Distributed Asymmetric Verification in Computational Grids

Michael Kuhn, Stefan Schmid, Roger Wattenhofer
Computer Engineering and
Networks Laboratory (TIK)
ETH Zurich
CH-8092 Zurich, Switzerland
{kuhnmi,schmiste,wattenhofer}@tik.ee.ethz.ch

Abstract

Lucrative incentives in grid computing do not only attract honest participants, but also cheaters. To prevent selfish behavior, verification mechanisms are required. Today's solutions mostly base on redundancy and inherently exhibit a considerable overhead. Often, however, the verification of a result takes much less time than its computation. In this paper we propose a distributed checking scheme that exploits this asymmetry. Our mechanism detects wrong results and excludes cheaters in a distributed manner and hence disburdens the central grid server. We show how the verification scheme is used in an application which aims at breaking the discrete logarithm problem by a parallel implementation of the Pollard-ρ algorithm. Our implementation extends the BOINC server software and is robust to various rational attacks even in the presence of colluders.

1 Introduction

Computational grids are typically used to solve large mathematical problems, for example for brute-force attacks of encrypted messages [10], to predict the climate [9], or to study the forming process of proteins [20] in computational biology. In these systems, a server distributes tasks to the participating clients; the clients subsequently process their input data, and send the calculated results back to the server. Systems such as *SETI@home* and *distributed.net* involve millions of users. The motivation for these users to contribute are manifold, including the altruistic desire to help [8] or the visual feast of nice pictures on a screensaver [23]. Another important motivation for participation is the assignment of *credit points* proportionally to a user's contribution. These points allow to reward hard-working clients in different ways; for instance, there may be a project website listing the project's top overall contributors, the "user of the day" [4], etc. A recently proposed lottery scheme [12] even offers monetary compensations for active participants.

Unfortunately, lucrative incentives also attract cheaters who seek to obtain these rewards with little or no contribution to the system. Such selfish behavior can be achieved by modifying the client software, for example.

This paper studies means to efficiently detect wrong results and to prevent cheaters from obtaining undeserved rewards in computational grids. After introducing and discussing the different parameters and properties influencing the design of cheater-resistant grid frameworks, we propose an algorithm for cheater identification. This algorithm is efficient in the sense that it aims at disburdening the server as much as possible by distributing the checking process to the participating hosts, implying a better scalability. In contrast to many existing solutions which often base on redundant computations, our mechanism exploits the fact that checking is often faster than computing the results from scratch.

In the second part of this paper, we apply our findings to the parallel Pollard-ρ algorithm for discrete logarithms [22, 26]. We implemented our algorithm on the *Berkeley Open Infrastructure for Network Computing* (BOINC) [2, 18] system, a grid framework written in C++ which has a large user basis. In order to incorporate distributed checking into BOINC, the server software had to be modified such that jobs can be assigned specifically to a certain client.

To the best of our knowledge, this is the first solution for efficient distributed checking in grid frameworks which exploits the asymmetry existing in many computational problems, coping effectively with both participants that are lazy during the checking process and with entire coalitions.

The remainder of this paper is organized as follows. In Section 2, related literature is reviewed. Our model and design choices are introduced in Section 3. Section´4 then presents the distributed checking mechanism. Simulation results are presented in the evaluation section, Section 5. We report on how to apply our algorithms to the parallel Pollard-ρ algorithm and on our BOINC implementation in Section 6. The paper is concluded in Section 7.

2 Related Work

The presence of cheaters in grid computing system is well-known and many countermeasures have been proposed in literature [13, 14, 16, 17, 24]. With the wide-spread use of rewarding mechanisms, also the problem of cheating is likely to gain importance. Recently, for instance, Douceur and Moscibroda [12] described a *Lottery Tree* mechanism, which encourages contributions (and the solicitation of new participants) in asymmetric *network-effect systems* such as BOINC. This mechanism motivates the use of monetary incentives to participate in such systems, which, of course, is also a good motivation for cheating.

To find wrong results created by lazy participants or cheaters, today, many systems simply replicate the work-units and use a majority rule to decide the correctness of results [19]. However, this approach often implies a waste of CPU cycles. For tasks whose verification is less expensive than redoing the computations, alternative approaches are preferable.

Golle et al. [16] propose a solution where a server secretely precomputes the results for a set of input values, the so called *ringers*. These ringers are then interspersed among the ordinary input values of a work-unit. If a lazy client does not compute the entire unit, it is likely to miss a ringer which can easily be detected by the server. The disadvantages of this approach are the need for precomputation, the redundant computation that inherently occurs, and the fact that a cheater is still likely to be undetected when cheating on a very small fraction of input values only. Lawson et al. have generalized this approach in [24].

Du et al. [14] present a commitment-based sampling scheme for cheater detection in grid computations based on *Merkle trees*. A server selects some samples which are assigned to a participant. The participant has to commit to its results which are subsequently checked. The drawback of their approach is the additional burden on the server which has to recompute some work-units itself.

There are also cryptographic protocols which allow to implement provably optimal systems in theory [1, 6]. However, these algorithms are often computationally expensive in practice.

In our system, the checking tasks are distributed to the participants. Distributing the load of checking has been studied before in the context of classic fault diagnosis. Beigel et al. [3], for example, have introduced a round-based distributed checking algorithm which is designed for a set of processors that have to check each other. The algorithm performs a preprocessing step in which processors are recursively paired up to test each other. *Hamiltonian paths* are then used to extract a strongly connected component containing only good clients. Unfortunately, this approach is problematic in our setting for different reasons. Most importantly, [3] assumes that processors are either good or faulty, but do not cheat on purpose. As a consequence, clients that correctly compute their ordinary tasks also behave properly in the checking phase. This does not hold for grid computing, where all sorts of rational attacks can occur, possibly even performed by colluding *groups* of clients rather than individuals. In particular, a node can stay honest during computation and only behave badly if it checks the result of another member of the collusion. Further, the round based algorithm cannot directly be applied to grid computing where hosts can join and leave at any time. Our system attempts to circumvent the resulting problems by using a more efficient scheme which is robust to various attacks, both in static and dynamic environments.

3 Model and Design Choices

This section introduces our model and our goals, and also gives some definitions. Moreover, we present different parameters and properties which we believe are useful to take into account when designing distributed checking algorithms.

We consider a grid framework consisting of a *server* and a potentially large number of *clients*. A client is the logic entity with which the server interacts. The server sends its *work-units* (or *tasks* or *jobs*) to clients, which return the corresponding set of *results*. A *participant* is a *user* who has registered an account for the project. She may use one or more *computers* (or *machines*) working for her in the project. Moreover, a computer can correspond to one or several clients. The computational resources of the clients are heterogeneous.

The server distributes two different kinds of tasks to the clients. Work-units are the main computational tasks of the grid computing framework; *checking units* require the client to perform a number of *checks* for the different results.

We assume that the main incentive for participation in the project are *credit points*: There may be websites listing the credit points earned by the different users, or there may even be ways to convert credit point into real money, by a lottery, for example.

In our system, a client earns credit points for both computing work-units and for computing checking units. The number of points is thereby proportional to the amount of work, such that a participant is indifferent between the two tasks.

It is of prime importance that the credit points be earned honestly, i.e., our algorithms' goal is to make sure that participants only get the credits they really deserve, and that the system is not flooded with wrong results.

In this paper we distinguish between two kinds of clients: *good* clients and *bad* clients (or *cheaters*). We consider a harsh model where all cheaters form a single coalition.

Today's systems such as Seti@Home are reported to have roughly 1% cheaters.

A checking algorithm which identifies dishonest behavior can achieve a higher effectiveness if it interplays with a mechanism to punish cheaters. We assume that a wrong result or a improper check implies that the corresponding client is a cheater.[1] Being debunked as a cheater basically implies that the corresponding client's user loses all its credits, and the corresponding account is closed. (See Section 4 for more details.)

Different parameters play an important role in the design and analysis of distributed checking algorithms. To characterize the power of cheaters, we introduce three variables:

1. *Maximum fraction of cheating clients (p)*: p is an upper bound on the fraction of cheaters among all clients. We assume that all cheating clients form one large coalition.

2. *Maximum computing power (q)*: q is an upper bound on the fraction of computing power the cheaters possess, compared to the total computing power in the system. In practice, q is limited, as an increase in q is expensive.

3. *Maximum fraction of incorrect results (r)*: r is an upper bound on the number of incorrect results the system is infiltrated with. Note that a small fraction of cheaters can feed a huge number of incorrect results into the system, since such results are typically random values that can be computed at virtually no cost. Thus, 1% of cheaters can easily be responsible for 99% of the results. We thus assume that $r = 1$, if no appropriate countermeasures are taken.

In the following, we discuss how these parameters influence the design of a distributed checking algorithm. Clearly, the larger p, the larger the likelihood of wrong results and checks. It is vital that a checking algorithm incorporates sufficient redundancy in the verification process to account for a given value of p. To keep p reasonably low we use two techniques: A punishment strategy that constantly removes cheaters from the system, and a partial coupling with q to mitigate *Sybil attacks* [11].

At the first glance, a natural approach would be to assign checking tasks proportional to the amount of results calculated by the individual clients. However, cheaters can provide results much faster than honest clients (as expressed by the value of r). Consequently, such an assignment would lead to a disproportional fraction of checking performed by the cheaters (r instead of p). To overcome this problem, a main ingredient to our algorithm is to have each client checking roughly the same amount of work, which ensures that cheaters get only a fraction p of the overall checking tasks assigned.

Our distributed checking algorithm makes use of the following three properties which facilitate a more effective verification process. We believe that in many architectures and applications, these properties can be fulfilled.

1. *Performance property*: The verification of a result using the checking function is considerably faster than the calculation of the result.

2. *Uniqueness property*: All reported results are either inherently unique or the checking function is able to verify each result's dependence on the input values.

3. *Fingerprint property*: A check calculates a fingerprint rather than a Boolean result. As a consequence only honestly performed checks can lead to a positive result.

The performance property is only important for performance reasons: If it is not fulfilled, our system will still work, but not more efficiently than simpler, alternative approaches. Many cryptographic protocols rely on such asymmetries.

The uniqueness property is useful for prohibiting *replay attacks*, that is, it prevents a client from simply sending old (correct) results over and over again.

The fingerprint property can be achieved by calculating a *witness* value. Such a witness value is a fingerprint which can be computed both during the calculation of the regular result and during the calculation of the check. The server can then compare the two *witnesses*. If they are equal, the check is positive, otherwise it is negative. The fingerprint property seeks to prevent *lazy checking attacks*: A lazy checker is a client that correctly computes its work-units but does not perform its checking tasks properly. Instead, it simply claims that the results are correct.

4 Distributed Checking

This section describes our distributed checking algorithm. A specific example of how to implement the ideas developed here is given in Section 6 for the parallel Pollard-ρ algorithm.

At the heart of our checking mechanism lies its distribution of checking tasks. While each client is allowed to compute as many results as possible, we maintain the invariant that all clients receive roughly the same amount of checking units. This is needed to prevent a small group of cheaters to check almost all results which in turn allows them to hide each other's wrong results. Due to our assumption that checking is much faster than recomputation, this

[1] We do not consider the problem that there may occur random errors at good clients; in practice, it would be possible to oblige clients performing *self-checks*; that is, after having computed a result, the client is responsible to briefly verify its result.

invariant can be maintained. However, we have to make sure that a user cannot create a large number of "virtual" clients; later in this section, we will briefly discuss how we defend against such Sybil attacks [11].

Under the assumption that the fraction of cheaters p is 20% or less, and given that checking is performed uniformly by all participants, the correctness of a result can be estimated by asking sufficiently many clients to perform a check and then taking the majority's decision. In our algorithm, the following adaptive strategy is applied: Each result is checked until the probability of a mistake falls below a certain threshold. Note that issuing the next check only after having received the result of the preceding check does not only allow to be adaptive with respect to the required upper bound on the error probability, but has also the advantage that a cheater does not know which other clients will verify the result in future, rendering collusion attacks more difficult.

Concretely, one objective of the distributed checking algorithm is to minimize the number of *false negatives* and the number of *false positives*. We define a false negative as a test which does not discover a wrong result, and a false positive a test which indicates that the result was wrong although it was actually correct.

Theorem 4.1 *Let α be the number of checks which are in favor of the result's correctness (including the result itself) and β be the number of checks stating that the result is wrong. Let P be the random variable denoting a false positive ($\alpha < \beta$), and N the random variable denoting a false negative ($\alpha > \beta$). We have*

$$\mathbb{P}[P] \leq \binom{\alpha+\beta}{\beta} p_P^{\beta} (1-p_P)^{\alpha}$$

$$\mathbb{P}[N] \leq \binom{\alpha+\beta}{\beta} p_N^{\beta} (1-p_N)^{\alpha},$$

where $p_P = \min(\beta/(\alpha+\beta), p)$ and $p_N = \min(\alpha/(\alpha+\beta), p)$.

PROOF. Let E denote the event that the result is correct. If no cheater tells the truth and if the fraction of cheaters is p, we have

$$\mathbb{P}[\alpha, \beta | E] = \binom{\alpha+\beta}{\beta} p^{\beta} (1-p)^{\alpha}.$$

Note that a cheater that behaves correctly for a given test implicitly reduces p. Let the actual fraction of wrong checks be $p' \leq p$. The only extremum of $\mathbb{P}[\alpha, \beta|E]$ is the maximum at $\delta\, \mathbb{P}[\alpha, \beta|E]/\delta\, p' = 0 \Rightarrow p' = \beta/(\alpha+\beta)$, hence the probability is maximized at $p' = \min(\beta/(\alpha+\beta), p)$. Therefore,

$$\mathbb{P}[\alpha, \beta | E] = \binom{\alpha+\beta}{\beta} p_P^{\beta} (1-p_P)^{\alpha}.$$

By the same argument, the claim also follows for false negatives. □

Note that often, quite large probabilities of false negatives can be acceptable: An example is the Pollard-ρ application considered in Section 6, a small number of wrong results hardly influence the total time until a real collision is found.

A simplified version of our distributed verification mechanism is summarized in Algorithm 1. We use θ_P and θ_N to denote acceptable thresholds for false positives and false negatives.

Algorithm 1 Distributed Checking Algorithm

1: **receive** result ρ with fingerprint F;
2: $\alpha, \beta := 0$;
3: **while** $((\alpha < \beta$ **and** $\mathbb{P}[\alpha, \beta|E] > \theta_P)$ **or** $(\alpha > \beta$ **and** $\mathbb{P}[\alpha, \beta|\bar{E}] > \theta_N))$
4: choose *online* client c *uniformly at random*;
5: assign ρ to c;
6: **if** $(c(\rho) = F)$ **then** $\alpha++$; **else** $\beta++$;
7: $\mathcal{S}_1 := \{c | c(\rho) = F\}$; $\mathcal{S}_2 := \{c | c(\rho) \neq F\}$;
8: **if** $\alpha > \beta$ **then** punish S_1; **else** punish S_2;

The parameters should be chosen such that the probability that a good client is accused falsely is small; however, the possibility will always exist. Observe that in our system, false positives are more problematic, as good clients are unjustly punished. One solution is to allow clients to contact the server and ask for a server check. If the server finds that the result was indeed correct, appropriate measures can be taken, for instance, all clients which falsely accused the server unjustly are punished. In order to protect the server from too many requests, we require a client to compute a certain number of work-units for free before it can contact the server. If it turns out that the participant was right, the credit points are rewarded ex post. Moreover, it is not rational to ask for a server check if the client's result is indeed wrong.

It remains to answer the question of how to guarantee that clients receive roughly the same amount of checking units. Our algorithm has to take into account the dynamics of the network, i.e., the fact that clients can join and leave at any time.

We propose the following solution: Whenever a result needs to be checked, we choose a client uniformly at random from all the clients which are currently online. We consider a client to be online if it recently contacted the server. If a client leaves before it has finished its checks, the unprocessed checks get distributed uniformly among the other clients. Figure 1 gives an example of the online times of six clients over time. Clients which are online at the considered moment are colored grey. The server notices that the second client has left and deletes its result queue; the pending

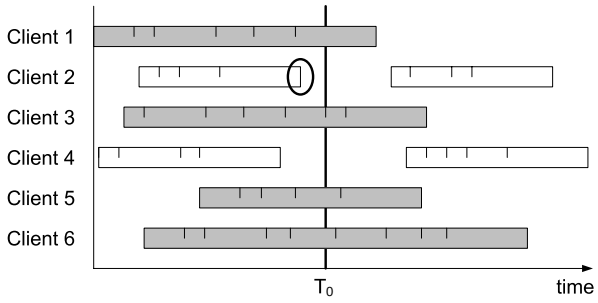

Figure 1. Visualization of checking process for 6 clients over time. At the time indicated by the vertical line, only the four grey clients are present. The server notices that the second client has left and distributes this client's pending checking jobs uniformly among clients 1, 3, 5, and 6. In addition, all new results are distributed for checking similarly.

jobs are distributed uniformly at random among the online clients. In addition, the newly arrived results, e.g., from client three, are also scheduled for checking.

Our solution is based on the assumption that cheaters can only perform Sybil attacks [11] in a very limited form. Fortunately, to the best of our knowledge, there have not been many Sybil attacks on today's grid computing systems. Moreover, a simple approach to restrain users from opening a large number of accounts, is the incorporation of Captcha mechanisms [7] or requiring users to register with a valid telephone or credit card number. In addition, we require that a new client first computes a certain number of checking units "for free", that is, without being rewarded any credit points. This renders the joining process expensive and the Sybil attack cumbersome, essentially coupling p to q. Concretely, in the beginning, we assign the new client only checking units for which we already know the result. Over time, the fraction of old checking units is reduced, and we add new checking units and work-units which have not been computed by the system at random; for these new jobs, the client gets credits as described above.

Finally, observe that the main mechanism of our algorithm, namely, that all clients perform roughly the same amount of checks, requires also that the performance differences of the clients are limited. Clearly, the more asymmetric the computational task, the more heterogeneity we can allow. However,—although very unlikely—during the grid's computations it might occur that we can no longer guarantee that a result is checked the mandatory number of times, because a small number of participants produces too many results. In this case, we issue only old checking units and wait until all checking units are finished, such that the clients can continue to earn credits. This comes at a certain cost as wasted resources; however, due to the small probability of the event, we believe is a useful solution to guarantee the results' integrity. Alternatively, it would also be possible to continue issuing work-units and just use a best effort approach to distribute the checks.

Possible extensions. So far, our checking algorithm employs a quite simple analysis to debunk cheaters. However, the server has actually access to much more information which could be exploited. We plan to integrate a more sophisticated analysis in future versions of our tool, and only briefly sketch some ideas here. The server knows the entire graph where a direct edge (u, v) denotes that client u has accused client v. The in-degree of a client is related to its trustworthiness. However, it is also important to take the clients' out-degree into account, as cheaters may seek to harm the reputation of good clients to hide other cheaters' behavior. We believe that an analysis in the spirit of *page rank* [5] can yield a more effective verification. In addition, in order to find colluders, clustering methods can be applied to the graph with the clients as vertices and where edges denote clients which computed the same fingerprints.

5 Evaluation

This section presents results of our simulation runs. The goal of our system is to balance the checks uniformly among the clients, while there is no such restriction on the distribution of the work-units. In all our experiments, we immediately replace an unmasked cheater with a new cheater in order to have a constant cheater fraction. Figure 2 shows a histogram of the workload due to the work-units (WU) and the checking units (CU) at the different clients. While the faster clients compute a larger share of all results, the checking workload is almost independent of the client's speed.

The number of checks per result depends on the probability of the false positives and the false negatives that we allow. Figure 3 shows the average number of pending checks at the clients for different error probabilities.

Finally, we have studied the effect of different cheater strategies. In the experiments, 10% of the client population were cheaters. The results are given in Figure 4. The first column states the percentage of results for which the cheater did not compute the correct result. The second column gives the percentage of wrong checks. In case a cheater checks another cheater's result, we assume that the result will always be accepted. The third column then states the percentage of results that the cheaters process, whereas the forth column gives the percentage of checks that are assigned to cheaters. Our simulations reveal that for all these cheater strategies, the influence of the bad clients, i.e., the percent-

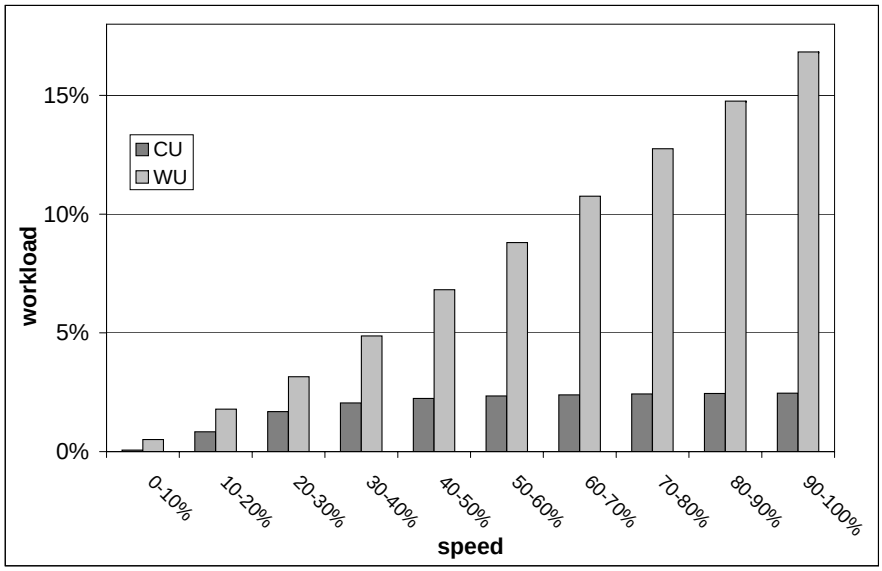

Figure 2. Histogram of the workload at the different clients. The speed of the clients is chosen uniformly at random from an interval $[1, 100]$ and it is assumed that checking is 50 times faster than computing a result. The fastest clients on the right of the x-axis compute much more work-units (WU) than the slower clients, but roughly the same amount of checking units (CU).

% Cheated on WU	% Cheated on CU	Processed WU	Processed CU
0%	0%	9.8%	9.9%
50%	5%	16.9%	8.3%
50%	50%	17.3%	8.3%
100%	10%	98.4%	10.7%
100%	100%	98.3%	10.3%

Figure 4. Effect of different cheater strategies.

age of checks they carry out, is limited.

6 Sample Application: Pollard-ρ in BOINC

This section shows how to apply our techniques for a sample application, namely for distributed verification in the parallel Pollard-ρ algorithm. We present our checking function in detail, and also briefly sketch the changes we had to implement in order to make the BOINC server capable of distributed checking.

6.1 The Parallel Pollard-ρ Algorithm

Let G be a *finite cyclic group* with a generator element $q \in G$. Given an element $r \in G$, we wish to find the smallest non-negative integer l such that $q^l = r$.[2] This problem is the well-known *discrete logarithm problem*, where l is called the *discrete logarithm* of r to the base q, i.e., $l = \log_q r$. While exponentiation can be done efficiently, it is widely believed that the opposite operation, i.e. computing the discrete logarithm, is computationally hard. In fact, many cryptographic protocols in use today rely on this assumption. In the following, we consider the problem where the group G is an *elliptic curve*. Although we seek to give as many details as possible to understand our distributed checking algorithms, we cannot provide all the cryptographic background and refer the interested reader to extensive literature in this field [21].

An efficient method to solve the discrete logarithm problem is the *Pollard-ρ algorithm* [22]. The algorithm com-

[2] We denote the group operation as a multiplication, and q^l denotes l subsequent applications of the group operation to q.

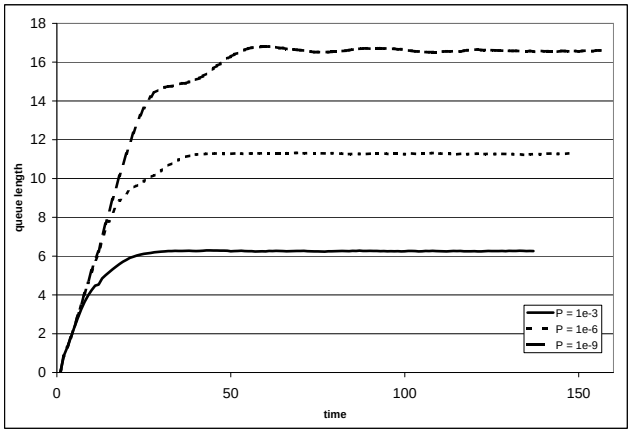

Figure 3. Number of pending checks at the different clients on average for three different error probabilities: $10^{-9}, 10^{-6}$, and 10^{-3} (for both false positives and false negatives). The simulation has been performed using 100,000 clients among which 20% were cheaters. The cheaters returned wrong results with probability 50% and did not check a result properly with probability 10%. Note that for larger error probabilities, the required number of correct results is reached earlier.

putes values a, b, A, $B \in Z_{|G|}$, such that $q^a r^b = q^A r^B$. From these values, l can be computed as the solution of the equation $(B - b)l = (a - A) (\text{mod } |G|)$. To find these exponents, the Pollard-ρ algorithm makes use of an *iteration function* $f : G \rightarrow G$: Given a starting value $q^{a_0} r^{b_0} = e_0 \in G$, a sequence e_i is computed according to the rule $e_{i+1} = f(e_i)$. As G is finite, this sequence is ultimately periodic so that there exist two uniquely determined smallest integers $T \geq 0, T_0 \geq 0$ such that $e_i = e_{i+T}$ $\forall i \geq T_0$. A pair (e_i, e_j) with $e_i = e_j, (i \neq j)$ is called a *collision*. A collision for which holds $A = a$ and $B = b$ is called a *duplicate*. The goal of the Pollard-ρ algorithm is to find a collision that is *not* a duplicate. From such a collision the discrete logarithm can be computed efficiently, as seen before. Pollard showed that due to the birthday paradox this theory can be applied to solve the discrete logarithm problem in G in expected runtime $O(\sqrt{|G|})$.

We have implemented a distributed version of the Pollard-ρ algorithm as follows. Each client c receives a pair of random values (a_0, b_0) from the server, calculates e_0 from them, and then performs the iterations according to the following iteration function on the elliptic curve:

$$f(e_i) = e_{i+1} = \begin{cases} e_i \cdot q & \text{if } e_i \in \mathcal{S}_1 \\ e_i^2 & \text{if } e_i \in \mathcal{S}_2 \\ e_i \cdot r & \text{if } e_i \in \mathcal{S}_3 \end{cases}$$

where $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3$ are random sets of roughly the same size which form a partition G. The series a_i and b_i are updated accordingly. The idea of the iterations is to perform a pseudo random walk (the sequence e_i) on the elliptic curve.

While a client could basically report all points it computes on the elliptic curve to the server such that the server can look for collisions, this constitutes a large burden on the server-side. Therefore, in our implementation, the clients only send a subset, the so-called *distinguished points* (e.g., points p for which a certain hash function $hash(p) = 0$), to the server. Note that since once two clients are on the same path, i.e., once they have computed the same results, they will continue computing the same results forever. Hence, collisions will be found quickly even if only distinguished points are reported. The situation is depicted in Figure 5.

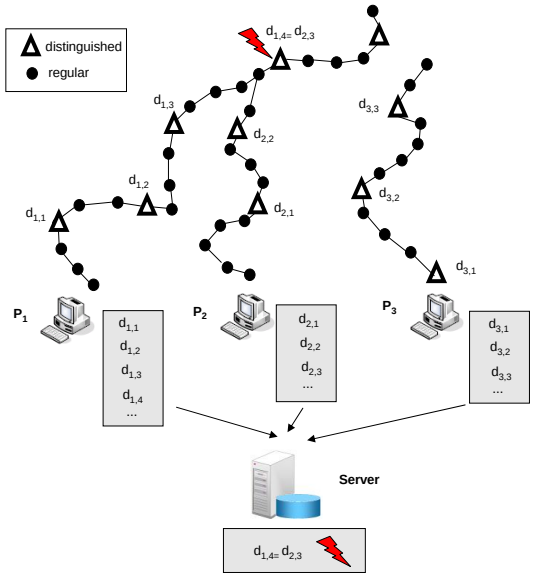

Figure 5. Distributed Pollard-ρ algorithm: The clients iterate along the group elements and report distinguished points to the server. The server maintains a database where points are stored for collision detection.

6.2 Distributed Checking

We have implemented our distributed checking mechanism in BOINC. Thereby we have extended the framework

in several ways such that it takes the specific properties of the distributed Pollard-ρ algorithm into account. This section presents some of our defense mechanisms against cheaters in more detail.

In order to apply our distributed checking algorithm, we have to come up with an efficient verification method to check the proper execution of a work-unit by a client (cf Section 3). After the completion of a result, a client reports the following:

- The distinguished point p_d, together with the corresponding exponents a_d and b_d.
- The point p_x that was reached x steps before the distinguished point, together with the corresponding exponents a_x and b_x. The value x is chosen by the server and is considerably smaller than the expected number of steps E to reach a distinguished point from a random starting point, i.e. $x \ll E$.
- The series S_k of steps that lead from p_x to p_d. This is our *witness* value.

The input of a checking unit consists of p_x, a_x, b_x, and $y > x$ chosen by the server. The verification process works as follows: The verifier iterates a maximum of y steps from p_x and records these steps. If it reaches a distinguished point, the iteration process is aborted and the performed series of steps S'_k is returned to the server, together with the distinguished point p'_d and the corresponding exponents. The server now verifies whether $S_k \equiv S'_k$ and $p'_d = p_d$. If so, the check is good, otherwise it is not.

In the following we will show that this checking mechanism fulfills the requirements from Section 3. The *performance property* is clearly met, as only $x \ll E$ number of iterations have to be performed. The *uniqueness property* follows from the characteristics of the Pollard-ρ algorithm. In our system, a client does not receive any credit points for a duplicate. As the probability to encounter a *duplicate* is roughly the same as the probability to encounter a collision that solves the problem, i.e., roughly approximately $1/\sqrt{|G|}$,[3], the resulting unfairness can be neglected. The *fingerprint property* requires that the witness value can only be calculated by honest computation and honest verification. The verification part follows from the fact, that it is impossible to guess the correct series, if the iterations are not honestly performed. The computation part is a slightly more involved. We have to show that it is not possible to create any points p_x, p_d (distinguished) and a correct series between them, without actually performing all the iterations from the random start values a_0 and b_0. As we require p_d to be a new distinguished point, it suffices to prevent a client from finding such a point (together with the correct exponents) in an "illegal" way. To find such a point, in average $H/2$ points have to be tried, where H is the cardinality of the output-domain of the underlying hash-function. The best a cheater can do is choosing arbitrary values $\tilde{a}$ and $\tilde{b}$ and check whether the resulting point $q^{\tilde{a}}r^{\tilde{b}}$ is distinguished.[4] Notice that the expected number of points to be visited is equal to the expected number of iterations. Therefore the potential gain is defined by the speedup of randomly selecting a point over calculating one iteration step. Once a candidate p_d is found, the corresponding point p_x and the steps from p_x to p_d have to be determined. The inversion of the iteration steps is considerably more expensive than the forward direction and for a randomly selected point a backward path might even not exist (cf Section 6.4). Thus, if the values x and H are chosen appropriately, properly following the path is the most rational option. Note that it is still possible to choose own values for the starting values $\tilde{a}_0$ and $\tilde{b}_0$ and then starting the iteration process. However, in this case a client equally contributes to the overall project and does therefore not need to be punished.

6.3 Remark on BOINC Implementation

In addition to the implementation of our parallel Pollard-ρ algorithm, several changes were necessary at the original BOINC code. In order to extend BOINC for distributed checking mechanisms, we had to adapt the BOINC server such that it generates two different kinds of jobs, regular work-units and checking units. Unfortunately, so far BOINC does not support the assignment of specific work-units to dedicated hosts as required by our verification scheme. We modified the scheduler such that every time a client arrives, it decides—depending on the client's state—whether the client is assigned a work-unit or a checking unit, and in case of checking units, selects an appropriate one.

6.4 Analysis of Backward Iteration

Instead of calculating an entire work-unit, a cheater might try to find a random distinguished point p_d, together with the parameters a_d and b_d. Such a point can only be found by choosing a and/or b at random and then checking whether the property of a distinguished point is given. In expectation, the same number of random points have to be generated for this property to be fulfilled as there are iter-

[3] In case of the ECC-109 challenge, the probability is approximately 2^{-55} for example.

[4] Note that although the server would learn about roughly the same amount of distinguished points this way, this behavior is problematic, as we lose the property that if two clients find the same point on the elliptic curve they will remain on the same path until a distinguished point is found; as a consequence, the expected running time of the algorithm is much larger.

ation steps per work-unit.[5] Moreover, observe that finding such a point is not sufficient, as our checking function requires the client to report p_x. Thus, the cheater is bound to iterate back x steps from p_d in order to find a valid point p_x. Fortunately, this backward iteration attack is expensive and irrational, as there is a certain probability that for a given point, there do not exist any predecessors.

Our simulations (cf Figure 6) show that the likelihood that all required predecessors exist for a given point declines sharply with the number of iterations x. For fifty iterations, for example, the success probability is already below 10%. Generating ten random points is more expensive than one forward iteration step. Thus, in expectation, this attack requires more computational resources than computing the work-units properly.

6.5 Towards the ECC Challenge

To increase trust into their security solutions, and stimulate research in the area of elliptic curve based cryptography, *Certicom*[6], has launched the *ECC challenge*[7] in 1997. The challenge consists of a series of discrete logarithm problems on elliptic curves of increasing complexity. Each solved problem is rewarded with a price ranging from $5k$\$ to $100k$\$, dependent on the problem's complexity. Clearly, attacking such a problem seems to be an ideal application for the grid computing system presented throughout this paper. In the following, we briefly want to sketch the required modifications in order to participate in this challenge. Some of the outlined optimizations are, in fact, already integrated in the client.

It is important to note that today even with a large number (say 1 million) of participants only the easiest not yet solved problem in this series is computationally feasible today. This problem is, however, defined on a special variant of elliptic curves that, the so called *Koblitz curves*. These curves are defined over the finite field F_{2^m}, and allow for a speedup of a factor $\sqrt{m}$ if the Pollard-ρ algorithm is modified accordingly, as pointed out by Gallant et al. [15] as well as by Wiener and Zuccherato [27]. In particular, these authors show that to a certain point $X_0 = a_0P + b_0Q$ on the curve, $m-1$ corresponding points $X_i = a_iP + b_iQ$ (including the values a_i and b_i) can be found with little effort. Selecting the point with lowest x-coordinate after each iteration reduces the search space by a factor of m which results in the mentioned speedup of $\sqrt{m}$. The most efficient way to calculate these points makes use of the so called *normal basis representation* for elliptic curves. Our client is currently implemented using the less efficient polynomial representation.

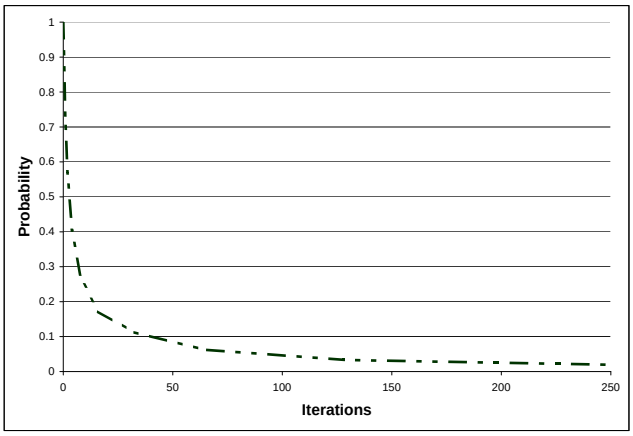

Figure 6. Success probability of backward iteration attack for different numbers of iterations.

Wiener and Zuccherato further present two additional improvements to the original Pollard-ρ algorithm, which are not restricted to Koblitz curves. First, they point out that, given a point $X = aP + bQ$, it is cheap to find the point $-X = (-a)P + (-b)Q$. They thus suggest to only consider the point with smaller y-coordinate after each iteration, which in fact reduces the search space by a factor of 2 and consequently results in a further speedup of $\sqrt{2}$. This speedup is bought at the cost of some countermeasures against trivial two-cycles, which complicates the implementation.

Finally, Wiener and Zuccherato refer to the work of Teske [25] that proposes the use of a better iteration function, which distinguishes 20 rather than 3 cases. This iteration function results in an effective running time very close to the theoretic estimates.

7 Conclusion

It is widely believed that the Internet will become more service-oriented in the near future. In line with this trend we can observe the emergence of computing frameworks consisting of a large number of machines on which scientists or companies run their tasks or applications for a certain time period and cost. To ensure the scalability of such services, many of them require the participants to contribute in order to get benefits. This tendency becomes manifest most significantly in peer-to-peer systems such as BitTorrent and is also present in grids.

It is often necessary to verify the work contributed by the participants. This paper has studied mechanisms to check

[5] Note that choosing a distinguished point first and then looking for a and b implies solving the discrete logarithm problem.

[6] See `www.certicom.com`.

[7] See `www.certicom.com/index.php?action=ecc,ecc_challenge`.

the results returned by the hosts in a grid computing framework in a distributed manner. By performing only a small number of additional checks, cheaters are prevented from obtaining credit points which they do not deserve. The system is resilient to more sophisticated rational attacks such as replay attacks and lazy checking. In preliminary experiments we have finally shown that the delineated mechanisms can smoothly be integrated in the widespread grid computing framework BOINC.

There remain several interesting directions for future research. For instance, while we have shown how to cope with selfish participants in our system, the question of how to prevent malicious attacks such as DDoS attacks or external attacks, remains open.

It would also be interesting to study distributed verification in other settings, such as a Napster-like peer-to-peer system where participants are required to check whether the shared files are of good quality, e.g., whether they do not contain a virus, or a collaborative editing project such as Wikipedia.

Acknowledgments

We are grateful to Fabian Kuhn, Christoph Renner, and Christoph Schwank for interesting discussions. We would also like to thank Patrik Hubschmid for his expertise and advice on our questions related to group theory. Our research is in part supported by the Swiss National Science Foundation.

References

[1] W. Aiello, S. N. Bhatt, R. Ostrovsky, and S. Rajagopalan. Fast Verification of Any Remote Procedure Call: Short Witness-Indistinguishable One-Round Proofs for NP. In *Proc. 27th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 463–474, 2000.

[2] D. P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proc. 5th IEEE/ACM International Workshop on Grid Computing.*, pages 4–10, 2004.

[3] R. Beigel, G. Margulis, and D. A. Spielman. Fault Diagnosis in a Small Constant Number of Parallel Testing Rounds. In *Proc. 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 21–29, 1993.

[4] Berkeley Open Infrastructure for Network Computing. BOINC Combined Statistics. In *http://boinc.netsoft-online.com/*, 2006.

[5] S. Brin and L. Page. The Anatomy of Large-Scale Hypertextual Web Search Engine. In *Proc. Computer Networks and ISDN Systems*, 1998.

[6] C. Cachin, S. Micali, and M. Stadler. Computationally Private Information Retrieval with Polylogarithmic Communication. *Lecture Notes in Computer Science*, 1592:402–414, 1999.

[7] Carnegie Mellon University. What is reCAPTCHA? 2004.

[8] C. Christensen, T. Aina, and D. Stainforth. The Challenge of Volunteer Computing With Lengthy Climate Model Simulations. *E-SCIENCE, IEEE Computer Society*, 2005.

[9] Climate Prediction. http://www.climateprediction.net/.

[10] distributed.net. http://www.distributed.net/.

[11] J. R. Douceur. The Sybil Attack. In *Proc. 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 251–260, 2002.

[12] J. R. Douceur and T. Moscibroda. Lottery Trees: Motivational Deployment of Networked Systems. In *Proc. ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2007.

[13] W. Du and M. T. Goodrich. Searching for High-Value Rare Events with Uncheatable Grid Computing. In *Proc. 3rd Applied Cryptography and Network Security Conference (ACNS)*, pages 122–137, 2005.

[14] W. Du, J. Jia, M. Mangal, and M. Murugesan. Uncheatable Grid Computing. In *Proc. 24th International Conference on Distributed Computing Systems (ICDCS)*, pages 4–11, 2004.

[15] R. Gallant, R. Lambert, and S. Vanstone. Improving the Parallelized Pollard Lambda Search on Binary Anomalous Curves. *Mathematics of Computation*, 1998.

[16] P. Golle and I. Mironov. Uncheatable Distributed Computations. In *Proc. 2001 Conference on Topics in Cryptology*, pages 425–440, 2001.

[17] P. Golle and S. G. Stubblebine. Secure Distributed Computing in a Commercial Environment. In *Proc. 5th International Conference on Financial Cryptography*, pages 289–304, 2002.

[18] http://boinc.berkeley.edu/. BOINC - Berkeley Open Infrastructure for Network Computing.

[19] L. Kahney. Cheaters Bow to Peer Pressure. *Wired Magazine, February 15*, 2001.

[20] S. M. Larson, C. D. Snow, M. R. Shirts, and V. S. Pande. Folding@Home and Genome@Home: Using Distributed Computing to Tackle Previously Intractable Problems in Compuational Biology. *Computational Genomics*, 2002.

[21] A. J. Menezes, P. C. van Oorshot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[22] J. M. Pollard. Monte Carlo Methods for Index Computation $(\mathrm{mod} p)$. *Math.Comp*, 32(143):918–924, 1978.

[23] M. Shirts and V. S. Pande. Screensavers of the World, Unite! *Science*, 290(5498):1903–1904.

[24] D. Szajda, B. Lawson, and J. Owen. Hardening Functions for Large Scale Distributed Computations. In *Proc. 2003 IEEE Symposium on Security and Privacy*, 2003.

[25] E. Teske. Speeding Up Pollard's Rho Method for Computing Discrete Logarithms. In *Proc. 3rd International Symposium on Algorithmic Number Theory (ANTS)*, pages 541–554, 1998.

[26] P. C. van Oorschot and M. J. Wiener. Parallel Collision Search with Application to Hash Functions and Discrete Logarithms. In *Proc. 2nd ACM Conference on Computer and Communications Security*, pages 210–218, 1994.

[27] M. J. Wiener and R. J. Zuccherato. Faster Attacks on Elliptic Curve Cryptosystems. In *Proc. Selected Areas in Cryptography (SAC)*, pages 190–200, 1998.