Mechanisms for Outsourcing Computation via a Decentralized Market

Scott Eisele
scott.r.eisele@vanderbilt.edu
Vanderbilt University
Nasville, TN

Taha Eghtesad
teghtesad@uh.edu
University of Houston
Houstan, TX

Nicholas Troutman
nicholas.troutman.pro@gmail.com
University of Houston
Houstan, TX

Aron Laszka
alaszka@uh.edu
University of Houston
Houstan, TX

Abhishek Dubey
abhishek.dubey@vanderbilt.edu
Vanderbilt University
Nasville, TN

ABSTRACT

As the number of personal computing and IoT devices grows rapidly, so does the amount of computational power that is available at the edge. Many of these devices are often idle and constitute an untapped resource which could be used for outsourcing computation. Existing solutions for harnessing this power, such as volunteer computing (e.g., BOINC), are centralized platforms in which a single organization or company can control participation and pricing. By contrast, an open market of computational resources, where resource owners and resource users trade directly with each other, could lead to greater participation and more competitive pricing. To provide an open market, we introduce MODɪCᴜM, a decentralized system for outsourcing computation. MODɪCᴜM deters participants from misbehaving—which is a key problem in decentralized systems—by resolving disputes via dedicated mediators and by imposing enforceable fines. However, unlike other decentralized outsourcing solutions, MODɪCᴜM minimizes computational overhead since it does not require global trust in mediation results. We provide analytical results proving that MODɪCᴜM can deter misbehavior, and we evaluate the overhead of MODɪCᴜM using experimental results based on an implementation of our platform.

CCS CONCEPTS

• Information systems → Computing platforms; **• Computing methodologies → Distributed algorithms**; **• Computer systems organization → Cloud computing**.

KEYWORDS

computation outsourcing, decentralized market, blockchain, decentralized job scheduling, smart contract

ACM Reference Format:
Scott Eisele, Taha Eghtesad, Nicholas Troutman, Aron Laszka, and Abhishek Dubey. 2020. Mechanisms for Outsourcing Computation via a Decentralized Market. In *The 14th ACM International Conference on Distributed and Event-based Systems (DEBS '20), July 13–17, 2020, Virtual Event, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3401025.3401737

1 INTRODUCTION

The number of computing devices—and thus computational power—available at the edge is growing rapidly; this trend is projected to continue in the future [20]. Many of these are end-user or IoT devices that are often idle since they were installed for a specific purpose, which they can serve without using their full computational power. Our goal is to harness these untapped computational resources by creating an open market for outsourcing computation to idle devices. Such a market would benefit device owners since they would receive payments for computation while incurring negligible costs. To illustrate, running an AWS Lambda instance with 512MB of memory for 1-hour costs \$0.03, while the electrical cost of operating a BeagleBone Black[1] single-board computer with 512MB of memory for an hour is 100 times less. Thus, it is feasible that a computation service could be provided economically.

Prior efforts to leverage these underutilized resources include *volunteer computing* projects, such as BOINC [10] and CMS@Home [19], in which users donate the computational resources of their personal devices to be used for scientific computation. Volunteer computing suffers from two limitations that prevent it from broader utility. First, the resources made available by volunteer computing participants are only accessible to specific users and projects. Second, it relies on systems "volunteering" their time as it does not include incentives to provide reliable access to computational resources, leading to the problem of low participation [23].

Participation can be incentivized through the implementation of a competitive market, which facilitates the discovery and allocation of supply and demand for computational resources and tasks. The market must provide mechanisms to address misbehavior and resolve any disputes[2] between the participants. Such a market could be managed by a central organization, as many in the sharing economy are (e.g., Uber, Airbnb). A central organization could mediate disputes. However, a centralized system presents a clear target for attackers, can be a single point of failure, and without competition may charge exorbitant fees. An alternative is to create an open

[1]https://github.com/beagleboard/beaglebone-black/wiki/System-Reference-Manual
[2]Disputes are disagreement between the parties about the correctness of the job execution. They may arise due to a fault or malicious behavior.

and decentralized market, where resource owners and resource users trade directly with each other, which could lead to greater participation, more competitive pricing, and improved reliability.

In distributed computing systems, faults and misbehavior are typically addressed using consensus algorithms. Recently, distributed ledgers have emerged as a novel mechanism to provide consensus in decentralized public systems [29]. Smart contracts extend the capabilities of a distributed ledger by enabling "trustless" computation on the stored data. In theory, smart contract implementations, e.g. the one used in Ethereum [28, 30], could be used to outsource complex computations. Since the computation is replicated on thousands of nodes, it is costly. To reduce costs, complex computations must be executed off-chain and only result aggregation, validation, and record keeping should be kept on the chain (e.g. [21]).

Prior efforts to construct outsourced computation markets using distributed ledgers include TrueBit [27], and iExec [4]. Unfortunately, these existing solutions have varying degrees of inefficiency due to extensive verification of the computation performed. There have been some efforts to ameliorate this situation. For example, TrueBit performs computation using a typical computer and relies on the distributed ledger only to complete disputed instructions; however, this approach is still quite inefficient. We discuss these existing solutions and their drawbacks in detail in Section 2.

Contributions: The key problem in implementing a decentralized market is the *efficient* resolution of disputes, which includes determining if the results of outsourced jobs are correct. This paper introduces *Mechanisms for Outsourcing via a Decentralized Computation Market* (MODiCuM), a distributed-ledger based platform for decentralized computation outsourcing. In contrast to other distributed-ledger based solutions (mentioned above), our approach retains computational efficiency by minimizing the amount of resources spent on verification through three ideas. First, it relies on *partially trusted mediators* for settling disputes instead of trying to establish global consensus on the computation results. Second, it *verifies random subsets* of results, which keeps verification costs low while supporting a wide range of jobs. Third, it deters misbehavior through *rewards and fines*, which are enforced by a distributed-ledger based smart contract. Thus, MODiCuM does not prevent cheating and misuse, but it deters rational agents from misbehavior. The specific contributions of this paper are as follows:

(1) We introduce a smart contract-based protocol and a platform architecture for incentivizing the participation of job creators, resource providers, and mediators.
(2) We present an analysis of the protocol by modeling the exchange of resources as a game, and show how we can select the values of various parameters (fines, deposits, and rewards), ensuring that honest participants will not lose, and the advantage of dishonest participants is bounded.
(3) We provide a proof-of-concept implementation of the protocol, built on top of Ethereum. Through comparison against AWS lambda we determine that due to the transaction costs currently associated with the main Ethereum blockchain, our implementation is currently suited only for very long running jobs. However, any improvements to Ethereum will benefit our platform. Additionally, our protocol is not limited to the use of a specific platform. Any platform that supports smart contract functionality can be utilized.

Outline: We begin by discussing related work in Section 2. Then, we introduce MODiCuM in Sections 3 and 4. We analyze the protocol in Section 5. In Section 6, we describe an implementation of our platform and provide experimental results on its performance. Finally, in Section 7, we present concluding remarks. Implementation is available at [6], and proofs and detailed specification can be found in our full paper [18].

2 RELATED WORK

In [17], the authors determine that for verifying outsourced computation the cryptographic approach is not practical and should instead use repeated executions. The computations however should not be duplicated more than twice. Their strategy is simple: outsource to two providers and compare results. The key challenge then is to prevent collusion. They propose *sabotaging collusion with smart contracts*. Essentially, they hold the providers accountable using security deposits and a smart contract. They also assume that if two providers intend to collude, the providers also use a smart contract to hold each other accountable. To counter this, the authors propose a third contract that states that the first provider who betrays the other, showing the colluder's contract as proof, is granted immunity and will receive a reward. In their work, they have not yet considered the scenario when the client may be an adversary. They also do not address the case when the contractors do not need a contract to trust each other.

The authors of [12] consider a case where there is a trusted third party that would be responsible for verifying a critical computation, except that it becomes a bottleneck for the rest of the system. It instead becomes a boss and outsources the verification task. To incentivize participation, the boss offers a reward, and to discourage misbehavior the boss requires a security deposit to enable it to enforce fines. The two verification strategies they consider are random double-checking by the boss and hiring multiple contractors to perform a job and comparing results. The authors do not consider the case when the boss is malicious or attempts to avoid paying out the rewards promised. They also do not discuss practical issues such as what if the contractors never return a result.

In iExec [16], the number of repeated executions required for verification is determined by a confidence threshold. After a result is computed, the pool scheduler checks if the results submitted achieve the desired level of confidence using Sarmenta's voting [25]. The workers register themselves to a scheduler that they choose to trust. If the scheduler breaks trust, the worker can leave to a competing pool. iExec checks tasks for non-determinism before allowing them to be deployed in the network. It does this by executing the task many times. This has the drawback that the task must take no inputs; otherwise, all execution paths would need to be tested.

In TrueBit [27], verification is provided via Verifiers, which duplicate the computation based on an incentive structure that rewards them for finding errors, and errors are intentionally injected into the system occasionally to guarantee benefits for verification. When a Verifier finds an error, it initiates a protocol where they compare the machine state at various points during the execution, in a manner similar to binary search. Once the step where the two

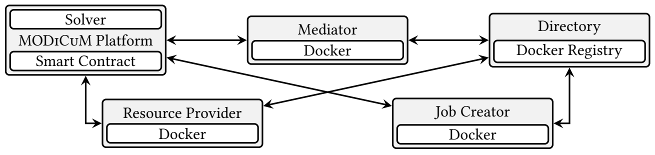


Figure 1: MODiCuM architecture. For ease of presentation, only one instance of Resource Provider, Job Creator, Mediator, Directory, and Solver is shown.

solutions deviate is found, the machine state is submitted to an Ethereum smart contract, which implements a virtual machine that acts as a mediator. The TrueBit virtual machine executes that specific step and determines which agent is at fault. The authors state that though TrueBit can theoretically process arbitrarily complex tasks, in practice the mediation is inefficient for complex tasks.

In each of these prior works, the authors assume that results must be globally accepted or have universal validity. In [27], the authors mention that using a trusted mediator is an option for resolving disputes regarding whether a task was done correctly, but such a solution is unacceptable because it does not provide universal validity. We argue however that many tasks—perhaps even the majority—do not require universal validity. For such tasks, only the agent who requested the execution of the task must be convinced of the validity of the result, which means that the systems proposed by prior work incur significant and unnecessary computational overhead for such tasks. Prior works also do not address the possibility that the job creator itself could manipulate the system by providing tasks that have non-deterministic or environment-dependent results.

3 MODICUM ARCHITECTURE

MODiCuM enables the allocation and execution of computational tasks on distributed resources that may be dishonest and may enter or exit the platform at any time. Conceptually, this requires resource management, i.e., the allocation of resources to maximize utility. We also need a service for managing job requests, which include both the job specification and the data products related to a job. Finally, we need a service that manages the market, which includes matching jobs to available resources, tracking the provenance of job products, accounting, and handling failures. Due to the decentralization of the system, *job creators* (JC) and *resource providers* (RP) can have their own strategies; however, the market must operate as a singleton.

To satisfy all of these requirements, we develop an architecture that consists of a distributed-ledger based *Smart Contract*, storage services (*Directory*), allocation services (*Solver*), *Resource Providers*, *Job Creators*, and *Mediators*. These components and actors are shown in Figure 1 and described in the following subsections. We start by describing first what a job is in MODiCuM.

3.1 MODiCuM Jobs

A job is a computing task that takes inputs and produces outputs. The jobs MODiCuM is designed to support are limited to deterministic, batch jobs that are not time critical, and are isolated, meaning they receive no information that allows them to discern which agent (RP, M) is executing them. We discuss the importance of our limitations in Section 4.4. Further the confidentiality of the job cannot be guaranteed since the computation is outsourced to any capable participant. It is essential that jobs can be executed independently of the host configuration. The state-of-the-art approach for achieving this is to use a container technology [15]. Consequently, we use Docker [22] images to package the jobs in our implementation. Docker containers provide an easy-to-use way of measuring and limiting resource consumption and securely running a process by separating the job from the underlying infrastructure. To avoid downloading a full image for every job execution, we support Docker layers. To execute an instance of a job, a base layer (i.e., OS or framework), an execution layer with job specific code, and a data layer are required. The base layer can be downloaded a priori, and the other layers are downloaded after a match. If the execution layer has already been downloaded, then only the new data layer is required. The job requirements, including the required resources, are specified in the offers that the agents make. These requirements are used by the resource provider to set the Docker container's resource constraints at runtime.

The cost of running a job depends on the amount of resources used. The resources we consider for feasibility are instruction count (computes from CPU speed and time on CPU), disk storage, memory, and bandwidth for downloading the job. The resources we use to compute the price are the instruction count and bandwidth used as reported by the RP where each is multiplied by the RP's asking price. Thus, the price of the job is calculated as follows:

$$\begin{aligned}\pi_c =& \mathit{result.instructionCount} \cdot \mathit{resourceOffer.instructionPrice} \\ &+ \mathit{result.bandwidthUsage} \cdot \mathit{resourceOffer.bandwidthPrice}\end{aligned} \tag{1}$$

Note that it is often difficult to estimate resource requirements precisely. Repeated executions of the same job result in approximately the same resource usage, but not exactly the same. Therefore, when constructing an offer, the JC must add margins to the estimated resource requirements. The upper limit of resource requirements should be set considering the maximum amount of resources that the JC is willing to pay for to account for the expected variance. If the JC did not provide this leeway, jobs would often exceed their resource allotments, forcing the JC to often pay for the execution of failed jobs.

3.2 MODiCuM Actors

3.2.1 Resource Provider and Job Creator. *Job Creators* (JC) have jobs to outsource and are willing to pay for computational resources. *Resource Providers* (RP) have available resources and are willing to let Job Creators use their hardware and electricity in exchange for monetary compensation. JCs post offers to the market specifying the jobs, quantities of resources required, deadlines for execution, required Docker base layers, computation architecture, and unit bid prices for resources; while RPs post offers specifying available resources, Docker base layers, computation architecture, and unit ask prices for resources. To prevent the JC and RP from cheating, they are required to include security deposits (see Eq. (2)) in the offers submitted to the platform. We provide more details on the costs and deposits later in Section 5. JCs and RPs also specify trusted Directories and Mediators, which we describe below.

A JC has multiple strategies for verifying the correctness of the results returned by an RP. Any reasonable verification strategy must cost less than what it would cost to simply execute the job. A straightforward verification strategy is to re-execute a random subset of jobs and compare the results with those provided by the RP. As long as the number of verified jobs is low enough, this strategy is viable. If the JC cannot execute the job itself, then it can post duplicate job offers instead and compare the results provided by different RPs. For some jobs, there exist verification algorithms that cost significantly less than execution, and so the JC may be able to verify every job. However, this requires the JC to implement an efficient algorithm for verification, which might be challenging.

3.2.2 Directory. *Directories* are network storage services that are available to both JCs and RPs for transferring jobs and job results. Directories are partially trusted by the actors, which means that actors (RPs and JCs) choose to trust certain Directories, but these are not necessarily trusted by the platform or by other RPs or JCs. Directories are paid by the JC and RP for making its services available for the duration of a job.

3.2.3 Solver. Matching a JC offer, and an RP offer requires computations that cannot be executed on a smart contract which has limited computation capabilities. Therefore, we extend the concept of hybrid solvers introduced in [21] and include *Solvers* in MODɪCᴜM. A Solver can be a standalone service, or it may be implemented by another actor (e.g., RPs and JCs may act as Solvers for their own offers). Unlike the smart contract, Solvers are not running on a trustworthy platform; hence, the contract has to check the feasibility of matches that the Solvers provide which is significantly easier computationally than finding matches. Solvers receive a fixed payment, set by the platform and paid by the JC and RP, for finding a match that is accepted by the platform.

3.2.4 Mediator. When there is a disagreement between a JC and an RP on the correctness of a job description or a job result (e.g., the RP claims that a job result is correct, while the JC claims that it is incorrect), a partially trusted *Mediator* decides who is at fault or "cheating." A Mediator is capable of executing the job in the same way as the RP, but it can be more expensive since it is expected to provide a more reliable service and to maintain its reputation in the ecosystem. Each Mediator sets a price which it is paid by the JC and RP for making its services available for the duration of a job. In case of mediation it is additionally compensated for the computations it executes.

3.2.5 Smart Contract. The *Smart Contract* (SC) is the cornerstone of our framework. Most communication in MODɪCᴜM is effectuated through function calls to the SC and through events emitted [3] by the SC. The SC is deployed and executed on a trustworthy decentralized platform, like the Ethereum blockchain [30], which enables it to enforce the rules of the MODɪCᴜM protocol described in the next section. It also enables actors to make financial deposits and to withdraw funds on conditions set by the SC. The functions provided by the smart contract can be found in our full paper [18].

[3] *Emitted* events in platforms such as Ethereum are recorded to the transaction logs of the ledger, which can be accessed by interested agents via polling. We use the word *emit* because that is word used for this functionality in Solidity, which we use for our proof-of-concept implementation.

4 MODICUM PROTOCOL

In this section, we discuss the operation protocol, possible misbehavior, the concept of mediation, and various faults that can occur and how they can be handled. Fig. 2a shows a possible activity sequence from registration to completion of a job. Fig. 2b represents the state of a job from the perspective of the smart contract.

4.1 Registration and Posting Offers

First, RPs and JCs register themselves with MODɪCᴜM (Fig. 2a: `registerResourceProvider`, `registerJobCreator`). Note that RPs and JCs need to register only once, and then they can make any number of offers. If an RP is interested in accepting jobs, it will send a resource offer to the platform (Figs. 2a and 2b: `postResOffer`). On the other side, a JC first creates a job and uploads the job to a Directory that it trusts and can use (Fig. 2a: `uploadJob`). Then, it posts a job offer (Figs. 2a and 2b: `postJobOffer`).

Note that any time before the offers are matched, RPs and JCs can cancel their offers. Cancellation can be due to unscheduled maintenance, or because their offers have not been matched for a long time and they wish to adjust their offer (e.g., increase maximum price) by cancelling the previous offer (Figs. 2a and 2b: `cancelJobOffer` and `cancelResOffer`) and posting a new one. Once matched, cancellation is no longer permitted.

4.2 Matching Offers

After receiving offers, the smart contract notifies the Solvers by emitting events (Figs. 2a and 2b: `JobOfferPosted`, `ResourceOffer-Posted`). The Solvers add these to their list of unmatched offers and attempt to find a match among them. After finding a match (which consists of the resource offer, job offer, and the Mediator) a Solver posts it to the smart contract (Figs. 2a and 2b: `postMatch`). The smart contract checks that the submitted match is feasible, if it is the match is recoreded and the JC and RP are notifed that their offers have been matched (Figs. 2a and 2b: `Matched`). Note that for a match to be feasible the resources specified in the RP offer must satisfy the requirements of the job specified by the JC offer. Additionally, they should have a common architecture, trusted mediator, and directory. The full feasibility specification can be found in our full paper [18]. With the `Matched` event, the contract pays the Solver the amount RP and JC specified as the matching incentive.

4.3 Execution by RP and Result Verification by the JC

After receiving notification of a match, the RP downloads the job from the Directory (Fig. 2a: `getJobImage`) and runs the job. While running the job, RP measures resource usage. Finally, when the job is done, it uploads results to the Directory (Fig. 2a: `uploadResult`) and reports the status of the job and resource usage measurements to the smart contract (Figs. 2a and 2b: `postResult`). The status of the job is the state which the job execution finished. Some possible termination states include `Completed` or `MemoryExceeded`. The full list of status codes can be found in our full paper [18] and we will discuss some of them in Section 4.4.

After receiving the notification that the result has been posted (Figs. 2a and 2b: `ResultPosted`) the JC downloads the result (Fig. 2a: `getResult`) and decides whether to verify it or not. It then accepts,

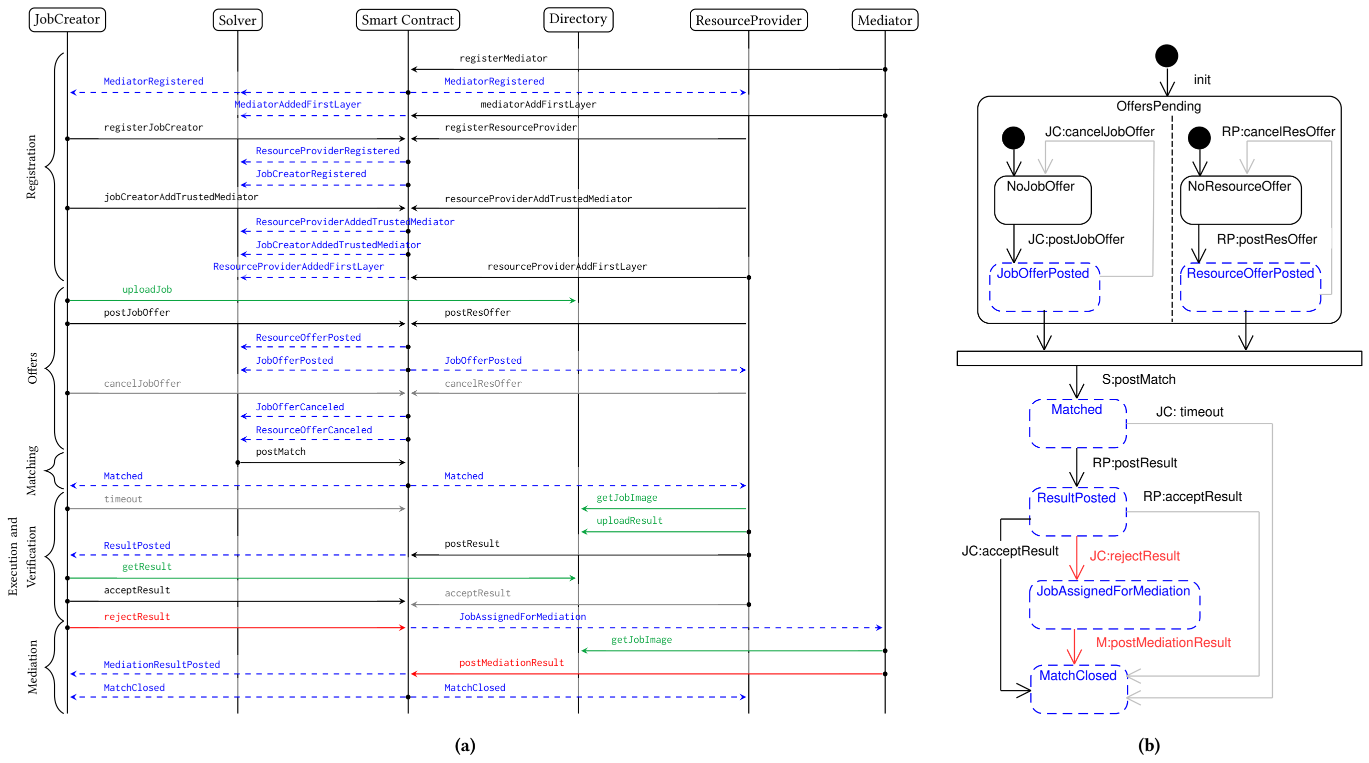


Figure 2: (a) Sequence diagram showing the outsourcing of a single job. Black arrows are function calls to the smart contract. Blue dashed lines are events emitted by the smart contract. Gray lines are optional function calls. Red lines are optional calls that are required in case of disagreement between RP and JC. Green lines are off-chain communication. Note that events are broadcast, and visible to all agents that can interact with the contract. (b) States of job in MODiCuM. Function calls to the smart contract are prefixed by the actors that make the calls.

ignores, or rejects the result. If the JC accepts the result (Figs. 2a and 2b: `acceptResult`) the contract returns deposits, pays the RP and Mediator, and closes the match (Fig. 2b: `MatchClosed`).

If the JC ignores the result, then after some time, the window for the JC to react closes and the RP is permitted to accept the result (Fig. 2b: `RP:acceptResult`) resulting in the match closing (Fig. 2b: `MatchClosed`). If the JC disagrees with the result, it rejects the result (Figs. 2a and 2b: `rejectResult`) with a reason code such as `WrongResults` or `ResultNotFound`; then, mediation follows (Figs. 2a and 2b: `JobAssignedForMediation`).

4.4 Faults and Mediation

There are essentially two types of faults that can occur in the system: (1) Connectivity: this can occur for the JC, RP, and the Mediator when they try to communicate with the Directory or the smart contract. Note that the JC, RP, and Mediator do not talk directly (see Fig. 1). Solver connectivity faults are not a concern since it only interacts with the smart contract and the failure of a Solver implies that Solver does not submit a solution, but others may. Hence, we only worry about the connectivity to smart contract and the directory. (2) Data (job input and results of execution): it can be malformed, return an exception when executed, not be available on the Directory, be verifiably incorrect, etc.

First, we discuss the connectivity with the smart contract. If the RP cannot communicate with the smart contract, it may not receive a notification that it has been matched and it is also unable to call `postResult`. These are both addressed by the JC having a timeout. If the JC calls `timeout` (see Figs. 2a and 2b) and the required waiting period has elapsed, then the smart contract pays the JC the estimated value of the job from the RPs deposit and returns the remainder. The timeout also addresses when the JC misses the `ResultPosted` message sent by the RP, since if the misses the message, it will attempt to timeout, which will fail because the result was posted, and then it will fetch the result. Smart contract connectivity failure also means that the JC cannot send `acceptResult` and that the RP never receives a notification that the JC accepted. This is addressed by allowing the RP to bypass the JC and call `acceptResult` if the platform specified duration for the JC to respond has elapsed. Connectivity failure also means that the JC and the RP may not receive the `MediationResultPosted` message. In this case, they may respond by removing that Mediator from their trusted list; and if the result eventually arrives, they can re-add the Mediator. This also addresses when the Mediator does not get the mediation request or cannot post the mediation result. In this case, mediation is considered to have failed; to release the security deposits, we enforce a timeout via the smart contract. In the event of this timeout, we instead pay the RP half of the JC's job estimate and return the remainder of the deposits. Obviously, the Mediator does not get paid since mediation failed.

Now, we discuss connectivity with the Directory. If a JC cannot connect to the Directory to submit a job, it simply tries to upload to another Directory it trusts. If it cannot connect to retrieve a result then the JC must either pay the RP or request mediation with the `DirectoryUnavailable` status. To verify the JC claim, the Mediator queries the Directory for its uptime. If the Directory reports that it was available for the entire job duration, then the Mediator assigns fault to the JC; otherwise, it assigns fault to the Directory. If the JC does not agree, it may remove the Directory and/or the Mediator from its trusted lists. Similarly, if the RP cannot connect to the Directory, either to fetch the job or upload the results, then it posts a result with the `DirectoryUnavailable` status. If the JC does not agree that the Directory is unavailable, then it requests mediation which proceeds as above with the RP rather than the JC. If the Mediator sends a mediation result of `DirectoryUnavailable` then the RP and JC may choose to remove the Directory, Mediator, or both from their trusted list.

Finally, we discuss the data faults. The data faults that the RP can detect and the corresponding result status are: 1) no job on the Directory (`JobNotFound`), 2) a job description error (`JobDescription-Error`), 3) excessive resource consumption (`ResourceExceeded`), and 4) an execution exception during execution (`ExceptionOccur-red`). The JC can request mediation if it disagrees with a claim of any of these faults, in addition to detecting no result on the Directory, or that the result is incorrect. Finally, the JC can request mediation if, after verification, the JC suspects that the resource usage claimed by the RP is too high. This could occur since the resource variation should be small, but the JC may have set a high resource limit to ensure the job completes.

If the JC requests mediation claiming there is a data fault, then the Mediator attempts to replicate the steps taken by the RP, with the distinction being that it re-executes the job n times (see Section 5.1, it is defined as a parameter of the smart contract), and compares its results with the RP's results. In two cases, the JC will be at fault: (1) All of the Mediator's results and RP's result are the same, which means that the RP has executed the job correctly. (2) The Mediator gets two different results when running the job, which means that JC has submitted a non-deterministic job. Otherwise, the Mediator assumes that the RP has submitted a wrong result. Another case is when the JC claims that the result is not on the Directory. In that instance, the Mediator attempts to retrieve the result from the Directory. If it cannot it faults the RP, if it can it faults the JC. If either agent disagrees it may remove that Directory, Mediator, or both from its trusted list.

The Mediator submits the verdict to the smart contract (Figs. 2a and 2b: `postMediationResult`), and the smart contract claims the security deposit. Of the deposit, *the actual job price* is used to compensate the damaged party for its losses, and π_m (which is the job price times the number of repeated executions n) goes to the Mediator to cover its mediation costs. In addition, Mediators always receive π_a as payment for making their service available. They receive this when a job is closed (Figs. 2a and 2b: `MatchClosed`).

For this mediation approach to work, the RP must not allow jobs to access any extra information (e.g., physical location, time) beyond what is in its description and Docker image. Otherwise, a JC could create a job that could determine where it is running (e.g., via connecting to a remote server) and produce different results on the RP and on the Mediator. Thus, the Mediator would always incorrectly blame and punish the RP. This is why the platform requires that jobs be (1) deterministic and punishes the JC if non-determinism is detected (otherwise, we could not use repeated executions to verify) and (2) batch (otherwise, the jobs could not be isolated). The deterministic restriction specifically requires that the Mediator gets the same result as the RP. This means that in practice non-deterministic jobs could be sent by the JC as long as the RP records the non-deterministic values instantiated on the RP (which are not part of the input posted on the Directory) as part of its result for use in verification. Some examples of jobs that could be computed in such an environment are machine learning tasks, or jobs that are similar to volunteer computing tasks, such as protein folding. Essentially, any batch data processing task is feasible.

4.4.1 *Collusion.*

A part of the challenge in designing a fair system is the problem of collusion. We enumerate all possible two-party collusions and discuss their objectives and how MODɪCᴜM addresses them. We do not consider more than two party collusions explicitly because they are indistinguishable from two-party collusions for the non-colluding agents in MODɪCᴜM.

- *Job Creator and Solver*: Since offers are public and any participant can act as a solver, the collusion between JCs and solvers is inevitable. The goal of this collusion is to match a JC's offers to the resource offers with the lowest unit price. Thus, the RP with the lowest-price offer will be matched first. This is harmless because every resource offer includes a minimum reservation price; thus, a JC cannot force an RP to perform computation for less than what the RP voluntarily accepts.
- *Job Creator and Mediator*: Both JC and Mediator can benefit from this collusion by taking the RP's security deposit and splitting it between them, while the JC can also benefit by having its jobs executed without paying. This can be achieved by the JC requesting mediation on a correct result and the Mediator ruling in favor of the JC. To an honest RP, this collusion will appear as a faulty Directory, faulty Mediator, or non-deterministic job, though it can eliminate the last by repeating the job execution. The RP removes the Mediator and Directory from its trusted list. Thus, a mediator can launch this attack only once per RP.
- *Job Creator and Directory*: This collusion is similar to the one between the JC and Mediator. JC and Directory can collude in multiple ways. For example, the Directory manipulates the job so that the RP will return `JobDescriptionError` result status. Then, JC will request mediation, the Directory will provide the correct job to the Mediator, and the Mediator will rule in favor of JC. Since the RP cannot distinguish this collusion from the one between a JC and mediator, as a response, the RP removes the Mediator and Directory from its trusted list. Thus, a Directory can launch this attack only once per RP.
- *Job Creator and Resource Provider*: There is no possible benefit from this collusion.
- *Resource Provider and Solver*: The goal of this collusion is the same as for the collusion between JC and Solver, and its impact

and mitigation are also the same. This collusion is desirable for the same reason as well.

- *Resource Provider and Mediator:* RP and Mediator can both benefit from this collusion by taking the JC's security deposit and splitting it between them; while the RP can also benefit by receiving payment for a job without actually executing it. This can be achieved by the RP returning any job result, which the JC might verify and request to be mediated, upon which Mediator will rule in favor of the RP. This collusion is mitigated in the same way as the collusion between JC and Mediator, except that the roles of JC and RP are reversed.
- *Resource Provider and Directory*: The goal of this collusion is the same as the RP and Mediator collusion, and it can be achieved and handled in a similar manner as the JC and Directory collusion. To an honest JC, collusion will appear as a faulty or colluding Directory. As a response, the JC removes the Mediator and Directory from its trusted list. Thus, a Directory can launch this attack only once per JC.
- *Directory and Solver*: There is no benefit from this collusion.
- *Directory and Mediator*: This collusion aims to ensure that the JC will request Mediation by manipulating data or availability, and then splitting the payment for Mediation. Depending on the Directory's manipulation and the Mediator's ruling, either the JC or RP will respond in the same way as if the RP or JC were colluding with the Directory or Mediator.
- *Mediator and Solver*: The goal in this case is for the solver to prioritize trades that include the Mediator, and further prioritize JCs and RPs with a history of requiring mediation. This could result in unfairness, i.e., some jobs may never get matched. However, this is not a real concern since the JC and RP can act as solvers and match their own offers.

From exploring the possible scenarios, we conclude that the JC and RP could be cheated once by a Mediator or a Directory; however, the faulty agent will be removed from the trusted list afterwards. Since the business model for Mediators and Directories is attracting RPs and JCs who trust and pay them for their services, they have strong incentives to build a positive reputation in the ecosystem. Thus, we assume they will behave honestly. The formal proof for this conjecture will be provided in future work.

5 ANALYZING PARTICIPANT BEHAVIOR AND UTILITIES

Here we formulate the agents' actions as a game and solve for strategies that result in a Nash equilibrium. We also show that platform parameters can be set so that a rational JC will follow the protocol with, at least, some minimum probability. The extensive-form representation of the game (shown in Fig. 3) is explained below.

5.1 Game-Theoretic Model

To understand the game, we first introduce a set of parameters in Table 1. Parameters denoted by p are probabilities, π are payouts, c are costs incurred by agents, and g are the costs for interacting with the platform. Many parameters have constraints on the valid values that they can take and on their relationships with other parameters.

Now, we consider the JC's choices. The JC has a job to outsource, whose execution provides a constant benefit b upon receiving the correct job result. The JC is willing to pay a job specific price, up to $\hat{\pi}_c$, which is appropriate for the resources required to have the result computed. The JC is able to verify if the job result is correct, but it incurs verification cost c_v by doing so. In order to mitigate this cost, the JC may choose to verify the result with some probability p_v, trading confidence for lower costs.

However, a dishonest JC can design non-deterministic jobs and we assume that the JC can always recover the correct result from

Table 1: MODıCuM Parameters and System Constraints

MODıCuM Smart Contract (SC)	
θ	penalty rate set by the contract
d	deposit by JC and RP for collateral prior to transaction
d_{min}	minimum security deposit
g_m	cost of requesting mediation
g_r	cost for an RP to participate; includes the costs of submitting the offer and the results, as well as partial payment to the Solver for a match accepted by the smart contract
g_j	cost for a JC to participate; includes the costs of submitting the job offer, as well as partial payment to the Solver for a match accepted by the smart contract
π_d	payout to wronged party when *deception* is detected
n	number of times Mediator executes a disputed job
Mediator (M) and Directory (D)	
π_m	payout to the *Mediator* when it is invoked
π_a	payout to the Mediator and Directory for being *available* for the duration of the job, which also covers the Solver match payment
Job Creator (JC)	
p_v	probability that JC verifies a job result (verification rate)
p_a	probability that a correct execution of a non-deterministic job returns a "normal" result for which the JC will not request mediation (functionally, this is an indicator of how honest the JC is)
π_c	payment from JC to RP for successfully *completing* a computation
b	JC's benefit for finished job minus cost of submitting job
c_v	JC's cost to verify a job result
Resource Provider (RP)	
π_r	payout that *resource* provider asks for completing the job
p_e	probability that RP intentionally *executes* the job correctly ($1 - p_e$ is the probability of cheating)
c_e	*execution* cost for RP to compute job
c_d	computational cost for RP to *deceive* and create wrong answer
System Constraints	
1	$b > \pi_c + \pi_a + g_j$ for honest JC
2	$\theta \geq 0$ the penalty rate cannot be negative
3	$n > 0$ else the mediator does not re-execute the job
4	$c_e > c_d > 0$ else the RP never has incentive to cheat
5	$\hat{\pi}_c \geq \pi_c \geq \pi_r > c_e$; if $c_e \geq \pi_r$, the RP will abort the job; and if $\pi_r > \hat{\pi}_c$, then that job and resource offer match is disallowed by the contract
6	$d \geq d_{min}$

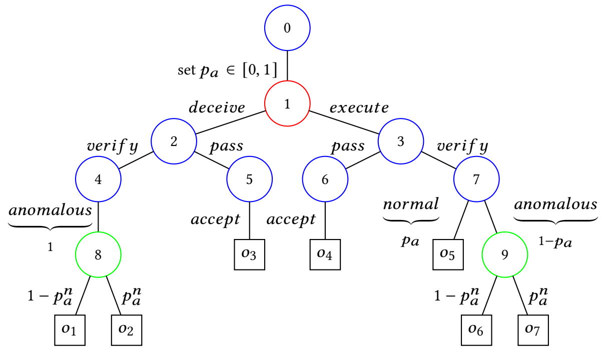


Figure 3: Extensive-form game produced by the MODICUM protocol. Blue nodes indicate JC moves, red nodes indicate RP moves, green nodes indicate Mediator's probabilistic outcome. The game is sequential, but the decisions are hidden, so we treat it as a simultaneous move game. Each outcome has payouts for the agents, which are found in Fig. 4

Outcome	Party	Contract Payoff	Self Benefit	Reward(r)
o_1	RP	$\pi_d - g_r - \pi_a$	$-c_d$	$\pi_d - c_d - g_r - \pi_a$
	JC	$-g_j - d - g_m - \pi_a$	$-c_v$	$-g_j - d - g_m - c_v - \pi_a$
o_2	RP	$-d - g_r - \pi_a$	$-c_d$	$-d - g_r - c_d - \pi_a$
	JC	$\pi_d - g_j - g_m - \pi_a$	$-c_v$	$\pi_d - g_j - g_m - c_v - \pi_a$
o_3	RP	$\pi_c - g_r - \pi_a$	$-c_d$	$\pi_c - g_r - c_d - \pi_a$
	JC	$-g_j - \pi_c - \pi_a$	0	$-g_j - \pi_c - \pi_a$
o_4	RP	$\pi_c - g_r - \pi_a$	$-c_e$	$\pi_c - g_r - c_e - \pi_a$
	JC	$-g_j - \pi_c - \pi_a$	b	$b - g_j - \pi_c - \pi_a$
o_5	RP	$\pi_c - g_r - \pi_a$	$-c_e$	$\pi_c - g_r - c_e - \pi_a$
	JC	$-g_j - \pi_c - \pi_a$	$b - c_v$	$b - g_j - \pi_c - c_v - \pi_a$
o_6	RP	$\pi_d - g_r - \pi_a$	$-c_e$	$\pi_d - g_r - c_e - \pi_a$
	JC	$-g_j - d - g_m - \pi_a$	$b - c_v$	$b - g_j - d - g_m - c_v - \pi_a$
o_7	RP	$-d - g_r - \pi_a$	$-c_e$	$-d - g_r - c_e - \pi_a$
	JC	$\pi_d - g_j - g_m - \pi_a$	$b - c_v$	$b + \pi_d - g_j - g_m - c_v - \pi_a$

Figure 4: Game outcomes and payments in Fig. 3. For example, o_1 is the outcome when the RP deceives, the JC verifies, and the Mediator finds non-determinism and faults the JC. The reward for the agents is the sum of contract payoff and self-benefit and is denoted r_{o_i}. The Mediator is not included in the table: in every outcome, it receives π_a; when the JC requests mediation, the M also receives π_m, which is drawn from the faulty party's deposit d.

any output[4]. The goal of the JC in designing a non-deterministic job is to get the correct output without having to pay the RP. It can accomplish this if it requests mediation, and when it does, the mediator concludes that the result returned by the RP is incorrect. Thus, if a JC designs a job to look "normal" to the mediator with probability p_a and "incorrect" with probability $1 - p_a$, then the JC will accept correct results with probability p_a and request mediation with probability $1 - p_a$. A simple illustration of such a job is one which returns a natural number as its solution, and changes the sign of that value (i.e., multiply by -1) with a fixed probability, creating a set of positive results and a set of negative results.

[4]Note that the RP and M cannot be expected to analyze the code, and hence cannot know that the job contains non-determinism.

The game starts with the JC choosing a probability value for p_a. A probability of $p_a = 0$ means that the JC is completely dishonest, and all results will be considered incorrect. A probability of $p_a = 1$, means the JC is honest and all correct outputs are accepted.

The RP makes the next move, choosing between honestly executing the job or forging a result to deceive the JC. The RP may be motivated to return a false result if the job execution cost c_e is higher than the deception cost c_d and the JC does not verify the result with some probability. The RP executes the job with probability p_e, where $p_e = 0$ means that the RP is completely dishonest and always attempts to deceive the JC, while $p_e = 1$ means that the RP is honest and executes the job correctly. Note that the correct result can be a fault code if the computation fails.

The JC makes the next move and selects its strategy, choosing between verifying the result or passing on the verification. The JC verifies the with probability p_v. If verification finds the result to be incorrect, the JC requests mediation to dispute the result.

To resolve the dispute, the Mediator must determine which agent is at fault. The Mediator does this by performing the steps that an RP would take to execute a job, repeating several times to detect non-determinism. When initialized the smart contract specifies a *verification count n*, which is the number of times the Mediator will execute a job checking for anomalies. Since the job has probability p_a of returning a normal result and the Mediator executes the job n times, the probability that the job returns a normal result in every execution is p_a^n. Thus, the Mediator detects a non-deterministic job with probability $1 - p_a^n$, and fines the JC for being dishonest.

As stated in Section 4.1, to deter cheating through fines, we require JCs and RPs to provide a *security deposit d* when submitting offers. We define the deposit to be dependent on the JC's estimate of the job price $\hat{\pi}_c$ and scaled by a penalty rate θ, which is set by the smart contract. The job price $\hat{\pi}_c$ estimated by the JC is the same as π_c except it uses the JC's bid prices and requested resources. The penalty rate must be set to a sufficiently high value to deter misbehavior. The security deposit must also cover the cost of potential mediation π_m, which we estimate as $\hat{\pi}_c \cdot n$ since the JC is willing to pay $\hat{\pi}_c$ and the Mediator must run n times. The deposit must also cover the availability costs of the Mediator and Directory as well as the Solver costs; we let π_a denote the sum of these costs. Thus, we define the minimum security deposit d_{min} as:

$$d_{min} = \underbrace{\hat{\pi}_c \cdot \theta}_{\text{penalty}} + \underbrace{(\hat{\pi}_c \cdot n)}_{\pi_m} + \pi_a \tag{2}$$

The game induced by the interactions of the actors described above has 7 possible outcomes. Each outcome has payouts for the agents as described in Fig. 4. To illustrate how the payouts are calculated, consider the following sequence. The JC pays g_j to submit a non-deterministic job with probability p_a of returning a normal result. The RP honestly executes the job incurring cost c_e and pays g_r to submit the result. Since the RP executed honestly, the JC receives the benefit b. The JC verifies the result, incurring cost c_v, and detects that the non-beneficial part of the result is anomalous. It then attempts to avoid paying π_c to the RP by requesting mediation, paying g_m. The Mediator executes the job n times and if in one or more of those executions it encounters an anomalous result, which occurs with probability $1 - p_a^n$, then it submits to the smart contract

Table 2: RP payoffs by decision

	verify	pass
execute	$\overbrace{-c_e - g_r - \pi_a + \pi_c\Big(np_a^n\,(p_a - 1) + p_a + p_a^n\theta\,(p_a - 1)\Big) + \pi_d\,(1 - p_a)\,(1 - p_a^n)}^{U_{EV}^{RP}}$	$\overbrace{\pi_c - c_e - g_r - \pi_a}^{U_{EP}^{RP}}$
deceive	$\underbrace{-c_d - g_r - \pi_a + p_a^n\pi_c\,(-n - \theta) + \pi_d\,(1 - p_a^n)}_{U_{DV}^{RP}}$	$\underbrace{\pi_c - c_d - g_r - \pi_a}_{U_{DP}^{RP}}$

Table 3: JC payoffs by decision

	verify	pass
execute	$\overbrace{b - g_j - \pi_c\,(n + \theta)\,(1 - p_a)\,(1 - p_a^n) + (1 - p_a)\,(-g_m + p_a^n\pi_d) - c_v - p_a\pi_c - \pi_a}^{U_{EV}^{JC}}$	$\overbrace{b - g_j - \pi_a - \pi_c}^{U_{EP}^{JC}}$
deceive	$\underbrace{-c_v - g_j - g_m + p_a^n\pi_d - \pi_a - \pi_c\,(n - p_a^n\,(n + \theta) + \theta)}_{U_{DV}^{JC}}$	$\underbrace{-g_j - \pi_a - \pi_c}_{U_{DP}^{JC}}$

that the JC is at fault and the JC loses its security deposit d for submitting non-deterministic jobs resulting in outcome o_6. Otherwise if all of the results from the repeated executions are normal then the JC successfully cheats and receives π_d as reparations for being "faulted" resulting in o_7. The payouts of the other outcomes are calculated similarly. The platform interaction costs are fixed properties of a given smart contract and its underlying platform.

Since we assume that Directories and Mediators always act honestly, we do not consider their strategic decisions in our game analysis. The expected utilities of both the RP and the JC are summarized in Tables 2 and 3, respectively. The table is constructed by considering the possible actions of the RP and JC. There are two possible actions for RP (execute and deceive) and two for JC (verify and pass) as illustrated by the tree in Fig. 3. Hence, the utilities of RP and JC depend on the four action combinations and their possible outcomes $o_1 \cdots o_7$. To understand how the utilities are calculated, consider the example of the case when RP chooses to execute, and JC chooses to verify. This is node 7 in Fig. 3, and there are three possible outcomes o_5, o_6, o_7. Outcome o_5 occurs with probability p_a, o_6 with probability $(1 - p_a)(1 - p_a^n)$, and o_7 with probability $(1 - p_a)p_a^n$. Thus,

$$U_{EV}^{JC} = p_a \cdot t_{o_5} + (1 - p_a)\big((1 - p_a^n) \cdot t_{o_6} + p_a^n \cdot t_{o_7}\big) \tag{3}$$

The utility for each combination of actions is denoted by utility U with superscript of the agent (i.e., RP and JC) and subscript of the action combination of RP and JC (i.e., EV is <execute,verify>, DP is <deceive,pass>, EP is <execute,pass>, and DV is <deceive,verify>). Note that we replace the total outcomes payoffs using Fig. 4 in Tables 2 and 3.

5.2 Equilibrium Analysis

Here we analyze the utility functions explained in previous section. For lack of space, we describe only the key results. Detailed proofs for these statements can be found in the appendices of the arXiv version of this paper [18].

The ideal operating conditions for the platform would be if the RP always *executed* ($p_e = 1$), the JC never had to *verify* ($p_v = 0$) and only submitted jobs which returned deterministic results ($p_a = 1$). However, if the agents are rational, these parameters do not constitute a Nash equilibrium. This is because if the JC does not verify, then the RP will choose to deceive rather than execute since $U_{EP}^{RP} < U_{DP}^{RP}$. The JCs utility in this case is always negative and so it is better off not participating in the platform. This proves the following theorem:

Theorem 1 (JC should not always pass). *If the JC always* passes *(i.e., $p_v = 0$), then the RP's best response is to always* deceive *(i.e., $p_e = 0$).*

If the RP always chose to deceive, the platform would serve no purpose. Therefore, we must ensure that if the JC chooses to verify, the RP prefers to execute. This occurs when $U_{DV}^{RP} < U_{EV}^{RP}$. We show in [18] that this is true if $p_a^{n+1} > \frac{1}{2}$. When these conditions are true, we can prove the following theorem:

Theorem 2 ($p_e > 0$). *If $p_v > 0$ and $p_a^{n+1} > \frac{1}{2}$, then a rational RP must execute the jobs with non-zero probability.*

Recall p_a is a parameter set by the JC, so to satisfy the condition on p_a in Theorem 2 we must show that the platform can set parameters to force the JC to choose a value for p_a that is greater than some lower bound. We assume that the JC is rational and chooses p_a to optimize its utility U^{JC}. To find the bound we take its derivative with respect to p_a, $\frac{\partial U^{JC}}{\partial p_a}$, and assess how each parameter shifts the optimal value for p_a. The trends are as n, θ, g_m increase p_a also increases, meanwhile as π_c and p_e increase p_a decreases. Knowing how varying each parameter shifts the optimal value for p_a we can select the worst-case values for each parameter, i.e. those that minimize optimal p_a, maximizing dishonesty. Specifically, if the parameter and p_a are inversely related, set the parameter to its maximum allowed value, and if they are directly related set the parameter to its minimum value. Thus, the worst-case values for each of the parameters are: $g_m = 0$, $p_e = 1$, $n = 1$, $\theta = 0$. The plot in Fig. 5 uses those parameters and shows that increasing n does cause the optimal value for p_a to increase. We see that when $n = 1$, the optimal $p_a = 0.5$; and when $n = 4$, the optimal $p_a = 0.943$. This can be summarized as:

Theorem 3 (Bounded p_a). *Setting the JC utility function parameters to minimize the optimal value for p_a (maximizing a rational JC's dishonesty) ensures that any deviation will increase p_a. The platform controls n and θ, and so controls the minimum optimal value of p_a.*

We want to minimize the number of times the Mediator has to replicate the computation, so we set $n = 2$ and set $\theta = 50$ which yields a minimum $p_a = 0.99$.

So far we have shown that we can ensure that a rational RP will prefer to execute when a JC verifies, and deceive when the JC passes, and we can limit the amount of cheating the JC can achieve through non-deterministic jobs. Next, we analyze the JC utilities to determine the Nash equilibria of the system.

Analyzing JC types: The preferences of the JC depend on the parameters in its utility function. We refer to each combination of

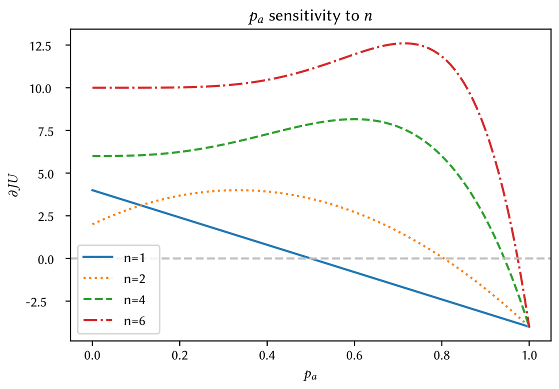


Figure 5: We vary the value of n and plot p_a against $\frac{\partial U^{JC}}{\partial p_a}$. This shows that as n increases, so does the optimal value of p_a (zero-crossing of derivative curve). Parameter values are $\pi_c = 2$, $g_m = 0$, $\theta = 0$, $c_v = 1$, $b = 4$, $p_e = 1$

preferences as a "type" of JC. We will call a JC that prefers to always verify as type 1. A JC that prefers to always pass is type 4; we have already covered this type and determined that a JC of this type will not participate. A JC that prefers to verify when the RP executes and pass when the RP deceives is type 2. A JC that prefers to pass when the RP executes and verify when the RP deceives is type 3. The JC has a preference because its utility is better in that case. These preferences are summarized in Table 4 with the $*$ symbol followed by the type that prefers that choice. This table is Table 3 refactored to remove terms that do not impact the JC's preference and to highlight the relationship between the preference and the cost of verification c_v. We consider the equilibrium for each type assuming the RP has been restricted as we discussed previously. The theorems below summarize these observations. Proofs are available in our full paper [18].

Theorem 4 (JC type 1). *If the JC is type 1, it will always verify ($p_v = 1$) since c_v is sufficiently low. This results in a pure strategy equilibrium <execute,verify>.*

Theorem 5 (JC type 2). *If the JC is type 2, it results in two pure strategy equilibria <execute,verify>, <deceive,pass>, and one mixed strategy Nash equilibrium where the JC randomly mixes between verifying and passing.*

Theorem 6 (JC type 3). *If the JC is type 3, it will result in a Nash equilibrium where the JC randomly mixes between verifying and passing, and the RP mixes between executing and deceiving.*

Strategies: Based on these preferences, a type 1 JC will always verify $p_v = 1$. Type 2 JCs may also choose to always verify, or choose a mixed strategy, setting p_v such that the RP receives the same utility regardless of whether it executes or deceives resulting in a Nash equilibrium. It achieves this by solving Eq. (4) for p_v, setting $c_e = \pi_c$ and $c_d = 0$. Type 3 JCs only have the option of solving Eq. (4) for p_v.

$$p_v \cdot U_{EV}^{RP} + (1 - p_v) \cdot U_{EP}^{RP} = p_v \cdot U_{DV}^{RP} + (1 - p_v) \cdot U_{DP}^{RP}$$
$$\text{Solve for } p_v; \quad p_v = \frac{c_e - c_d}{p_a^{n+1} \pi_c (\theta + n + 1)} \tag{4}$$

Table 4: Simplified JC payoffs to assess dominant strategy with $\pi_d = \pi_c$

	verify		pass
execute	$-\pi_c (1 - p_a) (1 - p_a^n) (n + \theta + 1)$	$+$	$c_v^{*3,4}$
	$(1 - p_a) (-g_m + 2\pi_c)^{*1,2}$		
deceive	$2\pi_c - g_m - \pi_c (1 - p_a^n) (n + \theta + 1)^{*1,3}$		$c_v^{*2,4}$

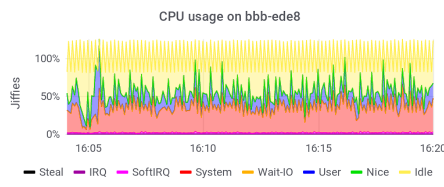
The RP's strategy changes depending on which type of JC it is working with. If the JC is type 1, it is simple: the RP must execute. However, the other two types can mix, so the RP must also mix. It does this by solving Eq. (5) for p_e. The challenge with this is that the RP does not know the value of c_v. However all other parameters are known once a match is made except p_a which from our work earlier we know that $p_a \geq .99$. Thus, the RP can sample c_v from a uniform distribution where $0 \leq c_v \leq P_{EV}^{JC}$ for type 2 and $0 \leq c_v \leq P_{DV}^{JC}$ for type 3 (P_{EV}^{JC} is the JCs preference value for <execute,verify> from Table 4). However, since the RP does not know which type of JC it is working with, it further mixes between the 3 strategies according to its belief on the distribution of the types of JC in the system.

$$p_e \cdot U_{EV}^{JC} + (1 - p_e) \cdot U_{DV}^{JC} = p_e \cdot U_{EP}^{JC} + (1 - p_e) \cdot U_{DP}^{JC}$$
$$\text{Solve for } p_e; \quad p_e = \frac{2\pi_c - c_v - g_m - \pi_c (1 - p_a^n) (n + \theta + 1)}{p_a (2\pi_c - g_m - \pi_c (1 - p_a^n) (n + \theta + 1))} \tag{5}$$

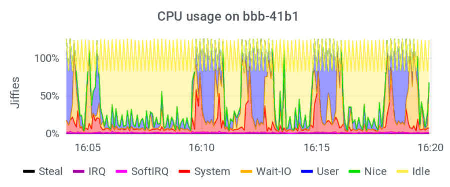
This analysis shows that we can limit the dishonesty of the JC, and that the JC and RP can compute strategies that will result in a mixed-strategy Nash equilibrium. Further, we can show that the JC will not have to verify frequently by examining Eq. (4) and showing that the maximum verification rate is low. First, we know that if $c_e < \pi_c$ (which should hold since otherwise the RP will always deceive), then $\frac{c_e - c_d}{\pi_c} \leq 1$. In this case, the verification rate is at its maximum when $c_e = \pi_c$ and $c_d = 0$. Simplifying Eq. (4), we find that $\frac{1}{p_a^{n+1} (\theta + n + 1)}$. In establishing Theorem 3, we showed that setting θ and n enforces a minimum value for p_a. Again to minimize computation replication we choose $n = 2$ and $\theta = 50$ and recall that this results in $p_a \geq 0.99$. Substituting these values into our simplified equation and find that $p_v = 0.02$. This means that the JC will verify 2% of the results. Since $p_a \geq 0.99$, mediation will occur at most 0.02% of the time.

6 MODICUM IMPLEMENTATION

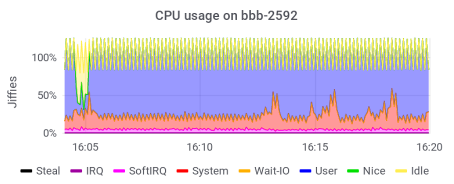
In this section, we describe an implementation of MODiCuM. The code is available on GitHub [6]. We use a private Ethereum network to provide smart contract functionality. MODiCuM actors, including Job Creators, Resource Providers, Solvers, and Directory services are implemented as Python services. The matching solver uses a greedy approach to match offers as they become available using a maximum bipartite matching algorithm. These actors use the JSON-RPC interface to connect to the Ethereum Geth client [3]. Note that each actor can be configured with any number of Geth clients, and the ledger can be implemented either as a private blockchain or we can use the main Ethereum chain as the ledger. We use Docker [22] images to package jobs. Jobs can be run securely by separating the job from the underlying infrastructure through



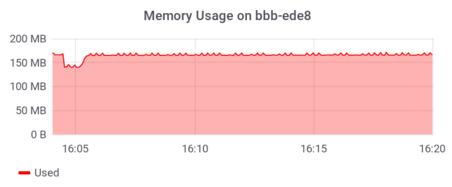
(a) Job Creator CPU



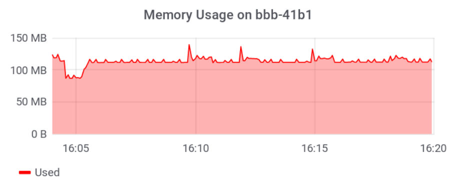
(b) Resource Provider CPU



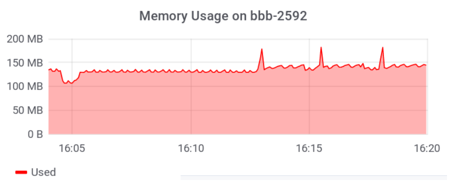
(c) Mediator CPU



(d) Job Creator RAM



(e) Resource Provider RAM



(f) Mediator RAM

Figure 6: MODICUM Total Resource usage. Each plot shows the resource consumption on a node.

isolation. This can be achieved using a hardening solution such as AppArmor [11] or seccomp [7] in conjunction with Docker. This protects computational nodes from erroneous or malicious jobs, if properly configured. Proper configuration has been discussed in other papers, for example [14], and we do not go into detail here. To determine job requirements, jobs are profiled using cAdvisor [1].

As part of the offer and matching specification, we require the JC to include the hash of the base of the Docker image. Additionally, during setup, Mediators and RPs specify a set of supported base Docker images (Fig. 2a: *mediatorAddFirstLayer* and *resourceProviderAddFirstLayer*). This permits some optimization since RPs are able to specify which base images they have installed, thus by matching them accordingly, we can reduce the bandwidth required to transfer the job by the size of the base image. Common base images vary between about 2MB - 200MB [5].

6.1 Experimental Evaluation

JCs, RPs, and a Mediator were deployed on a 32 node BeagleBone cluster with Ubuntu 18.04. We set up a private Ethereum network. The Solver and Directory were deployed on an Intel i7 laptop with 24GB RAM. The actors connect to the Geth client [3] each using a unique Ethereum account.

Measuring Gas Costs and Function Times: To measure the minimum cost of executing a job via MODICUM, we had a single JC submit 100 jobs and measured the function gas costs and call times independent of the job that was being executed. These can be found in our full paper [18]. The JC's average gas cost of a nominal execution is 592, 000 gas. At current Ethereum prices, this converts to \$0.168 per job for the JC [2]. Comparing this to Amazon Lambda pricing [9] on a machine with 512MB RAM (which a BeagleBone has), a job would have to last ~6 hours to incur the same cost. However, the electrical costs to run a BeagleBone (210-460mA @5V) at maximum load for that long, assuming \$0.12/kWh electricity price, is only \$0.0016. This illustrates that there is potential for such a transaction system to be a viable option compared to AWS. However, using Ethereum as the underlying mechanism is currently only viable for long running jobs; but work is underway to improve

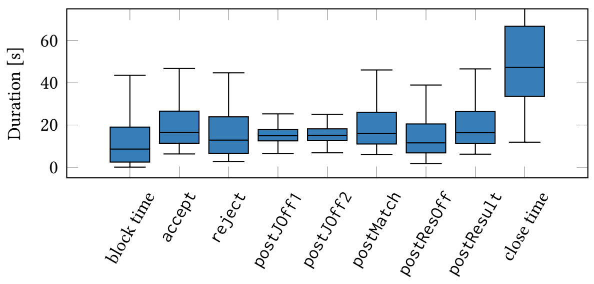


Figure 7: Duration of MODICUM function calls during nominal operation.

the efficiency of Ethereum [8]. We also measured the gas cost of a mediated execution: the JC's average gas cost in this case is 991, 000, and the Mediator's cost to post the mediation result is 187, 000.

During these tests, duration of the function calls was also measured (see Fig. 7). We note that the mean time for a block with transactions to be mined (block time in Fig. 7) is about every 10 seconds, and that function call delay is consistently about 5 to 10 seconds longer. This may be attributable to calls missing a recent block. The close time is the time measured between `MatchClosed` events. Since the jobs were run sequentially it is a measure of the cumulative time added to the execution of a job, in this test running a job through MODICUM added approximately 52 seconds.

Measuring the Overhead of Platform: To measure the overhead, we compared the execution of jobs run with Docker containers natively against jobs run in MODICUM. The job we used was the bodytrack computer vision application drawn from the PARSEC benchmarking suite [13]. PARSEC has been used to benchmark resource allocation platforms [26] as well as platforms for high performance computing [24] among others. We again ran 100 jobs, which took a total of 221 minutes, averaging 2 minutes per job. The mean time for a block to be mined was 31 seconds, meaning it took about 4 blocks to complete a job. The block time was likely longer in this experiment because as the blockchain grew block mining times appeared to increase, though we did not study this explicitly. This application tracks the 3D pose of a human body

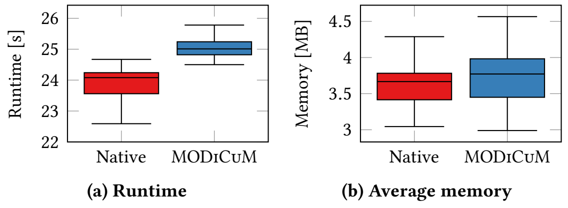


Figure 8: Running time and memory usage on MODiCuM and native execution.

through a sequence of images. In Figure 8a, we see that MODiCuM increases the runtime by about 1 second or 4%. In Figure 8b, the average memory consumption increases by 0.1MB, or 3%. To check mediation, we ran the jobs again, but rejected the results for all 100 jobs and requested mediation.

Resource consumption while running the benchmark on MODiCuM can be seen in Fig. 6. The nodes are at idle during the valley from 16:04:30 to 16:05:30 at which point the platform is started. From 16:05:30 onward, MODiCuM is running, and the bursts that can be seen, for example at 14:09:30-14:10:45 in Figure 6b, are when jobs are being executed. MODiCuM introduces about 20-25MB RAM overhead for each agent type. It introduces 80% CPU overhead on the Mediator, 30% on the Job Creator, and no apparent change to the Resource Provider. This is acceptable since the BBB devices are resource constrained and so the overhead will be less significant on more powerful compute nodes.

7 CONCLUSIONS

An open market of computational resources, where resource owners and resource users trade directly with each other has the potential for greater participation than volunteer computing and more competitive pricing than cloud computing. The key challenges associated with implementing such a market stem from the fact that any agent can participate and behave maliciously. Thus, mechanisms for detecting misbehavior and for efficiently resolving disputes are required. In this paper we propose a smart contract-based solution to enable such a market. Our design deters participants from misbehaving by resolving disputes via dedicated Mediators and by imposing enforceable fines through the smart contract. This is possible because we recognized that the results do not need to be globally accepted, convincing the JC will often suffice. We learned that due to the limitations of Ethereum our platform is only suitable for long running tasks, but there is space between the cost of electricity and AWS for a platform of this nature. Future work is looking into other platforms that support smart contracts, as well as leveraging improvements to Ethereum.

Acknowledgments We thank various anonymous reviewers for their insightful comments and valuable suggestions. We especially want to thank Prof. Gabor Karsai and Prof. Aniruddha Gokhale for their insight and comments on the paper. This work was supported in part by National Science Foundation (NSF) through award numbers 1647015 and 1818901. The views presented in this paper are those of the authors and do not reflect the opinion or endorsement NSF.

REFERENCES

[1] [n.d.]. Contained Advisor. Github, https://github.com/google/cadvisor.
[2] [n.d.]. ETH Gas Station. https://ethgasstation.info/calculatorTxV.php. Accessed: 2019-09-30.
[3] [n.d.]. Geth: Ethereum Command Line Interface. Online, https://github.com/ethereum/go-ethereum/wiki/geth.
[4] [n.d.]. iExec: Blockchain-Based Decentralized Cloud Computing. https://iex.ec/whitepaper/iExec-WPv3.0-English.pdf. Accessed: 12-01-2019.
[5] [n.d.]. Linux Container Images. http://crunchtools.com/comparison-linux-container-images/. Accessed: 2019-12-26.
[6] [n.d.]. MODiCuM GitHub. https://github.com/scope-lab-vu/MODiCuM.
[7] [n.d.]. seccomp. http://man7.org/linux/man-pages/man2/seccomp.2.html. Accessed: 2020-1-12.
[8] [n.d.]. Sharding roadmap. https://github.com/ethereum/wiki/wiki/Sharding-roadmap. Accessed: 2020-1-4.
[9] Amazon 2000. AWS Pricing Calculator. https://aws.amazon.com/lambda/pricing/. Accessed: 2019-09-30.
[10] D. P. Anderson, C. Christensen, and B. Allen. 2006. Designing a Runtime System for Volunteer Computing. In *Proceedings of the 2016 ACM/IEEE Conference on Supercomputing (SC)* (2006-11). 33–33.
[11] Mick Bauer. 2006. Paranoid Penguin: An Introduction to Novell AppArmor. *Linux Journal* 2006, 148 (Aug. 2006), 13.
[12] Mira Belenkiy, Melissa Chase, C Chris Erway, John Jannotti, Alptekin Küpçü, and Anna Lysyanskaya. 2008. Incentivizing Outsourced Computation. In *3rd Int. Workshop on Economics of Networked Systems*. ACM, 85–90.
[13] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 72–81.
[14] Thanh Bui. 2015. Analysis of Docker security. *arXiv preprint arXiv:1501.02967* (2015).
[15] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Queue* 14, 1 (2016), 10.
[16] Hadrien Croubois. [n.d.]. PoCo Series #2 — On the use of staking to prevent attacks. https://medium.com/iex-ec/poco-series-2-on-the-use-of-staking-to-prevent-attacks-2a5c700558bd. Accessed: 2019-12-26.
[17] Changyu Dong, Yilei Wang, Amjad Aldweesh, Patrick McCorry, and Aad van Moorsel. 2017. Betrayal, Distrust, and Rationality: Smart Counter-Collusion Contracts for Verifiable Cloud Computing. In *2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 211–227.
[18] Scott Eisele, Taha Eghtesad, Nicholas Troutman, Aron Laszka, and Abhishek Dubey. 2020. Mechanisms for Outsourcing Computation via a Decentralized Market. arXiv:2005.11429
[19] Laurence Field, D Spiga, I Reid, H Riahi, and L Cristella. 2018. CMS@ home: Integrating the Volunteer Cloud and High-Throughput Computing. *Computing and Software for Big Science* 2, 1 (2018), 2.
[20] IHS Technology. 2016. *IoT Platforms: Enabling the Internet of Things*. Technical Report. IHS Markit.
[21] Aron Laszka, Scott Eisele, Abhishek Dubey, Gabor Karsai, and Karla Kvaternik. 2018. TRANSAX: A Blockchain-Based Decentralized Forward-Trading Energy Exchange for Transactive Microgrids. In *24th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*.
[22] Dirk Merkel. 2014. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014), 2.
[23] Muhammad Nouman Durrani and Jawwad A. Shamsi. 2014. Volunteer Computing: Requirements, Challenges, and Solutions. *Journal of Network and Computer Applications* 39 (2014), 369–380. Issue Supplement C.
[24] Jianbao Ren, Yong Qi, Yuehua Dai, Yu Xuan, and Yi Shi. 2017. Nosv: A lightweight nested-virtualization VMM for hosting high performance computing on cloud. *Journal of Systems and Software* 124 (2017), 137 – 152.
[25] Luis FG Sarmenta. 2001. Sabotage-Tolerance Mechanisms for Volunteer Computing Systems. In *1st IEEE/ACM International Symposium on Cluster Computing and the Grid*. 337–346.
[26] S. Shekhar, A. D. Chhokra, A. Bhattacharjee, G. Aupy, and A. Gokhale. 2017. INDICES: Exploiting Edge Resources for Performance-Aware Cloud-Hosted Services. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*. 75–80. https://doi.org/10.1109/ICFEC.2017.16
[27] Jason Teutsch and Christian Reitwießner. 2017. A Scalable Verification Solution for Blockchains. Available online: https://people.cs.uchicago.edu/teutsch/papers/truebit.pdf.
[28] Sarah Underwood. 2016. Blockchain beyond Bitcoin. *Commun. ACM* 59, 11 (2016), 15–17.
[29] Rafael Brundo Uriarte and Rocco DeNicola. 2018. Blockchain-Based Decentralized Cloud/Fog Solutions: Challenges, Opportunities, and Standards. *IEEE Communications Standards Magazine* 2, 3 (2018), 22–28.
[30] Gavin Wood. 2014. *Ethereum: A secure decentralised generalised transaction ledger*. Technical Report EIP-150. Ethereum Project – Yellow Paper. 1–32 pages.