SERENE: A Collusion Resilient Replication-based Verification Framework

Amir Esmaeili
Department of Computer Science
University of Missouri St. Louis
St. Louis, USA
ae3wc@umsl.edu

Abderrahmen Mtibaa
Department of Computer Science
University of Missouri St. Louis
St. Louis, USA
amtibaa@umsl.edu

2024 IEEE 13th International Conference on Cloud Networking (CloudNet) | 979-8-3503-7656-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/CLOUDNET62863.2024.10815750

Abstract—**The rapid advancement of autonomous driving technology is accompanied by substantial challenges, particularly the reliance on remote task execution without ensuring a reliable and accurate returned results. This reliance on external compute servers, which may be malicious or rogue, represents a major security threat. While researchers have been exploring replication-based task verification as a simple, fast, and dependable *verifiable computing* method to assess the correctness of results, colluding malicious workers can easily defeat this method. Existing collusion detection and mitigation solutions often require the use of a trusted third party server or verified tasks which may be hard to guarantee, or solutions that assume the presence of a minority of colluding servers. We propose *SERENE*, a collusion resilient replication-based verification framework that detects, and mitigates colluding workers. Unlike state-of-the-art solutions, *SERENE* uses a lightweight detection algorithm that detects collusion based on a small set of verification tasks. Mitigation requires a two stage process to group the workers and identifying colluding from honest workers. We implement and compare *SERENE*'s performance to *Staab et. al*, resulting in an average of 50% and 60% accuracy improvement in detection and mitigation accuracy respectively.**

Index Terms—**Constrained Devices, Replication-based Task Verification, IoT Remote Task Execution, Worker Collusion Detection, Collusion Mitigation.**

I. Introduction

Autonomous driving systems are transforming the automotive industry, setting the stage for safer and more efficient transportation. These systems generate large volumes of heterogeneous sensory data, requiring substantial computational resources and real-time processing capabilities. Both onboard and remote processing power are crucial to handle these increasing demands. Tasks like real-time image processing for self-driving navigation are particularly critical and must be highly accurate. Verifiable computing [1], which ensures the correctness of results returned by onboard or remote servers, becomes essential, especially for such safety-critical tasks.

Verifiable computing solutions fall into one the following three main categories: (i) attaching probabilistically checkable proofs to each offloaded task to identify incorrect results with high probability [2], (ii) using Trusted Execution Environments (TEEs), such as Intel Software Guard executions (SGX) to ensure the integrity of computation execution and results [3], and (iii) redundantly requesting the execution of tasks from multiple external servers, and applying majority voting to find the correct result [4]. While proof-based, and TEE-based solutions are limited to some specific applications, and hardware dependability, replication-based methods are generic, easy to implement, and effective [5]. However, replication-based task verification is prone to a main security attack where two or more colluding workers submit the same incorrect results, and defeat the majority voting scheme of the replication-based mechanism (*i.e.,* collusion attack) [6].

Most collusion resilient replication-based solutions rely on: (1) enlarging the voting pool [7], (2) spot checking using a set of pre-defined trusted tasks [8], or trusted third party server to re-execute the task [6], and (3) incentivizing rational servers to betray collusion [9], to decrease the chance of collusion attack. While most of these existing solutions rely on trusted third parties or may prevent but not protect against collusion, recently similarity-based clustering solutions have emerged to probe the workers and identify clusters of workers to infer *colluding* versus *honest* workers [4], [6], [10]. However, these solutions fail to detect and mitigate collusion when colluding nodes represent the majority in the network. To the best of our knowledge, this problem remains unexplored.

We propose *SERENE*, a Collusion Resilient Replication-based Verification Framework. *SERENE* is implemented on top of any task replication-based framework to continuously monitor the list of workers and detect the presence of collusion, triggering a mitigation process to identify and isolate colluding workers in the network. *SERENE*'s detection relies on identifying two clusters of workers consistently disagreeing with each others. While this identification guarantees the presence of colluding workers, without assumptions of the size of colluding workers or the presence of trusted third party servers (used by state-of-the-art solutions), *SERENE* uses a three-step mitigation algorithm to partition the group of workers and identify the colluding ones.

The three main contribution of our paper are summarized as follows:

- We propose *SERENE* to detect and mitigate collusion attacks. *SERENE* can accurately identify and isolate *colluding* nodes even when they represent 90% of the worker population, without relying on any trusted servers or pre-checked tasks.

- *SERENE* decouples detection, and mitigation phases to run the mitigation approach once the presence of colluding workers in the network is detected. While *SERENE*'s detection algorithm is periodic and consciously monitor the workers behavior, it is designed to be lightweight and does not require costly lookups.
- We evaluate the *SERENE* performance with state-of-art Staab & Angel [11], which we refer to as SnE. Results show that mitigation accuracy of *SERENE* is more than double that of SnE. Furthermore, *SERENE* detects collusion 15% faster than SnE and 30% to 60% more accurate detection. Finally, we perform a set of benchmarking tests to assess the run time of *SERENE* and show that it performs faster than SnE in three tested platform while incurring slightly more resource utilization.

The remainder of this paper is organized as follows: Section II presents the literature review of verifiable computing solutions and related works. In the Section III, we briefly describe our system model, threat model, and the assumptions we used in this paper. *SERENE*'s detection and mitigation algorithms are discussed in Section IV. We evaluate *SERENE*'s performance and present our simulation results in Section V. Conclusion and future work are provided in Section VI.

II. Related Work

Verifying the correctness of remote workers execution has been recently investigated, and the proposed research solutions can be categorized into three main categories: (i) *Proof-based* approaches using probabilistically checkable proofs (PCPs) produced by workers, and attached to the results [2], [12], [13], (ii) *TEE-based* solutions that at the hardware level ensures code and data integrity for task offloaders. The Intel SGX [14], and ARM TrustZone [15] are two main TEE-based technologies, and (iii) *replication-based* methods that are very generic. In this approach, clients assign tasks to multiple workers, and final results are specified by a quorum (*e.g.,* majority voting) [16], [7], [8], [17], [9], [18].

While replication-based solutions are straightforward and easy to implement, they are still susceptible to colluding workers who can manipulate majority voting outcomes. Most research on collusion detection has focused on replication-based methods, with relatively few studies addressing collusion in TEE-based and proof-based solutions. In proof-based systems, collusion risks arise primarily from delegating verification or proof setup to a compromised third party [19]. In TEE-based solutions, the main vulnerability comes from rogue remote attestation, where a malicious entity can bypass or falsify attestation protocols [20].

However, replication-based collusion defense mechanisms are mainly divided into two different areas: Prevention [16], [7], [8], [17], [9], [18], or detection and mitigation [6], [11], [10], [21], [22], [23]. While in prevention solutions the main target is to incentivize colluding workers to betray the collusion [9] (by giving higher rewards in a game), or enlarging the voting pool to decrease the probability of winning of colluding servers in majority voting [7], nevertheless, the main weakness of prevention is it works like a vaccine for diseases and it can not guarantee to prevent from collusion.

On the other hand, detection, and mitigation is an approach to identify the colluding workers, and makes them isolated from the workers population. Silaghi et al. [6] used two-step algorithms to identify a majority pool of servers which will be considered benign and used for detecting colluding servers, while Staab et al. [11] applied a one-step graph clustering algorithm to identify both benign and colluding servers. Both of these approaches need time to complete the set of tasks before applying the detection algorithm, also percentage of colluding servers should be less than 50%. Moreover, algorithms to cut the worker's graph has additional overhead.

Zhao et al. [8], proposed spot checking solution to evaluate workers by sending tasks with already known results, to identify colluding servers. Spot checking tasks should be unpredictable, and it is hard to find these tasks. Some of solutions in this area assume a trust third party [24] for result verification, and some of them suppose there are multiple pre-arranged spot tasks [5]. Both assumptions are rarely feasible.

Unlike most of these proposed solutions, *SERENE* does not use trusted parties (*e.g.,* workers), or trusted pre-approved tasks, and can accurately detect and mitigate collusion even when the percentage of colluding workers exceeds 50% of the workers in the network.

III. System Model, Threat Model, and Assumption

A. System Model

We consider an untrustworthy edge computing network consisting of N workers nodes[1], $S_1 \ldots S_N$ (*e.g.,* edge computing servers) distributed across an area where clients can choose one or many workers at a time to outsource their computation tasks. We assume that clients perform replication-based verification by selecting a voting pool consisting of k random workers[2]. Users send tasks to a voting pool at any given time, and collect the majority vote to ensure task execution correctness.

B. Threat Model

The system threat model includes three types of workers which submit results to a given offloaded task:

- *honest workers:* A worker is called honest if it executes and returns correct results to all given task. We assume that honest workers can, rarely, return incorrect results with probability ϵ, due to hardware failure, or incompatibility. The list of honest workers is denoted by $\mathcal{H}$.
- *naive malicious workers:* This type of worker is malicious and always submits a random incorrect result (*e.g.,* do not execute the task and instead return any random result to save energy, or resources) for a given task. These workers work independently without any coordination with other

[1]In this paper, we use workers, servers, and edge nodes interchangeably
[2]Without loss of generality, we use $k = 3$ throughout the paper

malicious workers. The list of naive malicious workers is denoted by $\mathcal{M}$.

- *colluding workers:* Colluding workers perform sophisticated attacks; they collude only if they constitute the majority of the verification pool. In addition, they coordinate and decide to collude randomly to avoid verification detection. We denote the list of colluding workers by $\mathcal{C}$. A colluding worker $w \in \mathcal{C}$ colludes if and only if (1) it ensure there exists enough workers in the pool to form a majority, and (2) all colluding workers in the pool decide to collude with a fixed probability P_c. We also assume that colluding workers follow an *evasive* strategy, consisting of storing a list of previously seen tasks and if they a receive a task twice they assume it's a verification task and act as an honest worker for that task by returning the correct result.

C. Assumptions

We assume the followings:

- Clients have insufficient resources and are incapable of executing the task themselves, thus incapable of verifying the correctness of the results,
- Routers, switches, and clients are not malicious. The integrity of all messages exchanges is not compromised (*e.g.,* no man-in-the-middle attack).
- All colluding workers at the network edge implement the same strategy–they coordinate and agree on returning the same incorrect results with a *fixed* probability of collusion P_c.

IV. *SERENE*: Collusion Detection and Mitigation

Our Collusion Resilient Replication-based Verification Framework, *SERENE*, implements two main modules; (i) a module to detect colluding behavior among servers, and (ii) a collusion mitigation module.

A. SERENE*'s Detection Module*

We design *SERENE*'s collusion detection to be lightweight and fast in detecting any potential collusion of edge servers a user is communicating with.

Users store a set of collusion verification tasks (CVT) selected randomly from their genuine tasks–*i.e.,* tasks previously sent for compute verification. Once a task T_i is stored in CVT, *SERENE* tracks all results received for T_i and the corresponding server nodes returning these results.

$$CVT = \{T_1, \ldots, T_L\}, \tag{1}$$

$$T_i = [V^i, R_1^i, R_2^i, \ldots, R_N^i], \tag{2}$$

where V^i is the majority result for task T_i or $NULL$ when there is no majority recorded yet for task T_i, and R_j^i is the returned results received by server S_j when performed task T_i, or $\emptyset$ when S_j did not receive task T_i.

SERENE runs the collusion detection module periodically, every period Δt. It selects a collusion detection task $T_i \in CVT$ randomly and sends it to collusion detection pool, $C_P = \{S_j^i | R_j^i = \emptyset\}$, consisting of a set of servers that have never received the same task previously (*i.e.,* no results have been returned/recorded by servers). When *SERENE* fails to select k servers that have never received task T_i, *i.e.,* $|C_P| \leq k$, it removes T_i from CVT and replaces it with the most recent genuine task.

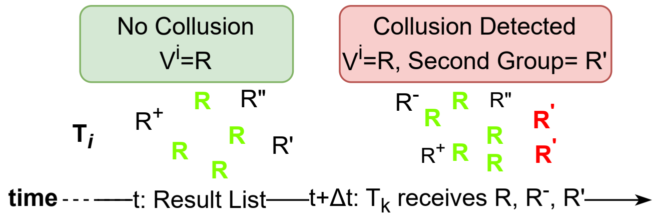


Fig. 1: In time t, there is just a group of majority ($V^i = R$), but in $t+\Delta t$, the received result R' makes the second majority group, and collusion detected.

The collusion detection algorithm runs as soon as *SERENE* receives r_j^i, a result from server $S_j^i \in C_P$. Collusion is triggered if and only if: (i) the result received does not agree with the recorded majority result, and (ii) there exist another same result returned by a different server (*i.e.,* servers agreeing on the same result which is different from the majority result), Alg. 1 Line 5. If collusion is not detected, *i.e.,Detection* function returns -1, the majority result value is updated if $V^i == NULL$ and the r_j^i is inserted into the CVT, *i.e.,* $R_j^i \leftarrow r_j^i$. However, if collusion is detected, *SERENE* wipes the CVT, and immediately initiates the mitigation module to detect and isolate colluding servers.

Figure 1 depicts an example, where a user sends a task T_i at time $t + \Delta t$, where the received results R and R'' did not trigger collusion detection, however received result R' did, because *SERENE* found another $R' \in CVT$ and $V^i = R \neq R'$, which resulted in finding two separate group of server agreeing on two different set of results, led to the detection of colluding servers.

Algorithm 1 *SERENE*'s *Detection* Function, after receiving r_j^i from server S_j

```
Require: $T_i, r_j^i$
1: if $r_j^i == V^i$ OR $V^i == NULL$ then
2:     return (-1) {No Collusion}
3: end if
4: if $ResultInCVT(r_j^i)$ AND $V^i \neq r_j^i$ then
5:     return (1) {Collusion Detected}
6: end if
7: return (-1)
```

B. SERENE*'s Mitigation Module*

As soon as collusion is detected, the mitigation module is activated to proactively probe a subset of servers in order to accurately classify them as *honest* or *colluding* workers.

SERENE's detection algorithm implies that there is at least two colluding nodes in the network, based on one verification task. The mitigation module consists of (i) clustering the nodes into two groups of servers that have similar behavior, which we refer to as *similarity-based grouping*, and (ii) identifying which group includes *honest* and which one includes *colluding* servers, we refer to this step as *identifying colluders*.

1) Similarity-based Partitioning into Two Unnamed Groups: As the detection helps indicate that there may exist two groups of servers, these groups are not exhaustive because they are formed based on a single task and a subset of probed nodes. Therefore, the mitigation module is designed to probe all the nodes in the network and construct an exhaustive undirected weighted similarity graph (SG), based on how often pairs of workers agree on the same result for the same task (*i.e.,* agree on voting outcomes).

Therefore, *SERENE* collects new votes based on current tasks until each pair of workers has been selected in 8[3] separate voting pools. Tasks are sent to only one pool of workers and the task, its k votes, and the k worker nodes returning the vote results are stored into a new task repository, $TR = \{T_i, i = 1 \dots L \mid T_i = \{(R_1^i, S_1), (R_2^i, S_2), (R_3^i, S_3)\}\}$ (assuming that the pool size is $k = 3$).

Then SG is constructed as a complete graph with N vertices, and $\frac{N \times (N-1)}{2}$ weighted edges, where weights are calculated as the ratio between how often two workers', S_i's and S_j's, votes agreed with each others by the number of times they appeared on the same voting pools: *i.e.,* the weighted edge connecting two worker S_i and S_j, $e_{i,j}$ is:

$$e_{i,j} = \frac{\sum_k \mathbb{1}_{R_i^k = R_j^k}}{\sum_k \mathbb{1}_{\exists (R_i^k \cap R_j^k)}}$$

The Similarity-based Grouping starts by isolating the naive malicious workers which will return results/votes while consistently disagreeing with all other nodes. Isolating these workers is simple; we apply the EigenTrust algorithm proposed by Kamvar et al. [25] to isolate naive malicious workers, which we will save into a list of malicious workers $\mathcal{M}$. After removing $\mathcal{M}$ from the SG graph, the resulting graph consists only of *honest* and *colluding* workers.

We use a graph partitioning algorithm to cut the graph into two sub-graphs forming two disjoint groups. Graph partitioning algorithms such as Markov Cluster Algorithm (MCL) [26], Minimum Cut Tree Clustering (MinCTC) [27], or Spectral Clustering (SP) [28], can be used. We select one of these algorithm attractively and stop as soon as the graph is partition into two sub-graphs.

However, if the graph partitioning algorithm fails to portions the SG graph, while we have detected the presence of colluding workers in the network (*i.e.,* based on the collusion detection module), *SERENE* employs a greedy heuristic to construct two groups of nodes based the outcome of the detection module. Assume the collusion detection module has detected collusion based on task T_i after receiving a voting results R_j^i from worker S_j.

[3]It has been shown that 8 is sufficient to have an accurate similarity graph [11]

Therefore, we construct two groups as follows; worker node S_j will form a group $G1$ with the other server worker S_k such that $R_j^i = R_k^i$ (*e.g.,* from the example of Figure 1, G1 will be formed by two worker nodes returning the red R' result), The other group $G2$ is formed by all worker nodes returning the majority vote result and the remaining worker nodes which did not return any voting result yet (*e.g.,* from the example of Figure 1, G2 will be formed by the all worker nodes returning the green R result as well as all other worker nodes which where not probed yet). $G2$ will then be updated by removing all naive malicious workers $\mathcal{M}$ computed prior to the group partitioning.

2) Group Identification: Identify and Isolate Colluding Workers: At this step, the main goal is to identify which group includes honest and which one includes colluding workers. *Unlike other state-of-the-art research, we do not assume the presence of a trusted third party servers or trusted tasks (i.e., with guaranteed results) to guarantee efficient identification of these two groups.*

SERENE constructs a subset of trusted tasks (TT) from the original task repository, TR. A task $T_k \in TR$ is called a trusted task if and only if $\exists i \in G1 \ and \ j \in G2 | R_i^k = R_j^k$, where $G1$ and $G2$ are the two unnamed groups identified in the previous step. In other words, trusted tasks are the ones where workers from the two disjoints groups have agreed upon–colluding workers did not collude for these trusted tasks, thus the result returned for this task can be trusted. Note that colluding workers may not decide to collude if they did not form a majority of the pool or with a probability of $1 - P_c$.

The list TT of trusted tasks will be utilized to classify the workers into honest and colluding workers. However, since colluding nodes may not collude all the time, *SERENE* sends multiple tasks for each worker of one of the two unnamed groups to classify the group, say $G1$, then the remaining group, $G2$, will constitute the other class of workers. We show, in sec. V-D, that this idea may not be sufficient and we may need to check the two groups instead of relying only on classifying only one and infer the other.

SERENE's group identification uses fewer trusted tasks and resources, if it starts the identification of the honest group, rather than the colluding workers group (details will be presented later in this section as we present the algorithm). In other words, if we start identification of $G1$ and $G1$ was classified as an *honest group*, verifying $G2$, consisting of colluding nodes is less complex and requires less trusted task. Otherwise, if $G1$ was classified as a *colluding group*, then verifying the *honest group*, $G2$ requires more trusted tasks.

SERENE uses the size of the group to predict its classification, using the assumption that honest workers are likely to outnumber colluding workers in the network. Note that *SERENE* is also able to correctly classify the worker even when this assumption is not accurate–*i.e.,* colluding workers represent the majority of the workers as we will explain in the algorithm and show results in the evaluation section V-D.

Say $G1$ is the bigger group, *SERENE* selects a pool of k workers $P \subseteq G1$ such that $S_j \in P$ have the minimum number of verification within $G1$, then for the selected pool P, *SERENE* identifies the first task $T_i \in TT$ such all workers $S_j \in P$ have never received task T_i, *i.e.,* find T_i such that $\forall S_j \in P, (S_j, *) \notin TT$. The verification algorithm stops when the minimum number of verification for all nodes is equal to e or when all tasks in TT have been utilized. The parameter e is the maximum number of trusted tasks verification used to tune the algorithm for accuracy.

SERENE assigns a reputation score Rs_i to each worker node S_i based on worker node S_i's votes after comparing these votes with the trusted tasks majority vote result. We assign a voting score of 1 if the vote agrees with majority and -1 otherwise. Let $V_i = \{v_1 \dots v_{e'}\}$ where $e' \leq e$ be the list of voting scores (*i.e.,* a list of 1 or -1). Rs_i is computed as follows:

$$Rs_i = \frac{\sum_{j=1}^{e'} v_j}{e'}$$

If $\forall S_i \in G1,\ Rs_i = 1$ then *SERENE* identify $G1$ as a honest group–all workers in G1 are honest. Otherwise, *SERENE* runs a k-mean clustering algorithm, with $K = 2$ to classify $G1$ into two subgroups, $G1.1$ and $G1.2$. Assume that we name $G1.1$ such that the average reputation score of $G1.1$, $\overline{R_{i \in G1.1}}$ is the highest — $\overline{R_{i \in G1.1}} > \overline{R_{i \in G1.2}}$. If (I) $|G1.1| \geq |G1.2|$, then, $G2 \leftarrow G2 \cup G1.2$, and $G1 \leftarrow G1.1$; $G1$ consists of honest workers and $G2$ consists primarily of colluding workers. Otherwise (II), *i.e.,* $|G1.1| < |G1.2|$, then, $G2 \leftarrow G2 \cup G1.1$, and $G1 \leftarrow G1.2$; $G1$ consists of colluding workers and $G2$ consists primarily of honest workers.

However, *SERENE* needs to verify that all workers $S_j \in G2$ are colluding (in case I or honest in case II) workers.

Case (I): The unverified unnamed group $G2$ consists primarily of colluding worker nodes. In this case, *SERENE* selects a pool of k workers from the unnamed group $G2$, *i.e.,* $P' = \{S_i, i = 1 \dots k \mid S_i \in G2\}$. *SERENE* selects tasks $T_i \in TT$ to verify the new pool P', similar to the selection for pool P. Any worker node S_i returning a vote which disagrees with the majority will be placed in the honest group, *i.e.,* $G1 \leftarrow G1 \cup S_i$, because S_i did not collude with the majority of colluding nodes, thus S_i is honest. We repeat this process until we exhaust all tasks $T_i \in TT$, because colluding nodes may not always collude and must be tested multiple times.

Case (II): The unverified unnamed group $G2$ consists primarily of honest worker nodes. In this case, *SERENE* iterates over all worker nodes $S_i \in G2$ and selects a pool of k workers formed with S_i and the remaining are colluding nodes, *i.e.,* $P' = S_1, S_2, \dots, S_k$ such that $\exists i \mid S_i \in G2$ AND $\forall j \neq i, S_j \in G1$. *SERENE* selects up to e tasks $T_i \in TT$ to verify the new pool P', similar to the selection for pool P. In this case, if worker node $S_i \in G2 \cap P'$ returns a vote which disagrees with the majority, then S_i is verified as honest and remains in $G2$, *SERENE* then selects/verifies another worker S_j until all nodes are verified. However, if after e task verification, all worker nodes in P' return the same vote then they are all colluding workers and *SERENE* will place S_i in the colluding group, *i.e.,* $G1 \leftarrow G1 \cup S_i$.

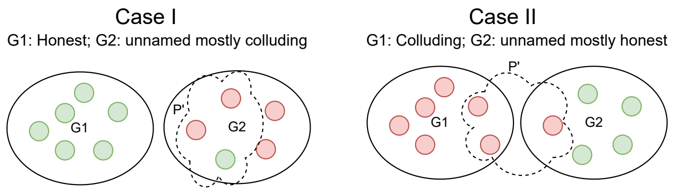

Case I
G1: Honest; G2: unnamed mostly colluding
G1
P'
G2
Case II
G1: Colluding; G2: unnamed mostly honest
P'
G1
G2

Fig. 2: Group identification example: In Case (I), where G1 consists of honest workers, *SERENE* selects pools P' entirely from G2; nodes disagreeing with majority are honest and added to G1; In Case (II), where G1 consists of colluding workers, P' is formed with two nodes from G1 and one from G2, until we verify all G2 members (*i.e.,* G2 members agreeing with majority are colluding and added to G1).

Figure 2 depicts two examples of case (I) and (II); in the first example G1 was named as $\mathcal{H}$ thus G2 is unnamed and consists mainly of colluding nodes, however the clustering algorithm may have misclassified few honest workers as colluding. In this case, *SERENE* selects its pool of workers for verification from G2, looking for any inconsistency in the votes which result in classifying the node as honest and adding it to G1. In case (II) (the sub-figure from the right in Figure 2), G2 consists primarily of honest workers, thus *SERENE* verifies worker by worker from G2 with a pool P' formed with colluding workers from G1. A pool showing zero inconsistencies, results in classifying the verified worker as colluding.

Finally, *SERENE* has identified all worker groups (i) naive malicious ($\mathcal{M}$), honest workers ($\mathcal{H}$), and colluding workers ($\mathcal{C}$). Afterward, *SERENE* clears all task sets, TR and TT and periodically reruns the detection module to identify any new potential *colluding* nodes.

V. Evaluation

In this section, we compare *SERENE*'s performance to the closest state-of-the-art algorithm, Staab & Angel, which we refer to as SnE [11], one of the existing collusion mitigation approaches that uses comparable set of assumptions. Note that none of the existing research has claimed or developed a method to detect and mitigate collusion when colluding workers exceeds 50% of the nodes in the network.

A. Simulation Setup

We implement *SERENE* and simulate different workload and network scenario using a python simulator we have developed. We consider a network of $N = 20$ workers. While we have considered naive malicious workers in our design, we do set $\mathcal{M} = \emptyset$ because identifying $\mathcal{M}$ is very trivial and has been solved by many state-of-the-art algorithms including SnE [11]. We simulate a 20 to 25 milliseconds random round trip time communication delay between all nodes at the edge. We implement two variations of SnE, SnE 8 and SnE 12 using using $e = 8$ (recommended value in [11]) and $e = 12$ observations per edge. We have tested other implementations using different clustering algorithm such as SP and MinCTC, however in this paper we show only results for SnE using MCL clustering which achieved the best results.

We vary the percentage of colluding workers $\mathcal{C}$ from 10%, to 90% from the set of N workers. Colluding nodes can collude with a probability P_c ranging from 10% (*i.e.,* rarely collude), to 90% (*i.e.,* mostly collude).

Each node generates a set of tasks at a rate of 1000 tasks per second. Each task is sent to $k = 3$ workers for task verification. We run the simulation for 100 seconds, with collusion starting randomly, following a uniform distribution between 3 seconds and 90 seconds. We repeat each simulation 100 times and measure the distribution of *SERENE*'s and SnE's results.

Parameters	**Acronym**	**Values**
Number of worker nodes	N	20
% colluding workers	$\|C\|$	$10\%, \ldots, 90\%$
Probability of collusion	P_c	10, 50, 90%
Error rate for honest workers	ϵ	0.3%
Simulation end time	–	100s
Maximum observation per edge	e	12 obs/edge
Users task generation rate	–	1000 task/sec
Communication round trip delay	RTT	$\{20 \ldots 25\}$ms

TABLE I: Simulation parameter for evaluation of *SERENE*

B. Evaluation Metrics

In our experiments, we evaluate *SERENE*'s and SnE's performances using the following metrics:

- *Collusion detection delay:* Measured as the difference between the time when *SERENE* detects the presence of collusion and the start time of the collusion. The start time of collusion is simulated (*i.e.,* known) in our evaluation. We also measure the number of epochs or iterations in addition to the detection delay which is reported in seconds.
- *Collusion detection accuracy:* We use the f1-score ratio of the accurate detection and the falsely detected collusion (*i.e.,* either collusion not detected, or falsely detected).
- *Collusion mitigation accuracy:* We measure the accuracy of *SERENE*'s mitigation algorithm as an f1-score ratio between the number of accurately detected and falsely detected (*i.e.,* colluding workers classified as honest or honest classified as colluding workers) colluding workers.
- *Collusion mitigation latency:* Measured as the difference in time between completing the mitigation (*e.g.,* for *SERENE*, this time is the end of the identification of both groups $G1$ and $G2$) and the start of the collusion.

C. SERENE*'s Collusion Detection Performance*

SERENE's collusion detection is measured using two main metrics; collusion detection delay and collusion detection accuracy.

Figure 3 compares the performance of *SERENE*'s and SnE's collusion detection performance in terms of collusion latency (sub-figures a and b) or collusion detection accuracy (sub-figure c). Figure 3-(a) and (b) we measure the detection latency in seconds and in number of iterations respectively and we plot the cumulative distribution function (CDF) (where Inf denotes infinite delays resulting from inability to detect collusion when collusion exists). Note that the CDFs include results from different simulation runs, as well as varying P_c and $\mathcal{C}$ values.

In Figure 3a, *SERENE* outperforms SnE and detects collusion up to 10 $\times$ faster. For half of the delays (50% percentile), *SERENE* detects collusion at 0.85 seconds or less however, SnE12 detects collusion at 50% longer delays, at 1.2 seconds when $e = 12$, and mostly unsuccessful delays when $e = 8$, SnE8. Moreover, while *SERENE* accurately detects collusion with more than 98%, SnE fails more than 30% and 80% of the time when using $e = 12$ and $e = 8$ respectively.

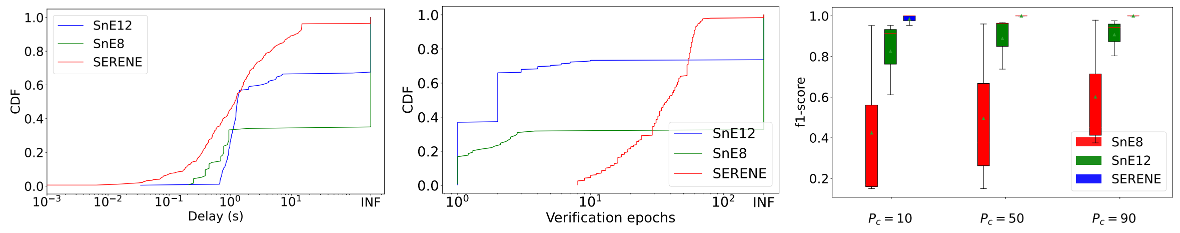
We also plot the CDF of the detection delay in algorithm epochs in Figure 3b. Note that the epochs of both *SERENE* and SnE are incomparable–*i.e.,*SnE operates periodically and waits to gather e observation per edge to perform its clustering, while *SERENE*'s epoch is one task at the time and detection occurs when a given task detects two groups. We show that *SERENE* detects collusion accurately in 90 or less tasks with half of the collusion scenario detected within 35 tasks.

The accuracy of both algorithms is further compared in Figure 3c. While all algorithms perform better as the probability of colluding increases among the workers, *SERENE* outperforms SnE and achieves a 98% accuracy or more in detecting collusion. However, SnE, and especially SnE8, show major variation in accuracy performance and failing to detect collusion from 6% to 27% of the time for SnE12 and 43% to 83% of the time for SnE8, when P_c =50%. In fact, SnE relies on periodic data gathering and triggers a clustering algorithm to find two groups of workers, however most of the time it clusters the network into two group of workers even in the absence of collusion. Notably, the authors did not test SnE algorithm in absence of collusion in their original paper [11].

In addition to *SERENE*'s high accuracy in detecting colluding workers, the detection algorithm, consisting of zero lookup and simple mathematics operations, is both fast and efficient. We conduct a bench-marking analysis and show results in section V-E.

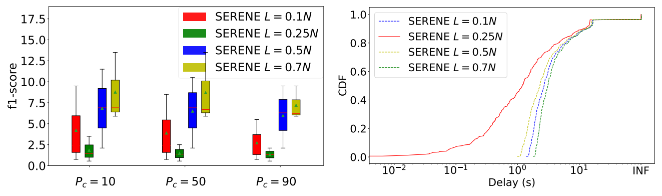
Impact of CVT size on *SERENE* detection performance: We vary the percentage of collusion P_c to analyze the impact of the size of the CVT table (consists of tasks used to detect the presence of two groups in the network). *SERENE* detects collusion faster as the probability of collusion increases, since the more collusion instances amongst workers the faster we detect two inconsistent groups, thus *SERENE* detects collusion faster.

Additionally, we show in Figure 4 that there may exist an optimal value of L ($L \approx .25 \times N$) and very large ($L = .7 \times N$) and very small ($L = .1 \times N$) values perform poorly. Large L values are discouraged because the larger the table size the longer to gather multiple votes for any given task (because *SERENE* chooses tasks randomly), thus the slower the detection. Moreover, very small table sizes



(a) CDF of detection delays in seconds (x-axis in logscale) (b) CDF of detection delays in epochs (x-axis in logscale) (c) Collusion detection accuracy

Fig. 3: Comparing *SERENE*'s and SnE's collusion detection delay (a) and (b) and accuracy (c); INF denotes infinite delay values due to unsuccessful collusion detection



(a) Collusion detection delay as a function on P_c (b) CDF of detection delay $\forall$ P_c, $\mathcal{C}$ (x-axis in logscale)

Fig. 4: Collusion detection delay as a function of task repository size, L

results of replacing the tasks that have been used to verify all workers, then *SERENE* replaces them which slows the detection. The CDF of all collusion detection delays $\forall$ P_c, $\mathcal{C}$ (Figure 4b) shows that $L = .25 \times N$ can achieve more than $10\times$ improvement in latency, making it an important tuning parameter for *SERENE*'s collusion detection.

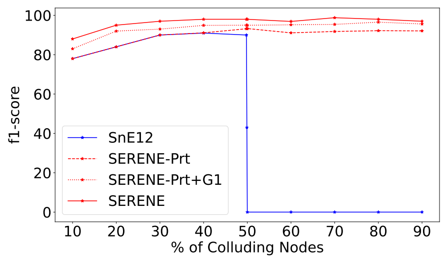
D. SERENE *Collusion Mitigation Performance*

Collusion detection is the first step towards addressing the collusion problem, efficient mitigation is also important. We evaluate, in Figure 5, *SERENE*'s mitigation performance by measuring both mitigation accuracy and latency and compare these performance to those of SnE. We compare *SERENE* only to SnE12 which exhibits the best performance compared to SnE8 and for better readability. Additionally, we examine multiple implementations of *SERENE* namely: (i) *SERENE*-Partitioning: an implementation of *SERENE* up to the similarity-based partitioning phase, where *SERENE* identifies two unnamed clusters G1 and G2 (*i.e.,* section IV-B2), (ii) *SERENE*-Partitioning+G1): an implementation of *SERENE* up to the identification of G1 and removal of misclassified G1 members (see section IV-B2), and (iii) *SERENE*: the full implementation of *SERENE* design.

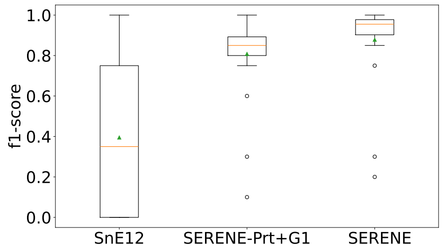
In Figure 5a, we demonstrate that all *SERENE* versions outperform SnE by up to 10% when the population of colluding workers is low and more than 95% more when the percentage of colluding workers exceed 50%. Note that SnE works only when colluding workers represent the minority group and it uses this assumption to identify $\mathcal{C}$. However, *SERENE* performs best when $\mathcal{C}$ is large because it detects the two groups and gathers more observations per edge faster. In addition, while *SERENE*-Partitioning performance is almost identical to SnE's performance for lower $\mathcal{C}$ values, the G1 identification phase only enhances *SERENE*'s performance by up to 5% while maintaining more than 85% accuracy when colluding workers are the majority in the network. *SERENE* achieves up to 95% accuracy when P_c =0.5. Moreover, Figure 5b shows similar performance when the distribution of delays for all P_c values in included–*SERENE* outperforms SnE12 and achieving consistent accuracy above 80% $\forall$ P_c and $\mathcal{C}$.

This major accuracy gain comes with a minor latency cost shown in Figure 5c. While SnE have almost constant mitigation delay which constitute the time needed to gather sufficient observation per edge to cluster the graph, *SERENE* employs the identification phase to identify both groups G1 and G2 (*SERENE* does not assume that colluding workers represent the majority in the network) and further verifies the correctness of the clustering (*e.g.,* misclassified workers). These steps help achieve the accuracy gains highlighted in Figure 5a. *SERENE*'s mitigation latency are 500ms to 1.5s more than SnE12 mitigation delays. We argue that accurate identification of colluding workers in the network is essential, thus the additional delay cost introduced by *SERENE* is justified; in fact, SnE with inaccurate identification of colluding nodes may use another detection and mitigation phased to identify the misclassified workers which will result in much larger latencies.

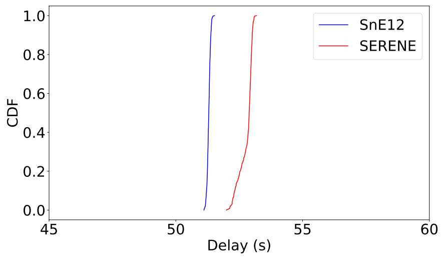
Effect of e on collusion mitigation accuracy: We have discussed the impact of e on SnE performance. *SERENE* can also be tuned with different values of e to enhance its mitigation accuracy (detection accuracy does not seem to be impacted by e). We compare, in Figure 6, the collusion mitigation accuracy of *SERENE* when $e = 5, 10, 20, 40$. We show that the impact of e is highlighted more for lower probability of collusion, P_c (more 30% accuracy gain when comparing $e = 5$ and $e = 10$). Lower P_c requires more observations for colluding nodes to consistently collude with each others. However, we also observe that *SERENE*-e=10 seems to achieve a good trade-off and there is not much improvement recorded as we increase e (less than 1.5% for



(a) Comparing mitigation accuracy for $P_c = 50\%$



(b) Collusion mitigation accuracy $\forall\ P_c$



(c) CDF of mitigation latency for all $\mathcal{C}$ and P_c values

Fig. 5: Comparing mitigation accuracy (a, b) and latency (c) for SnE, *SERENE* implementation up to the grouping phase (*SERENE*-Prt), *SERENE* implementation up to the identification of G1 (*SERENE*-Prt+G1), and *SERENE* (*i.e.,* the full design).

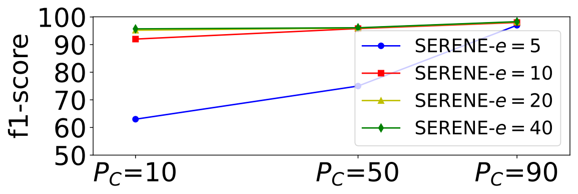
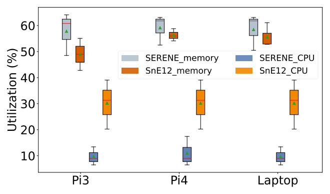
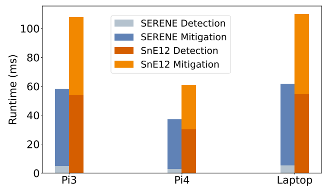


Fig. 6: Comparing the collusion mitigation accuracy for different observation per edge values (e).



(a) Memory and CPU utilization



(b) Run times

Fig. 7: The benchmarking tests for memory and cpu utilization (a) as well as run times (b) using a Raspberry Pi3, Raspberry Pi4, and a Laptop machine.

higher P_c values). Note that we choose $e = 12$ for all other experiments which achieves good trade off between accuracy and delay.

E. Prototyping and Benchmarking Tests

We conduct a series of benchmarking tests using various machines/platforms including: (i) a Raspberry Pi v3b featuring a quad-core 1.2 GHz 64-bit CPU and 1 GB of RAM, (ii) Raspberry Pi v4 with a quad-core 1.8 GHz CPU and 4 GB of RAM (Pi4), and (iii) an older laptop equipped with an Intel dual-core 2.0 GHz 64-bit CPU and 2 GB of RAM (Laptop).

We implement both *SERENE* and SnE on these three platforms and perform a set of benchmarking tests to measure CPU usage, memory usage, and run times of the proposed schemes. We repeat the experiments 50 times and plots the distribution of all gathered result data in Figure 7. While *SERENE* consumes 9% more memory to load its tables and executes the detection and mitigation algorithms (Figure 7a), it utilizes more than half the CPU when compared to SnE's CPU usage–*SERENE* uses on average 10% CPU while SnE uses 33%. Note that none of the tested platform shows over-utilization of the resources, thus the very small differences when we compare results across the three platforms.

Moreover, we can show in Figure 7b, that *SERENE* runs twice as fast than SnE across the three tested platforms. This gain is mainly due to the fast detection algorithm which runs 10$\times$ faster than SnE's collusion detection– *SERENE* detects in 5 milliseconds, while SnE detects collusion in 54 milliseconds. Note that both algorithms were implemented in the same fashion–*i.e.,* no parallel computing for *SERENE* or SnE.

VI. Conclusion and Future Work

We have introduced a new worker collusion resilient replication based task verification scheme called *SERENE*. Unlike state-of-the-art collusion resilient solutions, *SERENE* is lightweight and efficient to detect the presence of colluding workers, and isolate them. *SERENE*'s detection relies on identifying two clusters of workers consistently disagreeing with each others. Furthermore, *SERENE* uses a three step mitigation to partition the group of workers and identify the colluding ones. Our results show that *SERENE* detects the existence of collusion more accurately (with more than 98% success ratio, which represents 30% improvement compared to SnE), and 5$\times$ faster than SnE's detection delay.

In the future, we will consider the threat model where colluding nodes can be organized in different groups and collude only with their corresponding groups. This would make collusion mitigation more challenging, requiring an extension of *SERENE* to identify the group of honest/benign nodes among multiple clusters. Thus the mitigation phase should be updated to find different cluster of colluding ndoes.

Acknowledgment

This material is based upon work supported by the National Science Foundation under grant #2148358 and is supported in part by funds from OUSD R&E, NIST, and industry partners as specified in the Resilient & Intelligent NextG Systems (RINGS) program.

References

[1] M. Walfish and A. J. Blumberg, "Verifying computations without reexecuting them," *Communications of the ACM*, vol. 58, no. 2, pp. 74–84, 2015.

[2] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers," in *Annual Cryptology Conference*, pp. 465–482, Springer, 2010.

[3] P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Trusted execution environments: properties, applications, and challenges," *IEEE Security & Privacy*, vol. 18, no. 2, pp. 56–60, 2020.

[4] R. Canetti, B. Riva, and G. N. Rothblum, "Practical delegation of computation using multiple servers," in *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 445–454, 2011.

[5] G. Levitin, L. Xing, and Y. Dai, "Optimal spot-checking for collusion tolerance in computer grids," *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 2, pp. 301–312, 2017.

[6] G. C. Silaghi, F. Araujo, L. M. Silva, P. Domingues, and A. E. Arenas, "Defeating colluding nodes in desktop grid computing platforms," *Journal of Grid Computing*, vol. 7, no. 4, pp. 555–573, 2009.

[7] A. Küpçü, "Incentivized outsourced computation resistant to malicious contractors," *IEEE Transactions on Dependable and Secure Computing*, vol. 14, no. 6, pp. 633–649, 2015.

[8] S. Zhao, V. Lo, and C. G. Dickey, "Result verification and trust-based scheduling in peer-to-peer grids," in *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05)*, pp. 31–38, IEEE, 2005.

[9] C. Dong, Y. Wang, A. Aldweesh, P. McCorry, and A. van Moorsel, "Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 211–227, 2017.

[10] L.-C. Canon, E. Jeannot, and J. Weissman, "A dynamic approach for characterizing collusion in desktop grids," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1–12, IEEE, 2010.

[11] E. Staab and T. Engel, "Collusion detection for grid computing," in *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pp. 412–419, IEEE, 2009.

[12] M. Backes, D. Fiore, and R. M. Reischuk, "Verifiable delegation of computation on outsourced data," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 863–874, 2013.

[13] K. Elkhiyaoui, M. Önen, M. Azraoui, and R. Molva, "Efficient techniques for publicly verifiable delegation of computation," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pp. 119–128, 2016.

[14] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "Vc3: Trustworthy data analytics in the cloud using sgx," in *2015 IEEE symposium on security and privacy*, pp. 38–54, IEEE, 2015.

[15] N. O. Duarte, S. D. Yalew, N. Santos, and M. Correia, "Leveraging arm trustzone and verifiable computing to provide auditable mobile functions," in *Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, pp. 302–311, 2018.

[16] M. Belenkiy, M. Chase, C. C. Erway, J. Jannotti, A. Küpçü, and A. Lysyanskaya, "Incentivizing outsourced computation," in *Proceedings of the 3rd international workshop on Economics of networked systems*, pp. 85–90, 2008.

[17] K. Watanabe, M. Fukushi, and S. Horiguchi, "Collusion-resistant sabotage-tolerance mechanisms for volunteer computing systems," in *2009 IEEE International Conference on e-Business Engineering*, pp. 213–218, IEEE, 2009.

[18] Y. Kong, C. Peikert, G. Schoenebeck, and B. Tao, "Outsourcing computation: the minimal refereed mechanism," in *International Conference on Web and Internet Economics*, pp. 256–270, Springer, 2019.

[19] L. Wang, Y. Tian, and J. Xiong, "Achieving reliable and anti-collusive outsourcing computation and verification based on blockchain in 5g-enabled iot," *Digital Communications and Networks*, 2022.

[20] J. Ménétrey, C. Göttel, M. Pasin, P. Felber, and V. Schiavoni, "An exploratory study of attestation mechanisms for trusted execution environments," *arXiv preprint arXiv:2204.06790*, 2022.

[21] F. Araujo, J. Farinha, P. Domingues, G. C. Silaghi, and D. Kondo, "A maximum independent set approach for collusion detection in voting pools," *Journal of Parallel and Distributed Computing*, vol. 71, no. 10, pp. 1356–1366, 2011.

[22] A. Bendahmane, M. Essaaidi, A. El Moussaoui, and A. Younes, "The effectiveness of reputation-based voting for collusion tolerance in large-scale grids," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 6, pp. 665–674, 2014.

[23] J. Wang and A. Mtibaa, "CoVFeFE: Collusion-Resilient verifiable computing framework for Resource-Constrained devices at network edge," in *2024 IEEE 13th International Conference on Cloud Networking (CloudNet) (IEEE CloudNet 2024)*, (Rio de Janeiro, Brazil), p. 8.85, Nov. 2024.

[24] A. M. Sauber, A. Awad, A. F. Shawish, and P. M. El-Kafrawy, "A novel hadoop security model for addressing malicious collusive workers," *Computational Intelligence and Neuroscience*, vol. 2021, 2021.

[25] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina, "The eigentrust algorithm for reputation management in p2p networks," in *Proceedings of the 12th international conference on World Wide Web*, pp. 640–651, 2003.

[26] Markov Clustering Documentation, "Markov Clustering Documentation." https://markov-clustering.readthedocs.io/en/latest/. [Online; accessed March 15, 2024].

[27] U. Brandes, M. Gaertler, and D. Wagner, "Experiments on graph clustering algorithms," in *European symposium on algorithms*, pp. 568–579, Springer, 2003.

[28] Markov Clustering Documentation, "Markov Clustering Documentation." https://scikit-learn.org/stable/modules/generated/sklearn.cluster.SpectralClustering.html. [Online; accessed March 15, 2024].