This is the author manuscript, before publisher editing. Use the identifiers below to access the published version.

Detecting Opportunistic Cheaters
in Volunteer Computing

Matthäus Wander, Torben Weis, Arno Wacker
University of Duisburg-Essen, Distributed Systems Group,
Bismarckstraße 90, 47057 Duisburg, Germany
{matthaeus.wander|torben.weis|arno.wacker}@uni-due.de

Abstract—For computationally expensive but parallelizable search problems distributed computing approaches based on volunteer computing can be used. Volunteering users spend their computation time to gain some sort of credit or for the sake of appearing in a ranking. Some of the users may try to gain reward without investing their computation time, i.e. they cheat. Hence, a cheat detection mechanism against such opportunistic cheaters is needed. The simplest approach is the recalculation of all results by multiple users followed by a voting. This simple approach is inefficient since it increases the computational complexity by the factor of the executed recalculations. In this paper we propose a new and efficient approach for cheat detection in search problems using a combination of sample testing and result aggregation. Our approach provides a high probability of detecting a cheating user while reducing the computational complexity using sample testing and the required bandwidth using result aggregation. In a limited range, one can compensate a small available bandwidth with more computations, thus providing a trade-off between bandwidth and computational complexity.

Index Terms—cheat detection; result verification; volunteer computing; peer-to-peer computing

I. INTRODUCTION

Computationally expensive search problems which can be parallelized are usually solved by some sort of distributed computing approach, e.g. [1] [2]. These approaches use the computational resources of volunteering users, hence this is also referred to as volunteer computing. The incentive for such volunteering users can be a credit system or simply the fact of appearing in a ranking. Volunteer computing is different from using a server cluster, since the resources provided by the participating users are not under the administrative control of a single entity. In such an untrusted environment some users may try to cheat, i.e. gain reward without investing their computation time. To discourage such behavior a mechanism for detecting such opportunistic cheaters is needed. Clearly, the most simple approach is letting multiple users calculate the same parts of the job and compare the results (*voting*). This approach increases the number of total calculations needed by the factor of redundant calculations.

In search problems it is required that we apply a so-called score function to each data item in the search space, compare the results, and select the ones with the best score values. We use the cryptographical brute-force key search as an example for the class of embarrassingly parallel algorithms for search problems. Hereby, the **key searcher** performs a brute-force attack on an encrypted text (ciphertext) generated by a cryptographic cipher. Each so-called job slice comprises the trial decryption with a different key and the calculation of a score for the resulting decryption. The score function determines the fitness of the decrypted data to be meaningful information. Several slices are combined to a block, whereas all blocks represent the entire job (i.e. search space). The different blocks are then calculated by different volunteering users.

The goal is to detect opportunistic cheaters efficiently in the key searcher application and similar distributed search problems. Therefore we propose a new approach for cheat detection based on a combination of sample testing and result aggregation. We simulated our approach and provide insights into the interdependencies of the computational complexity and the required bandwidth to transmit the aggregated result. Therefore, the contribution of this paper is: (1) an approach for efficient cheat detection utilizing a combination of sample testing and result aggregation and (2) the analysis of the interdependency of the used parameters gained through simulations.

This paper is organized as follows: In Section II we present our system model used throughout the paper. Then we present in Section III our approach for efficient cheat detection. We provide the simulation results and discuss the interdependencies of the computational complexity and the required bandwidth in Section IV. After that we discuss related work in Section V and conclude our paper with a summary and outlook in Section VI.

II. SYSTEM MODEL

In this section we describe how we expect to organize a distributed job and the inherent constraints. Then, we define the different types of cheaters and explain why we chose to concentrate on opportunistic ones.

A. Distributed Computing Model

We consider a system for distributed computing with an untrusted user base, e.g. volunteer or peer-to-peer computing. Participants can register an identity and gather credits or a position in a ranking. The administrator has the ability to remove misbehaving identities. A locked out user can create a new identity (*sybil attack*) but will lose all credits gathered with a previous identity. The participants utilizing their computing power for a distributed job are called *workers*.

The active workers retrieve an allocation for one part of a distributed job, compute it, and pass the result back to the job manager or to a designated result storage.

We define a distributed job $\mathcal{J}$ as a set of k blocks $\mathcal{B}_i$, with the block being the work package calculated by a worker:

$$\mathcal{J} := \bigcup_{i=1}^{k} \mathcal{B}_i \tag{1}$$

Similar, each block $\mathcal{B}_i$ is comprised of s job slices S, with the slices being the atomic work unit:

$$\mathcal{B}_i := \bigcup_{j=1}^{s} \{S_j\} \tag{2}$$

We need to apply a certain work function to each slice. Hence this work function transforms an input slice S_j into a score result R_j. We call this score function f and define it as follows:

$$f(S_j) := R_j \tag{3}$$

Clearly, the score function f is highly application dependent. The CPU time to compute f may vary but is typically in the range of milliseconds or seconds. This influences the number of slices s of a block. A distributed computing job requires management effort, e.g. to allocate blocks to workers and to keep track of the status. This effort does not pay off if the blocks are too small in size. Having large blocks may be disadvantageous too, e.g. if an unreliable worker goes offline without reporting the intermediate result of a large block. A reasonable computation time of a block is in the range of minutes or hours.

B. Cheating Model

We consider cheating as using a different algorithm or method to obtain a block result than the worker is expected to use. For example, the worker may use random junk data or intentionally made-up data as block result without actually performing any time-consuming calculations. Cheating usually leads to erroneous block results, however it does not necessarily do so. For example, if the score function f uses two values *hit* or *miss* to rate the slice result, then a cheater returning always *miss* would guess the correct result in almost all cases. Still, this would eventually lead to a wrong result, constituting an undesirable source of error and being hardly detectable. We therefore require f to exhaust a sufficiently large range of values, which makes guesses unlikely to succeed. For our further considerations, we also assume that each slice S_j of a block yields a different score value R_j.

We can outline two types of cheaters: opportunistic and malicious cheaters. An **opportunistic** cheater tries to save own computational resources when earning credits. In order to achieve this goal the cheater accepts a falsification of the block result. Opportunists have an interest in not getting caught in order to not lose their credits earned so far. Therefore, an opportunistic cheater might invest some work to arrange a seemingly plausible result, as long as it is substantially less work than computing the result honestly.

In contrast a **malicious** cheater aims to falsify the job result and is willing to invest own resources to reach his goal at best. Malicious cheaters have no interest in earning credits. While malicious cheaters may have there own reasons, there is a rational motivation for opportunistic cheaters (gather most credits with least computation time). We target specifically for the efficient detection of opportunistic cheaters.

III. DETECTION APPROACH

To efficiently detect wrong results we use sample testing, i.e. we verify/recalculate only a small set v of randomly selected job slices of a block. Sample testing requires that all results (i.e. the score value for each key) are available for comparison. However, transmitting the results of the entire block (in sum of the entire job) leads to inefficient usage of bandwidth. Since volunteering users are usually connected over the Internet, they have only a limited bandwidth available. Therefore we need to aggregate the results of a block. As result aggregation we propose to transmit only the t best results of a certain block, the so-called *toplist*. Clearly, transmitting only partial results (i.e. the toplist) limits the detection efficiency of the sample testing used. Hence, there is a interdependency between the computational complexity (v) and the required bandwidth (proportional to t), providing a certain degree of trade-off.

A. Sample Testing

For the cheat detection we want to verify as few slices as possible while still detecting cheated blocks with high probability. Instead of recomputing a block as a whole we use sample testing to verify a subset of the slices of the block. The slices to be verified are randomly selected and the result of each slice is checked for plausibility with the block result. This sample testing approach has the advantage of providing a favorable cost-benefit ratio, where the computation time spent for verification is the cost and the probability to detect an opportunistic cheater is the benefit.

The following variables influence the success of sample testing:

- s is the number of slices in each block
- $v \leq s$ is the number of slices a worker recomputes (i.e. verifies) when inspecting a block
- $r \leq s$ is the number of slices in a cheated block which reveal upon inspection that the block is not correctly computed

We define the random variable X as the event that the block $\mathcal{B}$ is detected by a verifying worker as being cheated by verifying v out of s slices. There are r slices which can reveal that the block is not computed correctly. We compute the combinatorial options to select v slices for verification without hitting one of the r slices that would reveal cheating. We divide this value by the number of options to choose v slices out of all s slices. This results in the probability of a cheated block to pass the verification. Consequently, the

probability for the event variable X, i.e. detecting a cheated block, is the complementary probability:

$$P(X=v) := 1 - \frac{\binom{s-r}{v}}{\binom{s}{v}} \qquad (4)$$

B. Result Aggregation

Besides the actual verification effort v, the cheat detection success depends upon the number of revealing slices r. These slices reveal a block as cheated if any of them have been selected for verification. Thus, r indicates the ability to detect a cheated block. The size of r depends on 1) the number of slices that the cheater did not compute correctly, which we call c, and 2) the definition of the block result.

If we concatenate all slice scores together and use this as block result, it simply applies $r = c$, as every cheated slice reveals the cheat attempt. However, the upload bandwidth of the worker limits the size of the block result and renders this approach impracticable for scenarios with many slices per block. The slice results needs to be aggregated in a way to fit the bandwidth limits of the workers while preserving the ability for efficient verification with sample testing. For this purpose we propose to use a toplist as aggregated block result which requires the worker to deliver the t best slices consisting of a slice identifier (e.g. index number) and the corresponding score.

For example, the job description says to find the 100 best slices of a block. There are two possibilities to reveal an incorrect toplist: 1) The cheater did not calculate the score for a stated slice in the toplist, but gave some arbitrary number. This can be detected by recalculating the said slice and comparing its score (which we call *positive verification*). 2) The cheater left some slices out which in fact score better than an entry of the reported toplist. This can be revealed by finding such a slice and comparing it against the reported toplist (*negative verification*). Random slices which score below the reported toplist cannot reveal a cheated block.

The simplest opportunistic cheat attack by returning junk values is detectable with almost absolute certainty by performing a positive verification on each entry of the reported toplist. A smart opportunistic cheater would calculate some randomly chosen slices honestly and use these to report the toplist, saving computation time by skipping all other slices. This type of attack is detectable only by negative verification. However, negative verification requires the toplist scores to be correct, otherwise a cheater could claim excessively high scores which would never be reached by random negative verification. Thus, our detection approach is first to perform positive verification on all toplist entries and then to perform negative verification on a certain amount of randomly selected slices.

IV. EVALUATION

Before evaluating our approach, we first demonstrate the efficiency of sample testing for cheat detection. For instance, we set the number of slices $s := 1000$ and the number of
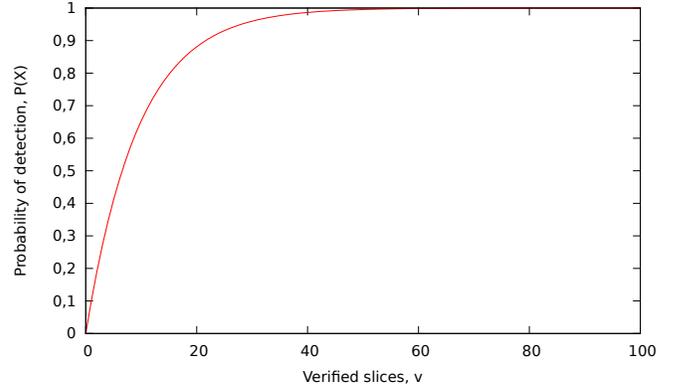

Fig. 1.   Detection of cheated blocks

revealing slices $r := 100$. The result of the analysis is shown in Figure 1. The y-axis shows the probability to detect the block as cheated subject to the number of verified slices v on the x-axis. A verification effort of $v < 100$, i.e. less than 10% of the slices of the block, is sufficient to achieve a probability of detection of almost 100%. When the size of the block s increases proportionally to the number of revealing slices r, the absolute number of necessary slices for verification stays roughly the same. This means for $s := 10\,000$ and $r := 1000$ we still gain a probability of detection of almost 100% with $v < 100$, which is less than 1% for this scenario.

A. Revealing Slices

As already discussed, the opportunistic cheater skips some slices and delivers a toplist that seems to be valid, i.e. that passes the positive verification. The number of revealing slices r is an essential parameter to calculate the probability to detect such a cheated block. The size of r depends upon the number of toplist entries t of the aggregated block result and the number of slices c that the cheater has skipped during computation.

We determined the number of revealing slices r of the key searcher by simulation. Given a block size of $s := 1000$ slices, we perform the smart cheating attack described above
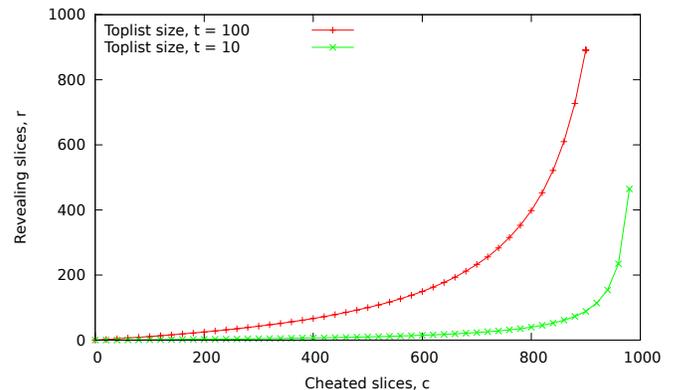

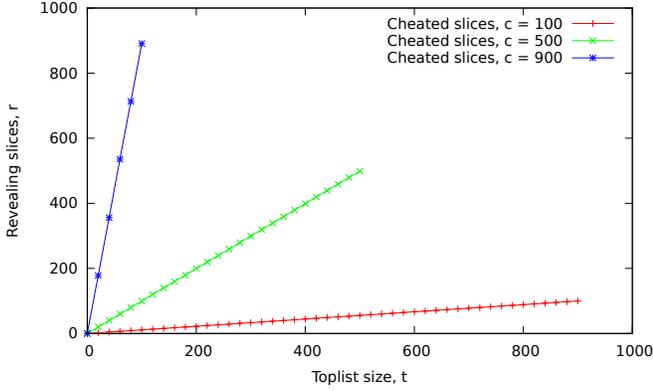Fig. 2.   Revealing slices with variable c

Fig. 3. Revealing slices with variable t

with a variable number of cheated slices c and then count the revealing slices r. Figure 2 shows the average result of 1000 simulation runs for toplists with 10 and 100 entries. The y-axis shows the number of slices which reveal the block as cheated, if any of them is chosen for negative verification. The x-axis shows the number of cheated slices c. The resulting number of revealing slices r is low for most numbers of cheated slices c. This can be compensated by increasing t. As we show in Figure 3, t has a linear impact on the size of r.

In Figure 2 and 3, r is defined only for $c \leq t$. If c is above t, the cheater does not calculate enough slices to fill a plausible toplist and thus will be caught by positive verification. If s, t and c increase proportionally, then r increases proportionally as well. For example, using $s := 1000$, $t := 100$, $c := 500$ leads to $r \approx 100$, while using $s := 10000$, $t := 1000$, $c := 5000$ leads to $r \approx 1000$. We discuss the implications of this effect later on.

B. Detection Probability

We now combine the calculation of revealing slices r with the sample testing approach to evaluate the actual probability to detect a cheater. We set the number of slices to $s := 1000$ and the number of cheated slices to $c := 500$ and simulated the probability to detect a block as cheated with a variable
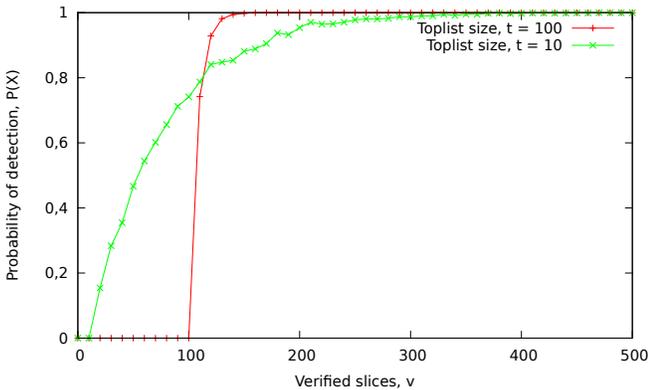

Fig. 4. Detection success of key searcher with variable v

number of verified slices v. As shown in Figure 4 we detect a cheated block almost certainly by verifying half of the slices of that block. As expected, the probability rises more sharply for $t := 100$ than for $t := 10$ because there are more revealing slices. However, the toplist size also dictates the minimum amount of slices to be verified. This is because before we can find any revealing slices by sample testing, our approach requires to positively verify the toplist. Therefore $P(X)$ is zero for $v < t$.
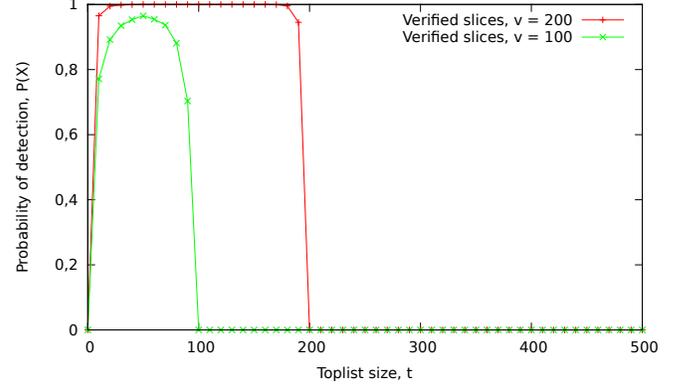

Fig. 5. Detection success of key searcher with variable t

We performed another simulation to demonstrate the impact of t on the detection probability. In Figure 5 we show the result for fixed verification efforts $v := 100$ and $v := 200$. The x-axis shows the toplist size t and the y-axis shows the detection probability. As one can see, increasing t compensates for a low v and improves $P(X)$, but only to a certain degree. Increasing t implicits to dedicate more verification effort to the positive verification. This in turn reduces the available verification effort for the random sample testing and thus may reduce $P(X)$.

C. Impact of Parameters

The probability of detection depends upon the parameters c, s, t and v. The number of cheated slices c arises from the cheater's behavior and cannot be affected by us in any way. The number of slices s of a block usually depends on the application and the distributed computing scenario, as explained in Section II-A. Therefore, s is constrained by practicability considerations and cannot be adjusted freely.

To control the cheat detection, we can adjust t or v. We demonstrate the interdependency of variable t and v and their impact on the probability of detection in Figure 6. For low values of t and v, $P(X)$ represents a slope. When increasing t and v, they reach their saturation at $P(X) = 1$. Increasing v at this point does not have any further effect, while, as explained above, increasing t can negatively impact $P(X)$. Therefore, choosing $t < \frac{v}{2}$ is a reasonable estimation.

The result shown in Figure 6 is an example scenario with $c := 500$ and $s := 1000$. The detection probability gets better for larger values of c or s, so this example poses a lower limit for any combinations of $c \geq 500$ and $s \geq 1000$. c affects
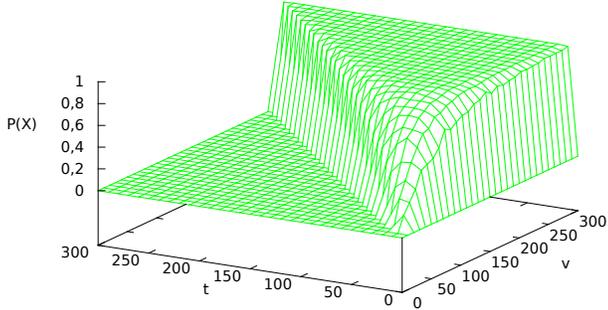
Fig. 6. Probability of detection with variable v, t

the number of revealing slices, therefore changing c affects the starting point and the size of the slope. Choosing a larger c leads to better $P(X)$ values. As explained above, s does not affect the ratio of revealing slices of a block. However, increasing s has a positive superlinear effect on the detection probability $P(X)$ of a single block. This is because sample testing yields better results when the input parameters increase proportionally. For the key searcher application the block sizes will in practice be larger than $s := 1000$. Therefore, the example above shows conservative results for this particular type of application.

As another example, we calculated $P(X)$ with $s := 10^7$, $c := 5 \cdot 10^6$ (50% of s), $t := 1000$ (0.01% of s). For $v := 5 \cdot 10^4$ (0.5% of s) we achieved a detection probability of $P(X) = 0.9945$ and for $v := 10^5$ (1% of s) we achieved $P(X) = 1$. Assuming that one toplist entry, consisting of an identifier and a score value, is 16 bytes long, then the whole block result requires 16 KB to be transmitted by the worker. A typical home user can transmit data of this size within less than a second to an Internet server. Thus, we can detect a block cheated by an opportunistic cheater with high probability with a feasible toplist size and a verification effort of 1% (and above) per block.

V. RELATED WORK

Sarmenta presented the idea of *spot-checking* for volunteer computing [3]. The job manager chooses random blocks which are recomputed in order to find cheaters. The results of redundant recomputations are used to estimate *credibility* values for workers and to achieve an error rate below a certain threshold. The model to compute the credibility depends on the ability to *blacklist* cheaters (exclude them from a job) and whether the load for redundant recomputation is distributed among workers or computed by the job manager herself. Similar to spot-checking Golle and Stubblebine discussed *probabilistic voting* on the results of blocks and proposed to use trust ratings to control the voting process [4]. Germain-Renaud suggests a different model to dynamically gather credibility values [5]. In these papers the authors examined measures on job level to deal with cheaters and to dynamically adapt the

verification amount by using trust. However, when a block has been chosen for verification, redundant recomputation is used. The difference to our approach is that the detection success of spot-checking whole blocks behaves linearly with the verification effort, while sliced verification can boost the detection probability.

Golle and Mironov proposed the *ringer scheme* [6]. Given a one-way function h, the job manager computes several $h(x_i) = y_i$, sends the set of y_i to the participant along with the real challenge y' and requires her to find the ringers x_i. The scheme is limited to applications for which the job manager can efficiently compute $h(x)$. Szajda et al. extended the ringer scheme to fit to optimization problems and Monte Carlo simulations [7]. They further explained that computing $h(x)$ can be too expensive for the job manager in practice, including the key searcher application that we have been looking into.

The *quiz scheme* by Zhao et al. is a general approach based on the same idea as the ringer scheme. The job manager assigns an easy verifiable quiz block together with a batch of real blocks to a participant [8]. The job manager accepts the batch results only if the quiz has been computed correctly. However there is no generic mechanism to efficiently create quizzes that are indistinguishable from real blocks.

Du and Goodrich examined how to deal with *hoarding cheaters* for search problems [9]. A hoarding cheater computes everything of a block but keeps the best results to itself, which is very similar to our definition of a malicious cheater. They proposed *chaff injection*, a kind of a bait, to reveal hoarding cheaters. Besides they discussed how to hide the actual solution from the workers for certain application types.

Du et al. presented the *commit-based sampling scheme* (CBS) to verify a block by recomputing randomly selected sample slices [10]. CBS is focused on commitment of slice results without having to transfer all of them to the job manager. While this can improve the verification detectability of our key searcher application virtually to $\frac{r}{s} = 1$, it comes along with a high commitment cost. In addition to the computation of s slices of a block the worker has to calculate $s-1$ cryptographic hash values which is not feasible for a typical application like the key searcher (calculating an MD5 or SHA-1 hash is somewhat faster than e.g. an AES decryption, but not in significant orders of magnitude [11]).

Monrose et al. proposed to detect cheaters by comparing checkpoints of distributed computations [12]. The job manager repeats the computation between randomly selected checkpoints to see whether the transition between two checkpoints has been computed correctly. This generic approach is not restricted to embarrassingly parallel applications and allows for cheat detection without manually adapting the application. However, the logging of the checkpoints and the recomputation between checkpoints is more time-consuming than application dependent slicing of embarrassingly parallel applications. The job manager poses a bottleneck for checkpoint-based verification, so Domingues et al. proposed to combine checkpointing with redundant recomputation [13]. While

this unburdens the job manager and allows for fine-grained detection levels, it binds half of the worker resources for redundancy. If the checkpointing approach could be improved in terms of performance, it might prove as suitable to enable our slicing approach to sequential applications.

VI. Summary and Outlook

In this paper we presented an efficient approach for detecting opportunistic cheaters in distributed search problems based on volunteer computing. The main idea of our approach is to combine the efficiency of sample testing with a result aggregation function for reducing the required bandwidth. We provided simulations showing the interdependencies of the computational complexity and the required bandwidth. Our results show that a small available bandwidth can be compensated with more calculations. However, the inverse is not true: litte verification effort cannot be compensated in all cases with more available bandwidth. Hence, in a limited range a trade-off between computational complexity and used bandwidth can be utilized. Our results also showed that we can achieve very high detection probabilities ($P(X) \approx 1$) with small toplists ($t \approx 1\%$) and computational effort ($v \approx 1\%$), thus fortifying our goal for an efficient approach.

As future work, we would like to integrate our approach together with an adaptive credibility-based cheat detection control. To detect both, opportunistic and malicious cheaters we would use two different approaches: our toplist approach to detect opportunistic cheaters and either a bait approach or probabilistic voting to detect malicious cheaters.

References

[1] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 4–10.

[2] M. Wander, A. Wacker, and T. Weis, "Towards Peer-to-Peer-based cryptanalysis," in *6th IEEE LCN Workshop on Security in Communication Networks (SICK 2010)*, Denver, CO, USA, Oct. 2010, in press.

[3] L. F. G. Sarmenta, "Sabotage-tolerance mechanisms for volunteer computing systems," in *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, 2001, p. 337.

[4] P. Golle and S. G. Stubblebine, "Secure distributed computing in a commercial environment," in *FC '01: Proceedings of the 5th International Conference on Financial Cryptography*. London, UK: Springer-Verlag, 2002, pp. 289–304.

[5] C. Germain-Renaud and N. Playez, "Result checking in global computing systems," in *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*. New York, NY, USA: ACM, 2003, pp. 226–233.

[6] P. Golle and I. Mironov, "Uncheatable distributed computations," in *CT-RSA 2001: Proceedings of the 2001 Conference on Topics in Cryptology*. London, UK: Springer-Verlag, 2001, pp. 425–440.

[7] D. Szajda, B. Lawson, and J. Owen, "Hardening functions for large scale distributed computations," in *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2003, p. 216.

[8] S. Zhao, V. Lo, and C. GauthierDickey, "Result verification and trust-based scheduling in peer-to-peer grids," *Peer-to-Peer Computing, IEEE International Conference on*, vol. 0, pp. 31–38, 2005.

[9] W. Du and M. T. Goodrich, "Searching for high-value rare events with uncheatable grid computing," in *ACNS*, 2005, pp. 122–137.

[10] W. Du, J. Jia, M. Mangal, and M. Murugesan, "Uncheatable grid computing," in *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 4–11.

[11] W. Dai, "Crypto++ 5.6.0 benchmarks," http://www.cryptopp.com/benchmarks.html, March 2009, accessed July 2010.

[12] F. Monrose, P. Wycko, and A. D. Rubin, "Distributed execution with remote audit," in *In Proceedings of the 1999 ISOC Network and Distributed System Security Symposium*, 1999, pp. 103–113.

[13] P. Domingues, B. Sousa, and L. Moura Silva, "Sabotage-tolerance and trust management in desktop grid computing," *Future Gener. Comput. Syst.*, vol. 23, no. 7, pp. 904–912, 2007.