ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

Shechi: A Secure Distributed Computation Compiler Based on Multiparty Homomorphic Encryption

Haris Smajlović* *University of Victoria*

David Froelicher* *MIT*

Ariya Shajii *Exaloop Inc.*

Bonnie Berger *MIT*

Hyunghoon Cho *Yale University*

Ibrahim Numanagić *University of Victoria*

Abstract

We present Shechi, an easy-to-use programming framework for secure high-performance computing on distributed datasets. Shechi automatically converts Pythonic code into a secure distributed equivalent using multiparty homomorphic encryption (MHE), combining homomorphic encryption (HE) and secure multiparty computation (SMC) techniques to enable efficient distributed computation. Shechi abstracts away considerations about the private and distributed aspects of the input data from end users through a familiar Pythonic syntax. Our framework introduces new data types for the efficient handling of distributed data as well as systematic compiler optimizations for cryptographic and distributed computations. We evaluate Shechi on a wide range of applications, including principal component analysis and complex genomic analysis tasks. Our results demonstrate Shechi's ability to uncover optimizations missed even by expert developers, achieving up to 15× runtime improvements over the prior state-of-the-art solutions and a 40-fold improvement in code expressiveness compared to code manually optimized by experts. Shechi represents the first MHE compiler, extending secure computation frameworks to the analysis of sensitive distributed datasets.

1 Introduction

Analyzing large and diverse datasets distributed among multiple stakeholders is crucial in numerous domains that involve sensitive data, such as biomedical and financial information. However, concerns about privacy and intellectual properties, as well as legal frameworks [31, 39] that restrict the sharing of sensitive data, have resulted in the creation of data silos that significantly hamper data analytics [72].

Modern cryptographic techniques offer promising strategies to address these concerns by allowing one to perform computations securely over private data. These methods enable multiple institutions to jointly run analyses on combined data through the exchange of encrypted information, typically leveraging fully homomorphic encryption (HE), secure multiparty computation (SMC), or their combination in the form of multiparty HE (MHE) to protect the privacy of each party's data [6, 18, 26–29, 42, 54, 74]. However, these methods come with several challenges that prevent their widespread adoption. In the case of HE, the computational overhead associated with its cryptographic primitives often leads to impractical runtimes on large datasets or necessitate the adoption of simplified, less accurate variants of the desired analysis. On the other hand, SMC involves interaction among multiple parties and typically suffers from a high communication overhead because all data must be shared and synchronized among the parties during the execution.

Although a hybrid framework based on MHE can, in principle, lead to a significant reduction in computational costs on large-scale distributed datasets compared to existing methods based on HE and SMC, to the best of our knowledge, there does not exist a compiler for MHE that can streamline the development of such protocols. As a result, secure distributed software development currently demands profound expertise in the often disparate domains of cryptography, distributed algorithms, and domain-specific analytics. It typically entails (i) manually designing a distributed yet equivalent version of the original algorithm, (ii) developing a secure version of this algorithm by manually integrating cryptographic primitives while considering their capabilities and limitations, (iii) implementing the algorithm using low-level cryptographic libraries, and (iv) manually optimizing its performance. Each of these steps can affect a solution's runtime by several orders of magnitude, making the difference between practical and infeasible solutions [29]. Moreover, these steps typically must be considered jointly, requiring intricate manual optimizations and resulting in complex implementations whose security is difficult to verify, either manually or automatically, and challenging to maintain in the future.

Here, we introduce Shechi, the first programming framework that automates the transformation of standard high-level Pythonic code into an efficient and secure MHE equivalent for execution on distributed datasets. Shechi enables end users

*Co-first authors.

to write code in familiar syntax without needing prior cryptography or distributed algorithms expertise. It functions as an end-to-end ahead-of-time compiler that analyzes and compiles compatible Pythonic code and optimizes it through a set of MHE-specific static and dynamic optimization passes. It is built on top of Codon [65–67], an LLVM-powered [44] ahead-of-time compiler that allows seamless integration of domain-specific optimization passes on top of Pythonic code. Shechi introduces new high-level code optimization strategies that operate on top of secure expressions and leverage the specific features of MHE to improve the performance of evaluating such expressions. It notably implements various compile-time and runtime code optimizations for parallel computation over large encrypted vectors, effective workload distribution among computing parties, and efficient local computations on non-encrypted data. Shechi also introduces new data types—local and distributed secure tensors—that encapsulate data partitioned among parties to orchestrate distributed computations on top of them. To overcome the computational limitations of HE and leverage the versatility of SMC computations without sharing all input data, Shechi also handles the dynamic orchestration between HE and SMC for specific computations that allow it, thus enabling the writing of complex applications that can handle large-scale private input data. For example, Shechi supports essential matrix and vector operations from the standard NumPy library [35] and provides a range of machine learning routines, including linear and logistic regression, support vector machines, and neural networks.

Our evaluations show that Shechi achieves comparable performance with low-level HE and SMC libraries and outperforms other state-of-the-art high-level secure compilers. We showcase the effectiveness of our solution through the design and evaluation of various large-scale data analytics workflows, including principal component analysis (PCA) and complex tasks in genomics. We also integrated a specialized Keras-like [19] library into Shechi, enabling, for the first time, the easy, flexible, and practical implementation and execution of neural network training in the MHE context. Many of these workflows are too complex for existing solutions to handle effectively due to their complexity and the scale of the data they process. For all applications, our solution led to easy-to-read, pseudocode-like Pythonic implementations of the algorithms and improved the expressiveness of the existing secure solutions by up to 40× and achieved up to 15× better runtime performance. Shechi also scales better than other approaches as the number of parties and data dimensions grows: we show that its runtime only increases by a factor of 1.5 compared to 5.5 for a secret-sharing-based SMC solution when the number of parties and data dimensions are quadrupled. Shechi's systematic approach enables it to match or even surpass the performance of manually optimized, expert-written code. Thus, our framework for secure distributed computation enables practitioners to write secure applications easily and constitutes an important step for the design and adoption of secure distributed solutions.

In summary, here we present:

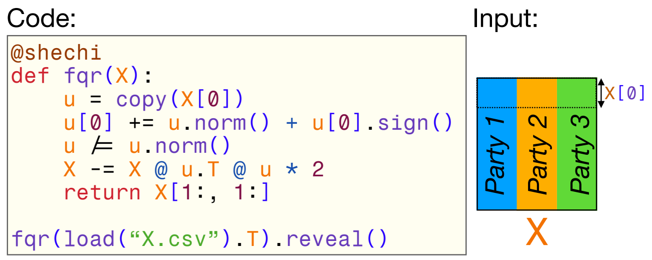
- Shechi, a new modular programming framework and compiler for secure distributed applications that leverage multiparty homomorphic encryption.
- New data types and a set of associated libraries designed to encapsulate data distributed among the parties and effectively orchestrate distributed computation.
- A set of compile-time code analysis and runtime optimization passes that enable translating standard Pythonic code into efficient, secure distributed equivalents.
- Demonstration on real-world datasets and applications that Shechi effectively translates conventional Pythonic code into secure and performant distributed workflows, even for complex tasks that cannot be practically instantiated with the existing secure compilers.

Our software is open source and publicly available at https://zenodo.org/records/14725520.

2 Problem Statement

In this work, we address the following problem: how to automatically transform a high-level Pythonic program that analyzes private structured data (such as matrices) split across multiple parties into a secure distributed equivalent with scalable performance, using multiparty cryptographic primitives for security guarantees. The data-holding parties should obtain only the final result and must not learn any other information about the other parties' data beyond what can be inferred from the final result (i.e., input privacy must be respected) or from the application code (which is considered public). We assume that the data-holding parties are honest-but-curious and non-colluding—in other words, they are willing to collaborate faithfully but might try to infer information from the protocol execution. Under this model, the parties execute the assigned programs without modification and do not use artificial inputs crafted to extract information about other parties' data.

We focus on secure analytics in cross-silo settings where each party holds a typically large-scale subset of the joint dataset. We consider general-purpose programs that analyze structured data (typically through vector and matrix operations), which are fundamental in many data-centric domains (e.g., machine learning, statistical analysis and image processing). An example of such code is shown in our lead example in Fig. 1 where the user defines a forward pass of QR decomposition [3]—an essential procedure in many data analytics workflows such as dimension reduction through PCA—by expressing it as a standard Numpy-backed Python procedure. This procedure has to be executed on the dataset in the matrix shape partitioned across multiple parties.



```
@shechi
def fqr(X):
    u = copy(X[0])
    u[0] += u.norm() + u[0].sign()
    u /= u.norm()
    X -= X @ u.T @ u * 2
    return X[1:, 1:]

fqr(load("X.csv").T).reveal()
```

Figure 1: Example of high-level Pythonic code (left; forward pass of QR decomposition) to be securely executed on the input `X` split vertically among 3 parties (right). Manually implementing this distributed procedure with existing secure solutions either requires large code that breaks down all steps and coordinates data among parties or results in non-scalable executables.

3 Related Work

Non-secure federated frameworks (NSFFs) [25, 30, 55, 58] typically distribute computations across multiple parties (i.e., workload distribution) to enable efficient and versatile operations on distributed datasets. However, they involve exchanging non-protected intermediate results among parties, potentially disclosing information about the input data [53, 76]. Existing secure solutions, on the other hand, usually require the input data to be encrypted and then shared among computing parties. Computations are then performed on the encrypted data, leading to communication and computation overheads that quickly grow with workflow complexity, number of computing parties, and data scale. We summarize existing frameworks and their comparisons with our approach in Table 1.

Several HE-based compilers have been proposed [32, 70] to facilitate the development of applications on encrypted datasets. They target tasks such as machine learning inference [2, 7–9, 11] or general analytics [20, 22, 71] and are typically built on top of HE schemes [10, 14] that enable efficient vector arithmetic through encoding (or packing), where multiple values are encrypted within a single ciphertext. These schemes support a limited number of multiplications due to the lack of practical bootstrapping, an operation required after a certain number of multiplications to ensure decryption correctness. Additionally, each multiplication must be followed by ciphertext maintenance operations to manage ciphertext size and scale. Existing compilers address these challenges at different levels of granularity. EVA [20, 23] optimizes circuit evaluation to reduce ciphertext maintenance overhead, while HECO [71] introduces advanced optimizations for utilizing encoding in programs requiring fine-grained access to vector elements. These compilers typically add an extra layer on top of existing low-level HE libraries, which allows them in some cases to benefit from underlying improvements but also limits the development of interlevel (or full-stack) optimizations. Furthermore, they focus primarily on vector operations, lacking native support for matrix operations.

SMC compilers and frameworks [36] typically offer more versatile computations compared to HE but require interactions between computing parties. In SMC, the input data are secret shared among multiple computing parties, which interact to compute on the data without learning any information about the data. Like HE solutions, SMC compilers provide varying levels of abstraction to users. For example, SMC frameworks such as MPyC [62], MP-SPDZ [41], and Sequre [68] transform high-level Pythonic code into SMC-enabled applications. The latter two also automatically optimize the underlying source code at compile-time. However, despite these and other developments in the SMC domain [36], SMC pipelines still do not scale well with the growth of the input data and the number of computing parties.

Hybrid solutions that extend HE techniques to a multiparty setup [56] (MHE) have been proposed for specific applications such as machine learning [27, 61, 74], PCA [29] and genomics [18]. These solutions leverage local, non-encrypted computation by each party to improve the overall efficiency of the protocol. However, developing practical MHE protocols poses a significant challenge. Similarly to HE compilers, effective use of MHE requires selecting appropriate parameters, managing multiplicative depth, ensuring correct decryption, and exploiting ciphertext encoding for practical performance. It also requires orchestrating computations across parties to leverage efficient local operations while ensuring security by encrypting any shared data and emulating a centralized execution by aggregating local intermediate results when needed.

For example, the matrix `X` in Fig. 1 is distributed across multiple parties where each party holds its own share as plaintext. It is easy to select the first row of `X` (as `u`) efficiently in plaintext without network delay. L_2 norm ($\sqrt{u@u^T}$) of `u` (`u.norm`) can be efficiently and securely evaluated via MHE by computing the partial squared norms at each party holding a partition of `u` before aggregation under encryption. The square root of the resulting squared norm can, on the other hand, be evaluated with an SMC routine based on bitwise decomposition that is usually more efficient, numerically stable and easier to parametrize than the HE equivalent. In either case, the exact order of the steps and, in turn, the performance for computing $X\ @\ u.T\ @\ u \cdot 2$ significantly depends on the way `X` is distributed and encoded.

These interconnected design decisions are pivotal, and a single change, such as encoding or aggregation methodology, can significantly impact overall performance. In the current state-of-the-art MHE methods, these decisions are made manually by developers through an iterative and repetitive process, often overlooking performance and security implications and leading to convoluted codebases.

Framework	**End-to-end confidentiality**	**Trust distribution**	**Workload distribution**	**Computation versatility**	**Native matrix ops.**	**High-level prog.**	**Full-stack opti.**	**Ahead-of-time comp.**	**Type check**
NSFF	✗	∼	✓	✓	✓	✓	✓	✓	✓
SMC (MP-SPDZ | MPyC | Sequre)	✓	✓	✗	✓	✓ | ✓ | ✓	✓ | ✓ | ✓	✓ | ✗ | ✓	✓ | ✗ | ✓	✓ | ✗ | ✓
HE (HEFac. | EVA | HECO)	✓	✗	✗	✗	✗ | ✗ | ✗	✓ | ✓ | ✗	✗ | ✗ | ✗	✗ | ✗ | ✓	✗ | ✗ | ✓
Our Approach	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 1: Survey of the current strategies for non-secure and secure distributed computing. NSFF stands for non-secure federated framework. End-to-end confidentiality ensures input data remains confidential throughout the protocol execution. Trust distribution means trust is shared among multiple parties. Computation versatility refers to the approach's ability to support a wide range of operations, similar to standard centralized data analysis. The rightmost columns indicate whether solutions natively support operations at the matrix level, whether they allow code to be written in a high-level programming language and to be compiled ahead of time, and whether the solutions can enforce strong typing.

4 Overview of Shechi

Shechi enables users without specialized expertise in cryptography and distributed computing to write non-secure standard code in a Python-like manner designed for a single machine and execute it across partitioned data in an MHE setting (Fig. 2) while maintaining data confidentiality. Partitioned data remains unencrypted with each party for local computations and is encrypted only when shared across parties for global calculations. This approach allows parties to leverage efficient plaintext operations throughout the protocol. In the end, only the final results are decrypted via collective decryption. For example, user can securely execute our lead example in Fig. 1 on partitioned data (`X`) by sharing the code or the executable compiled with Shechi with all parties and executing it with a single command without having to manually optimize it or orchestrate across multiple parties (Fig. 13 in Appendix G).

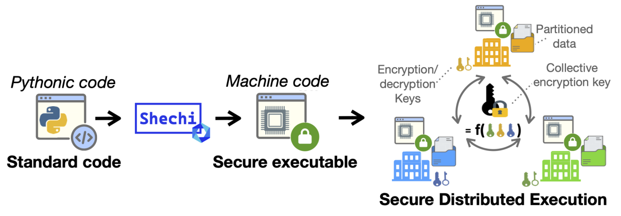


Figure 2: Shechi automatically translates standard centralized Pythonic code into a secure distributed workflow to be executed in a multiparty setting.

Shechi integrates multiple optimization strategies in a compiler framework, which are systematically applied to automatically translate high-level code into a secure distributed equivalent. It consists of four main elements:

1. *End-user Interface:* Shechi uses Python's syntax and semantics to allow users to write performant secure solutions via simple, pseudocode-like code.
2. *Secure Data Types:* Shechi introduces two new secure data types, named *secure distributed tensors* and *secure local tensors*, along with methods and protocols that orchestrate secure computation on top of it. Through distributed tensors, Shechi ensures the private data is kept locally at each party in plaintext while encrypting any shared data.
3. *Compiler Optimizations for Secure Distributed Computation:* On top of its secure data types, Shechi introduces optimization passes for analyzing and optimizing high-level secure code both at compile time and runtime. These passes enable Shechi to reduce the workload and optimize secure computations.
4. *Integrated MHE Libraries:* Shechi re-implements cryptographic libraries in its framework to enable low-level compiler optimizations.

Shechi's overall structure is depicted in Fig. 3 and we detail its components in the following sections.

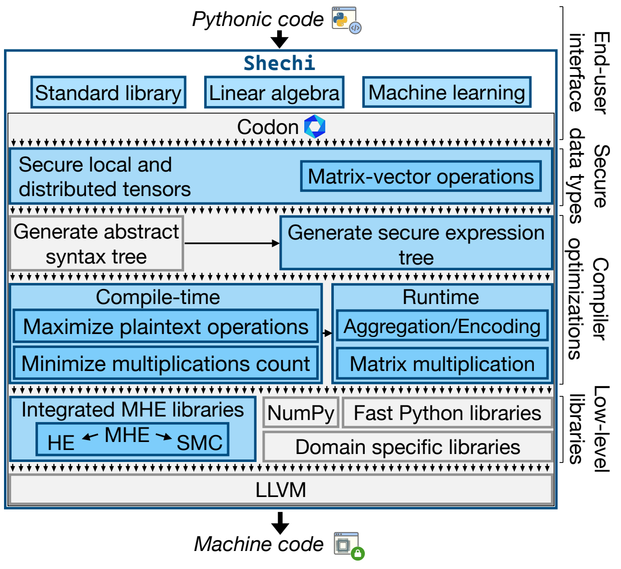


Figure 3: Shechi's overall structure. Shechi's novel components are highlighted in blue. Based on the Codon framework, Shechi introduces novel programming components to streamline Python-based codebases for secure distributed workflows.

5 System Components

5.1 Codon Compiler Framework

Shechi builds on top of the Codon compilation framework [67], a high-performance ahead-of-time compiler that compiles Pythonic code to native machine code through several steps: (i) parsing Pythonic code into an abstract syntax tree (AST) representation; (ii) statically type-checking the AST; (iii) generating a high-level intermediate representation (IR) of the type-checked AST that is later subject to various code analysis and optimization passes; and finally, (iv) converting the IR to the machine code through the LLVM compiler framework [44]. The Codon library also defines multiple specialized modules that provide efficient implementations of Python standard libraries, such as NumPy [35], and domain-specific optimizations [65], as well as standard compiler optimizations such as dead code elimination. While Codon provides a convenient framework for analyzing and optimizing Python code, it does not offer any native support for cryptographic schemes for secure computation or distributed computation, both of which are addressed by this work. Shechi introduces new secure data types with new matrix-vector arithmetic (*Secure data types* in Fig. 3), MHE-specific IR optimization passes (*Compiler passes*) and MHE-specialized modules (*Integrated MHE Libraries*).

5.2 Multiparty Homomorphic Encryption

MHE, first introduced as a distributed HE scheme [56] has then been extended to a hybrid scheme allowing to switch from data encrypted in HE under a collective key to secret-shared data in SMC [18]. In our setting, the decryption key is shared among the parties via n-out-of-n (additive) secret sharing, while the corresponding encryption key is known to all of them. Each party can independently compute in HE under the public key, but decryption requires collaboration from all parties. Interactive protocols can be used to replace the expensive HE bootstrapping operation and to convert HE ciphertexts into additive shares for SMC operations [18]. MHE can therefore be used to divide operations across HE and SMC while leveraging the strengths of both schemes.

We instantiate MHE with the Cheon-Kim-Kim-Song (CKKS) scheme [14] for distributed HE. In CKKS, plaintexts (unencrypted) and ciphertexts (encrypted data) are represented by polynomials of degree up to $\mathcal{N}-1$ (with $\mathcal{N}$ coefficients), each encoding a vector of up to $t = \mathcal{N}/2$ floating-point values. Security is based on the ring learning with errors (RLWE) problem [50] and some noise is added directly in the least significant bits of the encrypted values. CKKS supports a limited number of operations on top of data: additions, multiplications and vector rotations. All vector-wide operations are data-parallel and can be simultaneously executed on all vector values. However, intra-vector operations are usually costly as they require a sequence of HE operations. Additions and multiplications with plaintexts are faster than ciphertexts multiplications and rotations[1]. Furthermore, to maintain the ciphertext size and scale—values are scaled by a constant before encryption to ensure a high level of precision—ciphertexts have to be *rescaled* after any multiplication and *relinearized* after multiplication with another ciphertext. After a certain number of multiplications, the ciphertext also needs to be *refreshed* through a *bootstrapping* procedure to ensure correct decryption. This operation is prohibitively expensive in the standard CKKS scheme. To prevent information leakage upon decryption [49], a fresh noise with a variance larger than that of the ciphertext is added by each party before collective decryption, known as *smudging* [15, 48, 56] (Appendix F).

For SMC, we rely on a linear, n-out-of-n secret-sharing scheme [17, 18] in which private inputs are split into additive shares encoded in a prime field. Each operation (e.g., addition, multiplication etc.) is distributed and may incur heavy communication costs. Additions are simple share additions, while multiplications utilize Beaver multiplication triples [4]. For efficiency, this scheme adopts a server-aided model of preprocessing where a trusted third party generates the Beaver triples to facilitate the main interactive computations. This scheme enables a wide range of basic operations and notably supports efficient bit decomposition routines to convert secret shares into bitwise shares (i.e., in $\mathbb{F}_2$). It allows for efficient bitwise computations such as comparisons, division and square roots [51], which are not natively supported in HE.

6 Shechi's End-User Interface

Shechi takes as an input standard Pythonic code and converts it into fast executables that operate on top of distributed data (Fig. 2). Shechi enables users to run the Pythonic procedure (*fqr*) from Fig. 1 in MHE setup by simply preceding it with a specialized function decorator (`@shechi`). Shechi adopts NumPy's conventions and API calls to enable users to quickly write arithmetic expressions on top of both non-encrypted cleartext and encrypted distributed data. For instance, to perform secure matrix multiplication, it suffices to write a single operator `@` (Fig. 1), and leave all necessary secure and distributed computations to Shechi to be handled behind the scenes. In addition to the elementary operations between encrypted tensors (i.e., matrices and vectors), Shechi provides a set of low-level cryptographic primitives (e.g., encryption and basic arithmetic), secure equivalents of most of Python's built-in functions and popular NumPy operations, and built-in secure linear algebra routines. Finally, Shechi includes a machine learning module with secure regression analysis and neural networks library that implements the popular Keras's [19] API to facilitate secure machine learning workflows on distributed data.

[1]Up to 7× times faster in our experiments; see Appendix F.

For efficient MHE operations, Shechi introduces specific optimizations (compiler passes step in Fig. 3) at different compiler levels through both Codon's and LLVM's intermediate representations and applies them not only to the user's codebase but also within internal libraries and low-level cryptographic modules. Additionally, Shechi inherits Codon's generic compile-time optimizations such as dead code elimination, canonicalization, and common subexpression elimination, and benefits from its other features, such as performant equivalents of Python's built-in modules and popular libraries for the execution of non-secure code blocks (without the `@shechi` decorator).

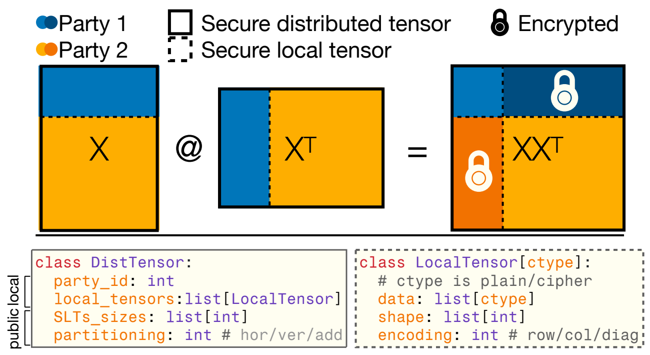
As an ahead-of-time, static compiler, Shechi does not support some of Python's runtime (dynamic) features, such as monkey patching or heterogeneous collections (which are rare in secure programs). Also, similar to other secure compilers, Shechi does not support generic conditional branching on top of encrypted operands but does support conditionals that can be expressed in a branchless fashion through masking or secure multiplexer [71]. Thus, while Shechi aims to be a drop-in replacement for the existing Python pipelines, users still need to account for these differences.

7 Shechi's Secure Data Types

In a distributed scenario, multiple parties hold the input data (e.g., `X` in Fig. 1) in cleartext. Computations on this data can result in new partially-encrypted operands. For example, the multiplication `X @ X.T` where `X` is split among two parties would result in a hybrid new matrix in which some parts are kept locally in cleartext, and some are encrypted (Fig. 4). Maintaining cleartext partitions is critical for performance, especially in a cross-silo setting where the input data are often of large scale. However, keeping track of the state of the data quickly becomes difficult when complex operations and a higher number of parties are introduced. Moreover, all collective operations (such as collective bootstrapping or switching to secret shares) must be carefully orchestrated on top of encrypted portions of the matrix and synchronized between the parties. To facilitate optimizations that jointly considers these issues, Shechi introduces two data types to streamline operations on secure distributed datasets: secure local and distributed tensors (Fig.4) and targeted optimizations (§.8).

7.1 Secure Local Tensors

Secure local tensors are n-dimensional array structures (either in cleartext or encrypted) that store a portion of the shared data residing at a single party. For example, each matrix partition (blue and yellow, dotted sections) in Fig. 4 is a local tensor. Similar to conventional tensor implementations like NumPy [35], Shechi keeps the n-dimensional data in a single, contiguous vector and lazily tracks the data dimensions (*shape* attribute in *LocalTensor class* in Fig. 4). This approach



```
class DistTensor:
  party_id: int
  local_tensors:list[LocalTensor]
  SLTs_sizes: list[int]
  partitioning: int # hor/ver/add
```

```
class LocalTensor[ctype]:
  # ctype is plain/cipher
  data: list[ctype]
  shape: list[int]
  encoding: int # row/col/diag
```

Figure 4: Secure distributed tensor example and definition. Matrix multiplication $X@X^\top$ is split among two parties. The data held locally at each party (bright parts) are used to compute the majority of the result in cleartext. Only a fraction of the data needs to be encrypted (parts with a lock). A secure distributed tensor (X, X^T or XX^T) is a list of secure local tensors and has global information about the partitioned data, i.e., all partitions' sizes (*SLTs_sizes*) and partitioning format.

enables the expensive HE tensor primitives, like transposing encrypted matrix, to be done in a lazy manner by simply adjusting the *encoding* attribute while, at the same time, allowing efficient execution of any operation on non-encrypted portion through efficient Codon's Numpy methods.

When encrypted, local tensors can be encoded either row-wise, column-wise or diagonal-wise. Different encodings require different algebraic procedures to be invoked for each arithmetic operation. Shechi provides such procedures (e.g., matrix multiplication) for local tensors of arbitrary shapes and encoding, and applies optimization passes to select the optimal encoding for each local tensor (§.8).

7.2 Secure Distributed Tensors

Secure distributed tensors represent vectors and matrices securely shared and distributed across multiple parties. They can be partitioned (or *split*) horizontally, vertically, or additively between the parties (i.e., each party holds some rows, columns, or additive shares of the original matrix; Fig. 4). Each share of a secure distributed tensor is a secure local tensor that is stored separately at each party together with auxiliary information such as dimensions of other shares and partitioning type. Auxiliary information is shared in cleartext among the parties. For example, matrix `X` in Fig. 4 is partitioned horizontally while its transpose `X.T` is partitioned vertically between two parties. Both parties see only their local non-encrypted partitions of `X` and `X.T`. Each operation on a distributed tensor is automatically translated into a series of operations on the underlying secure local tensors.

All operations are decoupled into local and global operations. Local operations, such as any element-wise operations on matrices, can be computed independently at each party without sharing any data. For example, any operation on the matrix X in Fig. 4 of the form $\beta_1 X + \cdots + \beta_n X$, where $\beta_1, \ldots, \beta_n$ are scalars, can be evaluated independently at each party. Global operations on secure distributed tensors often require data exchange because they interact with local tensors stored at different parties. In Fig. 6 (Step 2), this happens when $Xu^\top$ (composed of secure local tensors $X_i u_i^\top$ at each party i) is multiplied with the partitioned vector u. In contrast, Step 1 ($X @ u^\top$) can be computed solely through local multiplications between local tensors at each party. When needed, to distribute the workload and leverage efficient operations involving local non-encrypted data, Shechi first *aggregates* one operand among all parties (e.g., either $Xu^\top$ or u in Step 2) and then splits the computation into local operations between secure tensors and the aggregated distributed tensor at each party. The results of these operations remain encrypted throughout the computation. The amount of encrypted data in the result, as well as the partitioning type of the result, depends on which operand is aggregated. This choice impacts the performance of this operation and of all related downstream operations. Shechi analyzes the code block at compile time to select the operand to be encrypted and thus minimizes the cost of the overall distributed computation (§.8).

Both kinds of tensors are either encrypted with HE or secretly shared for SMC, and can be converted between the two representations depending on the operation (§. 8.3).

8 Shechi's Compiler Optimizations

Shechi's code optimizations start with code analysis, where the information on high-level code structure is captured, and proceed to compiler passes, where code is transformed into optimized counterparts. MHE-specific optimizations are then applied at runtime for an efficient and secure execution of the generated code.

8.1 Code Analysis & Optimization Workflow

To effectively orchestrate secure distributed computations while leveraging fast local operations and maintaining correctness, Shechi relies on insights from both the static code structure, available only at compile-time, and also at the dynamic information, such as data dimensions, available only at runtime. As finding good aggregation strategies for secure distributed tensors and encoding strategies for secure local tensors that effectively leverage MHE while minimizing computation is crucial for the performance of MHE programs, Shechi introduces a set of compiler passes that can address these (and similar) optimization problems in an MHE context. An overview of these optimizations applied to the expression `X @ u.T @ u*2` (Fig. 1) is shown in Figs. 5 and 6.

Shechi begins by analyzing the abstract syntax tree of each secure procedure at compile time to generate a *secure expression tree* (Step 1 in Fig. 5). This tree encapsulates each expression operating on top of secure data and is used for the subsequent analysis and optimization of such expressions. Each leaf node in the tree corresponds to an operand, while inner nodes correspond to arithmetic operations. Shechi automatically marks data resulting from collaborative computations as encrypted (e.g., `u` in Fig. 5), while other variables are labeled as cleartext. Before interactive operations, any cleartext variables that need to be shared are automatically encrypted at runtime. Additional static metadata, such as a data type, can also be stored in the tree. For optimizations that do not depend on runtime information, an optimization pass analyzes the secure expression tree and applies the necessary code transformation immediately at compile time (Step 2 in Fig. 5). Such passes include prioritization of lightweight plaintext over ciphertext computing and multiplicative redundancy minimization of secure expressions (detailed later in this section).

For decisions that require runtime information, Shechi encodes the secure expression tree as a dynamic object that can be accessed at runtime (Step 3 and `tree` in code snippet in Fig. 5). This runtime object includes the auxiliary information, such as data dimensions and partitioning type of the distributed tensors, encoding of the underlying local tensors, and other metadata required to facilitate the optimization. Optimizations at this stage typically generate a code that assesses various potential configurations at runtime and selects the best course of action. Note that this tree structure and the associated metadata format are general and thus suitable for additional optimization strategies in the future.

8.2 Compiler Passes

Aggregation & encoding optimization. Shechi executes all operations on secure distributed tensors through a combination of local and distributed operations. Element-wise operations (e.g., `* 2` in Fig. 1) and item selection (e.g., `u[0]`) are executed locally by the parties. Data reductions (such as summing all elements in a vector to compute `u.norm()`) are performed locally, with results aggregated among the parties as needed (e.g., summing local norms to obtain the global norm). Matrix multiplications, however, require more analysis, as their cost depends not only on how the data is combined across parties (i.e., which operands are *aggregated* first), but also on the dimensions of the operands and the encoding of underlying secure local tensors. The aggregation choice also determines the partitioning of the result, which in turn impacts the performance of downstream operations. For example, a subsequent multiplication of the result of `X @ u.T` with `u` (Step 2 in Fig.6) is executed either on a single partitioned matrix with the left aggregation strategy (*Aggregate: **left***), or on a full matrix shared additively at each party (*Aggregate: **right***). In this case, the latter is less efficient. Similarly, the choice of encoding for an unencrypted secure local tensor determines the encoding of the output and the cost of all subsequent op-

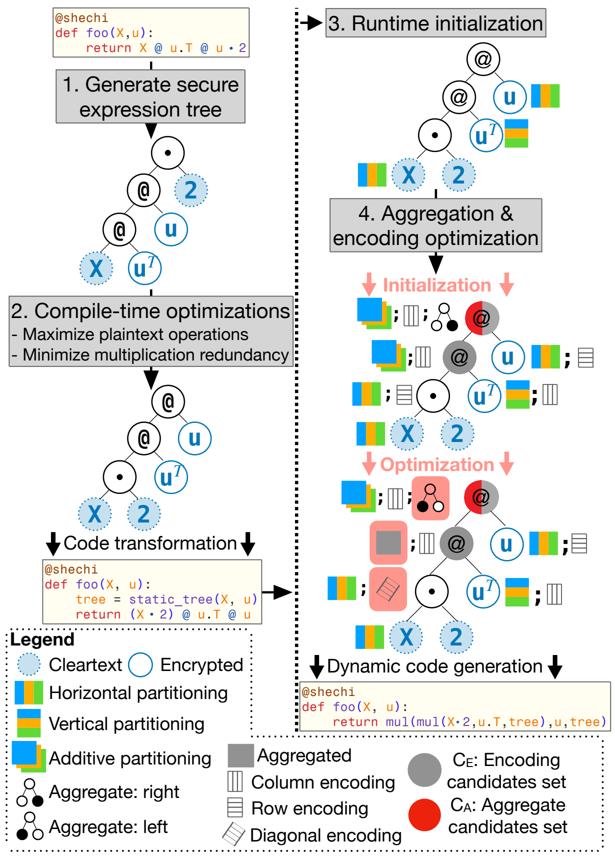

```
@shechi
def foo(X,u):
    return X @ u.T @ u • 2
```
```
@shechi
def foo(X, u):
    tree = static_tree(X, u)
    return (X • 2) @ u.T @ u
```
```
@shechi
def foo(X, u):
    return mul(mul(X•2,u.T,tree),u,tree)
```
1. Generate secure expression tree
2. Compile-time optimizations
- Maximize plaintext operations
- Minimize multiplication redundancy
Code transformation
3. Runtime initialization
4. Aggregation & encoding optimization
Initialization
Optimization
Dynamic code generation
Legend
Cleartext
Encrypted
Horizontal partitioning
Vertical partitioning
Additive partitioning
Aggregate: right
Aggregate: left
Aggregated
Column encoding
Row encoding
Diagonal encoding
CE: Encoding candidates set
CA: Aggregate candidates set

Figure 5: Shechi's code analysis and optimization passes. Shechi begins by generating a secure expression tree (Step 1). Optimizations on this tree can be either static (left) or dynamic (right). Compile-time (static) optimizations rewrite expressions for optimal MHE performance (Step 2). Runtime (dynamic) optimizations encode this tree as a runtime object (Step 3) and use it to optimize computations on secure data types, i.e., selecting an optimal aggregation and encoding strategy (Step 4). This part helps to distinguish between the local cleartext and global encrypted computing (each aggregation choice defines the encrypted portion of the data). This step is further explained in Fig. 6. The tree structure includes basic information about the code structure (such as the order of operations and variable types), as well as runtime metadata (such as matrix dimensions and encoding types).

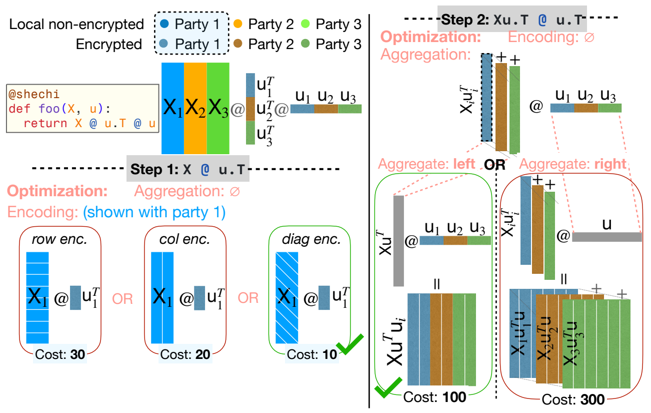

```
@shechi
def foo(X, u):
    return X @ u.T @ u
```

Figure 6: Aggregation & encoding Optimization. The performance of multiplication `X @ u.T @ u` depends on the choice of encoding of the non-encrypted operand (`X`) and the choice of aggregation strategy. In this example, Shechi first calculates `X @ u.T` (*Step 1*). As the shares of `X` are non-encrypted, the best encoding for local tensor `X` is chosen among all possible encodings (in this case, diagonal encoding). No operand needs to be aggregated in Step 1 and the output is encoded as a single vector. Afterwards, because multiplication with `u` requires communication between parties, Shechi selects the optimal aggregation strategy for distributed tensors (*Step 2*). Here, Shechi chooses the left aggregation strategy over the right. Both operands are already encoded from previous steps.

erations. For example, in `X @ u.T`, the local partitions of `X` are not encrypted at this point, while the partitions of `u.T` have been encrypted in earlier steps. The choice of tensor encoding for `X` will impact the cost of computing the remaining matrix operations (both Step 1 and Step 2 in Fig. 6).

Finding the optimal aggregation and encoding strategies is a combinatorial problem that can significantly impact a solution's runtime, particularly in complex workflows with large-scale inputs (§.10.4). The input data is initially kept in cleartext by each party, allowing arbitrary encoding upon encryption as needed. This provides flexibility to the encoding optimization algorithm in choosing the best encoding strategy. Although each party may manually set the initial encoding if desired, manual optimization becomes daunting given a complex protocol with a larger number of operations. Shechi finds the best aggregation and encoding strategy (Step 4 in Fig. 5; Fig.6) as follows. After extracting the necessary static information (such as the secure expression tree) at compile-time, Shechi extends it with the necessary dynamic metadata, such as the dimensions and initial partitioning of the existing variables, as soon as it is available at runtime (Step 3 in Fig. 5). It then pinpoints all multiplication nodes in the tree for which one secure distributed operand needs to be aggregated (C_A; red in Fig. 5) and for which the other operand's partitions (i.e., secure local tensors) are not encrypted and can be flexibly encoded (C_E; gray in Fig. 5). It initializes all aggregation and encoding strategies following a greedy approach, i.e., selecting the most cost-efficient approach at each node (Step 4 - Initialization). The cost of individual operations (e.g., addition, multiplication, etc.) is automatically computed based on the dimensions of the operands. The cost estimation methods are implemented as a weighted sum of all low-level operations used by a particular high-level operator, with the fixed weights estimated through our micro-benchmarks with default parameters (Table 2). Shechi then utilizes a first-choice hill-climbing local search heuristic to select the best aggregation strategy, finding the most efficient overall encoding strategy at each step through an exhaustive search (Step 4 - Optimization), as depicted in Algorithm 1 in Appendix A. We opted for the greedy heuristic approach since the problem of selecting the optimal aggregation and encoding strategy has not yet been solved, although related problems dealing with circuit evaluation have been shown to be computationally difficult [5, 12, 23, 60]. Without using the heuristic approach, an exhaustive search for both aggregation and encoding strategies would result in the overall complexity of $O(2^{|C_A|} \cdot 3^{|C_E|})$, where $|C_A|$ and $|C_E|$ are the sizes of aggregation and encoding candidate sets respectively, as there are 2 possibilities for each aggregation choice (left or right operand) and 3 possible encodings (row, column, or diagonal-wise). Using the hill climbing heuristic, this is reduced to $O(c \cdot |C_A| \cdot 3^{|C_E|})$ where the constant c is a predefined number of hill-climbing steps.

We note that initializing the tree with the above greedy approach often results in a nearly optimal solution in practice. In fact, it is often a good starting point for the search as in many tensor arithmetic workflows, such as PCA [33], the most computationally demanding parts (that are done on top of the largest tensors) are executed at the beginning of the workflow. Since the input data is in cleartext and often large-scale in the cross-silo setting, choosing the optimal encoding for this step alone (which is done by the greedy method in any case) already significantly impacts the overall runtime.

Maximizing efficient plaintext operations. Shechi's distributed setting allows it to distribute the workload among the parties and leverage efficient operations on local non-encrypted data. Plaintext computations can be orders of magnitude faster than operations on encrypted data and leveraging them in a distributed setting is a crucial optimization for large-scale computation, which is not supported by existing HE or SMC frameworks. To maximize the amount of such operations, Shechi reorders operations in the secure expression tree at compile time as follows. Encrypted and non-encrypted types are differentiated statically in Shechi thanks to Codon's strong type system. Shechi then maximizes the non-encrypted computation in consecutive associative operations, such as the series of element-wise multiplications, by iteratively pushing the operations that involve non-encrypted operands down towards the leaves of the secure expression tree to prioritize them over the operations involving encrypted data. As a result, the unnecessary encryption of data and invoking computation between the ciphertexts will be avoided. For example, when computing `X @ u.T @ u * 2` (Fig. 1), it is cheaper to first multiply 2 with `X` instead of `u` since the former is composed of non-encrypted partitions (Step 2 in Fig. 5).

Minimizing multiplication redundancy. In MHE, as in standard HE, multiplications are at least one order of magnitude slower than additions (Tab. 2). Shechi analyzes arithmetic expressions in the secure expression tree at compile time to minimize multiplication cost by identifying sums of the form $\sum_{i,j} t_i \cdot t_j$ and iteratively factoring out the common term t_k into $t_k \sum_l t_l + \sum_{i,j \neq k} t_i \cdot t_j$. Shechi also efficiently evaluates polynomial expressions with minimal complexity using the baby-step giant-step algorithm [34].

8.3 Optimizations at Runtime

Efficient matrix multiplication across encodings. To provide full flexibility in selecting the most efficient computation workflow, Shechi implements methods for all possible combinations of matrix encodings (i.e., row, column, and diagonal). These methods scale differently with input dimensions, requiring varying numbers of additions, multiplications, and rotations over ciphertexts [29, 40]. For instance, multiplying a row-wise encoded matrix of size $a \times b$ with another row-wise encoded matrix of size $b \times c$ requires $O(a \cdot c \cdot log_2(b))$ rotations, whereas a diagonally-encoded matrix requires $O(a\sqrt{b})$ rotations [29]. The output encodings also vary based on the input encodings. For example, in Fig. 6 (Step 1), if the matrix X is row-encoded, each element of $X @ u^\top$ (where u is a vector) is obtained in a separate ciphertext through the inner product of a row of X and $u^\top$ (i.e., the resulting column vector is row-encoded). Conversely, if X is column-encoded, $X @ u^\top$ can be obtained in a single ciphertext (i.e., a column-encoded column vector) by multiplying each column of X with a vector of the same size, composed of replicates of the corresponding element in $u^\top$ (obtained through masking, rotations, and additions), and then aggregating the result.

To capitalize on this variety of multiplication and encoding options, Shechi implements a generic method that determines the most efficient computation approach and encodings based on the input dimensions and states (i.e., encryption and any predefined encodings). This method also returns the cost of the preferred option and is used during the optimization phase (§.8.2; Step 4 and *Dynamic code generation* in Fig. 5; Fig. 6). In addition to previously established matrix encodings and matrix multiplications [29, 40], Shechi further expands the set of possible approaches by introducing a novel *cyclic-diagonal encoding* that is particularly suited for very tall or wide matrices by requiring a number of rotations that depends on the smaller dimension, see Appendix B.

Automatic instantiation of MHE. Shechi adopts default parameters ensuring 128-bit security for MHE (Appendix F), striking a balance across diverse applications. Shechi automates the execution of key instantiation protocols [56]. Each party independently samples secret key shares from predefined distributions, ensuring that the global secret key—defined as the sum of local shares—is never explicitly constructed. Consequently, decryption requires the collaboration of all parties. Other keys, such as the encryption key and evaluation keys required in MHE operations are publicly shared among all parties (Appendix F). Unlike existing HE solutions that are limited by the absence of a practical bootstrapping routine (§.5.2), thus requiring parameter selection to be tightly coupled with the specific circuit under evaluation (§.3), Shechi primarily bases this choice on balancing precision with performance. Shechi supports a cleartext execution of the protocol on local data, the results of which can facilitate analysis of the required precision and parameter selection.

Incorporating secret sharing-based SMC operations. Shechi uses HE operations to leverage efficient, parallelized local operations, including operations between local plaintext and encrypted shared data, and switches to secret sharing-based SMC to efficiently evaluate non-polynomial operations (e.g., comparison and division), which are difficult and often impractical to evaluate accurately in HE. Shechi identifies non-polynomial operations and invokes protocols to switch between CKKS ciphertexts and additive secret shares as needed [18, 29]. This process performs a blinded decryption to the secret-share domain, evaluates the function in this domain and allows the parties to obtain a fresh re-encryption of the result under the HE collective encryption key. We note that users have the option to restrict Shechi to distributed HE only (Appendix C).

Optimizing ciphertext maintenance. The optimal placement of rescaling, relinearization, and bootstrapping operations (§.5.2) is a difficult problem in general [5, 12, 23, 60]. For instance, optimal relinearization has been proven to be NP-hard [12, 23]. Due to the availability of efficient bootstrapping in MHE, placing these operations is an optimization feature rather than a limiting factor in centralized HE where the multiplicative depth and the number of rescaling operations are limited. Therefore, Shechi adopts a pragmatic strategy in which the ciphertext is relinearized immediately after each multiplication. Rescaling and bootstrapping are performed lazily only when necessary, i.e., before multiplication or addition with an operand of smaller scale. Shechi orchestrates these operations across parties through *counselling*—an automatic module that maintains the information of the ciphertext levels at each party and communicates it between parties with negligible network overhead (e.g., approximately 0.005% of the total bandwidth in our experiments).

9 Shechi's Integrated MHE Libraries

To enable full-stack optimizations, Shechi comprehensively implements and integrates both HE and SMC frameworks. As demonstrated in our evaluations (§.10), this allows Shechi to match the performance of existing HE and SMC solutions while supporting more complex distributed workflows than HE and providing better scalability than existing SMC tools. Notably, Shechi incorporates *Lattiseq*—a complete, optimized reimplementation of Lattigo's DCKKS (Distributed CKKS) scheme [27, 43, 56] in Codon [67]. In addition to preserving existing performance optimizations from Lattigo, such as fast matrix multiplication via number theoretic transform [16], Shechi adopts data-level parallelism (SIMD) in all operations and uses OpenMP threads to further parallelize operations on ciphertexts. To enable efficient SMC routines, such as bitwise operations and non-polynomial functions, Shechi incorporates additive secret sharing-based primitives from Sequre [68].

10 Performance Evaluation

We evaluate Shechi against existing cryptographic libraries and compilers for secure computing. First, we benchmark elementary operations, then assess performance on a set of basic applications with simple routines (e.g., Euclidean distance), and large-scale, complex data analysis workflows. Lastly, we provide scalability evaluation and an ablation study to demonstrate the impact of Shechi's optimizations.

10.1 Evaluation Settings

We simulated each party on a different machine with 12-core Intel i7-8700 CPUs (3.20GHz), 64 GB RAM, all connected via a LAN network with 1 Gb/s bandwidth and 0.5 ms latency. As this setup cannot handle the increased number of parties, the scalability and ablation studies were done on another machine with 192-core Intel Xeon Platinum 8260 CPU (2.40GHz) with 1 TB of RAM, simulating networking over UNIX sockets. For a fair comparison, we did not manually parallelize the tested workflows on top of the default parallelization in the underlying libraries. By default, we conduct all experiments with a minimal number of parties—a less favourable setup for distributed computing. The scalability benchmark shows how our solution's performance improves as the number of parties and input data dimensions increase.

10.2 Comparison with Existing Compilers

As there is currently no known MHE compiler to the best of our knowledge, we compare Shechi against cryptographic libraries and compilers that target similar general-purpose vector arithmetic tasks while offering the closest security guarantees: MP-SPDZ [41] and Sequre [68] for additive secret-share SMC; HEFactory [37] and EVA [23] for HE; and Lattigo [43, 64] and SEAL [13] libraries for low-level MHE and HE, respectively. We also provide comparison against MPyC [63] and HECO [71] (whose front-end is currently deprecated) in Appendix D.

10.2.1 Micro-Benchmarks

We demonstrate Shechi's low-level performance through micro-benchmarks in Table 2, evaluating basic algebraic operations on encrypted vectors using default parameters and a 128-bit security level for all tools (Appendix F).

Shechi's performance is generally comparable to the manually optimized and highly performant MHE primitives from Lattigo. Shechi's runtimes are also similar to single-party SEAL for low-level HE operations despite handling more complexity, such as bootstrapping and distributed switching between SMC and HE. Note that there is no clear winner overall among the evaluated tools. However, our goal was not to provide the fastest HE primitives in isolation but a set of primitives that together support efficient MHE operations. Future work includes continuous integration of improved primitives into our framework (§.11).

Shechi achieves comparable runtime to SMC computing operations. Additionally, Shechi allows computation to be done independently by computing parties over separate data shares in parallel, while SMC requires the parties' local data to be secret-shared and synchronized between all parties for computation. This enables a better workload distribution for some operations in Shechi (e.g., matrix multiplication), and better scaling with the number of parties, as demonstrated in §.10.2.2 and §.10.3.

10.2.2 Basic Workflows

We implemented two applications commonly used to benchmark HE compilers [11,23,71]—Euclidian L_2 distance calculation and matrix multiplication. L_2 distances are computed between 32 encrypted vectors of length 8192, while matrix multiplications are performed on matrices of size 128×8192. The results are shown in Fig. 7.

HE implementations use the recommended default parameters (Appendix F). MP-SPDZ, Lattigo, Sequre and Shechi implementations used available built-in methods for matrix multiplications, whereas we relied on a series of elementary vector operations (additions, multiplications, and rotations) to implement the standard version of matrix multiplication in other solutions that do not natively support these more complex, higher level operations. In HE solutions, the input data is encrypted by each party and transferred to a single computing party for computation since these solutions do not offer support for distributed computing. In SMC, the data is secret-shared among two computing parties, while distributed approaches, such as Lattigo and Shechi, split the data evenly and horizontally among two parties (each party holding half of the rows of each input matrix in a non-encrypted form). We note that this two-party scenario is the least favourable to Shechi compared to settings with more than two parties. As shown in Fig. 7, even in this scenario that is not aligned with its primary focus, Shechi's runtime outperforms HE-based alternatives across all applications. Due to the small scale of the input data, Shechi is slightly slower than the pure SMC solutions for computing L_2 distance. However, it is faster than MP-SPDZ when computing A `@` $B^\top$, and faster than both Sequre and MP-SPDZ when computing $A^\top$ `@` B even on small dimensions since, in this case, matrix multiplication can be performed locally on non-encrypted data before aggregating the results, efficiently leveraging distributed computing. As expected, Shechi's communication overhead is higher than the centralized and SMC approaches in these simple applications due to the expansion factor of MHE (a ciphertext encrypting 8192 values has a size of 2.6 MB with our default parameters) and low benefits from distributed computing in these scenarios. Note that these benefits quickly overcompensate the MHE expansion in other applications, including $A^\top$ `@` B and the large-scale benchmarks below. Finally, we note that Shechi's simple syntax allows users to write these simple applications in less than 4 lines of code.

10.2.3 Large-Scale and Complex Workflows

To demonstrate the practical usability, performance and versatility of Shechi, we implemented secure distributed equivalents of three complete applications that operate on large datasets: principal component analysis (`pca`), kinship estimation (`kinship`) [52], and genome-wide association study (`gwas`) [18]. The last two applications are from the domain of computational genomics and serve as a good example for Shechi's utility, as genomics data is both extremely sensitive with respect to privacy and scattered across multiple entities reluctant to share their data. We use a lung cancer study dataset [59], which contains patients' phenotypic information (e.g., age and sex), genotype data (i.e., vectors with the values 0, 1 and 2 for each variant), and a binary value indicating the presence of lung cancer. In this dataset, the input matrix has more than 600,000 variants (features) and 9,000 patients (samples). We compared Shechi's implementations against the existing state-of-the-art implementations that were originally done in Lattigo and Sequre. We note that neither of these complex workflows can be easily implemented with the other HE and SMC compilers mainly due to their low-level nature, as well as the scale of the input data and required computational depth. We refer to Appendix F for the detailed description of these workflows.

Shechi achieves on-par or better performance as existing manually-optimized secure solutions [18,29] while improving expressiveness by up to two orders of magnitude. In Fig. 7, we notably observe a $15\times$ speed improvement in `kinship` and $6\times$ network traffic reduction in `pca` experiments when compared against the manually optimized version in Lattigo. This is notably due to Shechi's encoding optimization (§.8.2), which automatically selects a more efficient encoding strategy than the one manually selected and employed in the existing Lattigo implementation. Our `kinship` implementation has only 4 lines of code, while the Lattigo equivalent has more than 160 lines of code as the user has to coordinate the computation across parties manually. The Lattigo version of `pca` was manually optimized to maximize the plaintext over ciphertext usage and minimize the matrix multiplications' cost. Shechi implements the same procedure as simple pseudocode in 10

Solution / Runtime [ms]	`encrypt` or `secret_share`	`add_const`	`add`	`mult_const`	`mult`	`rotate`	`decrypt` or `reveal`	`bootstrap`	`to_smc`	`to_mhe`
SEAL (HE)	15.9	**0.2**	0.4	4.3	26.9	21.6	**6.3**	–	–	–
Sequre (SMC)	**1.16**	0.77	1.28	1.59	**14.35**	**0.11**	7.72	–	–	–
MP-SPDZ (SMC)	3.46	2.5	2.5	30.00	71.07	**0.11**	8.00	–	–	–
Lattigo (MHE)	21.25	0.4	0.4	4.6	28.1	24.0	119	184	366	185
Shechi (MHE)	6.5	0.4	0.4	**0.64**	40.6	45.3	61.38	**94.8**	**103.8**	**65.2**

Table 2: Low-level primitives micro-benchmark. All operations are applied on top of 8192-element, encrypted vectors using parameters that ensure a 128-bit security level. `add_const` and `mult_const` refer to addition and multiplication with a public constant value. `secret_share` and `reveal` are SMC-only operations that correspond to encryption and decryption, respectively. Decryption is done locally by SEAL, and distributively in the other solutions. Dash (–) stands for "not applicable".

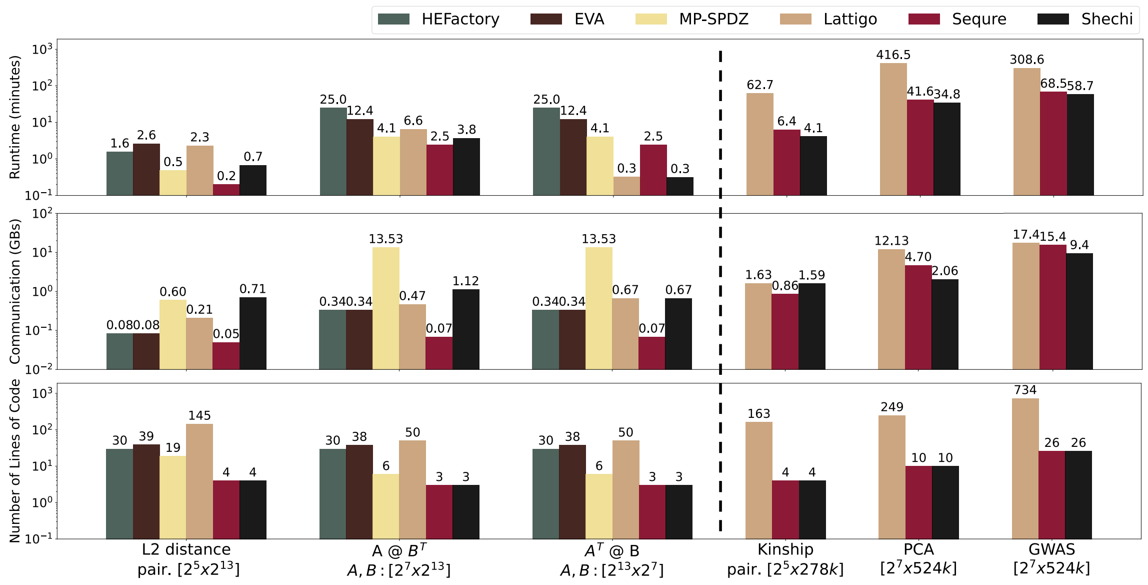


Figure 7: Runtime, communication and expressiveness comparison between Shechi and existing approaches. For distributed approaches the input data are evenly and horizontally split among 2 parties. Communication is measured as the maximum number of bytes any party sends. Expressiveness is measured as the number of lines of code without comments, blank lines, and debug statements.

lines of code—25× fewer than Lattigo implementation—and manages to automatically find the same optimization opportunities and even detect new optimization hotspots that the developers of the original pipeline missed. Similar performance improvements were observed for `gwas` that has 26 lines of code, compared to 734 lines found in Lattigo. Here, Shechi also relies on its own diagonal encoding (§.8) to encode and optimize computation on top of wide matrices that are inherently abundant in this application.

10.3 Scaling

We show in Fig. 8 that Shechi's runtime and communication costs increase linearly with the number of parties and samples and scale better than an SMC-only solution (Sequre). For this experiment, we ran PCA with a fixed number of 64 samples per party. In an 8-party setup, Shechi's runtime and communication are more than four and seven times smaller than Sequre's, respectively. This difference is expected to grow further with more parties since, in Shechi, the workload is distributed among the parties, and the communication overhead is mainly due to aggregating the intermediate results between the parties and collective bootstrapping. In Sequre, and SMC in general, however, communication increases faster with the number of parties than with Shechi because all computations require interactions among all parties. Fig. 12 in Appendix E illustrates how local operations benefit MHE for the common case of matrix multiplication, involving input matrices distributed among multiple parties. Lastly, we note that if a fixed number of samples is distributed among a larger number of parties, e.g., if the 128 samples that are split among 2 parties are instead split among 8 parties, Shechi's runtime decreases from 40 to 15 minutes, further showing its ability to effectively distribute the workload.

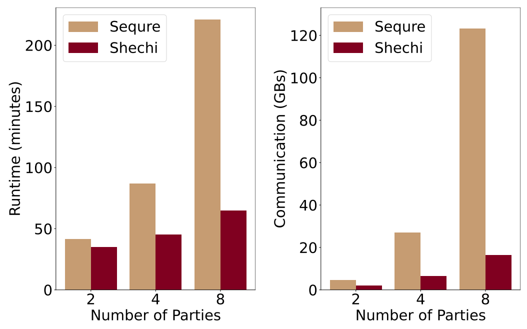


Figure 8: Shechi's scaling with the number of parties and samples. In this experiment, PCA is executed for the increasing number of parties and individuals, with a fixed number of 64 samples per party.

10.4 Necessity of Shechi's Optimizations

Shechi applies a set of compile-time and runtime optimizations to translate standard code into efficient distributed executions. Here, we demonstrate the impact of its encoding optimizations, which selects the most efficient matrix encoding at each step of the algorithm (§.8), a crucial segment of MHE operations. Fig. 9 (left) shows the performance gain achieved using Shechi's optimization for executing PCA on the lung cancer dataset with an increasing number of samples. The term *Shechi (no opt.)* refers to a typical use-case where the developer manually selects some encoding for each matrix multiplication (row-encoding for wide matrices and column-encoding for tall matrices in this use-case). Note that another common use case, where the developer relies on a single encoding, yields impractical runtimes for this application. Shechi automatically chooses from a set of three encodings and various matrix multiplication methods, both of which are easily extensible. We note that Shechi's searching for the most efficient aggregation and encoding strategy takes less than one second in our experiments and is negligible compared to the overall runtime. Fig. 9 (right) further demonstrates the importance of our optimizations, even for computing a simple expression over four matrices (`A`–`D`) of shapes 32×8192 for `A` and `B`, and 32×32 for `C` and `D`.

10.5 Neural Networks Library

The neural networks module of Shechi based on the Keras API [19] enables users to implement feed-forward neural networks in as few as 10 lines of code by simply defining layers and calling the desired fitting methods, as shown in Fig. 10. Shechi was able to perform training of privacy-preserving credit score evaluation [73] for approximately 100K individuals, split among 2 parties, with 16 features in a practical runtime of 7 hours. Using a similar codebase, we also evaluated drug-target interaction inference [38] over 150 individuals with 8192 features (one-hot encoded classes of proteins and

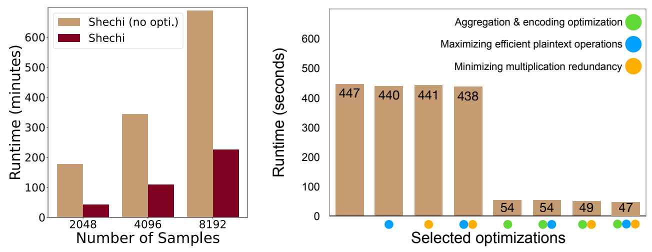


Figure 9: Shechi's runtime worsens up to 4× with one of the main optimizations turned off. PCA runtimes with and without aggregation and encoding optimization with the increasing number of samples (left), and the impact of all optimizations on evaluating an example expression `A @ B.T * (D + C) * 2` (right). Aggregation and encoding strategies are critical for the performance of MHE applications.

compounds) within the same time span. We trained a multinomial classification model (a single-layer neural network with softmax activations and cross-entropy loss) on the MNIST dataset [46], which consists of 58,000 images split among 2 parties (50,000 for training and 8,000 for inference). Both the training and inference steps were implemented in 5 lines of code (Appendix G), completed in 1.2 hours (1h for training and 0.2h for inference), and achieved an accuracy of 85%, which is comparable to cleartext training. Although these results demonstrate an effective use of our framework for relatively lightweight neural networks, we acknowledge that a more general application to deep networks remains an open problem and will require further advances in cryptographic techniques to be incorporated into our MHE framework.

```
@shechi
def credit(mhe, X, y, test_X, n_neurons, epochs):
    layers = (
        Input[type(X)](X.shape[0]),
        Dense[type(X)]("relu", n_neurons),
        Dense[type(X)]("linear", 1))
    model = Sequential(layers).compile(mhe,
        loss="hinge", optimizer="mbgd")
    model.fit(mhe, X=X, y=y, epochs=epochs)
    return model.predict(mhe, test_X).reveal()
```

Figure 10: Shechi's Keras-like neural networks interface enables simple implementation of network training and inference for privacy-preserving credit score evaluation.

10.6 Accuracy, Network, and Memory Usage

The security of CKKS, the HE scheme used in Shechi (§.5.2), requires some noise to be added directly in the least significant bits of the encrypted values. Shechi implements the same standard noise management methods as existing HE works [21, 47]. In Fig. 11, we illustrate that Shechi is accu-

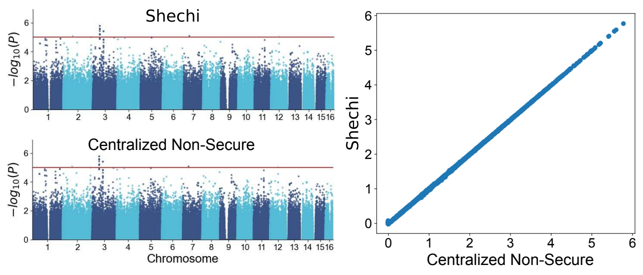


Figure 11: Shechi's secure distributed execution of GWAS produces results comparable to centralized non-secure execution, yielding similar association statistics, represented as the negative logarithm base-10 of the p-values.

rate and achieves the same results as a non-secure centralized equivalent for a GWAS study on 128 patients (samples) sampled from the lung cancer study dataset and evenly split between two parties. Each patient has $500,000$ variants (features). We observe that Shechi obtains similar results (i.e., association statistics) to a centralized non-secure execution, with a mean absolute error of 3.9×10^{-5}.

To assess the effect of network delay, we reproduced all experiments in a coast-to-coast network setup. We used Amazon AWS to deploy machines that match the clock speed and RAM of our LAN machines across the US. We observe that while the communication delay increases by a factor of at least $20\times$ and up to $108\times$ (with delays between 12 and 62 ms) when compared to our LAN setting, Shechi runtime increases only up to $2.2\times$ times. For example, Kinship runtime increased from 4 to 8 minutes, PCA from 35 to 76 minutes, and GWAS from 59 to 110 minutes. Lastly, we note that Shechi memory consumption was also better than the competition, with maximum resident set size being within the range of 15 GBs for Shechi, 25 GBs for Sequre, and more than 100 GBs for Lattigo per party. This means that one can use cheaper instances with Shechi to run the analysis.

11 Limitations and Future Work

Noise and Precision Management with CKKS. Shechi uses an approximate fully homomorphic encryption scheme (i.e. CKKS) where decrypted values contain some noise. The noise accumulates throughout computation, impacting the accuracy of complex workflows. CKKS guarantees moderate growth of relative noise (linear with respect to the number of operations) [14] and Shechi implements the same standard noise management methods as existing HE works [21,47]. However, users still need to account for precision in specific applications, e.g., with input data containing values smaller than the noise overhead. The required precision can be observed by users through local simulations (§.8.3).

Extended Support for Python Semantics. Shechi's static compiler backend, Codon, disallows certain dynamic features of Python, such as collections of heterogeneous types. Each tensor, either encrypted or not, must have a uniform type (boolean, integer, or floating-point). Furthermore, the syntax for branching (`if` statement) over encrypted values is intentionally disallowed in Shechi to differentiate it from branching over non-encrypted data, which is allowed. At this time, users should use the provided interface for masking or secure demultiplexer to simulate branching on top of encrypted data. Supporting standard syntax for secure branching is possible and will be explored in future work.

Alternative Secure Computation Primitives. Shechi supports full-stack optimization by reimplementing low-level HE and SMC primitives (§.9). However, its modular architecture allows seamless integration of other primitives, such as the multiparty instantiation of the exact BFV scheme [56] or improvements of SMC subroutines [51, 75], either through reimplementation or by leveraging the existing libraries.

Cryptographic Parameter Choice via Code Analysis. Unlike existing HE-based solutions that adjust parameters to circuit depth for practicality (due to the absence of practical bootstrapping), Shechi can optimize parameter selection primarily for performance (e.g., minimizing distributed bootstrapping operations that require inter-node communication). While Shechi's default parameters achieve a balanced tradeoff for diverse applications, future work includes integrating automatic HE parameter selection through static code analysis similar to other optimizations demonstrated in this work.

Enhancing Coordination Across Machines. Shechi facilitates collective operations through a *counselling* module, which automatically coordinates the exchange of necessary information for collaborative computations among parties. Future work will incorporate advanced distributed computing techniques, such as remote procedure calls [57], to further streamline and optimize this process (§.8.3).

12 Conclusion

Shechi is the first compiler that combines secure multi-party computation and fully homomorphic encryption to enable high-performance secure computing without sacrificing readability and maintainability. It achieves similar or enhanced performance compared to that of existing secure compilers, scales better with the number of parties and data dimensions, and unlocks more complex distributed workflows not supported by previous tools. Our systematic, multi-step approach reveals novel optimizations that domain experts may overlook. In addition, Shechi greatly simplifies the code for real-world applications and facilitates secure and efficient programming of distributed algorithms, empowering non-experts to develop effective data analysis tools. Thus, our work has the potential to promote the adoption of secure computation techniques and allow users in various domains to conduct collaborative studies that would otherwise be impossible or impractical due to privacy concerns.

13 Funding

This work was supported by the NSERC Discovery Grant (RGPIN-04973), the Canada Research Chairs Program, Canada Foundation for Innovation's JELF and B.C. Knowledge Development Fund (to I.N.), RM1 HG011558 and DP5 OD029574 (to H.C.), and NIH R01HG010959 (to B.B.).

14 Ethics Considerations

Our approach helps minimize risks to human subjects related to privacy breaches when datasets contain sensitive personal information. For system demonstration, we used only publicly accessible data under access control [59] in the dbGaP repository [69] (accession: phs000716.v1.p1).

15 Open Science

Our work facilitates the analysis of distributed datasets while ensuring data confidentiality and security, thus promoting open science and collaboration. To enable other researchers to build upon our work, we made our entire source code open source and publicly available at https://zenodo.org/records/14725520.

References

[1] PALISADE repository (v1.11.9-dev). https://gitlab.com/palisade/palisade-development/-/tags/v1.11.9-dev. (01.2025).

[2] Ehud Aharoni et al. HeLayers: A Tile Tensors Framework for Large Neural Networks on Encrypted Data. PETs 2023, 2023.

[3] E Anderson et al. Generalized qr factorization and its applications. Linear Algebra and its Applications, 162:243–271, 1992.

[4] Donald Beaver. Efficient multiparty protocols using circuit randomization. CRYPTO '91, page 420–432, Berlin, Heidelberg, 1991. Springer-Verlag.

[5] Fabrice Benhamouda et al. Optimization of bootstrapping in circuits. pages 2423–2433. SIAM, 2017.

[6] Marcelo Blatt et al. Secure large-scale genome-wide association studies using homomorphic encryption. Proceedings of the National Academy of Sciences, 117(21):11608–11613, 2020.

[7] Fabian Boemer et al. ngraph-he: a graph compiler for deep learning on homomorphically encrypted data. In Proceedings of the 16th ACM international conference on computing frontiers, pages 3–13, 2019.

[8] Fabian Boemer et al. ngraph-he2: A high-throughput framework for neural network inference on encrypted data. In Proceedings of the 7th ACM workshop on encrypted computing & applied homomorphic cryptography, pages 45–56, 2019.

[9] Fabian Boemer et al. Mp2ml: A mixed-protocol machine learning framework for private inference. In Proceedings of the 15th international conference on availability, reliability and security, pages 1–10, 2020.

[10] Zvika Brakerski et al. (leveled) fully homomorphic encryption without bootstrapping. ACM TOCT, 6(3):1–36, 2014.

[11] José Cabrero-Holgueras and Sergio Pastrana. Hefactory: A symbolic execution compiler for privacy-preserving deep learning with homomorphic encryption. SoftwareX, 22:101396, 2023.

[12] Hao Chen. Optimizing relinearization in circuits for homomorphic encryption. arXiv preprint arXiv:1711.06319, 2017.

[13] Hao Chen et al. Simple encrypted arithmetic library - seal v2.1. Cryptology ePrint Archive, Paper 2017/224, 2017. https://eprint.iacr.org/2017/224.

[14] Jung Hee Cheon et al. Homomorphic encryption for arithmetic of approximate numbers. In ASIACRYPT, 2017.

[15] Jung Hee Cheon et al. Remark on the security of CKKS scheme in practice. Cryptology ePrint Archive, Paper 2020/1581, 2020.

[16] Jung Hee others Cheon. A full rns variant of approximate homomorphic encryption. In SAC 2018, pages 347–368. Springer, 2019.

[17] Hyunghoon Cho et al. Secure Genome-Wide Association Analysis using Multiparty Computation. Nature biotechnology, 36(6):547–551, 2018.

[18] Hyunghoon Cho et al. Secure and Federated Genome-Wide Association Studies for Biobank-Scale Datasets. bioRxiv, pages 2022–11, 2022. [doi:10.1101/2022.11.30.518537].

[19] François Chollet et al. Keras. https://keras.io, 2015.

[20] Sangeeta Chowdhary et al. Eva improved: Compiler and extension library for ckks. In Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography, pages 43–55, 2021.

[21] Anamaria Costache et al. On the Precision Loss in Approximate HE. ePrint, 2022.

[22] Roshan Dathathri et al. Chet: an optimizing compiler for fully-homomorphic neural-network inferencing. In ACM SIGPLAN, pages 142–156, 2019.

[23] Roshan Dathathri et al. Eva: An encrypted vector arithmetic language and compiler for efficient homomorphic computation. In ACM SIGPLAN, pages 546–561, 2020.

[24] Leo de Castro et al. Fast vector oblivious linear evaluation from ring learning with errors. Cryptology ePrint Archive, Paper 2020/685, 2020.

[25] Flower A Friendly Federated Learning Framework. https://flower.ai/, (accessed: March 2024).

[26] David Froelicher et al. Unlynx: a decentralized system for privacy-conscious data sharing. PoPETS, 2017(4):232–250, 2017.

[27] David Froelicher et al. Scalable privacy-preserving distributed learning. In PoPETs, volume 2, pages 323–347, 2021.

[28] David Froelicher et al. Truly privacy-preserving federated analytics for precision medicine with multiparty homomorphic encryption. Nature communications, 12(1):1–10, 2021.

[29] David Froelicher et al. Scalable and privacy-preserving federated principal component analysis. In IEEE S&P, pages 888–905, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[30] Amadou Gaye et al. DataSHIELD: Taking the Analysis to the Data, not the Data to the Analysis. International journal of epidemiology, 43(6):1929–1944, 2014.

[31] The EU General Data Protection Regulation. https://eugdpr.org/, (accessed: October 2023).

[32] Charles Gouert et al. Sok: New insights into fully homomorphic encryption libraries via standardized benchmarks. Proceedings on PETs, 2023.

[33] Nathan Halko et al. Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions. SIAM review, 53(2):217–288, 2011.

[34] Kyoohyung Han and Dohyeong Ki. Better bootstrapping for approximate homomorphic encryption. In CT-RSA, 2020.

[35] Charles R. Harris et al. Array programming with NumPy. Nature, 585(7825):357–362, September 2020.

[36] Marcella Hastings et al. Sok: General purpose compilers for secure multi-party computation. In IEEE S&P, pages 1220–1237. IEEE, 2019.

[37] HEFactory Prototype. https://github.com/jcabrero/HEFactory, (accessed: April 2024).

[38] Brian Hie et al. Realizing private and practical pharmacological collaboration. Science, 362(6412):347–350, 2018.

[39] Health Insurance Portability and Accountability Act of 1996 (HIPAA). https://www.hhs.gov/hipaa/index.html, (accessed: October 2023).

[40] Xiaoqian Jiang et al. Secure Outsourced Matrix Computation and Application to Neural Networks. In ACM SIGSAC CCS, pages 1209–1222, 2018.

[41] Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. In ACM SIGSAC, pages 1575–1590, 2020.

[42] Can Kockan et al. Sketching algorithms for genomic data analysis and querying in a secure enclave. Nature methods, 17(3):295–301, 2020.

[43] Lattigo: A Library For Lattice-Based Homomorphic Encryption in Go. https://github.com/tuneinsight/lattigo, (accessed: July 2023).

[44] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In CGO, pages 75—–88, 2004.

[45] Chris Lattner et al. MLIR: Scaling compiler infrastructure for domain specific computation. In CGO, pages 2–14, 2021.

[46] Yann LeCun and Corinna Cortes. MNIST Handwritten Digit Database. http://yann.lecun.com/exdb/mnist/, 2010.

[47] Yongwoo Lee et al. High-Precision Bootstrapping for Approximate Homomorphic Encryption by Error Variance Minimization. In Eurocrypt, 2022.

[48] Baiyu Li et al. Securing approximate homomorphic encryption using differential privacy. In Yevgeniy Dodis and Thomas Shrimpton, editors, CRYPTO, pages 560–589, 2022.

[49] Baiyu Li and Daniele Micciancio. On the security of homomorphic encryption on approximate numbers. Cryptology ePrint Archive, Paper 2020/1533, 2020.

[50] Vadim Lyubashevsky et al. On Ideal Lattices and Learning with Errors over Rings. In EUROCRYPT, 2010.

[51] Eleftheria Makri et al. Rabbit: Efficient comparison for secure multi-party computation. In International Conference on Financial Cryptography and Data Security, pages 249–270. Springer, 2021.

[52] Ani Manichaikul et al. Robust relationship inference in genome-wide association studies. Bioinformatics, 26(22):2867–2873, 10 2010.

[53] Luca Melis et al. Exploiting unintended feature leakage in collaborative learning. In IEEE S&P, pages 691–706, 2019.

[54] Payman Mohassel and Yupeng Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In IEEE S&P, 2017.

[55] Arturo Moncada-Torres et al. VANTAGE6: an open source priVAcy preserviNg federaTed leArninG infrastructurE for Secure Insight eXchange. In AMIA Annual Symposium Proceedings, pages 870–877, 2020.

[56] Christian Mouchet et al. Multiparty homomorphic encryption from ring-learning-with-errors. In PoPETs, 2021.

[57] Bruce Jay Nelson. Remote procedure call. PhD thesis, USA, 1981. AAI8204168.

[58] NVIDIA FLARE (NVIDIA Federated Learning Application Runtime Environment. https://developer.nvidia.com/flare, (accessed: March 2024).

[59] Qing et al. Genome-wide association analysis identifies new lung cancer susceptibility loci in never-smoking women in Asia. Nature Genetics, 44(12):1330–1335, 2012.

[60] Hiroki Sato et al. Poster: Loop circuit optimization with bootstrapping over fully homomorphic encryption. In IEEE S&P, 2017.

[61] Sinem Sav et al. POSEIDON: Privacy-Preserving Federated Neural Network Learning. NDSS, 2021.

[62] Berry Schoenmakers. MPyC—Python package for secure multiparty computation. In Workshop on the Theory and Practice of MPC. https://github.com/lschoe/mpyc, 2018.

[63] Berry Schoenmakers. Mpyc—python package for secure multiparty computation. In Workshop on the Theory and Practice of MPC. https://github.com/lschoe/mpyc, 2018.

[64] MHE Cryptographic Library. https://github.com/hhcho/sfgwas, (accessed: April 2024).

[65] Ariya Shajii et al. Seq: a high-performance language for bioinformatics. 3(OOPSLA), oct 2019.

[66] Ariya Shajii et al. A Python-based programming language for high-performance computational genomics. Nature Biotechnology, 39(9):1062–1064, 2021.

[67] Ariya Shajii et al. Codon: A compiler for high-performance pythonic applications and dsls. In ACM SIGPLAN CC, page 191–202, 2023.

[68] Haris Smajlović et al. Sequre: a high-performance framework for secure multiparty computation enables biomedical data sharing. Genome Biology, 24(1):5, 2023.

[69] Kimberly A Tryka et al. Ncbi's database of genotypes and phenotypes: dbgap. Nucleic acids research, 42(D1):D975–D979, 2014.

[70] Alexander Viand et al. Sok: Fully homomorphic encryption compilers. IEEE S&P, 2021-May:1092–1108, 1 2021.

[71] Alexander Viand et al. {HECO}: Fully homomorphic encryption compiler. In USENIX Security, pages 4715–4732, 2023.

[72] Zhiyu Wan et al. Sociotechnical safeguards for genomic data privacy. Nature Reviews Genetics, pages 1–17, 2022.

[73] I-Cheng Yeh and Che hui Lien. The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients. Expert Systems with Applications, 36(2, Part 1):2473–2480, 2009.

[74] Wenting Zheng et al. Helen: Maliciously Secure Coopetitive Learning for Linear Models. In IEEE S&P, 2019.

[75] Lijing Zhou et al. Bicoptor: Two-round secure three-party non-linear computation without preprocessing for privacy-preserving machine learning. In IEEE S&P, 2023.

[76] Ligeng Zhu et al. Deep leakage from gradients. In NeurIPS, volume 32, 2019.

A Aggregation and Encoding Optimization

We describe here the first-choice hill-climbing local search heuristic that Shechi implements to select the best aggregation strategy while finding the most efficient overall encoding strategy at each step through an exhaustive search.

Algorithm 1 Aggregation and Encoding Optimization

```
Require: $tree$, $aggr\_candidates$, $enc\_candidates$: expr. tree and candidate
    nodes for aggregation and encoding; $c$: hill climbing limit
Output: updated $tree$
 1: $cur\_cost \leftarrow tree.cost()$ //estimated cost of tree eval.
 2: for $i = 0$ to $c$ do
 3:     $tree\_modified \leftarrow$ False
 4:     for $ni \in agg\_candidate$ do
 5:         $new\_tree \leftarrow tree.flip\_aggr(ni)$ //change aggr.
 6:         $new\_tree.resolve\_tree()$ //update tree complement
 7:         $min\_enc \leftarrow new\_tree.encoding$
 8:         $min\_cost \leftarrow new\_tree.cost()$
 9:         for $j = 0$ to $3^{|enc\_candidates|} - 1$ do
10:             $new\_tree.set\_encoding(j)$
11:             $bet\_cost \leftarrow new\_tree.cost()$
12:             if $bet\_cost < min\_cost$ then
13:                 $min\_cost \leftarrow bet\_cost$
14:                 $min\_encoding \leftarrow new\_tree.encoding$
15:             end if
16:         end for
17:         if $min\_cost < cur\_cost$ then
18:             $tree\_modified \leftarrow$ True
19:             $cur\_cost \leftarrow min\_cost$
20:             $tree \leftarrow new\_tree$
21:             $tree.set\_encoding(min\_encoding)$
22:             $tree.resolve\_tree()$
23:         end if
24:     end for
25:     if not $tree\_modified$ then
26:         break //reached local minimum
27:     end if
28: end for
```

B Novel Matrix Multiplication Method

In addition to previously established matrix encodings and matrix multiplications [29,40], Shechi further expands the set of possible approaches by introducing a novel *cyclic-diagonal encoding* that is particularly suited for very tall or wide matrices by requiring a number of rotations depending on the smaller dimension, where the i-th diagonal d_i of a matrix $A \in \mathbb{R}^{a \times b}$ is obtained as $d_i[j] = A[(i+j) \bmod m, j \bmod n]$, where $0 \le i \le \min(a,b)$ and $0 \le j \le \max(a,b)$, instead of $d_i[j] = A[(i+j) \bmod a, j]$ with $0 \le i \le a$ and $0 \le j \le b$, in standard diagonal encoding. Multiplying a row-wise encoded matrix by a matrix with this encoding requires $O(a \cdot \min(b,c) + log_2 \max(a,b))$ rotations and $O(a \cdot \min(b,c))$ multiplications, or even just $O(\min(a,b) \cdot \min(b,c))$ rotations and multiplications if both operands are cyclic-diagonal encoded. Note that the former is only lower than the existing diagonally-encoded matrix multiplication when $\min(b,c) < \sqrt{b}$. This method is particularly useful for highly asymmetric matrices since the complexity of matrix multiplication between two cyclic-diagonally-encoded matrices is determined only by the smaller dimensions of both matrices. It also ensures the shape of the encoded $a \times b$ matrix is always $\min(a,b) \times \max(a,b)$, consistently exposing data-level parallelism over the larger matrix dimension.

Solution / Runtime [ms]	`encr`	`add_const`	`add`	`mult_const`	`mult`	`rotate`	`decr`
MPyC (SMC)	3.02	2.86	4.98	$2.7 \cdot 10^4$	$2.8 \cdot 10^4$	1.59	4.83
HECO (HE)	16.92	0.36	0.79	4.2	25.85	42.17	6.3
Shechi (MHE)	6.5	0.4	0.4	0.64	40.6	45.3	61.38

Table 3: Low-level primitives micro-benchmark for MPyC and HECO. Each operation ensures a 128-bit security level. HECO's front-end is currently deprecated, and it does not support practical bootstrapping.

C Stronger Threat Model

To effectively evaluate non-polynomial functions (e.g., comparisons) that are not natively supported in HE, Shechi automatically transitions to an SMC secret-sharing scheme that uses a trusted dealer for efficiency purposes (§5.2). For users who prefer not to rely on a trusted dealer, Shechi offers the option to use only HE operations and bootstrapping. In this case, Shechi approximates non-linear functions via polynomial interpolation [27], using an interval and degree parametrized by the user. Previous works [27,29] have shown that the accuracy and time loss can be minimized depending on the polynomial degree and interval required. However, parametrizing these factors becomes challenging in complex applications, especially as users lack prior access to the complete dataset. To assist users, Shechi enables local simulation of the application's execution in standard Python environments by modifying just a single line of code. This feature also enables users to fine-tune other parameters, such as precision, when the default settings are not suited to their specific application.

D Comparison Against Other Frameworks

We also compared against MPyC [62] and HECO [71]. MPyC is an easy-to-use Python library for SMC based on *t-out-of-n* Shamir's secret sharing scheme with $t < \lfloor \frac{n}{2} \rfloor$. Its security setup significantly differs from Shechi's and the other SMC frameworks' in this paper that use a generally stronger *n-out-of-n* additive secret sharing scheme. We benchmarked MPyC in the smallest possible secure setup with four computing parties and, at most, one passively corrupt party (i.e. any two out of four parties can collude together to reconstruct the secret). While it is on par in some operations, MPyC's performance is generally, by order of magnitude, worse than Shechi's, due to the performance overhead inherited from its host language and underlying security scheme (Table 3).

HECO, on the other hand, is an HE compiler that translates high-level Pythonic code to high-performance executables through MLIR [45] and compile-time optimizations. HECO, however, deprecated their frontend language and enabled writing their programs directly in MLIR by calling the low-level HE primitives (add, mul, and rotate). We managed to implement microbenchmarks using their existing MLIR dialects (Table 3)), and we plan to extend and test it on larger benchmarks after their high-level operations are enabled.

E Scalability of MHE

The workload is distributed across all parties in MHE depending on the size of local datasets, while secret sharing-based SMC requires all parties to operate over the combined secret-shared dataset, which scales poorly as the size of the dataset and the number of parties grow (Fig. 12). As a result, compared to the two-party setting analyzed in some of our experiments, settings with more than two parties are expected to lead to a greater improvement of MHE over SMC protocols.

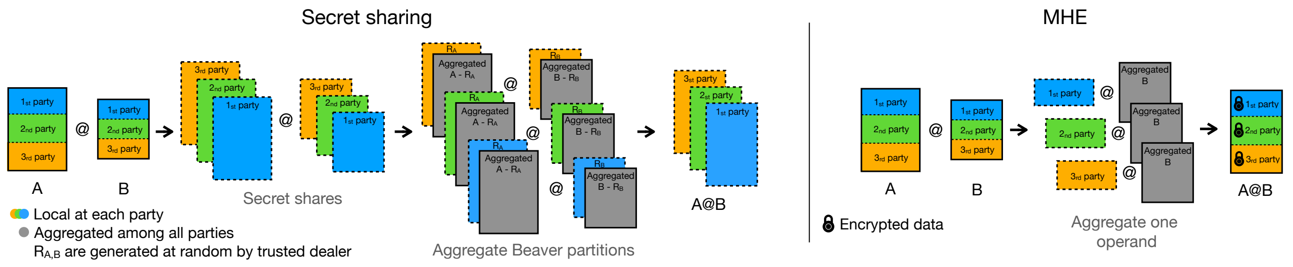
F Cryptographic Details & Parameters

We used parameters that ensure 128-bit security level across all frameworks. For HE, we used a standard parameter setting (from the Lattigo library [43]) to instantiate CKKS, with a 438-bit modulus and 10 levels (moduli chain: {35184372121601, 17179967489, 17179672577, 17180262401, 17180295169, 17179410433, 17180393473, 17180557313, 17180950529, 17178525697}), supporting 8192 plaintext slots with a default scale of 2^{34}, and standard deviations of 3.2 and 2^{20} for the encryption noise and smudging noise, respectively. Secret key shares are drawn at random from $\mathbb{Z}_3[x]/(X^{2^{14}}+1)$ at each party. The sum of secret key shares represents a collective secret key but is never computed nor revealed, i.e., decryption requires the participation of all parties. The generation of other keys, such as the public encryption key, leverage shared, private pseudorandom streams that are agreed upon by all parties beforehand [56]. These pseudorandom streams are also used for secret-sharing to boost performance [17, 18]. Additionally, in HE, each rotation value requires a different rotation key. There are multiple strategies to solve this issue: (i) generate a predefined number of rotation keys, (ii) generate all required keys based on runtime analysis of the expression tree, or (iii) generate all power-of-two rotation keys up to the maximum ciphertext size and translate all rotations to their bitwise equivalents. Shechi uses the combination of the first and third strategies.

We instantiated the SMC solutions on top of a 256-bit finite field (MPyC and Sequre) or $\mathbb{Z}_{2^{256}}$ ring (MP-SPDZ) with a 32-bit fraction in 64-bit fixed-point precision and 64-bit of additional statistical security padding. Since MP-SPDZ offers a multitude of SMC schemes, we opted for the *dealer-ring* scheme, which is based on additive secret sharing with a trusted dealer, who generates and distributes the Beaver multiplication triples to the parties, and thus is comparable to Sequre and Shechi's SMC module. We attribute the slightly higher runtimes of MP-SPDZ, compared to Sequre, to the better performance of the low-level executables generated by the custom end-to-end compiler with the LLVM backend in Sequre. The runtime improvements in other solutions are due to HE's superior performance in these microbenchmarks. MPyC's security scheme, on the other hand, is not configurable and is fixed to *t-out-of-n* Shamir's secret sharing (i.e. t participants out of n can collude to reveal the secret) with a $0 \leq t \leq \lfloor \frac{n}{2} \rfloor$ constraint. In other words, the smallest number of parties that still offer secure computing is $n = 4$ with $t = 2$.

Mitigating Privacy Leakage in Noise via Smudging. CKKS requires additional noise to be added to ciphertexts before decryption to prevent the recently exposed vulnerability [49], in which sensitive information is extracted from the noise in the decryption results. This vulnerability is present in the original collective decryption protocol implemented in Shechi (Protocol 3 in [56]). To address this, we incorporated the countermeasures adopted by CKKS implementations in popular libraries such as PALISADE and Lattigo [15, 24, 56]. The general approach involves adding 2^s bits of *smudging* noise for statistical security before the decryption results are revealed. However, given the performance constraints of current CKKS implementations, which provide up to 64-bit precision, the number of bits available for noise flooding is limited in these libraries, resulting in a low statistical security parameter; for example, PALISADE currently uses $s = 20$ [1]. Note that the precise number of bits of additional Gaussian noise for smudging required for a desired level of statistical security can be determined based on a tight analysis for CKKS [48]. These mitigations help mask noise resulting from general circuit evaluation under homomorphic encryption, including plaintext-ciphertext multiplications. In our multiparty setting, each party adds the required noise locally during partial decryption to ensure that the final (aggregated) decryption result does not leak information to any party. Table 4 shows that this smudging procedure has no significant impact on accuracy in our experiments across different levels of smudging noise. We note that a more sophisticated, dynamic mitigation strategy, allowing in some cases for the addition of noise of a lower norm, would require analyzing noise growth within the circuit [49] and adding an appropriate level of smudging noise. This remains an active area of research, and no widely implemented solution exists in current libraries. Leveraging Shechi's statistical code analysis framework to determine the expected noise growth in circuit evaluation is a meaningful direction for future work.

Principal Component Analysis (PCA). We reimplemented the recent secure distributed variant of the randomized PCA



(Appendix) Figure 12: MHE scales better than secret sharing when increasing the number of parties. Each party operates independently on a fraction of the input in MHE, while in secret sharing-based SMC, all parties perform the same amount of work over the secret shares of the entire pooled dataset, the size of which grows with the number of parties.

Solution	$\sigma_s = 2^{16}$	$\sigma_s = 2^{20}$	$\sigma_s = 2^{47}$
Kinship	$1.08 \cdot 10^{-0}$	$2.07 \cdot 10^{-0}$	$1.95 \cdot 10^{-0}$
PCA	$8.55 \cdot 10^{-5}$	$2.47 \cdot 10^{-4}$	$3.33 \cdot 10^{-8}$
GWAS	$4.73 \cdot 10^{-5}$	$4.02 \cdot 10^{-5}$	$1.67 \cdot 10^{-8}$

Table 4: The impact of increasing the standard deviation of the smudging noise on accuracy in our applications. We report the mean absolute difference between the offline, non-secure ground truth and Shechi for different values of the smudging sigma (standard deviation). The largest noise (i.e. 2^{47}) required different parameters where the default scale and moduli sizes were increased from 2^{34} to 2^{50}.

algorithm [29, 33]. We executed a joint PCA on 128 individuals split among two parties with 524,288 features and relied on the randomized PCA [33] algorithm to extract the first two components, using an oversampling parameter of 2 and 2 power iterations. For the eigendecomposition, we relied on the standard numerical computation via QR factorization, executing 5 iterations per eigenvalue.

Kinship Distance Computation (KING). We implemented a recent method [52] to compute kinship as a distance between two genotype vectors normalized by heterozygosity.

Genome-wide Association Study (GWAS). Our GWAS implementation is an adaptation of the state-of-the-art GWAS SMC method [18] and includes the comprehensive workflow required for an association study based on linear regression. It first captures the population structure into a low-dimensional matrix by executing a PCA and then conducts the association test via the Cochran-Armitage trend test, including both the PCs and patients' characteristics (e.g., age and sex) as covariates. GWAS aims at pinpointing genetic variations that exhibit correlations with a specific phenotype of interest, such as disease, susceptibility, or other quantitative biological traits. We executed a genome-wide association study in two standard steps: population stratification analysis and Cochran-Armitage trend test. For the first step, we relied on the PCA solution from the previous section. For the Cochran-Armitage trend test, we considered the two principal components from the first step as covariates, as well as two additional patients' features, i.e., sex and age. We computed the Cochran-Armitage trend test on 524,288 Single-nucleotide polymorphisms (SNPs).

G Code Examples

Shechi enables users to execute standard Python code by simply preceding it with one (Fig. 1) or two lines (Fig. 13) of initialization code specifying the partitioning of the data and setting up the communication and cryptographic environment. Using the latter approach allows the user to easily fine-tune cryptographic or input parameters. The network configuration is specified via CLI arguments when running the executables (see `https://github.com/0xTCG/sequre` for more examples and detailed instructions on how to install, compile, and run Shechi).

```python
from shechi import mhe, SDT

@shechi
def forward_qr(X):
    u = copy(X[0])
    u[0] += u.norm() + u[0].sign()
    u /= u.norm()
    X -= X @ u.T @ u * 2
    return X[1:, 1:]

mhe = mhe() # set multiparty environment
# Load data into a secure distributed tensor (SDT)
data = SDT.collective_load(mhe, "path.csv",
    rows_local=8192, cols=32, dtype=float).T

forward_qr(data)
```

(Appendix) Figure 13: Shechi's instantiation for secure execution on partitioned `data`.

```python
@shechi
def mnist(mhe, X, y, test_X, epochs):
    LogReg(mpc_initial_w, "multinomial").fit(
        mpc, X, y, epochs).predict(mhe, test_X).reveal()
```

(Appendix) Figure 14: Shechi-MNIST. The `LogReg` function is part of Shechi's library, supports linear, logistic and multinomial regression, and is written in about 100 lines of Pythonic code.