, t is essentially the time to concatenate the three “random” seeds and calculate their hash. The number of guess opportunities is $\lfloor b/t \rfloor$, and the chance of successfully obtaining a complete set of influenced Nodes on a single guess is bounded above by a^k . Therefore the chance of failure is at least $1 - a^k$, and the chance of failing $\lfloor b/t \rfloor$ times in a row is no less than

$$p = \left(1 - a^k\right)^{\lfloor b/t \rfloor}.$$

Hence the chance of successful attack over time interval b is at most $1 - p$.

The chance of success depends on one’s tolerance for cost, speed of executing the task, attacker compute resources, and willingness to tolerate a possible *orphaned* [23] block. Consider the following example parameters.

- $t = 10$ milliseconds,
- $b = 9$ seconds,
- $k = 8$ Nodes, and
- $a = 7\%$ of the network.

In this case, the probability of a successful collusion during time interval b is roughly 5×10^{-7} .

Taking into account advertised *time to finality* (TTF) [10] for some popular blockchains, in Table 1 we see that selecting relatively few Nodes suffices to render this attack ineffective in practice under the conservative assumption that the attacker learns the block hash immediately after the corresponding block broadcast.

5.4 Are spoofed transcripts possible?

Unfortunately, there exist ways to syntactically generate transcripts which have nothing to do with actual task execution. For example, one could simply tweak a valid transcript using a text editor. Naïve transcript edits invalidate Dispatcher, Hub, and Node signatures, so let us restrict our analysis to the case where these parties cooperate with the attack. Such

Selected Nodes (k)	% Colluding Nodes (a)	Time Interval (b)	Probability of Success ($1 - p$)
9	10%	6.5 sec <i>Cosmos</i>	7×10^{-7}
12	20%	2.1 sec <i>Avalanche</i>	9×10^{-7}
27	50%	0.9 sec <i>Aptos</i>	7×10^{-7}

Table 1: Sample collusion attack parameters (with $t = 10$ ms).

cooperation might indeed go undetected during inspection of a transcript ensemble and hence avoid a smoking gun. We remark that the protocol uses publicly verifiable random bits for Node selection, as opposed to having Nodes volunteer for tasks, because the latter approach makes transcript synthesis too easy.

Consider the following offline manipulation of random bits and timestamps which aims to generate a bogus but valid transcript.

1. The attacker chooses a block hash at some convenient, historical time during which colluding Nodes were registered.
2. The attacker keeps guessing timestamps offline until it finds one that selects the colluding Node set so that when the cooperating Dispatcher signs, Node selection looks legitimate.
3. The selected Nodes provide bogus, signed solutions, and the Hub signs these solutions into a transcript, even though the timestamps are not current.

An observer of this transcript can't tell that it's fabricated because the transcript is syntactically valid. The system requires two items which make this attack infeasible.

- a) The Hub submits its transcript to the Accounting Service immediately once available.
- b) The Accounting Service promptly commits the associated invoice to ledger.

Unlike the Hub and Dispatcher, which operate outside the ledger, the Accounting Service must publicly commit an authentic timestamp. If its timestamp contradicts those supplied by the Dispatcher and Hub, then transparency strikes with a smoking gun witnessing the anomaly.

Still, how difficult is it to convincingly synthesize such a transcript? While neither the Hub nor the Accounting Service can delay much due to the Accounting Service’s requisite commitment to ledger, the attacker could, and most likely would, precompute the bogus solution. The time allotted for solving the task can therefore be repurposed for guessing random bits, which brings us back to the collusion analysis from Section 5.3. If we assume that the maximum task timeout is 600 seconds and that the Accounting Service has 60 seconds to batch its final transcript for commitment to ledger, then absorbing these additional delays into the block delay b and keeping other parameters the same, the cooperating Hub, Dispatcher, and friends have roughly 4×10^{-5} chance of successfully executing this attack during the augmented time interval b .

By the same token, timestamp manipulation is, roughly speaking, transparent. During an actual task execution, the Hub cannot slide timestamps backwards by much, lest the Accounting Service miss its window for committing its invoice to ledger. The Dispatcher also cannot slide timestamps forward much because it can’t guess future block hashes.

5.5 Lazy Dispatcher versus red balloons

Faulty task announcements enable injection of unchallengeable solutions into a transcript while creating an illusion of unresponsive Nodes being offline. In order to cast light upon a smoking gun for such censorship, the system must reliably estimate the likelihood that a *non-atomic broadcast* [4] caused a missing solution.

While no proof of broadcast to almost all Nodes can vindicate a Dispatcher who knows exactly which few selected Nodes to censor, methodology employed in the DARPA Red Balloon Challenge suggests an appropriate alternative based on economic incentives and peer-to-peer dissemination [28]. At the moment of task broadcast, a few *red balloons* randomly become hidden behind a few Nodes in a way that no one, including the Dispatcher, knows their locations. A *recursive incentive structure* [5, 28] then rewards gossiping of the task broadcast through selected solver Nodes and towards red balloons. We remark that this anti-censorship scheme overlaps in effect with and complements time channels.

A *successful, directed path* [24] includes at least one selected solver Node

and ends at a red balloon. According to the *1/2-split contract* method [7] a Node sitting i hops from the red balloon along a successful path receives $1/2^{i+1}$ fraction of the total reward for that path. The astute reader may note that this distribution allocates only

$$\sum_{i=0}^{n-1} \frac{1}{2^{i+1}} = \sum_{i=1}^n \frac{1}{2^i} = 1 - \frac{1}{2^n}$$

fraction of the reward, where n is the number of Nodes in the successful path. The remaining 2^{-n} can be split among the non-origin Node(s) along the successful path to incentivize them not to erase preceding path edges.

Empirical results show [28] that the system above successfully incentivizes cooperation in red balloon hunts. While *Sybil* attacks [27] on the *1/2-split contract* scheme above can nearly double a gossiping Node’s gains, they do not impact downstream rewards. Moreover, bounding the length of successful paths, heavily weighting rewards toward the end of the successful paths, and paying significantly greater rewards for shorter, successful paths can eliminate Sybil attacks entirely [5, 8].

Prior to the task broadcast, say, once a week, each Node randomly chooses a list of 256-bit *lottery tickets*, sticks the lottery tickets into the leaves of a Merkle tree, computes the Merkle root of those leaves [22], and commits the root to ledger. Each Node can provide at most one root per week. Any Node that shares its raw lottery tickets or makes leaves too easy to guess has compromised its lottery claims, hence each Node has incentive to keep its lottery tickets private.

At the time of task issuance, the system randomly selects, using a block hash and a pair of task timestamps (Section 4.1.4), both a Merkle leaf position and some lottery prefixes. A Node has a red balloon if it can show a ledger-committed Merkle root whose lottery ticket at the randomly chosen leaf position matches one of the randomly selected lottery prefixes. There are sufficient leaves in each Merkle tree so that the same lottery tickets are unlikely to be called twice during the week.

In order ensure atomic broadcast, the Dispatcher broadcasts task announcements widely and timely to all Nodes. Under normal circumstances, this broadcast suffices and all Nodes respond to the Hub with their solutions. In the meantime, however, if a Node holding a red balloon ends up with signed evidence of a successful path passing through a selected Node, then it timely submits that path to the Hub via *commit-reveal*. If the Hub does not acknowledge the path, then the red balloon Node can *time-channel*.

Finally, let us attempt to discern whether a missing solution from a

particular Node N derives from Dispatcher censorship or N being offline. Let r be the number of successful paths reported for a given task, and let k be the number of selected solver Nodes for the task. If we assume that each independent, successful path is as likely to pass through one selected solver Node as another, then the chances of each successful path not passing through an online N is bounded above by $1 - 1/k$, hence the chance that none of the successful paths include N is

$$\left(\frac{k-1}{k}\right)^r.$$

For concreteness, if $k = 5$ selected Nodes and $r = 7$ successful paths, then there is more than a 21% chance that all paths would have missed N , which means N was likely offline. With the same $k = 5$ but $r = 3$ successful paths, we reach a 51% chance that all successful paths missed N , and therefore we more likely have a smoking gun against the Dispatcher.

While the expected number of successful paths must always be chosen large enough so as to absolve the Dispatcher, any effort made by the Dispatcher to decrease the number of successful paths can only tip probability more towards a smoking gun. Finally, N must be online whenever it belongs to a successful path because each included Node must sign its respective segment of the path, at which moment it reveals its peer-to-peer identity to the other Nodes. Bribery aimed at preferring a successful path with a particular selected solver Node over another is relatively ineffective: once a selected Node has heard about the task, censorship has already failed.

As a last resort, there exists a natural moment for the Task Requester to reissue its task. As depicted in Section 5.4, the Accounting Service cannot remunerate Nodes for performing tasks whose timestamps have expired.

5.6 A filesystem broadcast protocol

At first glance, it might seem that manifesting file retrievability reduces to the atomic broadcast scenario from Section 5.5. Large files, however, neither propagate like task metadata nor stick to ledgers. While the Dispatcher broadcasts tasks, the Hub and Nodes broadcast or exchange input, code or solution files. Also, in contrast to the task announcement scenario, future selected Nodes remain unpredictable during file broadcast. Appealing to standard cloud storage doesn't clarify who is to blame when a file goes missing because a broadcaster can pretend to use that service while in reality restricting its distribution to friendly Nodes.

5.6.1 Code files and large inputs

For performance and data availability reasons, code files and large inputs must be distributed to all relevant Nodes prior to use in tasks. The Dispatcher is, of course, blinded as to which Nodes will eventually be selected to execute these tasks. Each time a Task Developer uploads a code file to the Hub, or a Task Requester uploads an input file, the Dispatcher randomly selects (Section 4.1.4), say, five Nodes and asks whether each has succeeded to download the file. In case of a “no” or non-response, the Dispatcher repeats the process with five fresh Nodes. If the file was a code file which requires instrumentation, then the five confirmed Nodes immediately perform an instrumentation task. If the file was an input file, then the first time it gets used in a task, the Dispatcher obligates one of the corresponding Nodes to participate in that task. In each case, a “file not found” error from a Node which claimed to have previously downloaded the file results in a slash and file rebroadcast. If all five confirmed Nodes become unregistered prior to task issuance, the Hub rebroadcasts the file.

Nodes may choose to gossip code and input files to each other in order to increase the likelihood of receiving a bonus when all selected Nodes provide solutions. A selected Node does not know *a priori* the other selected Nodes, so gossiping to everyone is a good strategy.

5.6.2 Large solutions

Suppose that a Node produces a solution too large to fit in a time channel. The Node then must transmit to the Hub both a hash commitment of its solution for inclusion in transcript and the large file itself. In the event of discrepancy, the Node calls, by way of time channel, for a red balloon challenge (Section 5.5) and propagates the large file to the entire network of relevant Nodes. The selected solver and terminal Nodes in a successful path each provide signed chunks of the Merklized solution as proof of possession. The Hub provides its own address as one of the selected Nodes for this challenge, and if the Hub acquires the solution through the course of the Nodes’ peer-to-peer propagation, then all is well. Otherwise, the challenge results likely point to a smoking gun. This situation arises rarely because, whenever selected Nodes agree on a solution, the Hub need only receive the solution from one of them.

5.7 Deregistering unresponsive Nodes

In case a selected Node fails to respond to a task, as evidenced by a transcript ensemble including successful paths (Section 5.5), the Hub deregisters the unresponsive Node while citing that task on ledger. Both Hub failure to perform required deregistration and rogue Hub deregistration induce smoking guns relative to the transcript ensemble backdrop. The penalty for Node non-response exceeds the cost of losing a verification game so as to discourage coordinated solution disappearances.

5.8 On slightly misplaced timestamps

We argued in Section 5.4 that timestamps can't be moved by a significant amount, but what about smaller perturbations? If the Task Requester were to use a slightly future timestamp, then overall task processing is just delayed. If the Task Requester's timestamp were too early, then the Task Requester censors its own task because the Dispatcher can't broadcast the task without incurring a "lazy Dispatcher" smoking gun (Section 5.5).

We claim that the true, real-time timeout for each task actually coincides with the timeout implied by the Dispatcher's timestamp whenever the Task Requester's timestamp accurately reflects task issuance. First, observe that the Dispatcher's timestamp cannot differ much from the Task Requester's timestamp without triggering a smoking gun. If the Dispatcher were to delay task announcement, then again we land in the "lazy Dispatcher" case. Finally, the Dispatcher cannot physically announce the task earlier than its claimed timestamp since its timestamp coincides with the Task Requester's timestamp which, by assumption, coincides with actual task issuance.

5.9 Premature reveals

An agent who prematurely reveals information may influence transcript outcome without presenting a smoking gun. We remark that only selected Nodes need witness task correspondences with the Hub. According to protocol, neither the Hub nor Nodes access superfluous information.

All data related to a task at a given moment is either *public* or *premature*, meaning one can draw a bright line as to who was responsible for an information leak. For example, data or a transcript returned to the Task Requester immediately becomes public because the Task Requester may do as it pleases. It follows that the Task Requester cannot securely request both a synchronous snap response and Hub-attested transcript within a single

task because a fast, synchronous response might reclassify data and hence influence the results of the Hub-attested transcript.

The Accounting Service considers only cryptographically signed messages as definitive evidence of premature reveal. Significant rewards exist for time-channeling premature reveals, including the following.

1. A Node shares a signed, plaintext, rather than an encrypted solution, to the Hub. Such a solution becomes available prior to transcript release.
2. The Hub decrypts, signs and shares a Node’s plaintext solution prior to returning the corresponding transcript to the Task Requester and Accounting Service.

5.10 Bifurcated invoices

We argue that the Accounting Service cannot serve more than one invoice per transcript. Suppose that during the Invoicing Stage the Node observed an invoice paying it a positive reward, but there was another invoice batched for the same task slashing the Node. Since the Node sees all tasks batched in the ledger commitment, it raises an objection during the Rebuttal Stage. The Accounting Service must then indicate, during the Finalization Stage, a unique invoice for the task, and only that single invoice, along with any Node audits, is authoritative.

If the Node’s task is missing altogether from any batch, then after the task times out, the Node can raise an objection. We remark that the Accounting Service cannot batch a task prior to task timeout without bearing a residual smoking gun.

6 Platform architecture

Figure 2 depicts system functionality in three implementation components: *audit ledgers*, *coordination services*, and *Nodes*. Audit ledgers, which live on blockchains, underpin transparency and facilitate incentives. Cloud-based coordination services deploy tasks, oversee execution, record activities, and analyze outcomes. Nodes provide independent computation substrates.

6.1 Audit ledgers

The system uses several *ledger types*, each with unique characteristics, which cumulatively achieve transparency. We proceed to list the ledger types used

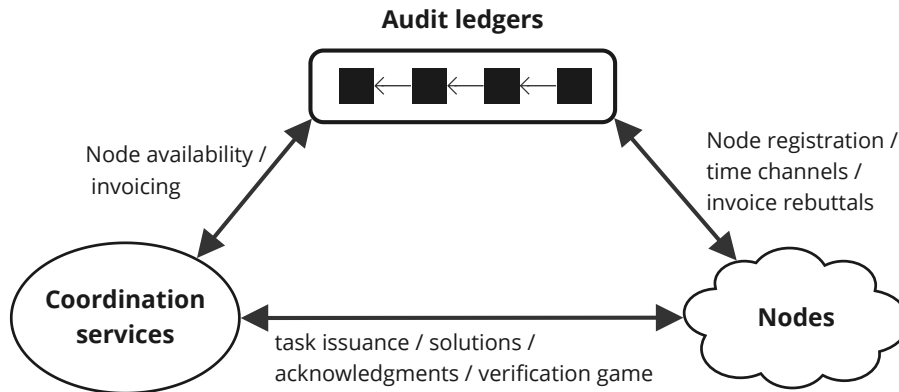


Figure 2: Platform overview.

by the protocol.

Node registration establishes node identity and availability.

Task registration advertises task availability and name.

Lottery tickets are commitments for receiving red balloons.

Randomness provides the block hash for random bits.

Time channel records authoritative timestamps for protocol events.

Invoice records Node payments and penalties.

Evidence records objections to invoices.

Payment executes financial settlement.

Each audit ledger has unique technical requirements for its underlying blockchain. We define these criteria and then match them to ledgers in Table 2.

Security. Decentralization ensures unpredictable block hashes.

Speed. Short, expected TTF prevents attacks from Sections 5.3–5.4.

Precision. Higher frequency blocks allow for more precise timestamps.

Variance. Low TTF variance.

Contracts. Supports simple database operations.

History. Supports historical queries to contract state.

Reliability. Low occurrence of dropped or delayed transactions.

Cost. Low expense associated with transactions.

Storage. Maximum data attachment per transaction.

Availability. Low chance that data might someday disappear.

Economics. Enables monetary value transfer.

Ledger type	<i>Contracts/History</i>	<i>Security/Speed</i>	<i>Precision</i>	<i>Variance</i>	<i>Reliability</i>	<i>Cost</i>	<i>Storage</i>	<i>Availability</i>	<i>Economics</i>
Node registration	●			●		◐	◐	●	
Task registration						◐	◐	●	
Lottery tickets						◐		●	
Randomness		●		●				●	
Time channel			●		●	●	●	●	
Invoice					◐	◐	◐	●	
Evidence					◐	●	●	●	
Payment									●

Table 2: Primary considerations for ledgers.

6.2 Preparing a task

In addition to metadata, each task includes a *codefile* in the form of an instrumented WebAssembly (WASM) module [32]. For performance reasons, Nodes execute tasks using *just-in-time compilation* [18] but fall back to *interpreters* [16] for observation during verification games.

6.2.1 Instrumentation

A canonically instrumented, WASM-based interpreter streamlines task creation directly from interpreted code (e.g., JavaScript or Python). One need only feed the source code along with its rolled up dependencies and program arguments directly as inputs into the precompiled interpreter codefile.

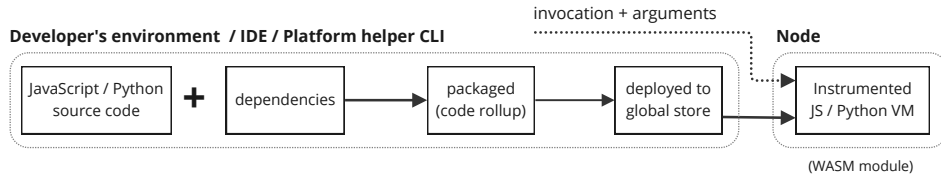


Figure 3: JavaScript and Python instrumentation pipeline.

For compiled languages (e.g., Rust or C++), on the other hand, the Task Developer must first compile to WASM using a standard toolchain. The Task Developer then obtains the final codefile by issuing to the network a special, precompiled instrumentation task which adds to the WASM file checkpointing, metering, initial state, filesystem hooks, and handling for non-determinism including floating point and NaN. The system can instrument compatible WASM modules without access to source file(s).

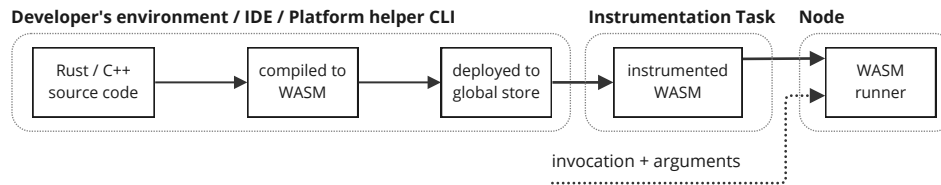


Figure 4: Rust and C++ instrumentation pipeline.

One might wonder why the Task Developer issues an instrumentation task in production rather than instrumenting locally. Suppose that each selected Node were to locally instrument its own WASM file. Even though the instrumentation process is deterministic, the system would have no way to achieve correct consensus on task execution were a Node to deviate from the instrumentation protocol. Indeed, one cannot initiate a verification game without agreement on initial state. Moreover, a Node could distort metering injections in order to claim payment for more work than it actually performed. Lastly, task execution would incur delays from runtime instrumentation. Even if some Nodes could cache the instrumented file ahead of time, not all selected Nodes would enjoy that luxury.

6.2.2 Pre-deployment

Prior to engaging coordination services, a Task Developer commits *task definitions* including task name, network routing endpoints, codefile content

address, and either default or custom metadata to a *task registration* ledger. A locally hosted *command-line interface* [9], or cli, streamlines task registration, deployment and other common operations.

6.3 Coordination services

Coordination services live in horizontally-scaled *containers* [34] on internet-attached hosts. This cloud based approach affords speed, cost efficiency, and network connectivity capabilities not present in blockchains. Coordination services communicate with Nodes via the Hub’s *message broker* and interact with audit ledgers through *ledger adapters* as illustrated in Figure 5.

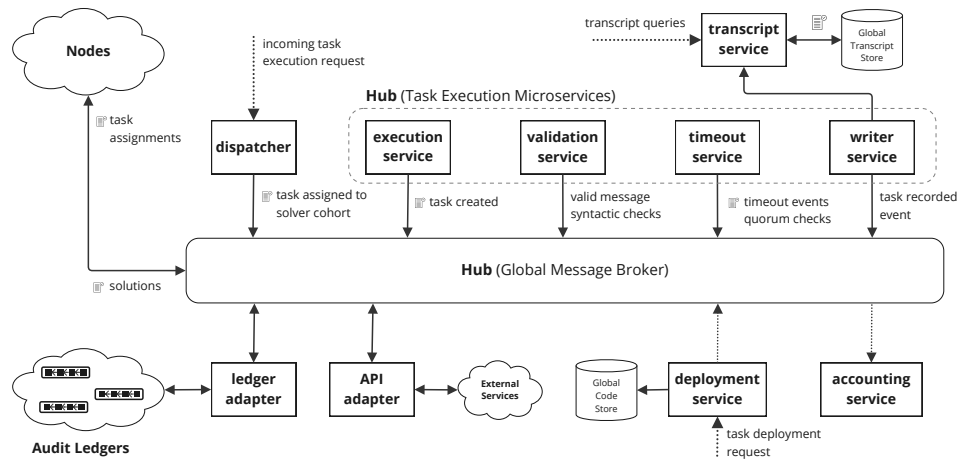



Figure 5: Hub and related coordination services.  denotes signed message.

6.3.1 Deployment

The *deployment service* writes registered task code to content-addressable *global code storage*. The Task Developer deploys task code to the *deployment service* which initiates task instrumentation. As Task Developers post code to the service through signed **https** api requests, their task registration details synchronize with the task registration ledger. This service initiates instrumentation of the deployed code by calling the instrumentation task.

6.3.2 Dispatcher

The Dispatcher processes task requests from Task Requesters and provides ingress interfaces for network protocols. The Dispatcher manages Node se-

lection, including collation of random bits from block hash and timestamp sources. After Node selection, the Dispatcher signs the task request and sends it to the Hub.

6.3.3 Hub

The Hub consists of a *global message broker* plus the *microservices* below which facilitate task execution.

Execution service. After receiving a task request from the Dispatcher, the Hub's *execution service* assigns a task ID, broadcasts the task to each assigned Node through the message bus, timestamps Node responses, and records errors. When the Task Requester calls for a snap response, the execution service provides it to the Dispatcher to close out the Task Requester's synchronous request.

Validation service. The *validation service* checks for syntactic correctness of responses and protocol adherence. Key checks include valid signatures, valid execution ID, valid Node address, message schema validation, and absence of duplicate messages.

Timeout service. The *timeout service* plays timekeeper, concurrently ensuring that each task adheres to the protocol's time limits. In addition it checks that necessary quorums, such as number of received solutions, and *events*, such as solution reveals, transpire before expiration.

Writer service. Once transcript messages are persisted in the Hub's database, the *writer service* passes events in the order received to the execution service.

Ledger adapter. The *ledger adapter* abstracts interaction with the various ledgers from the rest of the Hub. It also listens to relevant ledger events and forwards them to the appropriate queue inside the Hub for processing.

api adapter. The api adapter interacts with external interfaces.

6.3.4 Transcripts

The *transcript service* records and retrieves Hub-attested transcripts to and from *global transcript storage*. Cryptographic signatures ensure the integrity of each transcribed event, while a signed hash of the entire transcript certifies

Hub attestation. The Hub’s public key appears on the Node registration ledger.

6.3.5 Accounting

The Accounting Service calculates and issues payments and penalties according to Hub-attested transcripts and transcript ensembles. Its roles includes semantic validation of transcript ensembles, posting invoices to the invoice ledger, and processing settlements on the payment ledger.

6.4 Inside a Node

Nodes are compute containers running on internet-attached hosts. They register their identity, network address and related metadata on the Node registration ledger through a ledger adapter.

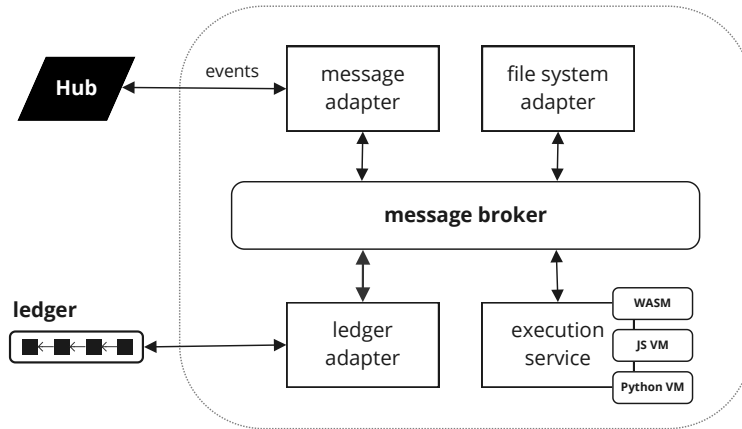


Figure 6: Node orchestration.

An *Orchestrator* instantiates processes running on each Node, and a *local* message broker provides inter-process communication. The *file system adapter* retrieves task code from global code storage and persists it in local storage. The *event adapter* listens for task requests and commits responses to the Hub’s message bus.

The Orchestrator starts a unique WASM *virtual machine* labeled by the content address of the code specified in the task request, and it terminates the virtual machine when execution completes. Multiple virtual machines can run concurrently, up to the compute container’s provisioned, processing capacity.

The event adapter monitors Hub messages and adds any disputed events to the time channel ledger through the ledger adapter. Finally, the *invoice monitoring service* receives invoice notices from the Accounting Service and commits any objections to the evidence ledger via the ledger adapter.

Acknowledgement

We thank Sami Mäkelä for helpful pointers regarding missing transcripts and faulty task broadcasts.

References

(last accessed Mar. 3, 2024)

- [1] Andreas M Antonopoulos. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*. O'Reilly Media, Inc., 2014. Chap. 7. ISBN: 9781449374044. URL: <https://www.oreilly.com/library/view/mastering-bitcoin/9781491902639/ch07.html#:~:text=The%20primary,compute%20it>.
- [2] *API*. URL: <https://en.wikipedia.org/wiki/API>.
- [3] Hazeline Asuncion. *About Data Provenance*. URL: <https://faculty.washington.edu/hazeline/ProvEco/generic.html>.
- [4] *Atomic broadcast*. URL: https://en.wikipedia.org/wiki/Atomic_broadcast.
- [5] Moshe Babaioff, Shahar Dobzinski, Sigal Oren, and Aviv Zohar. “On bitcoin and red balloons”. In: *Proceedings of the 13th ACM Conference on Electronic Commerce*. EC '12. Valencia, Spain: Association for Computing Machinery, 2012, pp. 56–73. ISBN: 9781450314152. DOI: 10.1145/2229012.2229022. arXiv: 1111.2626 [cs.GT].
- [6] *Binary search algorithm*. URL: https://en.wikipedia.org/wiki/Binary_search_algorithm.
- [7] Manuel Cebrian, Lorenzo Coviello, Andrea Vattani, and Panagiotis Voulgaris. “Finding red balloons with split contracts: robustness to individuals’ selfishness”. In: *STOC '12*. New York, New York, USA: Association for Computing Machinery, 2012, pp. 775–788. ISBN: 9781450312455. DOI: 10.1145/2213977.2214047. URL: <https://web.media.mit.edu/~cebrian/qin.pdf>.

- [8] Wei Chen, Yajun Wang, Dongxiao Yu, and Li Zhang. “Sybil-proof mechanisms in query incentive networks”. In: *Proceedings of the Fourteenth ACM Conference on Electronic Commerce*. EC '13. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 2018, pp. 197–214. ISBN: 9781450319621. DOI: [10.1145/2482540.2482588](https://doi.org/10.1145/2482540.2482588). arXiv: [1304.7432](https://arxiv.org/abs/1304.7432) [cs.GT].
- [9] *Command-line interface*. URL: https://en.wikipedia.org/wiki/Command-line_interface.
- [10] *Comparing crypto network’s time-to-finality (TTF)*. Feb. 2023. URL: <https://twitter.com/MessariCrypto/status/1631678346722529282>.
- [11] *Content Addressing: What It Is and How It Works*. URL: <https://fission.codes/blog/content-addressing-what-it-is-and-how-it-works/>.
- [12] Anthony Cuthbertson. *Bitcoin creator Satoshi Nakamoto now 15th richest person in the world*. 2021. URL: <https://www.independent.co.uk/tech/bitcoin-satoshi-nakamoto-wealth-net-worth-b1957878.html>.
- [13] *Directed acyclic graph*. URL: https://en.wikipedia.org/wiki/Directed_acyclic_graph.
- [14] *ERC-20 Token Standard*. URL: <https://ethereum.org/en/developers/docs/standards/tokens/erc-20/>.
- [15] *Instrumentation (computer programming)*. URL: [https://en.wikipedia.org/wiki/Instrumentation_\(computer_programming\)](https://en.wikipedia.org/wiki/Instrumentation_(computer_programming)).
- [16] *Interpreter (computing)*. URL: [https://en.wikipedia.org/wiki/Interpreter_\(computing\)](https://en.wikipedia.org/wiki/Interpreter_(computing)).
- [17] *Introduction — tln-docs*. URL: <https://docs.tlnotary.org/#%E2%91%A0-multi-party-tls-request>.
- [18] *Just-in-time compilation*. URL: https://en.wikipedia.org/wiki/Just-in-time_compilation.
- [19] Aleksandar Kuzmanovic. “Net Neutrality: Unexpected Solution to Blockchain Scaling”. In: *Communications of the ACM* 62.5 (Apr. 2019), pp. 50–55. ISSN: 0001-0782. DOI: [10.1145/3312525](https://doi.org/10.1145/3312525).

- [20] Leslie Lamport, Robert Shostak, and Marshall Pease. “The Byzantine Generals Problem”. In: *ACM Transactions on Programming Languages and Systems* 4.3 (July 1982), pp. 382–401. ISSN: 0164-0925. DOI: [10.1145/357172.357176](https://doi.org/10.1145/357172.357176).
- [21] *Merkle proofs Explained*. URL: <https://medium.com/crypto-0-nite/merkle-proofs-explained-6dd429623dc5>.
- [22] *Merkle tree*. URL: https://en.wikipedia.org/wiki/Merkle_tree.
- [23] *Orphaned Block Definition*. URL: <https://coinmarketcap.com/academy/glossary/orphaned-block>.
- [24] *Path (graph theory)*. URL: [https://en.wikipedia.org/wiki/Path_\(graph_theory\)](https://en.wikipedia.org/wiki/Path_(graph_theory)).
- [25] *Peer-to-peer*. URL: <https://en.wikipedia.org/wiki/Peer-to-peer>.
- [26] *Smoking gun*. URL: https://en.wikipedia.org/wiki/Smoking_gun.
- [27] *Sybil attack*. URL: https://en.wikipedia.org/wiki/Sybil_attack.
- [28] John C. Tang, Manuel Cebrian, Nicklaus A. Giacobe, Hyun-Woo Kim, Taemie Kim, and Douglas “Beaker” Wickert. “Reflecting on the DARPA Red Balloon Challenge”. In: *Communications of the ACM* 54.4 (Apr. 2011), pp. 78–85. ISSN: 0001-0782. DOI: [10.1145/1924421.1924441](https://doi.org/10.1145/1924421.1924441).
- [29] Jason Teutsch and Christian Reitwießner. “A Scalable Verification Solution for Blockchains”. In: *Aspects of Computation and Automata Theory with Applications*. Vol. 42. Lecture Notes Series, Institute for Mathematical Sciences, National University of Singapore. World Scientific, Nov. 2023, pp. 377–424. DOI: [10.1142/9789811278631_0015](https://doi.org/10.1142/9789811278631_0015). arXiv: [1908.04756](https://arxiv.org/abs/1908.04756) [cs.CR].
- [30] Apostolos Tzinas, Srivatsan Sridhar, and Dionysis Zindros. *On-Chain Timestamps Are Accurate*. Cryptology ePrint Archive, Paper 2023/1648. (to appear in Financial Cryptography and Data Security 2024). URL: <https://eprint.iacr.org/2023/1648>.
- [31] Michael Walfish and Andrew J. Blumberg. “Verifying Computations without Reexecuting Them”. In: *Communications of the ACM* 58.2 (Jan. 2015), pp. 74–84. ISSN: 0001-0782. DOI: [10.1145/2641562](https://doi.org/10.1145/2641562).
- [32] *WebAssembly*. URL: <https://webassembly.org>.
- [33] *What are smart contracts on blockchain?* URL: <https://www.ibm.com/topics/smart-contracts>.

- [34] *What is a Container?* URL: <https://www.docker.com/resources/what-container/>.
- [35] Zihan Zheng, Peichen Xie, Xian Zhang, Shuo Chen, Yang Chen, Xiaobing Guo, Guangzhong Sun, Guangyu Sun, and Lidong Zhou. *Agatha: Smart Contract for DNN Computation*. 2021. arXiv: [2105.04919](https://arxiv.org/abs/2105.04919) [cs.CR].