Incentivizing Outsourced Computation[*]

Brown University Technical Report CS-08-05

Mira Belenkiy[†]
Microsoft Corporation
mira@cs.brown.edu

Melissa Chase
Brown University
mchase@cs.brown.edu

C. Chris Erway
Brown University
cce@cs.brown.edu

John Jannotti
Brown University
jj@cs.brown.edu

Alptekin Küpçü
Brown University
kupcu@cs.brown.edu

Anna Lysyanskaya
Brown University
anna@cs.brown.edu

ABSTRACT

We describe different strategies a central authority, the *boss*, can use to distribute computation to untrusted *contractors*. Our problem is inspired by volunteer distributed computing projects such as SETI@home, which outsource computation to large numbers of participants. For many tasks, verifying a task's output requires as much work as computing it again; additionally, some tasks may produce certain outputs with greater probability than others. A selfish contractor may try to exploit these factors, by submitting potentially incorrect results and claiming a reward. Further, malicious contractors may respond incorrectly, to cause direct harm or to create additional overhead for result-checking.

We consider the scenario where there is a credit system whereby users can be rewarded for good work and fined for cheating. We show how to set rewards and fines that incentivize proper behavior from rational contractors, and mitigate the damage caused by malicious contractors. We analyze two strategies: random double-checking by the boss, and hiring multiple contractors to perform the same job.

We also present a bounty mechanism when multiple contractors are employed; the key insight is to give a reward to a contractor who catches another worker cheating. Furthermore, if we can assume that at least a small fraction h of the contractors are honest $(1\% - 10\%)$, then we can provide graceful degradation for the accuracy of the system and the work the boss has to perform. This is much better than the Byzantine approach, which typically assumes $h > 60\%$.

Categories and Subject Descriptors: C.2.4 [**Computer Systems Organization**]: Computer-Communication Networks—*Distributed Systems*; C.4 [**Computer Systems Organization**]: Performance of Systems
General Terms: Design, Economics, Performance, Security

1. INTRODUCTION

Many tasks exhibit an arbitrarily high appetite for computational resources. Distributed systems that coordinate computational contributions from thousands or millions of participants have become popular as a way to tackle these challenges. Examples include systems such as SETI@home [14] and Rosetta@home [13], which seek to analyze huge amounts of data in the search for extra-terrestrial life and a better understanding of protein folding, respectively. In these systems, every additional computational element added to the system provides greater utility.

This study is motivated by our efforts to build peer-to-peer systems that rely on cryptographic electronic cash (e-cash) to provide incentives for participation [2]. Such a system would prevent free-riding without sacrificing the privacy of its participants. However, the verification of e-coins by the bank is an expensive computational operation, and we wish to offload this work from the bank to the participants.

The naive solution is to simply give each peer a program to run (such as an e-coin verifier) and the input to this program (an e-coin). The peer would run the program and report the answer. There are several problems with this approach. First, without a reward, there is no incentive for participants to do any work. Second, even if the participants were compensated for their contribution, there is no incentive to perform the computation faithfully. Peers may report an answer at random or, perhaps, report an answer that they know *a priori* to be the most likely output of the computation (*e.g.,* that most e-coins are valid). Worse, if participants are malicious, they may choose to behave irrationally in order to force the bank to perform more work or accept incorrect results.

There problems are not limited to our e-cash application. SETI@home users have developed their own clients, for both malicious and selfish reasons [8, 10] (see Section 2). Multiplayer games cannot assume that players will not modify their clients to give themselves an in-game advantage.

Our solution assumes that there is some currency or credit system with which we can reward or fine contractors depending on their performance. This could be a reputation or credit system in which good contractors are awarded higher scores, or an actual currency which can be exchanged for some other services. This allows us to set incentives such that rational contractors will compute jobs correctly.

In this paper, we analyze how to the boss can set fines and rewards, and how often it will have to double-check the contractors' results in order to enforce the incentive structure. In Section 4, we define a game-theoretic framework to analyze different scenarios. Section 5 shows how to use collision resistant hash functions to increase the probability of getting a correct answer without increasing the fine-to-reward ratio and the amount of double-checking. In Section 6 we examine means of performing checks on contractors' answers,

[*]This work was supported in part by NSF Cybertrust award CNS #0627553.

[†]Work done while a graduate student at Brown University.

and consider outsourcing the same computation to multiple contractors, double-checking only if they disagree, as a way to reduce the amount of centralized double-checking. We also look at the effect of offering a bounty to a user who catches another contractor returning a wrong answer. Finally, in Section 7, we examine how to limit the damage that can be caused by malicious and colluding contractors, who seek to maximize the amount of centralized double-checking, or decrease the accuracy of submitted results.

2. RELATED WORK

Resource-sharing cluster systems such as Spawn [16], Popcorn [12], and Tycoon [6] focus on the efficient allocation of grid resources by providing auction mechanisms which award distributed resources to the highest bidder. Auctions provide a way to stem demand as computation becomes more expensive. However, these systems typically assume a federated—and friendly—environment where many parties wish to share a pool of trusted resources. Once awarded, resources are assumed to be available for use by the winner, without concern for malicious entities.

Our work has more in common with public-resource computing systems such as Distributed.net [4] and BOINC [3], which parcel and distribute computation to vast armies of volunteer users. BOINC provides scientific projects such as SETI@Home [14] and Rosetta@Home [13] with computational resources drawn from the idle CPU cycles of its users' home PCs, and its projects have attracted millions of participants. Greater participation is incentivized through a point system that rewards users who complete more work units with higher status on "leaderboards" published on the web.

BOINC's credits are not fungible—they are useful only for social status—yet even this incentive has greatly motivated participation, leading some to develop their own clients in an effort to claim more credit [15, 8]. In one case, a SETI@Home user developed an "optimized" client which returned outputs irreproducible by the official client, yet were otherwise indistinguishable. In another case, a patched client was released that simply performed no computation, returning bogus results [8, 10]. These examples and others provide inspiration for our model, which aims to address the problem of malicious and "corner-cutting" contractors who seek greater rewards by deviating from officially-sanctioned methods.

Systems based on Byzantine fault tolerance (BFT) [1] provide safety and liveness guarantees given a certain tolerable fraction of malicious users; typically at least two-third of participants must act correctly. The BAR model [7] provides incentive-compatible BFT primitives to extend these guarantees to both altruistic (*i.e.* correct) and rational nodes that may deviate from suggested protocols in pursuit of greater utility. Like these approaches, we also aim to incentivize rational nodes, but do not assume a quorum of correct nodes; instead we focus on incentives and probabilistic guarantees on accuracy that apply for varying fractions of altruistic and malicious users.

Checking intermediate computations has also been discussed for the problem of inverting one-way functions, where predefined intermediate steps are checked [5], and in general by redoing the computation until a randomly chosen intermediate step [9]. Molnar [8] suggests that contractors be required to provide a hash of the results of intermediate computations in order to force them to use the official algorithm. This is very similar to the approach we discuss in Section 5. However, the idea of using hashes was not formalized, and there was no discussion of how to combine this approach with incentive strategies for rational contractors.

3. MODEL

A central authority, the *boss*, will reward *contractors* to perform computational tasks, or *jobs*, on its behalf. The goal is to reduce the demand on the boss's own computational resources. We assume that contractors continually request new job assignments from the boss, but that they may freely choose when to stop requesting new jobs.

The boss will reward a contractor r for correctly completing a job. If the boss finds out that the contractor returned an incorrect result, the boss will fine the contractor f, which is subtracted from the contractor's accumulated earnings. The boss will not assign a job to a contractor unless the contractor has enough credit to pay the potential fine. As a result, we are concerned with reducing the fine-to-reward ratio (f/r): too high a ratio makes it harder for contractors to participate. As we will later see, there is a trade-off between the work the boss has to do and the f/r ratio.

Our definition of a *job* captures any efficiently computable task and its inputs. For the e-coin verification scenario, the only way the boss can make sure that a contractor properly verified an e-coin is to reverify it herself. Similarly, for the Folding@home project, the boss must refold the protein. For jobs in NP, the verification is much easier. However, the boss can only check an answer if the output of the computation is deterministic. If the job uses a randomized algorithm, the boss must provide the contractor with a random tape (*i.e.* a seed to a pseudo-random number generator).

The results of some jobs may be easier to predict than others. Consider a naïve decision problem formulation of the SETI@home project, "Is alien life detectable in this radio telescope data?" Or, for the e-coin verification task, "Do these values represent a valid e-coin?" A rational contractor may decide to conserve its computational resources and simply guess the most likely answer ("no" and "yes", respectively). We describe a hashing technique to detect incorrect answers, even for such highly skewed answer distributions.

Our payment- and penalty-based incentives assume the presence of an underlying economic framework in which the boss can enforce fines and rewards. In [2], peers use e-cash to exchange files; if the bank wishes to outsource tasks, it can easily increase and deduct account balances directly. BOINC similarly directly rewards users with credit that raises a user's ranking on the leadership board. A service provider boss (*e.g.,* a storage server) might reward contractors by providing them better service (*e.g.,* more storage), and fine them by reducing the service provided (*e.g.,* limiting their storage space). Real currencies might also be used if contractors offer the fine amount as deposit with the boss. Our model assumes only that a boss is able to withdraw f from and pay r to contractors.

4. BASIC CONSTRUCTION

Consider a contractor who has just been assigned a job by the boss. He faces two options: first, he may perform the job honestly, and receive a reward r. If we define the cost of computing a single job using the algorithm provided by the boss as $\mathsf{cost}(1)$, the expected utility $u(1)$ of an honest contractor is $u(1) = r - \mathsf{cost}(1)$. In this case, we assume that the boss sets r large enough to provide positive utility for the contractor, or he will refuse the job.

The contractor's second option is to return an output using an algorithm different from that specified by the boss. This might be possible, for example, if the contractor possesses *a priori* knowledge of the output distribution: it can simply guess the most likely output. Or, more generally, suppose the contractor has access to an alternative algorithm which provides a correct output with probability q (*e.g.,* SETI@home "optimized" client). Here, the contractor may still receive r, but risks being fined f if the boss discovers he has submitted an incorrect result.

We denote the probability that this *lazy contractor* will be caught submitting an incorrect result as p. However, we do not assume that the boss will be able to detect each incorrect result submitted and fine the guilty contractor: since checking the correctness of a submitted result may unduly waste computational resources. (We defer discussion of methods for checking results to Section 6.) Thus we can decompose p into two different values: the probability that the contractor's result is incorrect, and the probability that the result will be checked, when it is incorrect.

$$p = Pr[check \mid incorrect]\ Pr[incorrect]$$

We can analyze these two probabilities separately. First, let c be the probability that a contractor's result will be checked, conditioned on that contractor returning an incorrect result: $c = Pr[check \mid incorrect]$. The check can be performed by the boss or by other contractors. This also describes the case when the probability of a check is independent of the contractor's answer (*e.g.,* if the boss simply checks a fraction c of submitted outputs itself).

Next, we return to our definition of q, the probability of the contractor returning the correct answer using an alternate method. Clearly the probability that the contractor's answer is incorrect is $1 - q$. Thus

$$p = c(1 - q)$$

We also define the cost of the alternate method for obtaining a correct result with probability q as $\mathsf{cost}(q)$. We assume this cost is at most $\mathsf{cost}(1)$—otherwise, the contractor would simply run the suggested algorithm—and at least 0.

We can now define the expected utility $u(q)$ of a contractor, taking into account the probability p of being caught and his cost, as

$$u(q) = r(1 - p) - fp - \mathsf{cost}(q)$$

The contractor will receive a reward unless he is caught cheating, in which case he will be fined. Note that when $q = 1$, the contractor is performing the job correctly, and thus $p = 0$ and $u(q) = u(1)$ from our previous definition.

For a rational contractor, selecting a value of $q < 1$ and earning the expected utility $u(q)$ may present a lucrative choice, resulting in a potentially incorrect output. However, the boss can provide incentives to perform jobs correctly by setting f, r, and c.

THEOREM 1. *If the boss sets the fine-to-reward ratio to* $f/r \geq (1-p)/p$ *where* $p = c(1-\varepsilon)$ *then a rational contractor will return correct outputs at least* ε *of the time.*

PROOF. To prove this, we need to show that for any $q' < \varepsilon$, the resulting utility $u(q') < u(\varepsilon)$. Since we cannot argue about the cost functions of contractors realistically (contractors may value their resources differently, and it might also depend on the state of the contractor like his current load), we want to show $\forall q' < \varepsilon, u(q') \leq 0$. Remember, $u(q') = r(1 - p') - fp' - \mathsf{cost}(q')$, where $p' = c(1 - q')$. If we set $f/r \geq 1/p - 1 \geq 1/p' - 1$, then we guarantee that $r(1-p')-fp' \leq 0$. Thus, given such an f, r, c, any contractor who is not correct with probability at least ε will have negative utility. This means any rational contractor will either perform the job with accuracy at least ε, or will refuse to do the job. □

COROLLARY 1. *Any rational contractor will use the least costly algorithm that provides correct answers with at least* ε *probability.*

5. ACCURACY AND HASH FUNCTIONS

By setting the fine-to-reward ratio as above, the boss can require rational contractors to compute jobs correctly above a certain minimum accuracy requirement. Yet, obtaining high accuracy might require an infeasibly high fine-to-reward ratio, and for some applications even a small fraction of inaccurate results might be unacceptable.

Our concern is that there might be some alternate algorithm that costs the contractor very little (in terms of computation), and that produces the correct answer with some fairly high probability ϵ (*e.g.,* guessing a coin to be valid in the e-cash verification scenario). To prevent the contractor from using such an algorithm, we might have to set the fine-to-reward ratio unreasonably high.

Ideally we would like to ensure that the contractor actually runs the algorithm that we choose. Thus, instead of simply returning an answer, we could ask the contractor to send us the results of every intermediate computation. If we assume that the intermediate computations are small enough steps that the only way to get the correct intermediate result is by actually running the appropriate computation, then this will be sufficient to convince the boss that the contractor has run the computation correctly. Finally, to prevent the contractor from having to send a very large amount of information, we have him use a cryptographic hash function to hash all of this information into one short string. More formally:

DEFINITION 1. *An algorithm is assumed to be composed of a finite number of* ***atomic operations****. Each atomic operation is assumed to take a state information and output another state information. The* ***inner state*** *of an algorithm is defined as the concatenation of all the input/output states of the atomic operations of the algorithm, along with the definition of the algorithm in terms of atomic operations. The original algorithm for a given job is the one prescribed by the boss to the contractor. A hash function deterministically maps the inner state of an algorithm to a random* l*-bit string. Define* ***negligible*** *probability* $\mathsf{neg} = O(2^{-l})$.

We would like to assume that all algorithms which produce the correct result either have cost $\mathsf{cost}(1)$ or negligible success probability. However, there is always a potential mixed strategy which with some probability runs the original algorithm and with some probability makes a random guess of the inner state. Thus, we make the following assumption:

ASSUMPTION 1 (UNIQUE INNER STATE ASSUMPTION). (FOR INPUT DISTRIBUTION D AND NEGLIGIBLE neg')

Let $\mathsf{cost}(1)$ *be the cost of the original algorithm. We assume that any algorithm which has expected cost* $\gamma\mathsf{cost}(1)$ *(given a random input from D) will produce the correct inner state with probability at most* $\gamma+(1-\gamma)\mathsf{neg}'$ *(provided* $0 \le \gamma \le 1$*).*

Then we can say that a similar statement holds even after the application of the hash function:

THEOREM 2. *Let* $\mathsf{cost}(1)$ *be the cost of the original algorithm. Let* D, $\mathsf{neg}' < \mathsf{neg}$ *be such that the unique inner state assumption holds. Then under unique inner state assumption and the random oracle model*[1]*, any algorithm which when given a random input from D has expected cost* $\delta\mathsf{cost}(1) < \mathsf{cost}(1)$ *will produce the correct hash of the inner state with probability at most* $\delta + (1-\delta)\mathsf{neg}$ *(provided* $0 \le \delta \le 1$*).*

PROOF OF THEOREM 2. Consider the operation of the algorithm on a particular input. There are two ways that an algorithm can output the correct hash value. First, the algorithm might have queried the random oracle (to obtain the hash output) at the same inner state value as the original algorithm. That means by the unique inner state assumption that this operation must have cost $\gamma\mathsf{cost}(1)$ and succeed with probability $\gamma+(1-\gamma)\mathsf{neg}'$. Second, the algorithm might have produced the same hash without querying the random oracle at using the correct inner state. This has only negligible probability under the random oracle model. We have said that the algorithm has expected cost $\delta\mathsf{cost}(1)$. That means that it can be taking the first approach (following the correct probability) on at most $\frac{\delta}{\gamma}$ fraction of the inputs. Thus, on all other inputs, it has at best neg probability of success. That means that it's total success probability can be at most $\frac{\delta}{\gamma}(\gamma + (1-\gamma)\mathsf{neg}') + (1 - \frac{\delta}{\gamma})\mathsf{neg} \le \delta + (1-\delta)\mathsf{neg}$. $\square$

Finally, we conclude that if we set the parameters appropriately, a rational contractor will always use the original algorithm.

THEOREM 3. *Suppose that definition 1 holds for our input distribution. If* $\frac{f}{r} \ge \frac{1}{c}$*, and* $r > \mathsf{cost}(1)$ *and* $c > \mathsf{neg}/(1-\mathsf{neg})$*, then a rational contractor will use the original algorithm for the job.*

PROOF. Running the original algorithm results in utility $r - \mathsf{cost}(1)$. By theorem 2, any other algorithm will either have cost greater than $\mathsf{cost}(1)$ (and thus obviously lower utility), or will have cost $\delta\mathsf{cost}(1) < \mathsf{cost}(1)$ and success probability $\delta + (1-\delta)\mathsf{neg}$. That means the total utility will be $(\delta+(1-\delta)\mathsf{neg})r-(1-\delta-(1-\delta)\mathsf{neg})cf+(1-\delta-(1-\delta)\mathsf{neg})(1-c)r-\delta\mathsf{cost}(1)$. If f, r, c satisfy the conditions described in the theorem, then this utility will always be strictly less than $r - \mathsf{cost}(1)$, so the rational contractor will always run the original algorithm. $\square$

Using a hash function with output length 160 bits (*e.g.,* SHA-1), the boss can easily set f, r, c appropriately so that every rational contractor will use the original algorithm. For the rest of the paper, we can then assume $p \cong c$.

[1]The random oracle model is commonly used in cryptography. It assumes that the hash function behaves like a truly random function.

6. WHEN TO CHECK AN ANSWER

In Section 4, we analyzed how to set the fine-to-reward ratio f/r in terms of p, the probability that a contractor will be caught; *e.g.,* by setting $f/r = (1-p)/p$ the boss can provide incentives to rational contractors. In this section, we will examine different strategies the boss can use to actually catch the contractors. We will analyze $c = \Pr[\mathit{check}|\mathit{incorrect}]$, the probability that the boss or other contractors will check the answer of a contractor, conditioned on that contractor returning an incorrect answer.

6.1 Double Checking

A simple strategy is for the boss to randomly double-check an answers it gets with probability t. Here, the boss cannot know whether a job is incorrect until it has checked it, so $c = t$. Setting a low value of t allows the boss to reduce the amount of work needed for double-checking—but since c is inversely proportional to f/r, a high f/r may present an impractical barrier for contractors seeking jobs.

6.2 Hiring Multiple Contractors

The boss can try to minimize the amount of checking he has to do by farming out the same job to multiple contractors. The boss then double-checks a submitted result only if the contractors disagree.

The problem is that if all contractors output the same false answer, the boss will never catch them. In fact, the contractors find themselves in a situation similar to the the iterated prisoner's dilemma. The best strategy for all the contractors is to employ a tit-for-tat mechanism: they should cheat until another contractor performs the computation honestly [11].

We begin our analysis by assuming that a fraction h of the contractors will always perform the computation honestly: we call these contractors *diligent*. Later, we will show how to do away with this assumption. Suppose the boss chooses m contractors at random and assigns them the same job. We can describe c as the probability a contractor will be caught by other contractors if he submits an incorrect answer.

THEOREM 4. *Suppose the boss farms out a job to* m *contractors, each of which are honest with probability* h*, then the probability that a cheating contractor will be caught is* $c = 1 - (1-h)^{m-1}$.

PROOF. A contractor who submits an incorrect result will be caught only if there exists a diligent contractor in the group working on the same job. The probability that all of the other $m-1$ contractors are non-diligent is $L = (1-h)^{m-1}$. Thus the probability that at least one of the other $m-1$ contractors is diligent is $c = 1 - L$. $\square$

COROLLARY 2. *Suppose the boss farms out a job to* m *contractors, which are honest with probability* h*, then by computing* f/r *using* $p \cong c = 1-(1-h)^{m-1}$ *in section 4, the boss can guarantee that all rational contractors will act honestly all the time.*

This strategy still requires the boss to perform work when the results submitted by contractors are in disagreement. In a system where all the contractors are rational, there should be no disagreement at all. But if malicious or colluding contractors are present, they may try to force the boss to double-check by returning an incorrect answer. We analyze this behavior in Section 7.

6.3 Hybrid Strategy

The boss can also pursue a hybrid strategy: he can farm out a job to multiple contractors *and* randomly double-check some of the answers. Thus even if all contractors collude to give the same wrong answer, the boss can still catch them.

THEOREM 5. *Suppose the boss farms out a job to m contractors, which are honest with probability h. The boss also randomly double-checks the jobs with probability t when all the results agree. Then, $c = 1 - (1-t)(h^m + (1-h)^m)$.*

PROOF. The boss definitely checks the answer if there is at least one diligent and one cheating contractor in the group. This has probability $1 - h^m - (1-h)^m$. In any other case (probability $h^m + (1-h)^m$), all answers will agree and the boss will check with probability t. Therefore, we get $c = (1 - h^m - (1-h)^m) + (h^m + (1-h)^m)t = 1 - (1-t)(h^m + (1-h)^m)$. □

6.4 Hiring Two Rational Contractors

Now let us discuss how to shed the assumption that there are diligent contractors. In the iterated prisoner's dilemma it is assumed that in each round, a contractor plays against the same group of other contractors. In our scenario, the boss will randomly choose a new group of contractors for each job. The contractors are really playing a single round of the prisoner's dilemma many times with a different group of contractors. Thus, if we set f/r properly, the dominant strategy will be for the contractors to act honestly.

The table below computes the expected utilities $u(1)$ and $u(q)$ for a contractor depending on whether the other players all chose to be diligent or lazy. As before, q refers to the probability that a lazy contractor returns the correct answer. Please see Section 5 for how to use hashing to set q arbitrarily close to 0.

All Diligent	$u(1) = r - \mathsf{cost}(1)$
	$u(q) = rq - f(1-q) - \mathsf{cost}(q)$
All Lazy	$u(1) = r - \mathsf{cost}(1)$
	$u(q) = r - \mathsf{cost}(q)$

There are two Nash equilibria: If all other players cheat, a rational player will also cheat. If at least one player is honest, a rational player must also be honest.

We can break the cheating equilibria by introducing a bounty. If the contractors disagree on the output, the boss will check the computation and award b to all contractors who output the correct answer. Now the expected utility for being diligent when everyone else chooses to be lazy is $u(1) = r - \mathsf{cost}(1) + b(1-q)$.

THEOREM 6. *Suppose the boss asks two contractors to perform a job. Then the boss must set $f/r > 0$ and give a bounty of $b \geq r/(1-q)$ to honest contractors whenever they catch a cheating contractor.*

PROOF. We have that $r \geq \mathsf{cost}(1) \geq \mathsf{cost}(q)$. First, if all other players are diligent then a contractor is better off also acting honestly as long as

$$0 \geq rq - f(1-q) - \mathsf{cost}(q) > rq - f(1-q) - r.$$

As a result, we get $f/r > -1$. Since it makes no sense to have a negative fine (paying contractors for wrong answers) and since a negative reward (taking away money for right answers) discourages participation, we set $f/r > 0$. Second, if even one player is lazy, then the contractor has an incentive to be diligent as long as $r - \mathsf{cost}(1) + b(1-q) \geq r - \mathsf{cost}(q)$. The boss needs to set

$$b \geq \frac{r}{1-q} \geq \frac{\mathsf{cost}(1) - \mathsf{cost}(q)}{1-q}.$$

□

7. MALICIOUS CONTRACTORS

Malicious (or Byzantine) contractors attack the system: they want to reduce the accuracy of job results or increase the amount of double-checking the boss must do. They are irrational, or may pursue a utility function outside our model. Yet, to be able to stay in the system, they must keep at least a zero balance of utility (if they cannot afford the fine, they will not be hired by the boss). Malicious contractors may also collude, through centralized control (as in the Sybil attack), via external communication, and even by sharing resources (the reward r).

7.1 Independent Malicious Contractors

Even a malicious contractor must maintain a certain minimum balance in his bank account. Otherwise, the boss will not ask him to perform jobs. Thus, a malicious contractor intent on submitting as many incorrect results as possible must also compute jobs correctly *some* fraction of the time.

DEFINITION 2. *A malicious contractor will return the correct answer x fraction of the time, and an incorrect answer y fraction of the time; thus $x + y = 1$.*

We compute the utility of a single malicious contractor as

$$u(m) = xr + y(1-p)r - ypf,$$

where x and y are defined above and p is the probability that the contractor will be caught. We want to know how large a value y can the malicious contractor get away with while still maintaining a non-negative utility.

DEFINITION 3. *Let d be the deterrent factor, where the boss sets $f/r = d/p$. Observe that if $d = 1 - p$, this corresponds to our basic construction. Larger values of d indicate that the boss has decided to deter maliciousness by increasing the f/r ratio without decreasing the checks.*

THEOREM 7. *The fraction of incorrect results y that a malicious contractor can return to the boss is less than or equal to $1/(p+d)$.*

PROOF. The malicious contractor needs to have a non-negative balance: $0 \leq u(m) = xr + y(1-p)r - ypf$. We substitute $f = rd/p$ and $x = 1 - y$ in the inequality to get $0 \leq (1-y)r + y(1-p)r - yrd$. We get rid of r, and solve to get $y \leq 1/(p+d)$. □

COROLLARY 3. *Suppose the boss hires only one contractor for each job and sets $f/r = d/p$. Then the probability that the boss accepts an incorrect result is $g(1-p)/(p+d)$, where g is the fraction of malicious contractors in the system.*

Note that if the boss only randomly double-checks with some fixed probability, no malicious contractor can cause the boss to perform more work. However, in the setting where the same job is outsourced to multiple contractors and checked if there is disagreement, a malicious contractor can force the boss to perform a check by submitting an incorrect result, hence causing disagreement among the group.

7.2 Colluding Malicious Contractors

In our multiple-contractors scenario, the boss assigns each job to a randomly-selected group of size m, double-checking only when the contractors output different results, and fining those who submit an incorrect answer. We will examine two types of attacks by colluding contractors. In the first, the colluding contractors will try to trick the boss into accepting an incorrect answer. In the second, they will force the boss to perform extra checking by causing disagreements.

Theorem 8. *If the fraction of colluding contractors in the system is g, the probability that the boss accepts an incorrect result is at most g^m.*

Proof. The only way to trick the boss is if all the contractors in the group are colluders. For a group of size m, the probability that all group members are colluders is g^m. □

Colluding contractors may wish to force the boss to devote more resources to performing checks. The colluders can take advantage of the fact that if there is at least one colluder in the group, then one colluder can submit a wrong answer while the rest can submit the right answer and collect the reward. As a result, the overall utility of the colluding group can be high enough to allow the group to continue participating in the system.

Theorem 9. *The amount of work the boss needs to perform due to a group of maliciously colluding contractors which make up a g fraction of all the contractors is at most $pgm/(p+d)$.*

The proof of Theorem 9 requires the following Lemma. We omit the proof, which follows from the Binomial Theorem and basic algebra.

Lemma 1. *Let $P(k,m) = \binom{m}{k} g^k (1-g)^{m-k}$ be the probability that there are exactly k colluders in a group of size m. Furthermore, let $A = \sum_{k=1}^{m} P(k,m)$, be the probability that there is at least one colluder in the group. Then, $A = 1-(1-g)^m$. Finally, let $B = \sum_{k=1}^{m} P(k,m)k$. Then, $B = gm$.*

Proof of Theorem 9. We will first define the total utility of the colluding contractors. The contractors' strategy is simple. If there is at least one colluder in the group chosen by the bank, then one of the colluders will output a wrong answer with probability $y = 1-x$ while the rest output the correct answer. Then the total utility of the colluders for one job will be $xkr + y((k-1)r - f)$ for k colluders (with probability $x = 1-y$, all colluders will get the reward by outputting the correct answer, and with probability y only one of them will get fined while the rest will be rewarded). If we sum over the probability that the there are k colluding contractors in a group of size m, we get the total expected utility of the colluders.

$$\begin{aligned} u(c) &= \sum_{k=1}^{m} P(k,m)[xkr + y((k-1)r - f)] \\ &= xrB + yrB - yrA - yArd/p \end{aligned}$$

if we do the substitutions for A, B and f. The colluders want to maximize y while keeping their total utility positive: $u(c) \geq 0$. Then, rearranging the equation above gives us

$$y \leq \frac{B}{A(1+d/p)} = \frac{pgm}{(p+d)A}.$$

Next, we note that, a job will provide this group of colluders the ability to cheat in order to make the boss work more only if there is at least one colluder in that group. So, A fraction of the jobs will enable the colluders to force the boss for a check. Therefore, by multiplying y with A, we obtain the fraction of the time colluders can cause the boss to work, which is at most $pgm/(p+d)$. □

8. EVALUATION

Throughout the paper we have presented various methods by which the bank can tune the fine-to-reward ratio through setting other parameters. In this section, we show how the boss can select system parameters that balance performance trade-offs with protection against malicious contractors. We begin with the selection of the ratio f/r and group size m, depending on the percentage of honest contractors h in the system. The trade-off between high fine-to-reward ratio (which may present a barrier to entry for contractors) and large group size (which may unnecessarily waste effort due to redundant computation) is depicted in Figure 1. It can be seen from the figure that even a group size of 2 is enough to allow a reasonable fine-to-reward ratio, even in the presence of a very low percentage of honest contractors. Obviously, the higher the percentage of honest contractors, the smaller the group size required.

In the figure, we assumed that all the other contractors are rational. Assuming that the boss's view of the percentage of honest contractors is not higher than that of the contractors', the fine-to-reward ratios shown on the figure will provide incentives for rational contractors to always behave honestly. Next, we analyze the effect of irrational malicious and colluding contractors on the system when we set the fine-to-reward ratio so as to incentivize rational contractors.

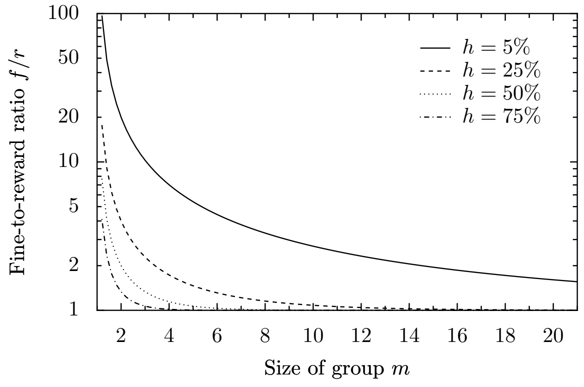

Figure 1: Example parameter settings for f/r and m that provide valid incentives assuming a fraction h of honest users. (Theorem 4)

Figure 2 shows the percentage of bogus results the irrational malicious and colluding contractors, who are not incentivized by our scheme, can cause the boss to accept. The boss can adjust the deterrent factor to deter malicious contractors by increasing the fine-to-reward ratio without decreasing the probability of catching them. The figure shows the case when the boss employs 2 contractors per job, and thus represents a worst-case multiple-contractor scenario. When more contractors are employed, the fraction of bogus results accepted by the boss will be lower, since the colluders

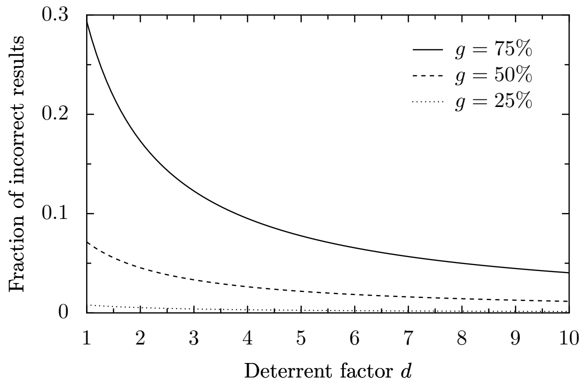

Figure 2: The maximum fraction of incorrect results that the boss will accept due to a fraction g of malicious contractors, for different settings of the deterrent factor d. (Corollary 3)

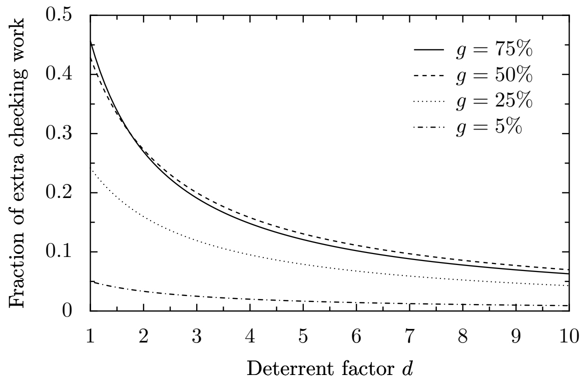

Figure 3: The maximum amount of extra double-checking work that a group of malicious colluders controlling a fraction g of all contractors can force the boss to perform, for different settings of the deterrent factor d. (Theorem 9)

need to control the entire group in order to cheat the boss.

Next, in Figure 3, we see the fraction of extra double-checking work the colluding contractors can force the boss to perform. The figure again uses a group size of 2. Increasing the group size makes things worse in this case: the reason is that the colluders can make the boss work only if there is at least one of them in the group the boss selects. When the group size increases, the chance of that happening increases. An interesting point to make is that if the boss's probability of catching the colluders increases, then he obviously needs to perform more work. Luckily, the fraction of bogus results that will be accepted is bounded as in Figure 2.

Note that the number of honest contractors do not affect the performance of the system, in terms of both the percentage of bogus results and extra work for the boss, once the fine-to-reward ratio is set. This is the case because once the ratio is set according to the fraction of honest contractors, then every rational contractor will have incentive to perform the job correctly. If the system is dynamic and the percentage of honest contractors decrease, the fine-to-reward ratio needs to be readjusted.

Our system can deter maliciousness without very high fine-to-reward ratios or large group sizes even if there are very few honest contractors in the system. In most cases (except when there is an extremely low number of honest users, *i.e.* $h = 0.05$, or an extremely high number of malicious users, *i.e.* $g = 0.75$), a deterrent factor of $d = 5$ and a group size of $m = 2$ is enough to result in a practical fine-to-reward ratio ($f/r \leq 25$), while guaranteeing at most 10% of bogus results and about 15% more work in very unrealistic highly adversarial scenarios (75% malicious), or almost no bogus results and about 5% more work in more realistic scenarios (5% malicious).

9. CONCLUSION AND FUTURE WORK

We have presented different techniques that can be applied for incentivizing outsourced computation, through redundant computation by the boss or other contractors. The hashing technique prevents the use of other algorithms than prescribed by the boss. Then, we showed how to set the fine-to-reward ratio in presence of irrational honest users (Section 6.2), or when the contractors cannot collude in large scale in the long run (Section 6.4). Finally, we have shown that using our techniques, a reasonable fine-to-reward ratio can incentivize all rational users to behave honestly, and limit the damage by irrational malicious contractors.

All of these techniques aim to decrease the amount of work our centralized boss needs to perform. We assumed that this boss can afford to pay all rewards and is capable of fining the contractors: another possibility is that multiple bosses might be in agreement with an entity of such power. Then, before a job is outsourced, each contractor might provide an escrow of the fine, so that the boss can claim it if cheating is detected. Additionally, bosses might provide different incentive structures f/r to different peers, offering higher prices to those willing to accept larger fines. In such a decentralized environment, designing a distributed, budget-balanced mechanism provides a direction for our future work.

Finally, the currency used by our system could also serve other purposes, *e.g.,* to buy data as in the currency-based P2P system of Belenkiy *et al.* [2]. In future work, we will study the effects of outsourcing e-coin verification on this system's virtual economy.

10. ACKNOWLEDGEMENTS

We would like to thank Eric Ely Rachlin and Markulf Kohlweiss for their helpful discussions, and Hatice Şahinoğlu for her help in proving some theorems.

11. REFERENCES

[1] A.S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proc. of ACM SOSP*, 2005.

[2] M. Belenkiy, M. Chase, C. Erway, J. Jannotti, A. Küpçü, A. Lysyanskaya, and E. Rachlin. Making P2P accountable without losing privacy. In *Proc. of WPES*, 2007.

[3] BOINC. `http://boinc.berkeley.edu`.

[4] Distributed.net. `http://www.distributed.net`.

[5] P. Golle, and I. Mironov. Uncheatable distributed computations. *CT-RSA*, 2001.

[6] K. Lai, L. Rasmusson, E. Adar, L. Zhang, and B.A. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent and Grid Systems*, 1(3):169–182, 2005.

[7] H.C. Li, A. Clement, E.L. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. BAR Gossip. *OSDI*, 2006.

[8] David Molnar. The SETI@Home problem. *ACM Crossroads*, Sep 2000.

[9] F. Monrose, P. Wyckoff, and A. Rubin. Distributed execution with remote audit. *ISOC NDSS*, 1999.

[10] Patch-free-processing. `http://web.archive.org/web/20070207064618/http://home.hccnet.nl/a.alfred/p-free-p1pfp.html`.

[11] Anatol Rapoport. Prisoner's dilemma - recollections and observations. *Game Theory as a Theory of Conflict Resolution*, 1974.

[12] O. Regev and N. Nisan. The POPCORN market. Online markets for computational resources. *Decision Support Systems*, 28(1-2):177–189, 2000.

[13] Rosetta@home. `http://boinc.bakerlab.org/rosetta/`.

[14] Seti@home. `http://setiathome.berkeley.edu`.

[15] truXoft Calibrating BOINC Core Client. `http://boinc.truxoft.com/core-cal.htm`.

[16] C.A. Waldspurger, T. Hogg, B.A. Huberman, J.O. Kephart, and W.S. Stornetta. Spawn: A Distributed Computational Economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, 1992.