# GIVS: Integrity Validation for Grid Security

**Giuliano Casale and**
**Stefano Zanero***

Dipartimento di Elettronica e Informazione,
Politecnico di Milano,
via Ponzio 34/5, I-2033 Milano, Italy
E-mail: casale@elet.polimi.it zanero@elet.polimi.it
*Corresponding author

**Abstract:** In recent years, Grid computing has been recognized as the next step in the evolution of distributed systems. However, only limited efforts have been made to address the security issues involved by these architectures. Questions like *"What does happen if a user of the grid is interested in making the overall computation fail?"* are still unanswered. In this paper we address the problem of granting the correctness of Grid computations. We introduce a Grid Integrity Validation Scheme (GIVS) that may reveal the presence of malicious hosts by statistical sampling of computations results. Performance overheads of GIVS strategies are evaluated using statistical models and simulation.

**Keywords:** Grid Computing; Integrity; Security

**Reference** to this paper should be made as follows: G. Casale, S.Zanero, 'GIVS: Integrity Validation for Grid Security', *Int.l Journal of Critical Infrastructures* , Vol. 2, No. 4, pp.xxx–xxx.

**Biographical notes:** Giuliano Casale received the MSc degree in computer science engineering in 2002 from Politecnico di Milano where is currently a PhD student. His research interests include performance evaluation and queueing networks modeling of complex computer and telecommunication systems.

Stefano Zanero received the MSc degree in computer science engineering in 2002 from Politecnico di Milano where is currently a PhD student. His research interests are related to information security, and include intrusion detection systems, web application security and computer virology. He has been an invited speaker in well known security conferences.

## 1 Introduction

Computational grids are touted as the next paradigm of computation [1]. Even now, they allow researchers to access the resources they need for the execution of computational intensive applications, such as simulations, mathematical problems,

Copyright © 200x Inderscience Enterprises Ltd.

and warehouse-scale data mining. A well known example of Grid environment is the Globus toolkit [2]. A full exploitation of the potential of such systems, however, would require grids to span a heterogeneous set of machines, extending beyond the boundaries of a single organization.

In this direction, the adoption of well designed schedulers ([3], [4], [5]) as well as the definition of computational power units for correctly addressing the distribution of multiple jobs among large grids [6] are primary research fields.

When a Grid extends beyond the systems of a single administrative authority it suddenly becomes a network of potentially untrusted nodes, on which a remote user submits a potentially harmful piece of code, or on the other hand potentially sensitive data, as already happens in generic peer-to-peer systems [7]. The ensuing problems of security and privacy have not been fully explored in literature yet.

We identify at least four different problems that arise when extending Grid computing beyond a trusted network:

1. Defending each participant from the effects of potentially malicious code executed on the Grid. This topic has been widely addressed by literature on mobile code, and there exist solutions such as sandboxing [8] and proof carrying code [9] [10].

2. A party could use the Grid in an unfair way, tapping computational resources without sharing his own. Game theoretic analysis suggests this as the most probable approach for a rational participant. Trust management systems have been proposed to solve this problem [11] [12].

3. The sensitivity of the data submitted for computation must be adequately protected. Until now, Grid computing deals with scientific data which is either of little sensitivity, or meant for publication. But in a near future grids are supposed to be used for industrial research or for marketing data (which is personal data and, as such, subject to regulations in most countries) and this problem will need to be addressed.

4. What does happen if one of the peers in the network is interested in making the overall computation fail ? It can manipulate the results of the tasks assigned to it. This problem has not been given a lot of consideration in previous literature.

Our paper will deal in particular with the last problem. We propose a general scheme, named Grid Integrity Validation Scheme (GIVS), based on problem replication and submission of test problems to the Grid hosts. We evaluate the effectiveness and the overhead of the proposed solution both analytically and through simulation, giving bounds on performance, security and computational overhead for various different strategies of validation.

The paper is organized as follows: in Section 2 we analyze the problem and we introduce our integrity scheme. In Section 3 GIVS is discussed in detail and the possible attacks are enumerated. Section 4 describes a statistical model for evaluating the security level and performance trade-offs of alternative validation scheme within GIVS. Section 5 discusses the introduction of a trust index to optimize distribution of problems. Finally, in Section 6 we draw conclusions and discuss future extensions of this work.

## 2 Problem statement and proposed solution

The problem of detecting whether or not a remote, untrusted machine is actually running a particular piece of mobile code is impossible to solve directly in a general manner. Checksums and digital signatures can be used to verify the identity of a piece of code we have received, but cannot be used to prove to a remote party that we are actually executing it. Cryptographic protocols for secure remote communication rely on the fact that each party holds a secret (either a shared password or the secret portion of an asymmetric keypair). There is no way to endow mobile code with a secret that cannot be accessed by someone controlling the host on which it is running. There is also no way to create a communication channel between the running code and a third party without MITM risks.

A possible way to solve the problem would be to design the code in such a way that tampering with results in a non-detectable way would require a much longer time than calculating the real solution. This is possible only in limited cases, and, additionally, it requires a correct estimation of the computation time, and thus implicitly relies on the fact that the malicious user will correctly declare the computational power he has available, something he cannot be trusted to do.

Since we cannot deal with the problem directly, we will need to check the correctness of the results by other means. In the best case, the problem we are dealing with will show some sort of checksum property, which will allow us to check the correctness of the results returned by the client without having to rerun the program. However, this is not general and not suitable for a general purpose Grid architecture. An additional problem is also that these types of checks would require either the programmer to take care of result validation himself, or to disclose the source code. Neither approach is desirable for a Grid architecture.

Our proposal is, then, to use some test problems to check the correctness of the results. The idea resembles a blind signature protocol [13], in which the submitter prepares $N$ similar documents, the signer opens $(N-1)$ documents checking they are correct, and then blindly signs the last document, with a probability $p = 1/N$ to be cheated. In our case, instead of documents we have computational tasks. We want to check that the peers are correctly processing the submitted data and code, by verifying just a sample of them. We obviously also require to perform our controls by harnessing the computational power of the Grid as much as possible, and not our own. This is the key idea of our scheme for integrity validation of Grid computation, or GIVS.

However, we need to understand how the number of test problems is related to the risk that someone can lie undetected, in order to balance the computational overhead required by cross checking with the security requirements that are necessary for the particular application. This topic is addressed in the following sections.

## 3 The Grid Integrity Validation Scheme

### 3.1 Test problems: classification, generation and use

In order to generate the test problems required for the integrity validation, a *bootstrap* phase is desirable. During the bootstrap, one or more trusted machines

compute correct solutions to a number of *test problems*. These problems are then used during normal Grid activity as test problems, and the solution computed by each untrusted hosts is compared to the correct solution. In fact, during normal Grid computations, each untrusted machine is asked to compute results to a set of problems, some of which are test problems. Clearly the latter act as canaries[a] against malicious users, since trying to corrupt the solution of a test problem would result in being identified as an attacker. While test problems seems an elegant and simple solution to the Grid integrity validation problem, some drawbacks can be identified. First, test problems require a bootstrap phase that slows down the computation startup and so it implies a waste of resources. Moreover, increasing the number of test problems could turn into a continuous performance overhead that could become unacceptable.

An alternative solution, that avoids part of these drawbacks, and that has to be preferred when a small number of trusted machines is available, consists in replicating a set of problems[b] of unknown solution throughout the Grid. The integrity validation is then performed comparing the different results given by the untrusted hosts to the same replicated problem. If all solutions match, the computations are all accepted as correct; instead, if a conflict is detected, a trusted machine is asked to compute the correct result, so that the malicious hosts can be identified. A clear advantage of this approach over the previous one is that in each period of the computation a set of unknown problems is solved as a side-effect of the security controls, and then the performance overhead can be limited. However, it must be noted that if all replicas of the same problem are sent to a group of cooperative malicious users, corrupted solutions could be misclassified as correct. Therefore, compared to the previous approach, we are now accepting a trade-off between the performance overheads imposed by the integrity validation and the degree to which we can rely on our own security controls. A quantitative comparison of the differences of these two strategies is then presented in Section 4.

### *3.2 Possible attacks to the validation scheme*

The objective of the malicious users is to make the computation fail. To do so, they must corrupt, without being detected as liars, a share of the problems. This critical target threshold depends on the type of computation and on its inherent robustness. This means that malicious users do not need to cheat on every problem they receive, but only on a share. However, malicious nodes are usually unable to know the number of problems that are being computed on the grid, and thus they are unable to properly estimate the optimal number of problems to corrupt in order to minimize the probability of being caught and maximize the probability of achieving the overall goal.

In addition, if a malicious user is ever able to detect that a problem is a test problem or a replica, it becomes useless, because the malicious user can calculate it correctly while cheating on other problems.

In the particular case where problems can be obfuscated (e.g. if the problems

[a] The name "canary" is used in information security engineering for the practice of inserting tamper-evident objects into systems for monitoring purposes.

[b] In the rest of the paper we will refer to these problems alternatively as *replicated problems* or just *replicas*.

are distributive with respect to multiplication for an arbitrary factor) the host submitting the problems could obfuscate them. However, the obfuscation scheme should be kept secret by the submitting host (thus violating Kerckhoffs's law[14] against security through obscurity), unless cryptographically strong. Unfortunately, only a very limited subset of problems can be solved in cryptographic format.

So how does a malicious user discover a test problem or a replica? Basically, if he receives a problem twice, he can guess that it probably is a test problem (either one computed by a trusted machine, or by another untrusted machine before), or a replica. Therefore his wisest choice is to correctly solve the problem. If we are dealing with isolated malicious users, this is clearly just a nuance (we can solve the problem with keeping track of the assigned problems). However, if two or more nodes agree to try to cheat, the problem becomes more complex. If a subset of $M$ machines are cooperating to cheat, if they receive any test problem in common, they can detect it. The same applies to replicas. Moreover, if any of the results previously computed by any of the $M$ machines is given to another machine as a test problem, it can be detected.

In addition, let us suppose that one of the machines has cheated on a problem, and this ends up being used as a test problem for another machine. If the second machine is honest, it will compute a different result, and the disagreement will set off an alarm. However, if the second machine is malicious, too, it can confirm the wrong result, which will probably be accepted.

If the control scheme is based solely on replicas, and a fixed number $n$ of replicas of a single problem exist on the Grid, and the $M$ rogue machines receive all the $n$ replicas, they can cheat on the problem result without being discovered. So, in this scheme, we must replicate each problem on a random number $n$ of hosts, with $n \geq 2$. With a larger number of replicas of a single problem, however, the probability of being detected and avoided by $n$ cooperative hosts becomes higher. As a basic precaution, we can try to assign test problems in such a way that every machine becomes a controller for every other machine at some point, and that controller and controlled are never constantly coupled.

The risks connected to this situation are evaluated in the next sections.

## 4 Performance-security trade-offs

### *4.1 Notation and basic definitions*

Let us begin to model a Grid composed of $H$ hosts, with $M \leq H$ malicious nodes that are cooperatively trying to disrupt the computation. In normal conditions we expect $M << H$.

Let $T$ be a period of activity of the Grid after the bootstrap phase. We discretize $T$ in the $K$ intervals $T_1, \ldots, T_K$, thus $\sum T_k = T$. In each period $T_k$, every host $h$ receives a collection $\mathbf{P_h^k} = (p_1^k, \ldots, p_P^k)$ of $P_h$ problems[c]. We assume the solution of each problem $p_i$ takes the same computational effort on all machines of the Grid. We can classify the problems $p_m \in \mathbf{P_h}$ in several classes: we denote test problems with $q_m$, replicas with $r_m$, and generic problems, which don't have any

[c] For the clarity of the notation, we will omit in the rest of the paper the indices $k$ when we do not need to refer to different time slices.

role in security controls, are denoted with $g_m$. Thus, for instance, the set $\mathbf{P_h} = (c_1, \ldots, c_{C_h}, r_1, \ldots, r_{R_h}, g_1, \ldots, g_{G_h})$ is composed of $C_h$ test problems, $R_h$ replicas and $G_h$ generic problems with $P_h = C_h + R_h + G_h$.

In the rest of this section we explore the two different strategies for GIVS described in Section 2: in the first one we use only test problems, while in the second all test problems are replaced with replicas. Our aim is to study, for each strategy, the performance overhead $OH$ introduced by the security controls, and the probability $p_{CF}$ of accepting a corrupted final value as a correct result.

Henceforth we will also limit our attention to Grid applications that can be effectively decomposed in a high number of subproblems, so that statistical methods can be applied with meaningful results. In particular, we will think of an "embarrassingly parallel" application, where the singular computational units are not deeply linked together.

### 4.2 First strategy: test problems

We now consider that each host receives a constant number of problems $P_h$, among which $G_h$ are generic problems, while $C_h = P_h - G_h$ are test problems. Then, no problem replicas are present in the sets $\mathbf{P_h}$ ($R_h = 0$). We denote with $\mathbf{q_h}$ the subset of test problems of $\mathbf{P_h}$ and with $C_h$ the cardinality of $\mathbf{q_h}$. All test problems $q_m \in \mathbf{q_h}$ are distinct and extracted randomly from a set of $C$ elements.

Under these assumptions, the performance overhead $OH$ of this scheme is given by $OH = H \cdot C_h$ since the solution of the test problems does not give any useful result for the applications running on the Grid.

We observe that the probability that two machines share the same control set is negligible. In fact, given that the probability of receiving a particular control set is the inverse of the number of possible distinct sets $\mathbf{q_h}$, we have

$$p(\mathbf{q_{h_k}}) = \binom{C}{C_h}^{-1} = \frac{C_h!\,(C - C_h)!}{C_h!}.$$

Now, observing that two random extractions $\mathbf{q_1}$ and $\mathbf{q_2}$ of the control sets are independent, the probability of two hosts $h_1$ and $h_2$ of receiving the same control set $\mathbf{q_{h_k}}$ is

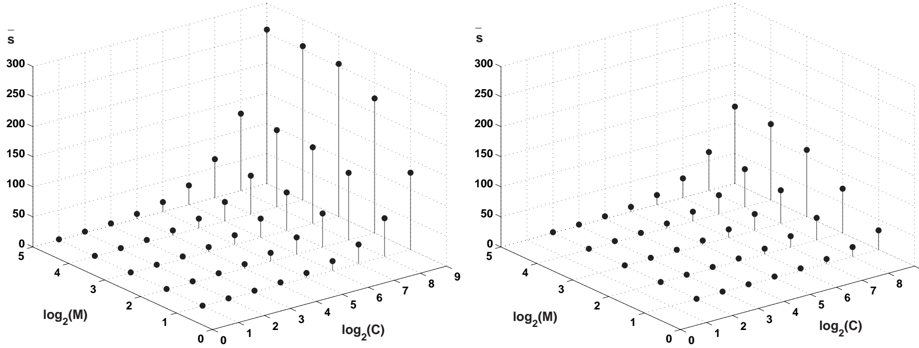
$$p(\mathbf{q_{h_1}} \equiv \mathbf{q_{h_2}}) = p(\mathbf{q_{h_k}})^2 = \left( \frac{C_h!\,(C - C_h)!}{C!} \right)^2.$$

In non-trivial cases this probability is extremely low. As an example, with $C = 5$ and $C_h = 3$, it narrows down to a modest 0.25%. As $C$ grows, this probability tends quickly to zero.

We now consider the probability $p(q_m \in \mathbf{q_h})$ that a particular problem $q_m$ belongs to a control set $\mathbf{q_h}$. This probability is modeled using an hypergeometric distribution[d], since it reduces to the probability of extracting a winning ball from

[d] We denote with *hypergeom*$(n + m,\, n,\, N,\, i)$ the probability of having $i$ successful selections from a hypergeometric distribution with $n$ winning balls out of $n + m$ and after $N$ extractions.

**Figure 1** Expected number of overlaps for different ratios of $C_h$ versus $C$. In the left figure, $C_h/C = 1/2$. On the right, $C_h/C = 1/4$.




a set of $C$, in $C_h$ extractions without replacement. Then we have

$$p(q_m \in \mathbf{q_h}) = hypergeom(C, 1, C_h, 1) =$$
$$= \binom{1}{1}\binom{C-1}{C_h-1}\binom{C}{C_h}^{-1} =$$
$$= \frac{(C-1)!}{C!}\frac{C_h!(C-C_h)!}{(C_h-1)!(C-C_h)!} = \frac{C_h}{C}$$

It is interesting to note that such probability is the same as if we would have dropped the no-replacement condition in the extraction, and so it suggest that, from a statistical point of view, the condition of uniqueness of the test problems in the same $\mathbf{q_h}$ can be dropped.

We now turn our attention to $p_{CF}$. In order to quantify such probability, we first need to estimate the mean number $\bar{s}$ of test problems that each user shares with the other malicious hosts. We can try to model such quantity as the probability $P_M(k)$ that each host of a group of $M$ has exactly $k$ overlaps with the others. For the case $M = 2$, host $i$ has probability $P_2(k) = hypergeom(C, C_h, C_h, k)$, since the winning problems are exactly the $C_h$ that belong to the control set $\mathbf{q_j}$ of host $j$. Unfortunately, if we try to extend this idea to the case with three or more hosts, we find that we cannot simply add together the pairwise probabilities, because we must account for the overlap between all three sets. This extension is quite difficult to derive analytically with a closed form, and due to its combinatorial nature, it raises some concerns about the computational complexity of a possible iterative description. Moreover, what we actually need is the mean number of overlaps $\bar{s}$, rather than the complete probability distribution, and so a simulation can provide accurate estimates of $\bar{s}$ at a reasonable computational cost and with limited modeling efforts.

In Figure 1 are shown the estimated $\bar{s}$, as a function of the number of malicious hosts $M$ and of the total number of test problems $C$ for the cases $C_h/C = 1/2$ and $C_h/C = 1/4$. All the experiments have been conducted using an Independent Replications Estimate of the mean $\bar{s}$ values with 50 replication of a total of over 1500 different samples. Confidence intervals have been computed for a 95% confi-

dence level; however, since such intervals have resulted to be extremely tight to the estimated $\bar{s}$, we have omitted them from all figures.

Assuming now to know the value of $\bar{s}$, we can model the success probability $p_{CF}$ of a malicious user using an hypergeometric distribution. We consider a malicious user $h$ who gives $b$ wrong answers to the problems of $\mathbf{P_h}$ and answers correctly to the $\bar{s}$ test problems that he has in common with other malicious users. In this case its probability of success $p_{CF}$ is the probability of never giving a wrong answer a test problem, i.e.

$$p_{CF} \cong hypergeom(P_h - \bar{s}_+, C_h - \bar{s}_+, b, 0) = \frac{(P_h - C_h)!(P_h - \bar{s}_+ - b)!}{(P - C_h - b)!(P_h - \bar{s}_+)!}$$

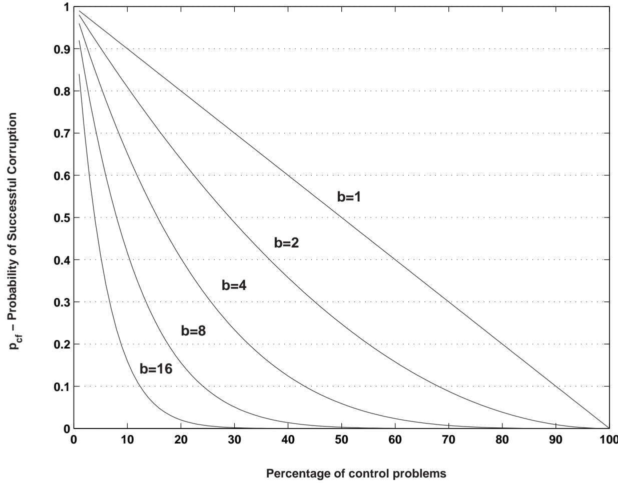
where $\bar{s}_+ = ceil(\bar{s})$ is used instead of $\bar{s}$ since the formulas require integer values. Please note that if $\bar{s}$ is integer the previous formula holds with the equality. The previous formula is defined only for $b < C_h - \bar{s}_+$, otherwise we have trivially that the user always corrupts a test problems and so $p_{CF} = 0$.

The qualitative behavior of $p_{CF}$ is shown in Figure 2, where the probability is plotted against the ratio of undetected test problems $(C_h - \bar{s}_+)$ to the number of problems $P_h$. The figure is plotted with $\bar{s} = 0$, since this term simply shifts the origin of the x-axis of a corresponding quantity (e.g. with $\bar{s} = 10$ $p_{CF}$ is 1 for $x = C_h/P_h = 10$ ). Several observations can be drawn from the shape of the $p_{CF}$. First, as intuitive, if a host tries to cheat on a single problem, its probability of success decreases linearly with the number of test problems. This has a very important consequence on every integrity validation schema: if the correctness of the global computation requires all subtasks to be computed correctly, a severe performance overhead is required to grant to grant the correctness of the results. Then, according to the model, a Grid provider would require a strong additional cost to grant the security of such applications. However, in many cases of interest a malicious user would need to affect a large number of tasks in to affect the global solution. For example, affecting a multiobjective optimization of a set of functions over a numerical domain would probably require the simultaneous corruption of the results on many different subdomains for each of the objective functions. In the case of multiple corruptions ($b >> 1$), the probability $p_{CF}$ drops quickly as the number of test problems grows. This suggests, if possible, to split the functional domain in the highest possible number of regions, although a trade-off with the involved communication overhead should be also taken into account [6]. However, even in this case, a large waste of computational resources is required to grant low values of $p_{CF}$. Therefore, using replicas could be a suitable way to handle these problems.

### *4.3 Second strategy: validation using replicated problems*

We are now going to extend the model developed for the first strategy to the case where replicated problems replace test problems. Problem sets have now the general form $\mathbf{P_h} = (r_1, \ldots, r_{R_h}, g_1, \ldots, g_{G_h}) = \mathbf{r_h} \cup \mathbf{g_h}$, and the replicas $r_m \in \mathbf{r_h}$ are drawn from a set of $U$ problems of unknown solution, in such a way that $\rho$ replicas of each problem are scheduled on the Grid. As explained before, the main advantage here is that, as a side effect of the security controls, the solution of $U$ problems is computed, and so less computational resources are wasted for security

**Figure 2** Probability $p_{CF}$ of accepting a corrupted result as function of $C_h/P_h$ (here is $\bar{s}_+ = 0$) and of the number of simultaneous wrong answers $b$




purposes. In fact, the minimum performance overhead when $\rho$ replicas of each of the $U$ problems are sent on the Grid is $OH^- = (\rho - 1)U$ that accounts for the fact that replicas also yield the solution of the unknown problems. However, we must handle the case where $m$ problems have been recognized as potentially corrupted, and so the performance overhead becomes $OH = (\rho - 1)U + 2m$ since we lack $m$ solutions and these must be recomputed by trusted machines. However, in normal conditions we expect $OH \cong OH^-$.

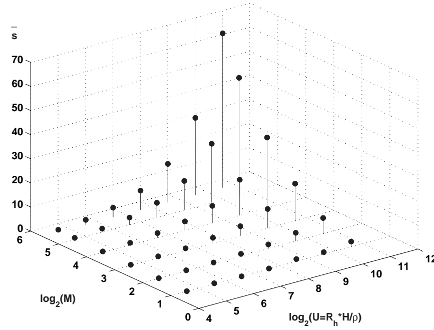
As observed for the first strategy, finding an analytical expression for the average number of overlappings $\bar{s}$ is quite difficult. Furthermore, the model of the second strategy should include additional constraints and problems that are difficult to be handled by means of a statistical model. In fact, if we want to ensure that exactly $\rho$ replicas of each of the $U$ problems are computed on the $H$ hosts, during the assignment of the replicas we must satisfy the constraint $C_h H = \rho U$.

Moreover, we observe that the constraint excludes the possibility of using a completely random assignment of the replicas: let us consider, for example, Table 4.3, where is shown a possible random assignment of $U = 10$ problems with a

**Table 1** An example of bad random assignment: any further assignment would alert $h_5$ that he is subject to a security control.

| *Host* | *Replicated problems* | | | |
|---|---|---|---|---|
| $h_1$ | 1 | 5 | 7 | 10 |
| $h_2$ | 3 | 7 | 8 | 10 |
| $h_3$ | 3 | 4 | 6 | 8 |
| $h_4$ | 1 | 4 | 5 | 6 |
| $h_5$ | 2 | 9 | **?** | **?** |
| *Unassigned replicated problems* | | | | |
| *Available* | 2 | 9 | | |

**Figure 3** Expected number of overlaps for the second strategy, with $H = 48$ and $\rho = 2$.




replication factor $\rho = 2$ (i.e. $\rho U = 20$). In this case we still need to assign two replicated problems to host $h_5$, but only the replicas of problems 2 and 9 are still unassigned. But this assignment is unacceptable, because it would break the condition that a host must receive distinct test problems. In this case, a random strategy would lock searching for an impossible assignment of the replicas.

A simple assignment scheme that avoids locking is the following: we assign randomly, until possible, the replicas. When a lock condition is detected, we change some of the previous assignment in order to solve correctly the assignment. For instance, in Table 4.3 we could easily avoid locking assigning replicas 2 and 9 to $h_4$ and moving 5 and 6 to $h_5$. It is clear that such heuristics is quite complex to be modeled analytically, while using simulation we can easily estimate the involved overlappings $\bar{s}$. In Figure 3 we show in logarithmic scale some numerical results from a simulation with $H = 48$ and $\rho = 2$: we can identify an exponential growth of the overlapping as the number of malicious users $M$ and the unknown problems $U$ grow. This turns to be a linear growth in linear scale, and so we conjecture an approximation $\bar{s} \cong f(M, U)$, where $f(\cdot, \cdot)$ is a linear function.

Given that $\bar{s}$ are known, the analysis of the probability $p_{CF}$ for the second strategy results in the same formula derived for the first strategy, where the quantities $C_h$ and $C$ are replaced respectively by $R_h$ and $U$. We also observe that, in most cases, the average overlappings $\bar{s}$ for the second strategy are generally lower that those of the first one, but this is affected by the presence of $H - M$ non-malicious hosts. The simulation of such users is a requirement for modeling the second strategy, since otherwise we would have trivially that $\bar{s} = R_h$. Such hosts, instead, are not considered in the simulations of the first strategy.

It is also interesting to note that $U$ is in general bigger than $C$ even with the optimal replication factor $\rho = 2$. This observation suggests that choosing between the two strategies should be done after a critical comparison of both the performance overheads $OH$ and the $p_{CF}$ for the two presented schemes, according to the specific characteristics of the Grid and its workload.

## 5 Introducing trust

We can also model the introduction of a trust index. We can suppose that a host, which has participated in the computation for a long time and behaved correctly, will continue to do so. Thus, the number of test problems and replicas for each host could slowly decrease over time.

In order to model the introduction of a trust scheme, we built a more complex simulation in Matlab. Each host receives a trust level of 1 at the beginning of the computation. At the end of each epoch, trust values are updated following these simple rules: if an host fails a test problem, it looses .5, while if it solves it correctly, it gains .25. Each host which has computed a mismatching replica receives a .05 penalty. Then, each host is classified as low, medium or highly trusted. A low trust host receives mostly test problem, and an equal share of true problems and replicas. A medium trust host receives mostly replicas, and an high trust host receives, symmetrically, mostly true problems. All the considerations seen in section 4.3 obviously apply also in this assignment phase. Malicious nodes cheat on an assigned percentage $p$ of problems, using node and group history as discussed to try to detect test problems and replicas.

The basic simulation parameters, unless otherwise stated, are: 500 nodes (10% of which malicious), 40 epochs, 20000 problems, of which 4000 are test problems, and 2000 are replicas with replication factor 2, leaving 12000 normal problems, and initially we suppose $p = 1$ (which means an evil host will cheat on any problem it does not recognize as a check or a replica). We consider a worst-case scenario where the malicious users are organized as a single group, thus sharing knowledge.

In the following, we denote with $S$ the number of replicas that were detected and thwarted, with $C$ the percentage of failed checks, with $Pow$ the share of computing power effectively used for solving real problems and not security checks, and with $Cor$ the percentage of computation effectively corrupted.

Table 4.3 summarizes what happens as the number of malicious nodes on the grid varies. In particular, when the number of abusers rises above 10 % there is an evident growth of the percentage of corrupted computations which are not detected by GIVS. This happens constantly: *there exists a critical threshold*, above which the system is unable to find out all the wrongdoers. Below this threshold, the trust system quickly finds out all the malicious nodes (e.g. if we are below 5% all the abusers are detected and ruled out within the first 5 epochs)tutti i malintenzionati vengono individuati. This threshold, experimentally, is dependent on the share of test problems and replicas, i.e. on the percentage of computational power we are willing to sacrifice in order to ensure correctness. Table 5 and 5 show instead how

**Table 2** Simulation results with different prevalence of malicious nodes

| *Evil Nodes %* | $S$ | $C$ | $Cor$ |
|---|---|---|---|
| 1% | 0 | 4.75 | 0.03 |
| 2% | 0 | 9.30 | 0.07 |
| 5% | 0 | 21.65 | 0.19 |
| 10% | 1.00 | 38.76 | 0.44 |
| 15% | 1.88 | 45.84 | 2.75 |
| 20% | 11.86 | 45.31 | 8.41 |

**Table 3** Simulation results with different quantities of replicas.

| *Replicas* % | $S$ | $Pow$ | $Cor$ |
|---|---|---|---|
| 5% | 0 | 77.5 | 0.83 |
| 10% | 0 | 75 | 0.57 |
| 15% | 0 | 72.5 | 0.49 |
| 20% | 1.00 | 70 | 0.44 |

**Table 4** Simulation results with different quantities of test problems

| *Check* % | $C$ | $Pow$ | $Cor$ |
|---|---|---|---|
| 5% | 46.3 | 85 | 4.2668 |
| 10% | 45.6 | 80 | 1.765 |
| 15% | 45.4 | 75 | 0.9727 |
| 20% | 38.7 | 70 | 0.3057 |

**Table 5** Percentage of corrupted problems, depending on various values of the probability of cheating $p$

| $p$ | 5% | 10% | 15% | 25% | 50% | 75% | 100% |
|---|---|---|---|---|---|---|---|
| $Cor$ | 0.29 | 0.54 | 0.75 | 1.40 | 1.25 | 0.84 | 0.30 |

the number of replicas and test problems affects $Cor$. There is a visible threshold in the second table: when the percentage of test problem is below the number of malicious hosts, the share of corrupted computations grows explosively.

Basically, what can be seen during the simulation runs is that the role of replicas is to spot suspect wrongdoers and lower their trust index: this causes an increasing number of checks to be assigned to them; when an host begins to fail checks it is progressively ostracized from the network. Critical thresholds exist (depending on the number of malicious hosts) beyond which some wrongdoers begin to cheat with impunity, so using a trust-based model is a double-edged sword.

Malicious hosts, however, could try not to cheat on each and every problem they receive, but on a percentage $p$ of them. As shown in Table 5, there exists an optimal (from the point of view of the attacker) percentage of corruptions which maximizes the overall corruption of the result.

## 6 Conclusions and future works

In this paper we have introduced GIVS, a Grid Integrity Validation Scheme, based on the idea of using test problems in order to detect the presence of malicious users. We have defined two different strategies for generating and distributing these problems, studying performance overhead and success probabilities of each using statistical models and simulation. We have shown that the cost for ensuring integrity is far from negligible. Thus, future extensions of our work include optimization techniques such as the introduction of trust and reputation mechanisms. Mechanism for introducing trust may help in reducing the overhead. However, if a malicious host needs to cheat on just a few problems in order to make the compu-

tation fail, it could wait until he is sufficiently trusted and then cheat with fewer probabilities for being detected. "Deeply parallel" applications also would need an extension of our analysis, because of the cascade effect of what happens in each stage of the computation over the next epochs. All these extensions, and better strategies for GIVS, will be further explored in future extensions of this work.

## Acknowledgements

This work was supported by the Italian Ministry of Education, Universities and Research (MIUR) in the framework of the FIRB-Perf project. The authors would like to acknowledge the helpful suggestions of the anonymous reviewers and participants of the International Workshop on Grid Computing Security and Resource Management (GSRM'05) for their helpful comments on an earlier version of this paper.

## References and Notes

**1** Ian Foster and Carl Kesselman. *The grid: blueprint for a new computing infrastructure.* Morgan Kaufmann Publishers Inc., 1999.

**2** I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int'l Journal of High Performance Computing Applications*, 15(3):220–222, 2001.

**3** Holly Dail, Henri Casanova, and Francine Berman. A modular scheduling approach for grid application development environments. Technical Report CS2002-0708, UCSD CSE, 2002. submitted to J. of Parallel and Distributed Computing.

**4** C. Liu, L. Yang, I. Foster, and D. Angulo. Design and evaluation of a resource selection framework for grid applications. In *Proc. of the 11th IEEE Symp. on High-Performance Distributed Computing*, July 2002.

**5** Carsten Ernemann, Volker Hamscher, and Ramin Yahyapour. Economic scheduling in grid computing. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2537 of *LNCS*, pages 128–152. Springer Verlag, 2002.

**6** Lorenzo Muttoni, Giuliano Casale, Federico Granata, and Stefano Zanero. Optimal number of nodes for computations in a grid environment. In *12th EuroMicro Conf. PDP04*, February 2004.

**7** Andrew Oram, editor. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies.* O'Reilly & Associates, Inc., 2001.

**8** Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proc. of the 14th ACM Symp. on O.S. princ.*, pages 203–216. ACM Press, 1993.

**9** Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for java. In *Proc. of the ACM SIGPLAN 2000 Conf. on Programming language design and implementation*, pages 95–107. ACM Press, 2000.

**10** George C. Necula. Proof-carrying code. In *Proc. of the 24th ACM SIGPLAN-SIGACT Symp. on Principles of programming languages*, pages 106–119. ACM Press, 1997.

**11** Ernesto Damiani, De Capitani di Vimercati, Stefano Paraboschi, Pierangela Samarati, and Fabio Violante. A reputation-based approach for choosing reliable resources in peer-to-peer networks. In *Proc. of the 9th ACM CCS*, pages 207–216. ACM Press, 2002.

**12** Karl Aberer and Zoran Despotovic. Managing trust in a peer-2-peer information system. In *Proc. of the 10th Int'l Conf. on Inf. and Knowledge Manag.*, pages 310–317. ACM Press, 2001.

**13** D. Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology - Crypto '82*, pages 199–203. Springer-Verlag, 1983.

**14** Auguste Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires*, IX:5–83, 161–191, Jan-Feb 1883.