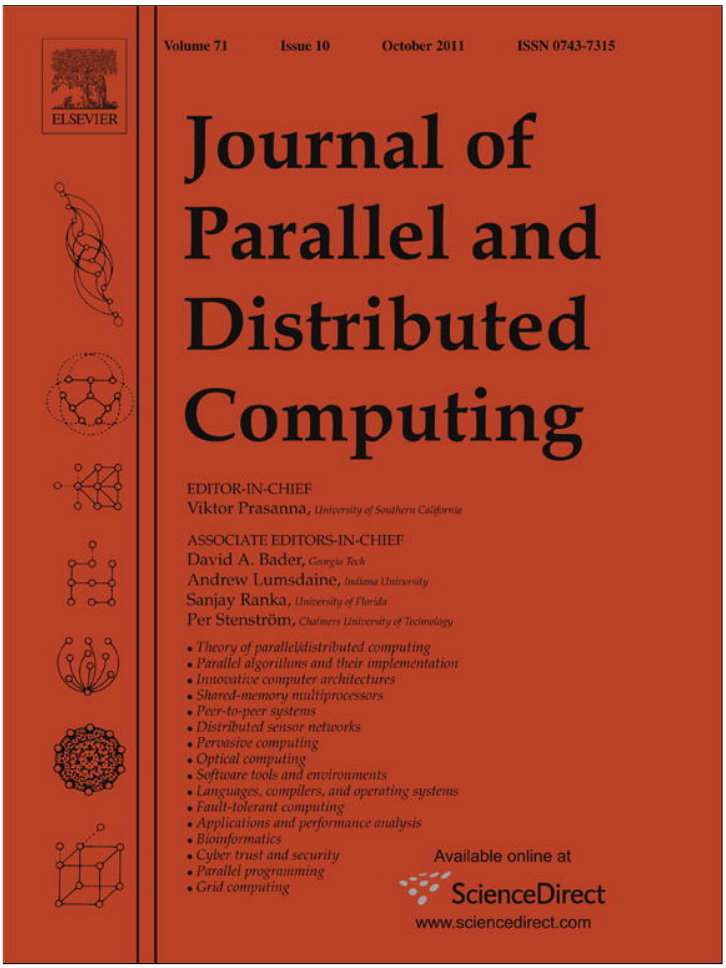


J. Parallel Distrib. Comput. 71 (2011) 1356–1366

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A maximum independent set approach for collusion detection in voting pools

Filipe Araujo [a,*], Jorge Farinha [a], Patricio Domingues [b], Gheorghe Cosmin Silaghi [c], Derrick Kondo [d]

[a] CISUC, Department of Informatics Engineering, University of Coimbra, Portugal
[b] Research Center for Informatics and Communications, School of Technology and Management of the Polytechnic Institute of Leiria, Portugal
[c] Department of Business Information Systems, Babeș-Bolyai University, Cluj, Romania
[d] Laboratoire d'Informatique de Grenoble, INRIA Rhône-Alpes, Grenoble, France

ARTICLE INFO

Article history:
Received 21 May 2010
Received in revised form
31 May 2011
Accepted 16 June 2011
Available online 24 June 2011

Keywords:
Collusion detection
Maximum independent set
Volunteer computing

ABSTRACT

From agreement problems to replicated software execution, we frequently find scenarios with voting pools. Unfortunately, Byzantine adversaries can join and collude to distort the results of an election. We address the problem of detecting these colluders, in scenarios where they repeatedly participate in voting decisions. We investigate different malicious strategies, such as naïve or colluding attacks, with fixed identifiers or in whitewashing attacks. Using a graph-theoretic approach, we frame collusion detection as a problem of identifying maximum independent sets. We then propose several new graph-based methods and show, via analysis and simulations, their effectiveness and practical applicability for collusion detection.

1. Introduction

We often find situations where people vote to decide on some course of action. Similarly, processes in distributed systems may also need to vote or compare results for a number of reasons, ranging from replica management [15] to many agreement problems [21,13], including replication of software components in safety-critical systems [22]. By running several replicas written by different teams and running on different hardware, the likelihood of correlated failures drops. In [31], the authors propose to use voting replicas to tolerate not only failures, but actual intrusion by hackers able to manipulate and even totally control the replica. In eBay and in other e-commerce sites, we can find recommender systems where users give feedback to each other, in a process we can consider as voting [1,29]. The BOINC middleware [2], namely the SETI@home project, provides another example of voting in distributed volunteer computing. Workers download workunits from the BOINC server, compute them, and send their results back. However, since volunteers are unreliable, the central supervisor assigns the same workunits to two or more workers. These form voting pools, which are decided by the central supervisor according to the majority of results.

Unfortunately, workers may resort to several forms of manipulation to negatively affect the decisions of the voting pools. In this paper, we tackle the problem of detecting colluding nodes that jointly sabotage these pools, in BOINC-like scenarios. We assume that nodes successively gather in different pools to vote. This enables the central supervisor to collect tallies, based on the history of each node. Additionally, one or more subsets of the nodes, which we name "malicious" or "incorrect", may compute wrong results, and may even *collude* (i.e., we consider particular cases of Byzantine behaviors [21]).

Some malicious nodes, which we term as "naïve malicious", act alone, e.g., because they have faulty hardware. Unlike these, colluder nodes have the ability to communicate using out-of-band mechanisms, to determine whether or not they should produce an erroneous result. We consider some different types of these colluder nodes: some of them always produce wrong results when they are in majority, while others may betray their peer colluders, or refrain from cheating, to disguise their behavior from the central supervisor. Moreover, since it is often impossible to ensure that each worker presents only one identifier to the system, we also consider colluder nodes capable of performing whitewashing attacks [11]. In this case, they may leave and later rejoin the system with a fresh identifier, to circumvent blacklists.

In this paper, we propose a number of collusion detection algorithms that aim to uncover malicious nodes. We base these algorithms on a graph defined by the votes that nodes do against each other, the *Votes Against Graph*, and on the *Maximum Independent Set Problem* (MIS). This will enable the central supervisor to take corrective measures to ensure validity of results. For example, it may request additional workers for some possibly incorrect voting pools.

* Corresponding author.
E-mail addresses: filipius@dei.uc.pt, filipius3@gmail.com (F. Araujo).

0743-7315/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2011.06.004

The main contributions of this paper are the following:

- When correct nodes define the largest plurality, we show that a MIS can limit the number of correct nodes the central supervisor considers as malicious, and also gives probabilistic limits on the number of attacks performed by malicious nodes. However, since the MIS problem is NP-complete [14], we must resort to heuristics to detect colluders.
- Hence, we propose a collusion detection algorithm based on the "Breadth-First Search" (BFS) of the votes against graph. BFS works very well if malicious nodes never vote against each other.
- For more general settings, we propose the "Against Votes" algorithm. One version of this algorithm is resistant to whitewashing attacks.
- We observed experimentally that either malicious nodes contribute to the community with valid votes, or the central supervisor may eventually spot them. This is a fundamental limitation of malicious nodes also observed in [32].

In Section 2, we present related work. In Section 3, we define the problem settings. In Section 4, we introduce the sabotage models we consider in this paper. In Section 5, we theoretically evaluate the application of the MIS problem to collusion detection. In Section 6, we present the collusion detection algorithms. In Section 7, we show the experimental results. Finally, in Section 8, we conclude the paper.

2. Related work

In this section, we summarize related work concerning detection of malicious behavior (also sabotage or manipulation) in voting pool scenarios. In volunteer computing, the sabotage tolerance concept was firstly coined by Sarmenta [28], to present techniques that can overcome the presence of malicious volunteers in the contributors' set. Early works on sabotage tolerance [28,16,9,2] assume a basic naïve model of saboteurs. These workers make mistakes with some individual sabotage probability, independently of the behaviors of other nodes. Given these assumptions, majority voting-based replication [2,28,34] was conceived as a simple but very effective [20] tool to fight saboteurs. Other more complicated techniques were devised, including result verification based on sampling [16,9,40,36] and credibility [28,40,36]. Sampling means that the trustworthiness of each worker is verified from time to time, e.g. by administrating them some quizzes or by auditing their results. In credibility-based schemes, the master observes the behavior of workers during the computation and estimates their reliability. Kondo et al. [20] proved the effectiveness of replication in BOINC-like setups, demonstrating that an error rate of 10^{-2} per project can be achieved, given the real-life observed behavior of workers. Replication has the big drawback of the computation cost—each task being scheduled to at least two workers. Sampling and credibility leave the door open for other attacks: a worker can behave well for a long period of time in order to gain reputation and attack after that.

With only naïve saboteurs, Wong [37] exploits the graph of nodes reflecting how these nodes are linked together in voting pools. For the naïve model of saboteurs, a detailed discussion can be found in [8].

More sophisticated malicious behaviors in volunteer computing were introduced in [30]. Given that some out-of-band communication is enabled between workers, they can collude and attack in concentration. To detect the colluding behavior, we described a statistical tool, which combined with further result audits, removes the negative effects of nodes' collusion on the global computation. The collusion problem was also analyzed by [32]. Based on the probability that two nodes are together in the same majority/minority group of a vote, they arrange the nodes in an undirected weighted graph and employ a clustering procedure to partition the graph in a cluster of nodes that correlate highly. This approach is very effective when the correlation probability of two nodes is estimated based on more than 10 observations in average, which might be difficult to attain in practical distributed system setups, characterized by a huge number of nodes and a very low probability that two nodes meet together in more than two voting pools. Canon et al. [5] evaluate a setting close to our own. They present two algorithms implementing a dynamic merge/split of groups of nodes with collusion or agreement behavior, given that workers' behavior is stationary during their participation in the system. They conclude that the collusion method is slightly better than the agreement one, while the agreement one is much simpler to implement because it does not rely on an external result certification mechanism. We differentiate from them as in our work we do not make any assumption about workers' behavior stability during their life in the system.

The setting of Fernandez et al. [12] relates to our own, as the authors consider sets of binary tasks (with 0 or 1 output) controlled by a master and distributed to workers. They compute upper bounds for the computational effort of volunteers, given some limitations on the power of (Byzantine) malicious nodes.

Electronic commerce is another setting that can be modeled with voting pools (games) and is also subject to manipulation. Tsvetovat et al. [35] show that coalitions among customers can be formed to buy many items from the buyer at a lower price and further, to benefit from this. Anonymous environments (like the Internet), with the same players bidding in successive voting pools allow manipulation, which is almost impossible to detect [38]. In combinatorial Internet auctions, the concern was to develop anonymity-proof interaction protocols [38] or false-name proof protocols [39]. In our distributed computing setup, we rely on the standard master–worker computational model with its simple pull or push scheduling; we differentiate from this huge amount of literature originating in artificial intelligence as we do not design the interaction. We search for the manipulation (sabotage) after it happened and try to limit its effects on the system.

Regarding the Maximum Independent Set problem, we can find frequent references to its use in settings involving combinatorial auctions, to demonstrate the complexity of determining bid winners [26,3].

3. Problem setting

In this paper, we consider a BOINC-like setup [2], comprised of a central supervisor that owns a set of workunits (or jobs), plus a fixed set of workers that volunteer to compute these workunits. To cope with incorrect nodes, the central supervisor creates *voting pools* of different nodes computing the same workunit replicas. Unless stated otherwise, neither do we assume any knowledge on how the central supervisor assigns nodes to voting pools, nor we depend on their size. Nevertheless, in Section 7, and for experimental purposes, we describe how we set these. Each worker does not participate twice in the same pool, but it can participate in any two different pools.

Upon construction of a p-nodes pool, workers perform some computation and produce an outcome (also *vote* or *result*) from the following domain: $\{1, 2, \ldots, p\}$. All correct nodes pick the same outcome. Each naïve malicious node picks one different outcome (and different from any other kind of node). Malicious colluders can pick any outcome, but different from all the outcomes of naïve malicious nodes. In this way, we represent naïve malicious nodes as bogus computers that produce random outputs.

We use majority to set pool decisions as soon as nodes upload all the results. The most important differences for BOINC starts at

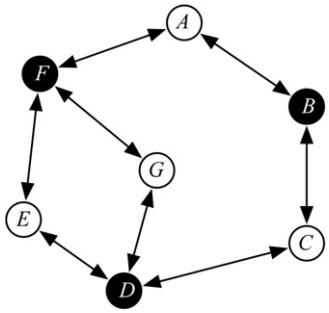


Fig. 1. Votes against graph.

this point. Given a list of pools, we tackle the problem of classifying nodes as correct or incorrect, based on the votes they did against each other. We refer to this as the *collusion detection problem*, and we propose several algorithms to solve it in Section 6. We do not need to wait for all the voting pools, before detecting colluders, although, in Section 7.3, we show that many algorithms need 30 or more pools per node to produce good results.

After determining which nodes are correct and incorrect, we have a few different options regarding their voting pools, like accepting or rejecting initial majority-based decisions, or requesting for additional workers to confirm results. For instance, the central supervisor may request additional computation in pools entirely made of incorrect workers (in the hope of finding actually correct results), or it can confirm results from workers it considers as correct, but that have a very short history in the system. However, this choice comes after the collusion detection algorithm and does not affect it. For this reason, we do not reassign workunits in our experiments. In Section 7, we just count the errors in pools caused or not identified by our collusion detection mechanisms.

Colluders are able to find each other using out-of-band communication, but they cannot determine the identifiers of correct nodes voting with them.

We initially assume a one-to-one relation between identifiers and workers. However, in many cases, this may be a very limiting assumption. Therefore, in Section 7, we also consider the whitewashing attack, where malicious workers may present new identifiers to the system, over time. This presents an apparently larger set of workers to the central supervisor.

We argue that the scope of our work may reach other settings besides BOINC. We cite two cases: one includes voting games [18,10]. Another related, although different scenarios exist in e-commerce system (like eBay), where buyers and sellers give grades to each other after each transaction. In this case, if the buyer has a positive opinion of the seller, they cast similar votes, whereas a negative opinion corresponds to casting different votes. One should note that buyers with contradictory opinions also vote against each other. Nonetheless, this setting is not exactly the same as ours, because sellers may also collude.

Based on the sequence of voting pools, we define a "graph of votes against", $G = (V, E)$, where V is the set of vertices, which we also call "workers" or "nodes". E is the set of edges of the graph. Edge E_{ij} connecting nodes $N_i, N_j \in V$ exists if and only if N_i and N_j voted against each other in some voting pool (independently of the number of workers per voting pools). We draw an example of such a graph in Fig. 1. In this graph, A has "votes against" B and F. B also voted against C, etc. However, this graph does not show the notion of "defeats". A node has a defeat in every voting pool where the decision is different from its vote. Therefore, this number can vary between 0 and the number of pools where the node participates.

Finally, we introduce the notion of *Maximum Independent Set* (MIS). In a graph, an MIS is a *maximal* independent set with largest cardinality [17]. Informally, this is the largest set of vertices from a graph such that no edge connects two of those vertices.

4. Sabotage model

To characterize erroneous hosts, we initially consider two models, as we did in [30]: *naïve malicious*, and *colluding malicious*. A naïve malicious node randomly commits mistakes in some workunits independently of the behavior of other nodes in the voting pool. This captures software or hardware malfunctions, e.g., resulting from CPU overclocking [33] or any hardware silent malfunction, like memory errors [25]. Unlike this, we assume that *colluding nodes* introduce the *same* error only when they are sure that their sabotage can be successful, because they form a majority in their voting pool. We further extend this model to take more voting pools into account.

We denote basic naïve malicious nodes by M_1-*type*. An M_1-type worker submits bad results with a constant probability, called a *sabotage probability*. Sarmenta proposed and studied this model in [28].

We also consider a predictable kind of saboteur, the M_2-*type* worker, which makes its decisions based on the single specific voting pool it is in. An M_2-type worker will sabotage a result whenever M_2 nodes form a majority in the voting pool. We assume that these nodes can communicate using some out-of-band mechanism, e.g., social networks in the Internet. We also assume the worst-case setting, where malicious nodes can determine the size of their voting pool.

Eq. (1), which we derived from the Hypergeometric distribution, gives the expected error probability $\epsilon(f, p, n)$ for M_2 nodes. In the equation, n is the total number of nodes in the population (correct and M_2-type nodes). Variable p is the voting pool size, which we consider to be odd. Hence, majority of colluders needs at least $(p+1)/2$ votes. Variable f is the fraction of M_2 nodes. We consider $fn \geq p$ and $(1-f)n \geq p$.

$$\epsilon(f, p, n) = \sum_{j=(p+1)/2}^{p} \frac{\binom{fn}{j}\binom{(1-f)n}{p-j}}{\binom{n}{p}}. \quad (1)$$

In our previous work [30], we considered one additional type of saboteurs deemed M_3, mixed malicious colluders, which change their behavior during their lifetime, either attacking naïvely or colluding. However, we did not consider this kind of worker in this paper, because it does not bring any new real behavior. Instead, we introduce another type of malicious worker, the M_4-*type*, which can betray its partnering colluders to cover its course of action. After finding peer colluders and deciding for the sabotage, an M_4 node may retreat and vote correctly. This is useful to overcome collusion detection mechanisms. Hence, we can say that M_4 nodes take the resistance against collusion detection algorithms a step further than M_2 nodes, because they disguise their behavior even further, by voting against each other. As we shall see, this causes widespread failure on some detection algorithms. The cost for colluders, however, is that they lose much of their attacking power.

The problem with the M_1 up to M_4 colluders is that they set their decision to collude based on the current voting pool alone, i.e., independently of what they did in past voting pools. Unlike this, the malicious M_5-type workers only decide to sabotage when they are sure to be undetectable by an MIS algorithm. To do this, they must keep a history of past actions. We only implemented this scheme for 3-node voting pools. If three M_5 nodes meet in the same voting pool, they will always sabotage. A single M_5 node never sabotages. The difficult decision occurs when only two M_5 nodes meet in the same voting pool. To tackle this case, each M_5 node keeps a set of peers that voted in coalition against correct nodes. Consider this to be S_1 and S_2, respectively for nodes N_1 and N_2. In the beginning $S_1 = \{N_1\}$ and $S_2 = \{N_2\}$. After N_1 and N_2 vote against a correct node, we define a new peer set $S_1 \cup S_2$, which

Table 1
Short description of malicious (also incorrect) nodes.

Name	Description
M_1	Naïve malicious node
M_2	Colluder malicious node
M_4	Unreliable colluder malicious, which may betray each other
M_5	MIS-resistant colluder malicious

becomes the peer set for all nodes in $S_1 \cup S_2$ (including N_1 and N_2). If $S_1 \cap S_2 \neq \emptyset$, N_1 and N_2 cannot sabotage together in the same voting pool. The previous scheme ensures that M_5 nodes will always belong to an MIS (given that all the other nodes are correct). Briefly, the idea of the proof is that the votes against graph is a tree (possibly a forest). Whatever root we pick for (each one of) the tree(s), all leafs are M_5 nodes. Also, correct and M_5 nodes do not have edges to their own type of nodes. Therefore, M_5 nodes outnumber correct nodes in this graph, thus forming an MIS.

To summarize, we deem as "correct" any node that never produces a bad result. In general, any other node is "incorrect" or "malicious". M_1 are "naïve" malicious, while M_2, M_4 and M_5 are "colluder" malicious. Table 1 shortly describes the colluder nodes we use throughout this text.

5. Analysis of maximum independent sets for collusion detection

Consider a votes against graph, like the one of Fig. 1, but with only three nodes A, B, C, where B and C voted together against node A. If we do not make any assumptions about malicious nodes, we can never be certain about which nodes are correct, e.g., A or B and C. In some distributed environments like BOINC we could explicitly audit some results. In other settings, like for instance in some trade, one could also perform similar verifications, but we are not considering such an option in this paper. Consequently, unambiguous detection of colluders, for a general votes against graph is impossible.

Nevertheless, we can still take advantage of one fact to approximate the set of correct nodes: given $G = (V, E)$, the set of correct workers must necessarily define an independent set $H \subset V$. If we assume that the largest plurality of nodes that do not vote against each other is correct, as in the BOINC application of Kondo et al. [20], the detection of colluded nodes resembles the MIS problem. However, it is trivial to find examples where the correct nodes define an MIS, and it is equally trivial to find counter-examples of this. This raises the question of knowing the precision we can expect from MIS. We demonstrate two results. Informally, Theorem 5.1 limits the number of correct nodes the central supervisor can misclassify as incorrect. Theorem 5.2 limits the probability that the central supervisor misclassifies an entire incorrect malicious group as correct.

In what follows, $G = (V, E)$ is the votes against graph, H is the set of correct nodes, B is the set of incorrect nodes. $V = B \cup H$ and $B \cap H = \emptyset$. $h = |H|$, and $b = |B|$. This entails that some malicious adversary controls b malicious nodes and none of the h correct nodes. We assume that $h > b$. $C \subseteq H$ is a subset of correct nodes, and $c = |C|$.

In Theorem 5.1, we show that MIS can misclassify at most b out of the h correct nodes, in the hopefully common situation where correct nodes outnumber incorrect ones.

Theorem 5.1. *If $h \geq b$, the Maximum Independent Set can exclude at most b correct nodes.*

Proof. Assume that $c > b$ correct nodes do not belong to MIS. Each one of these c correct nodes must have one or more edges to one or more of the b malicious nodes in MIS (otherwise they would be part of MIS). Since $c > b$, these b nodes must not all belong to MIS, which is a contradiction, and the theorem follows. □

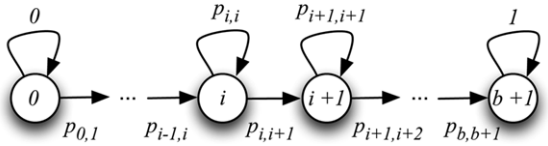


Fig. 2. Markov chain of the number of different correct nodes.

Now, consider that b malicious nodes voted x times (in x different voting pools) against nodes from H. Since malicious nodes do not know the identifiers of correct nodes they vote against, the best they can do is to play their odds: if the central supervisor randomly selects one correct node from H in x different trials (pools), what is the probability that *at most* c out of h possible identifiers exist among the x? Eq. (2) gives the exact value of this probability, $P(x, h, c)$ [19].

$$P(x, h, c) = \sum_{j=0}^{c} \left[\binom{h}{j} \sum_{k=0}^{j} (-1)^k \binom{j}{k} \left(\frac{j-k}{h} \right)^x \right]. \tag{2}$$

Based on this equation, we show that the probability that MIS includes all malicious nodes decreases to 0, as they repeatedly vote against unknown correct nodes. We assume that all correct nodes are equally likely to participate in any voting pool.

Theorem 5.2. *The probability that MIS includes the set B of malicious nodes, after having x votes against unknown correct nodes decreases with $O\left(\left(\frac{b}{h}\right)^x\right)$.*

Proof. We start with the assumption that all correct nodes belong to different voting pools. From Theorem 5.1, if b colluder nodes vote against $c > b$ (different) correct nodes, one or more of the b colluder nodes must not belong to MIS. Hence, MIS may only include the entire set B of colluder nodes if they have votes against $c \leq b$ *different* correct nodes. From Eq. (2):

$$\begin{aligned} P(x, h, b) &= \sum_{j=0}^{b} \left[\binom{h}{j} \sum_{k=0}^{j} (-1)^k \binom{j}{k} \left(\frac{j-k}{h} \right)^x \right] \\ &\leq \sum_{j=0}^{b} \left[\binom{h}{j} \left(\frac{j}{h} \right)^x \sum_{k=0}^{j} \binom{j}{k} \right] \\ &\leq \left(\frac{b}{h} \right)^x \sum_{j=0}^{b} \left[\binom{h}{j} \sum_{k=0}^{j} \binom{j}{k} \right] = \alpha \left(\frac{b}{h} \right)^x \end{aligned} \tag{3}$$

where we can treat $\alpha = \sum_{j=0}^{b} \left[\binom{h}{j} \sum_{k=0}^{j} \binom{j}{k} \right]$ as a constant, since b and h are fixed for a given setting.

We now consider the case where correct nodes can share voting pools. This corresponds to run some trials without replacement of correct nodes, because these cannot participate twice in the same voting pool. We resort to a Markov chain (refer to Fig. 2) to confirm the intuition that colluder nodes are more likely to vote against more than b different correct nodes. State i represents the case where colluder nodes voted against i *different* correct nodes. If the next vote is against one of the previous i correct nodes, the system stays in state i, otherwise, since the correct node is new, state changes to $i+1$. The probability of staying in the same state i is $p_{i,i}$; the probability of changing to state $i+1$ is $p_{i,i+1}$; $p_{i,i} + p_{i,i+1} = 1$. The final state $f = b + 1$ is absorbing. It corresponds to the case where colluder nodes voted against more than their own number of correct nodes. The probability of reaching the last state from state i, after x trials, $p_{i,f}^{(x)}$, for $0 \leq i < f$, is [4] (for simplicity we omit the number of correct nodes h from the notation):

$$p_{i,f}^{(x)} = p_{i,i} p_{i,f}^{(x-1)} + p_{i,i+1} p_{i+1,f}^{(x-1)}. \tag{4}$$

Note that when we only have one correct node per pool, $p_{0,f}^{(x)} = 1 - P(x, h, b)$. Since $p_{i,i+1} = 1 - p_{i,i}$, and $p_{i,f}^{(x-1)} < p_{i+1,f}^{(x-1)}$, when $p_{i,i}$ decreases, $p_{i,f}^{(x)}$ increases. When correct nodes share voting pools, $p_{i,i}$ is smaller than or equal to the case where they cannot share voting pools. This results in larger or equal $p_{i,f}^{(x)}$ for all i, including $p_{0,f}^{(x)}$. Hence, it is less or equally likely that colluder nodes vote against *at most* b out of h correct nodes after x trials, and the theorem follows. □

Note that $\alpha \left(\frac{b}{h}\right)^x$ may be larger than 1, for values of x below some threshold. Moreover, fraction b/h plays a crucial role on how fast the probability decreases. For instance, in some of the settings we test in this paper, $b = 200, h = 800$. If $x \geq 231$, i.e., if the entire set of malicious nodes votes against 231 or more correct nodes (without knowing their identifiers), malicious nodes have more than 0.5 chance of having 201 or more *different* neighbors in the votes against graph, because $P(230, 800, 200) \approxeq 0.47$. Additionally, this probability decreases to 0.41 and 0.35, for $x = 232$ and $x = 233$, respectively.

6. Collusion detection algorithms

Unfortunately, computing an MIS is an NP-complete problem [14]. Consequently, we need to use heuristics to identify malicious nodes. In this section, we present the different methods evaluated in this study.

6.1. Defeats

The simplest heuristic to determine colluders counts the number of defeats. One node has a defeat each time the pool decision, based on the majority, is different from its vote. This heuristic is specially tailored to detect M_2 nodes because these have no defeats. If only M_2 and good nodes exist, as the number of times nodes participate in voting pools grows to infinity, this algorithm will reach perfection. Unfortunately, in practice, this convergence is too slow, and this algorithm shows (predictably) disappointing results. Therefore, we exclude it from the plots of Section 7. Especially when the participation of nodes in voting pools is small, it is likely that many nodes, including correct nodes, will show no defeats, thus making impossible to distinguish correct from incorrect nodes.

6.2. Breadth-first search (BFS)

Neither M_2-type colluders nor correct nodes do ever vote against any node of the same type. Therefore, if only M_2 and correct nodes exist in a population, we can take advantage of this fact to spot M_2-type colluders. Consider the graph of votes against, $G = (V, E)$. We can run a breadth-first search [17] (BFS), starting at some random node to create layers of nodes. BFS has a cost of $O(|V|+|E|) = O(n^2)$, for $n = |V|$ nodes, in the worst case. Consider the example of Figs. 1 and 3, which respectively show a graph G and the resulting BFS graph. The latter figure also shows the distance from the root A (picked randomly) to every single node. Distance is the number of edges in the shortest path, from the root to a given node in the BFS graph. Since correct nodes never vote against each other and neither do M_2 nodes, alternate layers of the BFS graph have different types of nodes: all correct nodes will lie in the odd (even) layers and all M_2 nodes will lie in the even (odd) layer. We prove this in Theorem 6.1.

Theorem 6.1. *Consider the graph $G = (V, E)$ defined as the votes against graph, such that all nodes in V are either correct nodes or M_2 nodes. If G is connected, the BFS algorithm will find either an odd distance to all correct nodes and even distance to all M_2 nodes, or vice versa.*

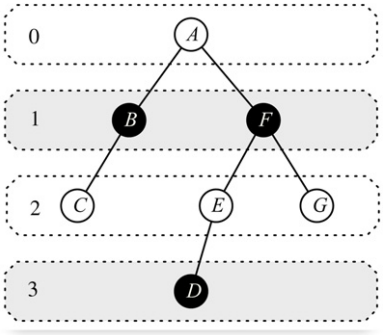


Fig. 3. Resulting BFS graph.

Proof. First, assume that the root used in the BFS algorithm is a correct node. Since the graph is connected, there must be one or more nodes at a distance 1. Since correct nodes do not vote against each other, all nodes at distance 1 must be malicious M_2 nodes. If, on the contrary, the root is an M_2 node, the same reasoning allows us to conclude that all nodes at distance 1 are correct nodes. Now, consider that all nodes at distance d from the root are M_2 nodes. No node at distance $d + 1$ can be an M_2 node, otherwise M_2 nodes would be voting against each other. If there is any node at distance $d + 1$, it must be a correct node. Again, we can use the exact same reasoning to show that if nodes at distance d are correct, then nodes at distance $d + 1$ are necessarily M_2. By induction, we can see that correct and M_2 nodes must alternate and thus keep their parity for arbitrarily large distances. □

We assume that the majority of nodes is correct [20] and, therefore, we take as correct the nodes in the most populated parity layers (even layers in the case of Fig. 3), as well as nodes without votes against. Also, note that, when G is disconnected, running BFS multiple times will result in a forest instead of a tree. Although we apply the same method on each tree, detection of malicious nodes becomes more difficult, as we show in Section 7.

6.3. Perfect threshold (PT)

This algorithm uses unfair knowledge of which nodes are correct and which nodes are malicious. It is unfair, because it is available for the simulator, but it would not be available in real life. The idea is to sort nodes according to their number of votes against. Then, based on this order, we set a threshold to separate correct nodes from malicious ones. Any node with more votes against than the threshold is a malicious node, and any remaining ones are correct. To provide an example, consider that we have 31 correct and 13 malicious nodes and that we set the threshold to be 8. Perfect threshold will consider any node with 8 or less votes against as correct and any node with more than 8 votes against as malicious. Now assume that 14 of the 31 correct nodes and 6 of the 13 malicious nodes have 8 or less votes against, whereas 17 of the 31 correct nodes and 7 of the 13 malicious nodes have more than 8 votes against. In this case, the algorithm will incorrectly classify 6 malicious nodes as correct ones and 17 correct nodes as malicious ones. What we do is to try every number $1, 2, 3, \ldots$, up to the highest number of votes against, as the threshold that separates correct nodes from malicious ones.

One of all possible thresholds will ensure the smallest number of misclassifications for both correct and malicious nodes (in the case of a tie we consider the smallest threshold). We call this algorithm "Perfect Threshold" because no other threshold ensures

a better separation. One should notice that we resort to omniscient knowledge about the nodes to achieve this separation, as we must already know beforehand which nodes are correct and which ones are malicious.

6.4. MIS-based heuristics: Higher Order (HO) and GMIS

Finding an MIS is an NP-complete problem [14]. Thus, we should use heuristics to solve it. From [6], we implemented and used the Higher-Order Ratio Heuristic Algorithm. This algorithm runs several iterations trying to improve the solution initially found. The authors do not give a precise limit for the number of iterations. Hence, we set this number to 70, because this number is much larger than what the graphs presented in [6] require to converge, thus offering a safety margin. Although the heuristic produces valid results in our experiments of Section 7, we could not reproduce the results for the graph specified in Figure 4 of [6]. (The problem is not related to the number of iterations, as the final results should emerge on the seventh iteration.)

We used another MIS-based heuristic from Resende et al., where they present a Greedy Randomized Adaptive Search Procedure (GRASP) [27]. The authors implemented and distributed GRASP in a set of Fortran routines called `gmis`. We do not include the results of `gmis`, because we found it to be orders of magnitude slower than other heuristics, while achieving results similar to Higher-Order algorithms in a limited set of experiments.

6.5. K-means

We also use the K-means clustering algorithm [24]. The idea is to cluster nodes according to their behavior. We use a two-dimensional space defined by the number of defeats and the number of votes against each node. The idea of using a clustering algorithm comes from the fact that the number of defeats and votes against define the behavior of a node, and nodes that have similar behaviors will tend to define groups within this two-dimensional space. One of the problems of K-means is that the number of clusters is an input variable. We set this number to 2, since we are interested in having two clusters: one of correct nodes, the other of malicious nodes. It is not difficult to anticipate that attaining such a sharp distinction may be difficult in practice. As we shall see in Section 7, this is especially true when there are more than two types of behaviors.

6.6. Against Votes 1 (AV1)

One very simple mechanism to detect colluders uses a counting of the votes against collected by each node (i.e., edges of G). The idea is to sort the nodes in descending order by votes against and start with the node with the largest number of votes against, then continuing down to the node with fewer votes against (possibly 0). The algorithm classifies any node that has more votes against than average as a malicious node. Then, whenever it finds a malicious node, it deletes it from the graph. Although this impacts the number of votes against of some other nodes (which voted against the node being deleted), we do not resort the list to reduce complexity.

The rationale for the algorithm is that even if some correct node collects many votes against, as we remove malicious nodes acting against it, its number of edges decreases. Consider Algorithm 1. `G` represents graph G as a dictionary, such that `g[i]` is the set of nodes that voted against `i`. For simplicity we do not show any set to store the nodes we eliminate in the process. The complexity of this algorithm is $O(m+n \log n)$, for m edges and n nodes. The $n \log n$ part is the sorting cost.

Algorithm 1 The Against Votes Algorithm 1

```
totalvotesagainst = sum of all set sizes in g
sort g in decreasing order of set sizes
for all nodes of g in the decreasing order do
  avg = totalvotesagainst/nbr. of nodes in g
  if votes against of node > avg then
    totalvotesagainst − = 2× set size of the node {Edges are
    bidirectional}
    Remove node from g
  end if
end for
```

6.7. Against Votes 2 (AV2)

In a whitewashing attack [11], malicious nodes may trivially change their identifiers. None of the previous algorithms can cope with this situation. In this variant of the Against Votes algorithm, we consider the votes against graph, G, but instead of using absolute numbers of votes, we make this number relative to the participation of the node in voting pools. More precisely, we consider the ratio between the number of nodes `i` votes against and the number of participations of `i` in voting pools. In this way, a node that sabotages only once, but participates in a small number of pools has a large ratio. We call this the "Against Votes 2" (AV2) and show it in Algorithm 2. Recomputation of the variable `sumratios`, which replaces the `totalvotesagainst` in Algorithm 1 is slightly different. Fortunately, this involves only the edges of `i`: for each edge of `i`, we subtract 1/participations of `i` and subtract 1/participations of the node in the other end of the edge.

Algorithm 2 The Against Votes Algorithm 2 (AV2)

```
sumratios = sum for all nodes i of votes against i/participations
of i
sort g in decreasing order of set sizes
for all nodes of g in the decreasing order do
  voteratio = votes against i/participations of i
  avg = sumratios/nbr. of nodes in g
  if voteratio > avg then
    Recompute sumratios {Only involves edges with i}
    Remove node from g
  end if
end for
```

7. Experiments

7.1. Experimental settings

In our experiments, we randomly assign 3 different nodes to each voting pool. All nodes have the same probability of participating in a given voting pool. Assignment of nodes to one voting pool is independent from the remaining voting pools. Then, we went through the following steps: (i) simulate a variable number of voting pools and set their decision according to the majority of results (this can be a no-decision if no majority exists), (ii) use one of the heuristics described before to classify nodes as correct or incorrect; (iii) based on the previous division, we go again through each 3-node voting pool, either *accepting* it, if at least one (considered to be) correct node supported the decision, or *discarding* it otherwise. If we decide to accept a voting pool with an incorrect decision, we consider it as a *false positive*. If we discard a voting pool with a correct decision, we count a *false negative*. Note that in the simulation we can always tell whether the final decision of a voting pool was correct or not.

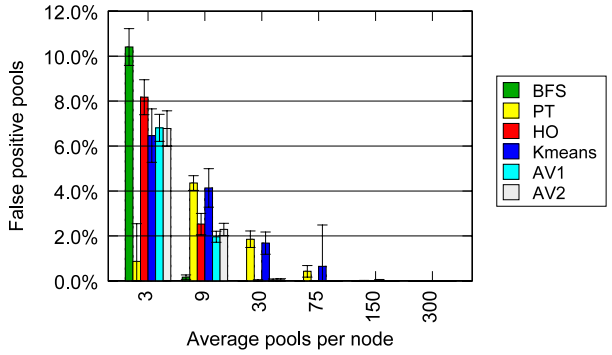


Fig. 4. Percentage of false positive pools for 20% of M_2.

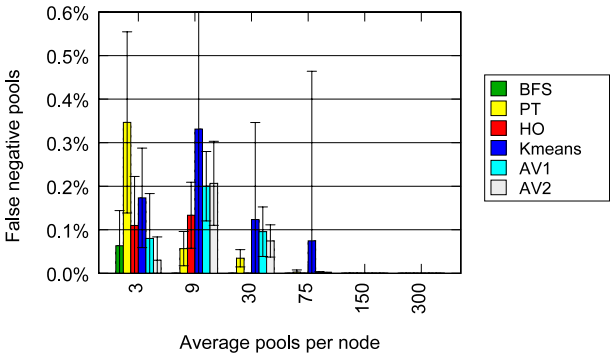


Fig. 5. Percentage of false negative pools for 20% of M_2.

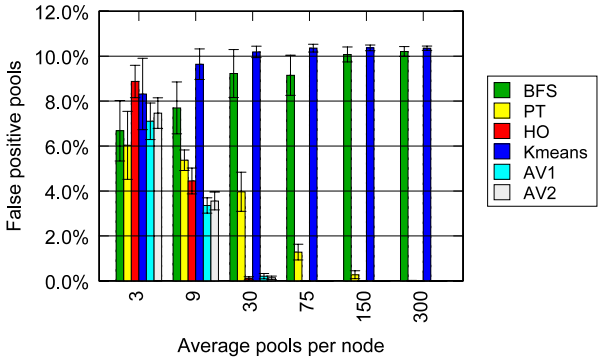


Fig. 6. False positives for 20% of M_1 and 20% of M_2.

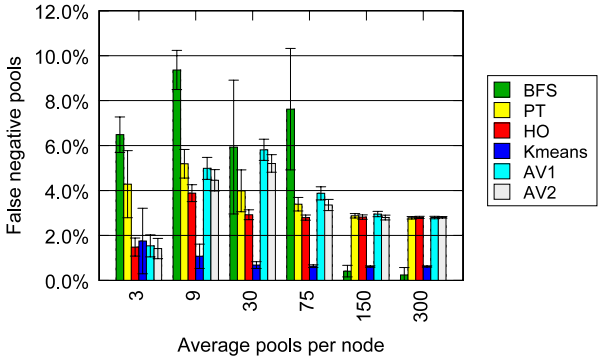


Fig. 7. False negatives for 20% of M_1 and 20% of M_2.

To simulate the voting pools (step *i*), we used a Java program. The program receives as arguments the number of nodes of each type that should exist, e.g., 20% of M_2 nodes. To generate the voting pools and the votes themselves we used the Java Mersenne Twister implementation in the Colt Distribution [7]. In our simulations M_1-type nodes produce a faulty result with 30% probability. After deciding for the collusion, M_4-type nodes keep their sabotage 50% of the time. In the remaining 50% they betray their peer colluders. We used 1000 different nodes, except when stated otherwise. Our plots show averages of 30 independent runs. Error bars show standard deviations.

7.2. Results

M_2-type workers. We start with scenarios that only contain correct workers and M_2-type colluders. We first used 5% of colluder nodes, and then raised this number to 20%. We only show results for the latter, as it is a much greater challenge for algorithms, which invariably have worse results (for instance, more false positives and/or false negatives). Figs. 4 and 5, respectively show false positives and false negatives for the different algorithms. For the *x*-axis of the plots, we use the average number of pools where each node participated. For example, average 30 pools per node, for 1000 nodes, means an overall $1000/3 \times 30 = 10{,}000$ voting pools. We use proportions in the false negatives and positives to enable comparisons for all numbers of voting pools. The plots reveal that all algorithms have nearly no false negatives, but many false positives. This means that they classify many malicious nodes as correct, especially when the number of times these nodes vote is very small.

Most algorithms show fair or poor results for a small number of pools, but improve quickly as this number grows. The case of the Perfect Threshold algorithm is interesting. For only a few votes against, when the number of pools is small, it manages to identify all malicious nodes (unfortunately including the correct nodes that voted against them in a group). This will always invalidate voting pools with incorrect results. As the number of pools increases this trivial identification becomes more difficult. In fact, despite using *a priori* (unfair) knowledge of which nodes are correct and which are malicious, the Perfect Threshold algorithm is not the best. Standard deviation in the false negatives is quite large in some cases, especially in the *K*-means algorithm, not only in this scenario, but also in settings we show ahead with other colluders (e.g., M_5). This results from misclassification of correct nodes, which is usually low, but has a few outliers causing a much larger number of false negatives. For example in the 30 runs of the 6 different numbers of pools per node, *K*-means resulted in more than 2% false negatives 3 times (in 180).

One word about the BFS method, which shows poor results when the number of pools per node is very small. In fact, since BFS misclassifies nearly all M_2 malicious nodes, the false positives will depend on simple majority-based decisions. Hence, we can compare this to the prediction of Eq. (1): $\epsilon(0.2, 3, 1000) \approxeq 10.37\%$. This is close to the average number of false positive pools we obtained (104 in 1000 pools). As we mentioned in Section 6.2, this results from a disconnected graph *G*, where malicious nodes have small majorities (e.g., 2-against-1). Another observation is that the Higher-Order yields very good results. This is a supporting evidence for the case that a connection exists between the MIS problem and collusion detection. Unfortunately, MIS algorithms are not very effective when the available number of contradictory votes is very small. The reason is easy to explain: colluders successfully act independently from each other only once (thus they manage to define small winning pluralities, just like in the BFS failures).

M_1 plus M_2-type workers. An experiment of Kondo et al. [20], measured the number of nodes that send bad results at least once around 35%. In Figs. 6 and 7, we depict the experiments for a mixture of malicious nodes near this figure: 20% are M_1 and 20% are M_2 nodes. If we compare Fig. 6 to Fig. 4, we can observe

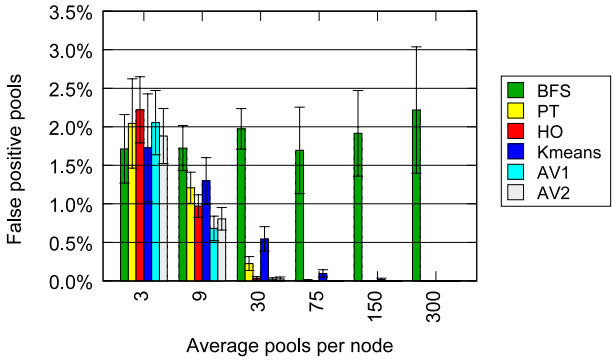


Fig. 8. False positives for 20% of M_4-type workers.

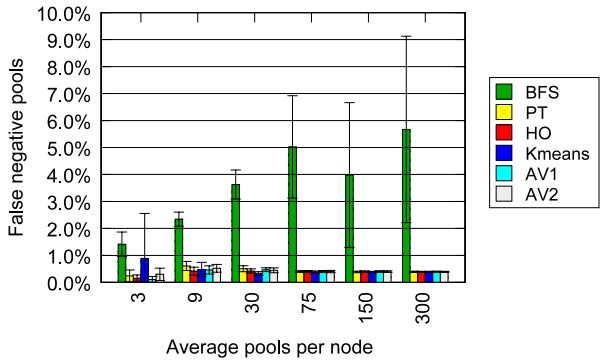


Fig. 9. False negatives for 20% of M_4-type workers.

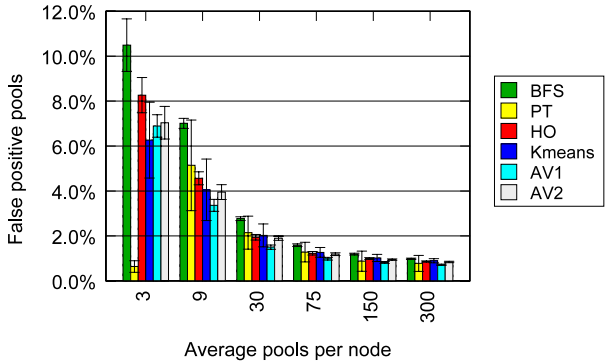


Fig. 10. False positives for 20% of M_5-type workers.

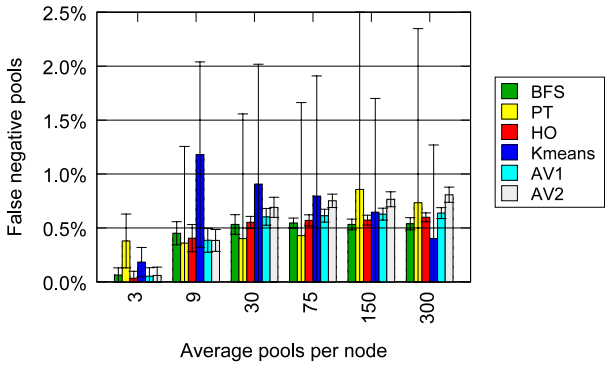


Fig. 11. False negatives for 20% of M_5-type workers.

that results for K-means and BFS degrade much more than for remaining algorithms. Moreover, false negatives do not converge to 0% in most algorithms. This happens because collusion detection algorithms classify many correct nodes as incorrect, due to the large number of votes against they have. As a result, even when correct nodes do the same vote as incorrect nodes in the same pool, the pool is discarded.

The problem with K-means has to do with the argument it receives specifying the number of clusters it should form. In this case, we have three kinds of workers, namely correct, naïve, and colluding. As we cannot anticipate the number of clusters, we form only two (correct and incorrect). Consequently, classification of K-means deteriorates. On the other hand, BFS starts to fail, because we no longer have the ideal pattern of Fig. 3, where correct and malicious nodes stay in alternate layers. M_1 nodes voting according to a random pattern, can also vote against M_2-type nodes, thus disrupting BFS. Regarding false negatives, BFS initially shows large values, which decrease with the number of pools per node. This happens, because BFS classifies all nodes as correct. In this way, it will accept all pools even bad ones. The same is true for K-means. As it classifies almost all nodes as correct, no correct pools are discarded (and conversely, many false positives exist).

M_4-type workers. We also tried using 20% of unreliable malicious nodes, as we show in Figs. 8 and 9. Since only two types of workers exist, once again, the K-means heuristic achieves reasonable results, slightly worse than MIS and perfect threshold algorithms. Again, BFS is strongly affected by malicious nodes voting against each other. For most algorithms, the overall number of false positives is smaller than in previous settings, because there are also fewer successful sabotages, due to the very nature of unreliable, M_4-type colluders. The false negatives plot is especially high for the BFS heuristics. Since the false positive number is also quite high, this means that this heuristic erroneously swaps the classification of many nodes.

M_5-type workers. Figs. 10 and 11 show results in a setting with 20% of M_5-type colluders. These results are quite interesting as they show that all the algorithms we tried are pretty much powerless against this type of collusion. A possible way of deterring the action of M_5-type colluders would be to make the size of the voting pools unknown. If colluders know this size, their most important limitation comes from their own number. For instance, as we have 200 colluders, they sabotage in less than 200 occasions, as we mentioned in Section 4, plus all the times they are alone in a voting pool. From the Hypergeometric distribution, this latter number gives approximately 0.79% of the times, which we can see as the limit in the plot of Fig. 10. To better understand this, note that we start with 1000 pools (3 pools per node) and end with 100,000 (300 pools per node). In the former case, (less than) 200 possible collusions add a lot to the 0.79% of the pools, while in the latter case, the extra 200 collusions have a negligible effect. The number of false negatives is always relatively small and results from correct nodes that algorithms take as incorrect, as a consequence of their votes against the majorities of colluder nodes. The behavior of HO, AV1 and AV2 is similar in these plots.

7.3. Evaluation of the algorithms

When we look at the overall results, several algorithms show good performance: Perfect Threshold, Against Votes, and Higher-Order. K-means is slightly worse when only two classes of nodes exist and very bad otherwise. Other clustering algorithms that do not depend on *a priori* knowledge of the number of clusters might be a possibility here [23]. BFS depends on the absence of against votes among malicious nodes. Finally, we should restate that the Perfect Threshold algorithm uses information that is not available in practice: it needs to be aware of which nodes are malicious to make a decision.

To distinguish the best algorithms, we ran a larger set of tests with 10 000, 20 000 and 30 000 nodes. This is more appropriate for

Table 2
Avg. 30 pools per nodes for 20% M_1 plus 20% M_2-type nodes.

Nodes	HO	AV1	AV2
False positives (%)			
10 000	0.13 ± 0.03	0.23 ± 0.04	0.16 ± 0.03
20 000	0.13 ± 0.02	0.23 ± 0.02	0.16 ± 0.02
30 000	0.13 ± 0.01	0.23 ± 0.02	0.16 ± 0.02
False negatives (%)			
10 000	2.95 ± 0.06	6.04 ± 0.17	5.12 ± 0.14
20 000	2.98 ± 0.06	5.99 ± 0.10	5.14 ± 0.12
30 000	2.98 ± 0.04	6.01 ± 0.08	5.10 ± 0.06
Time (s)			
10 000	70 ± 4.06	2.13 ± 0.16	1.94 ± 0.19
20 000	338 ± 18.31	5.55 ± 0.52	3.69 ± 0.18
30 000	900 ± 35.98	20.63 ± 1.10	5.67 ± 0.30

relatively small BOINC projects. Based on our previous plots, we used 30 pools per node, as this is often the threshold between poor and good results for some algorithms. We also added execution time of the algorithms to the evaluation. We implemented Higher-Order and Against Votes in Python. In Table 2, we show the percentage of false positives, false negatives, and the execution time in seconds on a computer with a 2.4 GHz Intel Core i5 CPU. We show results, in the form average $\pm$ standard deviation, for 30 runs. Up to 30,000 nodes, results of these three algorithms are pretty similar to what we get with only 1000 nodes. To see this, compare the number of false positives and negatives to those of Figs. 6 and 7. For this average number of pools per node, the Higher-Order shows the best number of false negative pools. On the other hand, its execution time grows significantly for larger sets of voter nodes. This might be a problem for very large projects. Another disadvantage of Higher-Order is that it requires a configuration parameter and [6] makes no recommendation on how to set it. In summary, if enough time is available, and for scenarios where nodes have nearly equal participation in the voting pools, the Higher-Order algorithm is better than the Against Votes approach.

7.4. Gain of the colluders

We assume that colluders would like to sabotage as much as possible, such that any result they produce is wrong. A good scenario for colluders occurs if the central supervisor swaps their classification against a number of correct nodes. In this case, the central supervisor may end up discarding good computation (many false negatives) for bad computation (many false positives).

On the other hand, if the central supervisor manages to refrain colluders from sabotaging, they may actually perform useful work. To assess this, we define a ratio between votes the central supervisor discards and colluder votes. We call this ratio "gain of the colluders". This assumes that each vote amounts to approximately the same quantity of work. Note that each time the central supervisor discards a p-node voting pool, it immediately discards p votes. Whenever this ratio is below 1, colluders are actual contributors to the community, for example to the computation effort. For the 30 000 node experiment, the ratio between votes discarded and colluder votes ranges between 33% and 40%, for the Higher-Order, AV1 and AV2 algorithms. Note also that the number of false positives is very small. We omit further results as they are consistently good for most algorithms, but this ratio can be even lower in many other settings. For instance, M_4 nodes usually achieve very low gains in the range 5%–20%. So we can conclude that colluders are actual contributors, as most of their results are valid.

Despite these interesting results, colluders may find their effort worthwhile if they manage to sabotage a few crucial results (i.e., they may accept low gains). Whether or not they find their real gain in the results they sabotage (the false positives), is something that depends on the scenario and that we cannot assess here.

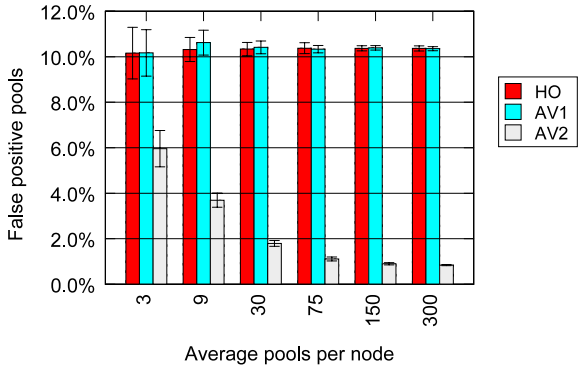


Fig. 12. Percentage of false positive pools in the whitewashing attack.

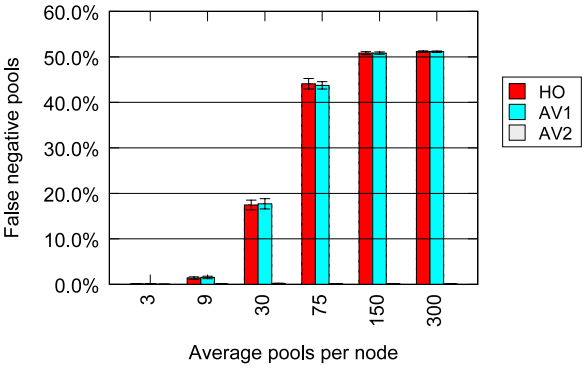


Fig. 13. Percentage of false negative pools in the whitewashing attack.

7.5. Further improvements

Some of the collusion detection algorithms we propose achieve a small number of false positives, in the range 0.1%–0.2%. Nevertheless, we recognize that it is very important to decrease this number even further. For this, we may complement our own collusion detection algorithms with other mechanisms. Although we do not explicitly explore this possibility here, one of the simplest possibilities could be to use some *auditing* mechanism. For instance, in pools with nodes that are likely to be incorrect, the central supervisor may use a few more worker nodes for verification purposes. This trades more redundancy for fewer false positives. We believe that this raises the interesting research question of determining the overhead necessary to achieve a given number of false positives under the presence of colluders, pretty much as Sarmenta did in his seminal work [28], for simple majority voting.

7.6. The whitewashing attack: anonymous malicious nodes

When nodes have fixed identifiers, a few collusion detection algorithms are quite effective. Unfortunately, enforcing this property may be impossible in most real settings. In this case, the same malicious nodes can change identifiers over time, thus distorting their real proportion in the votes against graph. This is the whitewashing attack [11]. The problem for MIS-based heuristics or for AV1 lies in the fact that correct nodes can become a minority, thus undermining the basic assumptions of Theorems 5.1 and 5.2. Algorithm AV2 tackles precisely this case, by penalizing nodes with a small number of participations. We depict the number of false positives and negatives in Figs. 12 and 13, respectively. There are 20% of malicious M_2 nodes that change

identifiers after each sabotage (attack). Results are quite bad in general, with the exception of the AV2 algorithm. Unlike most previous cases, the gain of colluders is quite large in these cases. For an average of 300 pools per node, we achieve a gain around 2.5 for HO and AV1, a clear indication that malicious nodes often pass undetected and hold someone else accountable. AV2 achieves a better result of 0.5. In the future, it might be interesting to evaluate scenarios where even the set of correct nodes changes quickly.

8. Conclusions

In this paper, we evaluated mechanisms to detect colluding nodes in repeating voting pools. We proposed several algorithms to detect collusion in these voting pools, namely breadth-first search (BFS), against votes (AV1) and heuristics based on the maximum independent set (MIS). We showed the strong and weak points of these, and a few more algorithms; AV1 and MIS-based algorithms perform very well in settings where nodes have similar rates of participation and cannot present multiple identifiers to the central supervisor.

However, these algorithms are completely powerless if nodes can change their identifiers to perform whitewashing attacks. This motivated us to develop a variant of AV1, called AV2. The latter considers the fraction of votes against over the number of times of participation, instead of simply counting the votes against. AV2 shows promising results, for the relatively low computational cost of $O(m + n \log n)$, where m is the number of edges in the votes against graph and n is the number of different existing identifiers.

Based on the theoretical results and on the simulations, we can conclude that graph-based approaches, and, in particular, heuristics for the Maximum Independent Set problem, can effectively detect (and thus deter) attacks for the types of colluder nodes we evaluated.

Acknowledgments

This work has been partially supported by the project PTDC/EIA-EIA/102212/2008, High-Performance Computing over the Large-Scale Internet. G.C. Silaghi acknowledges support from the Romanian Authority for Scientific Research under project IDEI 2452.

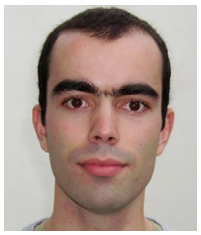
The authors are also grateful to the anonymous reviewers.

References

[1] G. Adomavicius, A. Tuzhilin, Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions, IEEE Transactions on Knowledge and Data Engineering 17 (2005) 734–749.
[2] 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, IEEE Computer Society, 2004, pp. 4–10.
[3] Y. Bartal, R. Gonen, N. Nisan, Incentive compatible multi unit combinatorial auctions, in: TARK'03: Proceedings of the 9th Conference on Theoretical Aspects of Rationality and Knowledge, ACM, New York, NY, USA, 2003, pp. 72–87.
[4] P. Bremaud, Markov Chains: Gibbs Fields, Monte Carlo Simulation, and Queues, corrected ed., Springer-Verlag New York Inc., 2001.
[5] L.-C. Canon, E. Jeannot, J. Weissman, A dynamic approach for characterizing collusion in desktop grids, in: IEEE International Parallel and Distributed Processing Symposium, IPDPS 2010.
[6] I. Chang, W.-Z. Shao, H.-H. Teh, Heuristic solutions for the general maximum independent set problem with applications to expert system design, in: 12th International Computer Software and Applications Conference, COMPSAC 88, Chicago, IL, USA, 1988, pp. 451–455.
[7] Colt—Welcome, 2004. http://acs.lbl.gov/software/colt/.
[8] P. Domingues, B. Sousa, L. Moura Silva, Sabotage-tolerance and trust management in desktop grid computing, Future Generation Computer Systems 23 (2007) 904–912.
[9] W. Du, J. Jia, M. Mangal, M. Murugesan, Uncheatable grid computing, in: ICDCS'04: Proceedings of the 24th International Conference on Distributed Computing Systems, ICDCS'04, IEEE Computer Society, 2004, pp. 4–11.
[10] P. Faliszewski, E. Elkind, M. Wooldridge, Boolean combinations of weighted voting games, in: AAMAS'09: Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 2009, pp. 185–192.
[11] M. Feldman, C. Papadimitriou, J. Chuang, I. Stoica, Free-riding and whitewashing in peer-to-peer systems, in: PINS'04: Proceedings of the ACM SIGCOMM Workshop on Practice and Theory of Incentives in Networked Systems, ACM, New York, NY, USA, 2004, pp. 228–236.
[12] A. Fernandez, L. Lopez, A. Santos, C. Georgiou, Reliably executing tasks in the presence of untrusted entities, in: SRDS'06: Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems, IEEE Computer Society, Washington, DC, USA, 2006, pp. 39–50.
[13] M.J. Fischer, N.A. Lynch, M.S. Paterson, Impossibility of distributed consensus with one faulty process, Journal of the ACM, JACM 32 (1985) 374–382.
[14] M. Garey, D. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman, 1979.
[15] D.K. Gifford, Weighted voting for replicated data, in: SOSP'79: Proceedings of the Seventh ACM Symposium on Operating Systems Principles, ACM, New York, NY, USA, 1979, pp. 150–162.
[16] P. Golle, I. Mironov, Uncheatable distributed computations, in: D. Naccache (Ed.), CT-RSA 2001: Procs. of the 2001 Conf. on Topics in Cryptology, in: Lecture Notes in Computer Science, vol. 2020, Springer, Berlin, Heidelberg, 2001, pp. 425–440.
[17] J.L. Gross, J. Yellen (Eds.), Handbook of Graph Theory, CRC Press, 2004.
[18] P. Harrenstein, W. van der Hoek, J.-J. Meyer, C. Witteveen, Boolean games, in: TARK'01: Proceedings of the 8th Conference on Theoretical Aspects of Rationality and Knowledge, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001, pp. 287–298.
[19] N.L. Johnson, S. Kotz, Urn Models and their Application: An Approach to Modern Discrete Probability Theory, Wiley, New York, 1977.
[20] D. Kondo, F. Araujo, P. Malecot, P. Domingues, L.M. Silva, G. Fedak, F. Cappello, Characterizing result errors in internet desktop grids, in: Euro-Par 2007: Proceedings of 13th International Euro-Par Conference, Rennes, France, August 28–31, 2007, in: Lecture Notes in Computer Science, vol. 4641, Springer, 2007, pp. 361–371.
[21] L. Lamport, R. Shostak, M. Pease, The byzantine generals problem, ACM Transactions on Programming Languages and Systems 4 (1982) 382–401.
[22] G. Latif-Shabgahi, J. Bass, S. Bennett, A taxonomy for software voting algorithms used in safety–critical systems, IEEE Transactions on Reliability 53 (2004) 319–328.
[23] U. Luxburg, A tutorial on spectral clustering, Statistics and Computing 17 (2007) 395–416.
[24] J.B. Macqueen, Some methods of classification and analysis of multivariate observations, in: Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability, vol. 1, pp. 281–297.
[25] D. Milojicic, A. Messer, J. Shau, G. Fu, A. Munoz, Increasing relevance of memory hardware errors: a case for recoverable programming models, in: EW 9: Proceedings of the 9th Workshop on ACM SIGOPS European Workshop, ACM, New York, NY, USA, 2000, pp. 97–102.
[26] M. Mito, S. Fujita, On heuristics for solving winner determination problem in combinatorial auctions, Journal of Heuristics 10 (2004) 507–523.
[27] M.G.C. Resende, T.A. Feo, S.H. Smith, Algorithm 787: Fortran subroutines for approximate solution of maximum independent set problems using grasp, ACM Transactions on Mathematical Software 24 (1998) 386–394.
[28] L.F.G. Sarmenta, Sabotage-tolerance mechanisms for volunteer computing systems, Future Generation Computer Systems 18 (2002) 561–572.
[29] J.B. Schafer, J. Konstan, J. Riedi, Recommender systems in e-commerce, in: EC'99: Proceedings of the 1st ACM Conference on Electronic Commerce, ACM, New York, NY, USA, 1999, pp. 158–166.
[30] G.C. Silaghi, F. Araujo, L. Silva, P. Domingues, A. Arenas, Defeating colluding nodes in desktop grid computing platforms, Journal of Grid Computing 7 (2009) 555–573.
[31] P. Sousa, A.N. Bessani, M. Correia, N.F. Neves, P. Verissimo, Highly available intrusion-tolerant services with proactive-reactive recovery, IEEE Transactions on Parallel and Distributed Systems 21 (2010) 452–465.
[32] E. Staab, T. Engel, Collusion detection for grid computing, in: CCGRID'09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, IEEE Computer Society, Washington, DC, USA, 2009, pp. 412–419.
[33] V. Subramanian, P.K. Ramesh, A.K. Somani, Managing the impact of on-chip temperature on the lifetime reliability of reliably overclocked systems, in: DEPEND'09: Proceedings of the 2009 Second International Conference on Dependability, IEEE Computer Society, Washington, DC, USA, 2009, pp. 156–161.
[34] M. Taufer, D. Anderson, P. Cicotti, C.L. Brooks III, Homogeneous redundancy: a technique to ensure integrity of molecular simulation results using public computing, IPDPS, in: 19th International Parallel and Distributed Processing Symposium, vol. 2, IEEE Computer Society, 2005, pp. 119–127.
[35] M. Tsvetovat, K. Sycara, Y. Chen, J. Ying, Customer coalitions in the electronic marketplace, in: Seventeenth National Conference on Artificial Intelligence, AAAI-2001, MIT Press, 2000, pp. 1133–1134. 17th National Conference on Artificial Intelligence (AAAI-2000)/12th Conference on Innovative Applications of Artificial Intelligence (IAAI-2000), AUSTIN, TX, JUL 30-AUG 03, 2000.
[36] K. Watanabe, M. Fukushi, S. Horiguchi, Optimal spot-checking for computation time minimization in volunteer computing, Journal of Grid Computing 7 (2009) 575–600.

[37] S. Wong, An authentication protocol in web-computing, in: 20th International Parallel and Distributed Processing Symposium, IPDPS 2006, Proceedings, 25–29 April 2006, Rhodes Island, Greece, IEEE Computer Society, 2006.
[38] M. Yokoo, V. Conitzer, T. Sandholm, N. Ohta, A. Iwasaki, Coalitional games in open anonymous environments, in: AAAI'05: Proceedings of the 20th National Conference on Artificial intelligence, AAAI Press, 2005, pp. 509–514.
[39] M. Yokoo, Y. Sakurai, S. Matsubara, The effect of false-name bids in combinatorial auctions: new fraud in internet auctions, Games and Economic Behavior 46 (2004) 174–188.
[40] S. Zhao, V. Lo, C. GauthierDickey, Result verification and trust-based scheduling in peer-to-peer grids, in: P2P'05: Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing, IEEE Computer Society, 2005, pp. 31–38.

Filipe Araujo is an Assistant Professor at the University of Coimbra. He received his graduation in Electrical Engineering in 1996 and his M.Sc. in Informatics Engineering in 1999, both from the University of Coimbra. He received his Ph.D. in 2006 from the University of Lisboa. He joined the University of Coimbra in 2006. He participated in several national and international projects. His current research interests are focused on Grid and parallel computing, peer-to-peer systems, mobile computing and security.



Jorge Farinha is a computer science student at the University of Coimbra. He participated in an international project—EDGeS (Enabling Desktop Grids for e-Science). His research interests are distributed/parallel computing and ubiquitous systems.

Patricio Domingues holds an M.Sc. and a Ph.D. in Informatics Engineering from the University of Coimbra, Portugal. He is currently a lecturer of the Informatics Engineering Department at the School of Technology and Management at the Polytechnic Institute of Leiria, also in Portugal. His main research interests include desktop grids, peer-to-peer systems and high-performance computing, especially many-core computing.

Gheorghe Cosmin Silaghi is an Associate Professor at the Babes-Bolyai University of Cluj-Napoca, Romania. He received a Bachelor degree in Business Information Systems in 2000 and his Engineering degree in Computer Science in 2002. In 2002, he received his M.Sc. in Artificial Intelligence from Free University of Amsterdam. G.C. Silaghi completed his Ph.D. in 2005. He joined the Babes-Bolyai University in 2000 and currently he is the Head of the Business Information Systems department. His current research interests are focused on resource management techniques in untrusted distributed environments, like peer-to-peer systems.

Derrick Kondo is a research scientist at INRIA Rhone-Alpes Grenoble. He received his Bachelor's in Computer Science at Stanford University, and his Master's and Ph.D. in Computer Science from the University of California at San Diego. He co-founded the Failure Trace Archive. He founded and serves as Chair/Co-Chair of the Workshop on Volunteer Computing and Desktop Grids (PCGrid). He was guest editor of a special issue of the Journal of Grid Computing on desktop grids. His research interests include the characterization, modeling, and simulation of large-scale distributed computing systems, and scheduling mechanisms and algorithms for volatile resources.