Adaptive Cheat Detection in Decentralized Volunteer Computing with Untrusted Nodes

Nils Kopal, Matthäus Wander, Christopher Konze, Henner Heck

Adaptive Cheat Detection in Decentralized Volunteer Computing with Untrusted Nodes

Nils Kopal[1], Matthäus Wander[2], Christopher Konze[1], and Henner Heck[1]

[1] University of Kassel, Applied Information Security, Kassel, Germany
[2] University of Duisburg-Essen, Distributed Systems Group, Duisburg, Germany

Abstract. In volunteer computing, participants donate computational resources in exchange for credit points. Cheat detection is necessary to prevent dishonest participants from receiving credit points, without actually providing these resources. We suggest a novel, scalable approach for cheat detection in decentralized volunteer computing systems using gossip communication. Each honest participant adapts its detection effort dynamically subject to the number of active participants, which we estimate based on observed system performance. This enables minimizing the detection overhead for each participant, while still achieving a high preselected detection rate for the overall system. Systems based on majority voting usually produce at least 100% overhead, whereas our approach, e.g. requires only 50.6% overhead in a network with 1 000 participants to achieve a 99.9% detection rate. Since our approach does not require trusted entities or an active cooperation between participants, it is robust even against colluding cheaters.

1 Introduction

Cheating is a well-known problem in distributed systems that rely on the computers of volunteers, i.e. volunteer computing systems [2]. In such a system, the computers of volunteers are interconnected and build a large-scale distributed system for distributed computations. We call a complete distributed computation a job and a partial computation, typically performed by a single node, a subjob. Researchers often use volunteer computing in cases where there is no funding for sufficient computational resources within their research projects. Many volunteer computing projects have a charitable background, i.e. the search for cancer medicaments, AIDS research, water research, etc. We distinguish two different classes of volunteer computing: The first class is the classic volunteer computing, which is based on a client-server approach, e.g. the Berkeley Open Infrastructure for Network Computing [1] (BOINC). Here, a server manages the distribution of all subjobs. The combination of the corresponding subjob results leads to the result of the overall job. Participating nodes in the volunteer computing network request subjobs from the central server and deliver results to the central server. Clearly, if the server fails, the complete job computation stops since no new

subjobs are assigned to requesting nodes. Additionally, a server requires maintenance which results in costs that have to be taken into account by researchers. The second class is decentralized volunteer computing, e.g. [9, 14]. Here, no central server exists. The participating nodes have to self-organize distribution and assignment of subjobs and distribution and storage of results.

To offer incentives to participate in volunteer computing projects, i.e. motivate people to donate their computational resources, volunteer computing systems usually maintain lists of their participating users with respect to their computational work spent [1]. People who donate resources typically aim to improve their position on such lists, causing them to provide more resources.

Besides well-behaving users, there also exist cheaters in volunteer computing systems. [3] Such a cheater delivers false or only partial correct results to gain more credit points than they deserve. Correct results are essential for researchers, which requires a verification of results coming from untrusted and potentially unreliable volunteers. Otherwise, partly false results could impair the overall job result up to the point of invalidating the joint effort of hundreds or thousands of volunteers. There exist well-known anti-cheating techniques for client-server based volunteer computing [6]. These include redundant computation, majority voting, and sample testing, which are performed or initiated by the central and trusted server. In a decentralized network, there is no central authority that manages the distribution of subjobs. Thus, techniques like redundant computation and majority voting cannot be easily used.

In an untrusted, decentralized network, each node needs to devote a portion of their resources to detect and correct false results. As this reduces the overall speedup of computation, only a subset of results can be selected to be verified for efficiency reasons. Sample testing suffices to detect cheaters in search problem applications with close to 100% probability [15].

In this paper, we present a method for cheat detection in untrusted, decentralized volunteer computing networks based on sample testing. The method works on top of a gossip-based communication. It is immune to colluding cheaters, because each node performs its cheat detection independent of each other. We show that to meet a given detection rate, the overall cheat detection effort per network can be kept constant. With an increasing size of the network, this allows us to reduce the cheat detection effort per node without negatively impairing the detection rate. We thus propose to adapt the cheat detection per node dynamically subject to the size of the network, which can be estimated by each node individually by evaluating the current network workload. In sum, the **contribution** is a cheat detection method that dynamically adapts the verification effort to the total number of participating nodes in untrusted volunteer computing networks without the need of any additional messages.

The rest of the paper is organized as follows: First, in Section 2 we present our system model for decentralized volunteer computing. Then, in Section 3 we present our application scenario. Here, we base our model on a distributed cryptanalysis scenario. Additionally, we present our cheater classes and our cheat detection algorithm. We also present our definition of effort for computations and

cheat detection. After that, in Section 4 we present our idea for an adaptive cheat detection method. In our evaluation in Section 5 we simulate the effort for cheat detection of a single computing node with fixed detection rates and effort. After that, we evaluate, on the basis of single nodes, the effort a complete network has to perform for cheat detection with fixed detection rates and effort. After that, we present an evaluation of our adaptive method. Then, in Section 6 we briefly present the related work in the field of cheat detection and prevention. We conclude our paper in Section 7 with a brief outlook on future work.

2 System Model

In a decentralized volunteer computing system, the knowledge of the participants is distributed using messages. Each participant can send messages directly to a certain number of other participants, which are called neighbors. Nodes disseminate knowledge by sending messages to their neighbors, which copy and forward them to their neighbors. Since such an approach leads to a high number of message transmissions, the amount of data that needs to be exchanged during a job computation has to be kept small. The basis for our gossip-based distribution is a network consisting of nodes. Each node is connected to randomly chosen neighbors. Nodes transmit the results of their subjob computations and their state of the overall job to their direct neighbors. Our distribution algorithms compute an embarrassingly parallel search problem (*search job*) with a result using a problem specific computation function. Such a search job can be parallelized by dividing it into independent subjobs, where each subjob can be computed by the same computation function. To get the overall result of a job, the results from all subjobs can be combined using a combination function. Furthermore, we assume that the combination function is associative, commutative, and idempotent. We assume that the size of the combination of two results equals the size of a single result. Since the jobs in our example scenario consist of several thousands, millions, or more subjobs, it is not possible to disseminate the state of each subjob. We use distribution algorithms [8–10], which divide the total computation space into subspaces. These subspaces are chosen small enough so that it can be disseminated in total. Nodes work in parallel on the same subspace. A node randomly selects a subjob, computes it, and sends the results to their neighbors. These merge all received results and forward them to their neighbors. Once the nodes finished a subspace, they move on to the next subspace until the job is complete.

3 Application Scenario

Our application scenario is the keysearching of a modern symmetric cipher, i.e. distributed brute-forcing an AES128 [4] encrypted text and searching for the decryption key. In our application scenario, we divide the complete search space ($\approx 2^{54}$ for passwords consisting of lowercase and uppercase characters, digits, and special characters) in subjobs, each consisting of $2^{20} = 1\,048\,576$ keys. To search

through a single AES128 subjob, a node decrypts the given ciphertext using every key within the range of that subjob. The goal of the cryptanalysis is to find the correct decryption key. To rate the keys, a node uses the Shannon entropy [13] function H as a cost function. With natural language, i.e. the original plaintext, the entropy value is mostly at its minimum with respect to all decryption keys. After performing all decryptions, a node generates a toplist of the k "best" keys of a subjob. Those keys are the ones that decrypt the given ciphertext to the plaintexts with the lowest entropy. Each node does this for different subjobs. After finishing a subjob, the nodes send their results to their neighbors. They combine the toplist of each received subjob result to create a global toplist over all subjobs.

In our scenario, a cheater would not test all keys of an AES128 subjob. The cheater has the motivation to earn credit points to achieve a high rank in a volunteer computing network without doing all of the required work, while avoiding the detection by the system's countermeasures [1]. Thus, a cheater, in general, may compute only parts of a given subjob and only delivers partially (correct) results. The more of a subjob a cheater computes, the harder it is for a cheat detection algorithm to detect this cheater. This is based on the fact that the more results of a subjob are computed correctly, the less results are missing that could be detected. We refer to a cheater that only computes parts of a subjob as an *opportunistic cheater*. Opposed to that, a *disturber* submits garbage results just for the sake of vandalism. Disturbers can be easily detected with the same means as opportunists. *Colluding cheaters* coordinate their efforts to persist cheated subjob results in the network. In some systems, colluding cheaters have an advantage over solitary opportunistic cheaters, but this does not apply to our approach.

BOINC-based solutions use multiple computations of subjobs by different nodes and make a majority decision on the correct result. In [15], we developed an approach for the detection of cheated results within a distributed computing scenario introduced by an opportunistic cheater. The detection is based on two different approaches: *Positive verification* and *negative verification*. With positive verification, we verify the correctness of a subjob computation and with negative verification, we try to find other (better) results, which the delivering node omitted. If either positive verification fails or negative verification succeeds, we found a cheated result and we decline it. For positive verification, we assume that we can check the results in very short times. Clearly, this is true for an AES128 subjob since the decryption using all of the keys of the subjob toplist can be done very quickly.

To compare our decentralized cheat detection mechanisms and methods with the state-of-the-art solutions (client-server-based, i.e. BOINC) we need to estimate the effort that is needed for the computations and additionally the cheat detection mechanisms. First, we define the effort for a single subjob computation as $\mathcal{E}_{subjob} = 1$. Furthermore, the effort to compute i subjobs is $\mathcal{E}_{node}(i) = i \cdot \mathcal{E}_{subjob} = i$, which is i times the amount of effort needed for a single subjob.

This is possible, since every subjob is, with respect to the needed computations, identical.

We define $\mathcal{E}_{Detect}(P_{Detect})$ as the effort needed for the computation of a detection algorithm with the detection rate P_{Detect}. For example, an effort equal to 0.5 means, that half of the subjob has to be recomputed to perform the cheat detection. Clearly, the effort function depends on the problem that is being computed by our volunteer computing network. Thus, the computation of $\mathcal{E}_{Detect}$ can not be generalized. In Section 5, we present an evaluation for the effort that is needed for the detection of cheated subjob results in a cryptanalysis scenario, i.e. keysearching for the key of a symmetric encryption algorithm.

4 Approach

In our approach, every node performs independently a partial cheat detection on every subjob result disseminated in the volunteer computing network. The approach needs no additional messages for executing the cheat detection. It is completely based on the assumption that every honest node provides its small part of cheat detection effort. This cheat detection effort is subject to a given fixed destination detection rate $P_{DetectNetwork}$ and the current workload of the network, to which our approach dynamically adapts to. We measure the workload in units of virtual nodes, which is a standard amount of processing power provided by a node. However, our approach is able to cope with heterogenous nodes providing more or less workload than a virtual node. Each node performs the following steps in our cheat detection approach for each newly received subjob result:

1. Determine the number of virtual nodes $n_{Virtual}$ currently connected to the network with the method introduced below.
2. Compute the amount of virtual nodes r that the node represents itself.
3. Based on the amount $n_{Virtual}$ compute target detection rate P_{Detect} of node.
4. Perform cheat detection with the computed node detection rate P_{Detect} on each newly received subjob result with effort $\mathcal{E}_{Detect}$.

4.1 Determine Workload of Network

We now present a method to infer the workload of a node and of the whole volunteer computing network, which is necessary to determine the effort for cheat detection required. Each node administrates a list $L_{Timeframe}$ of finished subjob results within a sliding time window $Timeframe$. The list $L_{Timeframe}$ contains the received amount of subjob results of the node, the node's neighbors and all of their neighbors, which follows from the gossip-based dissemination of job results. The node divides the amount of subjob results $\#(L_{Timeframe})$ by the timeframe time $Timeframe$ to obtain the current *workload* $W_{Network}$ of the network, i.e.

$$W_{Network} = \frac{\#(L_{Timeframe})}{Timeframe}. \tag{1}$$

To estimate the amount of nodes $n_{Virtual}$, we divide the workload $W_{Network}$ of the network by a constant virtual node workload $W_{VirtualNode}$, i.e.

$$n_{Virtual} = \frac{W_{Network}}{W_{VirtualNode}}. \tag{2}$$

Although we cannot compute the actual number of nodes or their workload, this virtual number of nodes suffices for our purposes. We assume that cheaters not only skip on result computation but also omit participation in the cheat detection process, since they have no benefit in doing so. We thus have to compensate for this amount by reducing the estimated workload by the amount of cheaters $C_{CheaterRate}$ that the system should be resistant against. Setting e.g. $C_{CheaterRate} = 0.05$, our approach achieves the target detection rate $P_{DetectNetwork}$ in a network with at most 5% of all claimed results to be cheated. If the actual amount of cheaters exceeds $C_{CheaterRate}$, the cheat detection still works but achieves a detection rate less than the anticipated $P_{DetectNetwork}$. The compensated amount of virtual nodes is thus

$$n_{Virtual} = \frac{W_{Network}}{W_{VirtualNode}} \cdot (1 - C_{CheaterRate}). \tag{3}$$

Each node now determines the number of virtual nodes r it represents with its own workload. It does so by dividing its current workload W_{Node} by $W_{VirtualNode}$, i.e.

$$r = \frac{W_{Node}}{W_{VirtualNode}}. \tag{4}$$

4.2 Perform Cheat Detection

Now that the node knows its computing power relative to the network's computing power, it computes the amount of effort it has to contribute so that the network reaches its target detection rate. Thus, it first computes the detection rate which a single virtual node has to add, i.e.

$$P_{DetectVirtualNode} = 1 - \sqrt[n_{Virtual}]{1 - P_{DetectNetwork}}. \tag{5}$$

After that, the node computes its node target detection rate based on that with the equation

$$P_{Detect} = 1 - (1 - P_{DetectVirtualNode})^r. \tag{6}$$

We combine the two equations to the following final single equation for the target node detection rate

$$P_{Detect} = 1 - (1 - P_{DetectNetwork})^{\frac{r}{n_{Virtual}}}. \tag{7}$$

Based on this computed local detection rate P_{Detect}, a node computes the verification effort $\mathcal{E}(P_{Detect})$ that it has to process, i.e. the amount that needs to be recomputed of each newly seen subjob result. This effort is also based on the type of cheater $T_{Cheater}$, the network has to be resistant again. A $T_{Cheater} = 0.5$

means, that the cheater only computes 50% of given subjobs. This target type of cheater is set by the user of our adaptive method. In our evaluation we empirically determine the function $\mathcal{E}(P_{Detect}, T_{Cheacter})$ for the 50% cheater. In other scenarios, this effort has to be either computed, if possible, or empirically determined by the user.

5 Evaluation

In this section, we first present our idea of two distinguished cheat detection classes: *Static* and *adaptive*. After that, we present the simulation of the effort of a single node in the static class. Then, we use the results of that simulation to evaluate the effort of a complete network based on the static class. After that, we present our method to let each node adapt its cheat detection rate and the needed mode effort with respect to the amount of nodes in the network. Doing so, we show how we keep the detection rate as well as the node effort constant.

5.1 Detection Classes

We differentiate cheat detection in two different classes: The *static class*, and the *adaptive class*. In the static class, a network or the nodes of the network perform cheat detection with a static, i.e. fixed, detection rate P_{Detect}. Thus, a node performing cheat detection on a received result has the probability of P_{Detect} to detect, if the result is not correct, i.e. cheated. In the next section we show, that this static detection class does not scale in a decentralized network since the increasing effort $\mathcal{E}_{Network}$ for detection reduces the speedup to an upper limit. With the *adaptive class* the nodes of a network adapt their detection rates P_{Detect} and effort $\mathcal{E}_{Detect}$ according to the amount of nodes within the network. We furthermore use the simulation result of a single node to estimate the correlation between effort and detection rate in the adaptive class.

5.2 Cheat Detection in the Static Class for a Single Node

We base our simulation on the AES128 scenario presented in the last sections. In our simulation, we let the simulator search for the 10 "best" AES128 keys in a cryptanalysis job. Thus, a simulated cheater that only searched through 50% of the keys of a subjob would only find 5 of these keys on average. Then, to simulate the detection of the cheater, a real node would first positively verify the best list, i.e. check the entropy-values of each entry. Clearly, all the entropy values within our simulation best list are correct, thus, we omitted this step in the simulation. After that, a node would randomly try to find "better" values, i.e. keys with lower entropy values. For that, we used different amounts of detection effort ranging from 0.0001% to 100%. We simulated the cheat detection performed by a single node with different cheaters with respect to the cheated amounts of computations. A cheater omits between 10% and 90% of all keys of

a subjob computation. It selects the keys, for which it actually does computations randomly. The cheat detection node randomly selects a dedicated amount of AES keys out of the subjob space to find lower entropy values, i.e. doing negative verification. In Figure 1 we show the results of our simulations. For each point in the graph, we did 10 000 simulations and calculated the average value over all simulation runs. The graph shows different amounts of cheated values starting from 90% (black line) going down to 10% (purple line). A cheater with 90% means that the cheater omitted 90% of the computations. In our graph, it can be seen that with higher amounts of negative verifications, i.e. the node effort (abscissa), the detection probability, i.e. the detection rate (ordinate), also increases. With an effort value $\mathcal{E}_{Detect} > 7\%$ our node would detect nearly every cheated subjob. Clearly, 7% of effort, i.e. recomputation of 7% of each subjob, is way too high for a real-world usage. But since not only one node performs cheat detection, but also all n nodes do, we can decrease the effort and detection probability at every node as shown in the next sections.

5.3 Cheat Detection Effort in the Static Class of a Complete Network

If only exactly one node would perform a single detection run on each subjob, in the previous section we evaluated that we need a recomputation of nearly 7% (random selections) out of every subjob in the best case to perform cheat detection as seen as black line (90%) in Figure 1. Here, the maximum value is reached at nearly 7%. For determining the real effort of a decentralized network,

Table 1. Different Scenarios - Detection Probability and Effort of a single Node

Scenario	Detection Probability P_{Detect}	Effort $\mathcal{E}_{Node}$
A (high)	1.22%	$\approx 0.1271895\%$
B (medium)	0.46%	$\approx 0.0490371\%$
C (low)	0.06%	$\approx 0.0129130\%$

we evaluated three different scenarios (A, B, and C) with different static detection probabilities P_{Detect} (from high to low) and corresponding node efforts $\mathcal{E}_{Node}$. We show the different detection probabilities and effort rates in Table 1. We extracted these values out of our simulations, as shown in Section 5.2. The basis for our computations is a volunteer computing job that consists of $j = 2^{32}$ different subjobs. First, we computed the effort $\mathcal{E}_{Client-Server}$ that a client-server solution, for example BOINC, would need for the cheat detection:

$$\mathcal{E}_{Client-Server} = 2 \cdot \mathcal{E}_{Node}(j) \tag{8}$$

$$= 2 \cdot \mathcal{E}_{Node}(2^{32}) \tag{9}$$

$$= 2^{33} \tag{10}$$

This amount of computations has to be performed in total by all the client-nodes in a client-server based volunteer computing network. Clearly, the value is independent from the amount of nodes, since the distribution of subjobs to the nodes is done by the server. We then computed the amounts for our three different scenarios with variable numbers of nodes. We show the result of these computations in the Figure 2. For comparison with the client-server case, we computed the *quotient* Q of the client-server case effort (*numerator*) and the decentralized cases (*denominator*) effort. A quotient of 1 means, that the network's effort is the same as the client-server's effort. For example, the effort

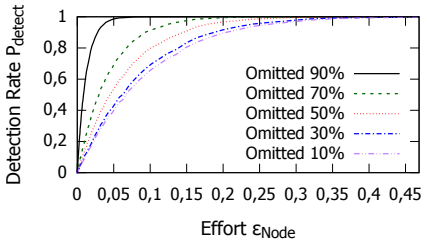

Fig. 1. Node Effort Simulation

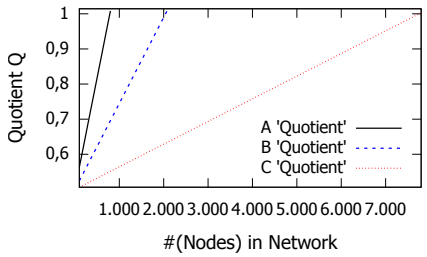

Fig. 2. Effort Quotient Q

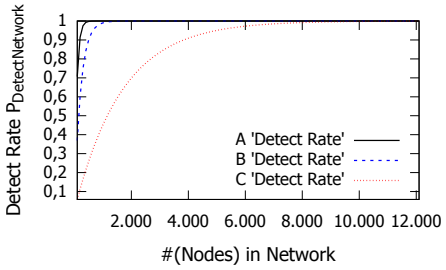

Fig. 3. Detect Rates $P_{DetectNetwork}$

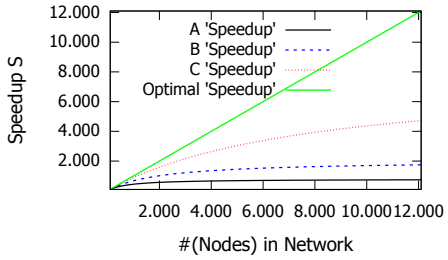

Fig. 4. Speedup S

$\mathcal{E}_{Decentralized}$ for scenario A with $P_{Detect} = 1.22\%$ and an assumed amount of nodes count $n = 700$ is:

$$\begin{aligned} \mathcal{E}_{Decentralized} &= \mathcal{E}_{Node}(j) + (n \cdot \mathcal{E}_{Detect}(j)) && (11)\\ &= 2^{32} + (700 \cdot 0.001271895 \cdot 2^{32}) && (12)\\ &= 8\,118\,891\,612.85 && (13) \end{aligned}$$

We now calculate the quotient Q of the client-server case $\mathcal{E}_{Client-Server}$ and the scenario A case:

$$Q = \frac{\mathcal{E}_{Decentralized}}{\mathcal{E}_{Client-Server}} = 0.944691034 \approx 94,45\% \quad (14)$$

Thus, a decentralized network (with parameters as in case A) needs $94,45\%$ of the computations that a client-server network needs. Then, we computed the corresponding detection rate $P_{DetectNetwork}$ of such a network. To compute that detection rate, we used the detection rate of a single node P_{Detect}:

$$P_{DetectNetwork} = 1 - (1 - P_{Detect})^n \tag{15}$$
$$= 1 - (1 - 0,0122)^{700} \tag{16}$$
$$= 0.999814512 \approx 99.98\% \tag{17}$$

With scenario A the detection rate is nearly 100% - only one out of 1 000 cheated subjobs would remain undetected on average in a network. Clearly, in a real volunteer computing scenario, we assume that there would never be a thousand cheated results disseminated in the network. By increasing the effort of a single node the detection probability of the network can also be increased. We also show the different detection rates of our scenarios in Figure 3.

Finally, we computed the speedup S of our scenarios. The speedup of a distributed system is the amount of parallel computed subjobs. If a network consists of n nodes the speedup is optimal if n different subjobs are processed in parallel. We computed the speedup S with the following equation

$$S = \frac{\mathcal{E}_{Node}(j)}{\mathcal{E}_{Decentralized}} \cdot n \tag{18}$$

where j is the total amount of subjobs, $\mathcal{E}_{Decentralized}$ is the total effort of the decentralized network, and n is the amount of nodes in the network.

We depicted the speedup graphs of our scenarios with different amounts of nodes in Figure 4. As a result of our evaluation it can be seen that with increasing the amount of nodes but keeping a constant effort $\mathcal{E}_{Node}$ for cheat detection at every node, the speedup is restricted to an upper bound. For scenario A this upper bound is ≈ 340, for B this upper bound is $\approx 1\,750$, and for C this upper bound is $\approx 4\,750$. Additionally, we added the optimal speedup (green line) to the graph. Here, the speedup S is equal to the amount of nodes n. Speedup values higher than these bounds cannot be reached with constant $\mathcal{E}_{Detect}$ values.

5.4 Our Adaptive Method

In this section, we present the evaluation which we performed with a simulator that implements the static and the adaptive cheat detection. In this evaluation we combine the adaptive method presented in section 4 with the estimation of node amount of section 4.1 needed for the adaptive method to create a prototype. The simulation time is represented in 'ticks'. A simulated subjob is 'computed' by a node waiting a defined amount of ticks, i.e. 'subjob duration'.

The simulated network is defined by the amount of nodes and their neighbors, the computational power of the nodes, cheater rates, cheat detection rates, cheat detection effort, etc. For cheaters and their corresponding detection effort $\mathcal{E}_{Detect}$, we used the $T_{Cheater} = 0.5$ cheater, i.e. a 50%-cheaters, as shown in Figure 1. We

extracted the detection rate and effort values and created a mapping function for our simulator.

With our simulations we show that our adaptive method outperforms the static cheat detection with respect to the effort needed by the nodes for performing the cheat detection. Furthermore, our simulations show that the static class does not scale with increasing amount of peers. Additionally, we show that the adaptive method needs less effort for cheat detection than a BOINC-based system needs.

We simulated different networks with sizes between 100 and 1 000 nodes, each node having 5 neighbors. Our simulator performed a simulation of a distributed job comprising of 320 000 subjobs. For the static cheat detection class, we set the detection rate of a single node $P_{DetectNode}$ to 5%, which results in an estimated $P_{DetectNetwork}$ of 99.4% for 100 nodes. We set the amount of cheaters in each network to 5% who cheat with 5% of their subjobs. Thus, 0.3% $\approx$ 800 of all subjobs disseminated within the simulation network were cheated on average. Furthermore, we set the virtual node workload for our algorithm to 0.2, thus, a virtual node finishes 0.2 subjobs in each simulation iteration. The real simulated nodes finished a subjob in 5 simulation ticks. We set the window of our algorithm to 10 ticks.

We present the results of our effort simulations in Figure 6. With the static approach, the effort increases proportionally to the number of nodes (red, dashed line). This is caused by the fact that each node performs cheat detection on every subjob result distributed in the network. The adaptive algorithm (blue, solid line) adapts dynamically to the amount of nodes in the network, keeping the effort at a rate around 162 000. This is about 50.6% of the overall amount of computed subjobs. A BOINC-based system (black line) would compute each subjob at least twice to enable majority voting, resulting in a minimum of 100% additional effort for the cheat detection. Directly compared to BOINC the quotient Q is $Q = \frac{\mathcal{E}_{OurMethod}}{\mathcal{E}_{Client-Server}} \frac{162\,000+320\,000}{320{,}000+320{,}000} = 0.753125$. I.e. our system needs $\approx$ 75% effort compared to a client-server system with majority voting needs, i.e BOINC.

In Figure 7 we depicted the cheat detection rates of the static and the adaptive methods. Additionally, we computed the detection rate of BOINC with colluding cheaters. BOINC achieves 99.7%, because there is a chance that BOINC gives the same sub job to two colliding cheaters, which results in an overlooked cheated sub job result despite redundant computation. The static method (red, dashed line) keeps a detection rate of 100%, but as already shown does not scale with respect to the effort. The adaptive method (blue, solid line) reaches a detection rate between 99.8% and 100%. The target detection rate was 99.9%, which is reached on average. Collusion among cheaters does not affect the detection rate, because each subjob result will be checked for correctness by each honest node, unlike e.g. with majority voting.

We finally present a comparison of the achieved speedup S of the static class, the adaptive class, and BOINC. We computed the speedup as shown in Section 5.3. In Figure 8 we show that the adaptive method performs best keeping the speedup at the highest rate (blue line). Close to this, we see BOINC (black

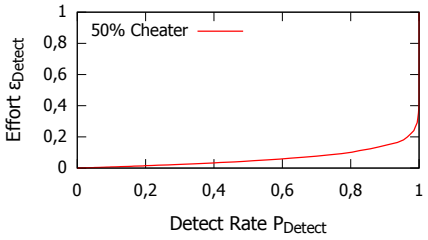

Fig. 5. Cheat Detection Effort of a Single Simulated Node – 50% Cheater Type

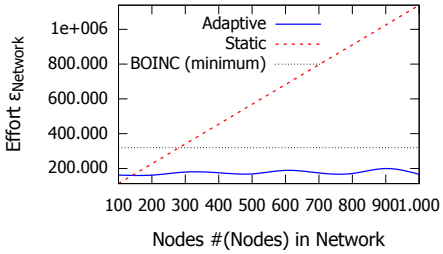

Fig. 6. Cheat Detection Effort – Static vs Adaptive Cheat Detection vs BOINC

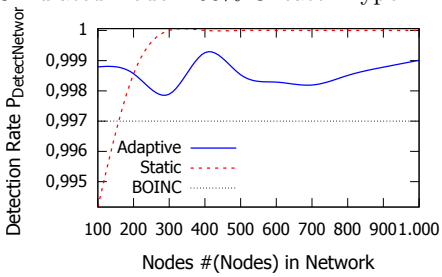

Fig. 7. Detection Rate – Static, Adaptive, BOINC

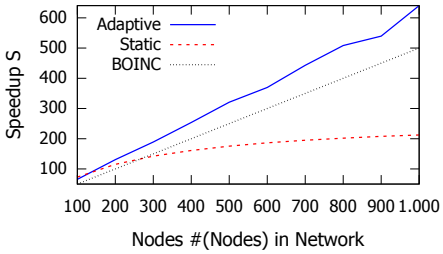

Fig. 8. Speedup – Static, Adaptive, BOINC

line). We furthermore see, that the static class reaches a speedup limit close to 210 which confirms that the static method does not scale.

6 Related Work

Prior work has considered cheat detection and cheat prevention in distributed computing, which includes volunteer computing. There are attempts to secure volunteer computing and grid computing by introducing mechanisms to either validate the correctness of results received from participants or by making it hard or impossible to cheat on given jobs. In [7], Golle and Mironov describe their idea of uncheatable distributed computations. They show two different security schemes, a weak and a strong one, that defend against cheating participants. The weak one depends on 'magic numbers' and the strong one depends on a supervisor and so called 'ringers'. Both schemes have in common that participants have to find either these magic numbers or the ringers to get rewarded for their done work. The main difference between their solution and ours is the organization of the computations. They use the supervisor who assigns subjobs to nodes. In our scenario, we have no central management since our network is completely unstructured and decentralized. Moca et al. present in [11] a method for distributed results checking for 'MapReduce' [5] in volunteer computing. They use a distributed result checker based on majority voting. Furthermore, they

developed a model for the error rate of 'MapReduce'. Compared to our solution, which only needs recomputation of very small parts of subjobs, their method is based on majority voting. Thus, their nodes have to redundantly compute complete subjobs. Hence, the total amount of recomputations is at minimum 2 to apply majority voting on their results. In this paper, we show that our method needs considerably less recomputations compared to their method. Zhao and Lo show their scheme 'Quiz' in [16], which inserts indistinguishable quiz tasks with verifiable results to a distributed job. They outperform the method of the replication of jobs in terms of accuracy and overhead under collusion assumptions. Compared to our solution, they need a central server that assigns the quiz tasks as well as the regular tasks to the clients. With our solution, we do not need a central server since every node in the network performs a small part of detection work. Sarmenta presents in [12] his sabotage-tolerant mechanisms for volunteer computing. He shows a method called 'credibility-based fault-tolerance' where he estimates the conditional probability of (sub/job)-results and workers being correct, based on the results of voting, spot-checking and other techniques, and then he uses these probability estimates to direct the use of further redundancy. Compared to our solution, they use a work pool-based master-worker model where the master assigns subjobs to workers. Here, the master randomly assigns redundant subjobs to workers. Compared to majority voting based techniques, their method reduces the total amount of recomputations, but also increases the chance of non-detected cheated subjob results. Our method also reduces the needed amount of recomputation but still keeps a very high detection probability with small network and node effort.

7 Conclusions

We introduced a cheat detection method for decentralized, gossip-based volunteer computing networks. The method is suitable for ad-hoc computations without setting up a central server. Nodes do not need to trust each other, which makes the method robust against colluding cheaters or cheaters who decide to abuse their acquired reputation. We are targeting opportunistic cheaters, which are attempting to collect the same reward as well-behaving users, e.g. credit points. However, cheaters provide job results that are incorrect and incomplete.

The method works by sample testing each job result that is disseminated in the network. With a static cheat detection effort per node, the scalability is limited as the system approaches an upper bound on the speedup with an increasing number of participating nodes. We thus adapt the cheat detection effort subject to the workload of the network, which is determined by each node based on the number of results disseminated in the network. This allows us to achieve a given target detection rate efficiently, e.g. 99.9% with only about 50.6% recomputations of subjobs. As we have shown with our simulations, such a decentralized sample testing is more efficient than a server-coordinated majority voting like e.g. being used by BOINC. In future work, we will analyze how cheat detection results can be used to exclude cheaters from our system and reduce

the detection effort by doing so. Furthermore, we will examine how such a node exclusion can be used by attackers to remove well-behaved nodes.

References

1. Anderson, D.P.: Boinc: A system for public-resource computing and storage. In: Fifth IEEE/ACM International Workshop on Grid Computing, 2004. pp. 4–10. IEEE (2004)
2. Anderson, D.P., Fedak, G.: The computational and storage potential of volunteer computing. In: Sixth IEEE International Symposium on Cluster Computing and the Grid, 2006. CCGRID 06. vol. 1, pp. 73–80. IEEE (2006)
3. Anderson, D.P., Korpela, E., Walton, R.: High-performance task distribution for volunteer computing. In: First International Conference on e-Science and Grid Computing, 2005. pp. 8–pp. IEEE (2005)
4. Daemen, J., Rijmen, V.: The design of Rijndael: AES-the advanced encryption standard. Springer Science & Business Media (2013)
5. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM 51(1), 107–113 (2008)
6. Domingues, P., Sousa, B., Moura Silva, L.: Sabotage-Tolerance and Trust Management in Desktop Grid Computing. Future Generation Computer Systems 23(7), 904–912 (2007)
7. Golle, P., Mironov, I.: Uncheatable Distributed Computations. In: Topics in Cryptology - CT-RSA 2001, pp. 425–440. Springer (2001)
8. Kopal, N., Heck, H., Wacker, A.: Simulating cheated results acceptance rates for gossip-based volunteer computing. International Journal of Mobile Network Design and Innovation 7(1), 56–67 (2017)
9. Kopal, N., Kieselmann, O., Wacker, A.: Self-organized volunteer computing. In: Organic Computing: Doctoral Dissertation Colloquium 2014. vol. 4, pp. 129–139. kassel university press GmbH (2014)
10. Kopal, N., Kieselmann, O., Wacker, A.: Simulating cheated results dissemination for volunteer computing. In: 2015 3rd International Conference on Future Internet of Things and Cloud (FiCloud). pp. 742–747. IEEE (2015)
11. Moca, M., Silaghi, G.C., Fedak, G.: Distributed Results Checking for MapReduce in Volunteer Computing. In: 2011 IEEE international symposium on Parallel and distributed processing workshops and Phd Forum (IPDPSW). pp. 1847–1854. IEEE (2011)
12. Sarmenta, L.F.: Sabotage-Tolerance Mechanisms for Volunteer Computing Systems. Future Generation Computer Systems 18(4), 561–572 (2002)
13. Shannon, C.E.: Prediction and entropy of printed english. Bell system technical journal 30(1), 50–64 (1951)
14. Wander, M., Wacker, A., Weis, T.: Towards peer-to-peer-based cryptanalysis. In: IEEE 35th Conference on Local Computer Networks (LCN), 2010. pp. 1005–1012. IEEE (2010)
15. Wander, M., Weis, T., Wacker, A.: Detecting opportunistic cheaters in volunteer computing. In: 2011 Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN). pp. 1–6. IEEE (2011)
16. Zhao, S., Lo, V., Dickey, C.G.: Result Verification and Trust-Based Scheduling in Peer-to-Peer Grids. In: Fifth IEEE International Conference on Peer-to-Peer Computing, 2005. P2P 2005. pp. 31–38. IEEE (2005)