We introduce Helium, a novel framework that supports scalable secure multiparty computation (MPC) for lightweight participants and tolerates churn. Helium relies on multiparty homomorphic encryption (MHE) as its core building block. While MHE schemes have been well studied in theory, prior works fall short of addressing critical considerations paramount for adoption such as supporting resource-constrained and unstably connected participants. In this work, we systematize the requirements of MHE-based MPC protocols from a practical lens, and we propose a novel execution mechanism that addresses those considerations. We implement this execution mechanism in Helium, which makes it the first implemented framework to support MPC under network churn based solely on cryptographic assumptions. We show that a Helium network of 30 parties connected with 100Mbits/s links and experiencing a system-wide churn rate of 40 failures per minute can compute the product between a fixed $512 \times 512$ secret matrix (e.g., a collectively-trained private model) and a fresh secret vector (e.g., a feature vector) 8.3 times per second. This is $\sim 1500$ times faster than a state-of-the-art MPC framework operating under no churn.

CSCS CONCEPTS
- Security and privacy → Public key encryption; Privacy-preserving protocols.

KEYWORDS
Secure multiparty computation; Homomorphic encryption

ACM Reference Format:

1 INTRODUCTION
Cryptographic techniques for secure multiparty computation (MPC) can alleviate the need for trust between actors and enable collaborations that may otherwise be impossible due to privacy concerns. For example, MPC techniques have found applications in medical research [43], fraud detection [7], trading [40], and social sciences [46]. But the deployment of MPC is hindered by practical considerations related to the particularly resource-demanding nature of current MPC solutions, even in the semi-honest setting.

In this work, we focus on a long-standing problem in MPC systems research: performing semi-honest MPC tasks among computationally weak and unreliably connected parties [5, 15–17, 21, 32]. The current approaches for which open-source implementations exist, such as SPDZ [31], are mostly based on applying linear secret-sharing schemes (LSSS) [29] to the input data. They rely on interactive circuit-evaluation procedures, hence they require the protocol participants to be online and responsive, not only to provide their inputs, but for the whole duration of the MPC process. Moreover, they require a costly offline phase to generate secret-shares of random correlated values. The output of this phase constitutes a secret state that is non-compact (i.e., of size proportional to the circuit size), that is single-use (i.e., it must be re-generated for each circuit evaluation), and that must be stored reliably by the participants. These requirements are too stringent for many applications where parties are running on low-end devices (e.g., laptops, mobile phones) and over unreliable connections. As a way around the high requirements of MPC, many works employ ad hoc solutions such as non-colluding servers [18, 25, 31, 34] or secure hardware. But these unfortunately introduce non-cryptographic assumptions for security. We propose Helium, the first implemented framework that enables MPC among weak participants and under churn, without introducing non-cryptographic assumptions.

MHE-based MPC. To enable low-requirement MPC, Helium is based on Multiparty Homomorphic Encryption (MHE) [4, 36, 39] in the helper-assisted setting [39]. An MHE scheme enables a set of parties to encrypt their inputs in such a way (i) that enables arithmetic computation to be performed over the encrypted data, and (ii) that enforces a joint cryptographic access control over the plaintext data. MHE schemes can be instantiated into a natural MPC protocol which, in its theoretical formulation, comprises two phases: a setup
In summary, we make the following contributions:

- **Challenges Identification (§3.2).** We identify the challenges that MHE-based MPC poses in practice. These challenges include preserving the protocol’s efficiency, security, and liveness in the presence of resource-constrained and churning participants.

- **Generic Solution Design (§4).** We design a non-monolithic execution of the MHE-based MPC protocol, that addresses the identified challenges. We do so by defining an abstraction that captures all interactive operations in the MHE scheme and by designing a generic execution mechanism for those operations.

- **The Helium Framework (§5).** We instantiate our generic execution method in Helium, an end-to-end framework for helper-assisted MHE-based MPC which has very low requirements for the parties: they can run with just several hundreds of megabytes of RAM, their communication overhead is independent of the number of parties, and they do not need to be simultaneously online and reachable for the computation to make progress.

- **Implementation (§6).** We build our generic solution on top of the Lattigo library, evaluate it experimentally, and make it open-source.\(^1\) We show that Helium can compute large matrix operations (e.g., as required to perform privacy-preserving machine learning) among a large number of parties in tens of milliseconds, even when experiencing a high system-wide churn rate of 40 failures per minute.

Helium is the first available open-source implementation of a generic MHE-based MPC protocol, and is, de facto, the first framework for passive-adversary generic MPC under churn that relies solely on cryptographic assumptions.

## 2 SYSTEM MODEL

Let \(\mathcal{P} = \{P_1, \ldots, P_N\}\) be a set of \(N\) parties. Parties in \(\mathcal{P}\) are resource-constrained and inconsistently connected (see Churn Model below).

To execute the protocol, the parties receive assistance from a helper \(H\) which is assumed to run on high-end hardware and to be consistently online (e.g., a cloud server).

**Adversarial Model.** We assume a passive adversary that corrupts the helper \(H\) and a subset \(\mathcal{A} \subset \mathcal{P}\) of \(T - 1\) parties, statically and for a fixed threshold parameter \(T\) (i.e., any subset of \(\mathcal{P}\) having a size at least \(T\) is guaranteed to contain at least one honest party). In addition, the adversary can eavesdrop on all the network traffic.

**Churn Model.** We assume that the parties in \(\mathcal{P}\) can be either online or offline. In the online state, a party runs the Helium process and is connected to the network. In the offline state, the party does not run the Helium process and is disconnected from the network. We define a failure as the event of a party transitioning from the online to the offline state and consider that failure events follow a random process for which the system-wide failure frequency is \(\lambda_f\).

For a party experiencing a failure event, we model the time before transitioning back to the online state as a random variable for which the expected value is \(\lambda_f^{-1}\) (i.e., \(\lambda_f\) is the per-node re-connection rate). Finally, to model low-resource computing (in which devices might reboot, or processes might be killed by the operating system to save resources), we assume that a failure event entails the full erasure of a party’s volatile memory.

---

\(^1\)The code repository is accessible at https://github.com/ChristianMct/helium
2.1 Framework Requirements

Let $M$ be a plaintext space ring, $f : M^N \to M$ be an arithmetic function, and $x_i \in M$ be a private input held by party $P_i$.

**Functionality.** From $pp$ a set of HE scheme parameters, $C_f$ an HE circuit that computes $f$, and $(x_1, \ldots, x_N)$ the parties’ plaintext inputs, Helium computes $f(x_1, \ldots, x_N)$ and outputs the result. It does so by running a protocol $\Pi_{\text{Helium}}$ among the parties in $\mathcal{P}$ and $\mathcal{H}$.

**Input Privacy.** Informally, we require that the execution of Helium does not reveal more information about $(x_1, \ldots, x_N)$ than what $f(x_1, \ldots, x_N)$ does. We require that there exists an ideal simulator $S_{\text{Helium}}$ that simulates the interactions of the honest parties in the execution of $\Pi_{\text{Helium}}$. For $\kappa$ a security parameter, the adversary’s advantage in distinguishing between the real and simulated protocols must be no more than $2^{-\kappa}$.

**Scalability.** We require that, when executing Helium to compute $f$ as above, each party in $\mathcal{P}$ has: (i) a computation overhead that is at most linear in $N$ and $T$, and (ii) a communication overhead that is sub-linear in $N$ and $T$.

**Liveness.** We require that Helium makes progress on the computation in real time (i.e., it is not waiting for a disconnected party) whenever the number of connected parties is equal to or above $T$.

**Lightweight Clients.** The parties in $\mathcal{P}$ are assumed to run on low-end hardware akin to embedded systems. They have access to a low amount of volatile RAM (i.e., in the orders of hundreds of megabytes) and small persistent storage (i.e., in the orders of hundreds of kilobytes). In addition to being inconsistently online (see Churn model above), they do not have a public address in the network and cannot wait for incoming connections.

2.2 Framework User Interface

Helium is de facto a sub-component in a larger distributed system, which we refer to as the user application. We consider two phases in the user application life-cycle: the conception and the operation phase. In the conception phase, the user-application designer translates the setting, the adversarial model ($\mathcal{P}, T$), and the target function $f$ into a homomorphic circuit $C_f$ and a set of HE parameters $pp$. We voluntarily leave the aspects related to circuit design and HE parameterization outside the scope of this work. Although these aspects are important for assisting non-experts during the conception phase, they are orthogonal to our contributions and are addressed by the literature on HE compilers [3, 10, 14, 19, 24, 47]. In the operational phase, Helium is initialized at each party from the public parameters $pp$, the circuit $C_f$, the identity of the parties in $\mathcal{P}$, and the address of the cloud helper $H$. Helium then generates the computation outputs by executing the MHE-based MPC protocol, which we now describe.

3 MHE-BASED MULTIPARTY COMPUTATION

We first recall some background on MHE-based MPC in §3.1. Then, in §3.2, we isolate and characterize challenges that these protocols face in practice.

3.1 MHE-based MPC: Background

The main primitive of MHE-based MPC is an MHE scheme, which is instantiated in a higher-level MPC protocol. We recall the semantics of MHE schemes below and we refer unfamiliar readers to Appendix A for a more exhaustive description. We focus on the threshold family of MHE schemes, which we use in our work.

**Threshold MHE Scheme.** An MHE scheme over a plaintext space $M$ is a set of algorithms and protocols $\text{MHE} = \{\Pi_{\text{EncKeyGen}}, \Pi_{\text{EvalKeyGen}}, \Pi_{\text{Eval}}, \Pi_{\text{Decrypt}}\}$. The secret-key generation protocol $\Pi_{\text{EncKeyGen}}$ generates a secret key per party; it can be seen as distributing an (ideal) collective secret key among the parties, according to a $T$-out-of-$N$ threshold secret-sharing scheme. Thus, the secret-key-dependent operations of the scheme ($\Pi_{\text{EncKeyGen}}, \Pi_{\text{EvalKeyGen}}, \Pi_{\text{Eval}}, \Pi_{\text{Decrypt}}$) are multiparty protocols that require the participation of at least $T$ parties.

**MPC Protocol.** Protocol 1 ($\Pi_{\text{MHE-MPC}}$) describes the MHE-based protocol in its theoretical formulation. It comprises two phases, Setup and Compute, each of which consists of running several MHE protocols as sub-programs. During the Setup phase, the parties collectively run the MHE key-generation sub-protocols to generate the private and public key material required for the Compute phase: (i) a collective encryption-key $\text{cpk}$ (with $\Pi_{\text{EncKeyGen}}$) for encrypting the inputs, and (ii) all the evaluation keys required to evaluate the target function (with multiple calls to $\Pi_{\text{EvalKeyGen}}$, one per operation type used in the homomorphic circuit). Both protocols require the parties to sample fresh common random values, which can be done from a common random string (CRS) denoted by $\text{crs}$. During the Compute phase, the parties encrypt their inputs under the collective public key $\text{cpk}$, evaluate the target function under homomorphic encryption (with Eval), and collectively decrypt the result with $\Pi_{\text{Decrypt}}$.

In this work, we consider the helper-assisted setting for the $\Pi_{\text{MHE-MPC}}$ protocol [39], in which a helper node $H$ assists the parties in the protocol execution (see §2). The role of the helper is two-fold: (i) It acts as an evaluator, i.e., it computes the homomorphic circuit on the parties’ encrypted inputs during the Eval step, and (ii) it acts as an aggregator, i.e., it collects parties shares in the sub-protocols, aggregates them, and makes the result available to the parties. As a result, the communication overhead of each party no longer depends on $N$.

**Monolithic Execution.** Prior works [4, 39] assume that the $\Pi_{\text{MHE-MPC}}$ protocol is executed in four broadcast rounds (two for the PublicKeyGen step, one for the Input step, and one for the Output step), and that each round is executed as a monolith. This means that (i) the parties execute these rounds in a predefined order, (ii) for any round that involves multiple sub-protocols, the parties compute a single round-share as the concatenation of the involved sub-protocols’ shares, and (iii) the protocol terminates after the Output round.

The advantages provided by a monolithic execution are mostly theoretical: First, it simplifies the process by which the parties sample fresh common random values from the CRS: Since the sequence of sub-protocol execution is pre-determined, the required values can be read in the same sequence from a single CRS. This ensures that the randomness (i) is the same for each party (which ensures correctness) and (ii) is fresh for each sub-protocol (which
We now discuss the challenges arising when instantiating the MHE sub-protocols and hence are undesirable in our weak-participants setting. Indeed, both are expensive and, as a result, securely re-integrating the returning parties' secrets permanently alters the security guarantees within the offline parties' shares when needed [4]. But this approach does not compose well with our session-like approach because exposing parties’ secrets permanently alters the security guarantees within the session (i.e., by giving more than T − 1 shares of the secret-key to the adversary). As a result, securely re-integrating the returning party would require interaction initiated by all parties, to either renew their secret-key shares (in a proactive fashion [30] or to re-create a session from scratch. Indeed, both are expensive and, hence, are undesirable in our weak-participants setting.

Instead, we use the T-out-of-N-threshold scheme introduced by Mouchet et al. [36], which enables each MHE sub-protocol to be run with any T-subset of parties, without compromising the keys of the failing parties (and thus also without compromising the session). However, this scheme has two important limitations: (i) the set of T participating parties to each sub-protocol needs to be known before the parties can generate their shares, and (ii) if any of the T parties...
in the set crashes before sharing its share, the protocol cannot complete until this party reconnects. We address (i) by having the helper keep a view of the network and decide on the sets of $T$ connected parties for each protocol. By choosing the $T$ parties right before the protocol execution, the helper can greatly reduce (but not completely annihilate) the probability of the failure case of (ii). To fully address (ii), we also introduce a sub-protocol retry mechanism that lets the helper execute the same protocol over a different set of parties. However, sub-protocol retries are not captured by the existing security analyses of MHE-based MPC [4, 39], and we need to ensure that they do not break the security of the scheme. This issue was left unaddressed by the work of Mouchet et al. [36] and we describe it as Challenge 4.

**Challenge 4. Security under Churn:** We make the critical observation that the existing security analyses of MHE-based MPC [4, 39] do not hold under churn. This is not only due to our retry mechanism but also because parties can output correlated shares as a result of a state-less restart. For example, a party might crash while transmitting its share to a sub-protocol, then re-transmit a new, yet correlated, share to the same sub-protocol when coming back online. We show in §4.3 that, in current (RL)WLE-based MHE, this leads to possible cryptographic attacks that were, to the best of our knowledge, not discussed in the existing literature on MHE.

We address this issue by specifying how the various randomness sources required by the sub-protocols are seeded, and we ensure that this results in a *resettable* [28] variant of the sub-protocols. As a result, we obtain a solution that covers both the case of a party re-outputting a share and the case of protocol retries.

### 3.3 Roadmap

In the rest of this work, we propose a novel execution flow for the $\Pi_{\text{MHE-MPC}}$ protocol that addresses the aforementioned challenges. We proceed with a constructive approach: In §4, we propose a generic execution mechanism for MHE sub-protocols. This mechanism enables running those sub-protocols independently and provides both liveness and security under churn. In §5, we instantiate this generic execution mechanism in our helper-assisted model and complement it with circuit-evaluation capabilities. As a result, we obtain a complete, non-monolithic execution flow for the $\Pi_{\text{MHE-MPC}}$ protocol.

### 4 PROTOCOL EXECUTION MECHANISM

In this section, we present our mechanism for executing the sub-protocols of the $\Pi_{\text{MHE-MPC}}$ protocol (Protocol 1). We proceed in two steps: In §4.1, we define an abstraction for RLWE-based MHE sub-protocols. This abstraction enables us to define an execution flow that is generic across the sub-protocols. In this execution flow, that we present in §4.2 and §4.3, the sub-protocols are run independently in an efficient and churn-tolerant way. Indeed, the ability to run protocols independently will be key for addressing Challenge 1, while efficiency and fault tolerance are key to addressing Challenges 2-4.

#### 4.1 The PAT Protocol Abstraction

We define an abstraction that captures the core functionality of all secret-key-dependent MHE sub-protocols.

**Preliminaries.** The ciphertext space of the MHE scheme is a polynomial ring $\mathcal{R}$ in which the RLWE problem is hard [33]. Informally, for $s$ and $e$ two secret values sampled from low-norm distributions over $\mathcal{R}$, the distribution of $(s + e, a)$ is computationally indistinguishable from the uniform distribution over $\mathcal{R}^2$. At initialization, the MHE $\Pi_{\text{SecKeyGen}}$ protocol privately outputs, to each session node $P_i \in \mathcal{P}$, a $T$-out-of-$N$-threshold secret-share $s_i \in \mathcal{R}$ of the collective secret key $s$ (see [36]). More precisely, to each party $P_i$, MHE $\Pi_{\text{SecKeyGen}}$ outputs a point $(a_i, S(a_i))$ of some secret polynomial $S \in \mathcal{R}[X]$ of degree $T - 1$ for which $S(0) = s$. Hence, any subset $\mathcal{P}'$ of $\mathcal{P}$ with $|\mathcal{P}'| \geq T$ could reconstruct $s$ from their shares $(s_i)_{P_i \in \mathcal{P}'}$ as

$$s = S(0) = \sum_{P_i \in \mathcal{P}'} \prod_{P_j \in \mathcal{P}'} \frac{\alpha_j}{\alpha_j - a_i} s_i = \sum_{P_i \in \mathcal{P}'} \Delta_i^{(\mathcal{P}')} s_i, \tag{1}$$

where $\Delta_i^{(\mathcal{P}')}$ denotes the Lagrange interpolation coefficient for the share of party $P_i$ in the reconstruction among set $\mathcal{P}'$.

**The PAT Protocol Abstraction.** Although MHE schemes consist of many secret-key-dependent sub-protocols, all these protocols implement the same functionality at their core: they compute a noisy affine function of the secret key. Namely, they output a public value of the form $a + e + x$, where $s$ is the collective secret key, $x$ is a protocol-specific value, $a$ is a publicly known polynomial and $e$ is some small error term [39]. For example, the MHE $\Pi_{\text{EncKeyGen}}$ protocol generates a collective public encryption key of the form $(\rho_0, \rho_1) = (s\rho_1 + e_{\text{pk}}, \rho_1)$ by setting $x = 0$ and $a = \rho_1$ to be a uniform value sampled from the common random string. Similarly, the MHE $\Pi_{\text{Decrypt}}$ protocol performs the decryption of a ciphertext $(c_0, c_1)$ by computing $m_{\text{noisy}} = s_{c_1} + c_0 + e_{\text{dec}}$ (i.e., $a = c_1$, $x = c_0$), which can be decoded into the plaintext message $m$.

To compute this noisy product, any group $\mathcal{P}'$ of size at least $T$ exploits the linearity of Shamir’s secret-sharing: each party in $\mathcal{P}'$ computes and discloses its respective linear term plus some additional fresh error [36]. The disclosed value is referred to as the party’s share in the protocol. Due to the added fresh error, the shares are safe to disclose (i.e., they do not compromise the parties’ secret keys) under the RLWE assumption and the noisy product can be computed by summing up the shares. Hence, we say that the MHE protocols have public aggregatable transcripts, and we refer to them as PAT protocols. More formally, MHE protocols have a common structure that we express as a tuple $\text{PAT} = (\text{GenShare}, \text{AggShare}, \text{Finalize})$ of algorithms with the following syntax:

- **Share Generation:**
  $$v_i \leftarrow \text{PAT.GenShare}(s_i, a, \mathcal{P}', \chi)$$
  From a secret-key share $s_i$, a publicly known polynomial $a$ and a set of participating parties $\mathcal{P}'$, GenShare outputs a share $v_i = \Delta_i^{(\mathcal{P}')} s_i a + e_i + x_i$ where $e_i \sim \chi$ and $x_i$ is protocol-specific value. In the public-key generation and decryption protocols ($\Pi_{\text{EncKeyGen}}$ and $\Pi_{\text{Decrypt}}$), $x_i = 0$. In the evaluation-key generation protocols ($\Pi_{\text{EvalKeyGen}}$), $x_i$ is a function of $s_i$ that depends on the target operation.
- **Share Aggregation:** \( v_{\text{agg}} \leftarrow \text{PAT.AgShare}(\{v_i\}_{P_i \in P'}) \)
  From the shares \( \{v_i\}_{P_i \in P'} \) of the participating parties \( P' \), \text{AggShare} outputs a single aggregated share
  \[
  v_{\text{agg}} = \sum_{P_i \in P'} v_i = sa + \sum_{P_i \in P'} e_i + x_i
  \]
- **Finalization:** \( \text{out} \leftarrow \text{PAT.Finalize}(v_{\text{agg}}, \text{in}) \)
  From \( v_{\text{agg}} \) the aggregation of all shares of the parties in \( P' \) and in some public auxiliary input polynomial, \text{Finalize} outputs the result out of the protocol. In the key-generation protocols (\( \Pi_{\text{EncKeyGen}} \) and \( \Pi_{\text{EvalKeyGen}} \)), the auxiliary input in is the public polynomial \( a \) and \text{Finalize} outputs the resulting key as \( \text{out} = (v_{\text{agg}}, a) = (sa + e + x, a) \). In the decryption protocol \( \Pi_{\text{Decrypt}} \), the auxiliary input in is the element \( c_0 \) of the ciphertext and \text{Finalize} outputs the decrypted ciphertext as \( \text{out} = c_0 + v_{\text{agg}} \).

Overall, the execution of the \( \Pi_{\text{MHE-MPC}} \) protocol (see Protocol 1) reduces to the execution of many PAT protocol instances. More specifically, to evaluate a circuit \( f \), the parties first parse \( f \) and obtain: (1) a list of public keys required for the MHE.Encrypt and MHE.Eval algorithms; those correspond to the list of public-key generation PAT protocols to be run in the Setup phase, (2) a list of inputs to provide in the Compute phase, and (3) a list of decryption gates in the circuit: those correspond to the list of decryption PAT protocols to be run in the Compute phase. Then, the actual execution consists of executing all the PAT protocols in the lists.

### 4.2 PAT Protocol Execution Mechanism

In this section, we present an execution mechanism for PAT protocols. For the sake of the exposition, we make two assumptions in the scope of this section:

**Assumption 1: Ideal Coordinator.** We assume the presence of an ideal Coordinator, that orchestrates the execution of the PAT protocols. It does so (i) by maintaining a queue of protocols to be executed, (ii) by keeping track of the state of the network, and (iii) by triggering the execution of PAT protocols among the online parties. By abstracting the concrete coordinator implementation, we present the execution mechanism in more generality than required by Helium. This is because our execution mechanism applies beyond the helper-assisted setting with lightweight clients; it is therefore of independent interest. In §5, we lift the ideal coordinator assumption by instantiating the coordinator in the helper-assisted setting.

**Assumption 2: Restricted Churn Model.** We assume that our ideal coordinator has an accurate view of the network and, hence, can choose the set of parties \( P' \) participating in a given PAT protocol in a reliable way. Moreover, we assume that the chosen parties do not fail before providing their share. This rules out the possibility of a PAT protocol failing, but is indeed unrealistic in practice. We lift the assumption of a restricted churn model in §4.3.

#### 4.2.1 Nodes

We refer to all actors in our system as nodes, and denote the set of all nodes by \( N \). Each node is associated with an identifier (i.e., a unique string of characters provided by the high-level user application) and holds a public key certificate for that identifier. There are two types of nodes: the session nodes and the helper nodes. The session nodes have inputs to the MPC (i.e., they are the parties in \( P \)), hence they hold a share of the collective secret key (ensuring their inputs’ access-control). The helper nodes are nodes in \( N \) that do not have private inputs to the MPC and, hence, do not hold a share of the collective secret key; they simply assist the session nodes in the protocol execution. We denote the set of helper nodes by \( H \). In Helium (§5), we consider the helper-assisted setting with a single helper, i.e., \( N \supseteq P \cup \{H\} \).

#### 4.2.2 Sessions

Analogously to secure communication protocols such as TLS, we view MHE sessions as a long-lived logical secure multiparty computation context. We define the public session parameters as

\[
\text{PubSessParams} := \{\text{SessID, SessNodes, HEPrams, PublicSeed}\}
\]

where SessID is a unique identifier for the session, SessNodes is a list of identities of the \( N \) nodes in \( P \), HEPrams are the MHE scheme parameters, and PublicSeed is a public bit-string seed for the public randomness source. These parameters are set by the user application. We define the session parameters for node \( P_i \) as

\[
\text{SessParams}_i := \{\text{PubSessParams}, \text{PrivateSeed}_i, \text{sk}_i\}
\]

where PubSessParams are the public session parameters, PrivateSeed\(_i\) is a private bit-string known only to node \( P_i \) and \( \text{sk}_i \) is the node’s share in the MHE ideal secret key. These parameters are generated or read from a file system by the framework. Each node must securely store the session parameters, and we require that they suffice for a session node to recover a session correctly and securely, for example after a node crash.

From this point onward, our discussion focuses on a single session, and hence on a single instance of the \( \Pi_{\text{MHE-MPC}} \) protocol. For conciseness, we now refer to the \( \Pi_{\text{MHE-MPC}} \) protocol as the session and to its sub-protocols (i.e., the PAT protocols) simply as protocols.

#### 4.2.3 Roles

For each PAT protocol execution, the nodes may assume two different roles:

- The **Protocol Participants** are the \( T \) session nodes that provide a share (generated with the PAT.GenShare method) in the PAT protocol. We denote the set of participants as \( P' \subseteq P \).
- The **Protocol Aggregator** is a designated node that collects the \( T \) shares from all protocol participants and aggregates them with the PAT.AgShare method. Due to the publicly aggregatable property of PAT protocols (§4.1), this role can in theory be assumed by any online and reachable node in \( N \).

#### 4.2.4 Coordination Messages

To coordinate the execution of PAT protocols, our mechanism relies on two types of coordination messages. We now describe these messages, and detail how they are used by our execution mechanism in §4.2.5.

- A **protocol signature**, by analogy to programming languages, designates a PAT protocol prototype. It is defined as a tuple

\[
\text{PSig} := \{\text{PType, PArgs}\}
\]

\[
\text{PType} \in \{\Pi_{\text{EncKeyGen}}, \Pi_{\text{EvalKeyGen}}, \Pi_{\text{Decrypt}}\}
\]

where PType designates the type of protocol, and PArgs denotes the public inputs (i.e., the arguments) of the protocol. For example, a protocol that generates a public evaluation key for an operation op is represented by the signature \( (\Pi_{\text{EvalKeyGen}}, \text{op}) \), while

https://www.rfc-editor.org/rfc/rfc8446
a decryption of a ciphertext is represented by \( \Pi_{\text{Decrypt,ctid}} \)
where ctid is an identifier for the ciphertext. A protocol signature constitutes a description of the functionality of a PAT protocol.

- A protocol descriptor extends a protocol signature with a role assignment. It is defined as a tuple

\[
P\text{Desc} := \{P\text{Sig}, P\text{Parts}, P\text{Aggr}\}
\]

where P\text{Sig} is the protocol signature, P\text{Parts} is the set of \( T \) session nodes that provide a share in the protocol, and P\text{Aggr} is the identity of the aggregator for this protocol. The protocol descriptor constitutes an unequivocal description of a given PAT protocol execution. As such, it can be viewed as the runtime version of the protocol signature.

4.2.5 Protocol Execution. To execute a PAT protocol with signature \( \text{sig} \), the nodes proceed as follows:

1. The coordinator picks a set of protocol participants \( P' \) and a protocol aggregator \( P_A \). The participants in \( P' \) can be any online session node (i.e., \( P' \subseteq P \)) and the aggregator can be any online and reachable node (i.e., \( P_A \in P \cup H \)). Then, the coordinator sends \( P\text{Desc} = \{P\text{Sig} = \text{sig}, P\text{Parts} = P', P\text{Aggr} = P_A\} \) to all nodes in \( P' \cup \{P_A\} \).
2. Upon receiving \( P\text{Desc} \), each protocol participant \( P_i \in P' \) computes its respective share as \( \pi_i = P\text{Sig}.\text{GenShare}(s_i, a, P'; \chi) \) and sends it to the aggregator \( P_A \). The aggregator aggregates the received shares on-the-fly, with \( P\text{Sig}.\text{AggShare} \).
3. Upon receiving all the \( T \) shares for the participants in \( P' \), the aggregator reports to the coordinator that the protocol has been completed successfully.

At the end of this execution, any node in the system can, if required, obtain the output of the PAT protocol by querying the aggregator for \( v_{\text{agg}} \), and by calling \( P\text{Sig}.\text{Finalize} \).

Public Polynomials. Note that Step 2 above requires each participant to obtain the public polynomial \( a \) (see §4.1). We now discuss how this is implemented in our execution flow. A tempting solution would be to provide the public polynomial as a field of some synchronization message (e.g., the P\text{Sig}.\text{PArgs} field). However, this is an unsatisfying solution from a performance perspective because the size of the polynomial \( a \) is in the order of kilobytes to megabytes. Sending it as an argument would make the coordination messages significantly larger. By keeping the synchronization messages small, we open the possibility for re-connecting nodes to rebuild the state of the session by downloading a concise history of protocol descriptors from the coordinator. We will exploit this in our helper-assisted setting, in §5.1. Instead, we let protocol participants derive or retrieve the public polynomial \( a \), depending on the type of PAT protocol being executed:

- Decryption case: In the decryption protocol \( (\Pi_{\text{Decrypt}}) \), \( a \) is an element of a ciphertext and we let the nodes retrieve this element, only when needed, by interacting with the network. We further discuss this point when we present the data layer of Helium in §5.3.
- Public-key generation case: In the public-key generation protocols \( (\Pi_{\text{EncKeyGen}} \text{ and } \Pi_{\text{EvalKeyGen}}) \), \( a \) is sampled from the CRS, which can be done locally. For the security of PAT protocols to hold, the public polynomial must not be reused across multiple protocols, i.e., there must be a fixed mapping between protocols and fixed, non-overlapping, sections of the CRS. To instantiate a long enough CRS without having to store it (which would be very inefficient), we can use the common approach of expanding it from a keyed PRF that we seed with the session’s public seed \( \text{PublicSeed} \) (see §4.2.2). Although this approach is satisfactory for a monolithic execution [4, 39], it is not in our case. This is because mapping from protocols to the CRS sections requires taking into account that (i) new protocols can be executed at any time in our session-like execution, and (ii) that not all nodes are online and participate in all the protocols. As a consequence, our approach requires random access to the CRS. To implement this, we employ a technique akin to branching of the CRS for each protocol: To sample \( a \) for a protocol with signature \( P\text{Sig} \), each participant initializes a keyed PRF with the key \( \text{PublicSeed} \parallel P\text{Sig} \), where \( \text{PublicSeed} \) is read from the session parameters (i.e., the base CRS, see §4.2.2), and then samples \( a \) from this PRF (i.e., the branched CRS).

Security. We now sketch a game-based argument for the security of our execution mechanism. Although the argument is straightforward in our restricted churn model, it is a useful first step toward a security argument in the non-restricted churn model (§4.3). The main intuition is that, in the restricted churn model, our execution mechanism simply emulates a monolithic execution. More specifically, each protocol in an ideal monolithic execution can be mapped to a single protocol in our non-monolithic execution. The only difference is that the protocols are executed independently, with dynamically chosen participant sets. We model this additional degree of freedom as an interactive game.

Game \( G_0 \). Let \( \mathcal{A} \) be a protocol coordinator and let \( P \) be a challenger session node with collective secret-key share \( s \) in the execution mechanism of §4.2. The game is parameterized by a bit \( b \) unknown to \( \mathcal{A} \), and unfolds as follows:

1. \( \mathcal{A} \) is initialized with the \( \text{PubSessParams} \) and a list \( L \) of \( \Pi \) protocol signatures. \( P \) generates its \( \text{SessParams} \) from \( s \) and \( \text{PubSessParams} \).
2. \( \mathcal{A} \) performs queries to \( P \), where each query \( Q_j \) is a protocol descriptor \( P\text{Desc}_j = \{\text{sig}_j, P'_j\} \) with \( \text{sig}_j \in L \), \( P'_j \subseteq P \) and \( |P'_j| = T \). If \( b = 0 \), \( P \) retrieves \( a_j \) according to \( \text{sig}_j \) and responds with \( \text{sig}_j, \text{GenShare}(s, a_j, P'_j) \). If \( b = 1 \), \( P \) responds with a uniformly random value in \( \mathbb{R} \).
3. Eventually, \( \mathcal{A} \) outputs a bit \( b' \) and it wins if \( b' = b \).

The main idea follows that of previous works [4, 36, 39], but in a game-based formulation: If \( b = 0 \), the response from \( P \) to query \( P\text{Desc}_j \) is of the form \( \Delta, s_{a_j} + e_j + x_j \). Since the term \( s_{a_j} + e_j \) follows the RLWE distribution [33], \( \mathcal{A} \) obtains no distinguishing advantage given that \( a_j \) and \( e_j \) are independent across all challenger’s responses. In our presented execution mechanism and restricted churn model, each protocol signature is executed only once (this does indeed not hold in the full model, as we discuss in §4.3). Hence, we can restrict the adversary to only query each signature in \( L \) once, and we can guarantee the independence of the \( a_j \) by ensuring that \( L \) does not contain any duplicated value. Indeed, this guarantees that \( a_j \) comes from a different CRS branch or a different ciphertext for each protocol. Since \( e_j \) is sampled fresh for each protocol, they are independent across the queries.

Note that the monolithic execution can also be modeled by a similar yet more constrained game, in which the queries (including
their participant sets) are fixed in advance, and are submitted to the challenger in a pre-defined order. Hence, the extra capability of \( \mathcal{A} \) is reduced to breaking down, re-ordering, and changing the participants of the protocols in the monolithic execution, which does not provide any advantage.

**Liveness under Restricted Churn.** Our execution mechanism ensures liveness in our restricted churn model. This is because the coordinator chooses the participants of a given protocol based on its view of the network, and those participants are assumed not to fail before they provide their shares. This is indeed an unrealistic assumption, which requires to be lifted in order to fully address Challenges 3 and 4. This is done in the next section (§4.3).

4.3 Secure Churn Handling

Recall that, in the T-out-of-N-threshold scheme of Mouchet et al. [36], the set \( \mathcal{P}' \) of protocol participants must be known by the protocol participants to generate their shares. As a result, any protocol participant failing before or while providing its share would stale the protocol. In their work, Mouchet et al. observe that, when such failures are rare, the solution of simply re-trying the PAT protocol execution would be efficient. However, they leave the exact formulation of this failure-retry mechanism undefined. To address the liveness and security challenges under churn (Challenges 3 and 4), we must address the issue of stale PAT protocols. We now do so by instantiating a secure failure-retry mechanism within our non-monolithic execution mechanism.

**Protocol Retry Mechanism.** Assuming that the coordinator can detect PAT protocol failures (we discuss how such detection is done in Helium in §5), failure handling reduces to the ability of securely retrying protocols. From the semantic perspective, the retry of a protocol participant failing before or while providing its share would stale the protocol. In their work, Mouchet et al. observe that, when such failures are rare, the solution of simply re-trying the PAT protocol execution would be efficient. However, they leave the exact formulation of this failure-retry mechanism undefined. To address the liveness and security challenges under churn (Challenges 3 and 4), we must address the issue of stale PAT protocols. We now do so by instantiating a secure failure-retry mechanism within our non-monolithic execution mechanism.

\[ PDesc' = \{ PSig = PDesc.PSig, PParts = \mathcal{P}''', PAggr = P_A \} \]

(i.e., an equivalent protocol, yet with new participants). From the security perspective, however, there is an important consideration that a single protocol signature may require several PAT protocol executions. As a result, our execution mechanism further deviates from a monolithic execution by requiring additional shares to be disclosed by the parties. We must ensure that this does not compromise any of the parties’ secrets.

**Insecurity of Naïve Protocol Retries.** As a consequence of introducing protocol retries, executing a protocol signature may require several PAT protocol executions. This invalidates the assumption that each signature is executed only once, which is necessary to ensure the independence of the public polynomial (see §4.2).

This gap provides a non-negligible advantage to \( \mathcal{A} \) in the game \( \mathcal{G}_0 \). By submitting two queries for the same signature and different participant sets \( \mathcal{P}' \) and \( \mathcal{P}'' \) (i.e., the initially attempted protocol and its retry), \( \mathcal{A} \) obtains two dependent shares of the form \( \Delta^{(\mathcal{P}'' \setminus \mathcal{P})}sa + e + x \) and \( \Delta^{(\mathcal{P}'' \setminus \mathcal{P})}sa + e' + x \), for which the RLWE secrets are linearly related.

As a consequence, \( \mathcal{A} \) can distinguish between the two related shares and two random ring elements (e.g., by multiplying both shares by the inverse of their Lagrange coefficient).

**Secure Retries via Public Randomness Initialization.** To prevent the above failure case, we need to ensure that common random polynomials are fresh for each retry. To achieve this, we start by defining the protocol public seed as

\[ \text{ProtPubSeed} := \text{PublicSeed} \parallel \text{PSig} \parallel h(\text{PParts}), \]

where \( h : \text{PowerSet}(\mathcal{P}) \to \{ 0, 1 \}^* \) is an injective function that maps participant lists to bit-strings (recall that PublicSeed is loaded from the session parameters, and PSig is the protocol’s signature, see §4.2). This seed is unique for each possible PAT protocol instance and can be publicly computed. It can be viewed as branching the CRS, not only for each protocol signature, but for each possible protocol execution among \( \mathcal{P} \) (see §4.2.5). Then, ensuring that common random polynomials are fresh requires accounting for two cases:

- **Public-key generation case:** In the \( \Pi_{\text{EncKeyGen}} \) and \( \Pi_{\text{EvalKeyGen}} \) protocols, we let the parties sample the public polynomial by reading from a keyed PRF initialized with the protocol public seed ProtPubSeed. As a result, the CRS branch from which each public polynomial is read is unique to the protocol descriptor (i.e., the protocol execution), and not only unique to the signature anymore. Hence, from the perspective of CRS sampling, retries are simply considered as a new protocol.

- **Decryption case:** The \( \Pi_{\text{Decrypt}} \) protocol operates on an input ciphertext \( (c_0, c_1) \) and the parties produce shares of the form \( s + e + x \) for some secret polynomials \( s \) and \( e \). Thus, we cannot simply sample a different \( c_1 \) element for each protocol retry (because \( c_1 \) is taken directly from the ciphertext). Instead, we propose a mechanism to re-randomize the ciphertext. Re-randomization of a ciphertext can easily be achieved in additive-homomorphic schemes, by simply adding a fresh encryption of zero to the ciphertext. Such a zero-encrypting ciphertext can be generated from the session’s public key cpk, by running the MHE.\text{Encrypt} algorithm. This leads to a simple approach where a designated node (e.g., the coordinator) can generate a new re-randomized ciphertext for each retry. But this is unsatisfactory because it would require sending this new ciphertext to the participants at each retry. Instead, we employ a more efficient solution: We let the parties re-randomize the ciphertext non-interactively, by running the MHE.\text{Encrypt} over the common random string. More specifically, the parties sample the secret polynomials required by the encryption algorithm from a keyed PRF initialized with the protocol public seed ProtPubSeed. The security of using a publicly re-randomized ciphertext to generate RLWE samples follows from Lemma 4 in [8].

As a result of applying the techniques above to the provisioning of the public polynomial in PAT protocols, we restore the independence requirements in the scope of coordinator-initiated retries. More specifically, we ensure that the challenger always outputs independent shares for each query in game \( \mathcal{G}_0 \). However, we observe that this does not hold for one corner case: if the adversary is able to submit the same query twice. Although it appears as an active corruption, this corner case is in fact relevant to our setting and we discuss it next.

**Insecurity of Stateless Restarts.** In our setting, it is possible for a participant to participate twice in the same protocol (i.e., to output two shares for the same \((\text{sig}, \mathcal{P})\) pair). This is the case when a participant experiences a failure event during the transmission of
the first share, reconnects and obtains the session state before the failure is detected, and re-uploads a share for the same protocol. This can also be the case when the retry is attempted after the failed parties have reconnected and the same participant set is chosen.

In terms of our security game, this is captured by enabling $\mathcal{A}$ to submit two times the same query and to reset the challenger (from the session parameters). In our execution mechanism as described to this point, this lets $\mathcal{A}$ obtain two responses of the form $\Delta(P^i)sa+\epsilon+x$ and $\Delta(P^i)sa+\epsilon'+x$ from the challenger. These responses are close to equal but differ in a small error term; this enables the adversary to distinguish the challenger’s behavior from that of a random function. Note that, concretely, this directly enables an attack where the adversary can average the two shares to gain information on the $sa$ term (from which computing $s$ is easy).

To restore security in the presence of stateless restarts, it is necessary for our execution mechanism to ensure resettability of the executed PAT protocols. A trivial solution to ensure that the challenger behaves like a random function even after resets would be to require the nodes to write their shares in their persistent storage until the protocol is completed. Such a solution, however, would not only be inefficient, but it would also contradict the low-persistent memory requirement (see §2). Instead, we propose to further specify how PAT protocols sample their randomness.

**Resettability via Private Randomness Initialization.** In our setting, achieving resettability of PAT protocols requires their execution to be deterministic given the session parameters (i.e., the only persisting node state). This is already the case for the $\Delta(P^i)sa$ term. Hence, we need to further ensure that the error terms sampled are also deterministic (given the private session parameters). We achieve this by seeding the PRNG used to sample these terms: For each participant node $P_i$ in a given PAT protocol execution, we define the node’s protocol secret seed as

$$\text{ProtSecSeed}_i := \text{ProtPubSeed} \parallel \text{PrivateSeed}_i,$$

and let each participant use $\text{ProtSecSeed}_i$ as a seed to sample error terms in the protocol. Through this initialization, all participants use fresh secret values when generating their shares for each protocol and they never output two different shares for the same protocol in the same session. This addresses the corner case related to repeated queries by the adversary.

### 4.4 Addressing Practical Challenges

We now summarise how the PAT execution mechanism presented in this section addresses the challenges of §3.2.

**Non-monolithic Execution (Challenge 1).** Our execution mechanism enables a node to run PAT protocols independently, within a defined session. As a result, the coordinator can adapt the execution to the network’s condition, by running the protocols in a streamlined fashion and by limiting the number of concurrent protocols.

**Resource-Constrained Nodes (Challenge 2).** The aggregator assumes most of the overhead of a PAT protocol execution, as it receives and aggregates the $T$ shares. On the protocol participant side, the network overhead for each PAT protocol is constant and their computation overhead only weakly depends on $N$ and $T$ [36]. Moreover, the critical state that the session nodes have to store reliably, i.e., the session parameters (see §4.2.2), is also compact: its size is dominated by the node’s secret-key share, which is a single polynomial in $\mathcal{R}$.

### 5 HELIUM

We introduce Helium, an end-to-end implementation of the full $\Pi_{\text{MHE-MPC}}$ protocol (Protocol 1) that addresses the challenges described in §3.2. Helium instantiates the PAT protocol execution mechanism of §4 in the helper-assisted setting, complementing it with circuit evaluation capabilities.

**Helper-assisted Protocol Layer.** In Helium, we let the helper node $H$ assume the roles of the coordinator and aggregator in our PAT protocol execution mechanism. Indeed, these roles do not require any secret session parameter, hence can be assumed by any honest-but-curious party. However, it is crucial that these roles are assumed by reliable machines because they keep the state of the session (i.e., the list of executed PAT protocols and their respective outputs). This is aligned with our system model (§2), which assumes $H$ to be highly reliable in terms of availability, but a passive adversary in terms of threat model. Indeed, the assumption of a reliable (yet curious) node is easy to realize nowadays, as availability and reliability are the core features ensured by cloud-computing services.

This design choice brings several advantages: (i) it enables resource constraint nodes to keep optimal, constant overhead by offloading the bulk of the protocol’s overhead (i.e., the reception and aggregation of $T$ shares) to a powerful machine, (ii) it centralizes all the coordination and all the non-security-critical state storage to a single node, which considerably benefits practicality and ease of deployment, and yet (iii) it keeps all the security-critical state (i.e., the session parameters) decentralized, which ensures input-privacy relying on cryptographic assumptions.

**Two-Services: Setup and Compute.** Helium relies on a two-services design: The Setup service implements the Setup phase. The nodes query the Setup service to obtain public keys for encryption and evaluation (i.e., cpk and evk in Protocol 1, respectively). We describe this service in §5.2. The Compute service implements the Compute phase. It offers an interface for the user-application to evaluate circuits. We describe this service in §5.3. To execute the various PAT protocols required for their functionality, both the Setup and Compute services query a protocol layer, that we describe next.

### 5.1 Protocol Layer

We now specify how Helium instantiates the PAT protocol execution mechanism of §4 in the helper-assisted setting. In essence, the helper manages a queue of protocol signatures to be executed, which we denote SigQueue. To coordinate the execution of those protocols, the helper manages a public log $\text{PLog}$ of synchronization messages of the form: $\text{SynMsg} := \{\text{PDesc}, \text{PStatus}\}$, where PDesc is
a protocol descriptor and \( P\text{Status} \in \{\text{Started, Completed, Failed}\} \) is a status indicator for the protocol. This log enables the session nodes to have a complete view of the session’s progress. The helper can orchestrate execution by appending synchronization messages to \( P\text{Log} \). Figure 1 details the helper-assisted execution, for the session nodes (left) and the helper node (right). The execution consists of an initialization routine, Initialize, and three non-terminating routines: Coordinate, ExecutePAT, and Interface, which are executed by all nodes. Coordinate processes the protocol log \( P\text{Log} \), and sends tasks (i.e., PAT protocol descriptors) to ExecutePAT. Interface handles requests from the Setup and Compute services.

**Workload Control.** In Helium, nodes control the workload by setting a limit on the number of concurrently executing PAT protocols. The helper manages its own workload by controlling the pace at which the Coordinate routine picks new signatures from SigQueue. It should set this limit to a manageable memory overhead (for storing one aggregated share per protocol) and inbound traffic (for receiving the \( T \) shares). The session nodes manage their workload by controlling the pace at which the ExecutePAT picks protocol descriptors from its queue. They should set this limit so that they have enough memory to store one share per protocol.

**Failure Handling.** The Coordinate routine implements the failure-retry mechanism. It is parameterized by a protocol completion deadline \( \text{PDeadline} \), which corresponds to the time after which a non-completed protocol is considered stale. When idle, Coordinate looks for stale protocols and re-queues their signatures to launch a retry of their execution. Thanks to the PAT formalism and the techniques described in §4.3, retries can be seen as new equivalent protocols that are secure to execute. Helium implements a simple approach where the helper marks the stale protocol as Failed before re-queuing it, even though it is (in theory) possible that the protocol completes before its retry. In practice, however, since the routine is idling when scheduling retries, the retry is likely to be executed immediately. The rationale for scheduling retries in idle periods (i.e., rather than enforcing a strict deadline for protocol completion) is that failed nodes can reconnect before the session stops making progress due to stale protocols. Hence, to avoid the additional cost of a retry, it is better to wait until no more progress can be made before triggering a new execution. The session node’s Initialize routine also plays a role in failure handling. It reconstructs the protocol state for a node (re-)connecting to a session (for the first time or after a crash). Because shares are deterministic given a party’s session parameters, PAT protocols are resettable, and executing them after a stateless restart does not compromise security.

---

**Figure 1:** Pseudo-code description of the helper-assisted execution of PAT protocols in Helium.
5.2 The Setup Service

The Setup service offers an interface for obtaining the public keys required to encrypt and evaluate circuits within the session, i.e., it fulfills the role of the Setup phase of the Π_{MHE-MPC} protocol. To generate or retrieve public keys, the Setup service acts as a translation layer over the protocol layer: It translates the queried-key’s type into the signature PSig of the protocol that generates this key in the MHE scheme. Then, the service submits PSig to the Interface of the protocol layer, waits for the result (i.e., the public key), and returns it. For example, when queried for the collective public key cpk, the service queries PSig = \{PType = EncKeyGen, PArgs = (\{}\} to the protocol layers and returns the returned cpk. When queried for the evaluation key of operation op, the service queries for PSig = \{PType = EvalKeyGen, PArgs = (op)\} and returns the returned evk_op.

Persistence. In practical implementations, the service can provide more functionality. For instance, it can cache the result of relevant PAT protocols into a persistent storage, to reduce the network overhead of reboots. For example, the session nodes might cache the result of the Π_{EncKeyGen} protocol as they will use this collective public key to encrypt their inputs throughout execution. Conversely, results from the Π_{EvalKeyGen} protocols are not used by the session nodes (as the helper evaluates the whole homomorphic evaluation circuit), and hence should not be cached. Note that, when caching the results of key-generation PAT protocols, the nodes can simply store the aggregated share (agg in Figure 1) along with the protocol descriptor (i.e., they do not need to store the public polynomial). This is because the key can be recomputed from this aggregation, the public seed, the protocol descriptor, and the Finalize method for that protocol. Hence, the storage cost of persisting public keys is divided by two. However, we observe that, consistently with our framework requirements (§2), PAT output persistence is not necessary for nodes to safely and efficiently re-connect to the session.

5.3 The Compute Service

The Compute service offers an interface to execute circuits. It implements most of Helium’s user-facing interface, fulfilling the role of the Π_{MHE-MPC} Compute phase (see §3).

The interface of the Compute service lets the user-application register circuits of the form: \{CName, CDef\}, where CName is a string identifier for the circuit and CDef is the circuit definition (i.e., an HE circuit representation of the function \(f\)) in a Helium-readable format (we provide more detail on circuit representations in §5.3.1). After circuit registration, the Compute service interface lets the user-application submit circuit evaluation requests in the form of a circuit signature: CSig = \{CName, CID, CRecvr\}, where CName is a registered circuit name, CID is a unique identifier for that circuit execution, and CRecvr is the identity of the designated output receiver. Then, the service of this receiver node outputs the plaintext computation result.

Service Execution. The circuit execution mechanism is similar to the protocol layer of §5.1. The helper node holds a queue of circuit signatures and maintains a log of started and completed circuits. By tracking this log, the session nodes can send their encrypted inputs when required. To obtain the required public keys, the Compute service makes queries to the Setup service.

At the session nodes, the Compute service queries the public encryption key cpk from the Setup service and then tracks the circuit execution log. Upon receiving a circuit signature CSig := \{CName, CID, CRecvr\}, it encrypts the node’s input with the cpk, then sends the inputs to the helper node. Then, if \(Ps = CRecvr\), the service waits for a completion message in the log and queries the helper for the result.

At the helper node, the Compute service queries the public evaluation key evk_op for each op required in the registered circuit(s). In parallel, it triggers the execution of circuit by appending circuit signatures to the circuit execution log. To decrypt the outputs, the Compute service queries the protocol layer with signatures of the form PSig = \{PType = Decrypt, PArgs = (CTLabel)\}, where CTLabel is a unique identifier in Helium’s data layer (see below) that corresponds to the ciphertext to decrypt.

Data Layer. All inputs and output values in Helium circuits are MHE ciphertexts, and Helium uses a unique identifier for each of these ciphertexts. Hence, the data layer of Helium consists of a simple key-value store that is hosted at the helper, and a traditional put and get interface (e.g., HTTP, or more advanced RPC protocols) for parties to upload their inputs and download their results.

We observe that the ability to exploit existing (plaintext-data) paradigms for our data layer is a considerable advantage of the MHE-based MPC approach. Indeed, whereas the data layer of LSSS-based MPC solutions is fundamentally decentralized and interactive, the data layer of MHE-based solutions only involves storing and retrieving centralized values (i.e., HE ciphertexts). To further exploit this advantage Helium uses a URI scheme as a ciphertext-identifier.

5.3.1 Circuit Representation. As for any MPC system, the evaluation mechanism takes as input the representation of the target circuit in some language and acts as an (interactive) interpreter for this language. In the case of MHE-based MPC, the circuit is simply a traditional HE circuit, with extra labels on the input and output wires to designate the providers and receivers, respectively. At this time, there exists no well-established language that is specifically designed to represent HE circuits. Thus, for our current implementation, we provide the user application with a Go interface for building MHE circuits. This interface (named helium.Session) exposes the usual HE operations (in our case, those provided by the

```go
```

Listing 1: The Helium program for two-party component-wise vector multiplication.
Lattigo library interface), as well as IO primitives (i.e., labeled input and output gates). Listing 1 provides an example of a simple Helium program for computing a component-wise vector product between two parties. The Go-interface-based approach has three benefits: First, we do not need to implement a specific interpreter. Instead, Helium exploits Go’s execution directly. Second, we enable the user application to fully control the circuit execution flow, including exploiting Go’s built-in parallelism primitives and hardware accelerators. Third, it does not preclude the user from designing their own language and interpreter, as long as this can be initialized and executed from a Go function.

6 IMPLEMENTATION AND EVALUATION

We implement Helium in Go. We rely on the Lattigo library [37] for the cryptographic operations and on the gRPC framework for the transport and service layers. The gRPC framework offers a remote procedure calls (RPC) abstraction, which is ideal for capturing and expressing the interactions among the nodes. This is because our \( \Pi_{\text{MHE-MPC}} \) protocol execution mechanism in the helper-assisted setting requires only client-server interactions in single-round protocols and server-to-client streaming for the synchronization messages. Our implementation also lets the user provide its own transport layer, through a generic interface.

Our experimental evaluation has two parts: In §6.1, we benchmark Helium in a network with low bandwidth and memory-limited clients, but without any churn yet. This enables us to run another MPC framework based on LSSS in the same setting and use it as a comparison baseline. In §6.2, we benchmark the performance of Helium under churn. Finally, in §6.3 we discuss our experimental results and the relevance of the helper-assisted MPC setting.

Experimental Setup. We use Docker containers to run all nodes over two machines with Intel Xeon E5-2680 v3 processors (2.5 GHz, 2×12 cores), 256GB of RAM, and connected using a LAN network of 30Gbits/sec. with a latency of 0.1ms. All containers executing the session nodes are running on the first machine, and the helper node runs on the second machine. At initialization, each session node \( P_i \) is started with an already established session SessParams (see §4.2.2), but no PAT protocol has been run yet (i.e., no public-key has been generated yet). The helper node is initialized with the session public parameters. To simulate low-end network conditions, we limit the bandwidth of each session-node container to 100Mbit/s and introduce an artificial delay of 30ms. We also limit the memory assigned to each session node’s container to 128MB.

MPC Task. We consider the task of a matrix-vector multiplication over a prime field. More specifically, we consider a scenario where the parties collectively hold a secret \( 512\times512 \) matrix \( M \), and a single party \( P_0 \) wants to obtain the product \( y = Mx_0 \) for some private input \( x_0 \). For example, the \( M \) could be a private linear model trained in a previously executed MPC task, and our considered MPC tasks correspond to evaluating this model over some private inputs \( x_0 \). In Helium, the helper already holds a collectively encrypted matrix \( M \). In the LSSS baseline, the parties already share their \( M \).

Figure 2: Cost per matrix-vector multiplication. Time (left) and per-party communication (right) for an increasing number of session nodes and several MPC implementations. Hemi, Semi, Soho, and Temi refer to the semi-honest protocols implemented in MP-SPDZ [31]. Average over 10 runs.

Parameters. We consider the BGV scheme with a ring degree of \( 2^{12} \) and a coefficient size of 109 bits. According to the current estimates, this corresponds to a security of 128 bits [1]. We use a 16-bit prime (79873) as our plaintext modulus.

Artifacts. We publish the code for the presented experiments in a separate repository. This repository imports Helium, implements the presented MPC task, and provides scripts for running the experiments. We further describe the artifacts in Appendix B.

6.1 Experiment I: Lightweightness & Scalability

In this experiment, we evaluate Helium’s execution when there is a large number of resource-constrained participants. We do not consider any churn and set the threshold to \( T = N \). Hence, we can use an existing LSSS-based MPC implementation as a baseline.

More specifically, we consider the semi-honest, dishonest majority protocols implemented in the MP-SPDZ library [31] at v0.3.6. Figure 2 shows the wall time and network traffic per circuit evaluation (i.e., per matrix-vector multiplication), at party \( P_0 \) (the result receiver). For comparison, we also show the per-circuit cost for the LSSS-based MPC protocols. We observe that Helium achieves its scalability and lightweight goals. This is mainly due to the properties of the MHE-based MPC protocol for which the network cost does not depend on \( N \). We also observe that the setup latency and time per circuit evaluation have a very weak dependency on \( N \), because the network communication still dominates the cost of aggregating the \( N \) shares in the PAT protocols.

6.2 Experiment II: Churn Tolerance

In this second experiment, we evaluate the performance of Helium under churn. We consider the same circuit and parameter as in the first experiment, yet this time for a computation among \( N = 30 \) session nodes. To the best of our knowledge, no MPC framework available to date supports churning parties. Moreover, 30-party computation would take a significant amount of time for LSSS-based systems even without churn. We therefore analyze our results compared to theoretical baselines.

---

3https://golang.org
4https://grpc.io
5https://docker.com

6Accessible at https://github.com/ChristianMct/helium-artifacts
For this experiment, we consider a Markovian failure-recovery model in which the nodes have a fixed probability to fail (respectively, re-connect) at each epoch, independently. We denote this probability $\lambda_f$ (respectively, $\lambda_r$), and we denote the number of online nodes in a given epoch as $N_{\text{online}}$. A property of this model is that the expected number of online nodes converges over time to an equilibrium $E[N_{\text{online}}] = N \frac{\lambda_r}{\lambda_f + \lambda_r}$. To match our failure model (see §2), we can compute the system-wide failure rate (at equilibrium) as $\lambda_f = \lambda_f \cdot E[N_{\text{online}}]$. We start all experiments with a random set of $E[N_{\text{online}}]$ online nodes. To simulate the churn, we kill and restart session nodes’ containers according to our failure model.

Figure 3 shows the performance of Helium for a system of $N = 30$ nodes, an increasing system-wide failure rate, and a fixed average re-connection time $\lambda_r^{-1} = 20$ [sec]. As a theoretical baseline, we consider the ideal execution time: the measured execution time for fixed $N_{\text{online}} = T$ (i.e., the churn-free execution time), divided by $Pr[N_{\text{online}} \geq T]$ (i.e., the expected fraction of time for which the system has at least $T$ online nodes). For our churn model, we have $Pr[N_{\text{online}} \geq T] = \sum_{t=0}^{T-1} \binom{N}{t} (\frac{\lambda_r}{\lambda_f + \lambda_r})^t (1 - \frac{\lambda_r}{\lambda_f + \lambda_r})^{N-t}$.

We observe that Helium successfully satisfies the churn-tolerance requirement of §2: For failure rates below the $E[N_{\text{online}}] \geq T$ threshold (plotted as dashed lines), we observe that the latency is close to the ideal one. Furthermore, whenever $Pr[N_{\text{online}} \geq T]$ is close to zero, the latency is close to linear in the failure rate with a very small slope. This is because few parties’ failures actually cause a PAT protocol failure (as the crashing parties might have already provided their shares or might not be involved at all in the currently running protocols). This observation is corroborated by the fact that the factor increases with $T$ (which increases the probability that a given party participates in a given protocol).

6.3 Relevance of Helper-assisted MPC

In both experiments, we demonstrate a new level of practicality for MPC. In Experiment I, we show that Helium scales to large numbers of parties, even when these parties are resource-constrained or have limited connectivity. In Experiment II, we demonstrate that our failure-handling mechanism tolerates high churn rates (i.e., in the order of one failure/re-connection per second) and, thus, enables performing MPC tasks live whenever the expected number of online nodes is above the threshold.

This new level of practicality is mainly made possible by Helium’s helper-assisted setting, which leads to one of the key messages of our work: In the current state of the art, MPC can only be performed by investing a considerable amount of resources (w.r.t. an equivalent plaintext computation). In resource-constrained settings, the only practical way of running MPC is by outsourcing the incurred processing, storage, and availability costs to third parties. MHE-based MPC, as instantiated in Helium, enables such outsourcing without introducing non-cryptographic assumptions.

7 RELATED WORK

Secure Multiparty Computation. Resilience to crashes and disconnections is a fundamental problem in MPC. Prior theoretical works have proposed solutions to this problem by relying on techniques such as Shamir’s secret sharing and error correcting codes which ensure that honest parties obtain the correct output of the MPC functionality [6, 20–23, 27] or that a dynamic set of participants can execute the computation task (known as fluid MPC) [15, 41]. Although these works generally consider stricter security models (e.g., covert and active security), it is still unclear how feasible their implementation is in practice. A notable exception is HoneyBadgerMPC and Asynchrnomix [32], which has a public implementation. However, it assumes a reliably-performed offline phase and its focus on malicious security makes it incompatible with lightweight participants. Other existing MPC implementations such as MP-SPDZ [31], MOTION [9], and HyperMPC [5], do not provide satisfactory solutions for MPC under churn in the general case: they do not tolerate node disconnections, and their execution incurs high bandwidth and memory overhead to the participants. To circumvent these limitations, most of the practical uses of MPC techniques to date rely on delegation to two servers [18, 25, 34] that are assumed not to collude. While Helium also uses delegation to achieve sub-linear costs for the participants, it solely relies on cryptographic assumptions to do so in the honest-but-curious model.

Verifiable (M)HE. Recently, significant advances have been made in verification techniques for MHE operations [12], HE encryption [13], and HE evaluation [2, 11, 48]. Integrating these methods in Helium is an interesting avenue for future work to extend it beyond the passive adversary setting.

8 CONCLUSION

Deploying MPC protocols in practice is notoriously challenging. Existing MPC frameworks require a large bandwidth and assume high availability of the participants. Helium is a significant leap toward practical MPC, as it enables scalable, lightweight, and churn-resistant MPC in challenging environments. Moreover, Helium is a milestone for the study of MHE-based MPC. This work is, to the best of our knowledge, the first one to consider the security implication of failures in RLWE-based MHE systems, and it stands out as their first open-source end-to-end implementation.
Acknowledgments

We thank the anonymous reviewers of the CCS 2024 program and artifact-evaluation committees for their valuable feedback. We thank Jean-Pierre Hubaux for early feedback on this work, and Clémence Altmeyerhenzien, Manon Michel, Adrian Cucos, and Giovanni Torrisi for their work on the various code prototypes that evolved into Helium. Christian Mouchet received support from the German Federal Ministry of Education and Research (BMBF) through funding of the ATLASS project under reference number 16KIS037. Apostolos Pyrgelis was supported by the Horizon Europe project HARPORCATES (Grant agreement 101069535).

REFERENCES


Helium: Scalable MPC among Lightweight Participants and under Churn

CCS ’24, October 14–18, 2024, Salt Lake City, UT, USA

A MHE SEMANTICS

Let $\mathcal{P}$ be a set of $n$ parties, and let the threshold $T$ be the size of the smallest subset of $\mathcal{P}$ that is guaranteed to contain at least one honest party. Given a plaintext space with arithmetic structure $M$, an MHE scheme over $\mathcal{P}$ and $M$ is a tuple of algorithms and multiparty protocols $\text{MHE} = (\text{GenParam}, \Pi_{\text{SecKeyGen}}, \Pi_{\text{EncKeyGen}}, \Pi_{\text{EvalKeyGen}}, \Pi_{\text{Encrypt}}, \Pi_{\text{Eval}}, \Pi_{\text{Decrypt}})$ with the following syntax:

- **Public parameters gen.**: $pp \leftarrow \text{GenParam}(\kappa, \mathcal{P}, T, F)$
  
  Given the security parameter $\kappa$, the set of the parties in $\mathcal{P}$, the threshold $T$, and a set $F$ of arithmetic functions $f : M^T \rightarrow M$, $\text{GenParam}$ outputs a public parameterization $pp$. It is an implicit argument to the following algorithms and protocols.

- **Secret-key generation**: $(sk_i)_{P_i \in \mathcal{P}} \leftarrow \Pi_{\text{SecKeyGen}}()$
  
  From the public parameters, $\Pi_{\text{SecKeyGen}}$ outputs a secret-key $sk_i$ to each party $P_i \in \mathcal{P}$.

- **Encryption-key gen.**: $cpk \leftarrow \Pi_{\text{EncKeyGen}}((sk_i)_{P_i \in \mathcal{P}^*})$
  
  From any subset of secret keys $(sk_i)_{P_i \in \mathcal{P}^*}$, $\Pi_{\text{EncKeyGen}}$ outputs a collective public encryption key $cpk$.

- **Eval. key gen.**: $evk_{op} \leftarrow \Pi_{\text{EvalKeyGen}}(\text{op}, (sk_i)_{P_i \in \mathcal{P}^*})$
  
  Given a homomorphic operation $op$ to be supported by the Eval algorithm and any subset of secret keys $(sk_i)_{P_i \in \mathcal{P}^*}$ such that $(\mathcal{P}^*) \subseteq \mathcal{P}$ and $|\mathcal{P}^*| \geq T$, $\Pi_{\text{EvalKeyGen}}$ outputs a public evaluation-key $evk_{op}$ for operation $op$.

- **Encryption**: $ct \leftarrow \Pi_{\text{Encrypt}}(m, pk)$
  
  Given the public encryption key $pk$, and a plaintext $m \in M$, $\Pi_{\text{Encrypt}}$ outputs a ciphertext $ct$ that is the encryption of $m$.

- **Evaluation**: $c_{res} \leftarrow \Pi_{\text{Eval}}(f, (evk_{op})_{op \in F}, ct_1, \ldots, ct_f)$
  
  Given an arithmetic function $f : M^T \rightarrow M$, the evaluation key $evk_{op}$ for each homomorphic operation $op$ used in $f$ and an $l$-tuple of ciphertexts $(ct_1, \ldots, ct_f)$ encrypting $(m_1, \ldots, m_l) \in M^l$, $\Pi_{\text{Eval}}$ outputs a ciphertext $c_{res}$ that is the encryption of $m_{res} = f(m_1, \ldots, m_l)$.

- **Decryption**: $m \leftarrow \Pi_{\text{Decrypt}}(ct, (sk_i)_{P_i \in \mathcal{P}^*})$
  
  Given ct an encryption of $m$, and any subset of secret keys $(sk_i)_{P_i \in \mathcal{P}^*}$ such that $(\mathcal{P}^*) \subseteq \mathcal{P}$ and $|\mathcal{P}^*| \geq T$, $\Pi_{\text{Decrypt}}$ outputs $m$.

Current MHE scheme constructions [39] are based on the ring-learning-with-errors (RLWE) problem [33]. The plaintext space of such schemes is a ring of polynomials of fixed (power-of-two) degree. Their Eval algorithm supports additions and multiplications in this ring. They also support homomorphic rotations over the coefficients of the message. Each homomorphic operation (besides the addition) requires its own evaluation key to be provided to the Eval algorithm, hence it requires the execution of a separate instance of the $\Pi_{\text{EvalKeyGen}}$ protocol. We note that the “rotation of $k$ positions” operation is considered an individual operation for each required value of $k$ in the circuit, and it is common for applications to generate many such evaluation keys. This is because rotations are costly and achieving a desired rotation by composition (e.g., of many rotations by $k = 1$) is often impractical.

B ARTIFACTS

The main artifact of this work is the implementation of Helium as a Go package. The package complies to the Go documentation standards and more details can be found in the repository’s README.

The code is:

- hosted at https://github.com/ChristianMct/helium,

This implementation is meant to be reusable and might be further extended, developed, and maintained.

A secondary artifact is the implementation of the experiments of §6. It consists of a Go application implementing the experiment on top of Helium and various scripts to build and run the experiments. Detailed instructions to run the experiments can be found in the repository’s README. The code is:

- hosted at https://github.com/ChristianMct/helium-artifacts,

The purpose of this code is solely to (re-)produce the results of this work and there is no plan to further maintain it.