Secure Distributed Computing in a Commercial Environment

Philippe Golle[1]* and Stuart Stubblebine[2]

[1] Department of Computer Science, Stanford University, Stanford, CA 94305, USA.
pgolle@cs.stanford.edu
[2] CertCo, 55 Broad St. - Suite 22, New York, NY 10004, USA.
stuart@stubblebine.com

Abstract. The recent successes of a number of nonprofit computing projects distributed over the Internet has generated intense interest in the potential commercial applications of distributed computing. In a commercial setting, where participants might be paid for their contributions, it is crucial to define a security framework to address the threat of cheating and offer guarantees that the computation has been correctly executed. This paper defines and analyzes such a security framework predicated on the assumption that participants are motivated by financial gain. We propose a scheme which deters participants from claiming credit for work they have not done, and puts a high cost on attempts to disrupt the computation. We achieve these two goals by integrating an algorithm to assign computations to participants, an algorithm to verify their work, and an algorithm to pay participants.

Keywords: Distributed computing.

1 Introduction

The Internet has created the possibility of cooperative computing on an unprecedented scale. Connected computers everywhere may join forces to execute in parallel tasks so computationally expensive that they were once reserved for supercomputers.

Several projects have demonstrated with success the spectacular power of distributing computations over the Internet. For example, the Search for Extra-Terrestrial Intelligence project (SETI@home) [SETI], which distributes to thousands of users the task of analyzing radio transmissions from space, has achieved a collective performance of tens of teraflops. Another Internet computation, the GIMPS project directed by Entropia.com, has discovered world-record prime numbers.

Participation in these computations has so far been limited to volunteers who support a particular project. But with the rapid growth of distributed computing applications, there is intense commercial interest in recruiting a lot more

* Supported by Stanford Graduate Fellowship

Internet users. Harnessed and marketed, the idle computer time of 25 million AOL users, for example, has the potential to generate tremendous profit. There are already a dozen companies [DC00] which have begun recruiting participants for the next generation of distributed applications. A sample of these applications includes those to accelerate anti-HIV drug design research, simulate protein folding, design and manufacture robotic lifeforms, produce digital entertainment, and simulate economic models.

Nevertheless, a major obstacle to the widespread growth of commercial distributed computing is the absence of a security framework to verify the correctness of the computation. Today, the results of computations are mostly taken on faith. Bob Metcalfe, the inventor of ethernet said, "...people with serious computations are not likely to trust results coming from unreliable machines owned by total strangers" [R00].

This paper begins to address the difficult issues of security and reliability in a commercial distributed environment. We define two security goals: preventing participants motivated by financial gain from claiming credit for work they have not done, and raising the cost of attempts by malicious participants to disrupt the computation. We propose a high-level infrastructure for administering distributed computations in a way that deters and detects cheating. Our approach is based on integrating an algorithm for assigning tasks to participants, an algorithm for checking their work, and a payment scheme.

The building block of our security schemes is the ability to verify the result of a computation. Much work has been devoted to that goal in a variety of contexts, with an emphasis on making the verification process general and efficient. While our security schemes build on these results, our focus is different. Rather than considering verification at the level of individual computations, we propose a high-level infrastructure for administering distributed computations. We assume the existence of an algorithm for double-checking computations, and study how best to integrate verification into the general organization of a distributed computation.

In the rest of this section, we start by reviewing related work. Section 2 presents our security framework. In section 3, we introduce our basic scheme for running secure distributed computations. In section 4 and 5, we discuss variants of our scheme. The first variant addresses the issue of accomodating participants with varying computational resources. The second variant improves on the computational overhead of the basic scheme. We conclude in section 6.

1.1 Related Work

We start with a brief survey of results on verifying computations. We will then review proposals on how to integrate the verification primitives into distributed computations.

A general method to detect faulty execution is to incorporate a checksum in the program, and execute the computation redundantly. If independent executions fail to produce the same checksum, a majority vote determines the correct result. Ideally, checksums capture the whole execution of the computation and

detect any accidental error with high probability. See [MWR99] for an example of secure checksums for general Java programs.

For specific applications, it is possible to design checksums with a shortcut for verification. Once produced, such checksums can be verified much more efficiently than by doing the computation all over again. Program checkers [BW97,BK89], proofs of work [GM00,JJ99] or uncheatable benchmarks [CLSY93] propose efficient checksums for specific arithmetic applications, such as factoring or repeated modular squarings. It is not known how to design efficient checksums for general computations.

To guard against malicious errors, checksums may be combined with cryptographic tools. Digital signatures guarantee the integrity of checksums, assuming it is impossible to analyze the code of the computation to recover the secret signing key. Thus, security against a malicious computing environment relies on the impossibility to reverse-engineer the computation code to find the key.

Computation on encrypted functions provides one way of hiding the code from the participant. Yao proved in [Y82] that any function may be computed with an encrypted circuit, which leaks no information about what is being computed, but encrypted circuits are too large for practical use. For restricted classes of functions [F85], computing on encrypted data has been shown to be practical. Code obfuscation is another approach to protect code from prying eyes. It is an assortment of ad-hoc techniques to produce garbled assembly code. Given the existence of efficient decompilation techniques, it provides short-lived security at best. Hohl proposes in [H97] the use of dynamic code obfuscation in conjunction with time restrictions.

We turn now to the problem of integrating the verification primitive into a distributed computation scheme to make it secure. An idea common to many schemes is to spot-check computations at random (see for example [GM00]). A very complete framework for spot-checking arbitrary computation in the Java environment is described in [MWR99]. [MRS93] proposes a scheme based on replication and voting to achieve fault-tolerance in the setting of mobile agents. The use of quorum systems has been proposed for the related problem of improving the efficiency and availability of data access while still protecting the integrity of replicated data.

Like these schemes, our approach is based on spot-checking work at random, but our focus is specifically to tie the verification algorithm with the algorithm for assigning tasks and the payment scheme.

2 Basic Framework

The supervisor of a distributed effort maintains a pool of registered participants, who are willing to run computations for the supervisor. Participants may range from large companies offering idle computer time at night to individual users with a single machine. The supervisor advertises the computational power under her control, and bids for large computations. Computations are divided into smaller tasks, each of which is assigned to one or possibly several participants. Partici-

pants execute the tasks independently and return the results to the supervisor. The supervisor compares the results for consistency, distributes payment where it is due and announces the result of the whole computation when it becomes available. Formally, we define a scheme for organizing distributed computations as follows.

Definition 1. *A scheme for organizing distributed computations consists of:*

- *A protocol to register new participants.*
- *A probabilistic algorithm S for assigning tasks to participants. Given a task T, the algorithm S specifies the set $S(T)$ of (one or more) participants to whom T should be assigned.*
- *A payment scheme to reward participants. It is defined by a payment function H which specifies how much a participant should be paid for executing a task.*
- *A protocol to take leave of participants who do not wish to be considered for future computations.*

In the registration step, participants signal their willingness to contribute to distributed computations. In keeping with the state of things on the Internet, we assume that participants do not necessarily reveal their physical identity to the supervisor. An entity may register an arbitrary number of distinct participants with the supervisor, and the supervisor can not track participants to the real world. Consequently, the supervisor's leverage over participants is limited to withholding payment. It is impossible for example to take legal action against a participant who failed to do the work, or who returned an incorrect result.

We assume that all the tasks distributed by the supervisor take approximately the same time to execute (say, one day) and are verifiable. A task T is verifiable if there exists a task T' such that the output of T' indicates with high probability whether T was executed correctly or not. We have presented in the section on related work various techniques for verifying tasks. See for example [MWR99] for a very general technique. From here onwards, we will ignore this issue and make the assumption that $T = T'$. In effect, we assume that if a task produces the same output in two independent runs, it was executed both times correctly. Observe that if randomized tasks are to be included in this model, participants must agree on a random-number generator and use the same seeds.

We also assume for simplicity that all participants are equally capable of handling any task, and of returning the result to the supervisor within the same time bounds (say, every day). This is not to say that all participants have the same computational resources. Faster participants will process more tasks within one period, but all participants will return their results by the end of the period. This model reflects reality: processor speeds are roughly comparable across computers, but some participants have many more computers at their disposal than others.

A participant may choose to take a temporary leave from the computation, for example during the week-end or a vacation. But while executing a task for the supervisor, the common rate of computation must be met and results returned by the deadline, under penalty of expulsion from the computation.

The computation proceeds as follows. Let us write $T_t(J)$ for the ordered set of tasks assigned to participant J at time t. A participant J unqueues one or more tasks from $T_t(J)$ at time t, executes them, and returns the result at time $t+1$. For the purposes of this discussion we assume it is safe for participants to observe and execute problem instances they are assigned.

Meanwhile, as new tasks are received, the supervisor assigns them to participants according to the algorithm S. A new task T is queued to the sets $T_t(J)$ for all $J \in S(T)$. The set $S(T)$ is only known to the supervisor, so that participants do not know whether the tasks they are assigned have also been assigned to someone else. The supervisor keeps track of who performs what computations. If a participant is caught cheating, the past computations of that participant, as well as dependent computations, can be rerun.

After returning a result, a participant may start executing a new task or take a leave. If participant J takes a leave at time t, all the tasks in $T_t(J)$ are redistributed among active participants as if they were new tasks, taking care however never to re-assign a task to a participant to whom it has already been assigned.

2.1 Security Framework

We study the interaction between the supervisor and the participants in game-theoretical terms. The supervisor assigns each task to one or several participants according to the probabilistic algorithm S. We assume that the algorithm S is public and known to all participants. The supervisor is trusted not to collude with participants and to distribute payment where it is due.

A participant can either cooperate with the supervisor and execute the computation correctly, or defect and return an incorrect result. In this paper, we assume that all errors are malicious and do not consider the possibility of errors for which the participant may not be responsible (hardware or software failures). A simple variant of our results would allow participants to return occasional incorrect results.

An adversary may control a potentially large number of distinct participants without the supervisor's knowledge, but me make the important assumption that such alliances can only be created before a task is assigned. Once a task is assigned, we assume that is is impossible for an adversary to find and corrupt the other participants to whom the same task was assigned. We justify this assumption as follows. In the limited time available, a low-scale effort to find the participants to whom a particular task has been assigned is bound to fail, while a large-scale effort would not go unnoticed and could be punished.

The following variables define the utility function of a participant:

- **Payment received per task: $H(J)$.** This is the amount paid by the supervisor to participant J for a task not known to have been incorrectly executed. Observe that the function H is independent of the task executed, since we assume that all tasks require the same computational effort.

- **Utility of (successfully) defecting:** E**.** The variable E is the utility of corrupting a computation. For example, E might be the reward paid by an adversary to disrupt the computation of a competitor.
- **Cost of getting caught defecting:** L**.** This is the loss incurred by a participant when the defection is detected by the supervisor. The cost L reflects the resulting punishment, in terms of payment being withheld by the supervisor for example.

With these variables, we can compute the expected utility of cooperating and the expected utility of defecting for a participant. Let us write P for the probability that cheating is *undetected* by the supervisor.

$$E[\text{Cooperating}] = H$$
$$E[\text{Defecting}] = (H + E) \cdot P - L \cdot (1 - P)$$

In accordance with standard economic theory, we assume that all participants are rational and either risk-averse or risk-neutral, but not risk-seeking. Let us recall that for a non-risk-seeking agent, the utility function of money is concave. Consequently, given two options with the same expected outcome, a non-risk-seeking participant will always choose the option with the smallest standard deviation.

Definition 2. Secure computations. *A computation is perfectly secure if no rational risk-neutral or risk-averse participant ever cheats.*

Proposition 1. *A computation is perfectly secure if for every participant involved* $E[\textit{Defecting}] \leq 0$.

Proof. Recall the important assumption that participants are given the freedom to take a leave from the computation at any time. The optimal strategy for a non-risk-seeking participant is to execute the highest-paying task for which $E[\text{Cooperating}] \geq E[\text{Defecting}]$, defect for all tasks for which $E[\text{defecting}] \geq 0$, and not take part in any other computation. Therefore to ensure that no rational participant ever defects, it is enough to guarantee that $E[\text{Defecting}] \leq 0$. □

3 Probabilistic Redundant Execution

We describe in this section our basic scheme with perfect security. New participants are required to execute a few tasks for free before being allowed to register. These tasks serve both as a barrier against frivolous registrations and as a "computational" deposit with the supervisor. This deposit will be forfeited in the event that cheating is detected. A participant who signals his intention to leave the computation is paid some amount to compensate for the tasks executed for free prior to registration.

Our approach to deterring cheating is to double-check some tasks with some probability and to ban from all future computations any participant who is

caught returning an incorrect result. Being banned is a loss for a participant, regardless of whether that participant intended to take part in future computations. Indeed, a participant who wanted to take part in more computations would have to go through the registration phase again. As for participants who did not wish to be considered for future computations, they have forfeited the amount that the supervisor would have returned to them had they left the computation honestly.

Let us now describe our scheme in detail. For simplicity, we start here with the assumption that all participants have the same computational resources. We will discuss in the next section how to adapt our scheme to participants of varying computational resources. The supervisor organizes the distributed computation as follows:

- **Registration.** In the registration step, a participant is asked to run $d+1 \geq 2$ unpaid tasks. The results of these tasks is known to the supervisor. The participant is allowed to register only if all $d+1$ tasks were executed correctly.
- **Probability distribution of assignments Q:** A task T is distributed to n distinct participants where the number n is chosen at random according to the probability distribution Q. The probability distribution Q is central to our scheme. We will compare in the next two sections several possible choices for the function Q.
- **Payment function:** The payment function is a constant amount α per task. Participants are free to withdraw the money they have earned at any time.
- **Severance.** A participant who notifies the supervisor of his desire to leave the computation is paid an amount $d\alpha$.

Definition 3. Probabilistic redundant execution.
Given a task T to execute, the supervisor draws a random number n from the probability distribution Q. The supervisor chooses n distinct participants uniformly at random from among the pool and assigns each to the task T. At the end of the computation, the supervisor collects the results and compares them for validity. A participant who fails to return a result by the deadline is banned from future computations.

If all the results agree, they determine the correct output of the task. (If the task T was assigned to a single participant, it is assumed to have been executed correctly.) Each participant is paid an amount α.

In the event that not all the results agree, or that some results were not returned by the deadline, the supervisor re-assigns the task to n' new participants where n' is drawn at random from the probability distribution Q. Should this second round also fail to produce an agreement, the task is assigned again until an agreement emerges. At that time, all participants who produced the correct result are paid, while all the others are banned from future computations.

According to Proposition 1, this scheme is perfectly secure if:

$$E[\text{Defecting}] = (H + E) \cdot P - L \cdot (1 - P) \leq 0$$

where P is the probability that cheating is *undetected*, $H = \alpha$ is the amount that a participant gets paid for running the task correctly, and $L > 0$ is the loss incurred if cheating is detected. For participants who intend to be considered for future computations, L is the cost of re-registering: $L = (d+1)\alpha$. For participants who do not intend to be considered for future computations, L is the amount forfeited by being ejected from the computation: $L = d\alpha$. Thus in either case, $L \geq d\alpha$.Let us define the ration $e = E/\alpha$. We can rewrite the condition for perfect security as:

$$P \leq \frac{d}{1+e+d}$$

The choice of P involves a trade-off between security and computational overhead. The more participants a task is assigned to, the smaller the probability P that cheating goes undetected, but also the higher the computational overhead. To get the lowest possible computational overhead, the supervisor should choose the largest value P for which the security condition above holds. In the following two sections, we study how P is affected by the choice of the probability distribution Q.

3.1 Exponentially Decreasing Q

Recall that the function Q is the probability distribution according to which the supervisor chooses the number of participants to whom a task is assigned. We study in this section the properties of our scheme when $Q = Q_c$ for one of the probability distributions Q_c defined as follows:

$$Q_c[n = i] = (1-c) \cdot c^{i-1} \text{ for all } i \geq 1$$

This is an exponentially decreasing probability distribution of coefficient $0 < c < 1$. The factor $(1-c)$ is a normalization term to ensure that the probabilities sum to 1.

Proposition 2. *Let us write p for the maximum fraction of all participants under the control of a single adversary. We have $0 < p < 1$. The scheme is perfectly secure as long as $(1 - c(1-p))^2 \leq (1-p)\frac{d}{1+e+d}$.*

Proposition 3. *When the scheme is perfectly secure, its average computational cost is $\frac{1}{1-c}$, and thus the average computational overhead is $\frac{c}{1-c}$.*

Proof. If the scheme is perfectly secure and all the participants are rational, cheating never occurs. Thus, there is never a need to redistribute a task. Given the probability distribution $Q_c[n = i] = (1-c) \cdot c^{i-1}$ for $i \geq 1$, it follows that the computational cost is $\sum_{i=1}^{\infty} i(1-c)c^{i-1} = (1-c)\sum_{j=1}^{\infty}\sum_{i=j}^{\infty} c^{i-1} = \frac{1}{1-c}$ □

Corollary 1. *Asymptotically, the computational overhead of this scheme grows like $\sqrt{1+\frac{e}{d}}$, for small enough values of p.*

The corollary follows directly from Proposition 2 and 3. Before proving proposition 2, let us consider a few numerical examples. The table below summarizes the characteristics of our scheme for different values of the parameters. Observe that our scheme remains very efficient as the maximum coalition size p increases. As expected, the computational overhead grows with the square root of the ratio e/d. The computational overhead becomes quite significant for large values of e/d. This comes as no surprise: the only way to defend against an adversary willing to pay much more to disrupt the computation than the supervisor is paying for correct execution, is to distribute the tasks to a large number of participants. Only the near certainty that cheating will be detected can redress the imbalance between what participants are offered to defect and what they are offered to cooperate.

Computational Overhead (c)	Max coalition size (p)	Ratio e/d
10%	1%	0.1
17%	10%	0.1
46%	1%	1
243%	1%	10

Proof. (Proposition 2)
Consider an adversary who controls a fraction $0 < p < 1$ of the total number of active participants, and has been assigned the same computation k times through various of these participants. We denote this event E.

Let us now compute P. Let P' denote the probability that cheating is not detected during a particular round (recall that the task may be distributed multiple times if the results returned after the first round do not all agree). Then

$$P \leq \frac{P'}{1 - p(1 - P')}$$

Let us compute P'. Recall that n denotes the total number of times that the task has been assigned.

$$\begin{aligned} P' &= \Pr[n = k | E] \\ &= \frac{\Pr[E | n = k] \Pr[n = k]}{\Pr[E]} \end{aligned}$$

Now $\Pr[E|n = k] = p^k$ and

$$\Pr[E] = \sum_{i=k}^{\infty} p^k (1-p)^{i-k} \binom{i}{k} \Pr[n = i]$$

Since $\Pr[n = i] = (1 - c)c^{i-1}$, we get after simplification:

$$P = \frac{c^k (1-p)^k}{\sum_{i=k}^{\infty} c^i (1-p)^i \binom{i}{k}}$$

Let us define the function

$$I_k = \sum_{i=k}^{\infty} c^i (1-p)^i \binom{i}{k}$$

It follows easily from the equality $\binom{i}{k} + \binom{i}{k+1} = \binom{i+1}{k+1}$ that

$$I_k = I_{k+1}\Big(\frac{1}{c(1-p)} - 1\Big)$$

And $I_0 = 1/(1-c(1-p))$. It follows that

$$P' = (1-c(1-p))^{k+1}$$

And thus $P \leq \frac{(1-c(1-p))^2}{1-p}$ for all $k \geq 1$. It follows that the scheme is perfectly secure as long as $(1-c(1-p))^2 \leq (1-p)\frac{d}{1+e+d}$. □

3.2 Another Definition of Q

We study here another family of distributions Q for the number of participants to whom a task is assigned. We wish to increase the minimal number of participants to whom a task may be assigned. As before, the functions Q are exponentially decreasing with parameter c, but we now also require Q to assign each task to at least s distinct participants. The new family of probability distributions is defined as follows:

$$Q_{c,s}[n = i] = (1-c) \cdot c^{i-s} \text{ for all } i \geq s$$
$$Q_{c,s}[n = i] = 0 \text{ for } i < s$$

Proposition 4. *Using the probability distribution $Q_{c,s}$, the scheme of section 3 (Probabilistic Redundant Execution) is perfectly secure as long as:* $(1-c(1-p))^{s+1} \leq (1-p)\frac{d}{1+e+d}$.

Proposition 5. *When this scheme is perfectly secure, its average computational overhead is* $s - 2 + \sqrt[s+1]{1+e/d}$.

The proofs of propositions 4 and 5 are omitted, as they are similar to the proofs of propositions 2 and 3 respectively. The family of functions $Q_{c,s}$ results in lower computational overhead for large values of the ratio e/d. Since each task is assigned to a minimum of s distinct participants, we have to pay upfront a computational overhead of s, but asymptotically the computational overhead grows like the $s+1$ root of e/d.

Observe that the definitions of Q analyzed in this section and the previous one are by no means the only possible. We could investigate yet other distributions. However, as we will see in section 5, the most computationally efficient schemes come from another approach: defining the function Q dynamically.

4 Participants With Varying Computational Resources

We have assumed in the previous section that all participants have the same resources to contribute to the distributed computation. This assumption is not only unrealistic, but also introduces the following threat. Consider an adversary with little computational power, who is bent on disrupting one particular computation (for example, that of a competitor). That adversary might register a very large number of inactive participants, which she would activate just before the target computation is distributed. In effect, it is possible for our adversary to briefly control a number of participants which is out of proportions with her real computational power, and inflict damage on targeted computations at little cost.

We address this issue in this section. We introduce the activity, a measure of the relative throughput of each participant compared to others. Based on this measure, we modify the assignment algorithm to enable faster participants to process more tasks than slower ones. We also counter the threat we have just described by allowing only gradual increases in activity.

The activity A_t is a probability distribution over the pool of participants, which evolves dynamically over time. Whereas before participants were drawn uniformly at random from the pool, they are now drawn at time t according to the activity A_t. This leads to the following variant of our basic scheme:

Definition 4. Scheme with activity. *The assignment algorithm S, the payment function H and the probability distribution Q are defined as in the previous section, and the scheme is run as described in Definition 3 (probabilistic redundant execution). The only difference is that participants needed at time t are drawn at random according to the distribution A_t instead of uniformly at random.*

Initially, A_0 is the uniform probability distribution over all registered participants. At the end of each time period, the activity A_t is updated, to reflect the throughput of each participant on the one hand, and on the other hand to account for the arrival or departure of participants.

Recall that we write $T_t(J)$ for the ordered set of tasks yet to be processed by participant J at time t. Let $n_t(J) = |T_t(J)|$ be the number of these tasks. We write $\tilde{n}_t$ for the average over all participants of the $n_t(J)$. We define A_{t+1} as a function of A_t as follows:

$$A_{t+1}(J) = A_t(J)\Big(1 + e \cdot (\tilde{n}_t - n_t(J))\Big)$$

for all active participants J. In this formula, the coefficient $e \geq 0$ is the elasticity of the activity. At the end of the time period t, a participant who has fewer tasks left to execute than average will see her activity increase. On the other hand, a participant who can not keep pace with the tasks assigned to him will see his activity decrease ($\tilde{n}_t - n_t(J) < 0$).

The choice of the value e involves a trade-off. A higher value of e will respond faster to changes in a participant's activity, and result in fewer computational

resources being left at times untapped. On the other hand, a lower value of e will more effectively prevent adversaries from creating spikes of activity. Experience will tell what elasticity is best suited to a particular set of participants.

Observe that the way we have defined the evolution of activity so far,

$$\sum_J A_{t+1}(J) = \sum_J A_t(J) = 1$$

and thus A_{t+1} is again a probability distribution over the set of all participants. But we still need to account for the arrival of new participants and the voluntary departure or expulsion of participants:

- $A_{t+1}(J) = 0$ for all participants J who leave the computation at time t (either voluntarily or as a result of submitting an incorrect result.)
- $A_{t+1}(J_i) = 1/(n.m)$ for the m new participants J_i who wish to join the computation at time t. Here n is the total number of participants, and m is the number of new participants. Observe that all new participants share equally one n^{th} of the total activity.
- Finally A_{t+1} is normalized so that the sum of the values adds up to 1.

Proposition 6. *This scheme is perfectly secure under the same condition as that given in Proposition* 2.

Proof. The proof is exactly similar to that of Proposition 2. Indeed, the new probability distribution does not affect the strategy of cheaters motivated by financial gain. □

The activity is an indirect measure of the current throughput of each participant, relative to other participants. Observe that an absolute measure of activity, while apparently simpler, can not cope with situations where the total computational power of all participants exceeds the work available. Indeed in such situations, participants who are lucky enough to be assigned some work see their absolute activity increase, as a result of which they get assigned even more work in the next time period. Short of creating bogus tasks to keep all participants busy, such a scheme would become unstable and unfair.

4.1 Security Implications

Our goal is to prevent an adversary from rapidly creating a large number of participants, who collectively represent a significant fraction of the distribution A, just ahead of the computation that is targeted for sabotage. If the process of amassing a significant share of activity takes a long time, an adversary will be forced to contribute much computational power for a long time before having a chance to disrupt a particular computation. We study the evolution of activity in two distinct situations.

Suppose first that work is scarce, i.e. the combined computational resources of all participants exceeds the work available. In that situation, the set of tasks

$T_t(J)$ yet to be processed by a participant J is either empty, or contains very few tasks, and for all J, $\tilde{n}_t \approx n_t(J)$. Consequently $A_{t+1}(J) \approx A_t(J)$. Regardless of the resources available to participants, they always keep more or less the activity that they started with. In that setting, it takes at least k time periods to accumulate a share k/n of the activity, where n is the total number of participants. In effect, it is practically impossible for an adversary to control a significant fraction of the activity.

Let us now consider the opposite situation in which the amount of potential work available exceeds the combined computational power of all participants. Recall the formula used to update A:

$$A_{t+1}(J) = A_t(J)\Big(1 + e \cdot (\tilde{n}_t - n_t(J))\Big)$$

To slow down the potential increase of activity, we can set the elasticity e to a small value. This has the drawback of making the distribution of tasks more rigid for everyone. A better solution might be to place an upper bound e_0 on the increase of activity. For example, we could define:

$$A_{t+1}(J) = A_t(J)\Big[1 + \min\big(e_0, e \cdot (\tilde{n}_t - n_t(J))\big)\Big]$$

5 Dynamic Probability Distribution

Let us return to the scheme of section 3, probabilistic redundant execution. We propose in this section a more computationally efficient variant of this scheme, based on a dynamic definition of the probability of re-assignment Q. Recall that in our original scheme, the function Q is defined statically, in the sense that the number of participants to whom a task is assigned is chosen independently of who these participants are. It makes sense however to adjust the number of participants involved according to how trustworthy they are. For example, we might have the same degree of confidence in a result returned independently by two trustworthy participants as we have in a result returned independently by four less trusted participants.

In our model, the trustworthiness of a participant is measured by the amount L that the participant stands to lose if cheating is detected. The registration step of our scheme ensures that $L \geq d\alpha$ for all participants. Using this value for L, we proved in section 3 that our scheme is secure as long as the probability P that cheating is undetected satisfies:

$$P \leq \frac{d}{1 + e + d}$$

The variant we propose here is based on estimating the value L more precisely for each participant. Consider a participant who has already earned an amount $d'\alpha$ for running computations, but has not yet withdrawn that amount. Since that amount would be forfeited alongside the deposit of $d\alpha$ should cheating be discovered, the total potential loss for this participant amounts to:

$$L = d\alpha + d'\alpha$$

Accordingly, the security condition for our scheme becomes:

$$P \leq \frac{d + d'}{1 + e + d + d'}$$

where as before P is the probability that cheating is undetected.

The following scheme takes advantage of this weaker condition:

Definition 5. Dynamic scheme. *Given a task T to execute, the supervisor chooses distinct participants iteratively at random from among the pool and assigns each to the task T. The choice of participants proceeds as follows. The supervisor chooses a first participant A_1 uniformly at random. With probability $1 - Q(A_1)$, the task is assigned to no other participant and this completes the assignment protocol (the discussion on how to choose the function Q follows the definition of the scheme). With probability $Q(A_1)$, the supervisor assigns T to another participant A_2 chosen uniformly at random from the set of all participants minus A_1. Conditionally on having chosen a second participant, the protocol selects a third participant A_3 with probability $Q(A_1, A_2)$, and selects no other participant with probability $1-Q(A_1, A_2)$. In the general case, conditionally on having chosen an i^{th} participant, the protocol selects and $(i+1)^{th}$ participant with probability $Q(A_1, \ldots, A_i)$ and selects no more participant with probability $1 - Q(A_1, \ldots, A_i)$.*

At the end of the computation, the supervisor collects the results and compares them for validity. If they are not all the same, the task is re-assigned to a new group of participants selected according to the same protocol (if the task T was assigned to a single participant, it is assumed to have been executed correctly). Should this second round also fail to produce an agreement, the task is re-assigned until a unanimous result emerges. At that time, all participants A_i who executed the task correctly are paid α, whereas the participants who returned an incorrect result are banned from future computations.

With the notations of section 3, we define the probability of re-assignment Q as follows:

$$Q(A_1, ..., A_i) = \frac{i + e}{i + e + \sum_{k=1}^{i} (d_k + d'_k)}$$

where d_k and d'_k are the deposits of participant A_k.

Proposition 7. *With the assumption that the maximum coalition size is a negligible fraction of the total number of active participants, the dynamic scheme is perfectly secure.*

Proof. Let us consider an adversary who has been assigned i times the same task T through various of the participants that he controls. Since we assume that the adversary controls only a negligible fraction of the total number of participants, he can not learn anything about the total number of participants to whom T has been assigned. The only information available to the adversary is that the

probability of re-assignment of a task already distributed to his i participants is $Q(A_1, \ldots, A_i)$. The probability that cheating is undetected is

$$P = 1 - Q(A_1, \ldots, A_i) = \frac{\sum_{k=1}^{i}(d_k + d'_k)}{i + e + \sum_{k=1}^{i}(d_k + d'_k)}$$

This is exactly the condition required for perfect security. □

The computational overhead of the dynamic scheme is hard to estimate, since it depends on the behavior of the participants. The more participants are inclined to leave with the supervisor the money they have earned, the more efficient the scheme. The faster participants withdraw their earnings, the less efficient the scheme.

6 Conclusion and Further Work

We give a security framework for distributed computing, based on the assumption that participants are motivated by financial gain. We present a secure scheme and its analysis, as well as two variants. The first variant addresses the issue of participants with varying computational resources. The second variant offers improved computational efficiency.

We are currently working on an implementation of the schemes proposed in this paper. It is hoped that this implementation will help us determine the optimal practical value of the elasticity for the activity (section 4), and the computational overhead of the dynamic scheme (section 5) depending on the population of participants.

Acknowledgments

The first author would like to thank Bo Loevschall and Ilya Mironov for helpful discussions and comments on the subject of this paper.

References

[BK89] M. Blum and S. Kannan. Programs That Check Their Work. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, 1989.

[BW97] M. Blum and H. Wasserman. Software reliability via run-time result-checking. In *Journal of the ACM*, vol. 44, no. 6, pp. 826-849, November 1997.

[CLSY93] J. Cai, R. Lipton, R. Sedgewick, and A. Yao. Towards uncheatable benchmarks. In *Proceedings of the 8th Annual Structure in Complexity Theory Conference*, pp. 2–11, 1993.

[DC00] CNET News.com. Buddy, can you spare some processing time? CNET Networks, Inc. 150 Chestnut Street San Francisco, CA 94111. `http://news.cnet.com/news/0-1003-200-2671550.html`

[F85] Joan Feigenbaum. Encrypting problem instances: Or..., Can you take advantage of someone without having to trust him? In *Proceedings of CRYPTO 1985*, pages 477-488. Lecture Notes in Computer Science, No. 218.

[GM00] P. Golle and I. Mironov. Uncheatable Distributed Computations. To appear in *Proceedings of the RSA Conference 2001, Cryptographers' Track.*

[H97] Fritz Hohl. An Approach to Solve the Problem of Malicious Hosts. Technical Report TR-1997-03, Universitat Stuttgart, Fakultat Informatik, Germany, March 1997.

[JJ99] M. Jakobson and A. Juels. Proofs of work and bread pudding protocols. In *Proceedings of the 1999 Communications and Multimedia Security Conference.*

[MRS93] Y. Minsky, R. van Renesse, F. Schneider, and S Stoller. Cryptographic Support for Fault-Tolerant Distributed Computing. In *Proceedings of the First ACM Conference on Computer and Communications Security, Nov. 1993.*

[MWR99] F. Monrose, P. Wyckoff, and A. Rubin. Distributed Execution with Remote Audit. In *Proceedings of the 1999 ISOC Network and Distributed System Security Symposium, pages 103-113, 1999.*

[R00] H. Rheingold. You got the power. In *Wired magazine*, August, 2000.

[SETI] The Search for Extraterrestrial Intelligence project. University of California, Berkeley. `http://setiathome.berkeley.edu`

[V97] G. Vigna. Protecting Mobile Agents through Tracing. In *Proceedings of the 3rd Workshop on Mobile Object Systems,* June 1997.

[Y82] A. Yao. Protocols for Secure Computations. In *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science,* 1982, pages 160-164.