CoVFeFE: Collusion-Resilient Verifiable Computing Framework for Resource-Constrained Devices at Network Edge

Jianyu Wang
Department of Computer Science
University of Missouri St. Louis
St. Louis, USA
jianyu.wang@mail.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/CLOUDNET62863.2024.10815825

***Abstract*—With the rapid growth of Internet of Vehicles (IoV) applications and the advancement of edge computing, resource-limited vehicles (and other IoT devices) increasingly rely on external servers to handle diverse and complex computational tasks. However, this dependence on external servers, which may be malicious or compromised, introduces significant security risks. Replication-based verifiable computing has been proposed as a solution to verify the accuracy of task results, but these approaches are vulnerable to collusion, where compromised servers return identical incorrect results to mislead the vehicle. Existing defenses against collusion either cannot ensure complete protection or become ineffective as the number of colluding servers rises. In this paper, we introduce *CoVFeFE*, a collusion-resilient verification framework designed to detect and mitigate collusion, even when the majority of servers are compromised. Our framework integrates a rapid detection mechanism that monitors computational conflicts, alongside a heuristic mitigation strategy that identifies and neutralizes colluding servers. Simulation results demonstrate that *CoVFeFE* outperforms existing solutions by successfully identifying all colluding servers, even when they constitute a majority $\geq$50%) of the network.**

***Index Terms*—Verifiable Computing, Internet of Vehicles, Edge Computing, Collusion**

I. Introduction

The Internet of Vehicles (IoV), a key component of the Internet of Things (IoT), has been proposed to enable secure and intelligent transportation within smart cities [1]. However, many vehicles lack the computational resources to process data locally. For instance, autonomous vehicles typically offload their computational tasks to remote servers, such as remote cloud/edge infrastructure (*e.g.,* roadside units) or to a self driving compute device located in the vehicle. This outsourcing model alleviates the resource constraints but introduces new challenges, particularly related to trust and security [2].

Verifiable computing, which ensures the correctness of outsourced computational results without requiring local (re-)execution, has been extensively studied in recent years [3]. Existing approaches generally fall into one of three categories: (i) using probabilistically checkable proofs to identify incorrect results with high confidence [4], (ii) employing Trusted Execution Environments (TEEs), such as Intel SGX, to guarantee the integrity of computation [5], and (iii) using replication-based schemes, where the same task is redundantly processed by multiple servers, and majority voting is used to determine the correct result [6]. While proof-based and TEE-based solutions often designed to work on specific applications or hardware, replication-based solutions are more widespread due to their simplicity and ease of implementation [7].

However, most replication-based approaches are vulnerable to collusion, where malicious servers cooperate to return identical incorrect results [8]. To combat this, various collusion prevention and detection techniques have been proposed. Prevention methods typically involve increasing the number of replicas [9] or incentivizing rational servers to expose colluders [10], while spot-checking methods utilize trusted tasks to randomly verify servers' behavior, offering an additional layer of protection [11]. Despite these measures, prevention strategies cannot fully eliminate the risk of collusion. As a result, detection and mitigation techniques have been developed to identify colluding servers [6], [12]–[14]. To the best of our knowledge, most existing methods assume that fewer than half of the servers in the network are colluding, rendering them ineffective if the majority of servers are compromised. In contrast, our work introduces a novel approach that detects and mitigates collusion even when the majority (*i.e.,* $\geq$50%) of servers in the network are malicious.

In this paper, we propose *CoVFeFE*, a collusion-resilient replication-based verification framework for resource-constrained devices at network edge. *CoVFeFE* is a client-side framework that runs on resource-constrained devices at the edge network, where the majority of servers can be malicious and colluding. Our contributions are summarized as follows: (**i**) *CoVFeFE* offers continuous detection and mitigation of collusion with high accuracy and minimal latency. Unlike existing solutions, it identifies collusion regardless of the percentage of malicious servers in the network by monitoring discrepancies in computation results from two different groups of servers processing the same task. Once a conflict is detected, a heuristic mitigation mechanism is triggered to identify the colluding servers, discard tasks with erroneous results, and correct any misclassification of benign servers. *CoVFeFE* compute its own trusted task list to enhance the efficiency of collusion mitigation; and (**ii**) we implement *CoVFeFE* and evaluate its performance through a series of experiments using the ns-3 network simulator. We compare *CoVFeFE* against two state-of-the-art replication-based solutions: baseline replication (*R_baseline*) and collusion-prevention replication (*R_Kupcu*). Our experimental results demonstrate that *CoVFeFE* outperforms both alternatives, successfully identifying all colluding servers even when the majority of servers are compromised, where the other two solutions fail as the number of colluding servers increases. Additionally, *CoVFeFE* significantly reduces overhead, achieving up to an $8\times$ and $100\times$ reduction compared to *R_baseline* and *R_Kupcu*, respectively.

II. Related Work

Verifiable computing solutions are often classified into three main categories: (i) *proof-based* solutions using probabilistically checkable proofs (PCPs) generated by servers together with computation results [4], [15], [16]; (ii) *TEE-based* solutions deploy remote attestation for a client to verify the launching of the trusted execution using isolated tamper-resistant environments, such as Intel SGX [17] and ARM TrustZone [18]; and (iii) *replication-based* solutions, famous for their generic use, outsource same computation tasks to multiple servers and verify the correctness using consensus protocols (*e.g.,* majority voting) on the returned results [19], [20]. Research has reduced the overhead of redundant communication by limiting task replicas [6], [21], but the risk and danger posed by colluding servers have long been pointed out, and remain a difficult challenge to address [22].

While collusion defense solutions have been mainly proposed for replication-based verification schemes, there have been few proof-based and TEE-based collusion research discussions and solutions [3]. Proof-based collusion is limited to the risk of delegating verification or proof setup to a colluding third party [23], whereas the collusion for TEE-based solutions is mainly due to a rogue remote attestation [24].

State-of-the-art replication-based collusion defense approaches have focused on either prevention [9], [10], [19], [25], [26] or detection [11]–[14], [27], [28] of collusion. Prevention approaches aim to discourage or reduce collusion occurrences by enlarging the number of replicas [9], deploying game theory-based fine and reward contracts to incentivize "rational" servers to betray collusion [10], or spot checking [11] by a list of trust tasks. While prevention can help reduce collusion, it cannot be avoided, hence detection and mitigation are necessary when collusion occurs.

Detection and mitigation solutions have focused on optimizing the detection of colluding servers [12], [13]. Silaghi et al. [12] 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. [13] applied a one-step graph clustering algorithm to identify both benign and colluding servers. However, all these solutions are able

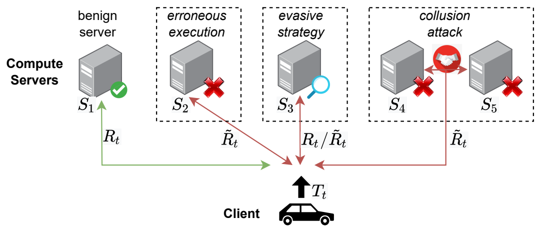


Fig. 1: Client sends task T_t to any of five servers and receives a result: S_1 (benign), $S_2 - S_5$ (malicious performing erroneous execution, evasive strategy and collusion attacks)

to identify benign servers only while assuming that colluding servers represent a minority in the network.

Unlike other collusion detection and mitigation schemes, *CoVFeFE* is able to detect and mitigate collusion even in the presence of more colluding than benign servers in the network. It uses a heuristic to determine the existence of collusion, then identifies and isolates the colluding servers.

III. System and Threat Model

A. System Model and Assumptions

We consider an edge network, where multiple client devices (which we refer to as clients) rely on an edge computing infrastructure to execute any client task T_t at time t requiring remote execution. This compute infrastructure consists of $\mathcal{S}$, a set of edge computing servers (servers for short), where each server $S_j, j = \{1, \cdots, N_S\}$. We assume that clients connect and send all computation tasks to the closest server at any given time. We refer to this closest server selected by the client as the current compute server with respect to the client.

We model each task T_t sent by a client at time t, and also assume that each task T_t has a unique correct result R_t associated with it. We also assume that a client is incapable of knowing R_t prior, thus requests execution of the task from remote servers and relies solely on the results returned by these servers. Note that results returned by servers after execution of any task may or may not be correct (refer to threat model for more attack details).

B. Threat Model

Here we present the edge computing threat vectors that *CoVFeFE* will help address. Servers may not be reliable or trustworthy [29], hence outsourcing computations to these servers may return erroneous results, particularly with misbehaving malicious edge nodes. We refer to this attack as the *erroneous execution* attack where malicious servers will always return an erroneous result $\tilde{R}_t \neq R_t$ for a given task T_t. This attack can be performed by skipping the execution of the task altogether or maliciously swapping the correct result. Replication-based verification can address the erroneous execution attack, but servers may implement extra strategies to evade verification. We consider that malicious servers can implement an *evasive strategy* that aims at identifying a verification mechanism in action and implementing a countermeasure. For instance, servers implementing an *evasive strategy* can return R_t when

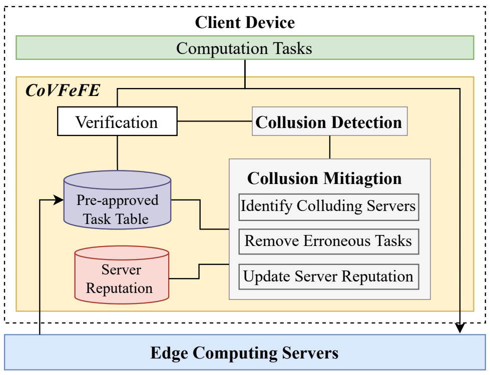


Fig. 2: Framework Overview: *CoVFeFE* performs task verification on the current compute server and detect/mitigate collusion using the tasks from the pre-approved task table.

verification is identified and $\tilde{R}_t$ otherwise for any task T_t. We also consider a *collusion attack*, where servers always collaborate to share the exact erroneous result $\tilde{R}_t$ for the same task T_t. We consider that colluding servers are irrational (*i.e.,* cannot be incentivized with money or credit) and cheat clients by tricking them into accepting incorrect results of erroneous execution. Our design aims for resiliency against all or any combination of these three malicious intents: (i) *erroneous execution*, (ii) *evasive strategy*, and (iii) *collusion attack*, which significantly degrade the performance of edge computing-aided IoV applications. Fig. 1 depicts a scenario where a client sends task T_t for remote execution to any server located at the edge of the network; in this example, five servers are available: one benign (S_1) and four malicious servers performing erroneous execution (S_2), evasive strategy (S_3) and collusion attack (S_4 and S_5). Finally, we assume in this paper that collusion is temporal, and there exists a time when the network is free of collusion.

IV. Framework Overview

As shown in Fig. 2, *CoVFeFE* interacts with any replication-based verification mechanisms by building its own list of pre-approved tasks and overwriting verification in the presence of server collusion. Note that this plug-and-play method, requires little to no overhead when there is no server collusion detected.

CoVFeFE's design is centered around the detection and mitigation of server collusion even when the majority of servers in the network are colluding. When most servers are colluding, monitoring result consensus can be misleading, and detection may fail. Thus, unlike other state-of-the-art solutions, which monitor a large list of tasks *CoVFeFE*, selects a list of verification tasks (pre-approved task table), and checks each task used for verification and detects collusion existence as soon as two different clusters of results are shown.

While detection can be efficient, *CoVFeFE* is unable to determine which cluster contains colluding servers and which one is benign. Thus, it implements a mitigation mechanism that aims at (1) identifying colluding servers, (2) removing the list of erroneous results caused by collusion consensus,

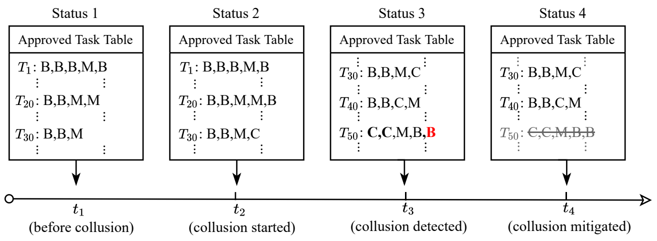


Fig. 3: Example of pre-approved task table statuses prior to and after collusion; A task "BBMC" in the pre-approved task table refers to a task that received results from two benign (B), one malicious (M), and one colluding (C) servers. Detection of collusion occurs when the client has verified two group of results in the poisoned task, T_{50}.

and (3) updating the list of benign servers which may have suffered from the detected collusion attack.

V. *CoVFeFE*'s Collusion Detection and Mitigation

CoVFeFE implements two main methods, namely the detection of collusion and the mitigation of colluding servers, which includes identifying the servers colluding and revoking any poisoned task and/or reputation miscalculation.

A. *Detection of Collusion*

CoVFeFE implements an efficient detection function that determines the presence of collusion after each verification. Once a task T_t is sent to verify a server S_j, *CoVFeFE* invokes the $CollusionExists(T_t, R_t)$ function as soon as S_j returns the result R_t for this verification (Alg. 1, L1). The function performs a lookup on all returned results for task T_t[1], and detects collusion if and only if there exists another server returning the same result R_t for this task, while another group of servers (≥ 2) share a different result $R'_t \neq R_t$ (this task called a poison task). Note that this detection function can only guarantee collusion presence but cannot determine if the verified server is colluding or benign. An example is depicted in Fig. 3 showing how a sample pre-approved task table may look prior to collusion and during collusion. The collusion is detected at time t_3, at status 3, where the client uses a poisoned task (T_{50}) to verify two benign servers (denoted by 'B' in Fig. 3). Note that detection can also occur if the client uses a non-poisoned task for verification, *e.g.,* T_{40} in Fig. 3 to verify a colluding server (denoted by 'C' in Fig. 3).

B. *Mitigation of Collusion*

As soon as collusion is detected, *CoVFeFE* triggers a mitigation process that aims to mitigate and correct any damage caused by colluding servers. Specifically, *CoVFeFE* identifies colluding servers, removes poisoned tasks from the pre-approved task table, and recovers the reputation of benign servers in the following steps:

Two-Phase Identification of Colluding Servers: In the first phase, *CoVFeFE* overrides the verification framework to verify

[1]All results and the corresponding servers returning them are stored for each task in pre-approved task table.

all servers at once using a minimal number of "trusted" non-poisoned tasks. *CoVFeFE* uses a heuristic to select these trusted tasks. The heuristic consists of selecting tasks that were generated as early as possible before the time of collusion detection (while in section III we assumed the collusion does not start from the beginning).

We model the selection of the minimum number of "trusted" pre-approved tasks as a *set cover problem* [30]. Given that the problem is NP-hard, we apply the greedy heuristic algorithm [31] to find the minimal set, but we modify its searching strategy. *CoVFeFE* selects the tasks that are generated as early as possible before the time of collusion while sharing the same largest coverage of servers that never received the task. For instance, if collusion is detected at a given time t, the earlier the task was created before this time t, the higher the chance it is not poisoned. For example, as shown in Fig. 3 status t_2/t_3, task T_{50}, which has been gathered after collusion, is more likely to be poisoned compared to task T_{20} gathered prior to collusion. Algorithm, $SelectTasksGreedy(\mathcal{S})$ searches the list of all servers $\mathcal{S}$ and returns $\mathcal{T}$, the minimum set of tasks along with the list of servers to be verified using each task.

Then *CoVFeFE* sends all tasks in $\mathcal{T}$ to the corresponding servers and compares the results returned by servers to the pre-approved results. Servers that return incorrect results are tagged as colluding and added to a list of colluding servers $\mathcal{C}$ as shown in Alg. 1, L3, $IdentifyCollludingServers(\mathcal{T})$. Note that this step may identify non-colluding malicious servers as colluding due to wrongly selection of $\mathcal{T}$, thus overestimating the number of colluding servers. We choose this conservative approach to identify all colluding servers.

While this first mitigation phase is fast and identifies a group of colluding servers, smarter colluding servers can easily evade this step using an advanced evasive strategy (as described in Section III). Therefore, *CoVFeFE* uses a second mitigation phase to identify each server individually, as shown in Alg. 1, L4, $VerifyBenignServers(\mathcal{C}, \mathcal{S})$. It verifies each "benign" server (*i.e.,* a server passing the first phase successfully) with the most similar "trusted" task to each server to avoid verification detection and evasion. We present and describe this selection process in detail in Section VI-B. By selecting these tasks, colluding servers will return incorrect results and therefore get identified and be added to $\mathcal{C}$.

Estimation of Collusion Start Time: *CoVFeFE* estimates the time when the collusion starts in the network to identify the set of tasks that may be poisoned, and further investigate/clean them. Those unlikely to be poisoned can be used for future collusion detection and mitigation if needed. While *CoVFeFE* can detect collusion at time t_d, collusion must have started earlier in the network. *CoVFeFE* estimates the collusion start time by measuring the ratio of colluding servers in the network (a higher server collusion ratio is often proportional to the percentage of tasks returned by colluding servers), thus shorter collusion detection delays (as per *CoVFeFE*'s detection mechanism). To this end, we estimate the delay preceding the collusion detection as $t_s = t_d - \theta \times \frac{|S|}{|C|}$, where θ is the error tuning parameter. While this estimation of the start of collusion may not be accurate or precise, this conservative estimation is only used to reduce the overhead of looking up poisoned tasks and revoking incorrect results for a smaller list of tasks instead of the entire pre-approved task table.

Algorithm 1 Collusion detection and mitigation after a verification using task T_t at time t

```
 1: if $CollusionExists(T_t, R_t)$ then
 2:     $\mathcal{T} \leftarrow SelectTasksGreedy(\mathcal{S})$
 3:     $\mathcal{C} \leftarrow IdentifyCollludingServers(\mathcal{T})$
 4:     $\mathcal{C} \leftarrow VerifyBenignServers(\mathcal{C}, \mathcal{S})$
 5:     $t_s \leftarrow EstimateCollusionStartTime(|\mathcal{C}|, |\mathcal{S}|)$
 6:     $RemovePoisonedTasks(t_s, \mathcal{C})$
 7:     for all server $S_j \in \mathcal{C}$ do
 8:         $Isolate(S_j)$
 9:     end for
10: end if
```

Removal of Poisoned Tasks and Update of Server Reputations: *CoVFeFE* performs poisoned task removal to clean up the pre-approved task table using the identified colluding servers, $\mathcal{C}$, and updates the reputation scores of misclassified benign servers or/and identified colluding servers. Specifically, the task removal function (Alg. 1, L6) locates any poisoned task, T_k, if and only if the stored pre-approved result, R_k satisfies the following two conditions: (i) R_k is identical to the result shared by any identified colluding server, and (ii) R_k is different from the result shared by any benign server (if exists). The second condition is required because identified colluding servers may have been benign (at any given time), thus their task results, generated prior to colluding, may have been "correct". After each task removal, *CoVFeFE* identifies the list of benign servers impacted by the miss-classification of this task and performs a revocation of the benign server, and update list of colluding servers.

Finally, *CoVFeFE* isolates the colluding servers in the list $\mathcal{C}$ and remove it from list of servers, as shown in Alg. 1 (L7-9).

Once the mitigation process is complete, clients resume the detection phase to monitor potential future collusion.

VI. Efficiency Enhancement of *CoVFeFE*

Considering the constrained resources of client devices, *CoVFeFE* implements a scheme to reduce overhead while ensuring the efficiency of collusion detection and mitigation using adaptive server probing to maintain a proper size of pre-approved task table. Moreover, *CoVFeFE* deploys a mechanism called verification masquerading, which aims at masking the verification process with tasks that look like legitimate client computation requests.

A. Adaptive Server Probing

To balance the overhead of maintaining the pre-approved task table and the sufficiency of pre-approved tasks for verification and collusion mitigation, *CoVFeFE* employs an adaptive probing scheme that periodically sends the current computation task T_t, to multiple compute servers besides the current compute server, gathers all results returned, and performs majority voting on them to determine the pre-approved result R_t for task T_t. This scheme adopts a high probing frequency if any server has a shortage of potential tasks used for verification (*e.g.,* when *CoVFeFE* just starts up) or a low probing frequency when sufficient tasks are gathered. Note that the exact strategy to adjust probing frequency can be decided by users without interfering with the efficacy of *CoVFeFE*. We omit the details from this paper for clarity.

B. *Verification Masquerading*

To make a robust solution for detection and mitigation approach, *CoVFeFE* implements a task recommendation algorithm that selects the most efficient task from pre-approved task table to verify any server. We consider that this task must satisfy the following two rules:

R1: not previously sent to the same server. Tasks in the pre-approved task table previously received by servers are easily identifiable if they are sent to the same servers again (note that these tasks have been sent for probing or may already be used for verification). Thus, the most efficient task T_k must ensure the current compute server is not in the set of servers that have previously received T_k.

R2: presenting realistic features. Recognizing tasks used for verification can be achieved if the servers monitor the environmental contexts (*e.g.,* time of the day, weather, etc.) when receiving a new task and store the states (*e.g.,* road condition, location, etc.) of the recently received task, then profile them as a set of features, $\mathcal{F}$, to identify if the new task is realistic that presents very similar features compared to $\mathcal{F}$. For instance, the selected task should be gathered at a similar time on a rainy day in a nearby location of recent tasks. Otherwise, the malicious servers may suspect an "unrealistic" task as verification and behave honestly to evade it.

Therefore, we develop a task recommendation algorithm that selects the most efficient task, T_k, following R1 and R2, while minimizing the probability of selecting a poisoned task. Specifically, *CoVFeFE* sets a similarity threshold, and measures the similarity score between the client's current (*i.e.,* real world) task T_t and each task in the pre-approved task table in a lookup order from the earliest to most recently gathered tasks. The lookup will stop if any T_k is found with a similarity score higher or equal to the threshold and obeys R1. If no such task exists, the first one is picked. In this paper, we adopt normalized L1-norm [32] as an example for evaluation purposes, which is a common metric used to measure feature similarity of multimedia tasks (*e.g.,* video, audio, or text):

$$ss(k,t) = (1 - \frac{|\mathcal{F}_k - \mathcal{F}_t|}{F \times M}) \times 100, \tag{1}$$

where $|\cdot|$ returns the L1-norm distance between two vectors, F and M are the size and the value range of the feature vectors, respectively. We assume all tasks have the same feature vector size F and value range M – they can be achieved by zeroing the features (which are not applicable for a given task) and value normalization. The L1-norm can be replaced with other metrics that most fit client applications.

VII. Evaluation

In this section, we describe our experiment setup, metrics, and evaluation results in absence and presence of colluding servers at the network edge.

A. *Experimental Setup*

We use the network simulator ns-3 [33] to implement *CoVFeFE* and create experimental networks. The simulation runs on a laptop with an Intel i5 quad-core 2GHz CPU and 16GB memory.

1) Implementation Scope: We implement a client application running *CoVFeFE* as described in Section IV, and a server application, which processes computing requests. Servers return correct result if they are benign, random result if they are malicious, or same random result if they are colluding servers.

We compare *CoVFeFE* to two state-of-the-art replication-based verification solutions: (i) baseline replication-based [34] (used in BOINC system for result validation), which we refer to as *R_baseline*, and (ii) collusion-resistant replication-based [9], which we refer to as *R_Kupcu*. *R_baseline* performs a strict majority voting on results of every task and identifies malicious nodes if their results disagree with the majority vote. Otherwise, the execution of task and verification is regarded as failed. *R_Kupcu*, however, requires unanimous agreement on the results and repeats the verification until all results returned by servers are unanimous. It employs a budget scheme distributing fines to servers disagreeing with each other to identify malicious and colluding servers. We consider 50% of malicious servers implementing an evasive strategy, which check if task similarity score is larger than 80 (as threshold for potential most efficient tasks) as described in Section III. In this paper, we do not quantitatively compare our method to other collusion detection methods [12], [13] because they fail in the case that more than half of servers are colluding, otherwise their detection accuracy varies depending on the amount of outsourced computation tasks.

TABLE I: Evaluation setup; Numbers in **bold** are nominal (default) values used when the value of a parameter is fixed.

Parameter	Value Range
***CoVFeFE* Specifications**	
Interval of client changing servers	$\mathcal{N}(30, 2)$
Number of task features (F)	4
Value Range of task features	[0, 100] ($M = 100$)
Parameter of collusion start estimation (θ)	200
Network Topology	
Number of clients	20
Number of compute servers	20
Number of ISP routers	10
Percentage (%) of malicious servers	$\{20, ..., \mathbf{60}, ..., 100\}$
Bandwidth of all networking links	5Mbps
Propagation delay of all networking links	20ms

2) Network Topology: Our simulations use random pervasive edge network topologies, all consisting of 50 nodes, including 20 clients, 20 compute servers, and 10 ISP routers. The network topology is random, and the link type is P2P Ethernet.

We set the bandwidth and delay of all links to $5Mbps$ [35] and $20ms$ [36], respectively. To simulate computation outsourcing, each client sends a computation task to its current compute server or multiple servers per second. We model user mobility, where each client changes its direct connection to a random ISP router which reroutes new tasks to the closest edge server (refer to Table I for detailed parameters).

3) Performance Metrics: We consider the following metrics to evaluate the performance of *CoVFeFE*: (i) **Malicious server detection accuracy (accuracy)**: this metric is measured as the ratio of the number of correctly identified malicious servers to the total number of malicious servers. (ii) **Malicious server detection false positive ratio (FPR)**: is measured as the ratio of the number of "benign" servers which have been classified as "malicious" to the total number of benign servers. Note that accuracy and FPR are both important to evaluate the performance of any verification algorithm. (iii) **Detection delay**: we measure the delay in time (seconds) or in the total number of verification attempts elapsed from when the client first communicates with a malicious server as the current compute server to the time client identifies the server as "malicious" (infinite is no server). (iv) **Overhead**: is measured as the ratio of the number of messages sent for pre-approved tasks (*e.g.,* probing) and verifying servers (*e.g.,* verification) to the total number of computation tasks the client needs to outsource.

All metrics shown in this subsection are measured as an average of 100 simulation runs. Each simulation run uses a different network topology and runs for 3,000 seconds.

B. Performance in Absence of Collusion

We first study the performance of *CoVFeFE* in the absence of any collusion: servers can be benign or malicious at the beginning of simulation, where malicious servers do not collude but may deploy evasive strategies. We investigate the impact of the percentage of malicious servers on the detection accuracy of *CoVFeFE*, and then compare the detection accuracy, delay and overhead of our solution to *R_baseline* and *R_Kupcu*.

1) Impact of Percentage of Servers' Maliciousness: When the percentage of malicious servers increases, most replication-based verifiable computing schemes exhibit limitations such as overhead or long delays of detection. We investigate the impact of the percentage of malicious servers in the network on *CoVFeFE*'s performance in Fig. 4. We show, in Fig. 4a, that *CoVFeFE*'s accuracy increases at different rates depending on the two main factors: when the percentage of malicious servers increases, (i) clients tend to switch to and thus verify them more often, and (ii) the size of pre-approved task table increases more slowly, resulting in slower detection (using tasks with lower similarity scores). In fact, we show that for 80% malicious servers, the detection is faster by almost 20 verification steps compared to 20% malicious servers, when both cases achieve 0.8 accuracy. Moreover, in case 80% of the servers are malicious, detection accuracy is slower at the beginning of the simulation, where the pre-approved task table is filling up, then ramps up very quickly as the probability of selecting a malicious server is much higher than lower maliciousness levels. Note that while *CoVFeFE*'s accuracy increment steps may vary, *CoVFeFE* achieves 100% accuracy for almost all considered maliciousness levels[2], however, if the number of benign servers is less than two (the minimum number for majority voting), such as in the case of 100% maliciousness, the pre-approved task table remains empty, thus detection becomes impossible.

We plot in Fig. 4b, the cumulative distribution function (CDF) of malicious server detection delays in seconds as we vary the level of maliciousness in the network. This figure further emphasizes the effects of running *CoVFeFE* with fewer tasks in pre-approved task table on the performance, where the increase of maliciousness in the network causes an increase in detection delay of up to $10\times$ in the case of 80% maliciousness compared to 20%. Fewer tasks in the pre-approved task table result in failing to verify servers in any given verification cycle and/or choosing a task with a low similarity score which will be inefficient as we assume all our servers run evasive strategies to evade verification using these tasks. Note that *CoVFeFE*'s detection delay remains very low with little to no variation when maliciousness in the network is less than 60%.

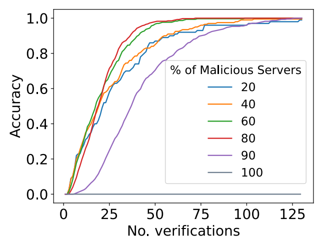
The similarity of tasks for verification is presented in Fig. 4c with box plots. This figure proves that the higher percentage of malicious servers would cause that less similar tasks are selected for verification, where the case of 80% maliciousness has 5% lower similarity scores (on average) and $2\times$ larger score variance than 20% maliciousness. Moreover, we observe that when the maliciousness is less than 60%, *CoVFeFE* can maintain a high level of task similarity (*i.e.,* minimum score$>$ 85) by employing adaptive probing frequency. However, as the percentage further increases, the similarity is sharply reduced to lower than 10 (on average) when the maliciousness is larger than 90%, because very few tasks are valid for verifying benign servers after all malicious servers are detected (*i.e.,* the tasks have been sent to benign servers when populating the pre-approved task table).

To summarize, *CoVFeFE* helps ensure high accuracy values and no false positives even when the percentage of malicious servers is close to 100% of compute servers in the network. This performance is achieved by selecting the best tasks previously stored and maintained.

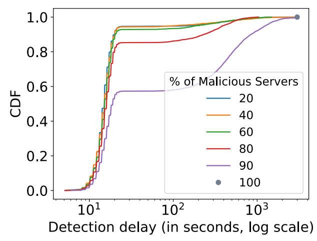
2) Comparing CoVFeFE *to State-of-the-Art Replication Based Verification Solutions:* We compare *CoVFeFE* performance to *R_baseline* and *R_Kupcu*, when 60% of the servers are malicious in the network (*i.e.,* under high server maliciousness).

Fig. 5 compares the (a) accuracy, (b) delay, and (c) overhead (similar trends were found for different maliciousness levels but are omitted from this paper for clarity). *CoVFeFE* outperforms *R_baseline* and *R_Kupcu* for almost all metrics and all parameters studied. For malicious server detection accuracy, while all three solutions achieve 100% accuracy by the end of

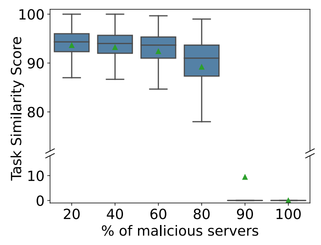
[2]false positive rates for all considered maliciousness levels are also zero (results were omitted for this no collusion case for space/clarity reasons)



(a) Accuracy

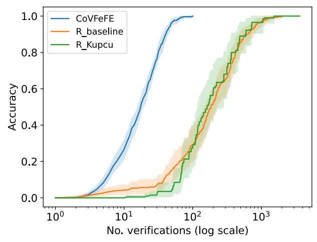


(b) Delay in seconds

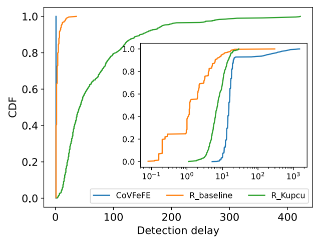


(c) Task similarity score

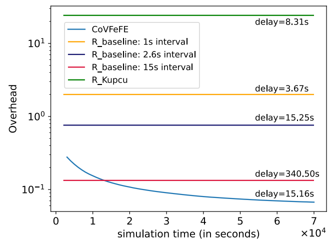
Fig. 4: Impact of the percentage of malicious servers on *CoVFeFE* (no collusion).



(a) Detection accuracy of malicious servers (all colluding)



(b) Delay in verification steps (outer) and in seconds (inner)



(c) Overhead (average delay shown as labels near lines)

Fig. 5: Comparing *CoVFeFE*, *R_baseline*, and *R_Kupcu* performance in the absence of collusion; 60% of servers are malicious (Error bars surrounding line-plots represent 95% confidence interval).

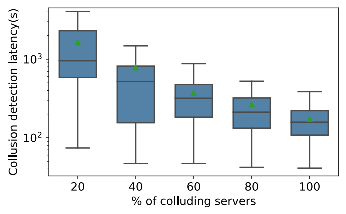
the simulation, however, *CoVFeFE* takes almost 10× smaller number of verification steps to identify all malicious servers as shown in Fig. 5a. Specifically, *CoVFeFE* identifies all malicious servers within 81 verification attempts, while *R_baseline* and *R_Kupcu* requires more than 1,000 to identify the same malicious servers. *R_Kupcu* has a slower accuracy increment than *R_baseline* at the beginning of simulation due to the all-matching result voting but catches up quickly because more servers are selected and verified. Note that Fig. 5a plots the 95% confidence interval (shade around the lines plots) emphasizing the lower variance of *CoVFeFE*'s accuracy measurements across all simulations. This accuracy gain is explained by the fact that *CoVFeFE* accurately detects any malicious server with a single verification step, while *R_baseline* or *R_Kupcu* takes up to 35 or 400 steps, respectively, to detect all malicious servers as shown in the outer figure of Fig. 5b. While *CoVFeFE* takes only one verification step, the detection delay in time can be longer than *R_baseline*/*R_Kupcu*'s delays, as shown in the inner figure of Fig. 5b, due to the adaptive verification frequency deployed by *CoVFeFE* to reduce the overhead. The frequency can be tuned if the client favors delay performance versus overhead, since *CoVFeFE* requires no more than one single verification step.

Delay-overhead trade-off is further highlighted in Fig. 5c, where we plot the average overhead of *CoVFeFE*, *R_baseline*, and *R_Kupcu* using different verification intervals ranging from 1 to 15 seconds (1s by default for *R_baseline* and *R_Kupcu*). While *CoVFeFE* uses adaptive probing and verification which reduces its overhead cost as the simulation advances (note that we run the simulation for 70,000s in this figure), *R_baseline*'s overhead remains constant over time and increases as the verification interval increases. Moreover, *R_Kupcu* introduces 100× more overhead due to repeated task outsourcing for determining correct results and extra communication with servers to request their budgets. Furthermore, we show that *CoVFeFE* achieves similar detection delay performance as *R_baseline* (2.6s verification interval), while having 8× less total overhead.

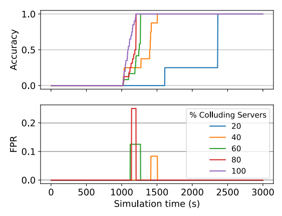
C. Impact of Server Collusion on Verification Performance

We evaluate the performance of *CoVFeFE* in the presence of various percentages of colluding servers. We simulate a scenario where the network starts with no collusion and server collusion is introduced at the time 1,000 seconds. Specifically, at time 1,000s, some benign servers turn malicious and colluding. Accordingly, the performance metrics are measured after collusion. We also compare *CoVFeFE* to both *R_baseline* and *R_Kupcu*, which shows pros and cons in the cases of varied collusion percentages.

1) CoVFeFE's Collusion Mitigation: We measure the collusion detection delay as the time elapsed between the start of the collusion (1,000s in our simulation), and the time each

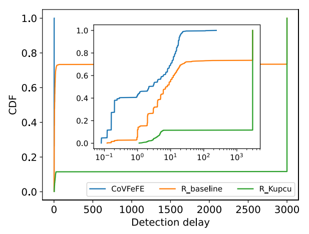


(a) Delay between collusion detection and start

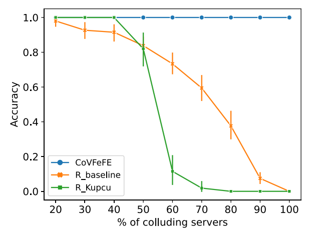


(b) Accuracy and FPR over time (a single simulation)

Fig. 6: Impact of the percentage of colluding servers in the network on *CoVFeFE*.



(a) Detection delay in verification steps (outer) and in seconds (inner); 60% collusion



(b) Detection accuracy as a function of the % of colluding servers in the network

Fig. 7: Comparing *CoVFeFE*, *R_baseline*, and *R_Kupcu* performance in the presence of collusion.

TABLE II: Cleaning-up the pre-approved task table: prior vs. post collusion mitigation. *Num*: number of poisoned tasks; *Percentage (%)*: percentage of poisoned tasks; *Rate*: number of poisoned tasks selected for verification per minute.

Percentage of Collusion	**Before Mitigation**			**After Mitigation**
	Num	Percentage	Rate	Percentage
20%	1.666	0.9%	0.173	0.0
40%	5.045	2.7%	0.185	0.0
60%	10.041	5.8%	0.217	0.0
80%	13.840	8.6%	0.243	0.0
100%	15.363	9.8%	0.278	0.0

collusion mitigation agent detects the presence of collusion (if any). We plot in Fig. 6a the distribution of collusion detection delays for different collusion levels. Expectedly, as the percentage of colluding servers increases in the network, *CoVFeFE*'s collusion mitigation detects collusion faster as the number of poisoned tasks in pre-approved task table increases with higher rates (*e.g.,* collusion detection is 10× faster on average when comparing 20% and 80% collusion). While slower collusion detection can delay the mitigation process, however in the cases of low collusion, poisoned tasks do not exceed 1% of the total tasks used for verification (as shown in Table II), thus its impact on detection accuracy is minor. We plot in Fig. 6b, the accuracy and FPR as a function of simulation time for one sample simulation (*i.e.,* not averaged across multiple runs) to highlight the impact of collusion detection on these metrics. We show that the longer the client takes to detect collusion, the slower it detects all colluding servers. As soon as *CoVFeFE* detects collusion, accuracy jumps to 100% and FPR returns to 0 as the mitigation process revokes the misclassification of benign servers. The collusion mitigation performs simultaneous and concurrent verification of all servers, which explains the sharp increase in accuracy and drop in FPR. Note that if the percentage of colluding servers is low in the network, *CoVFeFE* collusion detection is slower, however, this low collusion does not impact the accuracy of *CoVFeFE* which detects all malicious servers successively. The effectiveness of collusion mitigation is further presented in Table II showing that a higher percentage of colluding servers introduce a larger number of poisoned tasks into pre-approved task table, but they are successfully removed after mitigation. Note that the percentage of poisoned tasks in the table before mitigation indicates the maximal probability of collusion that misleads a client to accept erroneous results as correct. The percentage increases until the mitigation is triggered and the poisoned tasks are removed. *CoVFeFE*'s novelty is also highlighted as the percentage of colluding servers increases in the network and where most existing collusion defense strategies fail.

2) Collusion Mitigation of CoVFeFE *vs. State-of-the-Art:* Next, we compare *CoVFeFE* to *R_baseline* and *R_Kupcu* in the cases of varying percentages (*i.e.,* from 20% to 100%) of colluding servers. As shown in Fig. 7, *CoVFeFE* outperforms both considered solutions, achieving 100% accuracy and faster verification latency (both in the number of verification steps and in seconds) in detecting malicious servers as the percentage of colluding servers increases in the network. We plot in Fig. 7a, the CDF of detection delay in the number of verification steps (outer figure) and in time (inner figure), when 60% of the servers are colluding. *CoVFeFE* successively detects colluding servers within one or two verification steps and low latency (ranged from 0.1 to 20s), while *R_baseline* and *R_Kupcu* may take more than 50× verification steps and cannot guarantee to detect all colluding servers. This latency registered by *R_baseline* and *R_Kupcu* also have a direct impact on their high false positive rates as emphasized in Fig. 7b. Particularly, the all-matching result voting mechanism of *R_Kupcu* is a double-edged sword, which reduces the probability of selecting all colluding servers when the number of benign servers is larger than colluding servers, but this mechanism amplifies the influence of collusion when colluding servers are the majority. In contrast, *CoVFeFE* utilizes the pre-approved tasks to detect and mitigate collusion in the whole spectrum of percentages of colluding servers.

VIII. Conclusion

In this paper, we have proposed *CoVFeFE*, a collusion-resilient verification framework for resource-limited edge devices. *CoVFeFE* provides can detect collusion if the majority of servers in the network are colluding. Then, a heuristic mitigation mechanism is performed to identify colluding servers, remove tasks with erroneous results, and revoke the misclassification of benign servers. The results of evaluations show that *CoVFeFE* can achieve fast detection and close to 100% accuracy of identifying colluding servers. Moreover, *CoVFeFE* demonstrates high robustness against evasive strategies and can reduce communication overhead of client compared to the other two solutions. In the future, we will improve *CoVFeFE* against more types of collusion, such as servers probabilistically colluding or only colluding when they ensure the win of result consensus.

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] B. Ji, X. Zhang, S. Mumtaz, C. Han, C. Li, H. Wen, and D. Wang, "Survey on the internet of vehicles: Network architectures and applications," *IEEE Communications Standards Magazine*, vol. 4, no. 1, pp. 34–41, 2020.

[2] Z. Lv, D. Chen, and Q. Wang, "Diversified technologies in internet of vehicles under intelligent edge computing," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 4, pp. 2048–2059, 2020.

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

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

[5] 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.

[6] 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*, 2011, pp. 445–454.

[7] 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.

[8] Z. Shan, K. Ren, M. Blanton, and C. Wang, "Practical secure computation outsourcing: A survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 2, pp. 1–40, 2018.

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

[10] 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*, 2017, pp. 211–227.

[11] 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)*. IEEE, 2005, pp. 31–38.

[12] 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.

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

[14] 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)*. IEEE, 2010, pp. 1–12.

[15] 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*, 2013, pp. 863–874.

[16] 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*, 2016, pp. 119–128.

[17] 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*. IEEE, 2015, pp. 38–54.

[18] 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*, 2018, pp. 302–311.

[19] 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*, 2008, pp. 85–90.

[20] J. van den Hooff, M. F. Kaashoek, and N. Zeldovich, "Versum: Verifiable computations over large public logs," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1304–1316.

[21] Z. Chen, Y. Tian, J. Xiong, C. Peng, and J. Ma, "Towards reducing delegation overhead in replication-based verification: An incentive-compatible rational delegation computing scheme," *Information Sciences*, vol. 568, pp. 286–316, 2021.

[22] Z. Wang, S.-C. S. Cheung, and Y. Luo, "Information-theoretic secure multi-party computation with collusion deterrence," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 4, pp. 980–995, 2016.

[23] 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.

[24] 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.

[25] 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*. IEEE, 2009, pp. 213–218.

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

[27] 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.

[28] 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.

[29] H. Zeyu, X. Geming, W. Zhaohang, and Y. Sen, "Survey on edge computing security," in *2020 International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE)*. IEEE, 2020, pp. 96–105.

[30] T. Grossman and A. Wool, "Computational experience with approximation algorithms for the set covering problem," *European journal of operational research*, vol. 101, no. 1, pp. 81–92, 1997.

[31] P. Slavık, "A tight analysis of the greedy algorithm for set cover," *Journal of Algorithms*, vol. 25, no. 2, pp. 237–254, 1997.

[32] J. Yu, J. Amores, N. Sebe, and Q. Tian, "A new study on distance metrics as similarity measurement," in *2006 IEEE International Conference on Multimedia and Expo*. IEEE, 2006, pp. 533–536.

[33] G. F. Riley and T. R. Henderson, "The ns-3 network simulator," in *Modeling and tools for network simulation*. Springer, 2010, pp. 15–34.

[34] D. P. Anderson, "Boinc: a platform for volunteer computing," *Journal of Grid Computing*, vol. 18, no. 1, pp. 99–122, 2020.

[35] J. Kim, B.-H. Oh, and J. Lee, "Receive buffer based path management for mptcp in heterogeneous networks," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 2017, pp. 648–651.

[36] P. D. Barnes Jr, J. M. Brase, T. W. Canales, M. M. Damante, M. A. Horsley, D. R. Jefferson, and R. A. Soltz, "A benchmark model for parallel ns3," in *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*, 2012, pp. 375–377.