Helium: Scalable MPC among Lightweight Participants and under Churn

Christian Mouchet*
Hasso Plattner Institute
christian.mouchet@hpi.de

Sylvain Chatel*
CISPA Helmholtz Center for Information Security
sylvain.chatel@cispa.de

Apostolos Pyrgelis
RISE Research Institute
apostolos.pyrgelis@ri.se

Carmela Troncoso
EPFL
carmela.troncoso@epfl.ch

ABSTRACT

We introduce Helium, a novel framework that supports scalable secure multiparty computation 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 participants and ensuring liveness and security under network churn. 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 in Helium, which makes it the first implemented solution that effectively supports sub-linear-cost MPC among lightweight participants and under churn. This represents a significant leap in applied MPC, as most previously proposed frameworks require the participants to have high bandwidth and to be consistently online. We show that a Helium network of 30 parties connected with a 100Mbits/s link and experiencing a system-wide churn rate of 40 failures per minute can compute the product of a fixed secret 512×512 matrix (*e.g.,* a collectively trained model) with an input secret vector (*e.g.,* a feature vector) 8.3 times per second. This is ~1500 times faster than a state-of-the art MPC implementation *without* churn.

KEYWORDS

secure multiparty computation, homomorphic encryption

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 [42], fraud detection [7], trading [39], and social sciences [45]. But the deployment of MPC is hindered by practical considerations related to the particularly resource-demanding nature of current MPC solutions.

In this work, we focus on a long-standing problem in secure MPC systems research: performing MPC tasks among computationally weak and unreliably connected parties [5, 15–17, 21, 33]. The current approaches for which open-source implementations exist, such as SPDZ [32], are mostly based on linear secret-sharing schemes (LSSS) [30], requiring potentially many rounds of communication. Hence, to operate, such approaches assume the protocol participants to be online and responsive for the whole duration of the MPC process. This requirement is 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, 32, 35] or secure hardware, hence introducing non-cryptographic assumptions for security. We propose Helium, the first framework to enable 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, 37]. An MHE scheme enables a set of parties to encrypt their inputs in such a way that (i) it enables arithmetic computation to be performed over the encrypted data, (ii) it enforces a joint cryptographic access control over the plaintext data. MHE schemes can be instantiated into a natural MPC protocol. In its theoretical formulation, this protocol starts with a *setup phase* in which a set of public keys necessary for encryption and evaluation of circuits are created among the parties. Then, in the *compute phase*, the computation is performed homomorphically, under collective encryption, and the final result is obtained through a decryption protocol. Due to their low communication complexity and their small number of rounds, the MHE-based MPC approaches are an ideal choice for low-requirements MPC [4, 37]. Additionally, their *public-transcript* property [37] enables the parties to delegate most of the communication and computation costs to a single honest-but-curious third party (*e.g.,* a cloud server), without relying on additional non-cryptographic assumptions such as trusted hardware or non-collusion. Thanks to these properties, MHE-based MPC solutions have been successfully proposed in the secure computation literature, to perform tasks such as federated training of neural networks [43, 44] and distributed principal component analysis [27]. However, their wider adoption is still hindered by the lack of systematization and software implementations of MHE-based MPC. Indeed, whereas more than thirty LSSS-based frameworks were built over the last two decades [41], there was to date no open-source framework for MHE-based MPC. Hence, Helium is also the first such implementation.

Challenges of MHE-based MPC. In practice, several challenges arise when instantiating the MHE-based MPC protocol with realistic operating conditions. A first challenge is to support resource-constrained parties for which the RAM memory is smaller than the size of a

*Both authors were affiliated with EPFL at the time of this work.

single round-share in (the theoretical formulation of) the protocol. A second challenge is to tolerate poorly connected parties that may have low bandwidth and might experience frequent disconnections and re-connections from and to the network (*i.e., churn*). These challenges are left unaddressed by the prior works on MHE that typically consider an ideal execution model: a one-time *monolithic* execution of the protocol over a reliable network [36, 37]. Hence, an overarching challenge is to re-formulate the protocol in a more systematic, functional, and practical way.

In this work, we address these challenges by proposing a novel systematization of the MHE-based MPC protocol and by designing an associated protocol-execution mechanism. To support resource-constrained parties, we design a streamlined execution flow of the MHE-based MPC protocol in the server-assisted setting. To support churn, we extend the T-out-of-N-threshold scheme by Mouchet *et al.* [36] with a concrete retry mechanism, which was left as an open question by the authors. By doing so, we discover two failure cases arising when considering churn in current MHE schemes based on the ring learning with errors (RLWE) problem. These failure cases lead to cryptographic attacks, yet were left undiscussed due to the ideal execution models considered by theoretical works on MHE. Our execution mechanism prevents these failure cases, by ensuring that all interactive MHE operations are securely *resettable* [29].
In summary, we make the following contributions:

- **Practical Challenges Identification (§3.2).** We identify the challenges that MHE-based MPC poses in practical settings. These challenges include preserving the efficiency, security, and liveness of the protocol under resource-constrained participants and in the presence of churn.
- **Generic Solution Design (§4).** We design a *non-monolithic* execution of the MHE-based MPC protocol, that addresses the efficiency, security, and liveness challenges. We define an abstraction that captures all interactive operations in the MHE scheme and we define a generic execution method for those operations.
- **The Helium Framework (§5).** We instantiate our generic execution method in Helium, an end-to-end framework for 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 (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 the MHE-based generic-MPC protocol, and is, de facto, the first framework for passive-adversary MPC under churn that relies solely on cryptographic assumptions.

[1]The code repository is accessible at https://github.com/ChristianMct/helium

2 SYSTEM MODEL

Let $\mathcal{P} = \{P_1, ..., P_N\}$ 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 server).

Adversarial Model. We assume a passive adversary that can statically corrupt a subset $\mathcal{A} \subset \mathcal{P}$ of $T-1$ parties 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). The adversary can observe the internal state of all corrupted parties in $\mathcal{A} \cup \{H\}$. 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 Λ_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 λ_r^{-1} (*i.e.,* λ_r 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.

2.1 Requirements

Let $\mathcal{M}$ be a *plaintext space* ring, let $f : \mathcal{M}^N \rightarrow \mathcal{M}$ be an arithmetic function, and let $x_i \in \mathcal{M}$ 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.

Privacy. We consider the traditional notion of *input privacy* in MPC, by requiring 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 assume a *security* parameter λ and require that the advantage of the adversary $\mathcal{A}$ in breaking input privacy is no more than $2^{-\lambda}$.

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 a 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 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, 46]. 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.

MHE Scheme. An MHE scheme over a plaintext space $\mathcal{M}$ is a set of algorithms and protocols $\mathsf{MHE} = \{\Pi_{\mathsf{SecKeyGen}}, \Pi_{\mathsf{EncKeyGen}}, \Pi_{\mathsf{EvalKeyGen}}, \mathsf{Encrypt}, \mathsf{Eval}, \Pi_{\mathsf{Decrypt}}\}$. The secret-key generation protocol $\Pi_{\mathsf{SecKeyGen}}$ creates a secret key per party, which can be seen as a secret share of a collective secret key. The public-key generation protocols ($\Pi_{\mathsf{EncKeyGen}}$, $\Pi_{\mathsf{EvalKeyGen}}$) output public keys necessary for the encryption and evaluation algorithms, respectively. Both protocols require the parties to have access to a common random string (CRS) denoted by crs. The Encrypt algorithm uses the key generated by the $\Pi_{\mathsf{EncKeyGen}}$ protocol to encrypt the plaintext data. The Eval algorithm uses the keys generated by the $\Pi_{\mathsf{EvalKeyGen}}$ protocols to evaluate functions over the ciphertexts. Finally, the Π_{Decrypt} protocol decrypts the final result.

MPC Protocol. The MHE−MPC protocol shown in Protocol 1 has two phases, Setup and Compute. Each of these phases consists of running several MHE protocols as *sub-protocols*. 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 cpk (with $\Pi_{\mathsf{EncKeyGen}}$) for encrypting the inputs, and (ii) all the evaluation keys required to evaluate the target function (with multiple calls to $\Pi_{\mathsf{EvalKeyGen}}$, one per operation type used in the homomorphic circuit). During the Compute phase, the parties encrypt their inputs under the collective public key cpk, evaluate the target function under homomorphic encryption (with Eval), and collectively decrypt the result with Π_{Decrypt}.

Protocol 1. MHE−MPC ▷ *MHE-based MPC (helper-assisted, public output)*

Private input: x_i for each party $P_i \in \mathcal{P}$
Public input: f the circuit, crs a common random string
Output for R: $y = f(x_1, x_2, \ldots, x_N)$

Setup:

1. (SecKeyGen) all parties in $\mathcal{P}$ execute the secret-key generation protocol
$$\mathsf{sk}_i \leftarrow \mathsf{MHE}.\Pi_{\mathsf{SecKeyGen}}(),$$
2. (PubKeyGen) any subset of parties $\mathcal{P}' \subseteq \mathcal{P}$, with $|\mathcal{P}'| \geq T$ executes the required public-key generation protocols:
$$\mathsf{cpk} \leftarrow \mathsf{MHE}.\Pi^{\mathsf{crs}}_{\mathsf{EncKeyGen}}(\mathsf{sk}_1, \ldots, \mathsf{sk}_{|\mathcal{P}'|}),$$
$$\mathsf{evk}_{\mathsf{op}} \leftarrow \mathsf{MHE}.\Pi^{\mathsf{crs}}_{\mathsf{EvalKeyGen}}(\mathsf{op}, \mathsf{sk}_1, \ldots, \mathsf{sk}_{|\mathcal{P}'|}),\ \forall \mathsf{op} \in f.$$

Compute:

1. (Input) each party in $\mathcal{P}$ encrypts its input x_i as
$$c_i \leftarrow \mathsf{MHE.Encrypt}(x_i, \mathsf{cpk}),$$
and sends c_i to H.
2. (Eval) the helper H computes the encrypted output as
$$c' \leftarrow \mathsf{MHE.Eval}(f, \{\mathsf{evk}_{\mathsf{op}}\}_{\mathsf{op} \in f}, c_1, c_2, \ldots, c_N),$$
and sends c' to the parties in $\mathcal{P}$.
3. (Output) any subset of parties $\mathcal{P}' \subseteq \mathcal{P}$ with $|\mathcal{P}'| \geq T$ re-encrypts the output c' under the receiver's key as
$$c'_R \leftarrow \mathsf{MHE}.\Pi_{\mathsf{Decrypt}}(\mathsf{sk}_1, \ldots, \mathsf{sk}_{|\mathcal{P}'|}, \mathsf{pk}_R, c').$$

In this work, we consider the *helper-assisted setting* for the MHE−MPC protocol [37], in which a helper node H assists the parties in the protocol execution. 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 the total number of parties (N) in the computation.

Monolithic Execution. Prior works [4, 36, 37] assume that MHE−MPC-like protocols are executed in four broadcast rounds: SecKeyGen, PubKeyGen, Input, and Output (one for each interactive step of Protocol 1), and that these rounds *are 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 monolithic execution assumption implies that, by design, parties are synchronized. This makes it trivial to realize assumptions such as the access to the CRS, *e.g.,* for the public-key generation sub-protocols ($\Pi_{\mathsf{EncKeyGen}}$, $\Pi_{\mathsf{EvalKeyGen}}$). This common string ensures that randomness (i) is the same for each party, (ii) is fresh for each sub-protocol execution, and (iii) is uniformly distributed in the ciphertext-space ring. However, although such a monolithic execution is convenient to theoretically analyze the security and correctness of the MHE−MPC protocol, it leads to challenges when implementing it in practice.

3.2 MHE-based MPC: Practical Challenges

We now discuss the challenges arising when instantiating the MHE−MPC protocol in practice, and we outline our solutions to those challenges.

Challenge 1. Non-monolithic MHE-based MPC Execution: To implement the MHE-based MPC protocol in a usable and maintainable framework, a systematization effort is necessary. This is because the theoretical formulation does not capture several desirable features of MHE-based MPC. Notably, it does not account for the fact that both its phases can be run in parallel. In practice, the Compute phase should begin as soon as the collective key cpk is generated, and the evaluation should proceed as soon as the relevant evaluation keys are generated; this reduces the latency to output. Moreover, neither phase is required to terminate. Rather, the parties can evaluate arbitrarily many circuits with the generated keys and they can generate new evaluation keys to support new circuits. Finally, the theoretical formulation does not account for the common structure among the MHE sub-protocols and hence misses the opportunity to define generic execution strategies.

We propose a reformulation of the MHE−MPC protocol based on two abstractions: an external (user-facing) one and an internal (implementation-facing) one. Our user-facing abstraction is to re-frame the protocol in terms of a *session*: a logical computation context for which the data access control is cryptographically enforced. Our implementation-facing abstraction is a generalization of the MHE sub-protocols, which enables us to define an execution mechanism for such protocols, in a generic way.

Challenge 2. Supporting Resource-Constrained Nodes: As per our system model, the parties in $\mathcal{P}$ have constrained hardware resources, such as small RAM and low-power CPU. Under such restrictions, a monolithic execution of the MHE−MPC is not only undesirable but also impossible. This is because it might require the computation of more sub-protocol shares than parties can fit in their memory. Particularly at risk is the PubKeyGen step (step 2. in the Setup phase of Protocol 1), for which the round share per party is the concatenation of potentially many $\Pi_{\mathsf{EvalKeyGen}}$ shares (which we refer to as a *monolithic share*). For example, the monolithic share of a single party for executing the Setup phase of the encrypted neural network training circuit proposed in [44] (on the MNIST dataset) is as large as ∼ 3GB. This is because their circuit relies on many types of homomorphic automorphism operations, each one requiring a different evaluation key.

To address this challenge, we observe that the parties in $\mathcal{P}$ do not need to store their shares in the sub-protocols and they can free the allocated memory as soon as the share is sent. Hence, instead of computing a single monolithic share, we propose to run the sub-protocols independently and asynchronously. This enables the parties to limit the number of concurrently running sub-protocols and to execute them in a streamlined manner, hence to execute large setups such as that of [44] with only ∼64MB of RAM (as opposed to the ∼3*GB* necessary for the monolithic execution). However, we must ensure that independent execution of sub-protocols does not break the correctness and security of the protocol, which can be non-trivial under churn (see Challenge 4).

Challenge 3. Liveness under Churn: Our liveness requirement necessitates a T-out-of-N-threshold MHE scheme in order for T parties to make progress without waiting for offline parties. Asharov *et al.* proposed a direct approach to T-out-of-N-threshold MHE, which is to share the initial random coins of each party among the other parties with the Shamir secret-sharing scheme and to reconstruct the offline parties' shares when needed. 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 [31]) 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 providing 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, 37], 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, 37] 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 sate-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 (R)LWE-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* [29] 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 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 MHE–MPC protocol.

4 MHE–MPC SUB-PROTOCOLS EXECUTION

In this section, we present our mechanism for executing the sub-protocols of the MHE–MPC. 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 (*i.e.,* all sub-protocols beside the $\mathsf{MHE}.\Pi_{\mathsf{SecKeyGen}}$). This abstraction is framework-facing: it enables us to define our MHE–MPC protocol execution mechanism in terms of a generic sub-protocol executor.

Preliminaries. The ciphertext space of the MHE scheme is a polynomial ring $\mathcal{R}$ in which the RLWE problem is hard [34]. Informally, for a a publicly known element, and for s and e two secret values sampled from low-norm distributions over $\mathcal{R}$, the distribution of $(as+e, a)$ is computationally indistinguishable from the uniform distribution over $\mathcal{R}^2$.

At initialization, $\mathsf{MHE}.\Pi_{\mathsf{SecKeyGen}}$ 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, $\mathsf{MHE}.\Pi_{\mathsf{SecKeyGen}}$ outputs a point $(\alpha_i, S(\alpha_i))$ of some secret degree-$T-1$ polynomial $S \in \mathcal{R}[X]$ 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

$$s = S(0) = \sum_{P_i \in \mathcal{P}'} \prod_{\substack{P_j \in \mathcal{P}' \\ P_j \neq P_i}} \frac{\alpha_j}{\alpha_j - \alpha_i} s_i = \sum_{P_i \in \mathcal{P}'} \lambda_i^{(\mathcal{P}')} s_i, \qquad (1)$$

where $\lambda_i^{(\mathcal{P}')}$ denotes the Lagrange interpolation coefficient for the share of party P_i in the reconstruction among set $\mathcal{P}'$. Note that, in practice, the secret key s is never reconstructed.

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 form $as + e$, where s is the collective secret key, a is a public polynomial and e is some small error term [37]. For example, the $\mathsf{MHE}.\Pi_{\mathsf{EncKeyGen}}$ protocol generates a collective public encryption key of the form $(p_0, p_1) = (sp_1 + e_{\mathsf{pk}}, p_1)$ by setting a to be a uniform value sampled from the common random string. Similarly, the $\mathsf{MHE}.\Pi_{\mathsf{Decrypt}}$ protocol performs the decryption of a ciphertext (c_0, c_1) in two steps: It first computes a term $h = sc_1 + e_{\mathsf{dec}}$ (*i.e.,* $a = c_1$); then it computes a noisy message as $m_{noisy} \approx c_0 + h$ which can be decoded into m (provided that the noise is not too large).

To compute this noisy product, any group $\mathcal{P}'$ of size at least T exploits the linearity of the Shamir secret-sharing scheme: each party in $\mathcal{P}'$ computes and discloses its respective linear term (its *share* in the protocol) plus some additional fresh error. Due to this added fresh error term, the shares are safe to disclose (*i.e.,* 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 can be expressed as a tuple PAT = (GenShare, AggShare, Finalize) of algorithms with the following syntax and semantics:

- **Share Generation**: $v_i \leftarrow \mathsf{PAT.GenShare}(s_i, a, \mathcal{P}'; \chi)$
  From the secret-key share s_i, a publicly known polynomial a and a set of participating parties $\mathcal{P}'$, GenShare outputs a share $v_i = \lambda_i^{(\mathcal{P}')} s_i a + e_i$, with $e_i \leftarrow \chi$.
- **Share Aggregation**: $v_{\mathrm{agg}} \leftarrow \mathsf{PAT.AggShare}(\{v_i\}_{P_i \in \mathcal{P}'})$
  From the shares $\{v_i\}_{P_i \in \mathcal{P}'}$ of the participating parties $\mathcal{P}'$, AggShare outputs a single aggregated share
  $$v_{\mathrm{agg}} = \sum_{P_i \in \mathcal{P}'} v_i = sa + \sum_{P_i \in \mathcal{P}'} e_i$$
- **Finalization**: $\mathsf{out} \leftarrow \mathsf{PAT.Finalize}(v_{\mathrm{agg}}, \mathsf{in})$
  From the aggregation of all shares of the parties in $\mathcal{P}'$ and some public auxiliary input polynomial in, Finalize outputs the result out of the protocol.
  In the key-generation protocols ($\Pi_{\mathsf{EncKeyGen}}$ and $\Pi_{\mathsf{EvalKeyGen}}$), the auxiliary input in is the public polynomial a and Finalize outputs the resulting key as $\mathsf{out} = (v_{\mathrm{agg}}, a)$. In the decryption protocol (Π_{Decrypt}), the auxiliary input in is the element c_0 of the ciphertext and Finalize outputs the decrypted ciphertext as $\mathsf{out} = c_0 + v_{\mathrm{agg}}$.

Overall, the execution of the MHE–MPC protocol (see Protocol 1) reduces to the execution of many PAT protocol instances, which need to be orchestrated efficiently and securely. 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. (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 independently 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. This mechanism considers a session-like execution (addressing Challenge 1) and can be executed by resource-constrained session

nodes (addressing Challenge 2). However, it only partially addresses the liveness and security challenges (Challenges 3 and 4). This is because we assume, for the sake of the exposition, a restricted churn model in which parties involved in the execution of a PAT protocol do not fail during its execution. We then lift this assumption in §4.3.

4.2.1 Nodes. We refer to all actors in our system as *nodes*. 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 in Helium. First, *session nodes* which have inputs to the computation (*i.e.,* the parties in $\mathcal{P}$). Session nodes hold a share of the collective secret key (ensuring their inputs' access-control), hence are the nodes that provide inputs when executing the MHE sub-protocols. Second, a *helper node* that assists the parties in the computation. The helper node does not have private inputs to the computation and, hence, does not hold a share of the secret key.

4.2.2 Sessions. Analogously to secure communication protocols such as TLS[2], we view MHE sessions as a long-lived *logical* secure multiparty computation context. We define the *public session parameters* as

$$\mathsf{PubSessParams} := \{\mathsf{SessionID}, \mathsf{Nodes}, \mathsf{HEParams}, \mathsf{PublicSeed}\}.$$

where SessionID is a unique system-wide identifier for the session, Nodes is the identities of the N nodes in $\mathcal{P}$, HEParams 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

$$\mathsf{SessParams}_i := \{\mathsf{PubSessParams}, \mathsf{PrivateSeed}_i, \mathsf{sk}_i\}.$$

where PubSessParams are the public session parameters, $\mathsf{PrivateSeed}_i$ is a private bit-string seed for the randomness source of each party, and 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 (*e.g.,* after a node crash).

From this point onward, our discussion focuses on a single session, and hence on a single instance of a long-lived MHE−MPC protocol. For conciseness, in the following, we refer to the MHE−MPC protocol as *the session* and to its sub-protocols simply as *protocols*.

4.2.3 Roles. Recall that there are two kinds of nodes in our model: session nodes (who have a share of the secret key) and helper nodes (who do not). For each PAT protocol, the nodes can assume different *roles*:

- The *Protocol Participants* are the T session nodes that provide a share in the PAT protocol. We denote the set of participants as $\mathcal{P}' \subseteq \mathcal{P}$.
- The *Protocol Aggregator* is a designated node that collects the T shares from all protocol participants (generated with the PAT.GenShare method) and aggregates them (with the PAT.AggShare method). Due to the publicly aggregatable property of PAT protocols, this role can in theory be assumed by any node in the system. In our helper-assisted setting, the helper assumes this role, which results in communication overhead for the protocol participants that is independent of N and T.
- The *Coordinator* is a designated node that initiates and keeps track of the execution of PAT protocols, hence ensuring that the session (*i.e.,* the MHE−MPC protocol) progresses. As for the aggregator, the coordinator role does not require any secret session parameter, hence can be assumed by any node in the system. However, it is crucial that the coordinator reliably keeps the state of the session (*i.e.,* the list of executed PAT protocols). In our helper-assisted setting, the helper also assumes this role.

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
$$\mathsf{PSig} := \{\mathsf{PType}, \mathsf{PArgs}\}$$
where PType designates the type of protocol (*i.e.,* $\mathsf{PType} \in \{\Pi_{\mathsf{EncKeyGen}}, \Pi_{\mathsf{EvalKeyGen}}, \Pi_{\mathsf{Decrypt}}\}$), 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 $\{\Pi_{\mathsf{EvalKeyGen}}, \mathsf{op}\}$, while a decryption of ciphertext is represented by $\{\Pi_{\mathsf{Decrypt}}, \mathsf{ctid}\}$ 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
$$\mathsf{PDesc} := \{\mathsf{PSig}, \mathsf{PParticipants}, \mathsf{PAggregator}\}$$
where is PSig the protocol signature, PParticipants is the set of T session nodes that provide a share in the protocol, and PAggregator 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 sig, the nodes proceed as follows:

(1) The coordinator picks a set of protocol participants $\mathcal{P}'$ and a protocol aggregator P_A. The participants in $\mathcal{P}'$ can be any online session node (*i.e.,* $\mathcal{P}' \subseteq \mathcal{P}$) and the aggregator can be any online and reachable node (*i.e.,* $P_A \in \mathcal{P} \cup \{\mathcal{H}\}$). Then, the coordinator sends $\mathsf{PDesc} = \{\mathsf{PSig} = \mathsf{sig}, \mathsf{PParticipants} = \mathcal{P}', \mathsf{PAggregator} = P_A\}$ to all nodes in $\mathcal{P}' \cup \{P_A\}$.
(2) Upon receiving PDesc, each protocol participant $P_i \in \mathcal{P}'$ computes its respective share as $v_i = \Pi_{\mathsf{PSig}}.\mathsf{GenShare}(s_i, a, \mathcal{P}'; \chi)$ and sends it to the aggregator P_A. The aggregator aggregates the received shares on-the-fly, with $\Pi_{\mathsf{PSig}}.\mathsf{AggShare}$.
(3) Upon receiving all the T shares for the participants in $\mathcal{P}'$, 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_{agg}, and by computing $\Pi_{\mathsf{PSig}}.\mathsf{Finalize}(v_{agg}, a)$.

[2]https://www.rfc-editor.org/rfc/rfc8446

Public Polynomials. Note that Step 2 above requires each participant to obtain the public polynomial a. We now discuss how this is implemented in our execution flow. Indeed, a tempting solution would be to pass the protocol's public polynomial as a protocol argument in the PSig.PArgs field. This is an unsatisfying solution for two reasons: The first reason is performance-related: 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. We will exploit this in our helper-assisted setting, in §5.1. The second reason is security-related: to enable our churn-tolerant mechanism (described in §4.3) to work securely, public polynomials must be decoupled from the protocol signature. Instead, we let protocol participants derive or retrieve the public polynomial a:

In the decryption protocol, 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.

In the public-key generation protocols, 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 PublicSeed (see §4.2.2). Although this approach is satisfactory for a monolithic execution [4, 37], 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 *branch* the CRS for each protocol: to sample a for a protocol with signature PSig, each participant first computes a *protocol public seed* as

$$\mathsf{ProtPubSeed} \;:=\; \mathsf{PublicSeed}||\mathsf{PSig},$$

where PublicSeed is read from the session parameters (see §4.2.2), then samples a from a keyed PRF seeded with ProtPubSeed.

Liveness and Security under Restricted Churn. Since the coordinator chooses the protocol participants based on its view of the network, our execution mechanism ensures liveness as long as at least T parties are online. Indeed, this is because we consider a *restricted* churn model in which, once the set of protocol participants $\mathcal{P}'$ has been decided for a given PAT protocol, no participant fails before providing its share in that protocol. In such an ideal model, we observe that our mechanism simply emulates a monolithic execution by running each required protocol once (hence addresses Challenges 3 and 4 in this model). However, it is indeed unrealistic to assume anything about the precise time at which a node crashes. As a consequence, fully addressing Challenges 3 and 4 in our unrestricted churn model of §2 requires lifting those assumptions. This is done in the next section.

4.3 Secure Churn Handling

To address the liveness and security challenges under churn (Challenges 3 and 4), the protocol must operate correctly regardless of when participants fail during the PAT protocol execution [36].

Recall that, in the MHE scheme of Mouchet *et al.*, the set $\mathcal{P}'$ of protocol participants must be known by the protocol participants to generate their shares [36]. 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 is efficient. However, they leave the exact formulation of this failure-retry mechanism undefined. We now instantiate this mechanism in our non-monolithic execution setting.

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 with protocol descriptor PDesc can be naturally expressed in our execution mechanism of §4.2, by having the coordinator issue a new protocol descriptor PDesc′ with PDesc.PSig = PDesc′.PSig (*i.e.,* an *equivalent* protocol) and with a different participant set. From the security perspective, however, there is an important consideration: Nodes might participate in the same PAT protocol multiple times, as a result of a re-connection event or as part of a retry. As a consequence, our execution mechanism no longer emulates a monolithic execution; this requires us to study the security implications of PAT protocol retries.

Secure Protocol Retries. We must ensure that the *additional* shares sent by the parties as part of re-connection and retries do not break the simulatability of the MHE−MPC protocol. More specifically, we consider the two failure cases below:

- *Failure Case A*: A party generates and sends two shares for a PAT protocol with the public polynomial a. This is possible when a participant experiences a failure event during the transmission of the first share, reconnects before the timeout, and recomputes and transmits a share for the same protocol. In this case, the disclosure of both shares $\lambda_i^{(\mathcal{P}')} s_i a + e_i$ and $\lambda_i^{(\mathcal{P}')} s_i a + e_i'$ directly leads to an attack where the adversary can average the two shares to gain information on $s_i a$.
- *Failure Case B*: An (insecure) retry mechanism which simply re-runs the protocol with the same public polynomial a but over a different set of parties $\mathcal{P}''$. In this case, a participant would disclose two related shares $\lambda_i^{(\mathcal{P}')} s_i a + e_i$ and $\lambda_i^{(\mathcal{P}'')} s_i a + e_i'$, for which the RLWE secrets are linearly related. As a consequence, an adversary can distinguish between the two related shares and two random ring elements by multiplying both shares by the inverse of their Lagrange coefficient (which can be publicly computed); this breaks the simulatability of the protocols.

In both cases, security issues arise from disclosing two related shares. As a consequence, the existing security analysis of the MHE−MPC protocol [4, 37] would not apply to our execution mechanism, because the behavior of our nodes could not be simulated by an RLWE challenger oracle in these two failure cases. Failure case A corresponds to the ability of an adversary to rewind the execution of the oracle and force it to re-output a correlated challenge.

Failure case B corresponds to the ability of the adversary to directly re-query a correlated challenge.

We first observe that we can exclude both failure cases by requiring that nodes never output two different shares for the same PAT protocol. Hence, a trivial solution would be to require parties to write their shares in their persistent storage until the protocol is completed. Such a solution, however, would not only contradict the low-persistent memory requirement (see §2) but would also not prevent failure case B. Instead, we propose to further specify how PAT protocols sample their randomness, in order to make their execution completely deterministic (given all the session parameters).

Protocol Private Randomness Initialization. To prevent failure case A, we make the share of each node deterministic (given the node's session parameters) by ensuring that the fresh error terms sampled during the execution of a PAT protocol (see §4.1) is itself deterministic. We achieve this by seeding the PRNG used to sample these terms, as we discuss below.

We start by re-defining the *protocol public seed* (formerly introduced in §4.2.5), by integrating the participant set in the seed:

$$\mathsf{ProtPubSeed} \;:=\; \mathsf{PublicSeed}||\mathsf{PSig}||h(\mathsf{PParticipants}),$$

where $h : \mathsf{Powerset}(\mathcal{P}) \rightarrow \{1,0\}^*$ 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 §4.2.2). This seed is unique for each possible PAT protocol instance and can be publicly computed. Then, by concatenating the protocol public seed with the session private seed of the session node P_i, we obtain the node's *protocol secret seed*:

$$\mathsf{ProtSecSeed}_i \;:=\; \mathsf{PrivateSeed}_i||\mathsf{ProtPubSeed},$$

that the session nodes use as a private randomness source to sample error terms in a given protocol instance. 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.

Protocol Public Randomness Initialization. To prevent failure case B, common random polynomials must be fresh for each protocol execution. This requires accounting for two cases: when these public polynomials are sampled from the CRS (*i.e.,* in the key-generation protocols of the Setup phase) and when they are elements of ciphertexts (*i.e.,* in the decryption step of the Compute phase).

In the CRS sampling case, we let the parties sample the public polynomial by reading from a keyed PRF initialized with the newly redefined protocol public seed ProtPubSeed. Observe that the corresponding protocol public seed produces different public polynomials for each retry, since retries are new protocols with the same signature yet a different set of participants. Also, note that retries with the same participant set as in the original (failed) protocol are not excluded by our mechanism. From the security standpoint, this case is equivalent to that of having all participants of the original protocol re-send their shares, which is covered by preventing failure case A.

In the decryption protocol case, protocols operate on an input ciphertext (c_0, c_1) by producing one or multiple shares of the form $sc_1 + e$ for some secret polynomials s and e (*i.e.,* samples from the RLWE distribution). Since we cannot simply sample a different c_1 element for each protocol retry (because c_1 is taken from the ciphertext), we propose a mechanism to *re-randomize* the ciphertext.

The homomorphic property of MHE schemes enables re-randomization of ciphertexts through the homomorphic addition of the ciphertext to re-randomize with a fresh encryption of zero. Such a zero-encrypting ciphertext can be generated from the session's public key cpk, by running the MHE.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. Yet, 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.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].

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 nodes can adapt the execution to the current condition of the network, 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 $\mathcal{R}$.

Liveness and Security under Churn (Challenges 3 and 4). Our execution mechanism enables nodes to re-execute a failed or stale PAT protocol. As a result, achieving liveness under churn for the full protocol simply requires us to define a policy for when PAT protocols should be retried. Hence, the last step towards a complete implementation of the MHE–MPC protocol is to instantiate this execution mechanism and to complement it with homomorphic circuit evaluation capabilities.

5 HELIUM

We introduce Helium, an end-to-end implementation of the full 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.

Aggregator and Coordinator. In Helium, we let the helper node H assume the roles of the aggregator and the coordinator in the PAT protocol execution (see §4.2). Recall that we assume this node to be highly reliable in terms of availability, but a passive adversary in

terms of threat model. 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 rather than trust. 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.

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 PLog of *synchronization messages* of the form: $\mathsf{SynMsg} := \{\mathsf{PDesc}, \mathsf{PStatus}\}$, where PDesc is a protocol descriptor and $\mathsf{PStatus} \in \{\mathsf{Started}, \mathsf{Completed}, \mathsf{Failed}\}$ is a status indicator for the protocol defined by PDesc. 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 PLog.

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 PLog, 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 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* 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 *equivalent*, yet *uncorrelated*, protocols, and hence 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 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.

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 MHE–MPC. 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 $\mathsf{PSig} = \{\mathsf{PType} = \mathsf{EncKeyGen}, \mathsf{PArgs} = ()\}$ to the protocol layers and return the returned cpk. When queried for the evaluation key of operation op, the service queries for $\mathsf{PSig} = \{\mathsf{PType} = \mathsf{EvalKeyGen}, \mathsf{PArgs} = (\mathsf{op})\}$ and returns the returned $\mathsf{evk}_{\mathsf{op}}$.

Persistence. In practical implementations, the service can provide more functionality. For instance, it can cache into session nodes persistence storage the result of relevant PAT to reduce the network overhead at initialization. For example, nodes might cache the result of the $\Pi_{\mathsf{EncKeyGen}}$ protocol as they will use this collective public key to encrypt their inputs throughout execution. Conversely, results from the $\Pi_{\mathsf{EvalKeyGen}}$ 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. Then, they can reconstruct the result from the public seed, the protocol descriptor, and the Finalize method for that protocol. We note that persistence is not necessary for nodes to safely and efficiently restart from the session parameters (§2).

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 MHE–MPC Compute phase (see §3).

Initialize

Session Node P_i	Helper Node
• load the session parameters and initialize an empty PLog • query the helper for PLog messages until present • for each non-completed, non-failed PDesc in PLog: **if** $P_i \in$ PDesc.PParticipants: send PDesc to Execution	• load the public session parameters • initialize SigQueue, PLog • initialize a key-value store ResTable

Coordinate

Session Node P_i	Helper Node
Upon new SynMsg = {PDesc, PStatus}: **if** PStatus = Started and $P_i \in$ PDesc.PParticipants : send PDesc to the Execution routine • append SynMsg to PLog	**Upon** new signature sig from queue: • pick $\mathcal{P}'$ from the set of online nodes • append SynMsg={PDesc={sig, $\mathcal{P}'$}, PStatus=Started} to PLog **Upon** idle: • lookup the oldest non-completed PDesc in PLog **if** PDesc started for longer than PDeadline: • append SynMsg={PDesc, PStatus=Failed} to PLog • put PDesc.PSig back in SigQueue

ExecutePAT

Session Node P_i	Helper Node
Upon new PDesc = {sig, $\mathcal{P}'$}: **if** sig $\in$ {EncKeyGen, EvalKeyGen}: compute a = CRS(PDesc) **else if** sig = Decrypt: retrieve a from the helper send Π_{sig}.GenShare(s_i, a, $\mathcal{P}'$) to the helper	**Upon** new PDesc = {sig, $\mathcal{P}'$}: • collect the T shares from $\mathcal{P}'$ and set agg = Π_{sig}.AggShare • set ResTable[PDesc] = agg. • append SynMsg={PDesc, PStatus=Completed} to PLog **Upon** query PDesc from a session node: **return** agg = ResTable[PDesc] to the node.

Interface

Session Node P_i	Helper Node
Upon query sig from a service: • retrieve the latest completed PDesc for sig in PLog or wait • query agg from the helper node **if** sig $\in$ {EncKeyGen, EvalKeyGen}: compute a = CRS(PDesc) **else if** sig = Decrypt: retrieve a from the helper **return** Π_{sig}.Finalize(agg, a) to the service	**Upon** query sig from a service: **if** no completed PDesc for sig is in PLog: • put sig in SigQueue and wait • retrieve agg from the ResTable[PDesc]. **if** sig $\in$ {EncKeyGen, EvalKeyGen}: compute a = CRS(PDesc) **else if** sig = Decrypt: retrieve a from the ciphertext table **return** Π_{sig}.Finalize(agg, a) to the service

Figure 1: Psdeudo-code description of the helper-assisted execution in Helium.

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.,* a representation of the function f the application wants to compute) 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 P_i = 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 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 label 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 key-value store that is hosted at the helper, and a traditional *put* and *get* interface (*e.g.,* HTTP, FTP, or more advanced RPC protocols) for parties to upload their inputs and download their results.

The ability to exploit existing paradigms for our data layer is a considerable advantage of the MHE-based MPC approach. Indeed, whereas traditional MPC solutions based on secret-sharing of the data fundamentally require interaction among N parties for each input wire and among T parties for each output wire, both input

```
func(sess helium.Session) {
   // read the nodes' inputs
   op1 := sess.Input("//node-a/in")
   op2 := sess.Input("//node-b/in")

   // multiply the inputs
   res := sess.MulNew(op1, op2)
   sess.Relinearize(res, res)

   // decrypt and output the result
   resDec := sess.Decrypt(res)
   sess.Output("/out", resDec)
}
```

Listing 1: The Helium program for two-party component-wise vector multiplication.

and output MHE ciphertexts can be retrieved in a single interaction. To further exploit this advantage Helium uses a URI scheme as ciphertext identifier, which we describe in Appendix B.

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 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 implemented Helium in Go.[3] We rely on the Lattigo library [26] for the cryptographic operations and on the gRPC framework[4] 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 MHE–MPC protocol execution mechanism 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) if the user application already has its own.

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.

Experimental Setup. We use Docker containers[5] to run all nodes over two machines with Intel Xeon E5-2680 v3 processors (2.5 GHz, 2×12 cores), 256 GB 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_i (see §4.2.2), but no PAT protocol has been run yet (*i.e.,* no key has been generated yet). The helper node is initialized with the session public parameters. To simulate low-end network conditions, we limit the egress and ingress traffic of each container individually to 100Mbits 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 512×512 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 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 hold their shares of M.

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.

6.1 Experiment I: Lightweightness & Scalability

In this experiment, we evaluate Helium's execution when there is a large number of resource-constraints participants. We do not consider any churn and set the threshold to $T = N$, and thus 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 [32] 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 lightweightness 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 $N = 30$

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

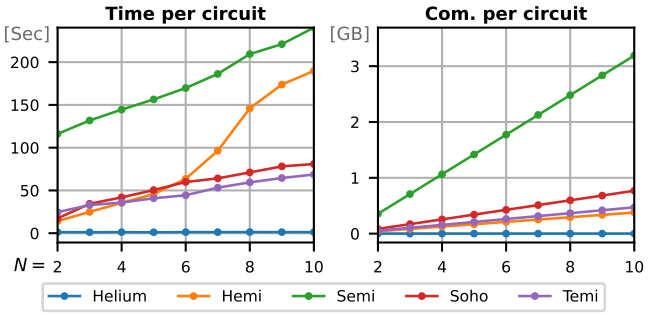

Figure 2: Cost per matrix-vector multiplication. Time (left) and per-party communication (right) for an increasing number of session nodes and several MPC implementations. Hemi, Semi, Soho, and Temi, refer to the semi-honest protocols implemented in MP-SPDZ [32]. Average over 10 runs.

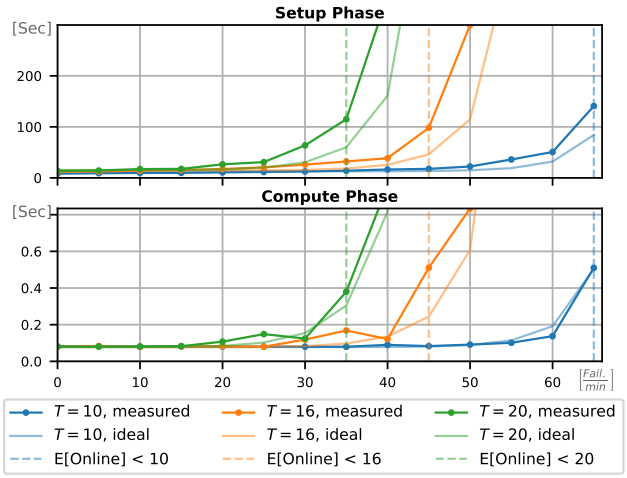

Figure 3: Setup phase latency (up) and average time per matrix-vector multiplication (down) for increasing failure rates. For $N = 30$ parties. Average over 10 runs.

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.

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 λ_f (respectively, λ_r). A property of this model is that the expected number of online nodes converges over time to an equilibrium $E[N_{online}] = N\frac{\lambda_r}{\lambda_f+\lambda_r}$. To map our failure model (see §2), we can compute the system-wide failure rate (at equilibrium) as $\Lambda_f = \lambda_f \cdot E[N_{online}]$. We start all experiments with a random set of $E[N_{online}]$ online nodes. To simulate failures and re-connections, we kill and restart the containers on the session nodes.

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 $N_{online} = T$ (*i.e.,* the churn-free execution time), divided by $Pr[N_{online} \geq T]$ (*i.e.,* the expected fraction of time for which the system has at least T online nodes). For our churn model, we have $Pr[N_{online} \geq T] = \sum_{t=0}^{T-1} \binom{N}{t} (\frac{\lambda_r}{\lambda_f+\lambda_r})^t (1 - \frac{\lambda_r}{\lambda_f+\lambda_r})^{N-t}$.

We observe that Helium successfully satisfies the churn-tolerance requirement of §2: For failure rates below the $E[N_{online}] \geq T$ threshold (plotted as dashed lines), we observe that the latency is close to the ideal one. Furthermore, whenever $\Pr[N_{online} \geq T]$ 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 Discussion

In both experiments, we demonstrate a new level of practicality for MPC. In Experiment I, we show that the Helium scales to large numbers of parties, even when these parties are resource-constrained or have limited connectivity. This is mainly due to the properties of the MHE-based MPC protocol and Helium's helper-assisted setting. In Experiment II, we show that our system can handle churn, which is an inevitable concern for systems with a large number of nodes. We demonstrate that our failure-handling mechanism enables 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.

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, 28] or that a dynamic set of participants can execute the computation task (known as *fluid* MPC) [15, 40]. 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 AsynchroMix [33], 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 [32], 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 requires high bandwidth and memory requirements for the participants (even in the absence of churn). To circumvent these limitations, most of the practical uses of MPC techniques to date introduce non-cryptographic assumptions such as non-collusion between a smaller number of servers [18, 25, 35]. While Helium also uses delegation to achieve

sub-linear costs for the participants, it solely relies on cryptographic assumptions to do so in the plain 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, 47]. As the overheads of these methods are still out of reach for resource-constrained participants, integrating them 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.

REFERENCES

[1] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. 2018. *Homomorphic Encryption Security Standard.* Technical Report. HomomorphicEncryption.org, Toronto, Canada.

[2] Diego F. Aranha, Anamaria Costache, Antonio Guimaraes, and Eduardo Soria Vazquez. 2024. A Practical Framework for Verifiable Computation over Encrypted Data. https://fhe.org/conferences/conference-2024/ 3rd Annual FHE.org Conference on Fully Homomorphic Encryption ; Conference date: 24-03-2024 Through 24-03-2024.

[3] David W Archer, José Manuel Calderón Trilla, Jason Dagit, Alex Malozemoff, Yuriy Polyakov, Kurt Rohloff, and Gerard Ryan. 2019. Ramparts: A programmer-friendly system for building homomorphic encryption applications. In *Proceedings of the 7th acm workshop on encrypted computing & applied homomorphic cryptography*. 57–68.

[4] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. 2012. Multiparty computation with low communication, computation and interaction via threshold FHE. In *Advances in Cryptology–EUROCRYPT 2012: 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings 31*. Springer, 483–501.

[5] Assi Barak, Martin Hirt, Lior Koskas, and Yehuda Lindell. 2018. An end-to-end system for large scale P2P MPC-as-a-service and low-bandwidth MPC for weak participants. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 695–712.

[6] Zuzana Beerliová-Trubíniová and Martin Hirt. 2008. Perfectly-secure MPC with linear communication complexity. In *Theory of Cryptography Conference*. Springer, 213–230.

[7] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. 2015. How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In *International Conference on Financial Cryptography and Data Security*. Springer, 227–234.

[8] Zvika Brakerski and Vinod Vaikuntanathan. 2011. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Advances in Cryptology–CRYPTO 2011: 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings 31*. Springer, 505–524.

[9] Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. 2022. Motion–a framework for mixed-protocol multi-party computation. *ACM Transactions on Privacy and Security* 25, 2 (2022), 1–35.

[10] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. 2015. Armadillo: a compilation chain for privacy preserving applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing*. 13–19.

[11] Sylvain Chatel, Christian Knabenhans, Apostolos Pyrgelis, and Jean-Pierre Hubaux. 2022. Verifiable Encodings for Secure Homomorphic Analytics. *arXiv preprint arXiv:2207.14071* (2022).

[12] Sylvain Chatel, Christian Mouchet, Ali Utkan Sahin, Apostolos Pyrgelis, Carmela Troncoso, and Jean-Pierre Hubaux. 2023. PELTA - Shielding Multiparty-FHE against Malicious Adversaries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (<conf-loc>, <city>Copenhagen</city>, <country>Denmark</country>, </conf-loc>) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 711–725. https://doi.org/10.1145/3576915.3623139

[13] Sylvain Chatel, Apostolos Pyrgelis, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2021. Privacy and Integrity Preserving Computations with CRISP. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2111–2128. https://www.usenix.org/conference/usenixsecurity21/presentation/chatel

[14] Eduardo Chielle, Oleg Mazonka, Homer Gamil, Nektarios Georgios Tsoutsos, and Michail Maniatakos. 2018. E3: A framework for compiling C++ programs with encrypted operands. *Cryptology ePrint Archive* (2018).

[15] Arka Rai Choudhuri, Aarushi Goel, Matthew Green, Abhishek Jain, and Gabriel Kaptchuk. 2021. Fluid MPC: secure multiparty computation with dynamic participants. In *Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part II 41*. Springer, 94–123.

[16] Ashish Choudhury, Martin Hirt, and Arpita Patra. 2013. Asynchronous multiparty computation with linear communication complexity. In *Distributed Computing: 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings 27*. Springer, 388–402.

[17] Ashish Choudhury and Arpita Patra. 2015. Optimally resilient asynchronous MPC with linear communication complexity. In *Proceedings of the 16th International Conference on Distributed Computing and Networking (ICDCN)*. 1–10.

[18] Henry Corrigan-Gibbs and Dan Boneh. 2017. Prio: Private, robust, and scalable computation of aggregate statistics. In *14th Symposium on Networked Systems Design and Implementation (NSDI 17)*. 259–282.

[19] Eric Crockett, Chris Peikert, and Chad Sharp. 2018. Alchemy: A language and compiler for homomorphic encryption made easy. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1020–1037.

[20] Ivan Damgård, Daniel Escudero, and Antigoni Polychroniadou. 2021. Phoenix: Secure computation in an unstable network with dropouts and comebacks. *Cryptology ePrint Archive* (2021).

[21] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. 2009. Asynchronous multiparty computation: Theory and implementation. In *Public Key Cryptography–PKC 2009: 12th International Conference on Practice and Theory in Public Key Cryptography, Irvine, CA, USA, March 18-20, 2009. Proceedings 12*. Springer, 160–179.

[22] Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. 2010. Perfectly secure multiparty computation and the computational overhead of cryptography. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 445–465.

[23] Ivan Damgård and Jesper Buus Nielsen. 2007. Scalable and unconditionally secure multiparty computation. In *Annual International Cryptology Conference*. Springer, 572–590.

[24] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. 2020. EVA: An encrypted vector arithmetic language and compiler for efficient homomorphic computation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 546–561.

[25] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY-A framework for efficient mixed-protocol secure two-party computation.. In *NDSS*.

[26] EPFL-LDS, Tune Insight SA. 2023. Lattigo v4. Online: https://github.com/tuneinsight/lattigo.

[27] David Froelicher, Hyunghoon Cho, Manaswitha Edupalli, Joao Sa Sousa, Jean-Philippe Bossuat, Apostolos Pyrgelis, Juan R Troncoso-Pastoriza, Bonnie Berger, and Jean-Pierre Hubaux. 2022. Scalable and Privacy-Preserving Federated Principal Component Analysis. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 888–905.

[28] Craig Gentry, Shai Halevi, Hugo Krawczyk, Bernardo Magri, Jesper Buus Nielsen, Tal Rabin, and Sophia Yakoubov. 2021. YOSO: You Only Speak Once: Secure MPC with Stateless Ephemeral Roles. In *Annual International Cryptology Conference*. Springer, 64–93.

[29] Vipul Goyal and Amit Sahai. 2009. Resettably secure computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 54–71.

[30] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. 2019. SoK: General purpose compilers for secure multi-party computation. In *2019 IEEE symposium on security and privacy (SP)*. IEEE, 1220–1237.

[31] Amir Herzberg, Stanisław Jarecki, Hugo Krawczyk, and Moti Yung. 1995. Proactive secret sharing or: How to cope with perpetual leakage. In *Advances in Cryptology—CRYPT0'95: 15th Annual International Cryptology Conference Santa Barbara, California, USA, August 27–31, 1995 Proceedings 15*. Springer, 339–352.

[32] Marcel Keller. 2020. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 1575–1590.

[33] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. 2019. 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*. 887–903.
[34] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2010. On Ideal Lattices and Learning with Errors over Rings. In *Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30-June 3, 2010, Proceedings*, Vol. 6110. Springer, 1.
[35] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A system for scalable privacy-preserving machine learning. In *2017 38th IEEE Symposium on Security and Privacy (SP)*. IEEE, 19–38.
[36] Christian Mouchet, Elliott Bertrand, and Jean-Pierre Hubaux. 2023. An Efficient Threshold Access-Structure for RLWE-Based Multiparty Homomorphic Encryption. *Journal of Cryptology* 36 (2023).
[37] Christian Mouchet, Juan Troncoso-Pastoriza, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. 2021. Multiparty Homomorphic Encryption from Ring-Learning-with-Errors. *Proceedings on Privacy Enhancing Technologies* 4 (2021), 291–311.
[38] Jeongeun Park. 2021. Homomorphic encryption for multiple users with less communications. *IEEE Access* 9 (2021), 135915–135926.
[39] Antigoni Polychroniadou, Gilad Asharov, Benjamin Diamond, Tucker Balch, Hans Buehler, Richard Hua, Suwen Gu, Greg Gimler, and Manuela Veloso. 2023. Prime Match: A Privacy-Preserving Inventory Matching System. *Cryptology ePrint Archive* (2023).
[40] Rahul Rachuri and Peter Scholl. 2022. Le mans: Dynamic and fluid MPC for dishonest majority. In *Annual International Cryptology Conference*. Springer, 719–749.
[41] Dragoş Rotaru. 2017. awesome-mpc. https://github.com/rdragos/awesome-mpc.
[42] Sinem Sav, Jean-Philippe Bossuat, Juan R Troncoso-Pastoriza, Manfred Claassen, and Jean-Pierre Hubaux. 2022. Privacy-preserving federated neural network learning for disease-associated cell classification. *Patterns* 3, 5 (2022), 100487.
[43] Sinem Sav, Abdulrahman Diaa, Apostolos Pyrgelis, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. 2022. Privacy-Preserving Federated Recurrent Neural Networks. *arXiv preprint arXiv:2207.13947* (2022).
[44] Sinem Sav, Apostolos Pyrgelis, Juan R Troncoso-Pastoriza, David Froelicher, Jean-Philippe Bossuat, Joao Sa Sousa, and Jean-Pierre Hubaux. 2021. POSEIDON: Privacy-preserving federated neural network learning. *28th Annual Network and Distributed System Security Symposium* (2021).
[45] Amos Treiber, Dirk Müllmann, Thomas Schneider, and Indra Spiecker genannt Döhmann. 2022. Data Protection Law and Multi-Party Computation: Applications to Information Exchange between Law Enforcement Agencies. In *Proceedings of the 21st Workshop on Privacy in the Electronic Society*. 69–82.
[46] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. 2023. {HECO}: Fully Homomorphic Encryption Compiler. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4715–4732.
[47] Alexander Viand, Christian Knabenhans, and Anwar Hithnawi. 2023. Verifiable fully homomorphic encryption. *arXiv preprint arXiv:2301.07041* (2023).

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 $\mathcal{M}$, an MHE scheme over $\mathcal{P}$ and $\mathcal{M}$ is a tuple of algorithms and multiparty protocols $\mathsf{MHE} = (\mathsf{GenParam}, \Pi_{\mathsf{SecKeyGen}}, \Pi_{\mathsf{EncKeyGen}}, \Pi_{\mathsf{EvalKeyGen}}, \mathsf{Encrypt}, \mathsf{Eval}, \Pi_{\mathsf{Decrypt}})$ whose elements have the following syntax and semantic:

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

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

- **Encryption-key gen.**: $\mathsf{cpk} \leftarrow \Pi_{\mathsf{EncKeyGen}}(\{\mathsf{sk}_i\}_{P_i \in \mathcal{P}'})$
  From any subset of secret keys $\{\mathsf{sk}_i\}_{P_i \in \mathcal{P}'}$ such that $\mathcal{P}' \subseteq \mathcal{P}$ and $|\mathcal{P}'| \geq T$, $\Pi_{\mathsf{EncKeyGen}}$ outputs a *collective public encryption key* cpk.

- **Eval.-key gen.**: $\mathsf{evk}_{\mathsf{op}} \leftarrow \Pi_{\mathsf{EvalKeyGen}}(\mathsf{op}, \{\mathsf{sk}_i\}_{P_i \in \mathcal{P}'})$
  Given a homomorphic operation op to be supported by the Eval algorithm and any subset of secret keys $\{\mathsf{sk}_i\}_{P_i \in \mathcal{P}'}$ such that $\mathcal{P}' \subseteq \mathcal{P}$ and $|\mathcal{P}'| \geq T$, $\Pi_{\mathsf{EvalKeyGen}}$ outputs a public evaluation-key $\mathsf{evk}_{\mathsf{op}}$ for operation op.

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

- **Evaluation**: $\mathsf{ct}_{\mathsf{res}} \leftarrow \mathsf{Eval}(f, \{\mathsf{evk}_{\mathsf{op}}\}_{\mathsf{op} \in f}, \mathsf{ct}_1, \ldots \mathsf{ct}_I)$
  Given an arithmetic function $f : \mathcal{M}^I \rightarrow \mathcal{M}$, the evaluation key $\mathsf{evk}_{\mathsf{op}}$ for each homomorphic operation op used in f and an I-tuple of ciphertexts $(\mathsf{ct}_1, ..., \mathsf{ct}_I)$ encrypting $(m_1, \ldots, m_I) \in \mathcal{M}^I$, Eval outputs a ciphertext $\mathsf{ct}_{\mathsf{res}}$ that is the encryption of $m_{\mathsf{res}} = f(m_1, \ldots, m_I)$.

- **Decryption**: $m \leftarrow \Pi_{\mathsf{Decrypt}}(\mathsf{ct}, \{\mathsf{sk}_i\}_{P_i \in \mathcal{P}'})$
  Given ct an encryption of m, and any subset of secret keys $\{\mathsf{sk}_i\}_{P_i \in \mathcal{P}'}$ such that $\mathcal{P}' \subseteq \mathcal{P}$ and $|\mathcal{P}'| \geq T$, Π_{Decrypt} outputs m.

Current MHE scheme constructions [37] are based on the *ring-learning with errors* (RLWE) problem [34]. 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 $\Pi_{\mathsf{EvalKeyGen}}$ 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. A notable aspect of the current MHE schemes is that all their protocols can be executed in a single round of communication [37, 38]. We provide a unified model for these protocols in Section 4.1.

B URI SCHEME

In our URI scheme, each ciphertext can be identified by its holder node identifier, circuit identifier, and ciphertext identifier:

```
helium://<NodeID>/<CircuitID>/<CiphertextID>
```

Depending on the context, parts of the identifiers can be omitted by the user application designer (as in Listing 1), and expanded by the framework at execution time. This enables the application designer to define Helium programs in a generic way and to execute them several times (see Challenge 1). For instance, the `SessionID` and `CircuitID` fields can be expanded to the session identifier in which the circuit was executed and the circuit identifier that was attributed to it. Similarly, the `NodeID` field (the host part) can be omitted for intermediate values and the output, as it can be assumed to be the circuit's evaluator. Note that when all three fields are present, the

URI enables a ciphertext to be located unambiguously within the system, hence is also a URL.