Incentivized Outsourced Computation Resistant to Malicious Contractors

Alptekin Küpçü
Koç University

December 12, 2014

Abstract

With the rise of Internet computing, outsourcing difficult computational tasks became an important need. Yet, once the computation is outsourced, the job owner loses control, and hence it is crucial to provide guarantees against malicious actions of the contractors involved. Cryptographers have an almost perfect solution, called fully homomorphic encryption, to this problem. This solution hides both the job itself and any inputs to it from the contractors, while still enabling them to perform the necessary computation over the encrypted data. This is a very strong security guarantee, but the current constructions are highly impractical.

In this paper, we propose a different approach to outsourcing computational tasks. We are not concerned with hiding the job or the data, but our main task is to ensure that the job is computed correctly. We also observe that not all contractors are malicious; rather, majority are rational. Thus, our approach brings together elements from cryptography, as well as game theory and mechanism design. We achieve the following results: (1) We incentivize all the rational contractors to perform the outsourced job correctly, (2) we guarantee high fraction (e.g., 99.9%) of correct results even in the existence of a relatively large fraction (e.g., 33%) of malicious irrational contractors in the system, (3) and we show that our system achieves these while being almost as efficient as running the job locally (e.g., with only 3% overhead). Such a high correctness guarantee was not known to be achieved with such efficiency.

Keywords: Outsourced Computation, Cloud Computation, Crowdsourcing, Fair Payments.

1 Introduction

Internet computing is an important field, both for academic research, and for the industry. With the rise of distributed computing, cloud computing, and crowdsourcing, outsourcing difficult computational tasks presented itself both as a challenge and as an opportunity. Mainly, two types of computation outsourcing is considered in the literature. In one type, the task is outsourced to a single powerful entity, who has more resources to run the task. For example, outsourcing a computational job to an Amazon EC2 server would be of this kind. In the second type, the job is outsourced to multiple small entities,

such as in the SETI@Home project. In the SETI@Home project, people's desktops and laptops are performing the computation. There are two possible gains from such an outsourcing: (1) the entities to which the job is outsourced collectively will have greater resources, or (2) simply put, the job will be outsourced, freeing up the resources of the outsourcer. In this paper, we will focus on this second type of outsourcing where a task is sent to multiple entities, calling the outsourcer *the boss* and the participating entities *the contractors*. **As long as the boss employs *multiple* contractors, our solution does *not* need to differentiate between outsourcing to multiple cloud providers or crowdsourcing to people's computers.**

In our scenario, the boss would like to outsource a computational task to *untrusted* contractors. Furthermore, the setting is that the *diligent* contractors who act honestly will be *rewarded* by the boss, whereas the *lazy* contractors who act maliciously (and get caught) will be *fined*. Contractors in the system are divided into three groups: *Honest* contractors who always perform the job diligently, *rational* contractors who would cheat if the utility of cheating is better than not doing so, and *malicious* contractors who would cheat as much as possible, even though its utility may be worse than acting diligently. Note that in essence, honest and malicious contractors are not rational. This setting has been analyzed in our previous work ([4]) and solutions were developed to incentivize rational contractors to act honestly. As explained in [4], *it is impossible to limit the damage of the malicious contractors without using fines*, since they are irrational. In our current work, we extend our previous work to handle the following:

- We present a game for multiple contractors in a framework where diligent contractors who help catch the lazy ones get an extra reward in the form of a *bounty*. We show how to set up this game to obtain a strict Nash equilibrium of all contractors acting diligently. Our previous work considered this setting only for *two* contractors. Moreover, we show that *the bounty is not necessarily an extra cost* to the boss.

- We analyze *four* different types of *malicious* contractors that correspond to various real world attack scenarios, and present results in mixed maliciousness setting. Our previous work only considers *two* of these types, and does not present any theoretical bounds for the bounty framework.

- We have implemented the boss and the contractors (including all types of malicious contractors) in Java, and present numbers from real runs. We then compare these results to the theoretical ones. Our previous work only presented theoretical results. Presenting real results is not only interesting, but it also is essential when one mixes different types of malicious contractors. Theoretical analysis of mixed fractions of various types of malicious contractors gets complicated very quickly.

- The experimental results show that it is possible for the boss to obtain **99.9% correct results** with **only 3% extra work**, even when about **a third of all the contractors are malicious**.

- Our previous work considers the boss as a trusted party, and lets the boss handle all monetary issues (i.e., the payment of rewards and the deduction of fines). In this work, we assume the boss is *untrusted*, and employ cryptographic payment

mechanisms to perform payments, with the help of a trusted bank, creating a realistic setting for practical use.

In doing these, our main goal is to ensure the outsourced job is computed correctly. We not only require that all the rational contractors will have incentive to compute the job honestly, but also enforce a bound on the damage any fraction of malicious contractors can cause. The main damage malicious contractors may cause on the system is *letting the boss accept incorrect results* without realizing it. We also consider an additional type of attack where the malicious contractors *elongate the job computation time* by sending incorrect results that are detected (since the job shall be re-outsourced in that case). We provide proven bounds limiting both of these attack types.

Thus, in summary, we present a multi-contractor outsourcing game and show that with proper values of fines, rewards, and bounties, we achieve a strict Nash equilibrium, and can incentivize all rational contractors to act diligently. We further analyze malicious contractors and present theoretical results about how much extra work they can cause, and how good the correctness guarantee of our system is in the presence of malicious contractors. Our experimental results confirm the validity of the theoretical results in this paper. Finally, we present a payment scheme for the fines and rewards, such that the boss and the contractors mutually distrust each other, but there is a trusted bank in the system. We argue that even though the boss would want to pay fewer rewards and the contractors would try not to pay the fines, using our scheme such attacks are thwarted.

1.1 Related Work

Outsourced computation approaches differ greatly in the type of guarantees they provide, the techniques they employ, and the efficiency of the solution. On one end of this broad spectrum lies solutions based on **fully homomorphic encryption** [25]. These solutions can provide very strong security guarantees where a single powerful entity is playing the role of the contractor. The security guarantee is that both the job to be computed and the input to it are hidden from the contractor (or anyone other than the boss). Unfortunately, these solutions are still highly impractical for general use [26].

Another line of work for outsourcing computation to multiple intercommunicating participants is based on **secure multi-party computation** (SMPC) [27, 6]. SMPC is a general technique that lets multiple users compute a *known* function over *private* inputs. Normally, it is not used for outsourced computation, but indeed for *joint* computation. Yet, recently there is interest in bringing these techniques to the outsourced computation setting [36, 35, 43]. Some special purpose SMPC schemes can be highly efficient, leading to practical outsourcing capabilities. Unfortunately, SMPC for general computation is *not* efficient, and thus we are still required to seek other options for outsourcing computation.

A main requirement from outsourced computation in terms of security is the **correctness** of the returned results. This can be achieved using **zero-knowledge proofs** [30, 28, 5]. This way, the contractor may prove to the boss that the computation is done properly, and the result that was returned is indeed the correct one. When the job or the input/output is not necessarily hidden from the contractor, one does not need the extra zero-knowledge property, and may use regular proofs. Outsourcing computation this way is called **verifiable computing** [24]. Similar to above, this technique is efficient for some

special cases, but *not* when used for general computation. In Section 6.2 we show that using our system results in more than **4 orders of magnitude better performance** at the contractor side compared to employing Pantry [10] verifiable computation (one of the state of the art in verifiable computation). See [59] for a survey of such techniques.

Other outsourced computation techniques include ones based on *attribute-based encryption* [49], *functional encryption* or *secure function evaluation* [50], *trusted hardware* [50], and *probabilistically checkable proofs* [29, 57]. There are also several implementations besides ours [48, 57, 52, 53, 10], and architectural contributions [11]. Some application-specific solutions exist as well, such as *algebraic computations* [8], *polynomial evaluation* [23, 7], *map-reduce* [61], *sequence comparisons* [3], *linear programming* [60], *modular exponentiation* [34, 18], and outsourcing to *sensor devices* [32]. Furthermore, distributed computation solutions such as *Byzantine agreement* or *Byzantine fault tolerance* techniques may also be employed [9, 1, 42], but again these techniques are inefficient when it comes to solutions supporting general computation.

Verifying correctness of an outsourced computation by making contractors perform **extra work that is easy to check** has also been an interesting direction [31, 22, 51, 55, 56]. In essence, the **inner state hash** technique we employ from Belenkiy et al. [4] also fits into this category. The contractors need to compute some hashes during the computation, and the boss can easily verify those by comparing against the hashes provided by the other contractors. We show in this paper that this is a very cheap technique in terms of wasted contractor time, much faster than some other proposals (e.g., Pantry [10]). Furthermore, our solution requires extremely small amount of work at the boss (just a simple equality comparison of results and hashes), while some other solutions require re-computing the outsourced task up to a point [45].

Incentivizing **crowdsourcing to *human* contractors** have also been considered [63, 33]. In such works, there is an additional mediator (e.g., Amazon Mechanical Turk, Yahoo Answers) and the main concern is to maximize the profits. See [62, 64] for a survey of such works.

In terms of **payments of the rewards and fines**, Carbunar and Tripunitara [15] present a system using *ringers* [31], achieving success rates up to 99%. This is an important contribution, since most schemes assume trusted payment exchanges. In comparison, our solution is cryptographically strong. Their scheme was later improved by Chen et al. [19], but without considering the payment of the *fines* as we do in our setting.

Lastly, it is worth mentioning how **real world systems** are working. Prominent examples include SETI@Home[1] and Rosetta@Home[2], as well as other BOINC[3], World Community Grid[4], and Distributed.net[5] projects. These projects are in the multi-contractor setting that we described. Unfortunately, these systems currently offer no guarantees on the correctness of the results [44]. Moreover, even though the current systems do employ rewards, they do *not* employ *fines*, and so as explained in [4], they *cannot* achieve the security guarantees we provide against irrational malicious contractors.

[1] `http://setiathome.berkeley.edu`
[2] `http://boinc.bakerlab.org/rosetta/`
[3] `http://boinc.berkeley.edu`
[4] `http://www.worldcommunitygrid.org`
[5] `http://www.distributed.net`

Interestingly enough, van Dijk and Juels claim that **cryptography alone is not enough for many cloud computing applications** [58]. In essence, this claim backs up our solution in this paper, since we employ *cryptography as well as game theory and mechanism design* to provide a secure system for outsourced computation.

1.2 Contributions

The need: As we have seen, **cryptographic solutions for general computation are inefficient**, **purely game-theoretical solutions fail to provide guarantees against irrational malicious contractors**, and Belenkiy et al. [4] solution does not provide fair payments unless the boss is trusted by all. Therefore, there is a need for efficiently outsourcing computation to untrusted contractors such that the results are correct with high probability and the overhead is small, even when there is a relatively high fraction of malicious contractors in the system. To that end, our contributions are as follows:

- We follow Belenkiy et al. [4] observation that fines are an efficient way to incentivize rational contractors and are *necessary* for resilience against malicious contractors. In addition to that observation, we create a **multi-player outsourced computation game** that includes an additional bounty given to the diligent contractors who help catch the lazy (cheating) contractors. We **incentivize all rational contractors to act diligently** in this new multi-player bounty framework.

- To the best of our knowledge, we categorize the malicious contractors into four different types with various abilities matching realistic scenarios, for the first time. This presents the opportunity to analyze a more realistic setup.

- For the first time, to the best of our knowledge, we consider payments of *rewards and fines* using a trusted bank. In the payment setting, we assume the boss is interested in paying fewer rewards and the contractors are interested in not paying the fine, if possible. We devise a **cryptographic mechanism to enforce fair payments**.

- We implement the boss, and different types of contractors, including honest and malicious ones. We present real experiment results from this implementation and compare against theoretical results. We also implemented the inner state hash mechanism for the first time, obtaining only 0.2% overhead.

- Via our implementation, we can, for the first time, present results for a system with different fractions of various types of malicious contractors. Our tests using a mixed fraction of different types of malicious contractors yield **99.9% correct results** from outsourced computations, even though **one-third of all the contractors are malicious**. We note that measuring this limit is a hard task to tackle theoretically, but becomes easy with our implementation.

- We show that our system's expected **outsourcing overhead** compared to executing the task locally is **2%, even when a quarter of all contractors are malicious**.

1.3 Overview

As discussed before, our goal is to **very efficiently ensure correctness** of the computation results. We are considering the setting where the boss wants to outsource the same job to multiple contractors. In general, job assignment to contractors will be done *randomly*. When all the contractors assigned to the same job return their answers, the boss will compare them. **If all the answers match, the boss will reward the contractors and accept that answer as the result of the computation.** We prove that this result will be correct with very high probabilities, even though a relatively high fraction of the users are malicious.

If there is a mismatch between the answers returned for the same job, then the boss must somehow find the correct answer, and then reward the contractors who returned the correct result and fine the ones who returned a different answer. In addition, the contractors who returned the correct result will be provided an extra reward in the form of a *bounty*, since they helped the boss catch the lazy or malicious contractors.

There are multiple methods for the boss to understand who returned the correct result, in cases of mismatches. One possibility is for the boss to re-do the computation and find the correct result himself. Another possibility is for him to use a different verification algorithm. For example, if the computation is an NP computation, then it has a verification algorithm in P. The boss can run this verification algorithm to determine which answer is the correct one. Yet another alternative is for the boss to re-outsource the computation to another random set of contractors, until an all-matching response set is obtained. Then, he can go back and compare the previous response sets and distribute the fines, rewards, and bounties accordingly. This last method is the one we suggest and analyze in this paper.[6]

In our presentation, we first describe the system assuming all the contractors are rational; they either act **diligently** and return the correct result, or act **lazily** and return a possibly incorrect answer. Then, we include irrational contractors who may behave **honestly** or **maliciously**. In both discussions (only rational contractors case, or including irrational contractors), we provide provable results in the strongest form possible: We consider the worst-case scenarios for the boss, and assume all lazy or malicious contractors are colluding together to harm the system (either by making the boss accept incorrect results, or by forcing him to spend his resources to deal with the mismatches). We carefully **formulate and prove our theorems against these strongest form of attackers**. Note that *purely game-theoretical* mechanisms fail to provide solutions against irrational malicious contractors.

Realize that contractors may cause two main types of harm to such an outsourcing system. The worst one is that they may (jointly) **try to make the boss accept an incorrect result without realizing** that it is incorrect. In our system, remember that if all the answers match, then the boss would accept the result as correct. Thus, if all contractors that are assigned to the same job return the same incorrect answer, then the boss will mistakenly consider it as the right answer. Unfortunately, this is a common scenario, where all lazy or malicious contractors use the same fake algorithm that we call a *q-algorithm*. (See Section 4.1 for more discussion and real examples of fake algorithm uses

[6]The boss may choose to take the majority answer in cases of mismatch, but this requires additional theoretical analysis not shown in this paper.

in practice.) Fortunately, we formulate our system according to this worst-case setting and show that by setting our system's parameters properly, **the boss can ensure a very high fraction of correct results**.

The other type of harm that the malicious contractors may cause is to **elongate the job computation time by returning inconsistent results**. Remember that the boss would re-outsource the job if the answers are not all matching. In such a case, the boss is not accepting an incorrect result, but instead is realizing that there is a problem, and hence needs to spend more effort in obtaining the correct answer eventually. We will again provide bounds on the expected total effort spent, and show that **the overhead would be very small** even in settings with a high fraction of malicious contractors.

Our experimental results are given for scenarios with high fraction of malicious contractors, confirming our theoretical results, as well as more realistic mixed-maliciousness scenarios where not all malicious contractors are colluding together, but rather some are independently malicious. We realize that in this realistic (but still conservative) setting, with proper fine to reward ratio settings, **the boss obtains more than 99.9% correct results with only 3% extra work, even though we assume about one-third of all contractors are malicious**.

Finally, we also tackle the problem of *fair payment of fines and rewards*. This is important in the settings where the contractors and the boss mutually distrust each other. We solve this problem using newly-developed optimistic fair exchange mechanisms that are specialized for this outsourced computation setting with multiple contractors. Note that in our scheme, the boss needs to give a reward each time all the answers returned by the contractors assigned to the same job match. Essentially, this means a contractor can prove he is diligent by making the trusted bank also compare these results, if the boss fails to keep his promise. To help this process, we use a simple timestamping server who certifies that a particular contractor indeed returned a particular result at a particular time. This timestamping server acts as a notary such that the bank can later verify the claimed results. Similarly, a contractor may try to cheat by not paying the fine even when he returns an incorrect result. We show that **our protocol achieves fair payments against cheating attempts by both the boss and the contractors**.

2 Setup and Model

Remember that we call the entity who outsources the job *the boss*, and the entities who perform the outsourced task *the contractors*. In our model, we assume there is a single boss (who might as well be acting on behalf of other entities outside the model), and there are many candidates for contractors. The reward given by the boss to the contractors, who the boss believes to have performed the job correctly, is denoted r, and the fine taken from the malicious contractors, who the boss catches to have cheated, is denoted f. Furthermore, in the bounty framework, the bounty given to the diligent contractors who help the boss catch the cheating contractors is denoted b.

We consider three main types of contractors. When we say that a contractor is *honest*, we mean she performs the job exactly as requested by the boss. If a contractor is *rational*, then she performs the job exactly as requested by the boss, as long as she has an incentive to do so. In particular, the utility of performing the job correctly must be greater than the

utility of doing anything else. Of course, by our assumption that the entity is willing to become a contractor, this implies that the reward r is greater than the cost of performing the job. Lastly, if a contractor is *malicious*, then she tries to harm the system, mainly by trying to get the boss accept incorrect results without detection of cheating, or trying to elongate the job completion time by forcing the boss to re-outsource, as long as these are within her capabilities. Since there is a fine taken from malicious contractors when they get caught, as we will see, it is *impossible* for a malicious contractor to always cheat. Indeed, most of the time, they are forced to act diligently.

To model the utilities of these different types of contractors, we first need to model the cost of executing a task. Note that dishonest contractors may try to employ some other, presumably cheaper, algorithm for the job, still trying to get the reward. We define a generic class of such algorithms as follows:

Definition 2.1 (q-Algorithm)**.** *An algorithm that a contractor uses to compute an assigned job that outputs the correct answer with probability q is a q-algorithm. When the contractor uses the original prescribed algorithm, it means $q = 1$. If the contractor cheats, it is necessarily the case that $q < 1$ (since using another algorithm that always returns the correct answer is not necessarily considered cheating, if such an algorithm exists). The cost of employing a q-algorithm is denoted as $\mathsf{cost}(q)$.*

We will denote with $\mathsf{cost}(1)$ the cost of performing the job exactly as prescribed by the boss. With this definition, if a contractor cheats, we require that $q < 1$ and $\mathsf{cost}(q) < \mathsf{cost}(1)$, meaning that the cost of cheating will be less than the cost of running the original task, since otherwise it would be irrational to cheat anyway. For malicious contractors, it may make sense to still cheat even if $\mathsf{cost}(q) = \mathsf{cost}(1)$, but the existence or non-existence of the equality will not change any of our results.

Following Belenkiy et al. [4], we assume that the outsourced task is composed of a finite number of *atomic operations*. Also define the ***inner state*** of an algorithm as the concatenation of all the input/outputs of the atomic operations of the algorithm. The critical observation here is that when one employs a *collision-resistant hash function*, the hash of the inner state of the original algorithm would, with high probability, be different from the hash of the inner state of any other q-algorithm. In other words, **the probability that *any* other q-algorithm outputs the correct inner state hash as the original algorithm is *negligible*** [4]. This essentially means that by concatenating the output with the inner state hashes, we have $q \approx 0$ for any algorithm other than the original algorithm.

The high level idea in such an outsourced computation setting is to outsource the same job to a group of contractors (group size $m > 1$), and then collect back their answers together with the hash value of the inner state. Even when the output of an algorithm is long or there are many steps of the algorithm, the hash values will be very short having constant length (e.g., 160 bits). The beauty of this approach is two-fold. First, as we show in the performance results, the overhead of the additional inner state bookkeeping and hashing necessary is very minimal, around 0.2% compared to the original algorithm running time. Second, the boss's job is extremely easy and fast: just compare outputs and inner state hashes for equality. If all the returned results are the same, the boss will just accept one of the answers (e.g., the first answer) as the correct answer. If there is a discrepancy, the job needs to be re-outsourced.

We now define the notation for the utility of a contractor. Remember that the *diligent* contractors are the ones who performed the job correctly by running the original algorithm, and the *lazy* contractors used some other q-algorithm instead. Let $\mathsf{u}_k(q)$ be the utility of a contractor when there are k lazy contractors in the rest of the group, and when he uses a q-algorithm. Further let $\mathsf{U}(k, m)$ be the total combined utility of k malicious contractors out of a group of m contractors to whom the job is outsourced.

3 Rational Contractors

In this section, we consider the case where the boss employs a group of m randomly-chosen *rational* contractors for a given job. The boss accepts an answer only if all the returned results match, and re-outsources the job otherwise. Our goal is to incentivize all rational contractors to act diligently.

3.1 Two Contractor Case

Belenkiy et al. [4] define a Prisoner's Dilemma-like game, where the contractors are socially better off if they act maliciously simultaneously. Table 1 shows a two player game, listing the expected utility of a player depending on whether the other player is diligent or lazy. Notation-wise, $u(1)$ denotes the utility of a diligent contractor, and $u(q)$ denotes the utility of a lazy (cheating) contractor.

If this contractor is diligent, regardless of what the other contractor does, she will be rewarded r, but she pays the cost $\mathsf{cost}(1)$ of performing the job as directed. If both contractors are lazy, the boss will accept an incorrect result, since he only verifies the results in case of a mismatch, and we are considering the worst-case scenario where both contractors use the same q-algorithm and return the same incorrect result. They will both be rewarded r, and pay only the cost $\mathsf{cost}(q)$ of the q-algorithm that they used. On the other hand, if this contractor is lazy while the other is diligent (second column first row), she will be caught if she returns an incorrect answer (which happens with probability $1 - q$). When she gets caught, she will be fined f. If she happens to return the correct answer (which happens with probability q),then she will be rewarded r for her work. In any case, she incurs the cost of the q-algorithm that she used. Table 1 presents this logic clearly.

Other $\diagdown$ This contractor	Diligent	Lazy
Diligent	$u(1) = r - \mathsf{cost}(1)$	$u(q) = rq - f(1-q) - \mathsf{cost}(q)$
Lazy	$u(1) = r - \mathsf{cost}(1)$	$u(q) = r - \mathsf{cost}(q)$

Table 1: 2-contractor game from [4].

There are two Nash equilibria in this game: Both players lazy (cheating), or both players diligent (honest). The goal is to incentivize all contractors to act diligently. For this purpose, Belenkiy et al. [4] introduce a bounty scheme which breaks the lazy equilibrium, leaving both contractors being diligent the only Nash equilibrium. The idea is that, if one contractor is diligent and the other is lazy, the diligent one who helped the boss catch the lazy contractor is rewarded an extra bounty b. Essentially, the cell at

the intersection of the *Lazy* row and the *Diligent* column of the table will be updated with a $+b$. Realize that this bounty must ensure that $r - \mathsf{cost}(1) + b > r - \mathsf{cost}(q) \Rightarrow b > \mathsf{cost}(1) - \mathsf{cost}(q)$ so that both players being lazy is no longer an equilibrium. Since $\mathsf{cost}(q) \geq 0$ we need $b > \mathsf{cost}(1)$. Unfortunately, they only consider *two* contractors being employed, and they do not consider the burden to the boss who needs to pay this extra reward (bounty).

3.2 Multi Contractor Case

We extend this 2-player game to m contractors, and precisely specify all the details. In particular, we lift the burden on the boss in terms of paying bounty. Our solution requires that the lazy contractors pay some extra fine, which will be distributed back to the diligent contractors by the boss. The total amount of the bounty given and the extra fine received will be equal. Even though the main goal here is to lift the burden on the boss, as we will see in Section 4, this technique also helps us deal with the malicious contractors. We now define our m-contractor game.

Remember that $\mathsf{u}_k(q)$ denotes the utility function of a contractor when there are $k \in \{0, 1, \ldots, m-1\}$ lazy contractors out of the other $m-1$ contractors in the same group (who are assigned the same job) and when this contractor uses a q-algorithm. When all the other $m-1$ contractors are lazy, the diligent contractor will obtain the following utility: $\mathsf{u}_{m-1}(1) = r + b(1-q) - \mathsf{cost}(1)$. This is because by returning the correct answer, she will get rewarded, and if the lazy contractors return a wrong answer (which happens with probability $1-q$, since we are assuming that the lazy contractors are all using the same deterministic q-algorithm)[7], she will get an extra bounty as well. In any case, by performing the job diligently, she pays the original algorithm's cost. At the other end of the spectrum, when all the other $m-1$ contractors are diligent, the lazy contractor will obtain the following utility: $\mathsf{u}_0(q) = rq - (f + b(m-1))(1-q) - \mathsf{cost}(q)$. The reason is that, when the q-algorithm used by the lazy contractor returns the correct answer (which happens with probability q), then she will obtain the reward r. But if she returns an incorrect answer (with probability $1-q$), the diligent contractors will ensure that the boss catches her, and thus she will need to pay both the fine and the bounty that will be given to the $m-1$ diligent contractors. In any case, she incurs the cost of the q-algorithm. Table 2 shows the m-contractor game.

Others $\setminus$ This	Diligent	Lazy
All Diligent	$\mathsf{u}_0(1) = r - \mathsf{cost}(1)$	$\mathsf{u}_0(q) = rq - (f + b(m-1))(1-q) - \mathsf{cost}(q)$
k Lazy	$\mathsf{u}_k(1) = r + b(1-q) - \mathsf{cost}(1)$	$\mathsf{u}_k(q) = rq - (f + \frac{b(m-k-1)}{k+1})(1-q) - \mathsf{cost}(q)$
All Lazy	$\mathsf{u}_{m-1}(1) = r + b(1-q) - \mathsf{cost}(1)$	$\mathsf{u}_{m-1}(q) = r - \mathsf{cost}(q)$

Table 2: Utility of a contractor in an m-contractor game, based on the actions of the other contractors.

[7]Remember that this is the worst-case for the boss since the probability of accepting incorrect answers is much higher. If the lazy contractors were all using independent q-algorithms, then this probability would have been negligibly smaller (due to the inner-state hash), and we would not need to analyze such an attack. More discussion and examples of real cases of using the same q-algorithm can be found in Section 4.1.

Theorem 3.1. *If the boss sets the bounty as $b > r/(1-q)$, then all diligent is a unique and strict Nash equilibrium in the game defined in Table 2.*

Proof. First, realize that if we generalize the 2-contractor game in Table 1 without any bounty, we still have two Nash equilibria. The goal is to break the all-lazy equilibrium and make the all-diligent equilibrium strict, so that no contractor will be better off being lazy. Hence, we have to start by breaking the all-lazy equilibrium. To do this, it must be the case that even if all other contractors are lazy ($k = m-1$), a contractor has to be strictly better off acting honestly:

$$\mathsf{u}_{m-1}(1) > \mathsf{u}_{m-1}(q) \Rightarrow r + b(1-q) - \mathsf{cost}(1) > r - \mathsf{cost}(q) \Rightarrow b > \frac{\mathsf{cost}(1) - \mathsf{cost}(q)}{1-q}$$

Since $\mathsf{cost}(1) \leq r$ (i.e., it is worth performing the task) and $\mathsf{cost}(q) \geq 0$ (i.e., no negative cost), then we have $\mathsf{cost}(1) - \mathsf{cost}(q) \leq r$. If the boss sets the bounty as

$$\frac{\mathsf{cost}(1) - \mathsf{cost}(q)}{1-q} \leq \frac{r}{1-q} < b,$$

then the all-lazy equilibrium will no longer be an equilibrium.

Second, we have to make sure that the all-diligent equilibrium is strict. Therefore, it must be the case that when all the other contractors are diligent ($k = 0$), it must be *strictly* better to be diligent as well:

$$\mathsf{u}_0(1) > \mathsf{u}_0(q) \Rightarrow r - \mathsf{cost}(1) > rq - (f + b(m-1))(1-q) - \mathsf{cost}(q)$$

This is also achieved by setting the bounty as before (since $m \geq 2$):

$$\frac{rq - r + \mathsf{cost}(1) - \mathsf{cost}(q)}{1-q} \leq \frac{rq}{1-q} = \frac{r}{1-q} - r < b - r \leq b + f \leq b(m-1) + f \qquad (1)$$

since the reward, fine, and bounty are all non-negative.

Lastly, when $0 < k < m-1$ of the contractors are lazy, it must be *strictly* better to be diligent as well:

$$\mathsf{u}_k(1) > \mathsf{u}_k(q) \Rightarrow r + b(1-q) - \mathsf{cost}(1) > rq - (f + b(m-k-1)/(k+1))(1-q) - \mathsf{cost}(q)$$

Using Equation 1 above, for all $k \in \{1, 2, \ldots, m-2\}$ we again have

$$\frac{rq - r + \mathsf{cost}(1) - \mathsf{cost}(q)}{1-q} < b - r \leq b + f \leq b + f + b(m-k-1)/(k+1)$$

□

Corollary 3.0.1. *Belenkiy et al. [4] show how to set q arbitrarily close to 0 by employing hash functions, based on their* unique inner state assumption. *Thus, it is sufficient for the boss to set $b \approx r$ for practical purposes.*

Note that we naturally obtain the same limit to bounty as Belenkiy et al. [4], since their game is a special case of our game with $m = 2$. Yet, observe that this bounty is no longer a cost to the boss, since it is taken as an extra fine from the lazy contractors. Next, we see how this works in presence of malicious contractors.

4 Malicious Contractors

Up till this section, we presented a solution to the outsourced computation problem with multiple contractors, assuming all the contractors are rational. However, as we first described our model, in reality there are honest and malicious entities in addition to the rational ones. Honest contractors are always welcome to an outsourced computation system. Thus, our goal is to limit possible damage that malicious contractors can incur.

There are two types of damage the malicious contractors can inflict on our system. In the first type of attack, the malicious contractors' goal is to make the boss accept an incorrect answer. In the second type, their goal is to make the boss perform extra work. These two attacks were analyzed by Belenkiy et al. [4] for a *no-bounty* scenario, and *without* taking into account all four different types of malicious contractors below. We now perform a detailed analysis for our m-contractor bounty-based game defined above, when malicious contractors are present.

Remember that in our setting, the boss accepts an answer, only if all the m returned answers match, including the hashes of the inner states. If all the answers match, there is no bounty to be given (since no contractor catches a lazy contractor). Therefore, this corresponds exactly to the case of Belenkiy et al. [4], and the related result is included in the Appendix for the sake of completeness of the results in this paper.

In terms of causing the boss perform extra work, the malicious contractors will aim to create a discrepancy among the returned results. In general, it may be enough to insert just one different answer to force the boss re-outsource the same job. Once we define the capabilities of different types of malicious contractors, we can analyze both results formally, and thereafter present the results from the real runs of the system.

4.1 Types of Malicious Contractors

We categorize malicious contractors into 4 types based on their capabilities. We also relate each type to real attacks.

Fully-Independent Malicious Contractors These contractors may act irrationally, but they do *not collude* at all. Each such malicious contractor acts completely independently. Remember that the boss selects a random group of $m \geq 2$ contractors, and outsources the same job to this group. The probability that fully-independent malicious contractors will return the same answer without computing the original algorithm is negligible, due to the *unique inner state assumption* of Belenkiy et al. [4].

Semi-Independent Malicious Contractors These contractors may provide the *same* wrong result. This may be achieved by the contractors downloading a fake client on purpose or accidentally (believing that it is the correct client software e.g., through phishing or DNS-spoofing). Such attacks are already known to exist in outsourcing systems[8,9,10]. This fake client may give the same incorrect answer to

[8] `http://tinyurl.com/truxoft`

[9] `http://home.hccnet.nl/adas/pfp-m20010130a.html`

[10] `http://tinyurl.com/rosettaextracredit`

all the contractors who downloaded it. Hence, we still consider these contractors kind of independent, but the results they provide are indeed dependent.

If there is at least one honest or rational contractor aside from these semi-independent malicious contractors in the group that is assigned the same job, then the cheating ones will be caught (since rational contractors will act diligently according to Theorem 3.1). Even though this is a very realistic type of attack, we will not delve deeper since its effect on the system is very limited (indeed, their effect is exponentially small in m). (See Appendix.)

Semi-Colluding Malicious Contractors There are multiple ways for malicious contractors to collude. For example, there may be a bulletin board type of web site where the one who found the result first can post it and the other contractors get it for free (their cost would be almost 0). However, if the results returned are correct, there is no problem to the boss. The only advantage to the these colluding set of malicious contractors would be that their costs are less and some of them will get paid without doing the job. *Yet, the boss is already willing to pay that money for obtaining the correct answer to the task.*

As another alternative, there may be a more advanced fake client that learns and communicates how many contractors are being employed for that assignment (i.e., m) and how many of these contractors are using this client (i.e., k). This way, it can act more intelligently, and return the same wrong answer only if all the contractors being employed are using this client (i.e., $k = m$). Even though the boss selects the contractors randomly, they can collude without knowing each other with this fake client set-up. Furthermore, the semi-colluding malicious contractors can collude in a way that they give inconsistent answers, so that the boss will need to re-outsource the job. To achieve this attack, the fake client can return the correct answer to some of the contractors and a wrong answer to some others. However, the colluders considered in this scenario are *independent in terms of their budget.*

Fully-Colluding Malicious Contractors In this type of attack, we assume that the malicious contractors can jointly decide on their outputs, as well as *share their budgets.* One possible realization will be through the most advanced version of the fake client, which controls the contractors' output and budget. Another possible realization would be through the Sybil attack [21], where one party impersonates many to make sure the random selection of the contractors would result in some of his identities being selected with some good probability.

Realize that this type of malicious contractors are the *hardest to defend against.* In the next section, we prove that *our system is resilient to even this type of contractors.*

4.2 Attacks of Fully-Colluding Malicious Contractors

Since fully-colluding malicious contractors are able to incur the most damage to the system, in this section we present provable bounds on this worst-case damage that can be caused in our system. Suppose the boss assigns the job to m contractors, where

k of them happen to be fully-colluding malicious contractors, where a g fraction of all contractors in the system are fully-colluding. A crucial observation is that, to be able to be employed by the boss, each contractor must keep a balance that is enough to pay the fine whenever necessary. Therefore, even malicious contractors must act diligently from time to time to ensure that they have enough balance to be employed. If a contractor does not have enough money to pay the fine, the boss will not hire him. Since colluders are trying to make the boss perform extra work by forcing him to re-outsource the job by submitting inconsistent answers, intuitively only one of them will submit a wrong answer, while the other colluders will still submit the correct answer. This way, they are trying to compensate for the fine that one colluder needs to pay. Remember, that they share the budget. The next theorem proves that this is indeed their best attack strategy.

Theorem 4.1. *For a group of k colluders to make the boss perform the most amount of extra work, their best strategy (the strategy with the highest combined utility) is that one colluder will submit a wrong answer (and get fined) and the rest ($k-1$ of them) will submit the right answer, and collect the reward and the bounty.*

Proof. Suppose the number of malicious contractors who act diligently is n with $0 \leq n \leq k$. Since their goal is to make the boss to do extra work, there has to be at least one non-matching answer ($n \leq k-1$).[11,12] The n diligent colluders will receive $r+b$ each as the reward plus the bounty, and the remaining $k-n$ contractors will have to pay the fine f each, and the extra fine $(m-k+n)b$ in total (i.e., there will be $m-k+n$ correct results since $m-k$ non-malicious contractors will act diligently, and n malicious contractors will do the same as well). The total utility (the payment received minus the total fine paid, also considering the costs incurred) of the group of malicious contractors will be:

$$\begin{aligned}\mathsf{U}(k,m) &= n[r+b(1-q)-\mathsf{cost}(1)] \\ &\quad +(k-n)[rq-(f+\frac{b(m-k+n)}{k-n})(1-q)-\mathsf{cost}(q)]\end{aligned}$$

Since the colluders want to trick the boss to do extra work, they must return at least one wrong answer with probability 1 (therefore $q=0$). Moreover, we can assume the worst case that $\mathsf{cost}(0)=0$ because the contractor can simply return a random answer. Therefore,

$$\begin{aligned}\mathsf{U}(k,m) &= n(r+b-\mathsf{cost}(1))-f(k-n)-b(m-k+n)) \\ &= n(r+f-\mathsf{cost}(1))-kf-b(m-k)\end{aligned}$$

In this utility function, the only part that is related to the number n is $(r+f-\mathsf{cost}(1))$. Since $r \geq \mathsf{cost}(1)$ and $f \geq 0$, we have $(r+f-\mathsf{cost}(1)) \geq 0$. Thus, whatever the values of r, f, b, m and k are; if n is higher, the colluders will be better off. Therefore, *the best strategy* for them is to set $n=k-1$. Hence one colluder will submit a wrong answer and the rest will submit the right answer. □

[11] If all of the contractors are malicious and colluding ($k=m$) and all of them are cheating ($n=0$), the boss cannot detect cheating since all the answers are the same wrong answers. In terms of making the boss accept an incorrect answer, please see Theorem A.1 in the Appendix.

[12] If the goal of the colluders is to make the boss to perform extra work, they should set $1 \leq n$. When $k \neq m$, they can set $n=0$ as well since this time there are other rational contractors who will act honestly. But the lower bound on n does not affect the proof of this theorem.

Now that we know the best strategy that will be employed by the fully-colluding malicious contractors, we bound the possible damage they can incur on the system.

Theorem 4.2. *If the fraction of fully-colluding malicious contractors in the system is g, and the boss outsources the job to a group of m contractors, then the fraction of jobs fully-colluding malicious contractors can cause extra work for boss is at most $rgm/(r+f)$, where r is the reward and f is the fine.*

Before we prove Theorem 4.2, we need two intermediate results. 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. Finally, let $B = \sum_{k=1}^{m} P(k,m)k$. According to Lemma A.1 in the Appendix, we have $A = 1-(1-g)^m$ and $B = gm$ [4]. We also have the following lemma:

Lemma 4.1. *$Am - B \geq 0$.*

Proof.

$$\begin{aligned} 0 \leq g \leq 1 \quad &\Rightarrow \quad 0 \leq 1-g \leq 1 \Rightarrow (1-g)^{m-1} \leq 1 \Rightarrow (1-g)^m \leq 1-g \\ &\Rightarrow \quad g \leq 1-(1-g)^m \Rightarrow gm \leq (1-(1-g)^m)m \\ &\Rightarrow \quad B \leq Am \Rightarrow Am - B \geq 0 \end{aligned}$$

□

of Theorem 4.2. Whatever the contractors do, they must keep their balance non-negative.[13] Since fully-colluding malicious contractors share their budget, it is necessary and sufficient for them to keep their total utility non-negative. Let y be the probability that colluders apply the best strategy presented in Theorem 4.1. Then, $x = 1-y$ is the probability that all colluders return the correct answer (when they are forced to do so to keep their shared budget non-negative). The boss wants to limit y and wants x to be high.

Let k be the number of colluders out of m contractors employed for the same task. Realize that all $m-k$ contractors will act diligently since they are either honest or rational who are incentivized to act diligently. When the colluders all act honestly (with probability x), they all get the reward r. (Since no one has cheated, no bounty is given.) When they apply their best strategy (with probability y), each of the $k-1$ diligent ones will get the bounty b and the reward r, and one of them will pay the fine f as well as the extra fine $b(m-1)$, which is equal to the total bounty given, including the bounty given to the other (honest/rational) contractors. Thus, the total utility of the colluders will be $\mathsf{U}(k,m) = xkr + y((k-1)(b+r) - (f+b(m-1))$. We now take the expectation over

[13] Indeed, they need to keep a balance at least as much as the fine to be employed at all, but for the sake of a worst-case proof, since the fine is non-negative, they must keep a non-negative budget.

different values of $1 \leq k \leq m$.[14]

$$\begin{aligned}
\sum_{k=1}^{m} P(k,m)\mathsf{U}(k,m) &= \sum_{k=1}^{m} P(k,m)[xkr + y((k-1)(b+r) - (f+b(m-1))] \\
&= \sum_{k=1}^{m} P(k,m)[xkr + ykr - yr - yf - yb(m-k))] \\
&= xrB + yrB - yrA - yfA - yb(mA - B) \\
&= rB - yrA - yfA - yb(mA - B)
\end{aligned}$$

At the last two steps, we substituted the values for A, B and used the fact that $x+y=1$. Colluders must keep this utility non-negative. Moreover, it is best for them to maximize y. Then, we can write

$$y \leq \frac{rB}{rA + fA + b(mA - B)} \leq \frac{rB}{rA + fA} = \frac{rgm}{A(r+f)}$$

using basic algebraic operations and Lemma 4.1.

If the colluding group cheats at y fraction of the jobs they were assigned, they can trick the boss to do extra work. The probability that there is at least one colluder among the employed group of contractors is A. Therefore, the fraction of the jobs where the colluders can force the boss to do extra work is Ay which is at most $rgm/(r+f)$. □

It is interesting to note that the result of this theorem also includes the result that Belenkiy et al. proved for the case where there is no bounty in the system. If we plug their way of setting the fine as $f = \frac{rd}{p}$, where they use p as the probability of catching a cheating contractor and d is some deterrent factor, then $rgm/(r+f)$ becomes $pgm/(p+d)$ ([4] Theorem 9). Therefore, we essentially proved that the bounty system, while incentivizing all rational contractors to act diligently, does not introduce any weaknesses against malicious contractors to the system.

Finally, we note that the payment of the bounty by the lazy contractors have a two-fold effect. First, it **eases the load of the boss** since he does not need to pay extra for the job. Second, it **keeps the damage by the malicious contractors low**.

4.3 Expected Completion Time of a Job

Remember that we are considering the boss's strategy to accept the returned result when all m contractors return a matching output. When there is a controversy in the answers, the boss must differentiate which answers are correct and which are not, so that he can correctly distribute the reward and the fine, as well as the bounty.

The first method is that the boss verifies the answers himself. This can be done in two ways: by running the original algorithm for the job himself, or using some other verification algorithm. In some applications, the verification can be very simple. For example, in the integer factorization problem, the computation of the factors is hard, but

[14] Remember that all our bounds are for the worst cases. Even when $k = m$ it is presumable that the attackers try to force the boss to perform extra work instead of making him accept an incorrect answer, by using their best strategy.

their verification is relatively simple. In general, for all problems that are believed to be in $NP - P$, there is no currently-known solution in P, and verification (in P) is simpler than computing the answer. Even if the job itself is in P, verification may be easier (e.g., consider a matrix multiplication job with complexity $O(n^3)$, for which there exists a (probabilistic) verification algorithm that is $O(n^2)$ [8]). However in some situations, the verification can be as hard as the job itself, or the boss may even need to re-compute the job himself to verify the outputs. Even if the fraction of verifications is low, this may greatly reduce the benefits of outsourcing. Besides, in some situations, the boss may not have enough power to verify the answers.

The second method is that the boss again employs contractors to verify the answers provided by the other contractors. This can be done by re-outsourcing the job to a new group of (m random) contractors, until all the answers in the same group are the same. This method is the method we suggest and analyze below. The drawback of this method is that it consumes even more resources of the system as a whole; but the key point here is that those resources do *not* belong to the boss. Theorem 4.3 investigates the amount of the extra resource consumption.

Theorem 4.3. *If the fraction of times that there is a dispute among the returned results is E (i.e., from Theorem 4.2), then the expected number of times a job is outsourced is $1/(1-E)$ (including the first outsourcing), assuming the boss keeps outsourcing until there is no dispute.*

Proof. Obviously, we outsource a job at least once. If there is a mismatch in the answers (which happens with probability E) then we re-outsource it. If there is a problem again (now with probability E^2), we outsource the same job again etc. The expected number of times a job is outsourced is (with $0 \leq E < 1$):

$$1 + E + E^2 + \cdots = \frac{1}{1-E}$$

□

Corollary 4.1.1. *The expected total number of the contractors employed for one job is $m/(1-E)$.*

Corollary 4.1.2. *If we assume that a job can normally be completed in time t, then the expected total time required to complete that job will be on the order of $t/(1-E)$.*[15]

Corollary 4.1.3. *Note that if the boss were to compute the job herself, she would have also spent at least t time to compute the job. Thus, the overhead of outsourcing in our system is around $E/(1-E)$ fraction of the local computation time.*[15]

[15]It is assumed that the time passed during the network communication is very small compared to the time required for computing the job, and the contractors always start the job immediately and send the results back once the algorithm is finished. Even though these assumptions may make it sound like in reality the actual time would be larger, it is possible that the contractors have faster machines and thus the total time in reality may even be smaller than the time that the boss would have needed to compute locally. Furthermore, as our performance results show, the slowdown due to the inner state computation is very small as well.

Our game-theoretic analysis ends here. We will provide evaluation results in Section 6. But before that, we provide a mechanism that ensures the payments of the fines and rewards are done fairly, even though both the boss and the contractors may try to cheat. The reader may choose to skip Section 5 for the first reading.

5 Ensuring Fair Payments

In the Belenkiy et al. [4] scenario, the contractors are untrusted, but the boss also acts as the bank, holding the balances of the contractors, and has the ability to fine and reward them at will. Unfortunately, this is not a practical setting, since it assumes the boss is ultimately trusted by all contractors. Instead, in practice, we have banks who are trusted to handle monetary transactions, and the boss may cheat since he may not want to pay the reward that the diligent contractors deserved. The contractors, as before, may try to cheat so as not to pay the fine. In this section, we show how to modify the payment system such that even when the boss and the contractors are all untrusted, the payments are fair as long as we have a trusted bank.

The techniques we employ in this section include electronic checks [17] or electronic cash [16], and fair exchange [2, 41]. Depending on the constructions, there may be specific choices of the underlying primitives to be employed for the sake of efficiency. For example, Küpçü and Lysyanskaya [41] recommend using Endorsed E-cash [13] with Camenisch-Shoup verifiable encryption [14] to obtain an efficient and anonymous protocol. On the other hand, they recommend using simple electronic checks if anonymity is not desired. Note that fair exchange requires a trusted third party [46, 37], called the arbiter [41, 39]. Since we already have a trusted bank in the system, and since fairness here is for the sake of the payment of the rewards and fines, we will employ the bank as the arbiter as well.

Our goal includes the fair processing of the following payments:

- Payment of the reward by the boss to the diligent contractors.
- Payment of the fine by the cheating contractors to the boss.

Note that even though we have presented a bounty setting, for the sake of simple presentation, we focus on the payments of the rewards and fines. Payment of the bounty can be handled via similar techniques, since it requires both the payments by the cheating contractors to the boss (like fines), and the payments by the boss to the diligent contractors (like rewards).

One key issue any payment system for outsourced computation needs to address is the definition of "*correct computation*", or equivalently, the "*diligent contractor*". The naive way would be to define the correct computation using the *desired output*. But the whole point of outsourcing computation is that the boss does *not* know the output ahead of the time. Yet, in our setting, there exists a very natural and useful definition. We define a **"*diligent contractor*" to be the one who returns the answer that agrees with all the m answers**, where m is the cardinality of the set of contractors employed for the same job. In case of mismatch between the m answers, the identification of the diligent and cheating contractors would be delayed, until further outsourcings of the same job are finished. Once an outsourcing of the same job returns m matching answers (call that answer *correct*), then the contractors who returned the same answer *correct* in any

previous outsourcing of the same job will be considered *diligent*. All other contractors are, by definition, cheating.

Observe that this definition of *diligent* allows colluding malicious contractors to be seen as diligent, if the whole group for the same outsourcing of a job consists of colluding malicious contractors. Yet, under the assumption that the boss accepts matching answers, they deserve getting paid the reward. Therefore, this does not constitute a problem of our system.

Any boss who wants to oursource a task to m contractors must have enough balance to pay m rewards (i.e., balance $\geq mr$). Similarly, any contractor who wants to be employed must have enough balance to pay the fine (i.e., balance $\geq f$). Yet, just obtaining a signed statement from the bank attesting to this is not enough, since the contractor may try to present such a signature to multiple bosses or for multiple jobs, even when she has balance enough to pay only one fine. The solution we propose involves escrowing the necessary amount to the trusted bank.

An escrow is encryption of some value under the public key of the bank. Since our context is payments, the escrow would contain some form of a payment (electronic checks [17] or electronic cash [16]). A *verifiable* escrow of a payment is essentially an encryption of the payment under the bank's public key, while the receiver of the verifiable escrow can verify (without decrypting) that it indeed contains the necessary payment. For example, as suggested by Küpçü and Lysyanskaya [41], if a Camenisch-Shoup verifiable escrow [14] contains an Endorsed E-cash [13] e-coin, then the recipient can verify that the e-coin is valid (including the amount that it is worth), without actually obtaining the payment. A verifiable escrow also contains a public non-malleable *contract*, which describes the conditions that the bank would decrypt it and give the payment to the requester.

It is important to note that the rewards and fines are one-use payments[16], and thus if a contractor already had the reward, or if the boss already obtained some fine, they gain no additional payment by trying to use it again.

Realize that if the boss did not send the reward, then we need a mechanism for the diligent contractors to obtain the reward (via the bank). Similarly, observe that a malicious contractor may choose not to send a response (or not to sign it, see below). As far as the boss is concerned, this is equivalent to sending a wrong result. For these purposes, we introduce *job timeouts*, *fine timeouts*, and *reward timeouts* to the system.[17,18] Moreover, to ensure that the contractors indeed performed the computation within the required time, we employ a *trusted* timestamping/notary server [12]. Assume that the boss is registered to the timestamping server so that the server knows him. Each time a contractor asks for a timestamp on some value, the server returns the stamp to the contractor, and forwards the value together with the stamp to the boss.

The **notation** we employ in the protocol is as follows: $VE(item; contract)$ specifies a verifiable escrow encrypting some *item* and labeled with a *contract*. The reward is denoted r, and the fine is denoted f, as usual. *jid* denotes the job identifier, *out* is the

[16]This is already simply done using serial numbers for both electronic checks and electronic cash.

[17]The standard way of introducing a timeout in a fair exchange is adding it to the contract of verifiable escrows [40].

[18]Obviously, the fine timeout –the deadline to send the verifiable escrow of the fine, signaling acceptance of the job– needs to be earlier than the job timeout, and the job timeout –the deadline to complete the job and return the response– needs to be earlier than the reward timeout –the deadline for the boss to send the reward or the wait signal to the contractor–.

output of the computation (including the inner state hash value). $sign_{sk}(msg)$ denotes a signature on the message msg using the secret signing key sk. The signature can be verified using the corresponding public verification key pk. Each contractor signs the output out and the job identifier jid. The timestamp t is created on these values and their signature s using the time of the trusted timestamping server. Similarly, the timestamp tf is the stamp over the verifiable escrow of the fine that the contractor sends.

5.1 Fair Payment Protocol

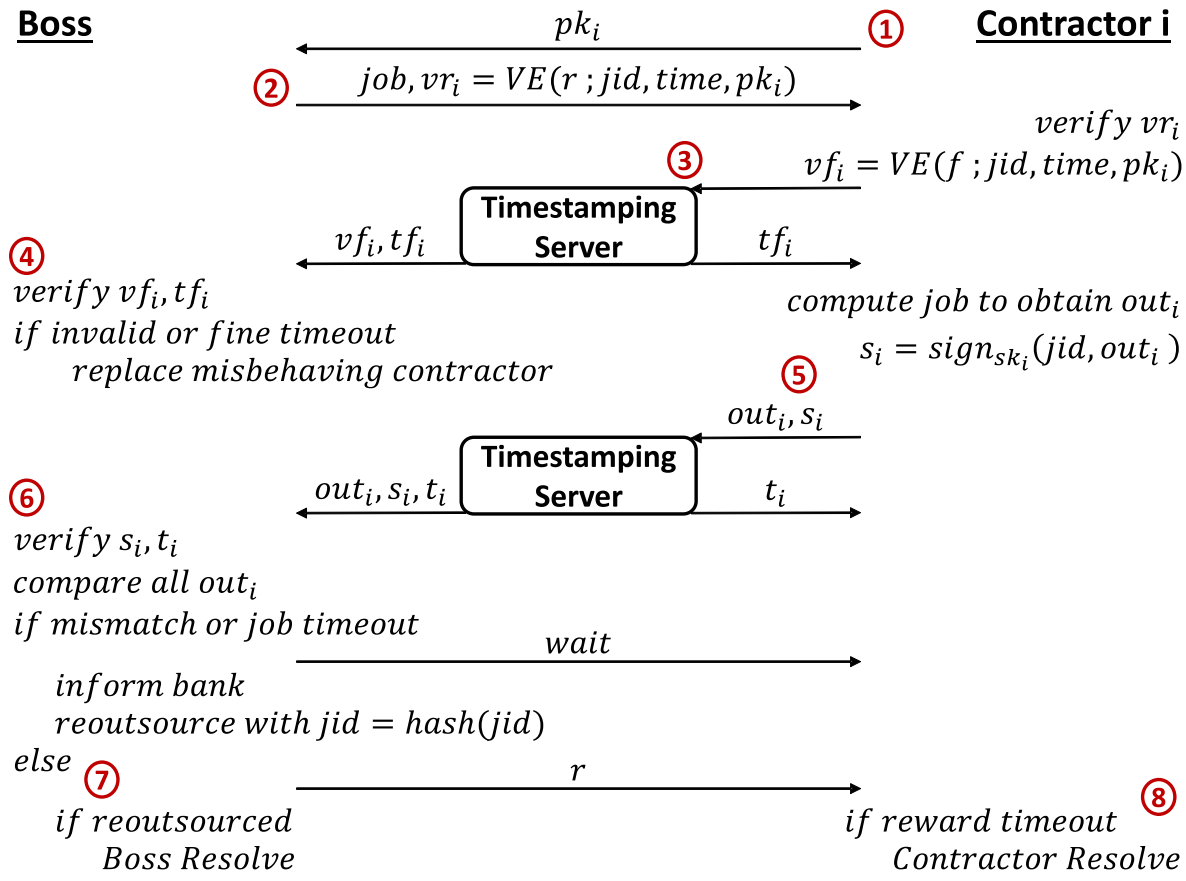


Figure 1: Our fair payment protocol overview.

The protocol summarized in Figure 1 works as follows (please employ the circled numbers to match the text to the figure). ① At the beginning of the protocol, each contractor i generates her signature public-secret key pair pk_i, sk_i, and sends the public verification key pk_i to the boss. This public key will be used to tie the contractors' answers to their escrowed fines and rewards. This way, the boss will be able to prove which contractors did indeed cheat, or the contractor will be able to prove that she did not cheat. If a public key infrastructure exists, then it can be employed instead (then we do not need this first step, and we do not need to put the signature public keys into the escrow contracts), but such an infrastructure is *not* necessary. Note that all public signature verification keys need to be *distinct*. Otherwise, the boss should not outsource

to those contractors.[19] Realize that this may also be thought as a one-time registration with the boss.

② Next, for the payment of the rewards, the boss creates m separate *verifiable* escrows vr_i each worth r, and sends one to each employed contractor. Each verifiable escrow will contain a unique job identifier jid in its contract, which is computed as the hash of the job (which is the prescribed program together with its input). The contract also indicates the fine timeout, the job timeout, and the reward timeout (all abbreviated in the figure notation as *time*). Furthermore, the contract of each vr_i indicates the signature public key pk_i of the contractor i. The boss also sends the job, but the contractors do not start computing it yet. Each contractor i will verify the escrow vr_i she receives (including the contract[20]), and stops if something is wrong.

③ Now that the contractors each have a verifiable escrow from the boss, they each create a verifiable escrow vf_i worth f. These escrows also contain the same contract. Each contractor sends her vf_i to the timestamping server who does two things: 1) creates a timestamp tf_i on it and sends this timestamp back to the contractor, and 2) forwards the verifiable escrow vf_i together with the timestamp tf_i to the boss.

④ Just as the contractors verified the boss's escrows, the boss also verifies each such escrow it receives. If something is wrong (some verifiable escrows did not verify or the boss did not receive some with valid timestamps before the fine timeout), the boss replaces the problematic contractor. To replace a contractor, the boss sends the job and verifiable escrow vr_j of the reward to some new contractor j (with updated fine timeout). If that new contractor sends back the verifiable escrow vf_j of the fine before her fine timeout, then the group is complete, and the boss starts waiting for the job results.

⑤ Once the contractor i sends her verifiable escrow vf_i in time and obtains back the timestamp tf_i, she starts computing the job (i.e., she does *not* need to wait until the fine timeout). At the end, she obtains the output out_i (including the inner state hash value), and then she creates a signature s_i on it (along with the job identifier) using the secret key sk_i matching the public key pk_i in the contract of her verifiable escrow vf_i. She sends the output and the signature out_i, s_i to the timestamping server. The timestamping server generates a timestamp t_i and sends it to the contractor, while forwarding the out_i, s_i, t_i to the boss. This will be used to prove that the contractor indeed computed the job before the job deadline.

⑥ Now the boss verifies the signatures (and timestamps) and compares all the outputs (including the inner state hashes). If they all match, he sends the rewards to the contractors, and the protocol ends. If there is a mismatch (or some contractors did not return the result until the job timeout, or sent the result with an incorrect inner state hash, or sent the result with an invalid signature), the boss tells the current set of m contractors to wait, and re-outsources the job to another group of m contractors (until he receives all matching answers). The boss also contacts the bank to inform about the mismatch for that particular job id so that the bank delays the reward timeout for that particular job. Each re-outsourcing requires the same procedures, but the re-outsourced job identifier is computed as the hash of the current job identifier.

[19] The boss may also ask for a signature on a random dummy message to ensure that the contractor knows the corresponding secret signing key.

[20] She checks the contract to ensure that the public key is hers, the jid is indeed the hash of the job, and the timeout values are as expected.

⑦ Finally, the boss receives m matching answers from the same (re-)outsourcing. Let us call the final matching set of m answers *correct*. The boss sends the reward to all contractors who also returned the same answer *correct* in any one of the outsourcings of the same job. He then contacts the bank to obtain the fines, if re-outsourcing was necessary. The protocol between the bank and the boss is called *Boss Resolve*.

⑧ After sending the result and obtaining the timestamp, each contractor waits to hear from the boss until the reward timeout. If the boss asks the contractor to wait, she waits for one more reward timeout. If the boss did *not* ask the contractor to wait *and* did *not* send the reward before the reward timeout, the contractor contacts the bank. This operation is called *Contractor Resolve*.

Boss Resolve: Remember that this protocol is executed only if the boss needed to re-outsource the job and needs to obtain some fines (assuming the lazy or malicious contractors did not already willingly pay the fines). For the boss to obtain the fine from the bank, he needs to prove that the contractor sent an incorrect result or did not send a signed result in time. We need to employ the help of the timestamping server here.

To obtain the fine of a cheating contractor i, the boss presents the verifiable escrow vf_i of that contractor with job identifier jid, and the m verifiable escrows, results, and signatures of those who sent the answer *correct*. For simplicity of the presentation, assume that the second time the job is outsourced, all results matched. Therefore, all m *correct* signed results must belong to the job identifier $jid' = hash(jid)$, and the bank verifies this.[21] The bank also verifies that all m results contain the same *correct* answer, and have valid signatures when verified using the public keys in the corresponding verifiable escrows of fines.

It is safe to assume that a cheating contractor would not send a signed incorrect answer, and that the boss who wants to obtain the fine would not present values in favor of the contractor. Therefore, the bank employs the help of the timestamping server, and proceeds as follows:

1. The bank asks the timestamping server for *all* the out_j, s_j values for the job identifier jid. There are two options:

   (a) If none of the results returned by the timestamping server has a verifying signature under the public key pk_i in the verifiable escrow vf_i, meaning that the contractor i did not perform the job, the bank decrypts vf_i to obtain f and pays the boss the fine f.

   (b) If one of the returned results has a verifying signature s_i under the public key pk_i in the verifiable escrow vf_i, then the bank uses that out_i value. If the associated timestamp is valid but *late* or the output out_i is incorrect (i.e., different from the m *correct* outputs), the bank decrypts vf_i to obtain f and pays the boss the fine f.

[21] It is very easy to generalize this idea. For example, assume the job is outsourced three times, where in the first two there were mismatches. Note that if the first outsourcing had job identifier jid, then the second one would have $jid' = hash(jid)$, and the third one would have $jid'' = hash(jid')$. Also observe that this means in the first two outsourcings, there was at least one cheating contractor each time. Suppose contractor i cheated in the first outsourcing, and contractor j cheated in the second. Thus, the boss must provide the bank vf_i with job identifier jid, vf_j with job identifier jid', and m matching outputs all with job identifier jid''. Furthermore, the bank knows how many times a particular job is re-outsourced, since each before re-outsourcing the boss informs the bank.

Contractor Resolve: This protocol is executed if the contractor did not get the reward until the reward timeout for a job she completed. Note that the reward timeout resets after a wait signal from the boss, and hence by the reward timeout here we mean the latest version of it.

If contractor i contacts the bank to obtain the reward, she needs to present out_i, s_i, t_i (the output together with the inner state hash, the signature, and the timestamp) and the verifiable escrow vr_i of her reward. She also presents her verifiable escrow vf_i of the fine, together with the timestamp tf_i on it. The bank proceeds as follows:

1. If it is before the reward timeout, the bank aborts. Note that at each re-outsourcing, the boss contacts the bank so that the reward timeout is delayed for one more outsourcing. By the reward timeout here, we mean the last version of it (for the last re-outsourcing of the same job).

2. If the verifiable escrow vf_i fails to verify, or the contracts of vf_i and vr_i are different, or tf_i timestamp is later than the fine timeout in the verifiable escrow contracts, or timestamp t_i is later than the job timeout in the verifiable escrow contracts, or the signature s_i does not verify using the public key in the verifiable escrow contracts, then the bank aborts the process.

3. At this point, if the process is continuing, it is already after the reward timeout, and the bank knows that all the values (except out_i) provided by the contractor are valid. Now, there are two options:

   (a) If the boss *never* contacted the bank about the job identifier in vr_i until the reward timeout, this means that there were no problems, and thus the bank decrypts vr_i to obtain r, and rewards the contractor.

   (b) If the boss performed Boss Resolve with the bank, then the bank checks if the answer given by contractor i is the *correct* answer for that job identifier (or a hash-related job identifier). If so, the bank again rewards the contractor.

5.2 Fairness Analysis

First of all, observe that if all contractors performed the job diligently and the boss acted honestly as well, then the bank is not involved at all. This is akin to the **optimistic** behavior in fair exchange protocols [2] or official arbitration protocols [38]. The timestamping server is always involved, but it is a very simple server that signs the given value together with the current time without any check, sends the timestamp to the contractor, forwards the value together with the timestamp to the boss, and stores the value together with the timestamp in case the bank asks. We are assuming the boss registered with the timestamping server initially, so that the timestamping server can easily forward the associated values to him. The timestamping server storage also need *not* be indefinite, as we discuss in the next section. Moreover, we took great care in our protocol to make sure the timestamping server only needs to know the boss, but not the individual contractors. This is one of the key points of our solution, and makes the design much harder (as otherwise anything can go through the timestamping server).

When the job is outsourced, note that its identifier jid is computed as the hash of the job. This helps create unique identifiers and ties the job to the verifiable escrows. The job identifiers for re-outsourcings are related in a hash-chain manner, which can be easily verified by the bank. Furthermore, the outputs and their inner state hashes are also tied to the job id via the contractors' signatures. Moreover, the signature ties a particular output to the related verifiable escrow. Finally, the timestamp bundles the output (including the inner state hash) and the signature all together. It also allows us to verify against the job timeout using the time value in the verifiable escrows' contracts. The verifiable escrow timestamp signals acceptance of the job by a contractor, and is necessary both for the contractor to obtain the reward, and for the boss to obtain the fine.

If we look at the protocol step by step, we observe that for a contractor to be able to obtain the reward, she must have sent her verifiable escrow of the fine before the fine timeout. That escrow essentially signals that the contractor accepts the job. She must also return the correct result before the job timeout to be able to obtain the reward.

Furthermore, realize that during Contractor Resolve, the contractor provides the correct result, signed and timestamped before the job deadline. This is one of the two ways for the contractor to obtain the reward. The second way would be if the boss never informed the bank about a re-outsourcing of the job. In such a case, this implies that all the results matched, and the contractor deserves the reward as long as she accepted the job by sending her verifiable escrow of the fine before the fine timeout and returned the result correctly before the job deadline, which can be checked via the timestamps. Since obtaining the same reward multiple times (e.g., once from the boss, and once from the bank) is useless due to one-use payments, this does not constitute a fairness problem.

As for the boss to obtain fines via the Boss Resolve process, he needs to prove via help from the timestamping server that the contractor did not compute the correct output. If the timestamping server already timestamped that contractor's result such that the time is late or the output is incorrect while the signature verifies, then the bank pays the contractor's fine to the boss. Alternatively, if the contractor never even contacted the timestamping server with proper results and a valid signature, then again the bank pays the contractor's fine to the boss.

Thus, if the contractor was diligent, and performed the job correctly on time, then the timestamping server will have the matching result, and the contractor will not be fined during Boss Resolve, and will be guaranteed to obtain her reward during Contractor Resolve. If the contractor cheated, then the bank will be paying the contractor's fine to the boss during Boss Resolve.

There is an interesting attack the contractors may try to perform. Consider that the boss chose $m = 2$ contractors, and sent them vr_1 and vr_2 verifiable escrows of the reward. Then, suppose that he received back vf_1 but not vf_2. Note that if the boss did not receive vf_2 until the fine timeout with a valid timestamp, it means it does not exist since otherwise the timestamping server would have forwarded that to the boss. At this point, the boss needs to outsource to a third contractor, by sending vr_3 and receiving vf_3. But now assume that contractors 2 and 3 are working together, such that the contractor 3 sends the correct output (including the inner state hash) to contractor 2.[22]

[22]If contractors 2 and 3 did not collude, it is improbable for contractor 2 to send back a correct output with a correct inner state hash for a job that it does not know.

Their goal is to obtain rewards such that the boss ends up paying 3 rewards, even though he outsourced the job to 2 contractors in total. Yet, realize that for the contractors to obtain the reward, they must have contacted the timestamping server such that: 1) the verifiable escrow of the fine is timestamped before the fine timeout, and 2) the correct result is timestamped before the job timeout. Still, since the boss did not receive the vf_2 before the fine timeout, he can rest assured that the bank will not reward contractor 2.[23]

Finally, the boss may also try to trick the re-outsourcing system, as best described via an example: Consider $m = 5$ contractors are assigned to a task, where 3 of them returned the correct answer (which can only be realized to be correct after some re-outsourcing of the same job returns 5 matching answers) and 2 of them an incorrect answer (similarly realized later on). Then the boss re-outsourced, and now another 3 contractors returned the correct answer, and another 2 of them returned an incorrect answer. Finally, after the third time the boss re-outsourced the job, all 5 contractors returned the same answer (and at this point the boss knows who returned the correct answer in the previous outsourcings). Interestingly enough, just to pay fewer rewards, the boss may try to claim to the bank that this is indeed what happened: "When I first outsourced, I received 4 incorrect answers and 1 correct answer, and when I outsourced again, I received 5 correct answers". He will not lie about the number of incorrect answers since he wants to obtain the fines, but he may lie about the total number of correct answers. Luckily, this attack is impossible in our protocol due to the use of unique job ids, collision resistant hash functions, and the fact that before each re-outsourcing the boss must contact the bank and use the hash of the previous job id as the next job id. This way, the boss cannot lie about the job identifiers and the bank can easily check what actually happened.

As future work, we imagine a mechanism that can handle fair payments for a bundle of jobs, instead of a single job, increasing efficiency. It may be possible that a tree-based payment mechanism enables payment of n bundled jobs in $\log n$ steps, increasing the efficiency.

5.3 Setting the timeouts

In this section we enlighten the practical use of our protocol by discussing the timeouts. Once the job and the verifiable escrow of the reward is sent, the boss starts waiting until the fine timeout. In normal operation, such a wait should allow for the network delays, verification of the reward escrow, preparation of the time escrow, and timestamping. Overall, we imagine that several seconds should be a realistic value, and can be set empirically. All timeouts should be per-job timeouts (e.g., if the job is large in size, network delays may require the fine timeout to be large, and if the job is long in time, the associated job timeout and the reward timeout need to be large).

The job timeout depends on the hardness of the job. It may be from several minutes, to several hours, or even days. What is important is that, the reward timeout is set as the job timeout, plus several seconds to minutes for the boss to compare the results and send the wait signal and inform the bank.

[23] In a real implementation, to adjust for timing differences between the boss and the timestamping server, we can let the boss wait for a little more than the fine timeout to replace a contractor. Eventually, the bank will employ the timestamping server's time value.

Overall, we imagine that for most of the cases, the fine timeout will be very insignificant compared to the job and reward timeouts so that replacing a contractor does not necessitate changing the job and reward timeouts. But, if the job is short, then replacing a contractor may require setting different job and reward timeouts for that contractor. In such a case, if this later added contractor did not return the result until close to the reward timeout of the original group of contractors, then the boss should signal the original group to wait, and inform the bank, even though there is not necessarily a mismatch.

Lastly, we are assuming that the job is expected to be re-outsourced only so many times (e.g., 5 times). As we have proven game-theoretically and experimentally, the number of expected outsourcings of the same job is actually very close to 1. Thus, this would be a very conservative system-wide value. What this would enable is that no contractor waits indefinitely (e.g., the boss can send at most 4 wait signals to a contractor for a particular job), and the timestamping server does not need to keep an indefinitely-growing storage (e.g., it needs to store timestamps associated with a particular job only for 5 reward timeouts).

6 Performance Evaluation

We implemented an outsourcing system that includes the boss, honest contractors, and all four types of malicious contractors. Note that once the system parameters are set according to our theorems, all rational contractors will become honest contractors, and hence there is no need to implement them separately. We further implemented a very simple mechanism to modify the actual jobs to also return the hash of the inner state. Then, we evaluated our solution under various scenarios, and confirmed that our theoretical results hold. All source code is available on our group's web page [20].

6.1 Inner State Hash Performance

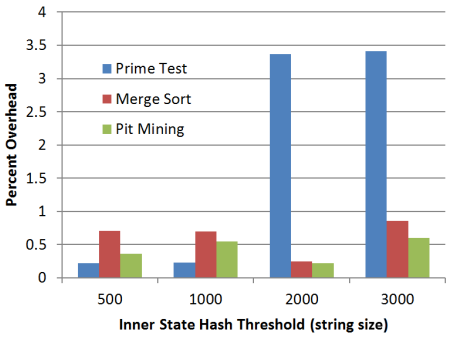


Figure 2: Percentage overhead of computing inner state hash that is necessary for the boss's comparisons.

Remember that our goal is to keep an inner state hash of an algorithm given a particular input. The main purpose of an inner state hash is to distinguish between jobs: if a different algorithm other than the original is employed, or a different input is used, then the inner state hashes should be different.

For implementing the inner state hash, we used the AspectJ Eclipse extension. We created an aspect that hashes all the inputs and return values of each method called in a Java code. Outsourcing in our system is very simple. Once a Java code containing the actual job is *compiled* with our AspectJ aspect, it is ready to be outsourced. (Similarly, an AspectC++ aspect can be used for C++ codes.) **No changes to the actual code is necessary.** At the end of its run, the program will output the hash of its inner state as well as its original output.

We tested the overhead of this inner state hash computation on a machine with 2.27 GHz CPU and 4 GB RAM, using SHA-1 as the hash function. If we follow the original proposal of Belenkiy et al. [4] and first compute the whole inner state with a final hash computation at the end, then the inner state would grow very big, consuming memory and slowing down the computation due to string concatenation operations. Therefore, we programmed our aspect to let the inner state grow up to a threshold, and each time the threshold length is reached, we hashed the inner state to shrink it back. We tested our setup with three computational tasks: a prime number test, merge sort algorithm, and a pit mining task.

Figure 2 shows the overhead of the inner state hash computation when compared to the original running time of these three algorithms. It shows that the overhead the inner state hash computation incurs is extremely small: around 0.2% for all tasks using tweaked thresholds. Therefore, outsourcing a task with this inner state hash comparison idea incurs virtually no cost over running just the task itself.

6.2 Fully Colluding Contractors

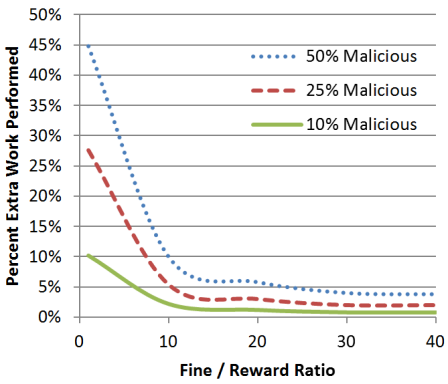


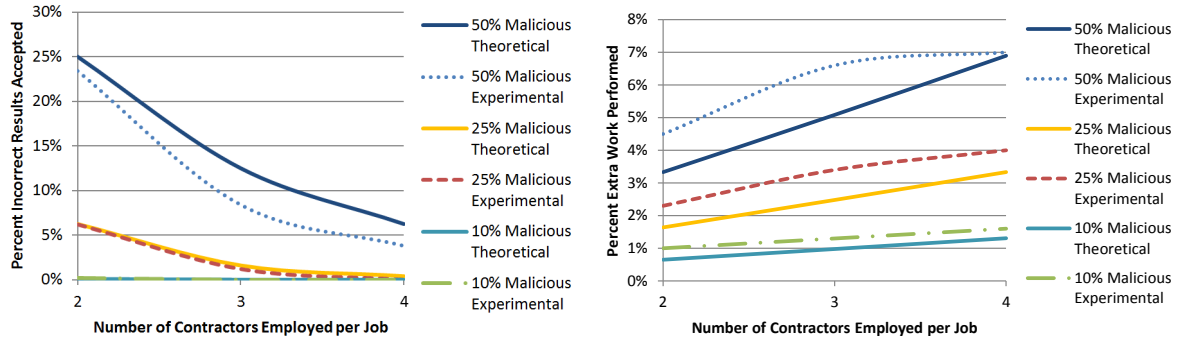
Figure 3: Extra work due to malicious contractors, based on experiments.

Parameters: Using our implementation of the boss, the honest contractors, and the four type of malicious contractors in Java, we tested various scenarios. We ran 1000 jobs for each scenario, where each job was outsourced to a group of 2 random contractors among a pool of 60 contractors. All malicious contractors were fully-colluding. All contractors were initiated with credits twice the value of the fines. Note that using the inner state hash, it does not matter which q-algorithm the lazy and malicious contractors use, since its output will be different from the original algorithm with high probability (because $q \approx 0$ including the fact that the inner state hash needs to be the correct as well).

Figure 3 shows the percentage of extra work that different fractions of fully-colluding malicious contractors may cause the boss to perform. As the fine to reward ratio increases, the extra work caused by the malicious contractors decreases quickly.

Consider the results of Theorem 4.3 and its corollaries in the light of Figure 3. In the theorem, the E value indeed corresponds to the values shown in the figure. Thus, the figure shows that the boss can set $E \leq 2\%$ (e.g., by setting fine to reward ratio as 20 even when a quarter of all contractors are malicious). Therefore, according to the theorem, a job is outsourced, in expectation, only 1.02 times. This corresponds to employing, on average, 2.04 contractors for each job. Including the 0.2% overhead of the inner state hash calculation, **the total overhead of our system per Corollary 4.1.3 is 2% in expectation**, even with the existence of a large percentage of malicious contractors in the system.

In comparison, consider Pantry verifiable computation scheme [10]. Pantry converts a DNA substring matching job that takes 0.2 seconds of local computation to 5.7 minutes



(a) Theory vs. Practice: Incorrect Results.

(b) Theory vs. Practice: Extra Work.

Figure 4: Comparison of theoretical and experimental results.

of outsourced computation, creating an overhead of 171000% (and 71560% overhead on average for the five different tasks they measured). Thus, for correctness of the answer, using our system results in more than **4 orders of magnitude better performance** than employing Pantry [10] verifiable computation. Note that most of the verifiable computation works mainly focus on the time spent by the verifier (i.e., the boss) [59]. While that time is small in those works and is extremely small in our work, what we focus on is the overhead of outsourcing the job versus performing it locally. The time required at the prover (i.e., the contractor) side in the verifiable computation works is as discussed above, and thus their performance degradation is extreme compared to our neglectable overhead. Even when we expect the contractor to be a powerful cloud server having much more resources than the boss, we do not expect it to have 71560% more resources.

Figure 4 compares the theoretical results with the experimental results, for a fixed fine to reward ratio of 30. Figure 4(a) shows the percentage of incorrect results accepted by the boss, as a result of all the contractors that are assigned to the same job being colluders. The figure proves that the theoretical results of Theorem A.1 closely match the reality. It also shows that as the group size, meaning the number of contractors employed for the same job, increases, the boss accepts almost no incorrect results.

Interestingly, while the increase in the group size diminishes the possibility of accepting incorrect results, it may increase the percentage of extra work required, as Figure 4(b) shows. The reason is that a larger group has a higher probability of containing at least one malicious contractor. The graph confirms the results of Theorem 4.2 and Corollary 4.1.3. Note that the main reason that the experimental results are slightly above the theoretical ones in this figure is due to the fact that we provide starting balance equal to *twice* the fine in our experiments. Overall, from Figure 4, we see that the practice closely follows the theoretical results, and verifies their validity.

6.3 Experiments with Mixed Fraction of Malicious Contractors

In this section, we use our implementation to present results involving a mix of various types of malicious contractors. To present a realistic, but still *overly cautious* scenario, we used figures from Symantec [54] and Panda Labs [47]. Panda Labs report says worldwide infection rate is around 32%. Thus, we take this as the total fraction of malicious users

in our system. Symantec ranks the top 10 malware, where the top malware caused 6.9% of all infections. Let us assume this top malware corresponds to the most destructive scenario, and all of its users are fully-colluding malicious contractors. Thus, in our test, we let $32\% * 6.9\% = 2.2\%$ of all users to be fully-colluding. The next best malware caused 5.1% of infections. Let us take this as the semi-colluding malicious contractor fraction. Hence, overall we let $32\% * 5.1\% = 1.6\%$ of all users to be semi-colluding. Out of the remaining $32\% - 2.2\% - 1.6\% = 28.2\%$, we let half to be semi-independent and the other half to be fully-independent.

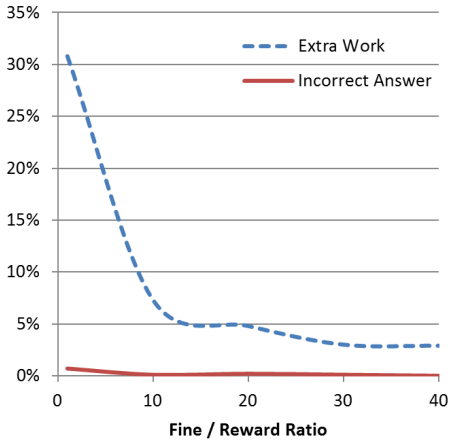


Figure 5: Extra work done and incorrect results accepted in mixed types of malicious contractor experiments.

Parameters: To simulate this scenario, we outsource a job to 100 contractors, 2 of which are fully-colluding, 2 of which are semi-colluding, 14 of which are semi-independent, and 14 of which are fully-independent. The remaining contractors act diligently, since they are incentivized to do so. We keep the group size as 2 to allow the possibility that colluding malicious contractors may make up the whole group, and outsource 1000 jobs. All contractors were initiated with credits twice the value of the fines.

Figure 5 shows the results of these experiments. With proper fine to reward ratio settings, **the boss obtains more than 99.9% correct results with only 3% extra work**. At this point we remind ourselves that this scenario still assumes a highly exaggerated setting where around one third of the contractors are malicious in one way or the other, and a good fraction of them are colluding or semi-independent.

7 Conclusions

In this work, we analyzed the outsourced computation setting where the boss outsources a job to multiple contractors, rewarding the diligent ones and fining the lazy ones. We have introduced a bounty mechanism and modeled the system as a multi-contractor game, where the diligent contractors who help catch the lazy ones are given an extra reward in the form of a bounty, which is taken as an extra penalty from the lazy contractors. We showed how the boss should set the relative fine, reward, and bounty to incentivize all rational contractors to act honestly. We proved that this extra bounty is indeed not an extra burden to the boss. We further categorized malicious contractors according to their capabilities, matching realistic scenarios, and theoretically analyzed the damage they can cause. We realized that purely game-theoretical solutions cannot prevent maliciousness, and existing cryptographic solutions are very inefficient. Therefore, we integrated cryptographic mechanisms together with game theoretic mechanism designs in our system.

We also presented, for the first time, a fair mechanism for the payment of *rewards and fines* using a trusted bank and a timestamping server, assuming both the boss and

the contractors may try to cheat the payment mechanism.

We implemented our outsourcing system, including the boss and the contractors (both honest and malicious). Our implementation not only helped verify our theoretical results, but also enabled presentation of results in a mixed malicious contractor setting, which is hard to analyze theoretically. We showed that **overall our outsourcing system, in expectation, incurs only 2% overhead against running a task locally, even when a quarter of all contractors are malicious and fully colluding.** We also presented that, in realistic but overly cautious experiments, with proper fine to reward ratio settings, **the boss obtains more than 99.9% correct results with only 3% extra work**. This is by far the best known efficiency when compared to the related work achieving such high levels of correctness guarantees.

APPENDIX

Theorems from [4] presented for the sake of convenience:

Theorem A.1. *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.*

Note that the theorem above applies to all types of malicious contractors in our case, except fully-independent ones.

Lemma A.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, assuming a g fraction of all contractors are colluding. 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, $B = gm$.*

Acknowledgements

The author acknowledges the support of TÜBİTAK, the Scientific and Technological Research Council of Turkey, under project numbers 111E019 and 112E115, as well as European Union COST Actions IC1206 and IC1306. The author also thanks Said Tahsin Dane, Onur Yüksel, Ezgi Kurt, and Egeyar Özlen Bağcıoğlu for their contributions.

References

[1] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, JeanPhilippe Martin, and Carl Porth. Bar fault tolerance for cooperative services. In *ACM SOSP*, 2005.

[2] Nadarajah Asokan, Victor Shoup, and Michael Waidner. Optimistic fair exchange of digital signatures. *IEEE Selected Areas in Communications*, 18:591–610, 2000.

[3] Mikhail J Atallah, Florian Kerschbaum, and Wenliang Du. Secure and private sequence comparisons. In *ACM WPES*, 2003.

[4] Mira Belenkiy, Melissa Chase, Chris Erway, John Jannotti, Alptekin Küpçü, and Anna Lysyanskaya. Incentivizing outsourced computation. In *NetEcon*, 2008.

[5] Mihir Bellare and Oded Goldreich. On defining proofs of knowledge. In *CRYPTO*, 1992.

[6] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *STOC*, 1993.

[7] Siavosh Benabbas, Rosario Gennaro, and Yevgeniy Vahlis. Veriable delegation of computation over large datasets. In *CRYPTO*, 2011.

[8] David Benjamin and Mikhail J. Atallah. Private and cheating-free outsourcing of algebraic computations. In *PST*, 2008.

[9] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of ACM*, 32:824–840, 1985.

[10] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. Cryptology ePrint Archive, Report 2013/356, 2013.

[11] Sven Bugiel, Stefan Nürnberger, Ahmad-Reza Sadeghi, and Thomas Schneider. Twin clouds: Secure cloud computing with low latency. In *CMS*, 2011.

[12] Christian Cachin(ed.). Specification of dependable trusted third parties. Technical report, IBM MAFTIA deliverable D26, 2001.

[13] Jan Camenisch, Anna Lysyanskaya, and Mira Meyerovich. Endorsed e-cash. In *IEEE Security and Privacy*, 2007.

[14] Jan Camenisch and Victor Shoup. Practical verifiable encryption and decryption of discrete logarithms. In *CRYPTO*, 2003.

[15] Bogdan Carbunar and Mahesh V. Tripunitara. Payments for outsourced computations. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 23(2):313–320, February 2012.

[16] David Chaum. Blind signatures for untraceable payments. In *CRYPTO*, 1982.

[17] David Chaum, Bert den Boer, Eugne van Heyst, Stig Mjlsnes, and Adri Steenbeek. Efficient offline electronic checks (extended abstract). In *EUROCRYPT*, 1989.

[18] Xiaofeng Chen, Jin Li, Jianfeng Ma, Qiang Tang, and Wenjing Lou. New algorithms for secure outsourcing of modular exponentiations. In *ESORICS*, 2012.

[19] Xiaofeng Chen, Jin Li, and W. Susilo. Efficient fair conditional payments for outsourcing computations. *IEEE Transactions on Information Forensics and Security*, 7(6):1687–1694, 2012.

[20] Cryptography, security, and privacy research group at koç university. http://crypto.ku.edu.tr.

[21] John R. Douceur. The sybil attack. In *IPTPS*, 2002.

[22] Wenliang Du, Mummoorthy Murugesan, and Jing Jia. Uncheatable grid computing. In *Algorithms and theory of computation handbook*. Chapman & Hall/CRC, 2010.

[23] Dario Fiore and Rosario Gennaro. Publicly verifiable delegation of large polynomials and matrix computations, with applications. In *ACM CCS*, 2012.

[24] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, 2010.

[25] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, 2009.

[26] Craig Gentry and Shai Halevi. Implementing gentrys fully-homomorphic encryption scheme. In *EUROCRYPT*, 2011.

[27] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *STOC*, 1987.

[28] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. *Journal of ACM*, 38:728, 1991.

[29] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. Delegating computation: Interactive proofs for muggles. In *STOC*, 2008.

[30] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18:208, 1989.

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

[32] Mina Guirguis, Robert Ogden, Zhaochen Song, Sobit Thapa, and Qijun Gu. Can you help me run these code segments on your mobile device? In *IEEE GLOBECOM*, 2011.

[33] Chien-Ju Ho, Yu Zhang, J Vaughan, and Mihaela Van Der Schaar. Towards social norm design for crowdsourcing markets. In *AAAI Workshops*, 2012.

[34] Susan Hohenberger and Anna Lysyanskaya. How to securely outsource cryptographic computations. In *TCC*, 2005.

[35] Seny Kamara, Payman Mohassel, and Mariana Raykova. Outsourcing multi-party computation. Cryptology ePrint Archive, Report 2011/272, 2011.

[36] Seny Kamara and Mariana Raykova. Secure outsourced computation in a multi-tenant cloud. In *WCSC*, 2011.

[37] Alptekin Küpçü. Distributing trusted third parties. *ACM SIGACT News Distributed Computing Column*, 44:92–112, 2013.

[38] Alptekin Küpçü. Official arbitration with secure cloud storage application. *The Computer Journal*, 2013.

[39] Alptekin Küpçü and Anna Lysyanskaya. Optimistic fair exchange with multiple arbiters. In *ESORICS*, 2010.

[40] Alptekin Küpçü and Anna Lysyanskaya. Usable optimistic fair exchange. In *CT-RSA*, 2010.

[41] Alptekin Küpçü and Anna Lysyanskaya. Usable optimistic fair exchange. *Computer Networks*, 56:50–63, 2012.

[42] H.C. Li, A. Clement, E.L. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. Bar gossip. In *OSDI*, 2006.

[43] Jake Loftus and Nigel P. Smart. Secure outsourced computation. In *AFRICACRYPT*, 2011.

[44] David Molnar. The seti@home problem. *ACM Crossroads*, Sep 2000. http://www.acm.org/crossroads/columns/onpatrol/september2000.html.

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

[46] Henning Pagnia and Felix C. Gartner. On the impossibility of fair exchange without a trusted third party. Technical report, Darmstadt University of Technology TUD-BS-1999-02, 1999.

[47] PandaLabs. Annual report 2012 summary, 2012.

[48] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Security and Privacy*, 2013.

[49] Bryan Parno, Mariana Raykova, and Vinod Vaikuntanathan. How to delegate and verify in public: Verifiable computation from attribute-based encryption. In *TCC*, 2012.

[50] Ahmad-Reza Sadeghi, Thomas Schneider, and Marcel Winandy. Token-based cloud computing. In *TRUST*, 2010.

[51] Luis FG Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. *Future Generation Computer Systems*, 18(4):561–572, 2002.

[52] Srinath Setty, Richard McPherson, Andrew J Blumberg, and Michael Walfish. Making argument systems for outsourced computation practical (sometimes). In *NDSS*, 2012.

[53] Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security*, 2012.

[54] Symantec. Internet security threat report 2013, 2013.

[55] Doug Szajda, Barry Lawson, and Jason Owen. Hardening functions for large scale distributed computations. In *IEEE Security and Privacy*, 2003.

[56] Doug Szajda, Barry Lawson, and Jason Owen. Toward an optimal redundancy strategy for distributed computations. In *IEEE Cluster Computing*, 2005.

[57] Justin Thaler, Mike Roberts, Michael Mitzenmacher, and Hanspeter Pfister. Verifiable computation with massively parallel interactive proofs. In *USENIX HotCloud*, 2012.

[58] Marten van Dijk and Ari Juels. On the impossibility of cryptography alone for privacy-preserving cloud computing. In *USENIX HotSec*, 2010.

[59] Michael Walfish and Andrew J Blumberg. Verifying computations without reexecuting them: from theoretical possibility to near practicality. Technical report, Electronic Colloquium on Computational Complexity, ECCC TR13-1265, 2014.

[60] Cong Wang, Kui Ren, and Jia Wang. Secure and practical outsourcing of linear programming in cloud computing. In *INFOCOM*, 2011.

[61] Yongzhi Wang, Jinpeng Wei, and Mudhakar Srivatsa. Result integrity check for mapreduce computation on hybrid clouds. In *IEEE CLOUD*, 2013.

[62] Man-Ching Yuen, I. King, and Kwong-Sak Leung. A survey of crowdsourcing systems. In *IEEE SocialCom*, 2011.

[63] Yu Zhang and M. van der Schaar. Reputation-based incentive protocols in crowdsourcing applications. In *IEEE INFOCOM*, 2012.

[64] Yuxiang Zhao and Qinghua Zhu. Evaluation on crowdsourcing research: Current status and future direction. *Information Systems Frontiers*, 16(3):417–434, 2014.