Secure Remote Execution of Sequential Computations

Ghassan O. Karame, Mario Strasser, and Srdjan Čapkun

ETH Zurich, Switzerland
karameg@inf.ethz.ch, strasser@tik.ee.ethz.ch, capkuns@inf.ethz.ch

Abstract. We describe a scheme that secures the remote execution of sequential computations in grid-computing scenarios. To the best of our knowledge, this is the first contribution that addresses the security of generic sequential computations. By dividing sequential tasks into smaller subtasks and permuting them among participants, we show that our scheme facilitates the insertion of selective redundancy and/or pre-computed functions (*ringers*) that are indistinguishable from other computations. We analyze the security of this proposal and we demonstrate that our scheme enables the detection of individual and colluding malicious participants. In addition, we show that our scheme can be equally used to securely track the progress of remote execution. We further investigate the damages introduced by possible chaining of errors within the remote execution and we discuss recovery mechanisms to counter these challenges. We validate our findings both analytically and empirically via simulations.

1 Introduction

The recent years have witnessed an increasing development of distributed computing platforms that leverage on the idle computing power of volunteer hosts (SETI@home [1], Distributed.net [2]) to run computationally expensive tasks on behalf of academic research groups, industry labs and even individual clients.

However, the open and untrusted environment in which these distributed computations are performed tends to cast suspicion on the results returned by the participants of these platforms. For instance, it is often the case that a participant (the task supervisor), that wishes to run a computationally expensive task, recruits several other nodes that agree to execute the task on its behalf in exchange of some form of reward (e.g., monetary remuneration). In typical scenarios, the supervisor has a limited computational capability and, therefore, cannot afford to check the computations itself.

One important challenge to address therefore is designing secure and efficient mechanisms to check the correctness of remote computations with minimum verification overhead. Although there is a large body of studies that address the security of distributed systems, there are few contributions that deal with the security of remote computations (e.g., [4], [5], [7]). With the exception of reputation/voting-based solutions [15], [16], [17], [18], an even smaller subset of these works proposes practical and efficient solutions to securely verify the execution of *generic* classes of sequential computations [4]. The latter class mainly refers to those sequential functions used in practice that are not necessarily repetitive, yet too computationally expensive to be run on the supervisor's machine (e.g., extensive simulations). The main challenge in securing these functions resides in the direct relation between intermediate results which might limit

the number of viable countermeasures that can be used to prevent possible misbehavior during execution. Namely, the sequential nature of these computations prevents the use of selective embedded "security checks" (e.g., pre-computed functions or ringers [4]) within the flow of computations. Currently, the only known method to efficiently verify (with low overhead) remote sequential computations is by relying on sample redundancy (e.g., [5]). However, this method is not entirely secure against collusion between malicious entities.

In this work, we propose a probabilistic scheme that secures *sequential* computations, in spite of collusion among malicious hosts. To the best of our knowledge, this is the first contribution that addresses this problem and that demonstrates that the ringer scheme – initially proposed by Golle *et al.* in [4] to secure parallel computations – can be equally used to efficiently secure their sequential counterpart. Another important aim of our work is to provide a practical framework that enables the supervisor to efficiently recover from possible misbehavior in the execution of its tasks. The major challenge in executing sequential computations is that a single erroneous intermediate computation renders the results of the entire sequential task useless to the supervisor. While several contributions rely on the use of selective sampling to check the credibility of remote hosts, as far as we are aware, no prior work has tackled the problem of efficiently recovering from undetected erroneous computations. In this paper, we address this issue and we show that by capitalizing on the high detection rate of our scheme, the correctness of the sequential functions can be ensured with a modest overhead in execution time. Our contributions in this work are summarized as follows:

- We show that by breaking each task into smaller pieces and by permuting the resulting subtasks among different participants, the supervisor can efficiently make use of indistinguishable pre-computed functions (or "ringers" [4]), combined with selective redundancy to probabilistically secure and track the remote executions of its sequential tasks. We validate our findings both analytically and empirically via extensive simulations.
- We evaluate the robustness of our proposal in practical settings and we discuss efficient solutions that enable the supervisor to recover from possible tampering with the execution of its tasks. We further analyze the resilience of our scheme to chaining of errors caused by incorrect intermediate results.

The remainder of this paper is organized as follows. In Section 2, we briefly overview the related work in the area. In Section 3, we present our solution and we analyze its resilience against individual and colluding malicious participants. In Section 4, we analyze efficient solutions that enable the supervisor to remotely ensure the correctness of its executing tasks and to counter possible chaining of errors caused by intermediate incorrect computations. In Section 5, we discuss further insights with respect to our scheme and we conclude the paper in Section 6.

2   Related Work

A comprehensive survey in the area of result-checking and self-correcting programs can be found in [6]. Golle *et al.* propose in [4] to secure a specific class of parallel problems:

inverse one-way functions (IOWF), where participants are required to compute the pre-images of several one-way functions. Their solution relies on inserting special-purpose *ringers* in each task. These correspond to the pre-computed images of randomly chosen elements. In [5], Szajda *et al.* extend this solution to secure non-sequential optimization problems and Monte Carlo simulations. In addition, the authors briefly propose assigning selective redundancy to secure repetitive sequential functions. Golle *et al.* further propose in [7] a security framework for commercial distributed computations that equally relies on selective redundancy. They further analyze the impact of varying the distribution that dictates the application of redundancy among participants. Similarly, Du *et al.* discuss in [10] a scheme that uses sampling techniques along with a Merkle-tree based commitment technique to secure non-sequential distributed problems in grid computing. Goodrich *et al.* discuss in [11] mechanisms to duplicate tasks among participants in grid computing applications as a mean to efficiently counter collusion among malicious participants.

Sanders *et al.* suggest computing with encrypted functions to provide security for mobile agents [8]. The major drawback of this proposal lies, however, in the fact that it might be very difficult to create encryption functions that result in correct executable procedures. Vigna *et al.* propose a mechanism based on execution tracing to protect the execution of mobile agents [9]. In [12], Yang *et al.* describe a method that uses the program counter values to monitor remotely executing computations. The supervisor, then, checks sample computations in order to detect possible misbehavior.

Several other proposals suggest the use of tamper-proof hardware/software [13] to prevent possible tampering with the results of the computations. However, tamper-proof software/hardware comes at high implementation costs nowadays. Another solution to the problem we consider in this paper would be for the supervisor to send an obfuscated executable code; however, existing code obfuscation techniques can only result in modest, best-effort efficacy nowadays [14].

3   Secure Verification and Tracking of Remote Execution

In this section, we describe our scheme that enables secure verification and tracking of the execution of sequential computations. and we analyze its resilience against individual and colluding malicious participants.

3.1   System and Attacker Model

Our computing platform consists of a *supervisor* interested in remotely running *several* sequential *tasks* (e.g., exhaustive simulations) on the machines of multiple *participants*. We assume that participants have considerable incentives to execute the tasks on behalf of the supervisor (e.g., in exchange of recognition, monetary reward). Detailed analysis of such incentive mechanisms is beyond the scope of this paper.

Individual tasks are independent of each others. However, each task can be divided into smaller sub-tasks that can be solved in a reasonable amount of CPU time. That is, owing to its sequential nature, a task function $f(x) = (g \circ h \circ j \circ k...)(x)$ can be divided into the sequential individual components $g()$, $h()$, etc.. The supervisor can directly

extract these sub-functions from the original code (e.g., subroutines) or, alternatively, the supervisor can use available tools that decompose a code piece into smaller sub-routines according to its control flow structure. Note that our analysis equally covers the case of multiple inputs per function. We further assume secure communication be-tween the supervisor and the participants and we abstract away the peculiarities of the communication channel, such as delays, congestion, jitter, etc..

We assume the existence of one or multiple malicious participants. These partic-ipants possess technical skills by which they can efficiently analyze, decompile, and modify executable code as necessary. Furthermore, these participants have knowledge of the measures used by the supervisor to prevent potential tampering with the compu-tations.

In our analysis, we assume that malicious participants are motivated to cheat in order to obtain credit without performing all of their assigned work. For instance, a selfish participant might only execute 50% of its assigned job and defect from running the rest of its tasks. In the mean time, it might decide to use its resources to run *another* task by a different participant to increase its benefit in the network. Here, two or more malicious participants might collude to increase their chances of not being detected.

Similar to [4], [5], [10], we do not consider the case where a malicious participant cheats only once in an attempt to disturb the computations (for personal gain or to achieve some e.g., political goal). To the best of our knowledge, with the exception of re-checking every subtask or relying on tamper-resistant software, little can be done by the supervisor to counter this threat. In this work, we assume, however, that untrusted participants are motivated to cheat in a considerable number of subtasks (e.g., $> 10\%$).

3.2   Securing the Remote Execution of Sequential Tasks

We assume that the supervisor is interested in executing N distinct and independent tasks on the remote machines of P different participants. Our scheme described here-after can be applied to arbitrarily chosen N and P such that $N \leq P$. Our scheme for securing the remote execution of N tasks unfolds as follows:

1. The supervisor first divides each task into M smaller subtasks. This can be achieved by decomposing the composite function of the task into its smaller functional com-ponents. Alternatively, the subtasks could be obtained by extracting the control flow structure of the task function. Note, here, that the subtasks do not necessarily have to be of the same computational length. The supervisor then proceeds to running the N tasks on the machines of P participants in M consecutive rounds as shown in Figure 1.
2. In round i, the supervisor picks an idle participant and according to some proba-bility, it decides to verify its credibility by inserting "security checks" within the computations; otherwise it randomly assigns to the participant a pending subtask. In our scheme, the supervisor evaluates the credibility of a participant by request-ing that it runs a subtask whose results are already known to the supervisor (a *ringer*) or by redundantly assigning the same subtask to another participant. We show later that this process is transparent to participants and that they cannot distin-guish whether they are running a legitimate subtask or whether their work is being

checked by the supervisor. Round i ends when all N participants are assigned to a job. In this way, the supervisor checks the work of several participants in each round.

3. At the beginning of round $i + 1$, the supervisor collects the results reported by the participants and checks the correctness of the ringers and the redundantly assigned subtasks. If these verifications pass, the supervisor re-permutes the next logical subtasks (since each task is sequential) among the participants that are considered to be honest while using the corresponding outputs of the last round as inputs to the subtasks of this round. Otherwise, if the supervisor detects inconsistent results, it stops interacting with the malicious node[1].

4. The supervisor repeats Steps *2)* and *3)* until all subtasks are executed.

Before analyzing the security of our proposal, we first describe the main rationale behind our approach.

The Main Intuition: The main intuition behind our proposal relies on the fact that by randomly permuting the execution of N different tasks, the supervisor gains a considerable advantage in securing their remote execution when compared to the scenario that features a single executing task.

The major challenge in securing sequential tasks resides in the fact that the output of one subtask is the input to the next consecutive subtask. This limits the number of viable "tricks" that the supervisor can make use to prevent participants from cheating. Note that redundant computations might not solely achieve a desired level of security in scenarios where collusion between malicious participants is possible. In this case, the supervisor might have to accept the burden of checking sample computations itself.

By permuting N different tasks among P participants (Figure 1), the sequential property of the tasks becomes largely transparent to the participants. This enables the supervisor to embed *pre-computed* indistinguishable "checks", *ringers*, within the subtasks assigned
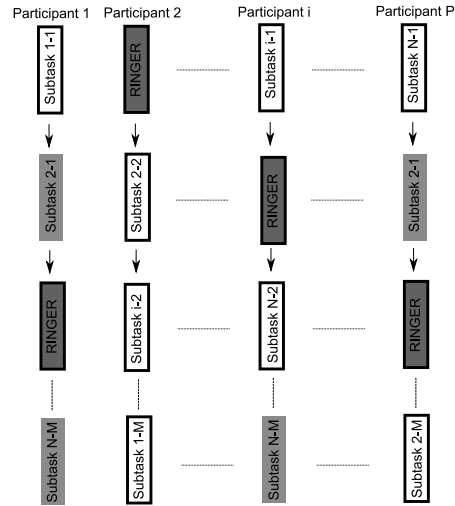
Fig. 1. N tasks assigned to P participants. Each box denotes a subtask. The notation "Subtask X-Y" refers to subtask # Y of task # X. The dark gray boxes refer to "ringer" subtasks (pre-computed subtasks whose result is known to the supervisor) and the boxes with shaded borders refer to redundantly assigned subtasks.

---

[1] It is out of the scope of this paper to discuss mechanisms to isolate malicious participants from the network. The supervisor can make use of cryptographic proofs to inform a central authority of its opinion about participants (e.g., [20]).

to participants. Ringers were first proposed in [4] to secure a special class of non-sequential inverse one-way functions. Provided that ringers are indistinguishable to participants from other subtasks, they provide a strong form of probabilistic security to remote computations. In fact, since ringers can be directly verified by the supervisor, they are resilient to collusion between malicious participants and do not incur computational burden on the supervisor, which makes them ideal for real-time grid-computing applications.
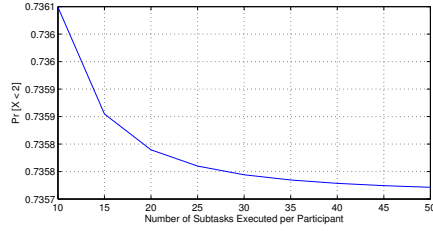
Generating Indistinguishable Ringers: In our scheme, ringers could be constructed from previously executed tasks. We point that a poor choice of ringer candidates might allow malicious colluding participants to abuse our scheme. If ringer subtasks were completely independent from each other, then malicious participants can collude and analyze the input/output of each subtask they execute: if the input of one subtask was provided previously by another participant, colluding participants could identify that the subtask in question is not a ringer subtask and that it pertains to a genuine task. As mentioned previously, if the supervisor uses previously executed tasks (or any "reasonable" executable code) as ringer tasks, the corresponding ringer subtasks are likely to share comparable computational workload when compared to other subtasks while inherently exhibiting a similar relation between its inputs and outputs. This would indeed ensure that the inserted ringers cannot be distinguished from other subtasks in the system, in spite of collusion among malicious participants.

Let the random variable X denote the number of subtasks pertaining to the same task run by the same participant. In our scheme, the probability that a participant runs *at most* one subtask pertaining to each task ($X \leq 1$) in M execution rounds is:

$$P[X \leq 1] = \sum_{i=0}^{i=1} \binom{M}{i} \left(\frac{1}{N}\right)^i \left(1 - \frac{1}{N}\right)^{M-i}, \qquad (1)$$

where N is the number of required tasks to be run. In Figure 2, we plot $P[X \leq 1]$ with respect to different values of N. Equation 1 suggests that by randomly permuting tasks and their corresponding subtasks among the participants in the system, the probability that a participant cannot distinguish which task it is actually running in each execution round is satisfactorily large (> 0.7) given a reasonable number of participants and tasks in the system ($N \leq P$). This suggests that it is highly unlikely for participants to establish a correlation between different tasks. Since participants do not know the number of tasks in the system, the supervisor can take advantage of this fact and probabilistically requests that participants run *ringer subtasks*. Recall that once the supervisor detects a malicious participant, it stops interacting with the detected node.

To ensure an acceptable level of security, the number of embedded ringers should be considerable when compared to the number of subtasks executed per participant (typically $> 20\%$). However, depending on the nature of the supervisor tasks (e.g., repetitive tasks such as in the GIMPS project [3]), finding a large number of ringers might be prohibitively expensive [5]. *To address this issue, the supervisor can combine the use of ringers along with sample redundancy.* For example, if the supervisor needs to randomly check 40% of a participant's work and possesses a number of ringer candidates that only account for 20% of the number of subtasks per participant, then the supervi-

M	P_C	P_R	P_P	P_M	Overhead	P
10	0.7	0.15	0.15	0.2	30%	**0.88**
10	0.9	0.15	0.15	0.2	30%	**0.94**
20	0.5	0.15	0.15	0.2	30%	**0.95**
20	0.5	0.25	0.25	0.2	50%	**0.99**
20	0.5	0.15	0.15	0.4	30%	**0.92**
20	0.5	0.2	0.1	0.4	30%	**0.94**
50	0.5	0.2	0.2	0.2	40%	**0.999**

Fig. 2. Probability that a participant runs at most one subtask pertaining to each task. Here, we assume that the number of subtasks per task is equal to the number of available tasks.
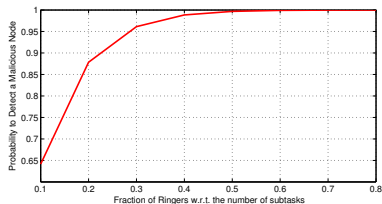
Fig. 3. Probability of detecting malicious participants with respect to different input parameters. M is the the number of subtasks per task.

sor can redundantly assign 20% of the subtasks to achieve its desired level of security. We show in the following subsection that this combination remains, to a large extent, resilient to collusion between malicious participants.

Coordination and Synchronization Overhead: Similar to [5], our scheme incurs an increased workload on the supervisor since it has to coordinate the permutation of subtasks among participants "on the fly". Given a modest number of tasks and participants (typically < 50), this process can be made very efficient in real-time settings through an automated interface that coordinates the computations between participants. In Section 3.4, we discuss in greater details the benefits of this solution. Furthermore, subtask permutation might require loose synchronization among the participants' machines to efficiently manage the time cost of computations. However, since the supervisor will presumably pick those participants with considerable computing performance, synchronization costs are likely to be satisfactorily negligible. Note that the synchronization costs in our scheme can be further minimized if the supervisor makes use of an optimal assignment of subtasks to participants that optimizes the total execution time. In Section 4.1, we show via simulations that our proposed scheme performs well even in the case when the subtasks are assigned to participants at random. Note that our scheme does not induce any additional communication overhead on the supervisor; the supervisor would have the same communication burden even if it would send the full tasks to the participants.

3.3   Security Analysis

Throughout our analysis, we make the worst case assumption that malicious participants will *always* collude with each other. That is, if the supervisor redundantly assigns subtasks to two or more malicious nodes, we assume that they will coordinate their results to avoid being detected. We further assume that the distribution of ringers and redundant subtasks is uniform per subtask. This suggests that the best strategy of a malicious participant to decrease the likelihood that it gets caught is to equally follow a uniform distribution of cheating per subtask. In case the supervisor detects inconsistencies between the results of the redundantly assigned subtasks without being able to

Fig. 4. Probability to detect cheating versus the fraction of inserted ringers. Here, the number of subtasks is 20. The network contains 20% malicious participants that cheat in 50% of their assigned subtasks.
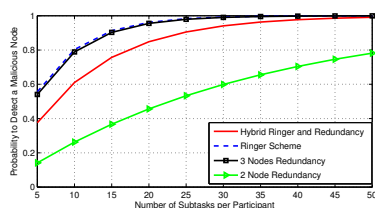
Fig. 5. Comparison of probabilistic schemes for securing sequential computations. Here, we assume that the supervisor randomly checks 30% of the subtasks. We consider scenarios where 20% of the participants are malicious and randomly cheat in 50% of the subtasks.

determine the genuine subtasks' outcome (i.e., with the absence of majority consensus), we assume in our analysis that it will proceed to re-run the corresponding subtasks (Section 4) to increase its confidence in their correctness. We do acknowledge that there might exist methods to redundantly assign subtasks among participants that might perform better than uniform distributions [11]; however, we show in this work that even the simplistic random redundancy achieves a satisfying level of security when combined with ringers. We point that the supervisor is not likely to benefit by choosing biased distributions in verifying the job of participants since these latter might change their cheating strategies accordingly in order to increase their gain. We, nevertheless, briefly discuss in Section 5 the implications of choosing non-uniform distributions on the performance of our scheme.

On the other hand, in our scheme, assuming that a malicious participant cheats with probability P_C per subtask and that the supervisor inserts an indistinguishable ringer with probability P_R per subtask or redundantly assigns the same subtask to another node with probability P_P, then the probability P_T of catching a malicious participant in a single subtask is given by: $P_T = P_C(P_R + P_P(1 - P_M))$.
Here, subtask redundancy will only be beneficial when the supervisor picks an honest participant (with probability $(1 - P_M)$, where P_M is the fraction of malicious nodes in the network).

Assuming that each participant is required to execute at least M different subtasks, then the probability that the supervisor catches potential misbehavior in our scheme is computed as follows:

$$P = 1 - (1 - P_T)^M = 1 - (1 - P_C(P_R + P_P(1 - P_M)))^M. \qquad (2)$$

Table 3 shows the probability of detecting a malicious participant with respect to various input parameters. As P_C increases, the probability to detect a malicious participant equally increases; in the case where malicious participants always cheat in their subtasks, the probability that they get detected in our scheme approaches 1. Note that the higher is the number of inserted ringers and/or redundant computations, the higher is the level of assurance in the correctness of the computations and the higher is the

induced time overhead of our scheme. In other words, the time required for the computations to complete increases by the amount of time needed to execute all the inserted ringers and redundant subtasks. Nevertheless, as shown in Figure 4, our scheme can ensure an acceptable security level even when the overhead resulting from inserting ringers and redundant computations is as low as 30%.

In Figure 5, we analytically compare the security of ringer-based schemes with solutions based on selective redundancy with respect to the number of subtasks per participant. Our findings suggest that ringer-based solutions considerably outperform redundancy-based schemes in scenarios where malicious participants collude. For instance, when the supervisor selectively checks 30% of the computations performed by a participant, by solely relying on ringers, the probability of detecting a malicious participant is as high as selectively assigning redundant subtasks to 3 different participants. Furthermore, even in hybrid solutions – equally combining ringer-schemes and selective redundancy – the probability of detecting possible cheating is at least twice as high when compared to redundancy-based solutions.

3.4 Secure Tracking of Remote Execution

As described earlier, the starting time of the computations required for each subtask is known to the supervisor. Since subtask permutation and verification is performed on the fly, each participant depends on the supervisor to acquire its newly assigned subtask function along with its corresponding input. This equally allows the supervisor to know the time each subtask took to complete; as shown in Figure 1, once a participant completes the execution of a subtask, it reports the result back to the supervisor, which then verifies the reported result (in case the subtask corresponded to a ringer or it was redundantly assigned) and sends back to the participant another subtask to execute.

Given the random permutation, participants cannot guess beforehand which subtask they are going to execute. This suggests that the starting time of each individual computation is solely dependent on the supervisor. Malicious participants could, still, try to trick the supervisor by reporting incorrect results. As explained previously, such misbehavior will be detected by the supervisor with high probability. Malicious participants could equally delay reporting the results to the supervisor. However, given that the participants are rational players aiming at maximizing their benefit in the network (e.g., claim credit for their work), participants are unlikely to benefit from this strategy.

We conclude that our scheme enables probabilistically secure tracking of remote execution of the supervisor tasks at a subtask granularity. That is, the supervisor knows at anytime during the execution process the number of executed (and pending) subtasks in each task.

4 Ensuring the Correctness of the Sequential Tasks

In the previous section, we analyzed the performance of our scheme in detecting malicious participants. In what follows, we evaluate other practical aspects and limitations related to our proposal.

From a practical perspective, one potential limitation of our scheme lies in the fact that a single undetected intermediate result renders the entire's task output erroneous; this limitation applies to all solutions that target sequential computations. Furthermore, due to task-permutation, one malicious participant is likely to cheat in several subtasks pertaining to different tasks before getting detected. This might result in the deterioration of the system's efficiency. In what follows, we analyze these limitations and we discuss efficient solutions to counter their impact.

Assuming a perfect random permutation of tasks among participants such that each participant runs at most a single subtask of each task and given that a malicious participant cheats with probability P_C per subtask, then the maximum fraction of incorrect tasks in the system T is computed as follows:

$$T = P_C(1 - (P_R + P_P(1 - P_M))), \tag{3}$$

where P_R and P_P denote the probability of inserting a ringer and redundant assignment per subtask respectively and P_M is the initial fraction of malicious nodes in the network.

The most intuitive solution to limit the damage that a malicious participant can cause is by increasing the number of checks (e.g., ringers, redundant assignment). This also comes at the benefit of increasing the confidence C in the correctness of the subtasks' results that were previously computed by a participant. C is computed as follows:

$$C = 1 - T = 1 - (P_C(1 - (P_R + P_P(1 - P_M)))). \tag{4}$$

The drawback of an increased confidence C is a prolonged task execution time; once the supervisor detects that a participant is malicious, it has to re-run all the tasks that the malicious participant participated in executing. This is needed to prevent a possible chaining of errors due to the sequential property of the tasks.

In what follows, we analyze this additional cost as a function of the confidence in the correctness of the tasks. More precisely, we compute the number of subtask executions (i.e., rounds) that are required to correctly complete a task, given an initial fraction of malicious nodes and the verification overhead per task (i.e., P_R and P_P).

Recall that detected malicious participants are isolated and no longer considered for subtask execution (in Section 5, we discuss the case where the supervisor might not be able to fully isolate malicious nodes). The fraction of malicious participants therefore decreases over time upon every detection. More specifically, let P_{M_i} be the fraction of malicious participants in the i-th round. Given that all N tasks can be executed in parallel (i.e., that $P \geq N$), the expected number of malicious participants that are newly detected in the i-th round is $N P_{M_i} P_C (P_R + P_P(1 - P_{M_i}))$. That is, after $i - 1$ rounds, a total of Q_{i-1} malicious nodes have been removed:

$$Q_{i-1} = \sum_{j=0}^{i-1} N P_{M_j} P_C (P_R + P_P(1 - P_{M_j})). \tag{5}$$

Hence, the expected fraction of malicious nodes in the i-th round is:

$$P_{M_i} = \frac{P_M P - Q_{i-1}}{P - Q_{i-1}} = \frac{P_M P - \sum_{j=0}^{i-1} N P_{M_j} P_C (P_R + P_P(1 - P_{M_j}))}{P - \sum_{j=0}^{i-1} N P_{M_j} P_C (P_R + P_P(1 - P_{M_j}))}. \tag{6}$$

The expected number of rounds i^* after which all malicious participants have been identified and isolated can now be derived by (numerically) solving the equation $\lfloor P_M P - \sum_{j=0}^{i-1} N P_{M_j} P_C (P_R + P_P (1 - P_{M_j})) \rfloor = 0$ for i. The *upper bound* $\overline{i^*}$ on the expected number of rounds to complete a task is computed as follows:

$$\overline{i^*} = i^* + M.$$

This suggests that after i^* rounds all the malicious nodes are eliminated; the final result can then be computed in M rounds. Figure 7 depicts this upper bound as a function of the initial fraction of malicious nodes (P_M) and the verification overhead (P_R and P_P).

Fig. 6. Markov chain representing a task execution for $M = 3$. The states $s_{i,j}$ represent the number j of subtasks that have been executed after i rounds. We omit some labels for purposes of better clarity.

To obtain a more accurate estimate of the expected running time of a task as a function of the confidence in its correctness, we model its execution with an absorbing Markov chain [19] (refer to Figure 6). In this chain, the states $s_{i,j}$ represent the number j of subtasks that have already been executed after i rounds and each state transition accounts for one subtask execution. We further associate to each state $s_{i,j}$ a random variable $Y_{i,j}$ that represents the number of required state transitions (i.e., rounds) to reach an absorbing state from $s_{i,j}$; once an absorbing state is reached, the processing of a task terminates.
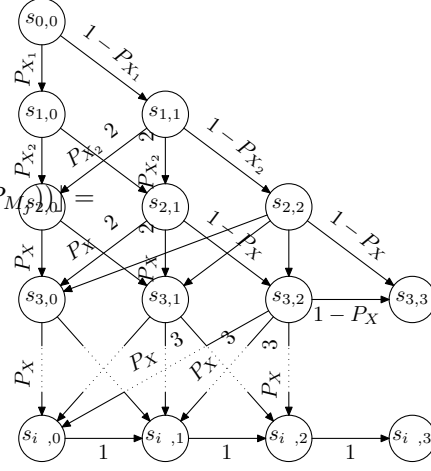
For example, consider the situation represented by state $s_{2,1}$. Here, after two rounds only the result of the first subtask has been accepted. As depicted in Figure 6, there essentially exists two possibilities on how the processing of the task can proceed:

1. If neither the currently used participant nor the participant that computed the previous subtask are identified as malicious in this round, both results are considered valid. This new state (two subtasks completed after three rounds) is represented by state $s_{3,2}$. From the analysis in Section 3.3, it follows that in round i a participant is identified as malicious with probability P_{Z_i}:

$$P_{Z_i} = P_{M_i} P_C (P_R + P_P (1 - P_{M_i})). \tag{7}$$

   The probability that we proceed to state $s_{3,2}$ (i.e., that none of the $j + 1 = 2$ participants we used so far has been identified as malicious) is thus given by $(1 - P_{Z_i})^{(j+1)} = (1 - P_{Z_i})^2$.
2. If at least one of the nodes that participated in the execution of the completed subtasks is identified as malicious, all subtasks subsequent to those executed by a malicious participant must be discarded. In state $s_{2,1}$, the likelihood of this event is
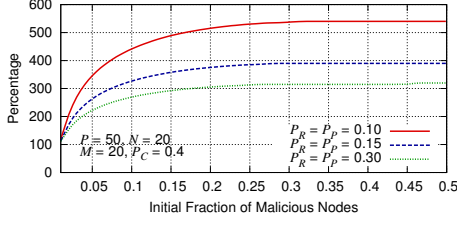
Fig. 7. Upper bound on the expected number of rounds to complete a task as a function of the initial fraction of malicious nodes and the verification overhead per task.
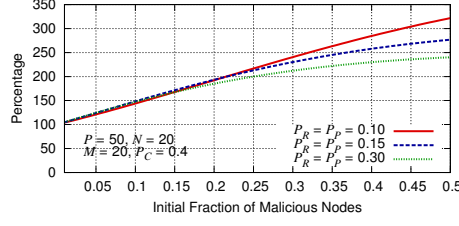
Fig. 8. Expected number of rounds to complete a task as a function of the initial fraction of malicious nodes and the verification overhead per task.

$1 - (1 - P_{Z_i})^{(j+1)} = 1 - (1 - P_{Z_i})^2$. The actual number of subtasks that must be discarded depends on the execution trace of the process; in the best case, only the current execution must be repeated. In the worst case, the first subtask was already run on a malicious participant and, due to the sequential property of the function, the entire task execution must be restarted. Given that subtasks are assigned to participants uniformly at random, any of these events is equally likely. In state $s_{2,1}$ this means that either the current or the current plus the already completed subtask must be discarded with probability $\frac{1}{2}(1 - (1 - P_{Z_i})^2)$ each. The former case is represented by the state $s_{3,1}$ the latter by the state $s_{3,0}$.

Based on the above observations, we derive an (recursive) expression for the expected number of required state transitions (i.e., rounds) to reach an absorbing state (i.e., to terminate the task execution) from the current state $s_{2,1}$. Let $P_{X_{i,j}} = 1 - (1 - P_{Z_i})^{(j+1)}$ be the probability that at least one subtask was run on a malicious participant, then $\mathsf{E}[Y_{2,1}]$ can be computed as:

$$\mathsf{E}[Y_{2,1}] = 1 + \frac{P_{X_{2,1}}}{1+1} \sum_{k=0}^{1} \mathsf{E}[Y_{3,k}] + (1 - P_{X_{2,1}})\mathsf{E}[Y_{3,3}].$$

Generalizing this result to an arbitrary state $s_{i,j}$ yields:

$$\mathsf{E}[Y_{i,j}] = \begin{cases} 0 \text{ , if } j = M, \\ M - j \text{ , if } i = i^*, \\ 1 + \frac{P_{X_i}}{j+1} \sum_{k=0}^{j} \mathsf{E}[Y_{i+1,k}] + (1 - P_{X_i})\mathsf{E}[Y_{i+1,j+1}], \text{otherwise.} \end{cases}$$

The first case trivially follows from the fact that a task terminates once all M subtasks have been executed and their results been accepted. The second case uses the observation that after i^* rounds all malicious nodes have been eliminated and that the remaining $M - j$ subtask therefore can be computed in $M - j$ rounds. The third case follows from the above discussion on the example of state $s_{2,1}$. Finally, the expected number of rounds to complete an entire task is given by $\mathsf{E}[Y_{0,0}]$ and can be computed by means of recursive insertion.
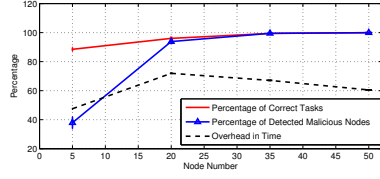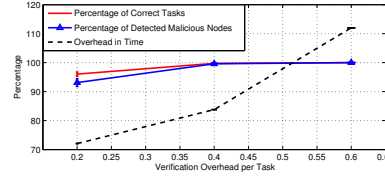
(a) Effect of the Number of Participants.
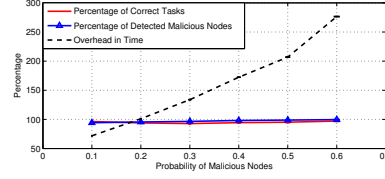
(b) Effect of the Probability of Malicious Participants.
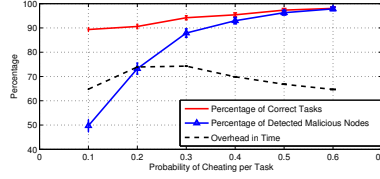
(c) Effect of the Probability of Cheating per Task.

(d) Effect of the Probability of Ringers/Redundancy per Task.

Fig. 9. Performance of our scheme with respect to various parameters. Each data point in our plots is averaged over 1000 measurements. We show the corresponding 95% confidence intervals. Our results suggest that our scheme can ensure that a substantial percentage of the tasks had been correctly executed while incurring an acceptable overhead in time (typically less than twice as much as the time needed to re-run all the N tasks on trusted nodes).

The expected number of rounds to complete a task as a function of the initial fraction of malicious nodes and the verification overhead per task (P_R and P_P) is shown in Figure 8. In this example, the number of tasks and subtasks is 20, the number of participants 50, and initially 10 % of the nodes are malicious and cheat with a probability of 40 %. We observe that even for a moderate initial fraction of malicious nodes, the actual expected value is significantly lower than the expected upper bound (depicted in Figure 7). *That is, to perform the task executions in parallel to the node set purification takes less time than it takes to first identify and remove the malicious nodes and subsequently execute the tasks on a set of honest nodes.* We therefore conclude that the naive approach in which one tries to "clear" the node set prior to the actual task executions (by running preliminary extra-computations) is *only* reasonable if the number of subtasks M is comparatively high. Note that the number of subtasks in which a task can be split depends on the structure of the task and on the induced overhead for communication and synchronization per subtask. In order to account for the impact of a non-equal number of subtasks per task as well as for other realistic conditions such as a (possibly) imperfect (pseudo-random) permutation of task assignments, we further evaluate the performance of our scheme by means of extensive simulations.

4.1   Evaluation Results

We implemented a C-based simulator to evaluate the performance of our proposal in realistic settings and with respect to various parameters. Our simulator is sequential

and round-based. It takes as input the number of tasks and subtasks (default value is 20, respectively), the number of participants (22 by default), the number of malicious nodes in the system (10% of the nodes are malicious by default), the probability of cheating per malicious node (default value = 40%) and the probability of inserting security checks within tasks (default value for the fraction of ringers = 10% and selective redundancy fraction = 10%). We chose a modest number of participants and tasks in our simulations (< 50) to better emulate realistic settings. Note, however, that the performance of our scheme considerably improves as the numbers of participants and tasks increase (Equation 2).

Throughout our simulations, we consider hybrid schemes where the supervisor equally relies on the use of embedded ringers and selective subtask redundancy. As discussed previously, such schemes are likely to be less resilient to malicious behavior when compared to the solutions where the supervisor solely makes use of ringers. We argue, therefore, that our findings presented thereafter correspond to a worst-case scenario, where the supervisor has only a limited number of ringer candidates. Conforming with our analysis in Section 3, we assume perfect collusion between malicious participants; whenever two or more malicious participants run the same subtask, they will all report the same incorrect result to the supervisor in an attempt to decrease the probability that they get caught. In our simulations, the various tasks are broken into smaller subroutines according to the control flow structure of their function. To better analyze synchronization costs in practical settings, we randomely vary the length of each subtask. The supervisor then permutes and assigns these subtasks among participants as shown in Figure 1. To ensure the correctness of the executing tasks, we adopt the recovery mechanism described in Section 4.

Our results[2] (Figure 9) confirm the analysis that we conducted in the previous sections; our proposed scheme provides a practical and robust tradeoff between the detection rate and the overhead in time with respect to the assurance level in the correctness of the tasks. In fact, even in scenarios featuring 20% colluding malicious nodes, our scheme can achieve a satisfactorily large detection rate and ensures that a substantial percentage of the tasks had been correctly executed ($> 90\%$) while incurring an overhead in time[3] that is typically less than twice as much as the time needed to execute all the N tasks on trusted nodes. Needless to mention, as the number of malicious nodes in the network increases, a larger fraction of ringers and/or redundancy is needed to prevent possible misbehavior in our scheme. In turn, this increases the number of required re-runs per task and subsequently the time required to complete the tasks' execution to ensure a satisfying level of confidence in the results (Figure 9(b)).

5   Discussion

Efficient recovery mechanisms from possible misbehavior within the remote execution constitute an important and orthogonal problem to securing distributed computations.

---

[2] In our plots, "Overhead in Time" refers to the *additional* overhead incurred by re-running a subset of the subtasks (e.g., an overhead of 50% means that our scheme results in 50% increase in execution time when compared to the time required to run a task in trusted settings).

[3] This equally includes the synchronization overhead among nodes in each round.

This problem is further aggravated when dealing with sequential computations; a single erroneous undetected subtask affects the correctness of the entire task. In the previous section, we attended to this "hitch" and we presented a framework that facilitates efficient recovery from erroneous computations.

Non-Uniform Sampling Distributions: Throughout our analysis, we considered the case where the supervisor checks a sample of the computations following a uniform distribution. Uniform distributions proved to provide a strong form of probabilistic security because they limit the number of viable countermeasures that untrusted hosts can adopt (these hosts equally have to adopt a uniform cheating strategy to increase their advantage in the network). However, uniform sampling distributions might be less optimal with respect to the efficiency of recovering from erroneous computations in sequential tasks; indeed, when relying on such distributions, the probability to catch an erroneous mistake in subtask # 1 is equal to the probability of catching an error in subtask # M. Although this comes at the benefit of a superior overall detection rate, this seems to be suboptimal in practice. In typical cases, the supervisor is more interested in catching misbehavior *earlier* during the execution since this implies less recovery overhead (an undetected error in subtask # 1 renders all subsequent computations erroneous). This suggests that biased sampling distributions – in which the supervisor checks more often the preliminary subtasks – are likely to require less re-runs to ensure the correctness of the tasks, thus boosting the recovery performance. Biased distributions come, however, at the cost of reduced detection rate; malicious participants can cheat more often in the final computations without being detected.

The optimal tradeoff between the performance in detecting malicious participants and the efficiency in recovering from malicious behavior emerges as an interesting research problem. For instance, one alternative would be to combine the use of both biased and uniform sampling distributions; this can be achieved by relying on biased checks within each task and permuting the tasks among participants such that these checks are almost uniform amongst the subtasks that each participant executes. Given this scheme, a participant is likely to expect an (almost) uniform sample-checking whereas, from the supervisor's point of view, early subtasks are checked more often than later ones. This solution requires, however, that the execution time of the tasks in the system equally follows a biased distribution in time.

Impact of Imperfect Node Isolation: So far, our analysis was based on the assumption that, once identified, malicious nodes can be completely excluded from the subsequent computation rounds. This corresponds to a closed network system where each entity can be uniquely identifiable. However, participants might be able to operate under several identities (Sybil attack [21]) or to re-enter the system with a new identity (i.e., whitewashing), which renders this assumption rather unrealistic in typical settings. In such "open" systems, the supervisor might not be able to fully isolate malicious participants; the overall fraction of malicious nodes tends to remain, to a large extent, constant in this case. Here, as opposed to a (partially) closed system, sample checking solely protects past computations performed by the participants and does not improve the conditions for the subsequently executed subtasks. Nevertheless, even in this case, our results show (refer to Figure 10) that our scheme achieves a high level of confidence in the computed
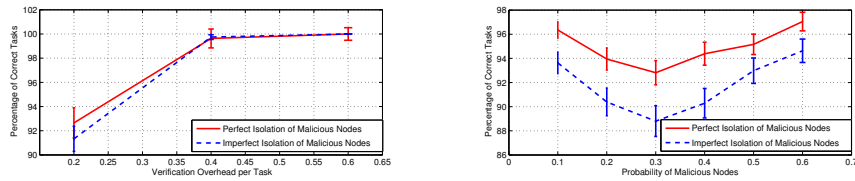
(a) Effect of the Probability of Ringers/Redundancy per Task.

(b) Effect of the Probability of Malicious Participants.

Fig. 10. Impact of Imperfect Isolation of Malicious Participants on the performance of our scheme. Our findings are derived from the simulation setup described in Section 4.1. Our results suggest that our proposed scheme achieves high confidence in the correctness of the tasks even in scenarios where perfect isolation of malicious participants might not be possible.

results. This indicates that although isolating malicious nodes is clearly beneficial, it is not a requirement to achieve reliable results in our scheme; due to the high performance of our scheme in detecting malicious behavior, the supervisor can ensure the correctness of its executing tasks by re-running a modest subset of the tasks, as described in Section 4. Note that the performance of the entire system can be further ameliorated through the use of reputation-based approaches (e.g., [15], [16], [17], [18]); in this case, each participant can be associated with a reputation value that indicates how trustworthy it is. Such an approach enables the supervisor to start with better knowledge of the participants' credibility (i.e., the initial probability of malicious participants is slightly lower when compared to open systems) and therefore bridges the gap between the aforementioned extremes: closed systems in which malicious participants can be fully isolated and open systems, where malicious nodes cannot be isolated from the system.

6   Conclusion

While there are several proposals that address the security of distributed non-sequential functions, the literature includes very few proposals for securing remote sequential computations. In this paper, we address this problem and we show that by permuting sequential tasks among several participants, efficient probabilistic measures can be used to secure the remote execution of tasks. More specifically, we demonstrate that our proposal enables a remote supervisor to selectively embed indistinguishable security checks within the sequential computations and we show that the resulting scheme facilitates the detection of individual and colluding malicious participants that cheat in a subset of the computations. We further discussed mechanisms that facilitate recovery from possible chaining of errors within the ongoing remote computations. Our findings indicate that by capitalizing on the high detection rate of our scheme to identify malicious participants, a satisfactorily modest number of re-runs per task can ensure high confidence levels in the correctness of all the tasks in the system, thus bounding the impact of malicious behavior.

References

1. SETI@home, *http://setiathome.ssl.berkeley.edu/*.
2. Distributed.Net, *http://distributed.net/*.
3. The Great Internet Mersenne Prime Search, Available from *http://www.mersenne.org/prime.htm*.
4. P. Golle and I. Mironov, "Uncheatable Distributed Computations", In *Proceedings of the RSA Conference*, 2001.
5. D. Szajda, B. Lawson and J. Owen, "Hardening Functions for Large Scale Distributed Computations", In *Proceedings of the IEEE Symposium on Security and Privacy*, 2003.
6. H. Wasserman and M. Blum, "Software Reliability via Runtime Result-Checking", In *Journal of the ACM*, 1997.
7. P. Golle and S. Stubblebine, "Secure Distributed Computing in a Commercial Environment", In *Proceedings of the International Conference on Financial Cryptography*, 2001.
8. T. Sander and C. F. Tschudin, "Protecting Mobile Agents Against Malicious Hosts", In *Mobile Agent Security*, 1998.
9. G. Vigna, "Protecting Mobile Agents Through Tracing", In *Proceedings of the ECOOP Workshop on Mobile Object Systems*, 1997.
10. W. Du, J. Jia, M. Mangal and M. Murugesan, "Uncheatable Grid Computing", In *Proceedings of ICDCS*, 2004.
11. M. T. Goodrich, "Pipelined Algorithms to Detect Cheating in Long-Term Grid Computations", In *Theoretical Computer Science*, 2008.
12. S. Yang, A.R. Butt, Y. Hu, S.P. Midkiff, "Lightweight Monitoring of the Progress of Remotely Executing Computations", In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, 2007.
13. H. Jin and J. Lotspiech, "Forensic Analysis for Tamper Resistant Software", In *Proceedings of ISSRE*, 2003.
14. C. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly", In *Proceedings of CCS*, 2003.
15. S. Kamvar, M. Schlosser, H. Garcia-Molina, "The EigenTrust Algorithm for Reputation Management in P2P Networks", In *WWW* 2003.
16. E. Damiani, S. Paraboschi, P. Samarati and F. Violante, "A Reputation-based Approach for Choosing Reliable Resources in Peer-to-Peer Networks" In *Proceedings of the ACM Conference on Computer and Communications Security*, 2005.
17. T. Dimitriou, G. Karame and I. Christou, "SuperTrust: A Secure and Efficient Framework for Handling Trust in Super Peer Networks", In *Proceedings of ACM PODC*, 2007.
18. G. Karame, I. Christou and T. Dimitriou, "A Secure Hybrid Reputation Management System for Super-Peer Networks", In *Proceedings of IEEE CCNC*, 2008.
19. M. Baron, "Probability and Statistics for Computer Science", Chapman & Hall/CRC, 2007.
20. A. Haeberlen, P. Kuznetsov, and P. Druschel, "PeerReview: Practical Accountability for Distributed System", In *Proceedings of ACM SOSP*, 2007.
21. J. Douceur, "The Sybil Attack". In *Proceedings of the IPTPS Workshop*, Cambridge, MA (USA), 2002.