4-7-2004

Securing distributed computations : in search of reliable large-scale compute power and refreshed redundancy

Edward P. Kenney
University of Richmond

MAT
Ken

Securing Distributed Computations

In Search of Reliable Large-Scale Compute Power and Refreshed Redundancy

University of Richmond, Department of Computer Science
Edward P. Kenney
Dr. Douglas Szajda, Advisor

April 7, 2004

Abstract: *Distributed computations allow computers connected via an internetwork To combine their computational power in order to achieve solutions to otherwise intractable problems. These computations harness unused processor power aggregated over thousands of computers, providing a tremendously valuable resource. Unfortunately security in this setting is a difficult matter. Code must be executed in potentially hostile environments, and participant identity is an intangible entity. In addition, consideration of application specifics must be of paramount concern. This paper presents a framework for evaluating the security of these computations and offers an updated approach to redundancy that can significantly impact the efficacy of distributed computation security mechanisms.*

This paper is part of the requirements for the honors program in computer science. The signatures below, by the advisor, a departmental reader, and a representative of the departmental honors committee, demonstrate that Edward Kenney has met all the requirements necessary to receive honors in computer science.

[illegible]

(advisor)

[illegible]

(reader)

[illegible]

(honors committee representative)

1 Introduction

The Internet may be the single largest technological advance or significant societal change in the last century. Not only does it allow access to more information than any human could ever hope to digest, but it produces the potential of having millions of computers combining their computational forces for the betterment of a single cause.. Ti is is the fundamental goal of distributed computing. A distributed system is defined to be a network of machines with some degree of œntralized direction. In a distributed computational system each machine will accept computational tasks from a supervisor in a master-slave relationship. Thus the compute power of many machines can be combined to solve difficult problems. A single user is potentially given the power of thousands of machines at his finger tips and the sum capability of the millions of computers whose processors sit idle much of the time is better utilized.

The potential is tremendous, but the realization of this potential is problematic. In an increasingly interconnected and technology savvy world, security becomes ever more important. There are, however, fundamental problems with securing mobile elements. Supervisors of a distributed computation must use results produced by code executed by a potentially malicious participant. Further complicating these matters, it is difficult to convincingly identify the human identity of an online user. The full potential of the scientific advance that is the Internet remains unrealized due to these limitations and hurdles. A problematic absence in the current literature is a set of common, assumptions about distributed computations, the security thereof, and the adversaries that must be considered. This paper describes a viable set of assumptions about the structure of a distributed computation and the nature of its security. It presents a novel set of implementations for redundancy that can significantly improve upon the security of a system.

Section 2 provides a distributed computation model. Section 3 provides currently implemented examples of distributed computations. Section 4 breaks down the fundamental characteristics of these computaitons and their platforms, and provides a rough taxonomy that may act as a guide in considering the security mechanisms that are needed and achievable for a given application. Section 5 addresses implementation issues. Section 6 details redundancy schemes that provide enhanced resistance to redundancy. Section 7 presents analysis of a simulation mode ling the implementation of the method presented in Section 6. Section 8 details the current state of mobile agents and the relationship of that subject area to distributed computation research. It also describes possible future research. Section 9 is the conclusion.

2 Distributed Computation – The Model

As previously mentioned, the Internet offers tremendous potential in terms of aggregate compute power. This can only be achieved, however, if the issues produced by a hostile, interconnected environment can be overcome. This section describes the computations under consideration: the problem domains applicable to distributed computing platforms, possible platform infrastructures, assumptions surrounding the typical adversary, and the security characteristics desired in such a computation. General terminology is introduced throughout.

Distributed computations are controlled by a central server (the machine), or supervisor (the human in charge). The administrator recruits participants either by financial reward or by an appeal for cooperation in a non-commercial endeavor. When an Internet user decides to become involved in a computation, he downloads a screensaver or applet to his machine. This allows the computation to be executed during processor "down-time", reducing the number of un-used processor cycles and contributing to the overall compute power of the system. With this setup completed, tasks can be assigned and completed on a dynamic basis. Tasks will necessarily be able to be completed by the average, "normal", personal computer in a few hours of processor time. Due to the fluctuating nature of that computer's use, task completion times will vary among the various participants in a computation, but all tasks will normally have a time-out mechanism that triggers reallocation of the task.

A *job* is defined as the assignment and distribution of tasks, their completion by the various participants, their collection by the central server, and ultimately the production of a result via the combination of many returned solutions. A *task* is defined as the set of *operations* assigned to a participant. Operation in this context constitutes the smallest unit of job execution. For example, in a DES Encryption Key search, the job would be equivalent to finding the correct encryption key, each task would consist of a subspace of the total key-space, and operations would consist of checking a single key.

It is important to define the security characteristics desired of these computations. In the context of a distributed computation we are looking for guarantees of the following:

1) Data Secrecy
2) Participant Authenticity
3) Data Integrity
4) Cheating Resistance
5) Disruption Resistance
6) Data Confidentiality

Assume that the central control server is protected via a PKI implementation. A sophisticated saboteur who is capable of compromising said server has the ability to intercept all messages from participant to administrator, mirroring a typical cryptographic scenario. Although this is a sophisticated attack, methods of protection against it are well known, and the compromise of those mechanisms is beyond the scope of this paper. The adversary is therefore assumed to be incapable of such work. A public key cryptographic protocol, including digital signatures, can assure data secrecy, participant authenticity, and data integrity. Further definition is necessary here. Data secrecy in this context refers to keeping data secret as it is transmitted from supervisor to participant. Participant authenticity is more specifically "username" authenticity, as it is impossible to reliably bind human identity to virtual identity. This will be discussed at length in section 5. See Rivest, Shamir, and Adleman [1] for further details on public key cryptographic protocols and their capabilities. This having been established, it is thus necessary to focus on cheating resistance, disruption resistance, and data confidentiality when considering the effectiveness of a proposed security mechanism, and it is these characteristics that will be the focus of this paper.

3 Examples

Distributed computation can be used to solve a large and varied assortment of computationally challenging problems. Several are briefly described here.

Exhaustive linear regression is a data-fitting technique that matches experimental data to a function. Given an experimental dataset and a necessarily finite set of possible fit-functions, which function provides the best match, or smallest cumulative discrepancy between values predicted by the function and actual data? This question is addressed in a brute force manner, by testing all possible fits (in practice this is accomplished by varying the coefficients of an n-dimensional function), making it an apt application of distributed computing. For further discussion of regression analysis see Draper and Smith [12].

Monte Carlo simulation, as a general class of problem, is also particularly well situated to take advantage of a distributed computation. By utilizing a simulation of a real-world situation, combined with randomization, this stochastic process can achieve results to problems that prove intractable to analytic techniques. Monte Carlo simulation is being used to achieve solutions in such areas as financial modeling, climate prediction, and graphics rendering. The need to complete computations on large numbers of randomly generated numbers makes these computations ideal for distributed computing.

SETI@home [5] is one of the most well-known distributed computations in current operation. Massive compute power is needed to analyze the terabytes of data on incoming signals from space collected by the project. If a signal indicates evidence of extraterrestrial life the supervisor is notified.. Absent the dedication of a supercomputer(s) to the project, the analysis would be completely intractable, yet SETI, which has millions of participants is succeeding in analyzing huge amounts of information from space in the search for extraterrestrial intelligence.

Folding is the process by which a protein, consisting of a string of amino acids, takes on the physical configuration that gives it biochemical functionality [9] A protein's folded structure is predetermined by it's amino-acidic composition, however determining this configuration *a priori* is quite difficult. Furthermore complete atomic simulation of the protein folding process is highly valuable to molecular biologists, but hindered by technical challenges. Proteins fold quite rapidly on a human scale, on the order of a microsecond in fact. Computational intensity, however, limits simulation to just 1 nanosecond per CPU-day [8]. Thus simulation of a full protein fold becomes an intractable problem via conventional means, warranting the use of a distributed computing platform.

FOLDING@home [10], a distributed computing project run by the Pande Group at Stanford University, seeks to parallelize the folding process by coupling molecular dynamics simulations via a method termed "ensemble dynamics" [6]. The ensemble dynamics method creates many independent folding series, where each series consists of a large number of trials. Here 'trial' denotes a simulation with a specific set of atomic location coordinates and a random set of thermodynamic forces acting upon those atoms. The trials within a series are distributed over thousands of participants , have the same initial set of location coordinates, and are run in parallel. This is a non-SIMD style computation that can run in parallel with novel approaches.

Also in the realm of molecular biology is the problem of sequence matching. This application is a prime example of an optimization problem. For DNA sequence comparison, for instance, given the Smith-Waterman local sequence comparison [14] for determining "likeness" of DNA sequences, distributed computation allows each participant to evaluate a set of possible matches in order to determine properties of a DNA sequence's evolution.

Finally, distributed computation is an effecti e means of addressing various instantiations of the traveling salesperson problem, and is an excellent option for "needle in a haystack"-like searches such as computation of a DES Encryption Key and the Great Internet Mersenne Prime Search [13].

4 Application Characteristics – A Rough Taxonomy

Given a specific distributed computation, it is useful to be able to classify that problem in terms of its various characteristics. This section provides a model for the computing platform, the problem domain, and the adversary, and can serve as a guide in terms of what security mechanisms are possible in a given context.

4.1 Model of the Platform

Platform dictates such factors as the incentive structures of the participants, trust relationships, expected privacy of data, resource ownership, and the anonymity and privacy of participants. These elements can affect the security threats faced, dictate the security options available, and determine the potential efficacy of those measures. The computing platform consists of a trusted central control server coordinating a large number of personal computers in a ``master-slave" relationship. These worker nodes, or participants are assigned tasks by the central control server. The worker nodes then complete the specified tasks and return significant results. "Significant" is necessarily context-dependent, but this is important as it allows for computation tuning, or changes in the results desired. Participants may be required to periodically send progress reports to the server, and there will always be a limit on the time permitted to complete a task, though this time limit may be as long as a few days. There are several platforms possible for this type of computation: enterprise, Internet, and hybrid. This differentiation is necessary because of significant differences in security issues arising from variations in the ``demographics" of the pool of participants.

4.1.1 – Enterprise Computations: An enterprise computation denotes a computation completed with the resources provided by a single company or institution. An example might be the donation of all computers on a University network to a distributed computation. The key result of this setup is that there is a strong likelihood that participants in this setting could know one another. Furthermore, a network administrator exercises near-complete control over her network resources. Thus collusion becomes a significant problem. An administrator could control multiple participants to achieve this, or multiple human identities r presenting computation participants could collude in an attack scenario.

4.1.2 – Internet Computations: An Internet computation utilizes the resources of potentially every computer on the Internet. Typically a computation supervisor advertises for his/her computation via some sort of awards system, financial or otherwise, or by appealing to the Internet population on the basis of helping to further a good cause. In this setting it is less likely that participants will know one another.

4.1.3 – Hybrid Computations: Hybrid computations are just that: computations in which organizations, companies, or institutions donate large numbers of computers to a computation, but that computation's participant base is filled out with the addition of other Internet users. This setting suffers less from the threat of collusion than, say enterprise computations, but more so than Internet computations.

4.2 Model of the Adversary

Having considered both the computational platform and the problem domain, it is important also to take into account a model of the adversary. It is important to assume that an adversary is both intelligent and goal oriented. He possesses significant technical skills, including the ability to efficiently decompile, analyze, and modify task or execution environment code. An adversary may attempt to cheat, by which it is meant that he tries to obtain credit for work that he has not performed, or he may try to disrupt the computation by intentionally returning incorrect results or failing to return significant results. In this model, a disrupting adversary, or saboteur is only deterred by a significant penalty, with severity on the order of loss of employment or criminal penalties. This observation is justified due to the lack of economic gain via disruption of a computation. A cheater, in comparison, is motivated by gaining credit for work not performed, and therefore *is* motivated by loss of credit.

Elaboration upon the two classes of opponent is needed. The first is the cheater, the second, the saboteur. We will call our cheater Calvin, and our saboteur Stanley. First consider Calvin. Calvin, as a cheater, is motivated by a desire to gain credit for work not actually performed. This may mean that he returns a random response without backing it with any work. We will assume that Calvin's intelligence at the very least allows him to make his response make sense in the context of the problem. Although blacklisting, as discussed later, is ineffective, a cheater such as Calvin can still be sanctioned via revocation of credit previously earned.

Stanley is a very different opponent. As a saboteur, his goal remains only to disrupt the computation. This is the adversary from whom might come a false positive. In fact, as discussed below, some applications might be especially susceptible to this type of attack. Reprimand against Stanley is more difficult affair. It must be assumed that he is not performing any illegal action, but rather just being dishonest. This then eliminates the option of severe punishment, such as criminal prosecution. A supervisor's only course of action, then, is to prevent Stanley's efforts from successfully disrupting the computation.

Consider two possible cheating strategies. The firsts shall be called unanimous denial. In this case, Calvin returns a negative response in all cases. In other words, he denies having received a significant result as a matter of policy. There are a number of observations to be made here. Suppose that there are s total significant results in the search space, and that each participant is given a total of g data elements to evaluate. Then the probability that Calvin will receive a significant result within his test space is $g \backslash s$. However, recognize that should Calvin return a negative response when in fact his test-space included the correct key, the administrator will find out. A standard procedure would be to re-distribute the key-space to another set of participants. Assume that this group does not contain a cheater, and upon completion of the computation, the administrator will know that Calvin cheated, assuming that record is kept of the distribution of tasks. Thus Calvin's probability of being caught is also $g \backslash s$. The key observation here is that in an easily checkable computation such as cryptographic key search, a cheater will *always* be caught. His probability of being caught is equivalent to his probability of receiving a significant answer.

Another strategy for Calvin might be to compute half of the assigned task. At the completion of half of the work then, Calvin could respond with a negative answer if nothing has been found. This reduces his chance of being caught, but also forces him to do more work. Thus he is cheating less, but risking less. It should also be apparent, at this point, that the first strategy might be an excellent option for Stanley. In the case that a saboteur is given the test-space containing a significant answer, he can purposely fail to return the result. Because this will force the entire computation to be re-computed, Stanley will have achieved disruption. Research is needed on what attack strategies are actually launched on distributed computations. It is clear however, that in an Internet computation Stanley can go about his saboteur activities in relative comfort in many scenarios.

4.3 Model of the Problem Domain

There are two key characteristics of the problem domain in a distributed computation: the degree to which it can be parallelized and its granularity. Distributed computing platforms are capable of solving any problem that can be run in parallel. Often this will mean an SIMD-style computation. Problems outside this classification can often be parallelized via new techniques, as is the case with protein folding. The key characteristic is that the computation, or job, is easily divided into tasks small enough to be solved by a PC in a ``reasonable" amount of time. "Reasonable" will typically be defined on the order of several hours of CPU time.

The individual tasks are independent of one another, and consist of one or more operations, the smallest independent subunit of job execution. The granularity of a job is determined largely by the characteristics of the associated operations. Some jobs require tasks that consist of relatively few operations. Each operation may then take a relatively long time to complete, and the granularity is therefore quite large. Other jobs require tasks consisting of a large number of shorter operations. The associated granularity here is quite small. Take, for example, a prime search. Here an operation is defined as determining the primality of a single candidate. This operation is relatively long, and a task would therefore consist of a few hundred operations. Consider, however, a search for a DES encryption key. Here an operation is defined as the test of a single candidate key. The operation is much smaller, and therefore the corresponding task can consist of hundreds of thousands of operations.

One key characteristic of a problem is whether or not it consists of a series of sequential jobs or not. Protein folding and GIMPS, as mentioned earlier, are sequential problems. Cryptographic key searches, by comparison, are not. A sequential problem magnifies the damage caused by a saboteur's fake positive, since further tasks are at minimum partia lly dependent upon that result and are thus significantly thrown off. The results returned by participants in such a computation are typically more difficult to verify as well. Thus not only are false positives more damaging, but they are more difficult to prevent. Lastly, sequential computations often require special techniques in order to run tasks in parallel and take advantage of a distributed computing platform, adding complexity to the task at hand.

Many problems can be grouped under the heading "Inversion of a One-Way Function". These problems include public key searches. The critical observation here is that these functions are eminently checkable. One-Way functions normally carry the property that, given the right information, the one-way operation is easy to conduct. Thus when a result is returned, the supervisor of the computation can easily verify the correctness of the result. This essentially nullifies the efforts of the cheater, since each result returned can be verified at acceptable cost to the supervisor of the computation. However, given that the supervisor of such a computation would be tempted to check each result's veracity, an opportunity is created for a saboteur. Indeed, this class of problem becomes susceptible to a denial of service attack. Because each result can be verified, a saboteur can affect the supervisor by returning a large number of significant results. They will be verified as incorrect, but the supervisor of the computation is forced to check the results in order to realize this. When collusion is entered into the equation, this attack can be very effective. Consider also that often computations in this class will have a single correct result. This opens the computation to a significant disadvantage in that an adversary, specifically a saboteur, can simply withhold the correct answer. This forces the supervisor of the computation to resend the job, effectively doubling its cost. Sure, should the correct result be identified on the second transmission the saboteur could be identified from the first transmission, but recall that there are really no viable means of punishment for a saboteur.

Another class of problems addresses optimization issues. DNA sequence comparison is a prime example of such a problem. These applications adapt well to distributed computing. Optimization requires some sort of "goodness" scoring technique. This means that these applications are inherently *very* tunable, allowing the supervisor of the computation tremendous flexibility in terms of which results are returned. This allows the supervisor to vary the closeness with which he monitors his participants. Furthermore, there are often several solutions that will come close to the desired result. Thus if the supervisor is willing to pay the cost of a limited amount of post-processing, he can assure that the platform remains resistant to a limited degree of tinkering. Lastly, optimization problems are well-suited for data confidentiality schemes, since the important result is selection of the correct data element as a best optimization. Application specific schemes can achieve this result without revealing the actual test data. Look for new results on this idea soon from Dr. Doug Szajda. Lastly, pattern matching applications are a sort of subset of optimization problems, but with the key characteristic that they most often lack structure to their data sets. This affords the bonus of being able to implement ringers effectively.

Model	Attribute	Advantage	Disadvantage
Platform	Internet	Decreased likelihood of collusion	Greater variation in connectivity
	Enterprise	Lesser variation in connectivity.	Increased likelihood of collusion.
	Hybrid	Middle-ground	Middle-ground.
Problem Domain	Sequential		- Disruption more damaging - Disruption more difficult to verify
	Non-Sequential	Disruption affects fewer participants	
	Inversion of a One-Way Function	Results easily verifiable	Open to DOS Attacks
	Optimization	- Natural resistance to disruption - Potential for Data Confidentiality - Ringers schemes implementable	

Figure 4.1

5 Implementation Matters

It is important when considering the efficacy of a security scheme, that the details of implementation be evaluated. Too frequently in security, fundamentally secure elements are combined in an insecure way, and this theme carries over to security in distributed computation. The ability to blacklist is an important consideration. Connection of human and virtual identity is a difficult task. The structure of the supervisor's server is also of paramount concern. Finally, care must be taken that data transmission does not counteract the effectiveness of schemes such as ringer inclusion.

5.1 Blacklisting

A logical reaction to detection of a cheating or sabotaging participant is to blacklist that participant from the computation in future cycles. Unfortunately this isn't practical. Sarmenta [15] notes several reasons for this unfortunate reality. The two most likely candidates for participant identification are IP and email addresses. Given the technical ability of the assumed attacker, however, spoofing an IP address is relatively simple. This is not even necessary if the adversary has access to a large number of IP addresses a network administrator perhaps). Some *legitimate* partic ipants, furthermore, might receive IP addresses dynamically. Similarly email addresses are available from sources such as yahoo [16] for free and in unlimited quantities. It is true that RTTs, or reverse Turing tests such as those produced by the CAPTCHA project at Carnegie Mellon University [16] can prevent the programming of a bot to automatically accumulate addresses. However a determined adversary would have no problem obtaining a significant number of email addresses, and hence different participant identities, with the dedication of only a few hours of mundane point an click on the Internet.

Requiring more personal, detailed information poses the threat of turning away many potential and much need participants. Large amounts of personal information, however, are required in other walks of life; perhaps utilizing other institutions people generally involve themselves with would be efficient. Take for instance, financial institutions. Suppose a participant were required to give name, address, and bank. Because the financial world is set up in such a way that a person's identity is difficult to fake, the distributed computation would have a way of identifying participants that was consequently difficult to falsify. Similarly, use of biometrics would achieve this aim. Unfortunately, of course, biometrics remains some time away from ubiquitous practicality. The fundamental problem here is that there is no way to link human identity to cyber identity, period. Until this problem is solved, it will always be possible for a sophisticated opponent to secure a falsified identity.

5.2 Natural Ordering

There are two issues that arise when considering how data is actually transmitted to participants. The first concerns natural order that may occur in data sets sent to participants. For instance, in order to reduce transmission cost, it is a common practice to impose a total ordering on the data. Subset begin and end points can then be used as the sole means of defining a data set to be computed by a participant. The problem here is that this practice completely nullifies the supervisor's ability to implement a ringer scheme. Ringers [3] are pre-computed operations within a task that *should* return significant results. These are then included in tasks as partialexecution checks on a participant. With an ordering imposed, however, these ringers are immediately spotted by the adversary. Utilization of ringers that fit into the ordering is equivalent to pre-computing the entire task. Care must be taken in even the manner in which data is transmitted to participants or the computation may be compromised.

5.3 Push vs. Pull

There are interesting implications due to the method of task distribution. Specifically this revolves around the issue of whether or not tasks are "pushed", or sent out by the server primarily, or "pulled", automatically doled out to the participants as requested. Pull servers are inherently more efficient, but allow the adversary yet another tool. A sophisticated attack could play to the pull property in order to get duplicate copies of redundantly assigned tasks. Thus even the dynamic of task distribution must be carefully considered.

6 Collusion Resistant Redundancy

Given the challenges in terms of implementation, it is possible to significantly improve on the security of a computation by addressing the way that redundancy is utilized. Simple redundancy constitutes the most obvious manner of assuring that results are computed accurately. In short, simple redundancy allocates each task to two participants and thus a check is inserted on each task. This technique is subject to several shortcomings. The supervisor of the computation is immediately saddled with twice the compute cost since each task is being sent out to multiple participants. Double the cost, of course, might be considered acceptable if the result was a significant improvement in security. Consider, however, that simple redundancy is particularly susceptible to collusion. Specifically, if an adversary controls both participants to whom a given task is assigned, then that participant has complete control over the task. He can return significant results (whether legitimate or not) or fail to return significant results as he pleases. Indeed, in the case of an encryption-key search computation, if an adversary controls both participants to whom is assigned the key-space containing the encryption key, the entire job would need to be redistributed. Now the supervisor is faced with four times the original cost of the computation. It is clear then that simple redundancy does not protect well against collusion in terms of result robustness, surely a technique that is so expensive must offer some other benefit, detection of colluding adversaries, for instance. This too is not the case. Even if simple redundancy does indeed detect a cheater or saboteur, it offers no way to catch all other participants under adversarial control.

There is a better way to implement redundancy. Although they come at the cost of increased task tracking on the part of the supervisor, the following array of redundancy strategies allows the supervisor to increase the likelihood of detecting an adversary and to detect all participants under control of a colluding adversary after detection of one participant. Twice the cost should produce twice the benefit!

Consider the following tasks to be evaluated.

Figure 6.1

Task A	Task B	Task C	Task D
A1	B1	C1	D1
A2	B2	C2	D2
A3	B3	C3	D3
A4	B4	C4	D4
A5	B5	C5	D5
A6	B6	C6	D6
A7	B7	C7	D7

Then using simple redundancy 8 participants would be needed, and allocation might look like the following:

Figure 6.2

P1	P2	P3	P4	P5	P6	P7	P8
A1	*A1*	**B1**	**B1**	**C1**	**C1**	**D1**	**D1**
A2	*A2*	**B2**	**B2**	**C2**	**C2**	**D2**	**D2**
A3	*A3*	**B3**	**B3**	**C3**	**C3**	**D3**	**D3**
A4	*A4*	**B4**	**B4**	**C4**	**C4**	**D4**	**D4**
A5	*A5*	**B5**	**B5**	**C5**	**C5**	**D5**	**D5**
A6	*A6*	**B6**	**B6**	**C6**	**C6**	**D6**	**D6**
A7	*A7*	**B7**	**B7**	**C7**	**C7**	**D7**	**D7**

6.1 Vertical Partitioning

As shown prior, simple redundancy suffers from significant shortcomings. By changing the granularity at which redundancy is implemented, however, new possibilities become apparent. In simple redundancy the above tasks were treated as units. Suppose they are subdivided as shown follows.

Figure 6.3

P1	P2	P3	P4	P5	P6	P7	P8
A1	*A1*	A2	A3	A4	A5	A6	A7
A2	*B1*	B1	B2	B3	B4	B5	B6
A3	*B2*	B7	B7	C1	C2	C3	C4
A4	*B3*	C1	C5	C5	C6	C7	D1
A5	*B4*	C2	C6	D2	D2	D3	D4
A6	*B5*	C3	C7	D3	D5	D5	D6
A7	*B6*	C4	D1	D4	D6	D7	D7

There are several properties about this scheme that are of note. Essentially the work of each participant is checked by *every* other participant in the computation. This is tremendously effective. The granularity of the redundancy has been changed, and it is easily demonstrated that with the proper algorithms, the scheme can actually be used to prevent cheating and sabotage.

Consider the possibility of checking each matching subtask in the computation. Because this consists of simple comparisons, it is extremely efficient. In fact, it is very likely to be more efficient than re-computation of even a single subtask. This method is easily summarized in the following algorithm.

Algorithm 6.1

```
Preliminary()
    For( Each participant ) {
        For( Each subtask )
            if( not checked )
                    verify redundant subtask
                    If( results conflict ) Flag Participants
```

Recall that cheating is defined as an attempt to gain credit for work not performed. It is clear that cheating is eliminated if you consider a cheater's possible strategies. Should a cheater control only a single participant, then attempting to return a result without actually performing the computation is fruitless, since the server immediately identifies the erroneous results. Every subtask has been checked by another participant. Now examine the instance in which a cheater has control of several adversaries. As previously mentioned, he is able to determine precisely those tasks that he has been redundantly assigned. By returning matching results he eludes detection without having performed the computation. However, he is still forced to compute the remaining subtasks or face being caught. Furthermore, the compute expense associated with determining which subtasks are redundant is likely to nullify the benefit of cheating on them. Thus algorithm 6.2 successfully eliminates cheating.

Simple result matching does not, however, solve the problem of sabotage. An adversary whose goal is to undermine the computation has a simple yet effective strategy. As stated above, the task of determining which subtasks a pair of participants have been assigned redundantly is easily accomplished. With this information in hand, the adversary has only to return erroneous, yet matching results for its redundantly assigned subtask, and he has successfully undermined the computation. Can this new form of redundancy affect an adversary's ability to collude? Consider the following algorithm:

Compute the equivalent of 1 task, by computing 2N - 1 subtasks. In figure 6.2, for example, one could compute all of task A, or a combination of subtasks from all 4 tasks. The key is that every participant has at least one subtask that will be computed and checked by the supervisor. The following algorithm can then be used to identify all colluding adversaries.

Algorithm 6.2

```
    Search(){
        For(each computed subtask){
            If(returned results are wrong and match)
                Mark participants as adversaries
        }
        Now, choose the first adversary found, call him A
        For(each subtask assigned to A){
            Compute subtask
            Check subtask
            If(subtask doesn't check)
                Mark new adversary
        }
    }
```

The thing to notice here is that it is never necessary to compute more than 2 tasks, and it can be guaranteed that all of the adversary's participants will be determined. As N grows larger, computation of 2 tasks becomes less of an expense. Algorithm 6.3, of course, has a determined probability of catching a sabotaging adversary, as computed by Szajda et. Al. in Appendix C.

6.2 Horizontal Partitioning

It quickly becomes obvious that vertical partitioning is not scalable on the order of a large-scale distributed computation. In fact, it becomes unreasonable for values of N greater than approximately 50 [4]. Horizontal Partitioning allows dynamic tuning of the methodology in order to scale to any application.

Consider a job consisting of M tasks. Horizontal partitioning divides the participant pool of $2M$ *participants* into C clusters . In each cluster vertical partitioning will be applied to the N tasks in that cluster, distributing them to $2N$ participants. The relationship is thus

Equation 6.1 $C = M / N$

Consider simple parameters. Let M = 4, N=2, and thus C=2. The 4 tasks will be denoted A,B,C, and D, and each will be split into 2N-1=3 subtasks. The resulting distribution would be as follows:

Figure 6.3

P1	P2	P3	P4	P5	P6	P7	P8
A1	A1	A2	A3	C1	C1	C2	C3
A2	B1	B1	B2	C2	D1	D1	D2
A3	B2	B2	B2	C3	D2	D3	D3

Cluster A Cluster B

This partitioning method, in addition to its complete scalability, addresses the issue of collusion more appropriately in that a pair of adversaries is no longer guaranteed to have a pair of subtasks in common, unless of course they are part of the same cluster. Horizontal furthermore offers the supervisor flexibility. The parameters can be changed at will within the constraint of Equation 6.1. In considering the horizontal partitioning method, a disadvantage should be pointed out. One cannot guarantee the successful determination of all adversarial participants because they will not all have redundantly shared subtasks.

Lastly, according to the mathematical proofs given by Szajda et. Al [4] the following properties hold:

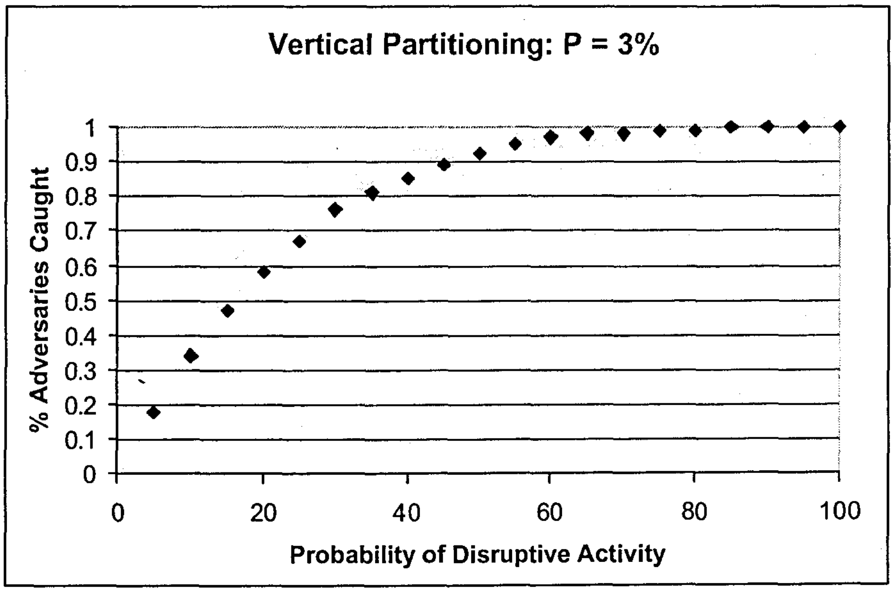
- For a given proportion of participants controlled by the adversary, the expected number of tasks controlled by the adversary is identical under each of the strategies.
- The stability of the strategy is superior in the sense that the variance of the number of tasks controlled by the adversary is greatly decreased.
- Given that an adversary with matching tasks or subtasks will disrupt the computation a consistent proportion of the time, the probability of detecting this activity is significantly improved over that achieved through simple redundancy.

6.3 Adversary Detection Analysis

Absent the availability of an actual distributed computation implementation, computer simulation offers the most effective means of gaining hard data on the efficacy of a scheme. Just such a simulation was developed to test the efficiency of the vertical partitioning method. This simulation was completed using the C programming language.

There are several variable elements to consider. First the proportion of the participant pool under the control of an adversary. This indicates either one human identity in control of multiple participant usernames, or multiple human identities colluding. The resulting situation is the same in either case. The second variable to consider is the proportion of the time that, when given the opportunity, an adversary will attempt to sabotage the computation. It would be useful to know what percentage of the time these adversaries are caught. Figure 6.4 depicts these figures when percentages of the population equal to 3, 6, 9, and 12 are controlled by the adversary. Interestingly, at 12% of the population, every attempt to disrupt the computation is caught. Even at 3 percent, a high percentage of sabotage attempts are foiled.

Figure 6.4



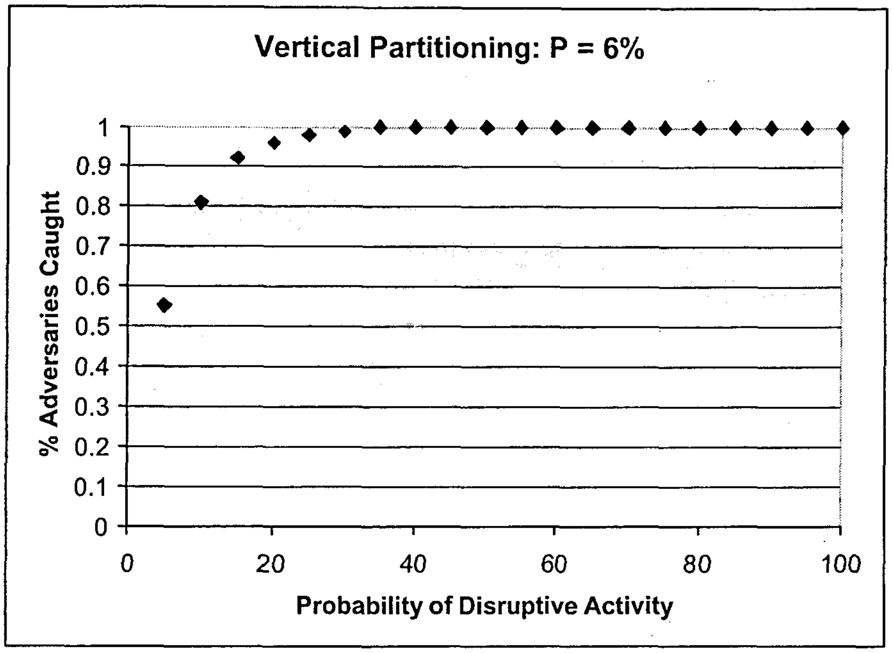
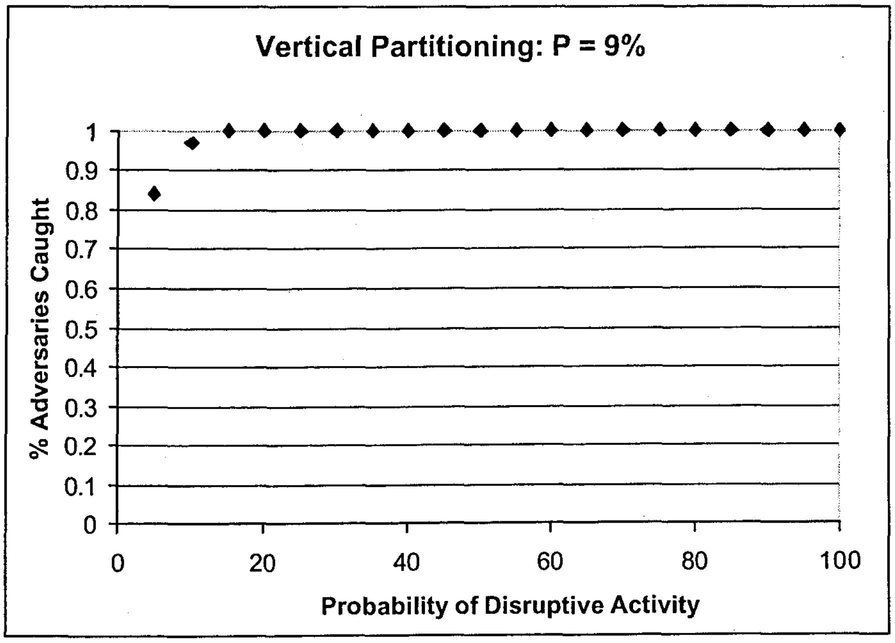
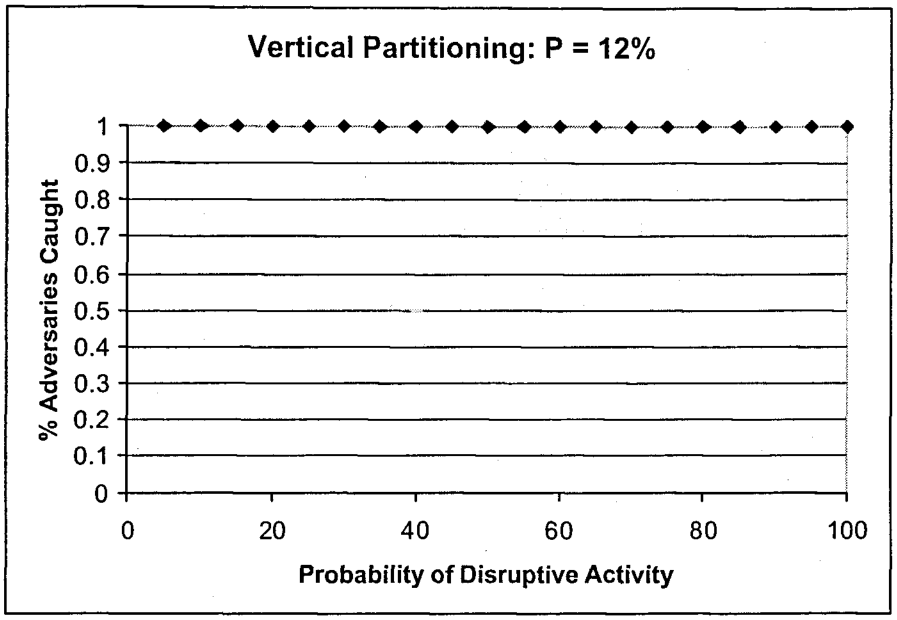


Figure 6.4 Continued





Consider, furthermore, that horizontal partitioning constitutes multiple implementations of vertical partitioning to allow scaling. Note also that Szajda et. al [4] proves that the proportion of tasks controlled by the adversary in both vertical and horizontal partitioning is the same, thus these results hold for horizontal partitioning as well.

The paper entitled "Collusion Resistant Redundancy", included as Appendix B in this paper, provides a further, formal treatment of horizontal and vertical partitioning.

8 Related and Future Research

There are two topics of potentially great interest to distributed computation security. The first is the similar, though unique problem of mobile agent security. The second concerns the implications of running distributed computing platforms on wireless networks, an area sure to see substantial interest in the near future.

Mobile agents are autonomous code fragments with the ability to move about a network. They are able to move around this network in order to perform checks at each node, to collect information, and to perform analysis. Agents can even be capable of communicating with one another, thus creating the possibility of online auction and bidding scenarios. The potential is truly endless. While there have been effective solutions produced to protect hosts from malicious agents, the dilemma of how to protect an agent from a malicious host has gone unsolved. This is a difficult problem. Agents are fundamentally dependent upon the hosts they visit to provide an execution environment, but remain unable to guarantee that that execution environment will act as expected. Furthermore, how is an agent to maintain the secrecy of the data that may be necessary for the completion of the task code at the host.

Agent technology is interesting in the context of distributed computations, because a solution to the quandary of securing mobile agents from malicious hosts would offer a potential solution to the problem of securing distributed computations. The computation code would simply be run by an agent and then returned to the supervisor. Thus, the study of proposed mobile agent security solutions is appropriate in this context.

There has been a great deal more research into this effort than there has been in the realm of distributed computation security, yet the problem remains unsolved. It is important to note in closing on this subject, that security in distributed computation *is* different than securing mobile agents against malicious hosts. There are many options open to the computation supervisor not open to the controller of a mobile agent, many of which have been detailed in this paper. It is a worthwhile reference, however, as the topics do have that distinct connection, and advances in one could very well provide quality insight into new security mechanisms in the other.

A great deal of future research in distributed computation may well center around wireless settings. The proliferation of PDA's and cell phones has created a vast and untapped resource. Issues with this medium will include the often weak processing power and memory capacity on these devices. Processor speeds on personal computers are such that significant processing ability can be achieved through the aggregation of a reasonable number of machines. Of course, if Moore's law applies to these devices, the significance of their processing ability may well increase with great rapidity. Another issue is battery life. Length of charge is one of the most crucial elements of a mobile device in terms of determining its usefulness to the consumer. Operating these mobile devices at close to capacity could put a significant dent in battery life, regardless of the enormous amounts of energy that full-color LCDs take up themselves. Finally, mobile devices are inherently less dependable in terms of connectivity. These concerns and others will need to be addressed if the wireless arena is to become conducive to distributed computation. Given the current rapid proliferation of these devices, these questions are almost certain to be asked.

9 Conclusion

The potential of distributed computation is enormous. Continued increases in the already tremendous processing power present in personal computers will only augment this potential. Ameliorating the difficulties currently present in the wireless arena and increased connectivity, too, will only increase the sum computational power available all over the globe. Given this bright outlook it is imperative that research be continued in the area of securing these computations. As potential compute power increases, so too will a desire to profit from this ability. Increased commercialization of the technology will only invite increased attacks against these systems. Full defense of a distributed computation is not a simple affair. Consider, though, the six security tenets which a supervisor hopes to achieve: data secrecy, participant authentication, data integrity, cheating resistance, disruption resistance, and data confidentiality. Given that the first of these three is adequately addressed via a PKI implementation including digital signatures, the task seems more tractable. Consider also that the Algorithm 6.1 effectively eliminates the effort of a cheating adversary with acceptable cost, and that collusion resistant redundancy improves the effort to achieve disruption resistance. It is these last two elements that prove most difficult. Application specific techniques are possible in achieving data confidentiality, but in short, disruption prevention and data confidentiality are still in need of further research.

This paper does, however, provide a means to examine a particular distributed computation and determine which types of attacks it, in particular, is subject to, as well as the options that are open in terms of security protocol, and to guide the security mechanisms that are put in place. Furthermore, redundancy *can* be implemented in a much more useful manner, and indeed improves upon the security of a computation by allowing the supervisor to reliably identify adversary controlled participants. The inability to blacklist shall remain a thorn in the distributed computation supervisor's side, but awareness of which participants are malicious does allow that supervisor to take non-expensive precautions towards that adversary.

Securing a large-scale distributed computation requires careful thought and analysis, ever keeping in mind that in security, it is often the implementation details that bring down the system. It is, however, a tractable problem, and will only benefit from future research to come.

References

[1] R.L. Rivest, A. Shamir, and L. Adleman, A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*. Volume 21, Issue 2: pages 120-126, February 1978.

[2] D. Szajda, B. Lawson, J. Owen, and E. Kenney. Issues in Securing Large-Scale Distributed Computations. Submitted to Proceedings of the 2004 ISOC Network and Distributed System Security Symposium.

[3] D. Szajda, B. Lawson, and J. Owen. Hardening functions for large-scale distributed computations. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy,* pages 216-224, Berkeley, CA, May 2003.

[4] D. Szajda, A. Charlesworth, B. Lawson, J. Owen, and E. Kenney. Collusion Resistant Redundancy. Submitted to Proceedings of the 2004 *IEEE Symposium on Security and Privacy.*

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

[6] Bojan Zagrovic, Eric J. Sorin, & Vijay S. Pande. Beta Hairpin Folding Simulations in Atomistic Detail Using an Implicit Solvent Model. *Journal of Molecular Biology*, 313(2): 151-169, 2001.

[7] Michael R. Shirts, Vijay S. Pande, Mathematical Analysis of Coupled Parallel Simulations. *Physical Review Letter*, 86 (22), May *2001.*

[8] Stefan M. Larson, Christopher D. Snow, Michael R. Shirts, and Vijay S. Pande, Folding@Home and Genome@Home: Using Distributed Computing to Tackle Previously Intractable problems in computational biology. *Comutational Genomics, 2002.*

[9] Lawrence Hunter, *Molecular Biology for Computer Scientists*.

[10] Michael R. Shirts, Vijay S. Pande, Atomistic Protein Folding Simulations on the Submillisecond Time Scale Using Worldwide Distributed Computing, *Wiley*, 86(22), May 2001.

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

[12] N. Draper and H. Smith. *Applied Regression Analysis.* John Wiley & Sons, 1966.

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

[14] T.F. Smith and M.S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology.* 147: 195-197, 1981.

[15] L. Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. In *Proceedings of the ACM/IEEE International Symposium on Cluster Computing and the Grid,* Brisbane, Australia, May 2001.

[16] CAPTCHA Project, Carnegie Mellon University, Pittsburgh, PA.
http://www.captcha.net

[17] Yahoo
http://www.yahoo.com

<u>APPENDIX A</u>

Issues in Securing Large Scale Distributed Computations

Issues in Securing Large-Scale Distributed Computations

Doug Szajda Barry Lawson Jason Owen Ed Kenney
University of Richmond
Richmond, Virginia
{dszajda, blawson, wowen, ekenney2}@richmond.edu

ABSTRACT *Many recent large-scale distributed computing applications utilize spare processor cycles of personal computers that are connected to the Internet. These computations run in untrusted environments, raising a number of security concerns, including the potential for disrupting computations and for claiming credit for computing that has not been completed. Though security research specific to these computations has increased in recent years, there has been little consideration of the ways in which application and platform structure and implementation attributes can influence the effectiveness of security mechanisms. In this paper we examine these issues using examples of actual implementations and proposed security solutions. We show in particular that the combination of attributes present in some current platform implementations creates a situation in which collusion is not only feasible but easily achieved. Because security measures based on redundancy are weak in the presence of collusion, applications protected by such measures (i.e., most current implementations) may be less safe with these mechanisms than without.*

1 Introduction

The advent of large-scale distributed computing platforms, consisting of many personal computers connected to the Internet, provides researchers and practitioners a new and relatively untapped source of computing power. By utilizing the spare processing cycles of these computers, computations are now possible that were once unobtainable without the use of a supercomputer. In a typical distributed computation, the computation is easily divisible into independent *tasks*, each of which can be processed by a typical personal computer in a few hours. A *participant* downloads code from the *supervisor* of the computation in order to establish an execution environment in which the supervisor can execute tasks. Each task is assigned and dispatched to a participant, and upon completion of the task significant results are returned to the supervisor. In this context, providing a level of assurance for results is difficult because the results are obtained by executing tasks in untrusted environments. Participants can intentionally or unintentionally corrupt results, and can attempt to claim credit for work not completed. The supervisor can validate results by assigning each task to two or more participants, but as a result at least half of the processor cycles are lost to the validation process.

In recent years, a small but growing collection of papers has emerged that cover various security issues specific to these computations. Golle and Mironov [19] 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 [20] present a security based administrative framework for commercial distributed computations. Monrose, Wyckoff, and Rubin [27] propose instrumenting host code in order to generate lightweight execution traces that can be used to verify program execution. Most recently Sarmenta [32] and Szajda, et al. [36] present probabilistic verification mechanisms that increase the likelihood that an attempt to disrupt or cheat a computation will be detected.

Collectively, these papers exhibit two common characteristics. First, the papers generally treat applications as abstractions, rarely discussing structural and implementation details. This is unfortunate because, as we show here, these details can significantly impact both the performance of an application and the ability of a proposed mechanism to secure the computation. Security mechanisms that work well in theory may prove difficult or costly in practice — the inherent structure of an application, the implementation of that application, and the platform on which it executes must all be considered. Second, all the papers rely to a certain extent on redundancy as a means for securing computations. We agree that, in some cases, redundantly assigning tasks may be the only feasible security mechanism. However, there are other cases for which redundancy may be inefficient and/or inappropriate — redundancy or self-correcting features may be inherent in the structure of the application itself. Moreover, the use of redundancy as a means to secure a computation can unwittingly open the computation to other attacks, most notably via collusion. The intent is not to discourage the use of redundancy, but instead to raise awareness to the ramifications that accompany redundancy.

The contribution of this paper is to bring to the forefront issues unique to large-scale distributed computations that, to date, are rarely considered. In particular, through examples of real implementations and proposed security mechanisms, we examine

- how the specific structure of an application can influence the efficacy of various security measures;
- how platform and application *implementation* subtleties can influence the efficacy of proposed security solutions;
- how some common assumptions, such as that of a low probability of collusion, are misgiven; and
- how computations that use redundancy for assurance may be relying on false assumptions.

Protection measures that fail to account for these issues can leave a computation vulnerable to attack.

To our knowledge, this is the first paper that considers application and platform specifics and how they relate to securing these computations. In the process of our discussion we present several attacks that are unique to these computations, point out the application and platform implementation decisions that create vulnerabilities to them, and discuss ways of providing increased resistance to these attacks. Our purpose in discussing these attacks is not to provide a roadmap for adversaries. Rather, we wish to create an awareness of these issues at this relatively early stage in the study of securing such computations, and to provide a common and realistic set of assumptions on which future research efforts can be based.

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.

The remainder of the paper is as follows. In Section 2 we present a model of the distributed computations and of the platforms under consideration. Section 3 discusses related work. Section A provides examples of applications amenable to the distributed metacomputation technique. The examples presented here describe existing implementations, and serve as the basis for our observations in the remaining sections. Sections 5 and 6 discuss applications and platform characteristics that can influence achievable levels of security, and some attacks specific to these computations are presented. Section 7 discusses some of the problems inherent in securing computations using redundancy. We present our conclusions in Section 8.

2 The Model

We consider parallel computations in which the primary computation, the *job*, is easily divided into *tasks* small enough to be executed by a PC in a "reasonable" amount of time, which may depend on the specific application. Tasks are independent and consist of one or more *operations*. Tasks can consist of a single extended operation, such as testing the primality of a Mersenne number, or a large number of smaller operations, as in a brute-force search for a cryptographic key.

The computing platform consists of a trusted server coordinating as many as several million personal computers in a "master-slave" relationship. These *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. Communication required for a computation is necessary (and permitted) only between individual participants and the supervisor. Participants may receive remuneration in a variety of forms for completing their assigned tasks.

[1]The term *participant* denotes both the node and the user depending on context.

With respect to any distributed computation, 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 task or execution environment code. In particular, the adversary has exact knowledge both of the algorithm used for the computation and of the security measures used to circumvent attacks. An adversary will attempt to disrupt the computation in one of two ways:

- the adversary will attempt to *cheat*, i.e., obtain credit for work not performed;
- the adversary will attempt to *sabotage* the computation by intentionally returning incorrect results or by intentionally failing to return significant results.

Additionally, we assume that collusion among adversaries is likely (see Section 6.4).

The reasons motivating an attack can vary based on many factors [9, 10]. For platonic applications (e.g., SETI@Home, Folding@Home, etc.), maximizing credits obtained (for recognition or for relatively small monetary remuneration) may be sufficient motivation to cheat. For business related applications (e.g., Monte Carlo financial modeling), an adversary may hold competitive business interests and therefore be intent on sabotaging the work. Moreover, the motivations for an attack may be unpredictable and/or unexpected, such as an attack by a quietly disgruntled employee. Any attempt to *categorize* the motivations of a potential adversary would therefore be incomplete at best. However, the realization that unknown adversaries exist with unknown motivations is paramount for achieving a well-secured computation. Furthermore, it is easy to dismiss certain potential attacks as irrational and therefore unlikely. The supervisor of a computation would be wise to instead consider *any* potential adversary as rational, except that the utility function of that adversary may or may not be known.

As the previous discussion highlights, many of the assumptions concerning a potential adversary are based on speculation. We strongly believe that a comprehensive study is needed that determines the type and frequency of occurrence of the attack strategies encountered in practice — i.e., a demographic of disruptions. For example, because the utility function of an adversary is unknown, the owner of a computation cannot determine a priori the frequency with which an adversary will cheat or attempt to sabotage the computation. Unfortunately, to date there is insufficient empirical data upon which to base a realistic threat model. As such, the assumption that an adversary will cheat according to a simple probability distribution is unlikely to accurately model an attack strategy of an adversary in practice. Further study, via experimentation and available statistical information, is therefore necessary to more accurately construct a realistic adversary framework.

3 Related work

Several recent implementations of distributed computing platforms address the general issues of fault-tolerance [7, 8, 11, 13, 29, 31], 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. [26] 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.

Golle and Mironov [19] study computations involving inversion of a one-way function (IOWF). These applications seek the pre-image x_0 of a distinguished value y_0 under a one-way function $f : D \rightarrow R$. In the absence of redundancy, only a single task contains the desired preimage providing little chance of detecting an adversary who falsely claims to have completed work. Golle and Mironov present several variations on a basic *ringer* scheme, in which values of f are precomputed and the results planted in task data spaces. Since participants are not able to distinguish which data corresponds to ringers, the probability of detecting cheating is increased. Because their methods focus on a single class of application they are naturally application specific. Neither application nor platform design and implementation issues are discussed.

Golle and Stubblebine [20] present a security based administrative framework for commercial distributed computations. Their methods rely on selective redundancy to increase adversary detection probabilities. Their framework provides flexibility by allowing the distributions that dictate the levels of redundancy to vary. They do not address application or platform specifics, focusing instead on a game theoretic model based on estimates of a participant's utility of disrupting the computation and cost of being caught defecting.

Monrose, Wyckoff, and Rubin [27] consider methods of assuring host participation in computations, assuming that users wish to maximize their profit by minimizing resources. The method requires instrumenting task code 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. Aside from considerations directly related to their instrumentation, such as the necessity of having tasks that can be transformed into checkable units with similar execution times, application and platform characteristics and implementation details are not discussed. Unlike the other articles discussed here, their methods do not rely on tasks being redundantly assigned. Rather the burden of checking proofs of execution fall to the server.

Szajda, Lawson, and Owen [36] present two general schemes for using probabilistically applied redundancy to give applications greater resistance to cheating. They divide applications into two broad classes: non-sequential, in which tasks consist of independent operations; and sequential, in which the operations that constitute an individual task can have dependencies and must be executed in a specific order. Their technique for non-sequential applications is essentially an extension of the Golle and Mironov ringer scheme to more general functions. They handle sequential applications by breaking computations into several stages, assigning N tasks to $K > N$ participants, and using probabilistic verification. Some application and platform specifics are mentioned, though only briefly and only in the context of applicability of their methods. They mention the possibility of colluding adversaries but assume that this occurs with low probability.

Sarmenta [32] proposes a credibility-based system in which multiple levels of redundancy are used, with parameters determined by a combination of security needs and participant reputations. Sarmenta notes that 6.3, blacklisting of participants is not possible in most distributed computations. Application and platform implementation specifics are not discussed. Collusion is considered during analysis of his system, but as in [36], the probability is assumed to be low.

There is also a body of literature related to the question of protecting mobile agents from malicious hosts (see e.g. [21, 22, 23, 30, 39, 38]). Because a primary difficulty in both problems is that of executing code in untrusted environments, it is natural to assume that potential solutions to either might be applicable in some form to the other. For future solutions this may be the case, but there are important differences that make the few solutions available for protecting malicious hosts difficult to apply to distributed metacomputations. First, many mobile agents execute in untrusted environments because they require data that is available only in those environments. Tasks in distributed metacomputations, on the other hand, possess all data necessary for complete execution, and therefore can be checked by the computation supervisor. There is thus the potential for solution in the metacomputation context that would fail for mobile agents. Mobile agents typically visit hosts for a relatively short period of time, whereas metacomputation task code may require several hours of running time. Thus time limited obfuscation approaches such as those advocated by Hohl [22, 23] are not applicable in the present setting. Finally, metacomputations deal with issues of scale not present in current mobile agent systems.

4 Examples

We use as examples five currently implemented applications: the exhaustive regression and Smith-Waterman sequence comparison implementations written for the Parabon Computation, Inc. Pioneer distributed computing platform; the protein folding implementation used by Folding@Home; the Search for Extra-Terrestrial Intelligence project conducted by SETI@Home; and the Great Internet Mersenne Prime Search (GIMPS) application developed by Entropia.com. Details of each of these examples are provided in Appendix A. A brief description of each is presented here.

Regression [14] is a technique for fitting experimental data to a (possibly N-dimensional) function of the form

$$y = a + b_1x_1 + b_2x_2 + \ldots + b_Nx_N,$$

where $x_1, x_2, \ldots, x_N$ represents the predictor variables and y is the response variable. The coefficients a and $b_1, \ldots, b_N$ are determined via least squares methods [14]. The goal is to find the smallest subset of predictor variables that significantly influences or determines the value of the response variable. Exhaustive regression addresses this goal using a brute force method. All possible combinations of predictor variables are tested in order to determine the single combination of predictors that produces the "best" fit.

Folding is the process by which a protein, consisting of a string of amino acids, takes on the physical configuration that gives the protein its biochemical functionality. A protein's folded structure is predetermined by its amino-acidic composition, but determining this configuration *a priori* is quite difficult. A protein folds in such a way that the total free-energy of the resulting structure is minimized. To solve this minimization of free energy problem, Folding@Home [15], a distributed computing project run by the Pande Group at Stanford University, transmits *molecular dynamics* (MD) *simulations to thousands of participants. Each MD simulation is initialized with the* same set of coordinates describing the locations and initial velocities of atoms, but each participant simulates a different fold by incorporating random thermodynamic forces [43]. The process of folding takes a protein through several metastable states before producing a final configuration. When any task transitions to a new state, all other tasks are restarted using the molecular coordinates of the transitioned task. Each realigned task continues using a unique random sequence of thermodynamic forces acting upon the molecules, allowing coverage of the entire configuration space.

Smith-Waterman [35] is a sequence comparison algorithm that utilizes a dynamic programming technique to find similarities between strings of DNA nucleotides or amino acids. Given a sequence, the algorithm is typically used to locate in a large database (e.g., from NIH) the sequence that most closely matches the given sequence. Furthermore, one may consider segments of each sequence that are most similar. Because the number of different sequence alignments is enormous, significant compute power is required to execute the dynamic programming algorithm, making Smith-Waterman amenable to distributed computing platforms.

The SETI@Home project [6, 25, 1] attempts to detect signals of extraterrestrial origin. Their efforts require a significant amount of compute time because the parameters of extraterrestrial signals are unknown and because required processing costs rise in accordance with increased search sensitivity. Given the large number of potential parameters, tasks require from 2.4 to 3.8 trillion floating point operations — about 10 to 12 hours on a 500MHz PC. SETI assigns tasks redundantly (with a factor of 2 or 3) and performs cross checking. SETI@Home is by far the largest distributed metacomputation project in existence. As of this writing they have registered over 4.6 million usernames and have processed over one billion tasks representing almost 1.6 million years of total CPU time.

The Great Internet Mersenne Prime Search is an ongoing project run by Entropia.com [18]. 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, but the primality of n is not a sufficient condition for the primality of M_n. The Lucas-Lehmer Theorem [12] states that M_n is prime if and only if $S(n-1) = 0 \pmod{M_n}$, where

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

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.

5 Application Specifics Matter

Failure to understand the peculiarities of a specific application can lead to difficulties when applying a security mechanism. In some cases, application characteristics are such that a proposed mechanism becomes ineffective without significant changes to the implementation. Often these subtleties are not obvious; in many cases, applications that appear similar may be affected in entirely different ways. Characteristics that can affect proposed security mechanisms include the ease with which task execution or results can be verified, structure inherent in the input data, and the form and content of returned results.

5.1 Inherent Structure in the Data

As an example of the kind of difficulty that can arise due to structure inherent in input data, consider a travelling salesperson application and the *ringer* mechanism proposed in [36] to handle optimization problems. An optimization computation involves finding the input (from an assumed huge set) that optimizes a specific objective function. Using the ringer mechanism, the supervisor of the computation is advised to assign a small fraction of the total input space redundantly, after which the best returned results are planted in subsequent task data spaces. An honest participant executing such a task will return each of these precomputed results as significant, thereby providing the supervisor with a partial execution check. Fundamental to this method is that the ringers are not distinguishable to the participant. In theory this is simple, but in practice there will likely be structure in the data that will expose the ringers.

Consider the implementation of travelling salesperson. A weighted (complete) graph representing cities and distances, along with the set of circuits to be checked, is transmitted to a participant. To keep communication costs low, the supervisor is likely to impose a total ordering on the data. In this way, endpoints of the data set can be sent rather than the entire data set itself (which could be very large). However, imposing a total ordering immediately exposes the ringers — if the ringers have been "injected" into a given task data space, they will not be a natural part of the ordering and will therefore be visible. On the other hand, finding good ringers that occur "naturally" in a particular task data space requires precomputing the entire task. For the implementation described, the ringer method is no better than assigning each task twice.

The implementation described above is relatively simplistic and inefficient. A great deal of redundancy can be eliminated if circuits are grouped carefully. For example, in a 50-city tour the circuits $\{1, 2, \ldots, 47, 48, 49, 1\}$ and $\{1, 2, \ldots, 47, 49, 48, 1\}$ differ in only two distances, so assigning them both to a single task would allow the cost of the path $\{1, 2, \ldots, 47\}$ to be computed only once. Regardless of the data structure detail necessary to implement such an idea, some inherent order or organization to the data structures will exist, thereby exposing the ringers.

The problem of exposing ringers is not unique to travelling salesperson — exhaustive regression suffers from the same difficulty. In Parabon Computation's implementation of exhaustive regression, the input data consists of bit strings, where each bit represents a single regressor. A regressor is included in a specific multiple regression if and only if the corresponding bit in the bit field is set. Rather than send several bit fields, a total order is applied by representing each bit field as an unsigned integer. In this way, bandwidth is saved by sending a delimiting start and end integer — the participant is instructed to compute using the bit fields corresponding to all integers between these delimiters. The result, however, is that the ringer strategy cannot be directly applied. If instead the data distribution scheme is changed to, say, one that simply sends each task an entire set of bit fields in some random order, increased communication costs and increased processing at the server result. Moreover, an adversary must not be able to recognize and predict the random ordering. Fortunately, the exhaustive regression input data format discussed here has the desirable property that any bit field of appropriate length is a valid input. By encrypting the randomly ordered bit fields, the server can implant ringers without fear of exposing the ringers or the random sequence used. As such, a participant can operate directly on an encrypted bit field, providing a security mechanism that addresses structure inherent to the data.

As a different example, consider a Smith-Waterman DNA sequence comparison application as described in Section A.3. As with the travelling salesperson problem, this application involves optimization. More specifically, the best matching pair of sequences is desired. The task input data consists of sequences over a small finite alphabet. In DNA sequence comparisons, however, there is no inherent order among the data, so ringers can be planted in task data without being exposed. Hence, we have shown that two optimization problems can, from a security standpoint, react to the ringer scheme in entirely different ways.

5.2 Return Values

There should always be something returned by a task, if only to provide notice to the server that the task has been completed. Progress reports, i.e., short notices that indicate the percentage of work completed and significant results obtained thus far, are also desirable. The reports can be used not only for measuring progress, but also for providing additional information that can be used to augment security mechanisms. For example, progress reports can be used to implement a more fine-grained version of redundancy. Rather than have two participants compute the same task, N tasks can be assigned to $N + 1$ participants, with N participants completing the entire assigned task while one computes the first $1/N$ of each of the N tasks. More generally, N tasks can be assigned in their entirety to N participants while various portions of the tasks are assigned to K additional participants. Assuming K is less than N, the *redundancy factor* (the number of participants assigned tasks divided by the number of tasks) can be significantly less than 2. Moreover, even if participants are aware that such partial redundancy is being used, they do not know how much of their own task is being checked, thereby providing an incentive to complete the entire task.

Progress reports also help minimize the impact of a type of denial of service attack unique to distributed metacomputations. Specifically, an adversary assigned a task can simply return nothing. There are many legitimate reasons why a task may not be returned. A participant may be busy with other work — task code typically runs either in the background or when the host computer has been idle for a fixed period of time. In addition, users with dial-up connections are only online intermittently, and can be offline for several days at a time. Thus the failure to return a result is not necessarily a malicious act. To address results not returned, most platforms enforce a timeout value, after which the task is reassigned to another participant. In the meantime, however, the new effect is that the work is not being completed.

In this context, an adversary controlling a single task poses little threat — the denial of service lasts only a few days. However, an adversary controlling several thousand tasks, using perhaps several hundred usernames, and continually adding to their collection of tasks and names becomes more than a nuisance. In addition, there is no mechanism to prevent reassigning the tasks to the adversary. We show in Section 6.4 that if platform implementations lack measures to prevent automated generation of usernames and task requests (and we know of no platform that currently has such measures), an adversary can easily acquire tens of thousands of tasks, more than enough to create a serious disruption

Progress reports minimize this attack by effectively reducing the required timeout length. One disadvantage of such an approach is the problem of an already overloaded server handling the resulting increase in traffic. Consider that SETI@home clients do not send progress reports. In the approximately 1600 days that SETI@Home has been operational, results for over one billion tasks have been received, yielding an average rate of more than 7 tasks per second. Because generating n progress reports per task means a corresponding n-fold increase in server load, progress reports may require an n-fold increase in communication and server capacity.

Requiring that all tasks generate at least one significant result is part of the motivation behind the various ringer schemes presented in [19] and [36]. In practice, if some variant of the ringer scheme cannot be applied, increasing the number of values returned from an application by lowering a threshold can be difficult. The tuning required to increase the number of return values is, in many cases, relatively coarse, and the danger exists that increasing the number of significant results may swamp the server. In addition, such tuning may not guarantee that all tasks will have significant results. For example, increasing the total number of significant results in an exhaustive regression application can always be achieved by lowering the threshold correlation. Short of setting the threshold to zero, however, lowering the threshold does not guarantee that each task will generate a significant result. Moreover, even slight changes to a threshold can have an overwhelming impact. During early testing of the Parabon exhaustive regression engine, tasks were assigned only 5000 multiple regressions each (estimated to take approximately an hour to execute). In one test, an assumed reasonable correlation threshold resulted in a overloaded server, with tasks transmitting more than 8MB of return data. Tuning return results for a Smith-Waterman DNA sequence comparison can be even more difficult — a short test run that precomputes sample similarity scores is often required to estimate a reasonable threshold. Furthermore, when an application is tuned to return more results, work is being moved from the tasks to the server because the returned results require postprocessing to remove extraneous "significant" results. In small numbers this may not seem an issue; in a case with 600,000 participants, the postprocessing quickly becomes problematic.

6 Platform Specifics Matter

In this section, we consider how characteristics of the design and implementation of the distributed computing platform can influence the security attributes of a system. In particular, we address task allocation mechanisms, connection characteristics, and parameter settings. We also consider *participant demographics*: those aspects of participant connection and computer use patterns that affect reachability, reliability, and efficiency. We show that the current inability to effectively blacklist malicious participants combined with a distribution mechanism that allows automated collection of both usernames and tasks leaves some systems vulnerable to large scale collusion. Indeed, download patterns of users of the largest of the current systems, SETI@Home, already demonstrate the potential for *hoarding*, i.e., the individual aggregation of thousands of tasks.

6.1 Push versus Pull: Models for Task Distribution

There are two basic mechanisms for distributing work to participants: a proactive "push" model in which the server assigns tasks to participants and dispatches those tasks, and a passive "pull" model in which tasks are dispatched at the request of the participant. The term *push* is not entirely appropriate because the server can only initiate communication with a participant who has a static IP address. Instead, servers implementing this model queue assigned tasks which are subsequently dispatched when the participant initiates communication. The important distinction is that participants in a pull model initiate task acquisition while participants in a push model must wait for tasks to be dispatched.

The choice of push versus pull is largely one of control versus efficiency. Push model servers retain greater control over the assignment of tasks, allowing for a more widespread distribution of tasks among available participants. Push servers can also assure that no username has multiple outstanding tasks, valuable for ensuring that redundantly dispatched tasks are assigned to different usernames. Pull servers, on the other hand, are attractive because they utilize participant processing resources more efficiently. Because distribution of tasks is by request in a pull system, factors such as participant machine speed, accessibility, availability, and reliability are accounted for automatically — participants with sufficient resources who connect regularly can download and concurrently execute several tasks. Moreover, the server does not require complex scheduling and distribution algorithms. The pull system is also work-conserving — a participant ready for new work will not wait to receive work.

Although both models have been used in practice, two of the largest current platforms, SETI@Home and Folding@Home, both use pull model servers.

6.2 Participant Demographics

Remote execution environments, participant computers, and the owners of the computers represent are very important aspects of any distributed platform. As such, the characteristics of each of these aspects have a significant influence on system design and configuration parameters. Three of the most important characteristics are *accessibility*, *reliability*, and *availability*.

Accessibility refers to the ease with which a particular participant can be contacted. Two factors influencing accessibility involve connection type — whether a participant has a continuous broadband connection and, if so, whether a static IP address has been assigned. Users with static IP addresses are highly accessible, facilitating server initiated communication and large data transfers. Broadband users who receive IP addresses via DHCP are also accessible, although server initiated contact is constrained. Because broadband users can retain their IP addresses for days at a time, the server will likely succeed in initiating communication within a reasonable interval after receiving a communication from the participant. Unfortunately, while the proportion of users with broadband access is increasing, dial-up connections still dominate . According to the most recent survey released in February 2003 by the UCLA Center for Communication, 75% of Internet users connected to online services via telephone modem while only 17% connected via broadband [4]; similar results were reported by the UK Office of Telecommunications [3]. For dial-up users, accessibility is influenced primarily by the length of time between connections. Many distributed compute clients are coded to communicate transparently with the server when the participant computer initially goes online.

Reliability and availability (i.e., processor availability) refer to the frequency with which results are returned and the proportion of host processor time available for execution of task code. Reliability is not a reflection of the goodness of returned results, but only whether results are returned at all. Factors influencing reliability and availability include whether the computer is owned by a business or an individual, usage patterns of the primary operator, and machine performance (e.g., a relatively slow processor with limited graphics hardware may take days to complete a task that would require only a few hours on a better equipped machine).

Participant demographics influence system security by dictating important system parameters and by determining the efficacy of proposed security solutions. In terms of design decisions, demographics can negate the purported advantages in control afforded by a push model, and in particular the ability to balance the distribution of tasks. For example, systems in which participants have high accessibility, availability, and reliability can count on frequent communication with the server and thus sufficient task dispatch rates. Since few tasks timeout and participants are accessible, distribution patterns will to a great extent match those intended by the server. Systems experiencing less favorable demographics will find that task distribution shifts to participants with better characteristics as tasks originally assigned or dispatched to slower machines timeout and are subsequently reassigned.

With regard to proposed security mechanisms, demographics can affect the performance of some counterattack strategies. Consider, for example, the Mersenne prime search mentioned in Section A and the associated strategy described in [36] to harden this application. The strategy is to share the work of computing N tasks among K participants, where $K > N$, so that redundancy increases the likelihood of the owner detecting malicious behavior.

To further enhance the strategy, the server can reassign the tasks to participants at random times[2]. In theory, this counterattack strategy is effective; in practice, however, its performance is dictated in large part by the demographics of the participants. The server can designate a priori to a participant the point in the computation that a partial result is to be returned. However, the server's ability to retrieve that result and reassign to a different participant is affected by the accessibility of the two participants. The time required to perform the entire computation is likely to increase as a result. Event if demographics do not affect the efficacy of a countermeasure, potential decline in performance must be considered. In general, the architect of any security mechanism that requires significant communication with the server would be wise to understand well the influence of demographics.

6.3 Blacklisting (The Participant Authentication Problem)

Sarmenta [32] points out the difficulty of blacklisting participants. Specifically, participants cannot be reliably identified using IP or email addresses. In the former case, many participants receive temporary addresses from their ISP via DHCP, and even static IP addresses can be forged[3] — replies from the server would be sent to the forged address, but an adversary with knowledge of the registration process (say from performing a legitimate registration earlier) could anticipate the replies necessary to complete the registration. Email addresses can also be forged, although this is unnecessary since services such as Yahoo provide free email addresses in quantity[4]. Moreover, even addresses that can be bound to a specific organization are of little help since an adversary may be using a compromised machine as a base for launching an attack. Requiring more detailed personal information such as a social security number or postal address is likely to discourage many potential volunteers.

The problems of authentication in general, and for users in a web setting in particular, have been well studied (e.g., see [16] for a survey of web authentication measures). The authentication required for blacklisting, however, has an added twist. Most authentication protocols involve a party that wishes to gain access to a restricted resource, and thus *wants* to be authenticated. In our context, an adversary *does not want* to be authenticated — the adversary gains from usernames than cannot be bound to personal identity. Participants already have access to the resource, so the onus is on the supervisor to prove who users are *not*, rather than who they are.

The bottom line is that in the absence of a reliable and ubiquitous technology for binding usernames to real people, usernames, rather than individuals, are all that can be removed from a computation.

6.4 Cumulative Effects: The Likelihood of Collusion

There is a general assumption among the papers that examine securing distributed computations that the probability of colluding participants is low. If one is considering the likelihood that distinct adversaries act in concert, then this may be true. In the present context, however, collusion involves multiple *participants*, not necessarily multiple adversaries. This is an important distinction because the probability of a single individual controlling several tasks is far from remote. Several commercial ventures, for example, have solicited participants from large institutions via packages through which credits gained from task execution are collectively donated to the institution or to charitable causes. A single malevolent systems administrator at such an institution could conceivably command the entire pool of participant nodes for a computation. If such administrators were the only individuals capable of controlling several participants, this might be acceptable. Unfortunately, as we show next, hoarding participants is relatively easy.

An adversary can gain control of a significant number of participants either by obtaining multiple usernames or through a single username that is able to obtain multiple tasks. An adversary holds the greatest advantage if able to do both, and he or she can in many of the systems currently in place. Since computation supervisors have no effective means of binding usernames to real people, obtaining usernames is trivial. Several of the registration procedures we tested, for example, required nothing more than downloading a registration program and entering the desired

[2]Although this enhancement does not appear in the paper, it was discussed during presentation of the original strategy.

[3]There is also the practical issue of requiring that potentially unsophisticated PC owners determine whether their IP address is static.

[4]Yahoo and some similar sites have recently implemented measures to prevent users from using scripts to collect thousands of email addresses, a technique widely used by spammers. These measures rely on reverse Turing tests [41] and have lowered the number of email requests at Yahoo by 90%. Still, there is no mechanism in place to prevent a determined adversary from manually collecting as many email addresses as desired.

username. Preventing a single username from gaining multiple outstanding tasks is possible, but can significantly decrease computational capacity. SETI@Home, for example, receives a great deal of work from individuals who execute multiple tasks concurrently. As of this writing, SETI's top ten contributors (of over 4.6 million participants) have completed over 11.5 million tasks, or more than 1% of their entire output. The top individual user has completed over 3 million tasks during the approximately 1600 days that SETI@Home has been operating (since March 1, 1999) for an average of approximately 1900 tasks completed each day. The potential for collusion is magnified by the ability to acquire usernames and download tasks via automated scripts.

Download statistics for SETI@Home indicate that such hoarding could evade detection. Anecdota indicate that the number of new usernames requested from SETI@Home in any 24 hour period varies widely, with as few as 1100 and as many as 5100 per day observed in recent weeks. An adversary using a script to obtain 300 usernames per day may go undetected. An inordinately large number of requests is likely to be detected, but detection in this case is useless without any means of identifying which of those requests are generated by an adversary[5]. Regarding usernames with multiple tasks, consider that SETI@Home receives about 6 requests for tasks every second. If an adversary with several usernames requests a task once every 5 seconds, the adversary can in theory gather about 17 000 tasks in 24 hours. Doing so would raise the average rate of requests seen at the server from 6 per second to only 6.2 per second. Fortunately, SETI@Home has validation inherent in their application — various regions of the sky are visited multiple times, so forged signals returned and good signals found but not returned will eventually be detected. Most other applications do not enjoy this advantage.

If redundancy is the primary means of securing a computation, then collusion is especially effective if an adversary is able to obtain redundant pairs of tasks. This attack is facilitated in many systems by the timeout values dictated by participant demographics. Although data concerning some of these parameters is difficult to obtain, it is not unreasonable to expect that users with dial-up connections require longer timeout values on average. However, a longer timeout provides an adversary with a larger window of time in which more tasks can be requested with the specific intent of obtaining redundantly assigned tasks. If the adversary is able to obtain redundant tasks within the timeout window, the ability to disrupt the computation via that specific task is assured.

Given that blacklisting is not possible and that preventing the assignment of multiple outstanding tasks to a single username may not be practical, we instead focus on ways in which the ability of the adversary to hoard tasks can be significantly slowed. We propose that automated hoarding be hampered by employing reverse Turing tests (RTTs) and memory bound functions.

Traditional Turing tests [37] involve a person trying to prove to another person that he or she is human rather than machine. In an RTT, a person tries to prove to a computer that he or she is human. The CAPTCHA project [41, 40] at Carnegie Mellon University is one of the more recent studies in this area. Although the accuracy of these tests is by nature difficult to verify formally, RTTs demonstrate great promise. Accurate RTTs can be valuable in the our context as a means for preventing the automated downloading of usernames and for preventing the automated downloading of many tasks in a short period of time. Moreover, RTTs can potentially accomplish this goal with minimal disruption to honest participants. One of the tests developed by the CAPTCHA project, for example, displays a distorted image of several words and asks the user to identify three of the words. Asking a participant for 15 seconds to complete such a test is reasonable — no technical knowledge is required nor the recollection or use of a password. Most importantly, RTTs are an imposition only to someone attempting to obtain many usernames. Clearly, RTTs do not *eliminate* the threat of an adversary obtaining multiple usernames, but the task is made more difficult and time consuming because the process cannot be automated.

Memory bound functions (MBFs) [5] are functions that are intended to evaluate at approximately the same speed on most currently used systems. As their name suggests, MBFs achieve this goal through the use of operations that require continual memory accesses. Although processor speeds can vary significantly, processor-memory *bus* speeds of machines built in the last five years vary by a factor of approximately two. MBFs can be used in our context to reduce the rate at which an adversary can gather usernames and tasks on a single machine, forcing the adversary to execute tasks on a network of workstations in order to achieve disruption in a reasonable amount of time. If, for example, a username is not registered and a task is not dispatched unless the result of a fifteen minute memory bound function is completed and returned, then the ability to hoard tasks is impaired. Since computation of the MBF is transparent to the user, the imposition on legitimate users is minimal. Unfortunately, MBFs make it difficult for legitimate users to execute several tasks concurrently. We feel, however, that it would not be unreasonable to require that a user wishing to obtain a large number of outstanding tasks identify himself or herself to the supervisor of the computation through more trustworthy means.

[5]Of course someone using a script to obtain thousands of usernames at once would likely be detected as well as stopped.

7 Redundancy

In some cases redundancy may be the only realistic security measure available to a distributed metacomputation. Indeed, to date almost every proposed solution revolves around some form of redundancy, whether by generating execution traces, using ringers, or applying partial redundancy in various schemes. Redundancy is not a panacea, however. For many computations redundancy is either inefficient or unnecessary. For those computations Supervisors should understand the underlying assumptions on which a redundant solution is based.

First, there are several computations for which redundancy is either cost ineffective or unnecessary. Using redundant tasks in a DES key search, for example, is not cost effective. The only way the computation can fail is if an adversary finds the key and fails to return it. If there are n tasks and the proportion of tasks under control of adversaries is p, then in the absence of redundancy the expected number of tasks that need to be executed is given by

$$n(1-p)\sum_{k=1}^{\infty} kp^{k-1} = \frac{n}{1-p}.$$

With redundancy, the expected number of tasks is at least $2n$. Thus as long as $n/(1-p) < 2n$, (i.e., if $p < 1/2$) the expected cost is less without redundancy.

Redundancy may not be necessary in cases where application structure provides *self correction* or in *cases exhibiting inherent redundancy* in the structure of the application. Consider, for example, the protein folding application currently being run by Folding@Home. The plausibility of results returned by these tasks is easily checked (the way which this is done is beyond the scope of this paper). Suppose, however, that a falsified result passes through this filter, causing a thousand other participants to recenter on the forged configuration. The folding application's stochastic nature ensures that the thousand trials quickly decorrelate. The trials will independently sample very different portions of the configuration space. Furthermore, consider that a given protein folds into a unique configuration. Since this configuration minimizes free energy for its particular string of amino acids, given any initial configuration of the protein the free-energy minima will eventually be reached. Regardless of the increased time to reach this minima, the computation will succeed. Thus, protein folding is inherently self-correcting. *The decision to use redundancy in this case depends on the perceived cost of the increased time required to reach a solution.* On the other hand, any Monte Carlo method is resilient to a small proportion of bad results and thus may not be expected to benefit from the increased costs of redundantly assigned tasks.

We now consider the conditions for which redundancy may be beneficial. First, an application should consist of tasks and overall computation results that are not easily verified; otherwise, there is no need for redundancy. Second, the supervisor of the computation should be willing to trust that matching results are valid or, if using voting schemes, that the majority result is correct. Otherwise the supervisor must validate each result and is therefore performing the redundancy himself.

The supervisor who places trust in matching or majority results, however, has based her solution on two implicit assumptions, neither of which may be valid. First assumption is that the probability of receiving matching *incorrect* answers is low. In the absence of collusion, this is likely the case. However, as discussed in Section 6.4, the odds of receiving matching incorrect answers are far from negligible, given that current platforms allow a single individual to acquire an effectively unlimited number of usernames and to download several thousand tasks.

The second assumption is more subtle and concerns the methods by which *mismatched answers* are handled. When mismatched answers are received, the next step is naturally to assign the task to another participant. Of course this further increases the number of tasks required for the computation as well the time[6] required to complete the computation. In assigning the task to an additional participant,however, there is the implicit assumption that this third participant is different from either of the first two. This is problematic because, as discussed in Section 6.3, *supervisors of computations have no knowledge of the identity of the person with whom they are communicating.* The misconception is that when redundancy is employed, each task instance is assigned to a different person. Because there is no means of reliably identifying adversaries there is no guarantee that tasks are not being reassigned to the same adversary regardless of the level of redundancy.

So the question that really should be considered when thinking about using redundancy is this: what are the implications of using redundancy in an environment in which there may be many participants under the control of adversaries and in which the supervisor of a computation cannot be sure to whom they are sending tasks!

[6]Some computations, such as SETI@Home, are ongoing so one can argue that the additional time is inconsequential. Most commercial computations, however, need to be completed in time to satisfy the customer.

Finally, the use of redundancy can impair an application more than if tasks had been assigned only once. Consider, for example, applications in which a select few tasks will contain the results of primary interest. In the absence of redundancy, this type of application can only be disrupted if an adversary is assigned one of the distinguished tasks. Redundancy, however, opens the computation to two attacks. First, regardless of which tasks are assigned an adversary can fail to return its assigned tasks. Since task results are not accepted until they are matched, the completion of this task is delayed for at least the task timeout length, and possibly longer (depending on how quickly the reassigned task is completed). Moreover, the adversary will not be identified as an adversary since he or she will not have returned a bad result. The second attack involves sending a forged good result. Such a result creates a mismatch, which again causes the task to be reassigned. In this case, the username can be removed from the computation, but as we have shown this provides little disincentive to an adversary capable of hoarding usernames. Both of these attacks increase both compute and time costs, as well as create additional work for the server. Moreover the attacks magnify the impact o an adversary, because the attacks force the assignment of another task.

To be clear, we do not discourage the use of redundancy when security needs justify the additional costs. However supervisors should be cognizant of the potentially fallacious assumptions on which redundancy is based and aware of the implications of its use.

8 Conclusions

In this paper, we discussed several security issues unique to distributed metacomputations. In the context of real applications, we have shown that subtleties in the structure of applications, in their implementation, and in the underlying computation platform can undermine the intent of security protocol mechanisms. We have shown that, as currently designed, many distributed computing platforms are unsettlingly vulnerable to collusion from participant nodes under the control of a single adversary. We presented specific recommendations with respect to the security issues raised, including the use of reverse Turing tests and memory bound functions to prevent an adversary from automated acquisition of usernames and tasks. Finally, we discussed the use of redundancy, specifically noting that in many cases redundancy is unnecessary or cost inefficient. Even worse, redundancy can leave a system vulnerable to attacks that would not be possible in the absence of redundancy. We reiterate that our purpose in discussing these issues is not to provide a road map for adversaries; rather, we wish to create a general awareness of these issues and to provide a common and realistic set of assumptions on which future research efforts can be based.

Acknowledgements

The authors wish to thank Dave Anderson and Jeff Cobb of the SETI@home project for providing us with statistics regarding the SETI@home platform. We thank Dr. Vijay Pande of the folding@home project for his helpful discussion regarding the folding@home implementation. We thank Doug Blair, Dale Newfield, Sean Cier, and Dustin Lucien, all formerly of Parabon Computation, Inc, for helpful discussions regarding implementations details of Smith-Waterman sequence comparisons, protein folding, and graphic rendering, respectively. We especially thank Art Charlesworth for his insights during the writing of this paper.

References

[1] *The SETI@home Problem*, September 2000.

[2] Personal conversation (email) with Jeff Cobb, July 2003.

[3] Consumers' use of internet: Oftel residential survey Q11 november 2002, January 2003.

[4] The UCLA Internet report: Surveying the digital future, year three, February 2003.

[5] M. Abadi, M. Burrows, T. Wobber, and M. Manasse. Moderately hard, memory-bound functions. In *Proceedings of 2003 Network and Distributed Systems Security Symposium*, pages 25–40, San Diego, California, February 2003.

[6] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An experiment in public resource computing. *Communications of the ACM*, 45(11):56–61, November 2002.

[7] 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.

[8] 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.

[9] A. Bissett and G. Shipton. Envy and destructiveness: understanding the unconscious motivations of the computer virus makers. In *Ethi-Comp Conference Proceedings*, 1998.

[10] A. Bissett and G. Shipton. Some human dimensions of computer virus creation and infection. *International Journal of Human Computer Studies*, 52:899–913, 2000.

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

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

[13] 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.

[14] N. Draper and H. Smith. *Applied Regression Analysis*. John Wiley & Sons, 1966.

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

[16] Kevin Fu, Emil Sit, Kendra Smith, and Nick Feamster. Dos and don'ts of client authentication on the web. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[17] Genbank. http://www.ncbi.nlm.nih.gov/Genbank/GenbankOverview.html.

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

[19] 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.

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

[21] F. Hohl. An approach to solve the problem of malicious hosts. Technical Report TR-1997-03, Universität Stuttgart, Fakultät Informatik, Germany, March 1997.

[22] Fritz Hohl. Time Limited Blackbox Security: Protecting Mobile Agents From Malicious Hosts. In Giovanni Vigna, editor, *Mobile Agent Security*, pages 92–113. Springer-Verlag: Heidelberg, Germany, 1998.

[23] Fritz Hohl. A protocol to detect malicious hosts attacks by using reference states. Technical Report TR-1999-09, 1999.

[24] L. Hunter. Molecular biology for computer scientists. In L. Hunter, editor, *Artificial Intelligence and Molecular Biology*, AAAI Press Series, chapter 1, pages 1–46. MIT Press, Cambridge, MA, 1993.

[25] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky. SETI@home—massively distributed computing for seti. *Computing in Science and Engineering*, 3(1):78–83, January/February 2001.

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

[27] 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.

[28] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[29] 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.

[30] Tomas Sander and Christian F. Tschudin. Protecting Mobile Agents Against Malicious Hosts. In Giovanni Vigna, editor, *Mobile Agent Security*, pages 44–60. Springer-Verlag: Heidelberg, Germany, 1998.

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

[32] L.F.G. Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. In *Proceedings of the ACM/IEEE International Symposium on Cluster Computing and the Grid*, Brisbane, Australia, May 2001.

[33] Michael R. Shirts and Vijay S. Pande. Atomistic protein folding simulations on the submillisecond time scale using worldwide distributed computing. *Wiley*, 86(22), May 2001.

[34] Michael R. Shirts and Vijay S. Pande. Mathematical analysis of coupled parallel simulations. *Physical Review Letters*, 86(22), May 2001.

[35] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[36] D. Szajda, B. Lawson, and J. Owen. Hardening functions for large-scale distributed computations. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 216–224, Berkeley, CA, May 2003.

[37] A. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[38] Giovanni Vigna. Protecting mobile agents through tracing. In *3rd ECOOP Workshop on Mobile Object Systems*, Jyväskylä, Finland, 1997.

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

[40] L. von Ahn, M. Blum, N. Hopper, and J. Langford. CAPTCHA: Using hard AI problems for security. In *Proceedings of Eurocrypt 2003*, 2003.

[41] L. von Ahn, M. Blum, and J. Langford. Telling humans and computers apart (automatically). *Communiations of the ACM*, to appear.

[42] M. Waterman. *Introduction to Computational Biology: Maps, Sequences, and Genomes.* Interdisciplinary Statistics. Chapman & Hall, 1995.

[43] Bojan Zagrovic, Eric J. Sorin, and Vijay S. Pande. Beta hairpin folding simulations in atomistic detail using an implicit solvent model. *Journal of Molecular Biology*, 313(2):151–169, 2001.

A Example Application Details

This section presents examples of applications to which the distributed computation technique has been applied. We do not discuss these applications in great detail, but instead provide an overview sufficient to illuminate general characteristics. These applications serve as a basis for the topics developed in Sections 5 and 6. Readers interested in more detailed descriptions of these applications are encouraged to consult the references provided for each.

A.1 Exhaustive Linear Regression

Regression [14] is a technique for fitting experimental data to a (possibly N-dimensional) function. It is used to determine the relationship between one or several *predictor* variables and a predetermined *response* variable that are observed in an experimental setting. Data observed in this fashion can be expressed by vectors of the form

$$(y_i, x_{i1}, x_{i2}, x_{i3}, \ldots, x_{iN})$$

where $x_1, x_2, \ldots, x_N$ represent the predictor variables, y is the response variable, and x_{ij} is the value of x_j in the ith observation. We assume k such observation vectors where $k \gg N$. The adjective linear refers to the assumption that the predictor variables are (possibly) related to the response variable y through an equation of the form

$$\hat{y} = a + b_1x_1 + b_2x_2 + \ldots + b_Nx_N, \tag{2}$$

which defines a hyperplane. The parameters a and $b_1, \ldots, b_N$ are determined via least squares methods [14]. The goodness of fit of a *specific* hyperplane (i.e., the least squares estimates for a and the b_i following from a particular subset of the $x_1, x_2, \ldots, x_N$) is determined by measuring the collective discrepancies between the predicted and observed experimental values, i.e., y_i and $\hat{y}$. For many reasonable measures of "discrepancy", formulas for finding the best fitting curve are well known and relatively easy to apply. More difficult, however, is determining the class of curve against which to fit and the smallest subset of predictor variables that significantly influences or determines the value of the response variable.

Exhaustive regression addresses the latter concern in a brute force manner. All possible combinations of predictor variables are tested in order to determine the single combination of predictors that produces the "best" fit. The process is computationally intensive — if there are N predictor variables, then there are $2^N - 1$ (the combination in which no predictors are present is not considered) possible regression equations. Fortunately, the information required to fit a curve to any single combination of predictors (i.e., performing an individual regression, the elementary operation of the tasks for this application) is a small set of statistics based on the observed data and is identical for each potential combination. Thus the problem is easily parallelized — specifying a task amounts to describing the sets of combinations of predictors on which a participant should perform regressions. Significant results are determined according to goodness of fit, with a goodness threshold determined a priori by the computation supervisor.

A.2 Protein Folding

Folding[7] is the process by which a protein, consisting of a string of amino acids, takes on the physical configuration that gives the protein its biochemical functionality. A protein's folded structure is predetermined by its amino-acidic composition, but determining this configuration *a priori* is quite difficult. In reality, proteins fold quite rapidly — on the order of a microsecond. However, the computational requirements for a complete simulation of the protein folding process limits simulation to just one nanosecond per CPU-day [15]. As a result, simulation of a protein fold, intractable using conventional computing power, is well-suited for distributed computing platforms.

A protein folds in such a way that the total free-energy of the resulting structure is minimized. If every possible structure is simulated and the associated free-energies calculated, the structure with the least free-energy is an appropriate prediction of the protein's ultimate folded structure. Molecular dynamics (MD) simulations are used to solve this problem of minimizing free-energy. Each MD simulation is initialized with a set of coordinates that describes the location of atoms in a molecule and the initial velocities of each atom. The structure of each molecule obeys Newtonian mechanical rules as the system changes over time. At regular time intervals, molecular snapshots are taken; collectively, these snapshots form a molecular dynamics trajectory that describes the evolution of the folding process.

Folding@Home[8], a distributed computing project run by the Pande Group at Stanford University, seeks to parallelize the folding process [33]. In this context, many simulation *trials* comprise a folding *series*. The trials within a series are distributed over thousands of participants — all participants start with the same initial location coordinates, but each participant simulates a different fold using random thermodynamic forces [43]. By collecting many independent folding series into an *ensemble*, Folding@Home is able to simulate the large-scale folding process in an efficient manner.

The process of folding takes a protein through several metastable states before producing a final configuration. A transition from one of these intermediate states to another occurs as a molecule passes a free energy barrier, where a significant energy peak is observed. When a trial transitions (i.e., crosses a free energy barrier), all simulations in the associated series are restarted using the molecular coordinates of the transitioned trial. Each realigned trial continues using a unique random sequence of thermodynamic forces acting upon the molecules. In this way, the entire configuration space is sampled. This scheme permits Folding@Home to achieve a linear speedup in simulation time [15].

A.3 Smith-Waterman Sequence Comparisons

We describe here the Smith-Waterman dynamic programming algorithm [35] for genetic sequence comparisons. The sequences that biologists study consist of either nucleotide bases (occurring in DNA fragments) or amino acids (the building blocks of proteins). For DNA sequences, the underlying alphabet is $\Sigma = \{A, C, T, G\}$ representing the

[7]The reader interested in the biology of protein folding should consult [24].

[8]For a comprehensive description of the Folding@Home methodology, the interested reader should consult [15, 34, 33, 43].

nucleic acids adenine, cytosine, thymine, and guanine. The underlying alphabet for protein sequences consists of 20 symbols, each representing an amino acid used in constructing proteins.

Consider a sequence $\mathcal{A} = a_1a_2\ldots a_n$ over Σ. Sequences evolve primarily in three ways: an element of a sequence is removed (a *deletion*), an element is inserted (an *insertion*), or an existing element is transformed into a different element (a *substitution*). As depicted in Figure 1, biologists track evolutionary changes by writing the original sequence above the new sequence with appropriate positions aligned. In Figure 1(a), a_2 undergoes a transformation from T to A; in (b), a_4 is deleted with '–' representing the *gap*; in (c), the element G is inserted into the second position. Figure 1(d) and (e) represent two example evolutions involving several mutations — both cases consist of exactly the same sequence of elements but with different alignments. It is important to note that there are *many* different possible alignments of two sequences. For example, two sequences of length 1000 have approximately 7.03×10^{763} distinct alignments!

$\mathcal{A}$: CTGTTA	$\mathcal{A}$: CTGTTA	$\mathcal{A}$: C-TGTTA	$\mathcal{A}$: C-TGT--TA--	$\mathcal{A}$: CT-GT--T-A
$\mathcal{B}$: CAGTTA	$\mathcal{B}$: CTG-TA	$\mathcal{B}$: CGTGTTA	$\mathcal{B}$: CTA-TGCT-CG	$\mathcal{B}$: CTA-TGCTCG
(a)	(b)	(c)	(d)	(e)

Figure 1: Example DNA Sequences

To measure the goodness of an alignment, a *scoring function* is weighted to reflect whether symbols in the same position in two different sequences match. Similarly, a *gap function* determines the penalty assessed for inserting or deleting an element to obtain an alignment. The similarity $S(\mathcal{A}, \mathcal{B})$ of sequences $\mathcal{A}$ and $\mathcal{B}$ is defined to be the maximum score over all alignments between the two sequences. Although the number of alignments is huge, a dynamic programming algorithm developed by Needleman and Wunsch [28], and later augmented by Smith and Waterman [35], allows the similarity of sequences of length n to be determined in $O(n^2)$ time.

Furthermore, one may consider *local* alignments, i.e., the segments of each sequence that are most similar. Because there are $\binom{n}{2}\binom{m}{2}$ different sequence alignments, the amount of compute required to consider all possibilities is enormous. Waterman notes that computing local alignments for two length n sequences takes $O(n^6)$ time using the algorithm above for computing S [42]. Fortunately, a dynamic programming technique reduces the required time considerably.

A researcher conducting a Smith-Waterman local sequence comparison has thousands of sequence segments culled from the genome of a particular organism. The researcher compares this information to the archived sequence information from a large public database such as the National Institutes of Health GenBank database [17], which as of the date of this writing lists almost 23 billion bases in over 18 million sequence entries. The compute job compares each of the thousands of sequence segments with each of the millions of sequences in the database.

Parabon Computation Inc. implemented a Smith-Waterman sequencing application. The application divides both the NIH database and the researcher's sequences into groups of approximately 100 sequences. Participants are sent the sequences (with job specific sequence identifiers) along with the scoring and gap functions. Participants then compare all pairs of sequences, returning the identifiers of pairs that score above a precomputed similarity threshold. Of the 10 000 or so comparisons run by each participant, only about 30 are expected to be significant. These are returned to the researcher for further analysis.

A.4 SETI@home

The SETI@Home project attempts to detect signals of extraterrestrial origin. Their efforts require a significant amount of compute because the parameters of an extraterrestrial signal are unknown and because required processing costs rise in accordance with increased search sensitivity.

SETI believes that an alien civilization generating a signal with the intention of contacting other civilizations would choose a narrowband signal that could be easily distinguished from the wide band celestial background noise. Obviously, many factors are unknown, such as the bandwidth and frequency of a signal, whether the signal will be pulsed, and if so the pulse period. Moreover, because of the rotation of the earth and the motion of a signal's planet of origin relative to the earth, doppler shifts would likely cause a narrowband signal to quickly drift out of band. Thus potential signals are tested according to several doppler drift rates. Finally, any potential alien signal is tested for celestial origin. As an example, radio frequency interference can be identified but the detection algorithms require nontrivial processing. Given the large number of potential parameters, tasks require from 2.4 to 3.8 trillion floating point operations — about 10 to 12 hours on a 500MHz PC.

SETI@Home performs a number of postprocessing steps on potential signals returned from tasks. Much of this work is devoted to identifying terrestrial signals. SETI also assigns tasks redundantly (with a factor of 2 or 3) and performs cross checking. Error rates are typically around 2% [2], though not all errors are the result of malicious activity. Some errors are introduced via the communication protocol, and others result from processor, disk, and memory errors. Although such errors in general occur with low probability, SETI@Home generates such a large number of floating point operations that even with error rates as low as one in 10^{18} machine instructions, several will be seen on any given day.

The SETI@Home core computing platform consists of three servers. One server holds a science database that contains time, sky coordinates, frequency, and other statistics for each task as well as the relevant parameters of potential signals returned by participants. A second server holds a user database that contains information on SETI@Home participants, the number of tasks completed, last connection time, etc. The third server stores tasks that are waiting to be dispatched along with the results of completed tasks. These servers communicate with the participants via HTTP.

A.5 Great Internet Mersenne Prime Search (GIMPS)

The Great Internet Mersenne Prime Search is an ongoing project run by Entropia.com [18]. 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[9], 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 [12] 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.

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

Appendix B

Collusion Resistant Redundancy

Collusion Resistant Redundancy for Distributed Metacomputations

Doug Szajda Arthur Charlesworth Barry Lawson Jason Owen Ed Kenney
University of Richmond
Richmond, Virginia
{dszajda, acharles, blawson, wowen, ekenney2}@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). These concerns are often addressed by assigning tasks redundantly.

Aside from the additional computational costs, a signi£cant disadvantage of redundancy is its vulnerability to colluding adversaries, since matching returned results are rarely veri£ed. This paper presents a general strategy for applying redundancy in a manner that provides signi£cantly increased resistance to collusion. Advantages over redundancy include an order of magnitude improvement in the probability of detecting colluding adversaries, as well as the ability to ef£ciently identify all colluding participants. Moreover, this improvement is achieved without an increase in the amount of computation required by participants, but instead with a slight increase in bookkeeping overhead. Our strategy is tunable—it generalizes simple redundancy and covers an entire spectrum from simple redundancy to the most expensive and most secure redundant assignment of tasks. Finally, our strategy is extensible in the sense that it does not preclude, and can be augmented by, the use of some previously proposed strategies for securing distributed metacomputations.

Keywords: distributed computation, probabilistic veri£cation, collusion

1. Introduction

The advent of large-scale distributed computing platforms, consisting of many personal computers connected to the Internet, provides researchers and practitioners a new and relatively untapped source of computing power. By utilizing the spare processing cycles of these computers, computations are now possible that were once unobtainable without the use of a supercomputer. In a typical distributed computation, the computation is easily divisible into independent *tasks*, each of which can be processed by a typical personal computer in a few hours. A *participant* downloads code from the *supervisor* of the computation in order to establish an execution environment in which the supervisor can execute tasks. Each task is assigned and dispatched to a participant, and upon completion of the task signi£cant results are returned to the supervisor. In this context, providing a level of assurance for results is dif£cult because the results are obtained by executing tasks in untrusted environments. Participants can intentionally or unintentionally corrupt results, and can attempt to claim credit for work not completed.

A common technique for securing these computations is simple redundancy—assigning each task to two participants. In addition to at least doubling the required cost of computation, simple redundancy is vulnerable to colluding adversaries because supervisors rarely verify matching returned results (for many applications verifying a task result requires recomputing the entire task). In the present context, this is a signi£cant weakness, since in many distributed computing platforms there are no mechanisms in place to prevent an adversary from obtaining multiple (even hundreds of) user names and downloading hundreds or even thousands of tasks[1]. Detection of colluding adversaries is dif£cult and/or expensive, and moreover detection of an ad-

[1]The Search for Extra-Terrestrial Intelligence project conducted by SETI@Home, for example, has experienced days in which more than 5000 new user names were assigned, and boasts participants who have *averaged* more than 1000 tasks completed each day in the four years since the project began.

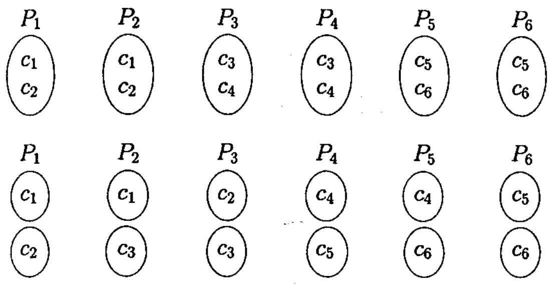


Figure 1. Alternative ways of assigning three tasks to six participants. Tasks in the top row are assigned using simple redundancy.

versary provides no information about additional colluding participants.

Our scheme is motivated by the notion that a supervisor who accepts the increased cost of computation associated with simple redundancy should receive a better return on their investment. Speci£cally, these same computational resources can be allocated such that an adversary is *much* more likely to be detected, and that detection reveals the identities of their colluding cohorts. Our mechanism requires no additional computation on the part of the participants, and only reasonable increases in bookkeeping costs for the supervisor.

As a simple example, consider a traveling salesperson computation involving only three cities, and assume that a participant can only compute the cost of two circuits. Let $c_1, c_2, \ldots, c_6$ denote the six possible circuits. Without redundancy, this would require three participants (each computing the cost of two circuits), so simple redundancy requires six, which we denote by $P_1, P_2, \ldots P_6$. Figure 1 shows two possible ways of assigning each participant two circuits. Under simple redundancy, each subset is assigned to two participants, so for example P_1 and P_2 act as checks on each other's work. If they are both controlled by a single adversary, then the costs returned for circuits c_1 and c_2 are compromised since they can return identical incorrect results that the supervisor will assume is correct. In this scheme, P_2 is the *only* check on the work of P_1 and vice-versa. If in addition P_3 is under the control of the adversary, the supervisor will have no knowledge of this even if she happens to be fortunate enough to detect the malicious behavior of P_1 and P_2.

Consider now the alternative assignment. Here, the work of P_1 is checked in part by P_2 and in part by P_3. If the cost of c_1 is returned incorrectly and the cost of c_2 is returned correctly, the supervisor knows that P_1 and P_2 are colluding participants. Moreover, the supervisor can determine whether P_3 is colluding with P_1 and P_2 by checking whether the cost of c_3 has been correctly computed. This additional information is gained without increasing the computational burden on any of the tasks.

Consider now the probability of detecting the malicious activity of P_1 and P_2 by verifying one full task worth of work. We assume, for reasons that will be made clear later, that in the simple redundancy case work can only be veri£ed at the task granularity, a constraint present in real distributed computations. With simple redundancy, the supervisor verifying a single task at random from the three will detect the malicious activity with probability $\frac{1}{3}$. In the alternative scheme, the supervisor is free to check any two circuits from among the six in the computation, so the probability of detection is now $1 - \frac{\binom{3}{2}}{\binom{6}{2}} = \frac{4}{5}$. If instead the supervisor opts to check only a single circuit, the probability of detection becomes $\frac{1}{3}$, so the supervisor acheives a detection level equal to simple redundancy with half the effort.

The strategies presented in this paper represent a spectrum of possible task assignments. At one extreme is simple redundancy, which is the least expensive strategy, but provides the least protection from collusion, the smallest probability of catching colluding participants, and no information about the identities of additional conspirators. At the other extreme is an assignment strategy we call *vertical partitioning*, which is of theoretical interest but does not scale to the dimensions of the typical distributed computation. Vertical partitioning is the most expensive of our strategies, but provides the greatest protection from colluding adversaries, the highest probability of detecting adversaries, and when detection occurs, the ability to identify *all* of the colluding participants. This is acheived through a distribution scheme in which part of the work of each participant is checked by *all* of the other participants in the computation. The primary cost of vertical partitioning is bookkeeping overhead—the amount of computation required by the participants is unchanged. Between these two extremes lies a wide range of assignments we call *clustering*. By varying parameter values, the supervisor can choose the level of partitioning that is both suited to their speci£c application and provides the desired level of protection.

This protection can be further augmented with the use of several other recently proposed security measures for distributed metacomputations. The ringers schemes proposed by Golle and Mironov [7], and Szajda, et. al. [17] can both improve the proability of detecting malicious activity and at the same time decrease some of the costs associated with our strategy. Some of the credibility based strategies proposed by Sarmenta [?] can be integrated with our scheme as well.

We note here that our strategy is not applicable to every distributed metacomputation. In particular, we require that tasks can be divided into subtasks, a property absent from some distributed computations. Sequential computations, for which the basic model is the repeated iteration of a function on a small number of inputs, cannot be subtasked if each task is assigned only a single seed value. Protein folding and some Mersenne Prime searches, for example, fall under this restriction.

In addition to the voluminous literature on securing general distributed computations, there is a small but growing body of literature dealing with securing the speci£c type of system considered here (see e.g. [7, 8, 10, 14, 17]. Some of these [8, 14] describe strategies for intelligently applying redundancy, including methods for determining appropriate levels of redundancy (i.e. triple, quadruple, etc.) required to meet application requirements. One [17] discusses a scheme that applies partial redundancy to sequential computations. None of these works, however, consider the redesign of redundancy itself. That is, strategies that improve the performance of a speci£c level of redundancy in a manner that increases the probability of detecting malicious activity and provides the supervisor with better information as to the identities of malicious participants.

In addition to presenting our strategy, we provide probabilistic analyses that prove the following:

- For a given proportion of participants controlled by the adversary, the expected number of *tasks* controlled by the adversary is identical under each of the strategies. We say that a task or subtask is *controlled* by the adversary if she controls both participants who have been assigned that task or subtask.

- The stability of our strategy is superior in the sense that the variance of the number of tasks controlled by the adversary is greatly decreased under our schemes (and is in fact zero under pure vertical partitioning).

- The probability of detection of adversaries who return matching bad results can be increased by an order of magnitude when using our strategy.

In addition, we provide ef£cient algorithms for task assignment and for identifying colluding cohorts once a pair of colluding adversaries has been 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 and introduce terminology. Section 3 presents the vertical partitioning scheme. Though impractical in its pure form, vertical partitioning and the corresponding analyses provide the basis for the clustering scheme presented in Section 5 In Section 5 we show how the ideas of vertical paritioning can be applied in practice via clustering. We discuss related work in Section 6 and present conclusions in Section 7.

2. The model

We consider parallel applications in which the primary *computation* is easily divided into *tasks* small enough to be solved by a PC in a "reasonable" amount of time (typically several hours of CPU time). The tasks are independent and, in general, consist of one or more *subtasks*. *Sequential* tasks consist of relatively few (often one) subtasks, each of which requires a long time to complete; *non-sequential* tasks consist of many subtasks, each of which completes quickly [17]. The scheme presented in this paper is ideally suited for non-sequential tasks. Nonetheless, the scheme is applicable to sequential tasks provided that the number of subtasks per task is suf£ciently large.

More formally, a computation consists of the evaluation of a function or algorithm $f : D \rightarrow R$ 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 task T_i is assigned a £lter function g_i with domain $P(R)$, the power set of R, and range $P(f(D_i))$, where $f(D_i) \equiv \{f(x) \,|\, x \in D_i\}$. 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 de£nition of g_i is necessary for situations in which the signi£cance of a computed value is relative to the values of f at other elements of D_i. For example, the £lter function for a task in a traveling salesperson computation might specify that a route is signi£cant if it is among the best £ve routes computed.

To introduce terminology used to describe the scheme presented in this paper, consider the example structure of assignments of a hypothetical computation from Table 1. As depicted in (a), the *computation* is £rst divided into $N = 4$ *tasks*. Each task is then divided into $2N - 1 = 7$ *subtasks*. Then $2N = 8$ combinations, each of $2N - 1 = 7$ subtasks, are created (one for each of the $2N = 8$ participants) such that each subtask appears in exactly two combinations (the details of creating the combinations are presented in Section 3). An *assignment*, i.e., one combination of subtasks, is then presented to each participant to compute. Moreover, each term de£ned above is represented in Table 1 as follows:

- the *computation* corresponds to the entire table in (a);

- each *task* corresponds to a column in (a);

- each *subtask* corresponds to a single element (e.g., B_2) from a column in (a);

- each *assignment* to a participant corresponds to a column in (b).

Note that for simple redundancy a task consists of only one subtask; in this context, a task, subtask, and assignment are equivalent.

Table 1. A division of four tasks and associated assignment of subtasks

Tasks			
1	2	3	4
A0	B0	C0	D0
A1	B1	C1	D1
A2	B2	C2	D2
A3	B3	C3	D3
A4	B4	C4	D4
A5	B5	C5	D5
A6	B6	C6	D6

(a) $N = 4$ tasks each divided into $2N - 1 = 7$ subtasks

Participant Assignments							
1	2	3	4	5	6	7	8
A0	A0	A1	A2	A3	A4	A5	A6
A1	B0	B0	B1	B2	B3	B4	B5
A2	B1	B6	B6	C0	C1	C2	C3
A3	B2	C0	C4	C4	C5	C6	D0
A4	B3	C1	C5	D1	D1	D2	D3
A5	B4	C2	C6	D2	D4	D4	D5
A6	B5	C3	D0	D3	D5	D6	D6

(b) Assignment of subtasks to the $2N = 8$ participants

The computing platform consists of a *supervisor* — a trusted central control server or server hierarchy coordinating many (typically 10^4 to 10^7) personal computers in a "master-slave" relationship. The slave nodes, or *participants*[2], are given work assignments by the supervisor. Participants download code, typically as a screen saver or applet, that serves as the local execution environment for work assignments. Because tasks in a computation are independent, communication is necessary (and allowed) only between individual participants and the supervisor. Participants receive remuneration, in one of a variety of forms, for completing their associated work assignment.

We assume the existence of one or more intelligent adversaries (i.e., persons). An adversary possesses signi£cant technical skills by which he or she can ef£ciently 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:

[2]We use the term *participant* to denote both a node and its owner. The speci£c meaning of a particular usage will be apparent from the context.

- 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 signi£cant results.

A single adversary may repeatedly attempt to disrupt the computation as results are (incorrectly) reported and new work is assigned.

We assume that collusion among multiple adversaries is possible. Furthermore, we assume it is likely that a single adversary (person) controls multiple participants (nodes). In typical applications, there is no current mechanism that prohibits an adversary from obtaining multiple accounts under the computation (one per node under control of the adversary). By using multiple participant accounts, a single adversary can gain control of multiple assignments. We assume that the adversary will disrupt the computation only by compromising those subtasks for which the adversary controls all copies. That is, if an adversary *does not* control all copies of a subtask, then the adversary will correctly compute that subtask; if an adversary *does* control all copies of a subtask, then the adversary will disrupt the computation using that subtask.

An adversary may be motivated to disrupt a 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 [16] or Folding@home [6]) in exchange for processor time, an adversary may attempt to cheat. 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 £rm. Finally, malicious intent alone, evidenced by the abundance of hackers and viruses propagating throughout the Internet, is suf£cient motivation for an adversary to return incorrect results or to not return signi£cant 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 veri£ed by other means. In addition, we do not consider attacks that result from the compromise of the supervisor or other trusted management nodes.

3. Vertical Partitioning

We assume that there are N tasks that are to be assigned to $2N$ participants. The idea behind vertical partitioning is to divide each task into subtasks and then assign them to participants such that

- Each subtask is assigned to exactly two participants.

```
Algorithm Task Assignment($N$):
    Input: The number $N$ of tasks.
    Output: Subtasks assigned to each participant

    create $subtask$ array containing the $N(2N-1)$
        subtasks (order is irrelevant)
    $i \leftarrow 0$
    create a complete graph $\mathcal{G}$ with $2N$ vertices
    Do traverse $\mathcal{G}$ in breadth-£rst fashion
        if edge $e$ is encountered for £rst time
            assign $subtask[i]$ to $e.vertices()$
            $++i$
```

Figure 2. Vertical partitioning task assignment algorithm. The algorithm runs in $\mathcal{O}(n^2)$ time.

- Each participant shares exactly one subtask with each of the other participants.

Since there are $2N$ participants, each can be paired with $2N-1$ other participants, so tasks must be divided into $2N-1$ subtasks. Moreover since the number of subtasks in the computation, $N(2N-1)$, is the same as the number of pairs of participants, such an assignment is always possible. An example assignment of 4 tasks to 8 participants is shown in Table 1. The task assignment algorithm is given in Figure 2.

There are two immediate consequences of assigning tasks in this way. First, subtasking shrinks the checkable unit of execution, which both reduces the burden of checking individual returned results, and allows the veri£cation process to cover more of the computation. The result is that the supervisor is given £ner control over which results are veri£ed. The second is that it spreads the responsibility for verifying the work of a single participant from one other participant to *all* other participants. This distribution facilitates the ef£cient identi£cation of all colluding parties once a single colluding pair have been identi£ed.

The bene£cial effects of enhanced £rst factor should not be discounted. Distributed computations that rely on redundancy are vulnerable to collusion because matching returned results are rarely veri£ed. The reason for this is cost: for many applications, the only way for the supervisor to verify a returned result is to recompute the entire task[3]. This is expensive, and obviously cannot be done for any significant proportion of the tasks. Subtasking, however, allows the supervisor to effectively check the work of N participants for the cost of verifying a single task. The improvement in the probability of detecting malicious activity is the results of this quantization effect. In essence, the supervisor who uses simple redundancy is locked into performing checks at the task granularity, which limits the ef£cacy of the checking effort.

[3]Of course they could assign the task to a third participant, but there is no guarantee that this participant is honest. Moreover most computations have little means of binding a username to an actual user, so there is no guarantee that they have really assigned the task to a third party, much as there is no guarantee that they assigned it redundantly in the £rst place.

```
Algorithm Adversary Search($Subtask$ Array):
    Input: Subtask results for bad participant $A$.
    Output: List of $A$'s malicious cohorts.

    for each subtask result returned by $A$
        if result returned by $A$ and participant $B$
                are incorrect and match
            Add $B$ to cohort list
```

Figure 3. Cohort Identi£cation Algorithm.

There is a subtlety here that is important for actual implementations. Speci£cally, it is not enough for the supervisor of the computation to "think" of each task as broken into subtasks. Rather, the participants need to know that they are in fact performing several subtasks as opposed to a single task, since then they are obliged to return signi£cant results *from each subtask.* In a computation in which tasks search for an optimum value among all inputs, there is a big difference, in terms of veri£ability, between returning the input that is optimum over the entire task, and returning the ten optimum inputs from each of ten subtasks.

Spreading the veri£cation mechanism allows for easy identi£cation of all colluding cohorts once a single pair has been detected. The algorithm, shown in Figure 3 is straightforward, and requires at worst that the supervisor performs an entire task worth of computation.

The supervisor can further reduce the cost of this operation by reassigning to these same participants a group of subtasks, perhaps from some previous computation, whose results are already known, thus eliminating the need to recompute the subtasks.

The distribution of checking, though bene£cial in many ways, comes with both good and bad news—it simultaneously limits and increases the amount of damage that can be in£icted by an adversary who controls both copies of some subtask. On the one hand, with simple redundancy, the probability of getting a matching task is small, just $\frac{N}{\binom{2N}{2}} = \frac{1}{2N-1}$, but when this does occur, the entire task is compromised. On the other hand, vertical partitioning guarantees that an adversary with control of two participants will control one subtask. They are thus assured that they can compromise at least a fraction $\frac{1}{2N-1}$ of a task. Compromising an entire task, on the other hand, requires controlling all $2N$ participants. We show however, in Section 4 that the expected number of subtasks under control of the adversary is identical in both simple redundancy and vertical partitioning.

There are other costs associated with vertical partitioning, primarily stemming from the management of subtasks. Subtasking introduces at least a factor $2N - 1$ increase in the cost of maintaining any task assignment database, since tracking a subtask is every bit as expensive as tracking a full task. For large N values this will likely become prohibitive. Handling the factor N increase in the number of returned results will also pose di££culties, and is a problem that can not always be easily handled by adjusting the criterion by which results are deemed signi£cant. Tuning applications so that tasks return the appropriate number of signi£cant results is often di££cult, and involves more than simply narrowing the £lter, since there is always the danger of creating a £lter so small that important results will be missed. Subtasking only exacerbates this tuning problem.

4. Analysis

We derive in this section several quantities related to the potential for malicious activity in a distributed computation. We show in particular that if an adversary controls a proportion p of the $2N$ participants in the computation, then the expected number of subtasks controlled under either strategy is the same, and given by

$$E(\#subtaskscompromised) = pN(2pN - 1). \qquad (1)$$

We derive as well the probabilities of detecting malicious activity under the assumption that the adversary cheats on any task or subtask they control.

XXX

In much of the analysis we compare either numbers of subtasks compromised or numbers of tasks compromised. The former technically does not make sense in the case of simple redundancy, where work can be compromised only at task granularity. To be technically correct, we should use a term like *effective task* or *equivalent* task. We decline to do so because our meaning should be clear throughout.

can be compromised only in both simple redundancy and vertical partitioning schemes. In particular, we examine the mean and the variance of the number

potential amount of damage that can be

look at the mean and variance of the number of work units that are controlled by the adversary.

We're going to need to be extra precise with out language here, since when talking about redundantly assigning tasks, we need to be clear about the difference between the number of distinct tasks and the number of things assigned to participants. Thus, I propose the following language: a *task* is what is assigned to a participant. In simple redundancy, there will be participants whose tasks are identical. Both copies of these tasks will represent the same *work unit*. Thus in what follows, if simple redundancy is used, we assume that there are N distinct work units that become $2N$ tasks.

We assume that a proportion p of the total assigned tasks (i.e. of $2N$) are under the control of an adversary. To make the calculations a little more tractable, we assume that the adversary controls an integral number of tasks (i.e. that $2pN$ is an integer).

4.1 Expected Number of Subtasks Under Adversary Control

In this section, we examine the expected number of work units that will end up being controlled by the adversary (i.e. he or she has been assigned both tasks corresponding to this work unit) under both simple redundancy and collusion-resistant redundancy.

With collusion resistant redundancy, every two tasks have one subtask in common. Thus an adversary contolling $2pN$ tasks will have both copies of $\binom{2pn}{2} = pN(2pN - 1)$ subtasks.

With simple redundancy, the calculations are a bit more involved. The Probability that the adversary will be assigned exactly k pairs of matching tasks is

$P(\text{exactly } k \text{ matches}) =$

$$\frac{\binom{N}{k}}{\binom{2N}{2pN}(2pN - 2k)!} \prod_{i=k}^{2pN-(k+1)} 2(N - i), \quad (2)$$

where we assume that the value of the right side product is 1 if the bottom limit is greater than the top limit (i.e. for $k = pN$, the probability of exactly k matches is $\binom{N}{pN}/\binom{2N}{2pN}$). To justify (2) note that there are $\binom{N}{k}$ ways of choosing exactly k pairs from a total of N pairs. Once the $2k$ tasks from the k pairs have been determined, there are now $2N - 2k$ tasks left, so there are this many choices for task $2k + 1$. Once this task has been chosen, there are $2N - (2k + 1)$ tasks left, but one of them would match task $2k+1$, so it cannot be chosen, leaving us with $2N-(2k+2)$ choices for task number $2k + 2$. Moving on to choosing the next task (number $2k + 3$) there are $2N - (2k + 2)$ tasks left, but we cannot choose the two that match choices $2k+1$ and $2k + 2$. Thus, there are $2N - (2k + 4)$ tasks that can be chosen. Continuing in this way to determine the number of ways of getting k matches leads to a product:

$$\binom{N}{k}[(2N - 2k)(2N - 2(k + 1))(2N - 2(k + 2))\ldots$$
$$\ldots(2N - 2(2pN - (k + 1)))].$$

This product, however, overcounts the number of ways of £lling in the remaining cards, because any particular valid combination is counted many times — once for each permutation of the elements of the combination. Thus, we need to divide by the number of such permutations, which is $(2pN - 2k)!$. Thus, noting that there are $\binom{2N}{2pN}$ groups of size $2pN$ from a group of size $2N$, we see that the probability is

$$\frac{\binom{N}{k}}{\binom{2N}{2pN}(2pN-2k)!}[(2N-2k)(2N-2(k+1))\dots \\ \dots(2N-2(2pN-(k+1)))].$$

Now, an adversary controlling $2pN$ tasks can control at most pN work units. Also, the number k of matching pairs must satisfy

$$k \geq 2pN - N. \qquad (3)$$

To see this, note that after the k pairs of matching tasks (totalling $2k$ tasks) have been chosen, one must choose another $2pN - 2k$ tasks without getting a match. If at some point the number of tasks remaining (i.e. unpicked) is less than the number of unmatched tasks that have been chosen, then by the pigeonhole principle (or some variant of it) we have a problem (you can't choose more than $m/2$ tasks from a total of m tasks without getting at least one pair). Now, the number of unpicked tasks is $2N - 2pN$, so we need to have $2N - 2pN \geq 2pN - 2k$. A little algebra gives (3). Thus if we let $\tau = \max\{0, 2pN - N\}$, then the expected number of pairs of matching tasks is given by

$$\frac{1}{\binom{2N}{2pN}}\sum_{k=\tau}^{pN}\frac{k}{(2pN-2k)!}\binom{N}{k}\prod_{i=k}^{2pN-(k+1)}2(N-i). \qquad (4)$$

Interestingly enough regardless of which scheme is used, the expected number of subtasks under the control of the adversary is exactly the same. Speci£cally, we claim that

$$pN(2pN-1) = \\ \frac{2N-1}{\binom{2N}{2pN}}\sum_{k=\tau}^{pN}\frac{k}{(2pN-2k)!}\binom{N}{k}\prod_{i=k}^{2pN-(k+1)}2(N-i) \qquad (5)$$

for all values of p such that pN is an integer. Note also the $2N-1$ term that changes the expected number of matching tasks to the expected number of matching subtasks. Stated another way, for a £xed nonnegative integer N, and integer L with $0 \leq L \leq N$ we seek to show that

$$L(2L-1) = \\ \frac{2N-1}{\binom{2N}{2L}}\sum_{k=\tau}^{L}\frac{k}{(2L-2k)!}\binom{N}{k}\prod_{i=k}^{2L-(k+1)}2(N-i), \qquad (6)$$

where here $\tau = \max\{0, 2L - N\}$. To see this, we £rst reduce the product to a more manageable form. The product contains $2L - (k+1) - k + 1 = 2L - 2k$ factors, so we have

$$\begin{aligned}\prod_{i=k}^{2L-(k+1)} & 2(N-i) = \\ &= 2^{2L-2k}(N-k)(N-(k+1))(N-(k+2))\dots \\ &\quad \dots(N-(k+(2L-2k-1))) \\ &= 2^{2L-2k}(N-k)(N-(k+1))(N-(k+2))\dots \\ &\quad \dots(N+k-2L+1) \\ &= 2^{2L-2k}\frac{(N-k)!}{(N+k-2L)!}\end{aligned}$$

Thus

$$\binom{N}{k}\prod_{i=k}^{2L-(k+1)}2(N-i) = \frac{N!}{k!}\frac{2^{2L-2k}}{(N+k-2L)!} \qquad (7)$$

and we can rewrite (2):

$$P(\text{exactly } k \text{ matches}) = \\ \frac{N!}{\binom{2N}{2pN}(2pN-2k)!}\frac{1}{k!}\frac{2^{2pN-2k}}{(N+k-2pN)!}. \qquad (8)$$

Substituting in (6), expanding the binomial coef£cient $\binom{2N}{2L}$ and moving some terms leaves the equation

$$\frac{(2N-1)(2L)!(2N-2L)!N!}{(2N)!}\times \qquad (9) \\ \sum_{k=\tau}^{L}\frac{k}{(2L-2k)!}\frac{1}{k!}\frac{2^{2L-2k}}{(N+k-2L)!} = L(2L-1).$$

So, how do we prove this? Well, let the math God go to work. First, note that by (2), (7), and the fact that the probabilities over all nonegative k of receiving exactly k matches must sum to one, we have, for each nonegative integer N, integer L with $0 \leq L \leq N$, and $\tau_{L,N} = \max\{0, 2L - N\}$

$$\begin{aligned}
1 &= \sum_{k=\tau_{L,N}}^{L} P(\text{exactly } k \text{ matches}) \\
&= \sum_{k=\tau_{L,N}}^{L} \frac{\binom{N}{k}}{\binom{2N}{2L}(2L-2k)!} \prod_{i=k}^{2L-(k+1)} 2(N-i) \\
&= \frac{(2L)!(2N-2L)!}{(2N)!} \times \\
&\quad \sum_{k=\tau_{L,N}}^{L} \frac{1}{(2L-2k)!} \binom{N}{k} \prod_{i=k}^{2L-(k+1)} 2(N-i) \\
&= \frac{(2L)!(2N-2L)!N!}{(2N)!} \times \\
&\quad \sum_{k=\tau_{L,N}}^{L} \frac{1}{(2L-2k)!} \frac{1}{k!} \frac{2^{2L-2k}}{(N+k-2L)!}
\end{aligned}$$

or equivalently,

$$\sum_{k=\tau_{L,N}}^{L} \frac{1}{(2L-2k)!} \frac{1}{k!} \frac{2^{2L-2k}}{(N+k-2L)!} = \frac{(2N)!}{(2L)!(2N-2L)!N!}. \quad (10)$$

Now, proving (9) is straightforward and trivial for N equal to zero or one. So, assume $N \geq 2$. Using the values of $N-1$ and $L-1$ in (10) gives

$$\begin{aligned}
&\sum_{k=\tau_{L-1,N-1}}^{L-1} \left[\frac{1}{(2(L-1)-2k)!} \frac{1}{k!} \frac{2^{2(L-1)-2k}}{((N-1)+k-2(L-1))!}\right] \\
&= \frac{2(N-1)!}{(2(L-1))!(2(N-1)-2(L-1))!(N-1)!} \\
&= \frac{(2N-2)!}{(2L-2)!(2N-2L)!(N-1)!}. \quad (11)
\end{aligned}$$

Now, we wish to make the substitution $k' = k+1$ in the left side of (11). Doing this moves the upper limit in the sum from $L-1$ to L, and changes the bottom limit from

$$\begin{aligned}
\tau_{L-1,N-1} &= \max\{0, 2(L-1) - (N-1)\} \\
&= \max\{0, 2L - N - 1\}
\end{aligned}$$

to $\max\{1, 2L - N\}$. Once the substition is made, however, it is easy to see that the k' equal zero term contributes nothing to the sum, so that we can replace $\max\{1, 2L-N\}$ with $\tau_{L,N}$. Making the substitution, then, gives

$$\begin{aligned}
&\sum_{k=\tau_{L-1,N-1}}^{L-1} \left[\frac{1}{(2(L-1)-2k)!} \frac{1}{k!} \frac{2^{2(L-1)-2k}}{((N-1)+k-2(L-1))!}\right] \\
&= \sum_{k'=\tau_{L,N}}^{L} \left[\frac{1}{(2(L-1)-2(k'-1))!} \frac{1}{(k'-1)!} \frac{2^{2(L-1)-2(k'-1)}}{((N-1)+(k'-1)-2(L-1))!}\right] \\
&= \sum_{k'=\tau_{L,N}}^{L} \frac{k'}{(2L-2k')!} \frac{1}{k'!} \frac{2^{2L-2k'}}{(N+k'-2L)!}
\end{aligned}$$

This last expression is exactly the sum on the left side of (9). Thus, combining this with (11) gives

$$\begin{aligned}
&\frac{(2N-1)(2L)!(2N-2L)!N!}{(2N)!} \times \\
&\quad \sum_{k=\tau}^{L} \frac{k}{(2L-2k)!} \frac{1}{k!} \frac{2^{2L-2k}}{(N+k-2L)!} \\
&= \frac{(2N-1)(2L)!(2N-2L)!N!}{(2N)!} \times \\
&\quad \frac{(2N-2)!}{(2L-2)!(2N-2L)!(N-1)!} \\
&= \frac{(2N-1)(2L)(2L-1)N}{2N(2N-1)} \\
&= L(2L-1).
\end{aligned}$$

Summarizing what we have shown, for both distribution schemes, the expected number of tasks and subtasks under control of the adversary are given by

$$\begin{aligned}
\text{Expected \# of subtasks} &= pN(2pN-1) \quad (12) \\
\text{Expected \# of tasks} &= \frac{pN(2pN-1)}{2N-1}
\end{aligned}$$

This is a double-edged sword. On the one hand, it would have been nice if the expected number of operations under the control of the adversary was less under our scheme than under simple redundancy. On the other hand, however, there is this argument: Under simple redundancy, an adversary who only controls the work units of two participants out of $2N$ has only probability

$$\frac{N}{\binom{N}{2}} = \frac{2}{N-1}$$

of receiving a matching pair of tasks, while under our unmodi£ed scheme, the adversary is guaranteed to possess one subtask that they completely control (i.e. can use to return fake corroborated results). What the expected value result says is that in fact in both schemes, the expected potential for corruption is the same, though the granularity is larger for simple redundancy (i.e. when they get matching tasks, they can do more damage, but it's harder to get more tasks).

Vertical paritioning also provides a bene£t in terms of stability, because the standard deviation of the number of subtasks under control of an adversary is zero. This is certainly not the case in simply redundancy (unless the adversary controls either all of the participants or none of them), where the variance of the number of subtasks is given by

$$\begin{aligned} &\mathrm{Var}((\#\text{ subtasks}) = \qquad (13) \\ &= \frac{2pN^2(2pN-1)(1-p)(2N(1-p)-1)}{2N-3}, \end{aligned}$$

for values of p for which pN is an integer. Note that if we consider this as a function of the variables p and N, then the function is symmetric about the line $p = 1/2$. That is, if we de£ne f by

$$\begin{aligned} &f(p, N) = \qquad (14) \\ &= \frac{2pN^2(2pN-1)(1-p)(2N(1-p)-1)}{2N-3}. \end{aligned}$$

then $f(p, N) = f(1-p, N)$. (Similarly, the right side of (25) is symmetric about the line $L = N/2$.)

Thus though the means are equal, in practice our scheme provides greater stability—for a given proportion of participants under the control of an adversary, there is a £xed number of subtasks they will control (in the absence of any veri£cation or other enhancement).

4.2 Detecting Malicious Behavior

In the remainder of this subsection, we examine the probabilities of detecting malicious activity assuming that the adversary will attempt to return an invalid result if and only if they have either both copies of a task (in simple redundancy) or both copies of a subtask (in collusion resistant redundancy).

We £rst look at simple redundancy, and speci£cally at the situation in which the supervisor attempts to detect cheating by verifying (i.e. computing) a single entire work unit. As before we assume that a proportion p of the $2N$ participants are under the control of the adversary, that pN is an integer value, which we denote by L, and that the $\tau_{L,N}$ are de£ned as in Section 4. We can estimate the probability of detecting the adversary under simple redundancy as follows. By our earlier work, the expected number of tasks under control of the adversary is $L(2L-1)/(2N-1)$. Since the supervisor is choosing one task from randomly from N, the probability of catching the cheater is

$$\frac{L(2L-1)}{2N-1}\frac{1}{N} = \frac{p(2pN-1)}{2N-1}.$$

This is turns out to be the exact probability, as we now show. Let A be the event that we detect them, and for $i \in [\tau_{L,N}, L]$ let B_i be the event that the adversary has been asigned exactly i pairs of matching tasks.

$$\begin{aligned} P(A) &= \sum_{k=\tau_{L,N}}^{L} P(AB_k) \\ &= \sum_{k=\tau_{L,N}}^{L} P(A|B_k)P(B_k) \\ &= \sum_{k=\tau_{L,N}}^{L} \frac{k}{N}P(B_k) \\ &= \frac{1}{N} \times \\ &\quad \frac{1}{\binom{2N}{2L}} \sum_{k=\tau_{L,N}}^{L} \frac{k}{(2L-2k)!}\frac{1}{k!}\frac{2^{2L-2k}}{(N+k-2L)!} \\ &= \frac{L(2L-1)}{N(2N-1)}. \end{aligned}$$

We now consider the analogous scenario under collusion resistant redundancy, but assuming that some number k of subtasks are veri£ed rather than a single task being veri£ed. Here, we know the exact number of subtask pairs assigned to the adversary: $L(2L-1)$. There are a total of $N(2N-1)$ distinct subtasks, so if we verify only a single randomly chosen subtask, the probability that we detect the adversary is $(L(2L-1))/(N(2N-1))$. Thus by doing only a fraction $1/(2N-1)$ of the work, we achieve the same probability of detecting the adversary. If we instead verify k of the tasks, then the probability of detecting the adversary is one minus the probability that we fail to check any subtask that they control:

$$1 - \frac{\binom{N(2N-1)-L(2L-1)}{k}}{\binom{N(2N-1)}{k}} \qquad (15)$$

We can get a lower bound for this quantity by noting that we are choosing subtasks without replacement. If we choose them with replacement, then the probability of choosing only tasks that the adversary does not control increases, so $P(A)$ decreases. But selecting with replacement implies a binomial distribution, and so we have

$$1 - \frac{\binom{N(2N-1)-L(2L-1)}{k}}{\binom{N(2N-1)}{k}} \geq 1 - \left(1 - \frac{L(2L-1)}{N(2N-1)}\right)^k.$$

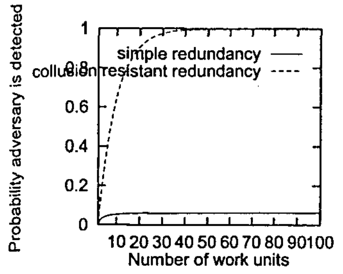


Figure 4. Graph showing the probability of detecting a cheater by checking a single task using simple redundancy versus checking an equivalent number of subtasks (i.e. $2N-1$ subtasks) under vertical partitioning. The graph shown is for $p = 0.25$.

Equivalently,

$$\begin{aligned} P(A) &= \\ &= 1 - \frac{\binom{N(2N-1)-pN(2pN-1)}{k}}{\binom{N(2N-1)}{k}} \\ &\geq 1 - \left(1 - \frac{p(2pN-1)}{2N-1}\right)^k \end{aligned} \qquad (16)$$

Figure 4 compares the functions for several values of p and N.

Let us generalize now to the situation in which simple redundancy is employed, but now the supervisor verifies (i.e. computes) m full tasks rather than just one. In this case, calculating the probability is a bit more involved, though we can still make an intelligent estimate. Once again the expected number of tasks under control of the adversary is $L(2L-1)/(2N-1)$. Now, however, the supervisor fails to catch the adversary only if all m of the tasks they select are not under the control of the adversary. There are expected to be

$$N - \frac{L(2L-1)}{2N-1}$$

of these. Thus the estimated probability of catching the adversary is

$$1 - \frac{\binom{N - \frac{L(2L-1)}{2N-1}}{m}}{\binom{N}{m}}$$

Of course the exact probability of catching the adversary is given by

$$\begin{aligned} P(A) &= \sum_{k=\tau_{L,N}}^{L} P(AB_k) \\ &= \sum_{k=\tau_{L,N}}^{L} P(A|B_k)P(B_k) \\ &= \sum_{k=\tau_{L,N}}^{L} \left(1 - \frac{\binom{N-k}{m}}{\binom{N}{m}}\right) P(B_k) \\ &= 1 - \sum_{k=\tau_{L,N}}^{L} \frac{\binom{N-k}{m}}{\binom{N}{m}} P(B_k) \\ &= 1 - \frac{1}{\binom{N}{m}} \sum_{k=\tau_{L,N}}^{L} \binom{N-k}{m} P(B_k) \\ &= 1 - \frac{1}{\binom{N}{m}} \sum_{k=\tau_{L,N}}^{L} \left[\binom{N-k}{m} \times \right. \\ &\qquad \left. \frac{N!}{\binom{2N}{2L}(2L-2k)!} \frac{1}{k!} \frac{2^{2L-2k}}{(N+k-2L)!}\right] \\ &= 1 - \frac{N!}{\binom{N}{m}\binom{2N}{2L}} \sum_{k=\tau_{L,N}}^{L} \left[\binom{N-k}{m} \times \right. \\ &\qquad \left. \frac{1}{(2L-2k)!} \frac{1}{k!} \frac{2^{2L-2k}}{(N+k-2L)!}\right] \\ &= 1 - \frac{(N-m)!}{\binom{2N}{2L}} \sum_{k=\tau_{L,N}}^{L} \left[\frac{(N-k)!}{(N-k-m)!} \times \right. \\ &\qquad \left. \frac{1}{(2L-2k)!} \frac{1}{k!} \frac{2^{2L-2k}}{(N+k-2L)!}\right] \end{aligned}$$

In addition to the increased probability of detecting an adversary, collusion resistant redudancy also provides a means for identifying many of the participants under the control of the adversary, because of the way the subtasks are spread throughout the $2N$ participants. If, in addition, we assume that the adversary adversary returns matching bad results whenever she controls a matching subtask, then *all* participants under the control of the adversary can be identified. The algorithm is as follows. First, whenever a bad result has been detected, the supervisor begins checking all other subtasks assigned to that adversary. Whenever a pair of matching bad results have been returned, the other participant who verified the result is then added to the list of adversarial participants, and all of their results are checked and so on. The number of checks required here is dependent on the proportion p of malicious participants, and is given by (figure this out).

For myself: looks like we will want some kind of list that we add new adversaries to. And we just keep checking subtasks for everything on the list until we have exhausted the list. We should run a sim and see how long this takes.

Of course there is then the question of what do you do with these participants once you know that they are bad. If you can blacklist, great, but you really can't.

Should also have a graph that does something like this: it looks at how many tasks have to be checked under simple redundancy in order to get a probability greater than 1/2 that the cheater will be detected (for a given p). Compare these amounts to the number of subtasks that need to be checked, and see what the total work difference is.

5 Clusters: Applying Collusion Resistance in Practice

Real distributed metacomputation can consist of millions of work units distributed to millions of participants. Certainly, our scheme as described thus far is not practical at these orders of magnitude, nor is it practical at values of N greater than around 50, since as mentioned earlier, bookkeeping costs can be signi£cantly increased at higher N values. Using this scheme in practice thus requires breaking the computation into several clusters, each of which consists of a reasonable sized number of work units. To keep our notation as consistent with the previous sections as possible, we will assume for the remainder, that the entire metacomputation consists of M work units. These are to be distributed to $2M$ participants (though these need not be $2M$ distinct participants). The M work units are to be divided into C clusters each containing N work units. Thus, $C = M/N$. The N work units in each cluster are to be distributed to $2N$ participants according to the vertical partitioning method.

Clustering has several advantages over pure vertical partitioning. First, unlike vertical partitioning, an adversary controlling multiple participants is no longer guaranteed to possess duplicates of any particular subtask, since tasks in different clusters are disjoint. There are also overhead advantages, since relatively low N values lead to decreased bookkeeping as compared to vertical partitioning. Most important, the notion of clustering provides the supervisor of a computation with signi£cant ¤exibility — by varying the parameters N and C the entire spectrum from simple redundancy (clustering with $N = 1$ and $C = M$) to vertical partitioning ($N = M$ and $C = 1$) can be covered.

We examine here how the introduction of clustering affects several of the probabilistic quantities previously considered. Determining closed form solutions for many of these quantities appears to be an intractible problem. Nevertheless, we have obtained exact expressions for these probabilities that allow them to be computed accurately.

We £rst consider the expected number of tasks under the control of the adversary if the computation uses a clustering scheme. Once again we assume that the adversary controls proportion p of the $2M$ participants in the computation, for a total of $2pM$ participants. Let $\{C_1, C_2, \ldots, C_C\}$ denote the clusters. We use vectors to describe speci£c assignments of participants to the adversary, with

$$\mathbf{v} = (k_1, k_2, \ldots, k_C)$$

denoting the event that the adversary has been assigned exactly k_i participants in cluster C_i. Such an assignment must of course always satisfy $\sum_{i=1}^{C} k_i = 2pM$. We de£ne the *subtask function*, $T(\mathbf{v})$ of an assignment vector $\mathbf{v}$, to be the number of subtasks controlled (i.e. the adversary has both copies of the subtasks) by the adversary given the participant assignment $\mathbf{v}$. Equivalently,

$$T(\mathbf{v}) = \sum_{i=1}^{C} \binom{k_i}{2}, \tag{17}$$

where we assume that the binomial coef£cient evaluates to zero if $k_i < 2$. Let $\mathcal{E}$ denote the set of all possible assignment vectors (which is of course determined by the values of $p, C, N, andM$) and for $0 \leq i \leq M(2N-1)$, let E_i be the set of vectors de£ned by

$$E_i = \{\mathbf{v} | T(\mathbf{v}) = i\}.$$

Finally, let S_i denote the event that the adversary controls exactly i subtasks, and let $P(\mathbf{v})$ denote the probability that the adversary receives a task assignment corresponding to vector $\mathbf{v}$. Then we have that the expected number of subtasks under control of an adversary with $2pM$ participants is given by

$$\begin{aligned} E(\text{\# of subtasks}) &= \sum_{i=0}^{pM(2N-1)} iP(S_i) \\ &= \sum_{i=0}^{pM(2N-1)} i \sum_{\mathbf{v} \in E_i} P(\mathbf{v}) \\ &= \sum_{\mathbf{v} \in \mathcal{E}} P(\mathbf{v}) T(\mathbf{v}). \end{aligned}$$

Determining the probability $P(\mathbf{v})$ of the adversary receiving the assignment corresponding to $\mathbf{v}$ is straightforward. There are $\binom{2N}{k_i}$ ways of receiving k_i participants from the $2N$ participants in cluster C_i, and there are a total of $\binom{2M}{2pM}$ groups of size $2pM$ participants that can be chosen from a group of size $2M$. Thus

$$P(\mathbf{v}) = \frac{1}{\binom{2M}{2pM}} \prod_{i=1}^{C} \binom{2N}{k_i} \tag{18}$$

Table 2. Task division and assignment for simple redundancy, clustering, and vertical partitioning. Assignments shown for $M = 4$ participants. The clustering example uses $C = 2$.

Simple Redundancy ($M = 4, N = 1, C = 4$)

Tasks			
1	2	3	4
A0	B0	C0	D0

Participants							
1	2	3	4	5	6	7	8
A0	A0	B0	B0	C0	C0	D0	D0

Clustering ($M = 4, N = 2, C = 2$)

Tasks			
1	2	3	4
A0	B0	C0	D0
A1	B1	C1	D1
A2	B2	C2	D2

Participants							
1	2	3	4	5	6	7	8
A0	A0	A1	A2	C0	C0	C1	C2
A1	B0	B0	B1	C1	D0	D0	D1
A2	B1	B2	B2	C2	D1	D2	D2

Vertical Partitioning ($M = 4, N = 4, C = 1$)

Tasks			
1	2	3	4
A0	B0	C0	D0
A1	B1	C1	D1
A2	B2	C2	D2
A3	B3	C3	D3
A4	B4	C4	D4
A5	B5	C5	D5
A6	B6	C6	D6

Participants							
1	2	3	4	5	6	7	8
A0	A0	A1	A2	A3	A4	A5	A6
A1	B0	B0	B1	B2	B3	B4	B5
A2	B1	B6	B6	C0	C1	C2	C3
A3	B2	C0	C4	C4	C5	C6	D0
A4	B3	C1	C5	D1	D1	D2	D3
A5	B4	C2	C6	D2	D4	D4	D5
A6	B5	C3	D0	D3	D5	D6	D6

and

$$E(\text{\# of subtasks}) = \qquad (19)$$
$$\frac{1}{\binom{2M}{2pM}} \sum_{\mathbf{v}=(k_1,\ldots,k_C)\in\mathcal{E}} \prod_{i=1}^{C} \binom{2N}{k_i} \sum_{j=1}^{C} \binom{k_i}{2}.$$

Equivalently,

$$E(\text{\# of tasks}) = \qquad (20)$$
$$\frac{1}{2N-1} \frac{1}{\binom{2M}{2pM}} \sum_{\mathbf{v}=(k_1,\ldots,k_C)\in\mathcal{E}} \prod_{i=1}^{C} \binom{2N}{k_i} \sum_{j=1}^{C} \binom{k_i}{2}$$

There are several special cases of input parameters (e.g. $p = (2pM - 1)/(2M - 1)$) for which one can easily calculate the value in (20). In each case, the value is identical to the values obtained for both pure vertical partitioning and simple redundancy. Moreover, we have computed several values for nontrivial parameter settings, and again in each case the expected number of tasks in all three scenarios is identical. We thus conjecture, but have not yet been able to prove, that (??) is in fact equal to the expected values for simple and vertical partitioning regardless of the value of C.

We look next at the probability of detecting an adversary who controls a single subtask, given that the supervisor verifies m subtasks. Let S_i once again denote the event that the adversary controls i subtasks, and let A be the event that their behavior is detected. Then

$$\begin{aligned} P(A) &= \sum_{i=1}^{pM(2N-1)} P(AS_i) \\ &= \sum_{i=1}^{pM(2N-1)} P(A|S_i)P(S_i) \\ &= \sum_{i=1}^{pM(2N-1)} P(A|S_i) \sum_{\mathbf{v}\in E_i} P(\mathbf{v}) \end{aligned}$$

Now, the supervisor will detect the adversary only if they verify one of the subtasks controlled by the adversary. Since there are a total of $M(2N - 1)$ subtasks in the computation, using the probability of the complement gives

$$P(A|S_i) = 1 - \frac{\binom{M(2N-1)-i}{m}}{\binom{M(2N-1)}{m}}.$$

Thus

$$\begin{aligned} P(A) &= \sum_{i=1}^{pM(2N-1)} P(A|S_i) \sum_{\mathrm{v}\in E_i} P(\mathrm{v}) \\ &= \sum_{\mathrm{v}\in\mathcal{E}} P(\mathrm{v})P(A|S_{T(\mathrm{v})}) \\ &= \sum_{\mathrm{v}\in\mathcal{E}} P(\mathrm{v}) \left(1 - \frac{\binom{M(2N-1)-T(\mathrm{v})}{m}}{\binom{M(2N-1)}{m}}.\right) \quad (21) \end{aligned}$$

Figure whatever compares this probability for the three schemes. In each case, we assume that the adversary has been assigned exactly two participants and in a manner most favorable for disruption. That is, in the data for simple redundancy, we assume that the adversary has been assigned identical work units and in clustering we assume that the two assigned participants are in the same cluster. In each case we assume that one equivalent task is veri£ed by the supervisor, so in pure vertical partitioning, $2M - 1$ subtasks are veri£ed, while in clustering $2N - 1$ subtasks are veri£ed. As expected, the probability for clustering falls between that for vertical partitioning and that for clustering, with the curve corresponding to increased C values moving toward the simple redundancy curve, and decreased C values moving toward the vertical paritioning curve.

Our £ndings are summarized in the following table. We assume as above that there are a total of M work units in the computation, and there are $2M$ participants. We divide the M participants into C clusters, each containing N work units and $2N$ participants, so $C = M/N$. We let

5.1 Augementing Our Strategy With Other Schemes

OK, so what augmented things are we talking about: First, varying the lengths of the subtasks. Second, using ringers (note that you have similar problems here if input have a speci£c order, since then the ringers can be recognized). The idea again is to make it impossible for the adversary to determine whether operations assigned to multiple tasks are ringers or not.

We also want to talk about doing this in a less than full redundancy style. That is, use a hybrid combination of ringers and this £ner grained redundancy.

6. 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 [20] 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 speci£c 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, 5, 11, 15], 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. [9] found that replication and voting schemes alone are not suf£cient 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 [19] proposes using cryptographic traces to detect tampering with agents. Speci£cally, 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 veri£cation 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 [13] discuss the idea of providing security for mobile agents by computing with encrypted functions [1, 12]. 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 dif£culty 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 bene£ts 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 func-

Table 3. Summary of results

Quantity	Simple Redundancy	Clusters	Vertical Partitioning	Summary
# rows in matrix	1	$2N-1$	$2M-1$	$SR \leq CLSTR \leq VP$
P(adversary cheats)	$\frac{1}{2M-1}$	$\frac{2N-1}{2M-1}$	1	$SR \leq CLSTR \leq VP$
# tasks compromised	1	$\frac{1}{2N-1}$	$\frac{1}{2M-1}$	$SR \leq CLSTR \leq VP$
Expected # tasks compromised	$\frac{pM(2pM-1)}{2M-1}$	$\frac{pM(2pM-1)}{2M-1}$?	$\frac{pM(2pM-1)}{2M-1}$	$SR = CLSTR = VP$
P(adv. detected)				$SR \leq CLSTR \leq VP$

tions exist, their methods, though interesting, present prac tical dif£culties.

In addition to the work of Golle and Mironov [7], two other works focus speci£cally on the issue of securing distributed metacomputations. Golle and Stubblebine [8] 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 ¤exibility, however, by varying the distributions that dictate the application of redundancy. Ef£cacy is measured by £rst 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 [10] deal with the problem of guaranteeing that a host participates in the computation, assuming that their goal is to maximize their pro£t by minimizing resources. The method involves recording traces of task execution. Speci£cally, 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 veri£er, 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.

Szajda, Lawson, and Owen [17] present two general schemes for using probabilistically applied redundancy to give applications greater resistance to cheating. They divide applications into two broad classes: non-sequential, in which tasks consist of independent operations; and sequential, in which the operations that constitute an individual task can have dependencies and must be executed in a speci£c order. Their technique for non-sequential applications is essentially an extension of the Golle and Mironov ringer scheme to more general functions. They handle sequential applications by breaking computations into several stages, assigning N tasks to $K > N$ participants, and using probabilistic veri£cation. Some application and platform speci£cs are mentioned, though only brie¤y and only in the context of applicability of their methods. They mention the possibility of colluding adversaries but assume that this occurs with low probability.

Sarmenta [14] proposes a credibility-based system in which multiple levels of redundancy are used, with parameters determined by a combination of security needs and participant reputations. Sarmenta notes that ??, blacklisting of participants is not possible in most distributed computations. Application and platform implementation speci£cs are not discussed. Collusion is considered during analysis of his system, but as in [17], the probability is assumed to be low.

7. Conclusions

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] 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.

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

[7] P. Golle and I. Mironov. Uncheatable distributed computations. In *Proceedings of the RSA Conference 2001, Cryptog-*

raphers' Track, pages 425–441, San Francisco, CA, 2001. Springer.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] L. Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. In *Proceedings of the ACM/IEEE International Symposium on Cluster Computing and the Grid*, Brisbane, Australia, May 2001.

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

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

[17] D. Szajda, B. Lawson, and J. Owen. Hardening functions for large-scale distributed computations. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 216–224, Berkeley, CA, May 2003.

[18] D. Szajda, B. Lawson, J. Owen, and E. Kenney. Issues in securing larg-scale distributed computations. Submitted to Proceedings of the 2004 ISOC Network and Distributed System Security Symposium.

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

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

A. The Variance of the number of tasks controlled by an adversary in simple redundancy

Collusion resistant redundancy provides a bene£t in terms of stability, because the standard deviation of the number of subtasks under control of an adversary is zero. This is certainly not the case in simply redundancy (unless the adversary controls either all of the participants or none of them). In this section we derive the variance of the number of subtasks under the control of the adversary using simple redundancy.

Following the work done in the previous section, if we let X be the random variable that describes the number of pairs of *tasks* (not subtasks) under the control of the adversary, then $E(X^2)$ is given by

$$E(X^2) = \frac{1}{\binom{2N}{2L}} \sum_{k=\tau}^{L} \frac{k^2}{(2L-2k)!} \binom{N}{k} \prod_{i=k}^{2L-(k+1)} 2(N-i). \quad (22)$$

We £nd a closed form for this using a method similar to what was used in the previous section—taking a known relation, in this case equation (9), and using the values $L-1$ and $N-1$ instead of L and N, and then making the substition of ($k' = k+1$ in the index of summation . This gives

$$\begin{aligned}
&\sum_{k=\tau_{L-1,N-1}}^{L-1} \left[\frac{k}{(2(L-1)-2k)!} \frac{1}{k!} \frac{2^{2(L-1)-2k}}{((N-1)+k-2(L-1))!} \right] \\
&= \sum_{k=\tau_{L,N}}^{L} \left[\frac{k-1}{(2(L-1)-2(k-1))!} \frac{1}{(k-1)!} \frac{2^{2(L-1)-2(k-1)}}{((N-1)+(k-1)-2(L-1))!} \right] \\
&= \sum_{k=\tau_{L,N}}^{L} \frac{k-1}{(2L-2k)!} \frac{1}{(k-1)!} \frac{2^{2L-2k}}{(N+k-2L)!} \\
&= \sum_{k=\tau_{L,N}}^{L} \frac{k}{(2L-2k)!} \frac{1}{(k-1)!} \frac{2^{2L-2k}}{(N+k-2L)!} \\
&\quad - \sum_{k=\tau_{L,N}}^{L} \frac{1}{(2L-2k)!} \frac{1}{(k-1)!} \frac{2^{2L-2k}}{(N+k-2L)!} \\
&= \sum_{k=\tau_{L,N}}^{L} \frac{k^2}{(2L-2k)!} \frac{1}{k!} \frac{2^{2L-2k}}{(N+k-2L)!} \\
&\quad - \sum_{k=\tau_{L,N}}^{L} \frac{k}{(2L-2k)!} \frac{1}{k!} \frac{2^{2L-2k}}{(N+k-2L)!}
\end{aligned}$$

Thus,

$$\sum_{k=\tau_{L,N}}^{L} \frac{k^2}{(2L-2k)!}\frac{1}{k!}\frac{2^{2L-2k}}{(N+k-2L)!}$$

$$= \sum_{k=\tau_{L-1,N-1}}^{L-1} \left[\frac{k}{(2(L-1)-2k)!}\frac{1}{k!} \frac{2^{2(L-1)-2k}}{((N-1)+k-2(L-1))!}\right]$$

$$+ \sum_{k=\tau_{L,N}}^{L} \frac{k}{(2L-2k)!}\frac{1}{k!}\frac{2^{2L-2k}}{(N+k-2L)!}$$

$$= \left[\frac{(L-1)(2(L-1)-1)}{(2(N-1)-1)(2(L-1))!}\times \frac{(2(N-1))!}{(2(N-1)-2(L-1))!(N-1)!}\right]$$

$$+ \frac{L(2L-1)(2N)!}{(2N-1)(2L)!(2N-2L)!N!}$$

$$= \frac{(L-1)(2L-3)(2N-2)!}{(2N-3)(2L-2)!(2N-2L)!(N-1)!} + \frac{L(2L-1)(2N)!}{(2N-1)(2L)!(2N-2L)!N!} \qquad (23)$$

Given this closed form solution for the sum, we have

$$E(X^2) =$$

$$= \frac{1}{\binom{2N}{2L}} \sum_{k=\tau}^{L} \frac{k^2}{(2L-2k)!}\binom{N}{k}\prod_{i=k}^{2L-(k+1)} 2(N-i).$$

$$= \frac{1}{\binom{2N}{2L}} \sum_{k=\tau}^{L} \frac{k^2}{(2L-2k)!}\frac{N!}{k!}\frac{2^{2L-2k}}{(N+k-2L)!}$$

$$= \frac{(2L)!(2N-2L)!N!}{(2N)!}\times \sum_{k=\tau}^{L} \frac{k^2}{(2L-2k)!}\frac{1}{k!}\frac{2^{2L-2k}}{(N+k-2L)!}$$

$$= \frac{(2L)!(2N-2L)!N!}{(2N)!}\times \left[\frac{(L-1)(2L-3)(2N-2)!}{(2N-3)(2L-2)!(2N-2L)!(N-1)!} + \frac{L(2L-1)(2N)!}{(2N-1)(2L)!(2N-2L)!N!}\right]$$

$$= \frac{L(2L-1)(L-1)(2L-3)}{(2N-1)(2N-3)} + \frac{L(2L-1)}{(2N-1)}$$

$$= \frac{L(2L-1)(2L^2-5L+2N)}{(2N-1)(2N-3)} \qquad (24)$$

It follows that

$$\mathrm{Var}(X) = E(X^2) - (E(X))^2 = \frac{L(2L-1)(2L^2-5L+2N)}{(2N-1)(2N-3)} - \frac{L^2(2L-1)^2}{(2N-1)^2}$$

so that the variance of the number of *subtasks* under the control of the adversary is given by

$$\mathrm{Var}((2N-1)X) = (2N-1)^2\,\mathrm{Var}(X) = \frac{L(2L-1)(2L^2-5L+2N)(2N-1)}{2N-3} - L^2(2L-1)^2.$$

With a little algebra, this can be reduced to

$$\mathrm{Var}((2N-1)X) = = \frac{2L(2L-1)(N-L)(2(N-L)-1)}{2N-3}. \qquad (25)$$

Replacing L with the value pN, we have, for values of p for which pN is an integer,

$$\mathrm{Var}((2N-1)X) = = \frac{2pN^2(2pN-1)(1-p)(2N(1-p)-1)}{2N-3}. \qquad (26)$$

Note that if we consider this as a function of the variables p and N, then the function is symmetric about the line $p = 1/2$. That is, if we de£ne f by

$$f(p,N) = = \frac{2pN^2(2pN-1)(1-p)(2N(1-p)-1)}{2N-3}. \qquad (27)$$

then $f(p,N) = f(1-p,N)$. (Similarly, the right side of (25) is symmetric about the line $L = N/2$.)

Thus though the means are equal, in practice our scheme provides greater stability—for a given proportion of participants under the control of an adversary, there is a £xed number of subtasks they will control (in the absence of any veri£cation or other enhancement).