# ReplayMPC: A Fast Failure Recovery Protocol for Secure Multiparty Computation Applications using Blockchain

Oscar G. Bautista*, Kemal Akkaya*, and Soamar Homsi†
*Department of Electrical and Computer Engineering, Florida International University, Miami, USA
† Air Force Research Laboratory, Information Warfare Division, Rome, NY, USA
Emails: {obaut004, kakkaya}@fiu.edu, soamar.homsi@us.af.mil

***Abstract*—Although recent performance improvements to Secure Multiparty Computation (SMPC) made it a practical solution for complex applications such as privacy-preserving machine learning (ML), other characteristics such as robustness are also critical for its practical viability. For instance, since ML training under SMPC may take longer times (e.g., hours or days in many cases), any interruption of the computation will require restarting the process, which results in more delays and waste of computing resources. While one can maintain exchanged SMPC messages in a separate database, their integrity and authenticity should be guaranteed to be able to re-use them later. Therefore, in this paper, we propose ReplayMPC, an efficient failure recovery mechanism for SMPC based on blockchain technology that enables resuming and re-synchronizing SMPC parties after any type of communication or system failures. Our approach allows SMPC parties to save computation state snapshots they use as restoration points during the recovery and then reproduce the last computation rounds by retrieving information from immutable messages stored on a blockchain. Our experiment results on Algorand blockchain show that recovery is much faster than starting the whole process from scratch, saving time, computation, and networking resources.**

***Index Terms*—secure multiparty computation, failure recovery, resilient MPC, blockchain broadcast channel, privacy-preserving machine learning**

## I. Introduction

Secure Multiparty Computation (SMPC) has been gaining attention as a viable solution for performing computation on sensitive data (e.g., medical records [1]), or data that may provide related entities with a competitive advantage if disclosed (e.g., financial organizations having financial scores and customer information [2], [3]). In SMPC, two or more parties jointly evaluate a function without knowing other parties' private inputs. Specifically, every SMPC party gets a share of each party's input and executes an SMPC protocol in rounds. Each SMPC round consists of a local computation performed by each party, followed by a message exchange through the network interconnecting all parties.

Although SMPC efficiency has improved during the last few years, it is still considerably slower than computing in the clear. For instance, performing Machine Learning (ML) training in SMPC can take hours or even days [4], [5], depending on the complexity and the size of the data. The increased execution time is due to the additional computation steps required to compute on secret shares and, more importantly, network communication overhead. However, this increased computation time also increases the risk of SMPC halting due to various reasons, including spontaneous node, OS, and communications failures. This poses challenges that may affect the robustness [6], which is the ability of the SMPC system to successfully compute an output in the presence of failures.

While there are already some existing SMPC approaches that can offer robustness by tolerating a certain number of node failures, this comes at the expense of increased execution time and reduced privacy. For instance, Shamir Secret-Sharing (SSS) [7] can be set up to tolerate the failure of less than 2/3 of the total number of SMPC parties in the joint computation, but this reduces privacy because, with the corruption of more than 1/3 of the SMPC parties, the attacker can reveal the secret inputs. In contrast, there are secret-sharing schemes where privacy of the output is guaranteed with a very high probability in the presence of up to $n-1$ malicious parties (e.g., SPDZ [8]). Nonetheless, such systems cannot inherently tolerate a single party failure. Therefore, there is a need to offer robustness without sacrificing privacy and performance degradation in all SMPC protocols.

In this work, we propose ReplayMPC, an efficient failure recovery protocol which allows parties to rejoin a halted computation execution, achieving re-synchronization with the current SMPC round. This prevents parties from discarding the computation progress made before identifying a failure; saving time and computation resources. We propose to leverage blockchain technology as an efficient communication network. Specifically, our recovery protocol leverages previous SMPC message exchanges from the ledger and previously saved internal computation states to reproduce the lost computation without interacting with the rest of their peers. We model different states of the SMPC parties using a finite state machine to keep track of the events and synchronize with other parties. This brings a self-recovery capability without relying on any other external entities.

We implemented and tested our proposed recovery protocol using a cloud-hosted privacy-preserving ML training use case and the SPDZ protocol. While the main goal is to confirm the correct re-synchronization of a disconnected party, we also asses the time it takes for such a party to reproduce a portion of the training computation during the recovery phase. The

Approved for Public Release on 15 Nov 2022; Distribution Unlimited; Case Number: AFRL-2022-5506.

results of our experiments show that our approach enhances the robustness of an SMPC system by reducing the recovery time more than 87% compared to the case when the computation executes normally from scratch.

This paper is organized as follows: Section II summarizes previous works. Section III overviews key aspects of SMPC and ML and describes our system model and general architecture components. Section IV describes our recovery algorithm. Section V presents security considerations for our proposed approach. We discuss our experimental evaluation and results in Section VI. Finally, we present our conclusions in Section VII.

## II. Related Work

Recent research on Robust SMPC systems mainly leverages the inherent characteristic of threshold secret-sharing (i.e., secret-sharing schemes used by $n$ parties, where $t$ shares are needed to reconstruct the secret, and $1 < t \leq n$) such as SSS [7]. For instance, HoneyBadgerMPC (HBMPC) [9] and BFR-MPC [10] guarantee an output delivery within a certain limit determined by the secret-sharing threshold. Similarly, PARMPC [11] addresses robustness by extending SPDZ-like protocols to support threshold secret-sharing. The above approaches have no specific requirement for bounding $t$. Nonetheless, the smaller the $t$ is, the more robust the approach is because the scheme would allow more parties to stop collaborating before causing a computation halt. Additionally, since PARMPC's primary goal is public accountability, the general approach is (reasonably) slower than the traditional SPDZ protocol they extended. Different from these works, our approach fits SMPC systems irrespective of the number of corrupted parties. It focuses on enabling a node to recover from a failure using any broadcast-based SMPC protocol (especially, those with a fixed number of parties). Additionally, while the threshold secret-sharing approaches reduce the security (privacy) of the data (i.e., the more parties allowed to drop out of the computation, the fewer parties are required to reconstruct the secret.), our approach maintains the same level of security of the implemented SMPC protocol. Finally, BFR-MPC uses blockchain contracts to guarantee fairness and integrity in every round, which increases the total execution time. In contrast, ReplayMPC relies on blockchain and its efficient and secure broadcast communication channel [6]. To the best of our knowledge, we are the first to utilize blockchain for SMPC recovery purposes.

A. Choudhuri et al. [12] proposes fluid MPC protocols in the honest majority setting where the set of parties does not need to be constant for the whole MPC computation. Instead, some or all parties may leave the current computation round, and new players would replace them without interrupting the computation flow. This might increase the robustness of a system against network failures or DoS attacks, provided that the party leaving the computation can send their computation state to the replacement party (i.e., coordinated replacement). In contrast, our proposed approach leverages an immutable history of message exchanges that allows the parties to recover from unexpected failures, such as network-based issues, and enables a replacement of a permanently failed party to replay the previous rounds in much less time compared to having all parties restart the computation.

## III. Preliminaries and System Model

### A. Preliminaries

*1) SMPC Overview:* An SMPC system consists of two or more parties that perform the secure computation, a network over which the parties share messages, and a protocol that parties follow to evaluate a function interactively. SMPC achieves secrecy by implementing a secret-sharing protocol. Essentially, each party holds a piece (a secret share) of private variables and does not leak any information to other parties.

The main properties of SMPC are privacy and correctness. Privacy is the guarantee that the secret share does not leak information about the data. Correctness indicates whether the result is correct or covertly manipulated. These properties depend heavily on the behavior of the parties. For instance, semi-honest parties follow the rules but try to infer more information than intended (some literature also refers to them as honest-but-curious parties). In contrast, malicious parties may deviate from the protocol to cause harm (e.g., colluding with other parties to undermine privacy or introduce errors to change the results). For instance, SSS [7] protocols are secure against semi-honest adversaries. The number of such adversarial parties is a threshold $t$ of the total number of parties $n$, where typically $t < n/2$ or $t < n/3$. Another example of SMPC protocol is SPDZ by Damgård et al. [8], which is maliciously secure with a dishonest majority (i.e., up to $n - 1$). This protocol uses additive secret sharing, in which parties get shares of the private values and authentication tags, which are used at the end to verify the correctness of the computation. Furthermore, this protocol follows the offline/online phase paradigm. First, it computes some correlated random materials, also known as *raw materials*, in a computationally-heavy and input-independent *offline phase*. This phase is followed by a much more efficient *online phase*, where the parties jointly evaluate a function with the (private) input data. The offline phase is also referred to as pre-processing phase since it is independent of the private data.

*2) Privacy-Preserving Machine Learning:* Machine Learning (ML) training, in general, is an iterative process where a mathematical model is initialized with random parameters and evaluated on the training set of inputs to produce an output. An optimization algorithm uses the average error of the model outputs with respect to the known (i.e., real) outputs to update the model parameters. This process is repeated across several iterations over the training dataset until the model's accuracy exceeds an established threshold. Once the training algorithm consumes the training dataset completely, it completes one *epoch*. Generally, an ML training algorithm takes several epochs to reach the accuracy threshold.

Several ML applications are now relying on SMPC as a suitable solution for their privacy requirements [13]. However, training an ML model in SMPC may take a lot of time and consequently increase the risk of failure at the node or communication network levels. Therefore, long SMPC computations

such as ML training are ideal use cases to employ our proposed failure recovery mechanism.

### B. System Model

We assume the availability of an SMPC setup consisting of a set $\mathcal{P} = \{P_1, P_2, ...P_n\}$ of nodes, also referred to as parties. These parties may be hosted in separate cloud locations. We also assume the existence of a broadcast channel among SMPC nodes based on a blockchain network (i.e., parties can access the same blockchain to receive messages in the form of transactions from the other parties).

In our case, we use a private Algorand blockchain-based SMPC system [6] that enables access to fast-propagated transactions. Such an approach outperforms general SMPC in geographically distributed deployments. The Algorand-based system consists of the following components.

*a) SMPC Node:* This node is capable of communicating with the blockchain to submit and receive transactions. An SMPC node runs two processes as depicted in Fig. 1. The first one is an Algorand process responsible for the blockchain-related tasks (i.e., submitting and receiving transactions). The second one is an SMPC process that executes the specific SMPC protocol, and communiates with the first one through an API. The security stance of an SMPC node depends on the instantiated SMPC protocol's threat model. As such, a node behaving semi-honestly may try to learn other parties' inputs, and a node controlled by a malicious adversary may also try to change the results. Note that an SMPC node connects to other SMPC nodes through relay nodes, which are explained next.

*b) Relay Node:* These nodes are the backbone of the Algorand network. The Algorand blockchain uses relay nodes for fast propagation of transactions and other messages over the network. The relay nodes run a gossip protocol [14] which de-duplicate and groups the messages to improve the efficiency of the transaction propagation. The Algorand relay nodes also store the whole ledger and can respond to requests to retrieve any block since the initial creation of the blockchain. The SMPC system setup (e.g., number of SMPC parties, size of the secret shares and the function to evaluate, etc.) defines the storage requirements of a relay/archival node. For reference, evaluating the ResNet-50 neural network requires over 41 GB of storage [15].

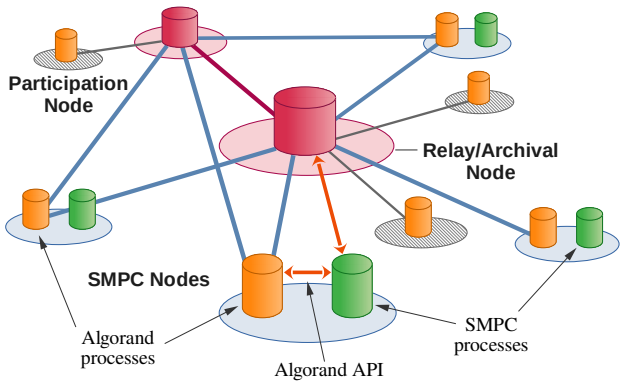



Fig. 1. MPC-ABC System Architecture

*c) Participation Node:* These nodes run the blockchain's consensus protocol, adding new blocks to the ledger. While at least two participation nodes can run the consensus protocol, Algorand requires that honest participants hold at least 2/3 of the voting stake. Therefore, a minimum of three nodes is recommended to add a margin that covers an eventual failure. Furthermore, unlike an SMPC node, a dishonest participation node may try to prevent transaction confirmation but cannot modify authentic transactions as their sender signed them. Like any other node in the Algorand network, the participation nodes connect to relay nodes.

## IV. Blockchain-based SMPC Recovery

### A. Motivation and Overview

While some SMPC protocols can tolerate a certain number of parties leaving the computation without impeding the remaining parties from computing the intended result [16], [17] (e.g., protocols based on threshold secret-sharing), other protocols based on additive secret-sharing strictly require the interaction of all parties to continue with the execution [8]. Nonetheless, even when the SMPC parties use threshold secret-sharing, there is a limit to the number of parties that the system tolerates becoming unresponsive, beyond which the execution halts. For instance, let us consider the case of a group of SMPC parties performing an ML training task using a protocol based on additive secret-sharing. In this scenario, a party may stop responding because of external or internal causes (e.g., a communication or OS related issue that temporarily disconnects the party from the network). Consequently, the SMPC execution halts since the remaining parties await the messages from the failed party. Eventually, the parties abort after a certain timeout period without returning any result. This may affect applications such as privacy-preserving ML training, which can typically take a long time, and waste much resources if halted and restarted.

To tackle this issue, we need a mechanism that allows restoring the SMPC execution as soon as the minimum number of required parties are online and interacting with each other. Our proposed protocol aims to enable the SMPC parties to recover from a halt state, thus offering (improved) robustness for SMPC. The main idea behind our approach is to leverage saved states during the computation that the parties can load later for resuming the computation and providing access to an immutable history of communications to recompute the last portion of the SMPC quickly. While state saving can be less challenging to do for local computations within each SMPC node, reaching a state that depends on other parties is a major challenge. Moreover, although each node can have its state replicated to a backup node (potentially increasing the attack surface), a recovery protocol in such a scenario would require all parties to coordinate to return to the last common state, discarding a portion of the computation. Therefore, ReplayMPC consists of two stages. First, we introduce a *computation state*, essentially a snapshot of the SMPC variables stored by a

specific party at determined progress marks. Second, we present a novel re-computation mechanism for the recovering parties to make up the lost portion of the computation based on the data provided from the first stage. Once the computation state synchronizes with the other SMPC parties, they can resume the SMPC execution. This mechanism utilizes a blockchain with an efficient propagation network, which is the key element to (i) guarantee the integrity and authenticity of previously shared information among SMPC parties and (ii) not negatively impact the execution compared to SMPC without recovery features.

In the next subsections, we describe the details of the two stages of our protocol. The second stage has two cases which will be discussed separately.

### B. Saving the Computation State

During an SMPC computation, at the end of any given iteration, each SMPC party has a set of variables, including secret shares, that are unique for that specific party and iteration number. Additionally, depending on the specific SMPC protocol, the computation may consume raw materials (i.e., seemingly random but correlated data computed in advance) in a synchronized fashion with all the other parties. These raw materials may include different types of values (e.g., Beaver triples, matrix triples, secure random bits, and others).

Therefore, we define the *computation state* to include all the variables, counters, and indices that enables a party to resume a computation from a specific progress mark. This data needs to be saved locally and periodically. As such, when a recovering party restores this state in the first stage, it can identically reproduce, in the second stage, the next portion of the computation which was halted.

As an example, let us consider the case of ML training, where the SMPC parties go over the training dataset in a synchronized manner over several iterations. At each iteration the weights of the ML model are updated. Therefore, for this ML use case, the computation state includes, the current iteration number, the index of the data samples to process next, the indices of each raw material type, and the current value share of the ML model weights, as shown in Table I. Note that this data can be stored on a local database. The frequency of storing this data can be tuned based on how much delay the application can tolerate during a failure recovery. The impact of the integrity of this data will be discussed in the Security Analysis section.

TABLE I
COMPUTATION STATE CONTENT EXAMPLE FOR AN ML USE CASE

| Value | Type | Quantity |
|---|---|---|
| Model weight value-shares | BLOB | model-specific |
| Raw material indices per type | Integer | one per type |
| Dataset mini-batch index | Integer | 1 |
| Iteration number | Integer | 1 |
| Txn sequence number | Integer | 1 |
| Blockchain block number | Integer | 1 |
| SMPC job ID | String | 1 |

*Notes:* BLOB stands for Binary Large Object. Txn is an abbreviation for Transaction.

### C. Modeling Recovery via a State Machine

SMPC protocols generally specify each party's steps to evaluate a function. However, no specification generally indicates how to proceed when failures occur. For instance, when a message is not received, the parties would stay indefinitely waiting for the messages to arrive until a receive timeout is triggered, and the application will likely stop. Therefore, if a failure recovery is to be initiated, one needs to first detect the problem, understand its nature and then initiate a recovery based on the specific situation. This suggests that we can model the solution as a *finite state machine* to comprehensively cover all the cases.

Let us start with the modeling of a *normal* SMPC execution. A state machine would have two possible transitions in such a case: complete successfully and move to the *end* state or *abort*. However, a failure-aware SMPC system should define additional machine states that allow the *detection* of an abnormal condition which would *halt* the SMPC execution and initiates the *recovery* state in an attempt to resume the normal execution.

Considering such states in mind, we propose the finite state machine diagram for ReplayMPC shown in Fig. 2. We define four states (*Start*, *Normal*, *Halt* and *Recovery*) and two terminal states (*Stop* and *End*). Initially, a new SMPC job has no saved state, and the parties transition directly to a *normal* execution state after the *Start* state. During normal state any failure is possible. This can either cause the application to abort (i.e., transition to *Stop* state) or delay the message reception from other parties (i.e., transition to *Halt* state). These two states cover the *detection* part of our proposed recovery protocol as will be detailed below. In the case that the computation halts for some time or the application stops completely, the parties need to initiate the recovery task (i.e., transition to *Recovery* which can be direct or indirect via *Start* state). The party moves to *End* state if the computation is completed successfully.

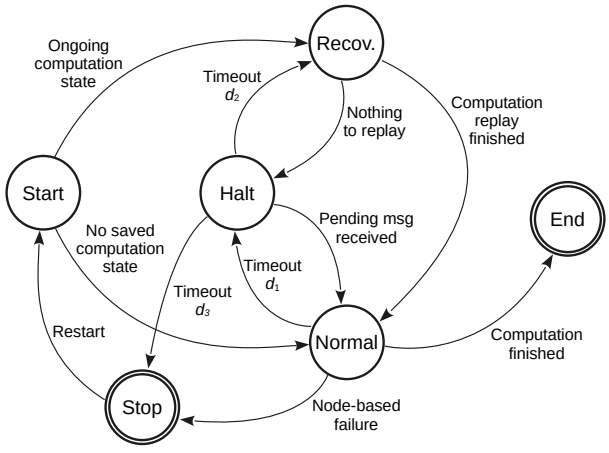
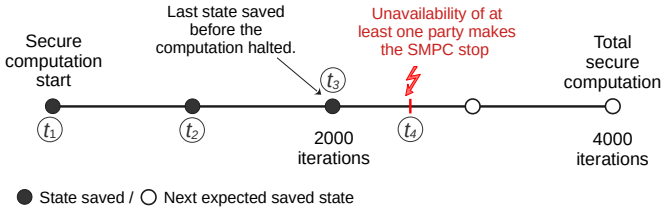


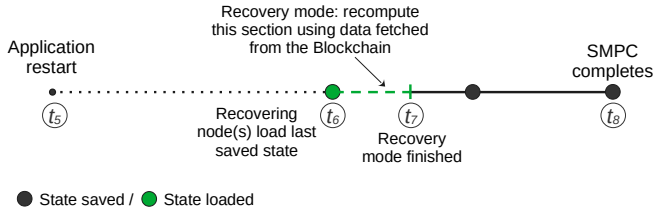

Fig. 2. Failure-aware SMPC System's Finite State Machine Diagram

Now let us focus on the detection mechanism to employ through an example. Let us assume that three parties $(P_1, P_2, P_3)$ perform an SMPC. The timeline for the events on $P_1$ is shown in Fig. 3(a). Note that $P_1$ would save computation

states at times $t_2$ and $t_3$. Now, let us assume that $P_1$ experiences an issue that stops the SMPC execution at time $t_4$. At that point, $P_2$ and $P_3$ keep waiting for messages from $P_1$ that will not arrive. Thus, we propose utilizing a communication timeout mechanism for the detection of an issue. The timeout denoted as $d_1$ in Fig. 2 can be set to a short time. At this point, $P_2$ and $P_3$ transition to *Halt* state and wait for another fixed amount of time (i.e., $d_2$) before they initiate a recovery protocol instead of aborting or keep waiting for messages indefinitely. Note that since $P_1$'s application stops (e.g., system-level failure, power outage, etc.), then it transitions directly from *Normal* to *Stop* state (i.e., in *Halt* state, the application is running and waiting for messages while in *Stop* state the application is not running).




(a) Normal Progress of a Secure ML Training




(b) Recovery Timeline

Fig. 3. Example Timeline for party $P_1$ of the Proposed SMPC Recovery Protocol

### D. Recovery Mechanism for Node-based Failure

If there is a node failure, the assumption is that the SMPC application will re-start from scratch on the failed node. Following our example in the previous subsection, this means that at time $t_5$, $P_1$ will verify whether it has a saved computation state upon restart. In case there is a saved state, $P_1$ proceeds to load it at time $t_6$, determines if it is an unfinished computation and restores the shares of the ML model weights, and the different indices from the database. $P_1$ also loads other stored information, such as an identifier for the current SMPC job, the iteration number, and the transaction sequence number.

After loading the last saved computation state, $P_1$ transitions to *Recovery* state (Fig. 2) between $t_6$ and $t_7$ as shown in Fig. 3(b). In this state, a recovering party reproduces a portion of the computation starting from the last saved state, until it progresses the computation to the last computed SMPC round, at which all the other SMPC parties are waiting. $P_1$ reproduces the computation by retrieving other parties' previous messages from the blockchain, which contains the history of all previous communications, starting from the block pointed to by the last saved state.

The reproduction of previous SMPC rounds is much faster in recovery mode compared to the case when the parties originally computed it. This is because the transactions are available from the relay nodes, and a single block may contain information for several SMPC rounds. Moreover, each SMPC node maintains the last 1000 blocks in its local database, which makes the retrieval process of the last rounds even faster since it is not affected by the network delay.

Once the recovering party $P_1$ has finished recomputing the last SMPC rounds, it can transition from the *Recovery* state to the *Normal* SMPC execution state at time $t_7$ (Fig. 3(b)). This transition typically occurs when $P_1$ sends the transaction(s) corresponding to the current round that the other parties expect to receive to resume the execution. $P_1$ can identify these missing transactions because they are missing in the retrieved blocks too. Finally, after $P_1$ submits the missing transactions, all parties resume the SMPC execution.

Fig. 4 presents the pseudo-code for the procedure followed by a crashed party to advance to the current computation state and resume the SMPC execution.

**Require:** On-recovery party that successfully loaded the SMPC application and reconnected to the rest of the nodes.
1: $state \leftarrow$ load_state()
2: $start\_block \leftarrow state.block\_number$
3: $stop\_block \leftarrow$ Current blockchain block
4: Party runs `recovery_mode_mpc`($start\_block$, $stop\_block$)
5: Party sends the pending SMPC messages
6: Interactive SMPC execution resumes

Fig. 4. Recovery Protocol for Restarting SMPC Node

Note that the above process is followed within the failing party. We need to also manage the state transition on the healthy parties. Initially, $P_2$ and $P_3$ move from *Halt* state to *Recovery* after a timeout $d_2$ as shown in Fig. 2. They do that to determine whether they are at fault or just need to keep waiting for the failed party to recover. In this case, $P_2$ and $P_3$ find out that they have no SMPC rounds to replay (i.e., recompute) and return to *Halt* state. When $P_2$ and $P_3$ are in the *Halt* state, they keep waiting for messages from the unresponsive party. They do so by making an API request to the local Blockchain process, but no new transactions are posted during this time. Therefore, the request for transactions from the local temporary pool returns nothing. Consequently, the SMPC process will make a new request again and again. The continuous submission of API requests increases the CPU usage of these hosts. We propose delaying the time between requests when the response is empty in this scenario. Additionally, we do not want to delay the execution unnecessarily during normal execution. Consequently, we propose an exponentially increasing timer from a low starting value up to a conservative maximum, which is similar to Ethernet's exponential backoff mechanism. For instance, we may want to approximate such maximum delay to a value lower than the block time; this way, the request will not miss transactions that were available from the local pool before the blockchain's consensus protocol packed them into blocks.

The healthy parties could generally wait for the failed party

to resume the computation indefinitely until the computation resumes. Alternatively, our protocol offers the option for healthy parties to save the current computation state and exit after a predefined maximum time $d_3$ has elapsed. This timeout can be adjusted based on certain parameters that may include the restart time of a machine. In such a case, the computation can be restarted at a later time when the failed party (or a substitute) is back online. Fig. 5 presents the pseudo-code for a general protocol followed by a healthy SMPC party during recovery.

**Require:** Computation stopped for $d_1$ time, and the party is in *Halt* state and not at fault.
1: $resume \leftarrow$ false
2: $abort \leftarrow$ false
3: $abort\_timer \leftarrow$ `Timer`($d_3$)
4: **while** not ($resume$ or $abort$) **do**
5: $abort \leftarrow$ `is_expired`($abort\_timer$)
6: **if** $abort$ **then**
7: save state & exit
8: **end if**
9: `wait`(*exponential_backoff_timer*)
10: $resume \leftarrow$ received pending messages?
11: **end while**
12: Party resumes interactive SMPC execution

Fig. 5. Recovery Protocol for the Healthy Parties

### *E. Recovery Mechanism for Network-based Failure*

A problem that may occur more frequently on SMPC is the loss of synchronization due to errors in the communication channel (e.g., a message not received or a packet malformed). Of course, the parties may not crash, but at the very least, they keep waiting for those messages, and thus the computation will abort eventually.

In this scenario, all parties first transition from the *Normal* to the *Halt* state (refer to Fig. 2) after a timeout $d_1$ and record the current blockchain block number. Next, the parties wait for a time $d_2$ before attempting a *Recovery* state. This is because the parties do not know which of them needs to take action to resume the computation. Therefore, they look inside the last few blocks in the blockchain ledger in order to find the messages they did not receive and the last messages they sent. The length of $d_2$ depends on how much time it takes new transactions to appear in the blockchain. For instance, in the case of Algorand, new transactions are typically confirmed after two blocks, which is equivalent to approximately nine seconds. In this *Recovery* state, the parties verify whether the problem is related to them and requires their action. After examining the last blockchain blocks, a party does not need to take additional action if the following holds: (i) the party does not find any new messages from the other parties, and (ii) the party finds all its last round messages. The parties (e.g., $P_2$ and $P_3$) that confirm themselves unrelated to the problem are the healthy parties and thus return to the *Halt* state. The rest of the protocol is the same as in the previous case.

On the other hand, the party (e.g., $P_1$) which determines that the problem is related to itself proceeds to repeat the last SMPC round (using the info on blockchain). For $P_1$, this happens at time $t_6$ in Fig. 3(b). Eventually $P_1$ transitions to the *Normal* state at time $t_7$. From this point, it can resume the normal SMPC execution with the other parties. Fig. 6 shows the algorithm for the faulty party.

**Require:** The faulty party records the current blockchain block number as *fail_at_block*.
1: $start\_block \leftarrow$ *fail_at_block*
2: `wait`($d_2$)
3: $stop\_block \leftarrow$ Current blockchain block
4: Party runs `recovery_mode_mpc`($start\_block$, $stop\_block$)
5: Party sends the pending SMPC messages
6: Interactive SMPC execution resumes

Fig. 6. Recovery Protocol for Network-related Loss of Sync

## V. Security Considerations

ReplayMPC enhances the specific SMPC system that implements it. As such, its provisions and protocols enable a crashed SMPC party to recover from a failure, either fortuitous or deliberately launched by an external entity (e.g., non-persistent DoS, attack on the infrastructure, or other attacks intended to prevent the SMPC execution from successful completion). Recall from Section III-B that the goal of dishonest behavior are different for SMPC and participation nodes. Therefore, since the specific SMPC protocol runs on top of this blockchain-based broadcast network, the particular protocol characteristics, threat model (e.g., malicious versus semi-honest adversary), and vulnerabilities are not impacted by the proposed recovery mechanism. For instance, a maliciously secure SMPC protocol with dishonest majority will still be secure (i.e., privacy protected) against $n-1$ corruptions, but the unavailability of a single party would still prevent the computation from completion. Nonetheless, we briefly discuss the implications of relying on information fetched from the blockchain to recompute a portion or all of an SMPC job.

During the regular SMPC execution, the blockchain stores the SMPC message exchanges as the payload of blockchain transactions. Therefore, the integrity of these records derives from the blockchain security guarantees. In other words, what the SMPC parties submit will be recorded in the blockchain. For instance, in the case of Algorand, the blockchain Proof-of-Stake (PoS) consensus algorithm guarantees the confirmation of transactions when honest participants of the consensus protocol hold two-thirds of the online stake [14]. As mentioned in Section III-B, a corruption of a participation node does not undermine privacy. The main effect of a possible corruption by the adversary would be to refrain from confirming authentic transactions, leaving them out of the blockchain. Therefore, by having at least two-thirds of the voting stake under the control of honest participation nodes, we can be assured that all messages exchanged are added to the blockchain and not modified.

However, an attacker may compromise the specific relay node SMPC parties are connected to and can change the response by something else so that the SMPC party performs recomputation using altered data. Note that this can also happen

if the attacker can modify the reply from the relay node on the communication channel during transmission. A countermeasure for this attack is to request such SMPC messages from two or more relay nodes and compare the responses.

Note that integrity attacks on the saved computation states are also possible. This will result in supplying a modified input to the SMPC computation, which is equivalent to having this party controlled by the adversary. Maliciously-secure SMPC protocols have mechanisms that verify the correctness of the result and abort if final verification fails.

## VI. Experimental Evaluation

### A. Experiment Setup

We set up a proof of concept SMPC system deploying four SMPC parties in the following Google Cloud Platform (GCP) locations: us-central1, us-west3, us-east4, and us-east1. We also selected Algorand as the blockchain that supports communication among parties because it provides a fast broadcast channel and block generation. Additionally, Algorand is open source, allowing a private (i.e., permissioned) network deployment. We deployed such a private Algorand blockchain network with one relay node also hosted on the cloud, specifically in us-central1. Each SMPC node is configured with 4 vCPUs, 4 GB of RAM and a 15 GB solid state drive (SSD). The relay node is configured with 4 vCPUs, 4 GB of RAM and a 20 GB SSD. Additionally, we assume a one-time availability of a coordination server which requests SMPC jobs to the SMPC nodes and receives the computation results at the end. This coordination server does not need to be hosted on the cloud.. We implemented ReplayMPC on top of the SPDZ protocol in Python, and conducted experiments with two to four SMPC nodes.

### B. Metrics and Benchmarks

We consider two metrics. First, we measure the *computation state saving overhead*, which is defined as the total delay added to the regular SMPC execution (i.e., without the recovery feature) due to saving the computation state periodically during the computation. Second, we assess the *recovery time*, which is the time it takes for a recovering party to 'catch up' with the current computation state of the rest of the parties. Specifically, this includes the time to load the last saved computation state and the re-computation time of the missing SMPC rounds. A special case occurs when there are no saved computation states, where the recovering party re-computes all the rounds from the beginning.

To the best of our knowledge, this is the first work that implements an SMPC recovery strategy. Therefore, we use the regular (i.e., without recovery mechanism) SMPC execution time as a benchmark. This way, we can measure the additional time that comes with our recovery approach.

### C. Machine Learning Use Case

We evaluated ReplayMPC when running a privacy-preserving ML training using the Condition Based Maintenance of Naval Propulsion Plants dataset [18] for Linear Regression. This dataset contains 16 features and over 11,000 samples. The size of this dataset requires an SMPC training time long enough to measure the recovery time across experiments. After feature selection and feature engineering, we reduced the dataset to ten features and used 75% of samples (i.e., 8952 samples) for the privacy-preserving ML training. Moreover, we used the mini-batch stochastic gradient descent algorithm with a batch size of 110 samples and 80 epochs.

### D. Performance Evaluation

*1) Overhead of Computation State Management on Regular SMPC:* Since any recovery requires accessing previously stored information, in the first experiment, we assessed the overhead of computation state savings when performing an ML training using SMPC. For each experiment, the parties are configured to perform 6560 iterations obtained by completing 80 training epochs with 82 iterations each. We configured the SMPC nodes to save the computation state every 410 iterations (or 5 epochs). This means that the nodes saved their computation state 16 times during the whole execution. In general, the selection of the period between saving of the states depends on how much time the users would be willing to tolerate in case of a failure. Recall that at each state, the nodes record the parameters indicated in Table I.

We repeated this experiment with different number (2, 3 and 4) of SMPC parties and present the total training time in Table II. Recall that the SMPC nodes are geographically distributed and the total execution time is dependent on the different delays between the SMPC parties and the Algorand relay node. Therefore, we give the maximum and minimum training times obtained for all combination of nodes in the SMPC set.

TABLE II
Total Private ML Training Time with Computation State Saving

| # of SMPC Parties | Training Time (min.) |
|---|---|
| 2 | 73.38 to 81.99 |
| 3 | 95.12 to 105.32 |
| 4 | 114.75 |

We present these total training times to be able to understand the weight of computation state overhead in the training time. We found that in average the computation state overhead ranges between 5.7 ms and 6.9 ms. Adding up the times for the 16 states, the total overhead of computation state saving takes a little over 100 ms in total. As can be seen from Table II, the total training time is over at least 73 minutes and thus the overhead is negligible compared to the total SMPC execution time.

We repeated this experiment by saving the computation state more frequently as shown in Table III. The results indicate that even frequent saving will not significantly impact the total training time.

*2) Experiments for Node-based Failure Recovery:* In this experiment, we simulated an SMPC execution which halts due to a node failure (e.g., system level, loss of power, etc.) that causes the SMPC application to abort and be temporarily

TABLE III
OVERHEAD OF COMPUTATION STATE SAVING

| # of Snapshots Saved | Total Overhead |
|---|---|
| 16 | 100.8 ms |
| 40 | 252 ms |
| 80 | 504 ms |

disconnected. We measured the recovery time starting from the moment the recovering party re-launched the SMPC application until the parties resumed the interactive execution. To this end, we stopped the participation of party #2 halfway through the training, specifically, at the iteration number 3430. Recall that the SMPC parties were configured to save their computation state every 410 iterations in these experiments. This means, party #2 saved its last computation state, before crashing, just after the iteration number 3280 (i.e., $8 \times 410$).

Upon re-launching the SMPC application, party #2 loads this computation state and replays the next 150 iterations to advance to the iteration number 3430 in recovery-mode, where it does not need to submit any transactions to the blockchain. Instead, party #2 reproduces the last computation rounds by fetching previous broadcast messages from the local copy of the ledger. Since it re-initiated the recovery protocol right after we stopped it, the history of the previous communications was still available within the last 1000 blocks of the ledger stored locally (in the same node) by default. Note that as an alternative the recovering party can fetch any messages from the relay node, even if they are not recent.

As shown in Table IV, this recovery took just a few seconds mainly due to only recomputing the last SMPC rounds. Loading the computation state is almost instantaneous. If there is no recovery mechanism, then all the parties would need to restart from scratch and run SMPC rounds until the iteration 3430. The execution times for this are reported in the first column of Table IV. As can be seen, ReplayMPC brings huge time savings which increases with the number of SMPC parties involved.

TABLE IV
SMPC REGULAR AND RECOVERY TIME FOLLOWING A TEMPORARY NODE OUTAGE

| # of SMPC Parties | SMPC w/o Recovery (min.) | ReplayMPC (sec.) |
|---|---|---|
| 2 | 38.37 | 9.3 |
| 3 | 49.73 | 10.9 |
| 4 | 60.4 | 12.4 |

*Note:* w/o is an abbreviation for 'without'.

*3) Experiments for Replacing the Failed Party:* There are cases when the failed party will not recover and a substitute party needs to recompute all the SMPC rounds from the beginning. This is because the new party does not have access to the failed party's saved computation state snapshots as well as the local blocks of the ledger.

In this experiment, we measured the recovery time when a new party fetches previously exchanged SMPC messages from the relay node. Naturally, the link delay between the SMPC party and the relay node will affect the performance of the computation replay. However, it is still much faster than the regular SMPC execution.

Table V compares the recovery time of the first 3430 iterations with respect to the regular SMPC execution for varying number of parties. Note that they are over 8 times as fast as the regular SMPC execution.

TABLE V
SMPC REGULAR AND RECOVERY TIME AIDED BY THE RELAY NODE

| # of SMPC Parties | SMPC w/o Recovery (min.) | ReplayMPC (min.) | Recovery Time Reduction |
|---|---|---|---|
| 2 | 38.37 | 4.98 | 87.0% |
| 3 | 49.73 | 6.12 | 87.7% |
| 4 | 60.4 | 7.15 | 88.2% |

These results indicate that a new party, even without having access to locally saved computation state, can use our blockchain-based recovery approach to recompute the previous SMPC rounds much faster and use fewer resources instead of starting the same SMPC job from scratch.

*4) Simultaneous Recovery of Several SMPC parties:* In this experiment we evaluated the performance impact of having more than one SMPC party fetching previous SMPC messages from the relay node during the recovery. We repeated the previous experiment using 4 parties and interrupted the computation at the iteration number 3430 to compare the results with the previous experiment. We aborted the participation of parties number #2 and #3 (equivalent to half of the SMPC set) and restarted the application simultaneously on both parties.

During the recovery, both parties fetch previous SMPC messages from the relay node. Therefore, we would expect to observe an increased delay due to having a single relay node handling requests from two SMPC parties. However, the results shown in Table VI indicate no variation in the recovery time of two parties compared to having just one party reproducing the last SMPC rounds. The recovery time reported by party #3 is slightly higher because its communication delay with the relay node is higher than the link delay from party #2. Nonetheless, its recovery time is also unaffected when it is the only party running the recovery protocol or simultaneously with another party.

TABLE VI
RECOVERY TIME FOR TWO SIMULTANEOUSLY FAILED PARTIES

| | Recovery Time for a single Party | Recovery Time for two simultaneously failed parties |
|---|---|---|
| Party 2 | 7.15 min. | 7.13 min. |
| Party 3 | 7.6 min. | 7.62 min. |

## VII. CONCLUSION

We introduced ReplayMPC, a novel and efficient blockchain-based failure recovery protocol that enhances the robustness of any SMPC protocol, especially those based on full-threshold secret-sharing techniques. The proposed mechanism allows crashed or temporarily out-of-synch parties to rejoin the current computation job, saving time and computation resources. We

implemented ReplayMPC in Python and tested it on an ML training using the SPDZ protocol. Our experiment results show a recovery time reduction of more than 87% compared to the case when the computation has to start from scratch. This is achieved without significantly increasing the delay of the normal SMPC execution.

## ACKNOWLEDGMENT

This research was partly supported by the U.S. National Science Foundation, award number US-NSF-1663051, and the Air Force Research Laboratory / Information Directorate (AFRL/RI) Internship program for summer 2021, Rome, NY.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

## REFERENCES

[1] X. Dong, D. A. Randolph, C. Weng, A. N. Kho, J. M. Rogers, and X. Wang, "Developing high performance secure multi-party computation protocols in healthcare: A case study of patient risk stratification," *AMIA Summits on Translational Science Proceedings*, vol. 2021, pp. 200–209, May 2021.

[2] P. Bogetoft, I. Damgård, T. Jakobsen, K. Nielsen, J. Pagter, and T. Toft, "A practical implementation of secure auctions based on multiparty integer computation," in *Financial Cryptography and Data Security*, G. Di Crescenzo and A. Rubin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 142–147.

[3] C. S. Jutla, "Upending stock market structure using secure multi-party computation," Cryptology ePrint Archive, Paper 2015/550, 2015. [Online]. Available: https://eprint.iacr.org/2015/550

[4] S. Wagh, D. Gupta, and N. Chandran, "SecureNN: 3-Party Secure Computation for Neural Network Training," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 3, pp. 26–49, Jun. 2019.

[5] N. Agrawal, A. Shahin Shamsabadi, M. J. Kusner, and A. Gascón, "QUOTIENT: Two-Party Secure Neural Network Training and Prediction," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 1231–1247.

[6] O. G. Bautista, M. H. Manshaei, R. Hernandez, K. Akkaya, S. Homsi, and S. Uluagac, "Mpc-abc: Blockchain-based network communication for efficiently secure multiparty computation," *Journal of Network and Systems Management*, vol. 31, 2023.

[7] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, p. 612–613, nov 1979. [Online]. Available: https://doi.org/10.1145/359168.359176

[8] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart, "Practical Covertly Secure MPC for Dishonest Majority – Or: Breaking the SPDZ Limits," in *Computer Security – ESORICS 2013*, J. Crampton, S. Jajodia, and K. Mayes, Eds. Berlin, Heidelberg: Springer, 2013, pp. 1–18.

[9] D. Lu, T. Yurek, S. Kulshreshtha, R. Govind, A. Kate, and A. Miller, "Honeybadgermpc and asynchromix: Practical asynchronous mpc and its application to anonymous communication," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 887–903.

[10] H. Gao, Z. Ma, S. Luo, and Z. Wang, "Bfr-mpc: A blockchain-based fair and robust multi-party computation scheme," *IEEE Access*, vol. 7, pp. 110 439–110 450, 2019.

[11] M. Rivinius, P. Reisert, D. Rausch, and R. Küsters, "Publicly accountable robust multi-party computation," Cryptology ePrint Archive, Paper 2022/436, 2022. [Online]. Available: https://eprint.iacr.org/2022/436

[12] A. R. Choudhuri, A. Goel, M. Green, A. Jain, and G. Kaptchuk, "Fluid mpc: Secure multiparty computation with dynamic participants," in *Advances in Cryptology – CRYPTO 2021*. Springer International Publishing, 2021, pp. 94–123.

[13] R. Xu, N. Baracaldo, and J. Joshi, "Privacy-preserving machine learning: Methods, challenges and directions," Sep. 2021, arXiv:2108.04417 [cs]. [Online]. Available: http://arxiv.org/abs/2108.04417

[14] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 51–68.

[15] H. Chen, M. Kim, I. Razenshteyn, D. Rotaru, Y. Song, and S. Wagh, "Maliciously secure matrix multiplication with applications to private deep learning," in *Advances in Cryptology – ASIACRYPT 2020*. Cham: Springer International Publishing, 2020, pp. 31–59.

[16] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen, "Asynchronous multiparty computation: Theory and implementation," in *Public Key Cryptography – PKC 2009*, S. Jarecki and G. Tsudik, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 160–179.

[17] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos, "SEPIA: Privacy-Preserving aggregation of Multi-Domain network events and statistics," in *19th USENIX Security Symposium (USENIX Security 10)*. Washington, DC: USENIX Association, Aug. 2010.

[18] A. Coraddu, L. Oneto, A. Ghio, S. Savio, D. Anguita, and M. Figari, "Condition based maintenance of naval propulsion plants data set," UCI Machine Learning Repository, 2014. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/condition+based+maintenance+of+naval+propulsion+plants