Secure Coded Cooperative Computation at the Heterogeneous Edge against Byzantine Attacks

Yasaman Keshtkarjahromi*, Rawad Bitar†, Venkat Dasari‡, Salim El Rouayheb† and Hulya Seferoglu§
yasaman.keshtkarjahromi@seagate.com, rawad.bitar@rutgers.edu, salim.elrouayheb@rutgers.edu, hulya@uic.edu
*Seagate Technology, Storage Research Group, †Rutgers University, New Jersey,
‡US Army Research Lab, §University of Illinois at Chicago

***Abstract*—Edge computing is emerging as a new paradigm to allow processing data at the edge of the network, where data is typically generated and collected, by exploiting multiple devices at the edge collectively. However, offloading tasks to other devices leaves the edge computing applications at the complete mercy of an attacker. One of the attacks, which is also the focus of this work, is Byzantine attacks, where one or more devices can corrupt the offloaded tasks. Furthermore, exploiting the potential of edge computing is challenging mainly due to the heterogeneous and time-varying nature of the devices at the edge. In this paper, we develop a secure coded cooperative computation mechanism (SC^3) that provides both security and computation efficiency guarantees by gracefully combining homomorphic hash functions and coded cooperative computation. Homomorphic hash functions are used against Byzantine attacks and coded cooperative computation is used to improve computation efficiency when edge resources are heterogeneous and time-varying. Simulations results show that SC^3 improves task completion delay significantly.**

I. Introduction

Edge computing is emerging as a new paradigm to allow processing data at the edge of the network, where data is typically generated and collected. This paradigm advocates offloading tasks from an edge device to other edge/end devices including mobile devices, and/or servers in close proximity. Edge computing can be used in Internet of Things (IoT) applications which connects an exponentially increasing number of devices, including smartphones, wireless sensors, and health monitoring devices at the edge. Many IoT applications require processing the data collected by these devices through computationally intensive algorithms with stringent reliability, security and latency constraints. In many scenarios, these algorithms cannot be run locally on computationally-limited IoT-devices, and are rather outsourced to other devices.

One of the promising solutions to handle computationally-intensive tasks is computation offloading, which advocates offloading tasks to remote servers or to cloud computing platforms. Yet, offloading tasks to remote servers or to the cloud could be a luxury that cannot be afforded by most edge applications, where connectivity to remote servers can be expensive, energy consuming, lost or compromised. In addition, offloading tasks to remote servers may not be efficient in terms of delay, especially when data is generated and collected at the edge. This makes edge computing a promising solution to handle computationally-intensive tasks.

However, offloading tasks to other devices leaves the edge computing applications at the complete mercy of an attacker. One of the attacks, which is the focus of this work, is *Byzantine attacks*, where one or more devices (workers) can corrupt the offloaded tasks. Furthermore, exploiting the potential of edge computing is challenging mainly due to the heterogeneous and time-varying nature of the devices at the edge. Thus, our goal is to develop a secure, dynamic, and heterogeneity-aware edge computing mechanism that provides both security and computation efficiency guarantees.

Our key tool is the graceful use of coded cooperative computation and homomorphic hash functions. Coded computation advocates mixing data in computationally-intensive tasks by employing erasure codes and offloading these tasks to other devices for computation [1]–[13]. The following canonical example demonstrates the effectiveness of coded computation.

Example 1: Consider the setup where a master device wishes to offload a task to 3 workers. The master has a large data matrix A and wants to compute matrix vector product $A\mathbf{x}$. The master device divides the matrix A row-wise equally into two smaller matrices A_1 and A_2, which are then encoded using a $(3, 2)$ Maximum Distance Separable (MDS) code[1] to give $B_1 = A_1$, $B_2 = A_2$ and $B_3 = A_1+A_2$, and sends each to a different worker. Also, the master device sends $\mathbf{x}$ to workers and ask them to compute $B_i\mathbf{x}$, $i \in \{1, 2, 3\}$. When the master receives the computed values (*i.e.,* $B_i\mathbf{x}$) from at least two out of three workers, it can decode its desired task, which is the computation of $A\mathbf{x}$. The power of coded computations is that it makes $B_3 = A_1 + A_2$ act as a "joker" redundant task that can replace any one of the other two tasks if a worker ends up straggling, *i.e.,* being slow or unresponsive. □

This example demonstrates the benefit of coding for edge computing. However, the very nature of task offloading to workers makes the computation framework vulnerable to attacks. We focus on Byzantine attacks in this work. For example, if workers 1 and 3 in Example 1 corrupt $B_1\mathbf{x}$ and $B_3\mathbf{x}$, the master can only obtain a wrong value of $A\mathbf{x}$. Thus, it is crucial to develop a secure coded computation mechanism for edge devices against this type of attacks.

This work was supported in parts by the Army Research Lab (ARL) under Grant W911NF-1820181, National Science Foundation (NSF) under Grants CNS-1801708 and CNS-1801630, and the National Institute of Standards and Technology (NIST) under Grant 70NANB17H188.

[1]An (n, k) MDS code divides the master's data into k chunks and encodes it into n chunks ($n > k$) such that any k chunks out of n are sufficient to recover the original data.

In this paper, we develop a secure coded cooperative computation (SC^3) mechanism which uses homomorphic hash functions. Example 2 illustrates the main idea of homomorphic hash functions in coded computation.

Example 2: Consider the same setup in Example 1, and assume that worker i returns the computed value $\tilde{\mathbf{y}}_i$ to the master device. If worker i is an honest worker, $\tilde{\mathbf{y}}_i = B_i\mathbf{x}$ holds. The master device checks the integrity of $\tilde{\mathbf{y}}_i$ by calculating its hash function $h(\tilde{\mathbf{y}}_i)$, where h is a homomorphic hash function. (The details of the homomorphic hash function which we use will be provided in Section II.) The master also calculates $h(B_i\mathbf{x})$ using its local information , *i.e.,* using $h(\mathbf{x})$. If the master finds that $h(\tilde{\mathbf{y}}_i) \neq h(B_i\mathbf{x})$, it concludes that the computed value is corrupted. Otherwise, it is declared as safe. $\square$

The above example shows how homomorphic hash functions can be used for coded computation. However, existing hash-based solutions [14], [15] introduce high computational overhead, which is not suitable for edge applications, where computation power and energy are typically limited. In this paper, we use homomorphic hash functions and coded computation gracefully and efficiently. In particular, we develop and analyze light-weight and heavy-weight integrity check tools for coded computation using homomorphic hash functions. We design SC^3 by exploiting both light- and heavy-weight tools. The following are the key contributions of this work:

- We use a homomorphic hash function in [15] and show that the hash of a linear combination of computed values can be constructed by the hashes of the original tasks.
- We develop light- and heavy-weight integrity check tools for coded computation, and analyze these tools in terms of computation complexity and attack detection probability. We also analyze the trade-off between using light- and heavy-weight tools for different number of tasks.
- We design SC^3 by exploiting light- and heavy-weight tools. If an attack is detected, SC^3 can pinpoint which tasks are corrupted.
- We evaluate SC^3 for different number and strength of malicious (Byzantine) workers. The simulation results show that our algorithm significantly improves task completion delay as compared to the baseline.

The structure of the rest of this paper is as follows. Section II presents our system model. Section III presents light- and heavy-weight integrity check tools. Section IV presents our secure coded cooperative computation (SC^3) mechanism. Section V provides simulation results of SC^3. Section VI presents related work. Section VII concludes the paper.

II. System Model

Setup. We consider a master/worker setup at the edge of the network, where the master device offloads its computationally intensive tasks to workers w_n, $n \in \mathcal{N}$ (where $|\mathcal{N}| = N$) via device-to-device (D2D) links such as Wi-Fi Direct and/or Bluetooth. The master device divides a task into smaller sub-tasks, and offloads them to parallel processing workers.

Task Model. Our focus is on computation of linear functions; *i.e.,* the master device would like to compute the multiplication of matrix A with vector $\mathbf{x}$; $y = A\mathbf{x}$, where $A = (a_{i,j}) \in \mathbb{F}_{\psi}^{R\times C}$, $\mathbf{x} = (x_i) \in \mathbb{F}_{\psi}^{C\times 1}$, and $\mathbb{F}_{\psi}$ is a finite field. The motivation of focusing on linear functions stems from matrix multiplication applications where computing linear functions is a building block of several iterative algorithms such as gradient descent.

Coding. We divide matrix A into R rows denoted by A_i, $i = 1, \dots, R$. The master device applies Fountain coding [16]–[18] across rows to create coded information packets $\mathbf{q}_j \triangleq \sum_{i=1}^{R} \gamma_{i,j} A_i$, $j = 1, 2, \dots, R + \epsilon$, where ϵ is the overhead required by Fountain coding[2], and $\gamma_{i,j} \in \{0, 1\}$ are coding coefficients of Fountain coding and the information packet $\mathbf{q}_j$ is a row vector with size C. Rateless coding enabled by Fountain codes is compatible with our goal to deal with heterogeneity and time-varying nature of resources. In other words, coded packets are transmitted to workers depending on the amount of their resources (as described in Section IV-A) and Fountain codes are flexible to achieve this goal.

Worker & Attack Model. The workers incur random delays while executing the task assigned to them by the master device. The workers have different computation and communication specifications resulting in a heterogeneous environment which includes workers that are significantly slower than others, known as stragglers. Moreover, the workers cannot be trusted by the master. In particular, we consider Byzantine attacks, where one or more workers can corrupt the tasks that are assigned to them.

Homomorphic Hash Function. We consider the following hash function that maps a large number a to an output with much smaller size

$$h(a) \triangleq \mod (g^{\mod (a,q)}, r), \tag{1}$$

where q is a prime number selected randomly from the field $\mathbb{F}_{\phi}$, r is a prime number that satisfies $q|(r-1)$ (*i.e.,* $r-1$ is divisible by q) and g is a number in $\mathbb{F}_r$ which is calculated as $g = b^{(r-1)/q}$ for a random selection of $b \in \mathbb{F}_r, b \neq 1$ [14], [15]. The defined hash function is a collision-resistant hash function with the property that when ϕ increases, a is compressed less; *i.e.,* $h(a)$ becomes a better approximation of a for larger ϕ. However, the computational cost of calculating $h(a)$ increases for larger ϕ. Thus, there is a trade-off between computational complexity and better approximation of a in calculating $h(a)$. Our goal is to exploit this trade-off in the context of coded computation as described in the next sections. Another property of the defined hash function is homomorphism, *i.e.,* $h(\sum_i c_i a_i) = \prod_i h(a_i)^{c_i}$, which we will exploit in matrix-vector multiplication (in Section III).

Delay Model. Each packet transmitted from the master to a worker w_n, $n = 1, 2, ..., N$ experiences the following delays: (i) transmission delay for sending the packet from the master to the worker, (ii) computation delay for computing the multiplication of the packet by the vector $\mathbf{x}$, and (iii) transmission delay for sending the computed packet from

[2]The overhead required by Fountain coding is typically as low as %5 [18].

the worker w_n back to the master. We denote by $\beta_{n,i}$ the computation time of the i^{th} packet at worker n.

III. Light- and Heavy-Weight Integrity Check Tools for Coded Computation

In this section we present how homomorphic hash functions considered in [14], [15] and defined in (1) are used gracefully with coded computation. We first show that (1) can be applied to coded computation. Then, we develop light- and heavy-weight integrity check tools. The tools we develop in this section will be building blocks of our secure coded cooperative computation mechanism (SC^3).

A. Homomorphic Hash Function for Coded Computation

Let us consider that Z_n coded information packets are offloaded to worker w_n. The i^{th} packet offloaded to w_n is $\mathbf{p}_{n,i} \in \{\mathbf{q}_1, \dots, \mathbf{q}_{R+\epsilon}\}$, which can be represented as $\mathbf{p}_{n,i} = (p_{n,i,1}, \dots, p_{n,i,C})$, where $p_{n,i,j}$ is the j^{th} element of vector $\mathbf{p}_{n,i}$. Worker w_n calculates $y_{n,i} = \mathbf{p}_{n,i}\mathbf{x}$ and sends it back to the master device.

Assume that the master receives $\tilde{y}_{n,i}$ from w_n, where $\tilde{y}_{n,i} = y_{n,i}$ if packet is not corrupted. The master device checks the integrity of packets calculated at w_n according to the following rule. First, it calculates

$$\alpha_n = h(\sum_{i=1}^{Z_n} c_i \tilde{y}_{n,i}), \tag{2}$$

using the hash function defined in (1), where c_i's are coefficients (We will discuss how c_i is selected later in this section.). Next, it calculates

$$\beta_n = \mod \Big(\prod_{j=1}^{C} h(x_j)^{\mod \left((\sum_{i=1}^{Z_n} c_i p_{n,i,j}), q \right)}, r \Big), \tag{3}$$

where x_j is the j^{th} element of vector $\mathbf{x}$, and q and r are the parameters of the hash function defined in (1). β_n in (3) is calculated by the master device using its local data $\mathbf{p}_{n,i}$ and $\mathbf{x}$. β_n is used to check α_n as described in the next theorem.

Theorem 1: If w_n does not corrupt packets, *i.e.,* $\tilde{y}_{n,i} = y_{n,i}$, $\forall i$, and c_i is a nonzero integer, then $\alpha_n = \beta_n$ holds.

Proof: The proof is provided in our technical report [19]. ■

We note that Theorem 1 is necessary, but not sufficient condition to determine if w_n is malicious or not. The sufficiency condition depends on how c_i is selected as explained next.

B. Light-Weight Integrity Check (LW Function)

The light-weight integrity check (LW function) uses Theorem 1 to determine if workers corrupt packets or not. In particular, LW function calculates α_n in (2) and β_n in (3) by selecting c_i randomly and uniformly from $\{-1, 1\}$. LW function concludes that packets processed by w_n are not corrupted if $\alpha_n = \beta_n$. However, as we discussed earlier, this condition is not always a sufficient condition, so LW function detects attacks with some probability, which is provided next.

1) Probability of Attack Detection: We first consider a pair-wise Byzantine attack, where malicious worker w_n corrupts two packets out of Z_n packets by adding and subtracting terms. For example, $\tilde{y}_{n,i} = y_{n,i} + \delta_i$ and $\tilde{y}_{n,j} = y_{n,j} - \delta_j$, for any arbitrary i,$j \leq Z_n$ satisfying $i \neq j$. In this attack pattern, if $|\delta_i| \neq |\delta_j|$, and considering that the coefficients are selected from $\{-1, 1\}$ in LW function, the attack is detected with 100% probability. On the other hand, if attack is symmetric, *i.e.,* $|\delta_i| = |\delta_j|$, the probability of detecting the attack is 50%. As symmetrical attacks are the most difficult ones to detect, we focus on this scenario in the next lemma.

Lemma 2: Consider an attack where the malicious worker w_n selects an even number $\tilde{Z}_n$ randomly out of Z_n packets and corrupt them by adding δ to half of them, and subtracting δ from the other half. The probability of attack detection by LW function is

$$P_{\text{detect}}^{\text{LW}} = 1 - \Big(\frac{\tilde{Z}_n!}{2^{\tilde{Z}_n} \big((\tilde{Z}_n/2)! \big)^2} \Big). \tag{4}$$

Proof: The proof is provided in our technical report [19]. ■

As seen from Lemma 2, the probability of attack detection increases with increasing number of corrupted packets. This result intuitively holds for any attack pattern as the coefficients (c_i) are selected randomly for each packet and estimating these values by an attacker becomes difficult for larger set of corrupted packets. The other attack patterns and detection probabilities are provided in our technical paper [19].

Lemma 3: The probability of attack detection when LW function is used and for any attack pattern is lower bounded by $P_{\text{detect}}^{\text{LW}} \geq 0.5$.

Proof: The proof is provided in our technical report [19]. ■

2) Computational Complexity:

Theorem 4: The computational complexity of LW function for checking Z_n packets calculated by w_n is $O(CM(r)\log_2 q)$, where C is the size of each information packet, $M(r)$ is the complexity of multiplication in $\mathbb{F}_r$, and r and q are the parameters of the hash function defined in (1).

Proof: The complexity of LW function consists of two parts; calculation of α_n in (2) and β_n in (3). We first analyze computational complexity of calculating α_n. The sum $\sum_{i=1}^{Z_n} c_i \tilde{y}_{n,i}$ only has addition and subtraction as $c_i \in \{-1, 1\}$, and can be ignored. The complexity of the modular exponentiation while calculating the hash function is $O(M(r)\log_2 q)$ by using the method of exponentiation by squaring.

Similarly, we can calculate the computational complexity of calculating β_n. The complexity for computing $\sum_{i=1}^{Z_n} c_i p_{n,i,j}$ corresponds to the complexity of addition and subtraction, which is negligible. The complexity of computing $\mod (\prod_{j=1}^{C} h(x_j)^{\mod (\sum_{i=1}^{Z_n} c_i p_{n,i,j}, q)}, r)$ has two components: (i) Calculating the modular exponentiations $h(x_j)^{\mod (\sum_{i=1}^{Z_n} c_i p_{n,i,j}, q)}, \forall j = 1, 2, ..., C$: The complexity for this calculation is $O(M(r)\log_2 q)$ for one modular exponentiation and $O(CM(r)\log_2 q)$ for all C modular exponentiations. (ii) Multiplying all the calculated modular exponentiations, *i.e.,* $\prod_{j=1}^{C} h(x_j)^{\mod (\sum_{i=1}^{Z_n} c_i p_{n,i,j}, q)}$ in $\mathbb{F}_r$: The complexity for this calculation is $O((C-1)M(r))$. Thus, the total complexity of LW function becomes $O(CM(r)\log_2 q)$. This concludes the proof. ■

Noting that the computational complexity of calculating the original matrix multiplication is $O(RCM(\psi))$, where $M(\psi)$ is the complexity of multiplication in $\mathbb{F}_\psi$. As seen, the complexity of the LW function is significantly low, compared to the original task. This means LW function provides security check with low complexity. However, the probability of attack detection using LW function could be as low as 50%, which may not be acceptable in some applications. Thus, we provide a heavy-weight integrity check tool (HW function) in the next section. Our ultimate goal is to use LW and HW functions together for higher attack detection probability while still having low computational complexity.

C. Heavy-Weight Integrity Check (HW Function)

The heavy weight integrity check (HW function) uses Theorem 1 similar to the LW function, but chooses the coefficients c_i from a larger field $\mathbb{F}_\phi$ rather than $\{-1, 1\}$. This selection, *i.e.,* choosing coefficients from a larger field, comes with larger attack detection probability and computational complexity as described next.

1) Probability of Attack Detection:

Lemma 5: The probability that HW function detects a Byzantine attack with any attack pattern is expressed as

$$P_{\text{detect}}^{\text{HW}} = 1 - \frac{1}{q} \tag{5}$$

where q is the parameter of the hash function in (1).

Proof: The proof is provided in our technical report [19]. ■

As seen from Lemma 5, the attack detection probability increases with increasing q. Next, we present the computational complexity of HW function.

2) Computational Complexity:

Theorem 6: The computational complexity of HW function for checking Z_n packets calculated by w_n is $O(CZ_nM(\phi))$.

Proof: The proof follows the same logic of the proof of Theorem 4, *i.e.,* the complexity of HW function depends on calculating α_n in (2) and β_n in (3). The difference as compared to the proof of Theorem 4 is that c_i's are selected from a larger field, so reducing multiplication to addition in $\sum_{i=1}^{Z_n} c_i\tilde{y}_{n,i}$ of (2) and $\sum_{i=1}^{Z_n} c_i p_{n,i,j}$ of (3) cannot be made. In particular, the complexity of calculating these terms is $O(Z_nM(\phi))$. Following the similar steps in the proof of Theorem 4, we can conclude that the computational complexity of HW function becomes $O(CM(r)\log_2 q)+O(CZ_nM(\phi))$. Since the second term dominates the computational complexity for large R (hence Z_n), we calculate the computational complexity as $O(CZ_nM(\phi))$. This concludes the proof. ■

We can approximate Z_n to $(R+\epsilon)/N$ on average assuming that coded information packets are distributed homogeneously across workers, where R is the number of information packets, ϵ is the Fountain coding overhead, and N is the number of workers. Thus, the computational complexity of HW function across all workers becomes $NO(\frac{C(R+\epsilon))}{N}M(\phi))$. As we discussed earlier, the computational complexity of the original matrix multiplication is $O(RCM(\psi))$. We also note that $M(\psi) >> M(\phi)$. This means that even though HW function is computationally complex as compared to LW function, it is still computationally-efficient with respect to the original matrix multiplication (considering that ϵ is small and approaches to 0 with increasing number of packets).

D. Light- versus Heavy-Weight Integrity Check

In this section, we investigate employing LW function multiple rounds/times to achieve higher attack detection probability with low computational complexity. LW function is used to check Z_n packets computed by w_n by selecting c_i uniformly randomly from $\{-1, 1\}$. Let us call this the first round. In the second round, we can use LW function again, but selected values of c_i will be different from the first round. Thus, if an attack is not detected in the first round, it may still be detected in the next round. Thus, using LW function over multiple rounds will increase the attack detection probability. The next theorem characterizes the performance of LW function when used in multiple rounds as compared to HW function.

Theorem 7: The attack detection probability of multiple-round LW function is equal to the attack detection probability of HW function in (5) when LW function is used for $\log_2(q)$ rounds. Furthermore, the computational complexity of $\log_2(q)$-round LW function is lower than HW function if the following condition is satisfied.

$$Z_n \geq \frac{M(r)}{M(\psi)} \log_2 q(1 + \log_2 q), \tag{6}$$

Proof: The proof is provided in our technical report [19]. ■

IV. SC^3: Secure Coded Cooperative Computation

In this section, we present our secure coded cooperative computation (SC^3) mechanism. SC^3 consists of packet offloading, attack detection, and attack recovery modules.

A. Dynamic Packet Offloading

The dynamic packet offloading module of SC^3 is based on [1]. In particular, the master offloads coded packets gradually to workers and receives two ACKs for each transmitted packet; one confirming the receipt of the packet by the worker, and the second one (piggybacked to the computed packet) showing that the packet is computed by the worker. Then, based on the frequency of the received ACKs, the master decides to transmit more/less coded packets to that worker. In particular, each packet $\mathbf{p}_{n,i}$ is transmitted to each worker w_n before or right after the computed packet $\mathbf{p}_{n,i-1}\mathbf{x}$ is received at the master. For this purpose, the average per packet computing time $E[\beta_{n,i}]$ is calculated for each worker w_n dynamically based on the previously received ACKs. Each packet $\mathbf{p}_{n,i}$ is transmitted after waiting $E[\beta_{n,i}]$ from the time $\mathbf{p}_{n,i-1}$ is sent or right after packet $\mathbf{p}_{n,i-1}\mathbf{x}$ is received at the master, thus reducing the idle time at the workers. This policy is shown to approach the optimal task completion delay and maximizes the workers' efficiency and is shown to improve task completion delay significantly compared with the literature [1].

B. Attack Detection

Assume that the set of received packets from worker n is $\mathcal{Z}_n$ ($Z_n = |\mathcal{Z}_n|$) at the master device at time t while the dynamic packet offloading process continues. If a new calculated packet $\tilde{y}_{n,i}$ is received, the following updates are made; $Z_n = Z_n + 1$, and $\mathcal{Z}_n = \mathcal{Z}_n \bigcup \tilde{y}_{n,i}$. The attack detection module of SC^3 is applied on $\mathcal{Z}_n$ periodically and consists of two phases.

The first phase applies LW function on the packets in $\mathcal{Z}_n$ for any worker $n \in \mathcal{N}$. Let us assume that attack is detected in the packets coming from w_{n^*}. Then, all the packets in $\mathcal{Z}_{n^*}$ are discarded and the malicious worker w_{n^*} is removed from the set of workers, *i.e.,* $\mathcal{N} = \mathcal{N} - w_{n^*}$. As we discussed earlier, the attack detection probability of LW function increases with increasing corrupted packets. Thus, if an attack is detected in this phase, it can be considered that most of the packets are corrupted, so we can discard all the received packets.

The goal of the second phase is to detect any attacks, which are not detected in the first phase. Both HW and multiple-round LW functions are used in this phase. In particular, if the inequality in Theorem 7 is satisfied, LW function is used for $\log_2(q)$ times. Otherwise, HW function is used. If an attack is not detected, all the packets in Z_n are labeled as *verified* packets. Otherwise, *i.e.,* if an attack is detected, the attack recovery module, which is described later in this section, starts. The attack detection module stops when the number of verified packets from all workers reaches $R + \epsilon$.

C. Attack Recovery

If an attack is detected in the second phase of the attack detection module of SC^3, we consider that a small number of packets are corrupted. Otherwise, the first phase of the attack detection module could have detected the attack and discarded all the packets. Thus, the goal of the attack recovery module is to recover from small number of corrupted packets without discarding all the packets.

Let us assume that an attack is detected among the packets sent from w_{n^*}, *i.e.,* in $\mathcal{Z}_{n^*}$. In order to pinpoint the packets that are corrupted, we use a binary search algorithm. In particular, $\mathcal{Z}_{n^*}$ is divided into two disjoint sets; $\mathcal{Z}^1_{n^*}$ and $\mathcal{Z}^2_{n^*}$. The second phase of the attack detection module is run over these two sets. If an attack is not detected on any of these sets, all the packets in that set are verified. Otherwise, the binary search (this set splitting) continues over the sets that an attack is detected. When the size of a splitted set is one, *i.e.,* it has one packet in it, and an attack is detected, the packet in that set is declared a corrupted packet and discarded. As seen, the attack recovery module can still verify some of the packets coming from a malicious worker. This is important to efficiently utilize available resources while still providing security guarantees.

V. Performance Evaluation

In this section, we evaluate the performance of our algorithm; Secure Coded Cooperative Computation (SC^3) via simulations. We consider master/worker setup, where some of the workers are malicious. Each computed packet $y_{n,i}$ is corrupted by the malicious worker w_n with probability ρ_c. The computing resources are heterogeneous and vary across workers, where per packet computing delay $\beta_{n,i}$ is an i.i.d. random variable following a shifted exponential distribution. We compare SC^3 with the baseline HW-only that uses HW function to detect corrupted packets, while SC^3 uses both LW and HW functions gracefully. In HW-only, if a worker is detected as malicious, all the packets coming from that worker are discarded.

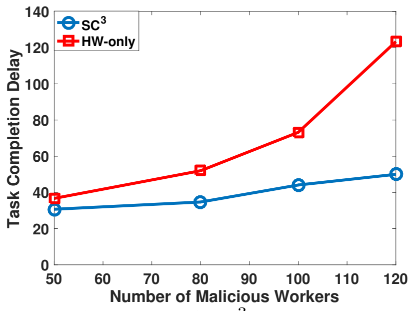

Fig. 1. Task completion delay of SC^3 as compared to HW-only with increasing number of malicious workers.

Task Completion Delay vs. Number of Malicious Workers. Fig. 1 compares the task completion delay of SC^3 with HW-only for increasing number of malicious workers. The task completion delay is the time that takes to collect $R + \epsilon$ computed and verified packets ($y_{n,i}$'s). In this setup, the total number of workers is $N = 150$, the number of rows in matrix A is $R = 1K$, the number of columns is $C = 1K$, the overhead of Fountain codes is 5%, the probability of packet corruption is $\rho_c = 0.3$, and per-packet computing delay is a shifted exponential random variable with the mean selected uniformly between 1 and 6 for each worker.

The task completion delay of SC^3 and HW-only increases with increasing number of malicious workers. When the number of malicious workers increases, there will be more corrupted packets in the system. These corrupted packets are detected and discarded by SC^3 and HW-only. As more packets are discarded when the number of malicious workers is higher, the task completion delay increases. The increase in the task completion delay of SC^3 is less than HW-only thanks to (i) using both LW and HW functions to reduce computational complexity, and (ii) attack recovery module of SC^3.

Task Completion Delay versus Packet Corruption Probability. Fig. 2 compares the task completion delay of SC^3 with HW-only for different values of ρ_c, the probability that a delivered packet by a malicious worker is corrupted. The number of workers, the number of rows in A, Fountain coding overhead, and per-packet computing delay are the same as the previous setup above. The number of malicious workers is 50.

The task completion delay of HW-only does not change with increasing packet corruption probability. The reason is that HW-only does not have attack recovery feature and discards all the packets coming from a malicious worker. On the other hand, task completion delay of SC^3 is significantly lower than HW-only especially when the packet corruption probability is low thanks to using both LW and HW functions and employing the attack recovery module.

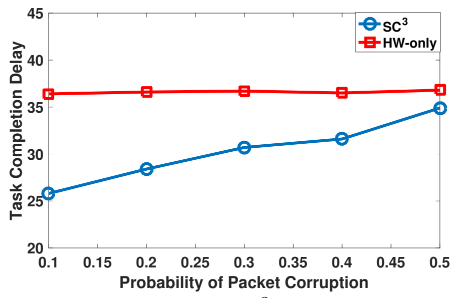

Fig. 2. Task completion delay of SC^3 as compared to HW-only with increasing probability of packet corruption.

VI. Related Work

Coded computation, advocating mixing data in computationally intensive tasks by employing erasure codes and offloading these tasks to other devices for computation, has recently received a lot of attention, [1]–[13]. For example, coded cooperative computation is shown to provide higher reliability, smaller delay, and reduced communication cost in MapReduce framework [20], where computationally intensive tasks are offloaded to distributed server clusters [21]. The effectiveness of coded computation in terms of task completion delay has been investigated in [1], [7], [11]. In [22], the same problem is considered, but with the assumption that workers are heterogeneous in terms of their resources. In [1], a dynamic and adaptive algorithm with reduced task completion time is introduced for heterogeneous workers. As compared to this line of work, we consider secure coded computation by focusing on Byzantine attacks.

There is existing work at the intersection of coded computation and security by specifically focusing on privacy [2], [23]–[25]. As compared to this line of work, we focus on Byzantine attacks and use homomorphic hash functions. Homomorphic hash functions have been widely used for transmission of network coded data. Corrupted network coded packets are detected by applying homomorphic hash functions that we consider in this work [14]. The hash function is applied to random linear combinations of network coded packets in [15]. SC^3, although similar to these work, is more efficient in terms of computational efficiency, which was not the main concern of [14], [15] as their focus was on transmitting network coded packets, not computation.

VII. Conclusion

In this paper, we focused on secure edge computing against Byzantine attacks. We considered a master/worker scenario where honest and malicious workers with heterogeneous resources are connected to a master device. We designed a secure coded cooperative computation mechanism (SC^3) that provides both security and computation efficiency guarantees by gracefully combining homomorphic hash functions, and coded cooperative computation. Homomorphic hash functions are used against Byzantine attacks and coded cooperative computation is used to improve computation efficiency when edge resources are heterogeneous and time-varying. Simulations results show that SC^3 improves task completion delay significantly.

References

[1] Y. Keshtkarjahromi, Y. Xing, and H. Seferoglu, "Dynamic heterogeneity-aware coded cooperative computation at the edge," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, Sept 2018.

[2] R. Bitar, P. Parag, and S. El Rouayheb, "Minimizing latency for secure distributed computing," in *Information Theory (ISIT), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 2900–2904.

[3] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "A unified coding framework for distributed computing with straggling servers," in *Globecom Workshops (GC Wkshps), 2016 IEEE*. IEEE, 2016, pp. 1–6.

[4] S. Dutta, V. Cadambe, and P. Grover, "Coded convolution for parallel and distributed computing within a deadline," *arXiv preprint arXiv:1705.03875*, 2017.

[5] Y. Yang, P. Grover, and S. Kar, "Computing linear transformations with unreliable components," *IEEE Trans. on Information Theory*, 2017.

[6] W. Halbawi, N. Azizan-Ruhi, F. Salehi, and B. Hassibi, "Improving distributed gradient descent using reed-solomon codes," *arXiv preprint arXiv:1706.05436*, 2017.

[7] Q. Yu, M. Maddah-Ali, and S. Avestimehr, "Polynomial codes: an optimal design for high-dimensional coded matrix multiplication," in *Advances in Neural Information Processing Systems*, 2017.

[8] S. Dutta, V. Cadambe, and P. Grover, "Short-dot: Computing large linear transforms distributedly using coded short dot products," in *NIPS*, 2016, pp. 2092–2100.

[9] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient coding: Avoiding stragglers in distributed learning," in *International Conference on Machine Learning*, 2017, pp. 3368–3376.

[10] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "Fundamental tradeoff between computation and communication in distributed computing," in *IEEE International Symposium on Information Theory (ISIT)*, 2016.

[11] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding up distributed machine learning using codes," *IEEE Transactions on Information Theory*, vol. 64, no. 3, pp. 1514–1529, 2018.

[12] C. Karakus, Y. Sun, S. Diggavi, and W. Yin, "Straggler mitigation in distributed optimization through data encoding," in *Advances in Neural Information Processing Systems*, 2017, pp. 5434–5442.

[13] M. F. Aktas, P. Peng, and E. Soljanin, "Effective straggler mitigation: Which clones should attack and when?" *ACM SIGMETRICS Performance Evaluation Review*, vol. 45, no. 2, pp. 12–14, 2017.

[14] M. N. Krohn, M. J. Freedman, and D. Mazieres, "On-the-fly verification of rateless erasure codes for efficient content distribution," in *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*. IEEE, 2004, pp. 226–240.

[15] C. Gkantsidis and P. Rodriguez, "Cooperative security for network coding file distribution." in *INFOCOM*, vol. 3, no. 2006, 2006.

[16] M. Luby, "Lt codes," in *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, Nov 2002, pp. 271–280.

[17] A. Shokrollahi, "Raptor codes," *IEEE/ACM Transactions on Networking (TON)*, vol. 14, no. SI, pp. 2551–2567, 2006.

[18] D. J. MacKay, "Fountain codes," *IEE Proceedings-Communications*, vol. 152, no. 6, pp. 1062–1068, 2005.

[19] Y. Keshtkarjahromi, R. Bitar, V. Dasari, S. E. Rouayheb, and H. Seferoglu, "Secure coded cooperative computation at the heterogeneous edge against byzantine attacks," https://www.dropbox.com/s/cnfqspd4wtzo6qn/SC3.pdf?dl=0.

[20] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[21] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "Coded mapreduce," in *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 2015, pp. 964–971.

[22] A. Reisizadeh, S. Prakash, R. Pedarsani, and A. S. Avestimehr, "Coded computation over heterogeneous clusters," *IEEE Transactions on Information Theory*, 2019.

[23] H. Yang and J. Lee, "Secure distributed computing with straggling servers using polynomial codes," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 1, pp. 141–150, Jan 2019.

[24] Q. Yu, N. Raviv, J. So, and A. S. Avestimehr, "Lagrange coded computing: Optimal design for resiliency, security and privacy," *arXiv preprint, arXiv:1806.00939*, 2018.

[25] R. Bitar, Y. Xing, Y. Keshtkarjahromi, V. Dasari, S. El Rouayheb, and H. Seferoglu, "Prac: Private and rateless adaptive coded computation at the edge," in *SPIE Defense + Commercial Sensing*, 2019.