Hardening Functions for Large-Scale Distributed Computations

Doug Szajda Barry Lawson Jason Owen
University of Richmond
Richmond, Virginia
{dszajda, blawson, wowen}@richmond.edu

Abstract

Many recent large-scale distributed computing applications utilize spare processor cycles of personal computers that are connected to the Internet. The resulting distributed computing platforms provide computational power that previously was available only through the use of expensive supercomputers. However, distributed computations running in untrusted environments raise a number of security concerns, including the potential for disrupting computations and for claiming credit for computing that has not been completed (i.e., cheating). This paper presents two strategies for hardening selected applications that utilize such distributed computations. Specifically, we show that carefully seeding certain tasks with precomputed data can significantly increase resistance to cheating and to disrupting the computation. We obtain similar results for sequential tasks by sharing the computation of N tasks among $K > N$ nodes. In each case, the associated cost is significantly less than the cost of assigning tasks redundantly.

Keywords: distributed computation, probabilistic verification, ringers

1. Introduction

The past few years have seen the development of distributed computing platforms designed to utilize the spare processor cycles of a large number of personal computers attached to the Internet (see [2, 3, 4, 6, 14, 17] for academic endeavors, [19] for a list of commercial platforms). The computing power harnessed by these systems can top several petaflops, making them well suited for solving some SIMD-style parallel computations that previously required the use of supercomputers. Application domains benefiting from this technique include DNA gene sequence comparisons and protein folding in the biotechnology industry, advanced graphics rendering in the entertainment industry, exhaustive regression and other statistical applications in the financial industry, some forms of data mining, and Monte Carlo simulations. Endeavors of a more academic nature have included searches for new Mersenne primes (GIMPS) [8] and encryption keys, the Search for Extra Terrestrial Intelligence Project [18], and the Folding@home project [7]. The typical computation in this setting is easily divisible into independent tasks small enough to be handled in a few hours by an average personal computer.

In the common scenario, the *supervisor* of a distributed computation platform recruits *participants* who agree to allow the supervisor to execute code on their personal computers, either in exchange for some form of remuneration (in a commercial setting) or on a voluntary basis. Participants then download code that serves as the local execution environment for assigned computational *tasks*. For a given computation, participants are chosen, tasks are assigned and transmitted, and as tasks are completed *significant* results are collected by the supervisor. Though task results may be related, the tasks themselves are independent, so communication is necessary only between individual participating computers and the supervisor.

The emergence of these platforms has facilitated access to supercomputer-like processing speeds and enabled computations that would previously have been impractical. Providing assurance levels for results is difficult because the results are obtained by executing code in untrusted environments. Concerns include the potential for participants to intentionally or unintentionally corrupt results, and for participants to claim credit for work not completed. Validity of results can often be achieved by redundantly assigning tasks to multiple participants, but such an approach is inefficient and expensive — the processor cycles required to do so are the fundamental resource of firms providing the distributed computing service.

While there is a large body of literature concerning the security of distributed systems, there are few studies dealing with the specific type of system considered here. Golle and Mironov [9] consider computations involving inversion of a one-way function (IOWF). They present several protection mechanisms and use game theoretic arguments to measure

the efficacy of their strategies. Golle and Stubblebine [10] present a security based administrative framework for commercial distributed computations. Monrose, Wyckoff, and Rubin [13] propose instrumenting host code in order to generate lightweight execution traces that can be used to verify program execution.

Some of the work presented here extends the methods developed in [9] for the class of computations involving inversion of a one-way function. An IOWF computation seeks the pre-image x_0 of a distinguished value y_0 under a one-way function $f : D \to R$. IOWF computations consist of an exhaustive search of the domain D, with each participating host assigned a portion of the domain. In an unmodified IOWF computation, there is a strong incentive for a malicious participant to claim credit for work not completed because only a single subdomain will contain x_0, and the probability that any single participant is assigned this subdomain is low. Golle and Mironov's solution is to seed each task with *ringers*, images of randomly chosen elements of the corresponding subdomain. Participants are instructed to return any pre-image that maps to a ringer. This basic strategy is augmented in several ways (e.g., by varying the number of ringers in each subdomain) in order to achieve varying levels of assurance.

Our first strategy extends this basic ringer mechanism to more general classes of applications, including optimization and Monte Carlo simulations. Security is achieved by carefully choosing ringers so that they remain indistinguishable from genuine significant results. This is a crucial property, since any participant who recognizes the ringers planted in their tasks can circumvent the ringers scheme. Our second strategy addresses the problem of securing sequential applications. In a sequential application, the values of a function f computed during a task are dependent on the values previously computed during that task. Typically, a sequential task consists of evaluating the elements of the sequence $x_n = f(x_{n-1})$ beginning with a single input value x_0. We use the term *hardening* because neither of these strategies guarantee that the resulting computation returns a correct result, nor do they prevent an adversary from disrupting a computation. Instead they significantly increase the likelihood that abnormal activity will be detected.

The remainder of the paper is organized as follows. In Section 2 we present our model of the distributed computations and platforms under consideration. Sections 3 and 4 cover strategies for hardening non-sequential and sequential computations. We discuss related work in Section 5 and present conclusions in Section 6.

2. The model

We consider parallel computations in which the primary computation, the *job*, is easily divided into *tasks* small enough to be solved by a PC in a "reasonable" amount of time (typically on the order of several hours of CPU time). Individual tasks are independent of one another, and consist of one or more *operations*. Some jobs require tasks that consist of relatively few operations, each of which takes a relatively long time to complete, while others require tasks consisting of a large number of shorter operations. Regardless, the key characteristic of an operation is that it is the smallest *independent* unit of job execution. As an example, searching for primes typically requires long operations, where an operation consists of determining the primality of a single candidate. In this case, a task might consist of only a single operation. As a contrasting example, searching for a DES encryption key requires much smaller operations — the test of each candidate key is a single operation, and the corresponding task may consist of hundreds of thousands of operations. Hence, the granularity of a job is determined largely by the characteristics of the associated operations.

The computing platform consists of a trusted central control server or server hierarchy (which we denote by the blanket term *supervisor*) coordinating typically between 10^4 and 10^7 personal computers in a "master-slave" relationship. These *slave* nodes, or *participants*[1], are assigned tasks by the supervisor. Participants download code, typically in the form of a screen saver or applet, that serves as the local execution environment for tasks. Because tasks are independent, communication required for a computation is necessary (and allowed) only between individual participants and the supervisor. Participants receive remuneration, in one of a variety of forms, for completing their assigned task.

Formally, a job consists of the evaluation of a function or algorithm $f : D \to R$ for every input value $x \in D$. Tasks are created by partitioning D into subsets D_i, with the understanding that task T_i will evaluate f for every input $x \in D_i$. In addition to a subset of the data space, each T_i is assigned a filter function G_i with domain $P(R)$, the power set of R, and range $P(f(D_i))$, where $f(D_i) \equiv \{f(x) \mid x \in D_i\}$. For $x \in D_i$, $f(x)$ is a significant result if and only if $f(x) \in G_i(f(D_i))$. Generality in the definition of G_i is necessary for situations in which the significance of a computed value is relative to the values of f at other elements of D_i. For example, the filter function for a task in a traveling salesperson computation might specify that a route is significant if it is among the best five cycles computed.

In a non-sequential computation, the computed values of f in a task are independent of one another. In a sequential computation, computed values of f are dependent on previously computed values of f. Typically, a sequential task is given a single data value x_0 and asked to evaluate the first m elements of the sequence $x_n \equiv f_n(x_0)$, where f_n is the nth

[1] We use the term *participant* to denote both the nodes and their owners. The specific meaning of a particular usage will be apparent from the context.

order composition of the function f, $f_n = \underbrace{f \circ f \circ \cdots \circ f}_{n}$.

We assume the existence of one or more intelligent adversaries. An adversary possesses significant technical skills by which he or she can efficiently decompile, analyze, and/or modify executable code as necessary. In particular, the adversary has knowledge both of the algorithm used for the computation and of the measures used to prevent corruption. Each adversary will intentionally attempt to disrupt the overall computation in one of three ways:

- the adversary attempts to *cheat*, i.e., tries to obtain credit for work not performed;
- the adversary intentionally returns incorrect results;
- the adversary intentionally fails to return significant results.

A single adversary may repeatedly attempt to disrupt the computation as results are (incorrectly) reported and new tasks assigned. Additionally, we assume that collusion among multiple adversaries is possible. Provided the number of colluding adversaries is small relative to the number of participants, the solutions presented in this paper are suitable for hardening computations. If the proportion of adversaries is large, the validity of results returned by a distributed computation is in jeopardy regardless of the strategy used (unless the supervisor reverts to verification by recomputing the results).

An adversary may be motivated to disrupt the computation for one of several reasons. If participants receive some form of recognition (e.g., distinction as a top contributor of processing hours as in SETI@home [18] or Folding@home [7]) in exchange for processor time, an adversary may attempt to cheat, as defined above. If instead participants receive monetary remuneration, the motivation to cheat is greater still. An adversary may be motivated to return incorrect results if, for example, the adversary is a business competitor of the supervisor's firm. In this case, the adversary wants to disrupt the computation only if he or she can guarantee not being caught (for fear of severe consequences). Finally, malicious intent alone, evidenced by the abundance of hackers and viruses propagating throughout the Internet, is sufficient motivation for an adversary to return incorrect results or to not return significant results.

Attacks that result from compromises of data in transit are beyond the scope of this paper — we assume the integrity of such data is verified by other means. In addition, we do not consider attacks that result from the compromise of the central server or other trusted management nodes.

3. Hardening non-sequential computations

Golle and Mironov's basic ringer strategy for IOWF computations works as follows. Before the data for a task is transmitted, the supervisor chooses n uniformly distributed random values $x_1, x_2, \ldots, x_n$ from D_i, and computes the corresponding images. The participant is given the set $S \equiv \{f(x_1), f(x_2), \ldots, f(x_n), y_0\}$ and instructed to return to the supervisor any element of D_i that maps onto an element of S.

Our extension of this strategy is to plant each portion D_i of a task's data space with values r_i such that the following *Non-Sequential Computation Hardening Properties* hold.

1. The supervisor of the computation knows $f(r_i)$ for each i.
2. Participants cannot distinguish the r_i from other data values, regardless of the number of tasks a participant completes.
3. Participants do not know the number of r_i in their data space.
4. For some known proportion of the r_i, $f(r_i)$ is a significant result. This ensures that the supervisor has some indication of whether each participant has actually performed the assigned work.
5. It is at least as easy to implement the modification to the computation than to redundantly assign tasks.

Additionally, the following property is desirable, but not necessary.

6. The same set of r_i can be used for many different partitions of the data space so that the effort of computing the $f(r_i)$ is amortized over a large number of tasks.

A participant p_i will not be paid unless all $f(r_i)$ are returned for all $r_i \in D_i$. The participant cannot be certain all $f(r_i)$ are returned unless the *entire* task is completed. Therefore, given a set of r_i satisfying the properties above, a rational participant will complete all of the work assigned.

3.1. A practical consideration

Meeting the non-sequential hardening properties can be difficult in practice because the r_i are indistinguishable from other data only if they generate results that are truly significant to the computation. Any result can be declared significant by the supervisor, but such a result will not fool a participant who understands the computation. For example, a supervisor in a traveling salesperson computation might stipulate that any circuit with weight 100 is deemed significant. However, a participant generating a large number of

circuits with weight less than 100 will know that ringers have been artificially planted, and can return the cycles corresponding to ringers while withholding better results. Finding genuinely good ringers involves either performing some sort of approximation algorithm or precomputing a large part of the computation. In the former case, the approximation algorithm will be available to the participants, so they can determine which values are likely to be ringers. In the latter case, having the supervisor perform significant portions of the computation means losing much of the advantage gained by parallelism.

The problem is made more difficult because ringers that appear to be "hidden" in theory can be very visible in practice. In theory the supervisor of an IOWF computation seeking to find the inverse image x_0 of y_0 under the function f can generate ringers by arbitrarily choosing elements $x_1, x_2, \ldots, x_n$ from the domain of f and sending a participant the set $\{f(x_1), f(x_2), \ldots, f(x_n), y_0\}$. In practice this can be difficult to apply. As a specific example, consider the following two variations on the search for a DES encryption key.

Variation 1

> Each task is given plaintext P, ciphertext C, and a portion K_i of the key space K. The participant is then instructed to compute the set $\{E_k(P) : k \in K_i\}$, where E is a DES encryption function, and return the key k_0 such that $E_{k_0}(P) = C$, if found.

Variation 2

> Each task is given ciphertext C and a portion K_i of the key space K. The participant is then instructed to compute the set $\{D_k(C) : k \in K_i\}$, where D is a DES decryption function, and return any key that generates plausible plaintext.

Augmenting Variation 1 using the basic ringer strategy is straightforward. For each task T_i, the supervisor chooses keys $k_1, k_2, \ldots, k_n \in K_i$ (the ringers) and precomputes the ciphertexts $\{E_{k_1}(P), E_{k_2}(P), \ldots, E_{k_n}(P)\}$. The set $S = \{E_{k_1}(P), E_{k_2}(P), \ldots, E_{k_n}(P), C\}$ is sent to the participant, who is instructed to return any key that maps P to a ciphertext in S. In this situation, an adversary cannot distinguish the planted data values. Moreover, finding ringers unique to each portion K_i of the key space is trivial, so the strategy is effective even in the presence of collusion. In order to implement the ringer strategy for Variation 2, however, the supervisor must find keys k_i such that $D_{k_i}(C)$ generates plausible plaintext. This can be difficult, and may even be more expensive than assigning tasks redundantly. In addition, if the ringers are to remain hidden in the face of colluding adversaries, the supervisor is faced with the daunting task of finding keys in each K_i that decrypt C to plausible plaintext. Herein lies the subtlety. In theory, finding ringers for IOWF computations should be straightforward; in practice, however, it can be prohibitively expensive.

3.2. Hardening optimization problems

Several of the applications mentioned in Section 1 are by nature optimization problems. These include traveling salesperson problems, certain gene sequencing problems, and exhaustive regression. Fortunately, for these problems one can choose ringers that meet the non-sequential hardening properties by assigning a small proportion of the tasks redundantly and then using the *significant* results from these to seed the remaining tasks.

Formally, we consider a computation attempting to optimize a function f on a domain D to obtain a single optimizing value x or instead some set of order statistics for f. The basic algorithm is as follows.

1. Designate a proportion p of tasks as the *initial distribution*.
2. Distribute each task t_i in the initial distribution to two participants, A_i and B_i, selected at random.
3. When values are returned from the initial distribution, check for each i the results returned by A_i against those of B_i for correctness. If the results returned by A_i and B_i do not match, discard the results (and, in the same manner, further test A_i and B_i to identify which is returning incorrect results).
4. Retain the k best results and use them as ringers for the remaining tasks to verify the work of other participants.
5. Distribute the remaining tasks to other participants.

Note that if the two selected participants, A_i and B_i, in the initial distribution are colluding adversaries, the supervisor will be unable to determine incorrect results initially. However, provided the proportion of adversaries in the participant pool is small, the supervisor will eventually determine inconsistency in the results. An honest participant, C_i, not in the initial distribution will eventually provide (good) results that do not match results from the initial distribution. The supervisor then is faced with one of two scenarios: either C_i is an adversary, or A_i and B_i are colluding adversaries. With modest effort, the supervisor can determine which scenario is present. Moreover, random selection of A_i and B_i should reduce the likelihood that the two participants are colluding adversaries.

Table 1. The probability of obtaining at least k of the best n results in the first fraction p of the data space

n	k	p	Probability
50	8	0.25	0.9547
150	5	0.1	$\approx P(Z < 2.72) = 0.9967$
10000	100	0.02	$\approx P(Z < 7.143) = 1$

The probability[2] that at least k of the best n results will be obtained from the initial distribution of tasks is given by $\sum_{i=k}^{n} \binom{n}{i} p^i (1-p)^{n-i}$, assuming that both n and k are much less than the size of the data space, and that the proportion of incorrect results is small. Values for some combinations of n, k, and p are shown in Table 1. In particular, note that it is a virtual certainty that 100 of the top 10000 results will be obtained if only 2% of the tasks are contained in the initial distribution. For a traveling salesperson problem with 50 cities, or for an exhaustive regression on 32 variables, the best 10000 results represents better than the 99.99999th percentile.

The filter function G must be carefully specified to ensure ringers remain both hidden and significant. If an adversary is able to determine the ringers, the adversary can disrupt the computation by returning those ringers while at the same time omitting better results or including incorrect results. Consider a traveling salesman problem. If the initial distribution returns five good circuits, $c_1, c_2, \ldots, c_5$, with lengths 100, 105, 102, 113, and 104, then G can be expressed in three general forms.

- Return any circuit whose length is 100, 105, 102, 113, 104, or less than 100.
- Return the ten best circuits you find.
- Return any circuit whose length is less than 120.

In the first case, the participant is given information that can be used to identify the ringers. In the second, a participant cannot identify the ringers, but if a particular data subspace contains many good circuits it may turn out that some ringers are not significant. As a result, the supervisor can verify the correctness of the returned circuits, but has no measure of assurance that all circuits in this task have been evaluated. The last form leaves ringers hidden and guarantees their significance, though it may also lead to excess results being returned. Thus the exact cutoff used must be tuned to specific applications.

[2]Technically, this probability should be adjusted to reflect the expected number of incorrect results returned in the initial distribution, which we expect will be relatively small.

This strategy is advantageous because no precomputing is required on the part of the supervisor and the computation is hardened at a fraction of the compute cost of simple redundancy. Additionally, the ringers obtained in the initial distribution can be used to seed the remaining tasks, freeing the supervisor from having to compute ringers for each individual task. The strategy is collusion resistant because even a modestly small number of ringers can be combined in enough ways to reduce the probability that colluding participants (or participants who complete multiple tasks) will be able to determine the ringers. Moreover, as additional good results are obtained from the remaining tasks, these results can be used to seed any further tasks that may be assigned.

The disadvantage of this method is that the time cost of an individual computation is at least doubled, assuming that all tasks require approximately the same amount of time to complete, because tasks are distributed in two waves rather than all at once. By running multiple jobs concurrently, however, overall job throughput rates can be reduced to a factor of $1 + p$ times unmodified job throughput rates.

3.3. Hardening Monte Carlo simulations

Monte Carlo simulation (see [11]) is a technique that employs random numbers to solve problems in which time plays no substantive role. The technique involves simulating a random experiment a large number, say N, of times and recording the number of times, say C, that an event of interest occurs. The law of large numbers asserts that if N is large, the ratio C/N should be a good point estimate of the probability of the event occurring.

As a simple example, consider the problem of finding the area of a region S contained in the square $U \equiv [0,1] \times [0,1]$ in the xy-plane. Using Monte Carlo simulation, one can choose N points from a uniform distribution in U, and count the number of points, C, that lie in S. The approximation for the area would then be C/N.

This example is not well suited for a large scale distributed computation, but serves as an illustration of how the seeding technique can be applied to Monte Carlo simulations in general. The supervisor chooses a particular implementation for the random number generator (ensuring portability) and some number k of seeds. Before any tasks are assigned, an initial run of N/k replications is computed using one of the seeds s' chosen arbitrarily. This seed becomes the ringer for the remaining task assignments. Participants are then sent the code for the generator along with k seeds (including s'), and are instructed to run N/k replications with each of the seeds, returning the area estimate corresponding to each seed. An adversary cannot determine which of the k seeds is the ringer, and therefore cannot return results for fewer than k seeds without raising suspicion.

The returned results can be checked for validity using the initial run generated with s'. In effect, the supervisor has managed to provide a measure of assurance while performing only $1/k$ of the work.

By their very nature, Monte Carlo simulations provide a form of redundancy because, provided the number of replications is sufficiently large, each task should return an estimate similar to the other tasks. However, seeding as described here augments the redundancy by enhancing the resistance to collusion.

4. Hardening sequential computations

As described in Section 2, in a sequential computation a task t_i is given a single data value x_0 along with the function f and asked to evaluate the first m_i elements of the sequence $x_n \equiv f_n(x_0)$, where f_n is the nth order composition of the function f and m_i is typically very large.

Unlike a non-sequential task, in which the participant computes the value of a function on several *independent* inputs, the intermediate results in a sequential computation are highly dependent. Thus, seeding the data is impractical. Moreover, in many cases the validity of a returned value can only be checked by recomputing the entire task. The Great Internet Mersenne Prime Search (GIMPS) [8] conducted by Entropia.com provides a case study. The nth Mersenne number, denoted M_n, is defined by $M_n \equiv 2^n - 1$. A Mersenne number can only be prime if n is prime[3], but the primality of n is not a sufficient condition for the primality of M_n (e.g., M_{67} is not prime). The Lucas-Lehmer Theorem [5] states that M_n is prime if and only if $S(n-1) = 0 \pmod{M_n}$, where

$$S(k+1) = \begin{cases} 4 & k = 0 \\ S(k)^2 - 2 & k = 1, 2, \ldots \end{cases}$$

Thus a GIMPS task consists of checking a single candidate. Considering that the most recent GIMPS success was the discovery of the Mersenne prime $2^{13,466,917} - 1$, the number of iterations required in such a task is significant.

We now propose a new strategy for hardening sequential computations that provides probabilistic verification. In [20], Syverson briefly introduces the idea of probabilistic verification of a sequential computation by dividing the computation into smaller pieces. However, the strategy we propose differs in that the supervisor is not required to precompute or recompute results. Our strategy for sequential computations shares the work of computing N tasks among K participants, $p_1, p_2, \ldots, p_K$, where $K > N$ is a very small proportion of the total number of participants in the computation. In addition, we assume that each of the N tasks requires roughly m iterations. The algorithm is as follows.

1. Tasks are divided into S segments, the first $S-1$ of these consisting of $J = m/S$ iterations, and the last containing a variable number of iterations (since the number of iterations for each task may not be exactly m).

2. Each participant p_i is given an initial value x_{i0} and instructed to compute the first J iterations using that value.

3. When each of the p_i has completed J iterations, it stops and returns the last value computed, $f_J(x_{i0})$, to the supervisor.

4. The supervisor checks the correctness of the $f_J(x_{i0})$ corresponding to redundantly assigned subtasks.

5. The supervisor permutes the N distinct values in the set
$$\{f_J(x_{10}), f_J(x_{20}), \ldots, f_J(x_{K0})\}$$
and assigns these values to the K participants as initial values for the next segment.

6. The process is repeated until all S segments have been completed.

If the *redundancy factor* K/N is less than 2 and each subtask is assigned to no more than two nodes[4], then in the absence of collusion, the probability of a cheater being caught in a given segment is $\frac{2(K-N)}{K}$. (For example, if $K = 6$ and $N = 4$, and nodes 1–6 are assigned tasks 1–4 according to node 1 $\rightarrow$ task 1, node 2 $\rightarrow$ task 2, node 3 $\rightarrow$ task 3, node 4 $\rightarrow$ task 4, node 5 $\rightarrow$ task 2, node 6 $\rightarrow$ task 4, then the work of nodes 2 and 5 are checked against each other, as is the work of nodes 4 and 6. Thus four of six nodes are checked. In general, $2(K-N)$ of the nodes will have their work checked.) Thus the probability of a cheater being caught in S segments is $1 - \left(\frac{2N-K}{K}\right)^S$. If a participant cheats in a fraction p of the segments, then the probability of being caught decreases to $1 - \left(\frac{2N-K}{K}\right)^{pS}$. Table 2 provides examples for various input parameters of the probability of a cheater being detected.

Note for the probabilities given here, if a cheater cheats exactly L times then $p = L/S$ so the probability of being caught is given by $1 - \left(\frac{2N-K}{K}\right)^L$. Note that the last equation is not independent of S; in fact S is an upper bound for L. In order to have at least a probability P of catching the cheater, one needs

$$K > \frac{2N}{1 + (1-P)^{\frac{1}{L}}}.$$

A small value of S (thereby limiting L) means that more redundancy will be required for a given level of security.

[3] If m divides n, then $(2^m - 1)$ divides $(2^n - 1)$.

[4] We assume $K/N < 2$; otherwise, simple redundancy requires less work than our approach.

Table 2. Probabilities of catching a cheater

nodes (K)	tasks (N)	segments (S)	cheating frequency (p)	P(cheater caught)
5	4	5	1	0.9222
5	4	10	1	0.9939
5	4	10	.2	0.64
5	4	20	.2	0.8704
10	9	10	1	0.8926
10	9	10	.2	0.36

Table 3. Redundancy factors for various P values

$L = 1$

P	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
K/N	1.05	1.11	1.18	1.25	1.33	1.43	1.54	1.67	1.82	2.0

$L = 2$

P	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
K/N	1.03	1.06	1.09	1.13	1.17	1.23	1.29	1.38	1.52	2.0

Table 3 shows the redundancy factors required for $L = 1$ and $L = 2$ with various P values.

The primary advantage of this strategy is that far fewer task compute cycles are required than for simple redundancy. Similar to our seeding strategy for non-sequential tasks (except Monte Carlo simulations), there is no need for the supervisor to precompute any values. The method is also collusion resistant (unless a supervisor is unfortunate enough to select a group consisting entirely of colluding nodes) because the returned results are permuted and reassigned. In addition, the method is tunable — a supervisor can set security levels by varying the redundancy factor. Finally, the technique can be applied to non-sequential computations as well.

The primary disadvantage of the scheme is an increased workload for the supervisor, who experiences an increase in both coordination and communication costs due to the node synchronization requirements. The need for synchronization also increases the time cost of a computation, which can be especially expensive if many of the volunteer PCs are connected to the Internet via modems or operate sporadically because owners use the machines for their own purposes. As more PC owners move to high speed 24/7 connectivity, synchronization costs will likely be less of an issue (e.g., jobs can be run late at night). From a security standpoint, the strategy does not protect well against a malicious adversary who decides to cheat just once. Moreover, the amount of damage a cheater can do is magnified, because incorrect results that are not caught become input values in subsequent segments of the computation.

5. Related work

The present problem relates to the validation of code execution, so its historical roots lie in the areas of result-checking and self-correcting programs. Wasserman and Blum [22] provide an excellent survey of the results in this area. While of theoretical interest, it is not directly applicable here because much of the work is limited to specific arithmetic functions, and checking is limited to verifying function behavior on a single input, rather than on all inputs. Result checkers for general computations remain elusive.

Several recent implementations of distributed computing platforms address the general issues of fault-tolerance [2, 3, 4, 6, 14, 17], but assume a fault model in which errors that occur are not the result of malicious intent. The solutions presented are typically a combination of redundancy with voting and spot checking. In a preliminary investigation of the problem of fault-tolerant distributed computing, Minsky et al. [12] found that replication and voting schemes alone are not sufficient for solving the problem. They assert that cryptographic support is required as well, but only sketch the methods they envision for solving this.

There have been a number of efforts aimed at protecting mobile agents from malicious hosts. Vigna [21] proposes using cryptographic traces to detect tampering with agents. Specifically, an untrusted host that is providing the execution environment for a mobile agent is required to generate, and for a short while store, a trace of the agent execution. Upon completion of the execution, the untrusted host returns a hash of the trace, and if requested by the originating

host, the complete trace. This of course means that verification of the correct execution is provided by having the code executed twice, once on the trusted node, and once on the untrusted node. In addition, as Vigna notes, even if traces are compressed, they can be huge. While there are mechanisms that can be used to decrease the size of traces, the communication overhead remains far too great to be practical for a metacomputation.

Sanders and Tschudin [16] discuss the idea of providing security for mobile agents by computing with encrypted functions [1, 15]. The idea is to use an encryption function E to encrypt the code for a procedure $\mathcal{P}$, obtaining a second function $E(\mathcal{P})$ that provides little information about $\mathcal{P}$. An untrusted second party then executes $E(\mathcal{P})$ on a given input x and returns the result, which is then decrypted to obtain $\mathcal{P}(x)$. The difficulty here lies in creating encryption functions that map executable procedures to executable procedures. There are other requirements for E, including resistance to chosen plaintext attacks, ciphertext only attacks, and other attacks. Abadi and Feigenbaum [1] present an encryption function for a general boolean circuit, but their method requires a great deal of interaction between the communicating parties. Sanders and Tschudin add the constraint that the encryption function should not be interactive, since frequent communication between an agent and the server from which it originated effectively eliminates the benefits gained from agent autonomy. The methods they present apply to procedures that evaluate restricted classes of polynomials and rational functions. Because no methods are presented for more general procedures, however, and because it is not even known whether such encryption functions exist, their methods, though interesting, present practical difficulties.

In addition to the work of Golle and Mironov [9], two other works focus specifically on the issue of securing distributed metacomputations. Golle and Stubblebine [10] present a security based administrative framework for commercial distributed computations. Their method, like those presented here, relies on selective redundancy to increase the probability that a cheater is detected. They provide increased flexibility, however, by varying the distributions that dictate the application of redundancy. Efficacy is measured by first developing a game theoretic model based on estimates of the participant's utility of disrupting the computation and cost of being caught defecting, and then determining distribution parameters that guarantee that, for every participant involved, the expected value of defecting from the computation is less than or equal to zero. The differences between their methods and those presented here lie in the particulars of how redundancy is applied and with the granularity of redundancy.

Monrose, Wyckoff, and Rubin [13] deal with the problem of guaranteeing that a host participates in the computation, assuming that their goal is to maximize their profit by minimizing resources. The method involves recording traces of task execution. Specifically, task code is instrumented at compile-time so that it produces checkable state points that constitute a proof of execution. On completion of the task, the participant sends results and the proof to a verifier, which then runs a portion of the execution and checks it against the returned state checkpoints. However, this approach requires the undesirable need to recompute results.

6. Conclusions

We have presented two strategies for hardening large-scale distributed computations against malicious behavior by participating hosts. The first, applicable to many non-sequential computations (such as optimization problems), requires seeding task data with ringers in a manner that prevents participants from being able to distinguish the ringers from genuinely significant results. The second strategy, applicable to sequential computations (such as GIMPS), advocates sharing the work of computing N tasks among $K > N$ nodes. Relative to an unmodified task, a small increase is incurred in the average execution time of a task modified to execute with our strategies. However, the *overall* computing costs are significantly decreased compared to redundantly assigning entire tasks. In addition, both strategies provide supervisors protection against participants who fail to complete assigned tasks, and provide a measure of assurance of the validity of returned results.

Acknowledgments

We would like to thank the anonymous reviewers whose comments helped us in preparing the final version of the paper.

References

[1] M. Abadi and J. Feigenbaum. Secure circuit evaluation: A protocol based on hiding information from an oracle. *Journal of Cryptology*, 2(1):1–12, 1990.

[2] J. Baldeschwieler, R. Blumofe, and E. Brewer. Atlas: An infrastructure for global computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[3] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the web. In *Proc. of the 9th Int'l Conf. on Parallel and Distributed Computing Systems (PDCS-96)*, 1996.

[4] T. Brecht, H. Sandhu, M. Shan, and J. Talbot. Paraweb: Towards world-wide supercomputing. In *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[5] J. Bruce. A really trivial proof of the Lucas-Lehmer test. *American Mathematical Monthly*, 100:370–371, 1993.

[6] P. Capello, B. Christiansen, M. Ionescu, M. Neary, K. Schauser, and D. Wu. Javelin: Internet-based parallel computing using Java. *Concurrency: Practice and Experience*, 9(11):1139–1160, 1997.

[7] The Folding@home Project. Stanford University. http://www.stanford.edu/group/pandegroup/cosm/.

[8] The Great Internet Mersenne Prime Search. http://www.mersenne.org/prime.htm.

[9] P. Golle and I. Mironov. Uncheatable distributed computations. In *Proceedings of the RSA Conference 2001, Cryptographers' Track*, pages 425–441, San Francisco, CA, 2001. Springer.

[10] P. Golle and S. Stubblebine. Secure distributed computing in a commercial environment. 2001. http://crypto.stanford.edu/∼pgolle/papers/payout.html.

[11] A. Law and W. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, 3rd edition, 2000.

[12] Y. Minsky, R. van Renesse, F. Schneider, and S. Stoller. Cryptographic support for fault-tolerant distributed computing. In *Seventh ACM SIGOPS European Workshop*, pages 109–114, Connemara, Ireland, 1996.

[13] 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.

[14] N. Nisan, S. London, O. Regev, and N. Camiel. Globally distributed computing over the internet—the Popcorn project. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 592–601, Amsterdam, Netherlands, May 1998.

[15] R. Rivest, L. Adleman, and M. Dertouzos. On data banks and privacy homomorphisms. In R. D. Millo, D. Dobkin, A. Jones, and R. Lipton, editors, *Foundations of Secure Computation*, pages 169–179. Academic Press, New York, 1978.

[16] T. Sander and C. F. Tschudin. Protecting Mobile Agents Against Malicious Hosts. In G. Vigna, editor, *Mobile Agent Security*, pages 44–60. Springer-Verlag: Heidelberg, Germany, 1998.

[17] L. Sarmenta and S. Hirano. Bayanihan: Building and studying web-based volunteer computing systems using java. *Future Generation Computer Systems*, 15(5/6), 1999.

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

[19] T. Stein. A cycle built for few. In *Red Herring* magazine, December 2000.

[20] P. Syverson. Weakly secret bit commitment: Applications to lotteries and fair exchange. In *PCSFW: Proceedings of The 11th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1998.

[21] G. Vigna. Cryptographic Traces for Mobile Agents. In G. Vigna, editor, *Mobile Agent Security*, pages 137–153. Springer-Verlag: Heidelberg, Germany, 1998.

[22] H. Wasserman and M. Blum. Software reliability via runtime result-checking. *Journal of the ACM*, 44(6):826–849, 1997.