Secure Computational Genomics

by

Haris Smajlović
M.Sc., University of Sarajevo, 2017
B.Sc., University of Sarajevo, 2015

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

© Haris Smajlović, 2024
University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by
photocopying or other means, without the permission of the author.

Secure Computational Genomics

by

Haris Smajlović
M.Sc., University of Sarajevo, 2017
B.Sc., University of Sarajevo, 2015

Supervisory Committee

Ibrahim Numanagić, Supervisor
(University of Victoria, Computer Science)

Sean Chester, Department Member
(University of Victoria, Computer Science)

Riham AlTawy, Outside Member
(University of Victoria, Electrical and Computer Engineering)

ABSTRACT

Scattered between different biobanks and healthcare providers across multiple countries, biomedical data is extensively used for research purposes. Collaboration and sharing of such data between multiple institutions often provide access to more diverse datasets and a chance to conduct comprehensive studies. However, these collaboration efforts are usually hindered by privacy issues that render the pooling of such data at a centralized database impossible. To enable collaborative studies on top of such datasets, we present two easy-to-use domain-specific frameworks, Sequre and Shechi, for secure, high-performance computing on private, distributed datasets. Our frameworks automatically convert Pythonic code into a secure distributed equivalent using secure multiparty computation (SMC) in Sequre and, for the first time, multiparty homomorphic encryption (MHE) in Shechi to enable efficient distributed computation. They abstract away considerations about the private and distributed aspects of the input data from end users through a familiar Pythonic syntax and by introducing new data types for the efficient handling of distributed data as well as systematic compiler optimizations for cryptographic and distributed computation. We evaluate our framework on a wide range of applications, including complex genomic analysis tasks and statistical analysis of private electronic health records (EHRs). Our results demonstrate Sequre's and 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. Finally, our solution enables the utilization of distributed datasets as a whole to conduct collective studies between non-trusting private data proprietors and, as a result, facilitates data sharing and collaboration efforts in privacy-sensitive fields such as biomedicine.

ACKNOWLEDGEMENTS AND DEDICATION

First and foremost, my sincere and deepest gratitude belongs to my supervisor, Dr. Ibrahim Numanagić, who has been guiding me through this journey and offering his unrelenting support, patience, and advice for the last five years. I also thank my wonderful lab colleagues, with whom I created some unforgettable memories. This includes Hamza Išerić, Mansoureh Jalilkhany, Qinghui Zhou, Tristan Zaborniak, Mazyar Ghezelji, Sourena N. Moghaddasi, and Dr. Ümit Akırmak.

I want to thank the authors of Codon for the opportunity to collaborate with them. Here, I would like to mention Dr. Ariya Shajii and Dr. Bonnie Berger, who also co-authored Sequre and Shechi. I am grateful for their advice and help. Also, I must mention Gabriel Ramirez, Jessica Ray, and Dr. Saman Amarasinghe, whose weekly meetings I always looked forward to.

I would also like to thank the co-inventor of Sequre and Shechi, Dr. Hyunghoon Cho, for his help and guidance. I must mention here the co-first author of Shechi, Dr. David Froelicher, to whom I am thankful for bearing with me through many deadlines, day-long meetings, and discussions. Finally, I would like to thank the authors of Secure-MICE, namely Dr. Yi Lian, Dr. Qi Long, and especially Dr. Xiaoqian Jiang, whose invaluable insights and advice helped finalize my Ph.D. journey.

I want to extend my gratitude to Dr. Sean Chester, whose suggestions on performance optimizations underpin Sequre, Dr. Riham AlTawy, and Dr. Yun William Yu for their extensive reviews, which substantially improved the quality of this thesis.

Finally, I would like to thank my wonderful family for their unlimited love and support. This includes my dear wife Merisa, my dear parents Esad and Enisa, and my beloved brother Faruk. I dedicate this thesis to them.

Contents

Supervisory Committee **ii**

Abstract **iv**

Acknowledgements **v**

Contents **vi**

List of Tables **ix**

List of Figures **x**

1 Introduction **1**

2 Preliminaries **7**

2.1 Privacy-Enhancing Technologies 7

2.1.1 Secure Multiparty Computation 7

2.1.2 Homomorphic Encryption 12

2.1.3 Multiparty Homomorphic Encryption 14

2.2 Compiler Preliminaries: Codon Framework 15

2.2.1 Localized Type System With Delayed Instantiation 16

2.2.2 Intermediate Representation 18

2.3 Related Work 23

3 Sequre **26**

3.1 Sequre at a Glance 28

3.2 Sequre's SMC Framework 29

3.2.1 SMC-specific Optimizations 31

3.3 Applications 36

3.3.1 Secure Genome-Wide Association Studies 39

3.3.2 Secure Drug-Target Interaction Prediction 42
3.3.3 Secure Metagenomic Binning 44
3.3.4 Sequre and Other SMC Frameworks 48
3.3.5 Local-area Network Environment 54
3.4 Sequre's Standard Library 54
3.5 Benchmark and Hardware Setup 57

4 Shechi **59**
4.1 Problem Statement 62
4.2 Shechi at a Glance 64
4.3 Shechi's End-User Interface 64
4.4 Shechi's Secure Data Types 65
4.4.1 Secure Local Tensors 66
4.4.2 Secure Distributed Tensors 67
4.5 Shechi's Compiler Optimizations 68
4.5.1 Code Analysis & Optimization Workflow 68
4.5.2 Compiler Passes 70
4.5.3 Optimizations at Runtime 73
4.6 Shechi's Integrated MHE Libraries 75
4.7 Performance Evaluation 75
4.7.1 Evaluation Settings 76
4.7.2 Comparison with Existing Compilers 76
4.7.3 Scaling 80
4.7.4 Necessity of Shechi's Optimizations 81
4.7.5 Accuracy, Network, and Memory Usage 82
4.8 Implementation Details 83
4.8.1 Aggregation and Encoding Optimization 83
4.8.2 Novel Matrix Multiplication Method 83
4.8.3 Cryptographic Details & Parameters 83
4.8.4 Complex Workflows Details 85
4.9 Users' Choices and Stronger Threat Model 86
4.10 More Comparison Against Other Frameworks 86

5 Beyond Computational Genomics: Secure MICE algorithm **88**
5.1 Methods 90

5.2 Results 98

6 Conclusion **105**

Bibliography **107**

A A Short, Practical Guide to Sequre **127**

A.1 Data Pre-processing and Secret Sharing 127

A.2 Configuring the Network 128

A.3 Secure Training 129

A.4 Complete SMC Implementation of PlassClass 130

A.5 Deployment 131

B Other Applications: Machine Learning Module **134**

B.1 Linear Support Vector Machines 134

B.2 Neural Networks 135

C Selected Code Listings **138**

List of Tables

Table 2.1 Secure computing survey. 23

Table 3.1 Sequre GWAS results. 40
Table 3.2 Sequre GWAS projection. 41
Table 3.3 Sequre DTI results. 44
Table 3.4 Sequre metagenomic binning results. 48
Table 3.5 Sequre cross-comparison. 53
Table 3.6 Sequre LAN results. 54
Table 3.7 Supported SharedTensor operations. 56
Table 3.8 Tools, versions, and datasets. 58

Table 4.1 Low-level primitives micro-benchmark. 77
Table 4.2 Other micro-benchmarks. 87

Table 5.1 Secure MICE: real-data results. 102
Table 5.2 Secure MICE: scenario 1 results. 103
Table 5.3 Secure MICE: scenario 2 results. 103
Table 5.4 Secure MICE: scenario 3 results. 104
Table 5.5 Secure MICE: scenario 4 results. 104

List of Figures

Figure 2.1 Overview of SMC. 8
Figure 2.2 Object metadata overhead in Python. 15
Figure 2.3 Cases that cannot be type-checked. 16
Figure 2.4 Program that cannot be type-checked. 18
Figure 2.6 CIR hierarchy 18
Figure 2.5 Monomorphization example. 19
Figure 2.7 Codon's compilation pipeline. 20
Figure 2.8 CIR pass example: integer addition. 21
Figure 2.9 Example of bidirectional compilation in Codon IR. 22

Figure 3.1 Sequre's compiler workflow. 30
Figure 3.2 Beaver caching example. 32
Figure 3.3 Sequre's polynomial optimization: term expansion. 34
Figure 3.4 Sequre's algebraic structures. 34
Figure 3.5 Pattern-matching transformations. 36
Figure 3.6 Sequre results. 37
Figure 3.7 Sequre GWAS accuracy. 39
Figure 3.8 Sequre GWAS manhattan plot. 42

Figure 4.1 Shechi overview. 62
Figure 4.2 Shechi high-level example. 62
Figure 4.3 Shechi's overall structure. 63
Figure 4.4 Secure distributed tensor. 66
Figure 4.5 Shechi's code analysis and optimization passes. 69
Figure 4.6 Aggregation & encoding Optimization. 71
Figure 4.7 Shechi's results. 79
Figure 4.8 Shechi's scaling. 80
Figure 4.9 Shechi's ablation. 81
Figure 4.10 Shechi's accuracy. 82

Figure 5.1 Multiple imputation. 91
Figure 5.2 Multiple imputation via SMC. 92
Figure 5.3 Multiple imputation via MHE. 96

Figure A.1 PlassClass preprocessing. 128
Figure A.2 Sequre's secret sharing routine. 128
Figure A.3 Client compile and run instruction. 128
Figure A.4 Network configuration. 129
Figure A.5 Linear SVM training. 129
Figure A.6 PlassClass training in Sequre. 130
Figure A.7 PlassClass client call. 131
Figure A.8 PlassClass server call. 132
Figure A.9 PlassClass training output. 133

Figure B.1 Linear SVM training in Sequre/Shechi. 136
Figure B.2 Linear SVM inference in Sequre/Shechi. 136
Figure B.3 Neural networks in Sequre/Shechi. 137

Figure C.1 Shechi's instantiation. 138
Figure C.2 Kinship computation. 139
Figure C.3 QR decomposition. 139
Figure C.4 Shechi-PCA. 140
Figure C.5 Shechi-GWAS. 141

Chapter 1

Introduction

The hurdle of safe means of storage, protection, and sharing of genomic information is a major challenge that hinders the safeguarding of genomic data and collaborative research [13, 33]. The standard means of data protection, in which proprietors keep the data enclosed within tightly access-controlled silos, works only in limited circumstances where, ideally, the data is never intended for sharing. In such instances, data owners typically maintain data in its raw, non-encrypted form and offer sharing services through prolonged legal processes. Needless to say, each raw data sharing attempt inadvertently increases the leakage risk despite the strict legal sharing protocols. Even if data is not intended for sharing, however, it is still at risk, as recently demonstrated by the theft of millions of identified genomic information from a major genomic company that resulted in blackmail attempts and lawsuits [33]. A common approach to facilitating data sharing, in general, is via de-identification, where private information is redacted prior to sharing, and the data is shared in its raw, non-encrypted form. This approach, however, has been proven to be challenging in genomics by recent studies due to the possibility of triangulating back to the genome owner from the genome itself [20]. In most recent studies, however, the secure means of protecting genomic privacy is found in *Privacy-Enhancing Technologies (PETs)* [33]. These technologies enable computing on top of private data without breaking confidentiality. One such example is Differential Privacy (DP) [47], which ensures that the two images of a randomized function are indistinguishable when given two datasets that differ by a single entry as an input. This is usually achieved by adding some noise to the data and, intuitively, offers a trade-off between utility and preserving privacy. While DP is an active area of research in genomics, it is still of limited applicability since achieving its theoretical guarantees is hindered

by the statistical properties of genomic data. For example, *linkage disequilibrium* property allows for multiple, non-random locations or regions of the genome to be correlated [143]. Another promising approach is found in technologies that enable direct computation on top of encrypted data without decryption. Such technologies include Homomorphic Encryption (HE) [21, 22, 31], Secure Multiparty Computation (SMC) [52], and Multiparty Homomorphic Encryption (MHE) [117]. Homomorphic encryption uses an encryption scheme in which the encryption function is homomorphic with respect to addition and multiplication[1]. Secure multiparty computation, on the other hand, uses either *garbled circuits* or *secret-sharing* [52]—encryption schemes in which the decryption function is distributed between multiple computing parties—to enable specialized secure distributed computation. Finally, Multiparty Homomorphic Encryption (MHE) marries HE and SMC to utilize the best of both approaches.

Each PET is suitable for a different use case and, at the same time, suffers from a unique set of drawbacks. For example, the theoretical guarantees of differential privacy are hard to obtain in genomics, Fully Homomorphic Encryption suffers from a significant computation overhead, while Secure Multiparty Computation introduces network overhead that scales quadratically with the increase of the number of parties and data size. Also, both HE and SMC require domain-level expertise and often produce convoluted codebases. Finally, MHE combines HE and SMC into a protocol that enables both centralized and distributed computing and scales linearly with the increase of study participants and data size but requires even more domain-level expertise and produces codebases that are order of magnitude more complex than even those of SMC and HE.

Here, we utilized *Codon* [141], a compiler framework for compiling Pythonic code to fast executables and building Pythonic domain-specific frameworks, to build two frameworks with Pythonic syntax for secure computing: Sequre, a framework for SMC, and Shechi, a framework for MHE. These frameworks hide complex cryptographic workflows behind an easy-to-use Pythonic syntax and use compile-time and runtime optimizations to produce executables that surpass the performance of state-of-the-art secure solutions. They also ship with high-level libraries for the essential matrix and vector operations akin to standard NumPy library [72] and a range of

[1]Encryption function in HE may be homomorphic with respect to one or multiple operations. HE scheme that offers multiple homomorphic operations is called the *Fully* Homomorphic Encryption scheme (FHE). In this work, we focus exclusively on FHE and refer to it, unless explicitly noted, as HE throughout the text.

machine learning modules, including linear and logistic regression, support vector machines, and neural networks[2].

Sequre is a Python-like, high-performance domain-specific framework for developing secure SMC algorithms. It uses standard Python syntax and semantics to ease the development of secure pipelines and the transformation of existing code into SMC equivalents. Its novel optimizations utilize Codon's *intermediate representation* (IR) of the Python source code (i.e., a logical representation of a program's execution flow that can be statically analyzed) to remove unnecessary computation and select the best SMC routines and optimization approaches for each computational step. As a result, Sequre enables high-performance and simple codebases that do not require extensive SMC-related modifications.

We demonstrate Sequre's performance and usability by employing it to implement two genomics pipelines, such as genome-wide association study (GWAS) [34] and metagenomic binning [106,124], and one workflow in pharmacology, namely the drug-target interaction (DTI) inference [77]. To our knowledge, a secure SMC protocol for metagenomic binning has not been previously developed. We implemented each pipeline in only 80–160 lines of high-level Python code, the functionality of which is equivalent to the original algorithm. We achieved up to 7× reduction in code length compared to the existing state-of-the-art pipelines. Furthermore, the overall execution time of these pipelines was reduced by 3–4×, and the network utilization was also 17% lower. Where possible, we also compared Sequre to an existing Python-based SMC framework, PySyft (SyMPC) [166], and showed that various machine learning tasks can be performed 2× faster while providing comparable security guarantees. We also micro-benchmarked Sequre's performance and compared it to ten existing SMC frameworks using a standard benchmark suite [73]. Sequre achieved the best runtime performance in most cases while being one of the most accessible frameworks.

We expect Sequre to enable practitioners without expertise in SMC and cryptography to easily write efficient SMC algorithms for various genomics workflows. Also, while Sequre has been tested exclusively on large-scale genomic applications, it applies to non-genomic fields as well. Furthermore, the improved readability and usability of Sequre programs can simplify the sharing and maintenance of secure workflows.

Sequre and Secure Multiparty Computation, in general, offer a provably secure

[2]We introduce Sequre and Shechi in the following paragraphs through adaptation of our existing texts that are either already published [144,145] or have passed the rebuttal stage as of the time of this writing.

means of protecting and sharing private data between multiple non-trusting parties. However, while Sequre enables practical runtimes for applications that operate on large-scale data, it still comes with one important drawback that renders it impractical for use cases that involve many study participants. Namely, it requires network communication between all study participants, and for this reason, it scales quadratically with respect to the number of collaborating parties. On the other hand, while the computation overhead associated with the Homomorphic Encryption primitives often leads to impractical runtimes on large datasets, its multiparty variant (i.e. MHE) amortizes this cost by enabling computation on local, non-encrypted data at each party and independent, parallel computing between the parties. Although 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, MHE 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, increasing the difference between practical and infeasible solutions [58]. 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.

Thus, 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 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. 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.

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 [35] 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 Sequre 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.

Finally, we extended the application of Sequre and Shechi beyond computational genomics and pharmacology to solve the problem of conducting statistical studies on top of privately owned, distributed, incomplete Electronic Health Records (EHRs) in U.S. hospitals. Here, for the first time, we implemented a provably secure variant of the *multipile imputation with chained equations* (MICE) algorithm using SMC and MHE. Our goal was to meet the accuracy of the offline, non-secure implementations of MICE, which Sequre and Shechi ultimately achieved. More importantly, Sequre provided practical runtimes, in some cases even faster than the offline counterpart written in Python, often running in less than a few seconds (from 38ms for 500

patients to 12s for more than $50,000$ patients). On the other hand, while Shechi was as accurate as other solutions, it unsurprisingly provided less practical runtimes for datasets of these dimensions. However, we expect Shechi to outperform Sequre in case scenarios with much larger datasets and the number of collaborators. Thus, together with Sequre, our frameworks for secure distributed computation enable practitioners to write performant, secure applications easily and constitute an important step for the design and adoption of secure distributed solutions.

Chapter 2

Preliminaries

Our work is underpinned by two main components: privacy-enhancing technologies and compilers. In the following sections, we explore the cryptographic schemes and the compiler framework employed by our solutions. Specifically, we used Secure Multiparty Computation (SMC) and Multiparty Homomorphic Encryption (MHE) schemes, as well as Codon, a framework for building high-performance, Pythonic domain-specific frameworks.

2.1 Privacy-Enhancing Technologies

Here we present the essentials of the particular privacy-enhancing technologies employed in our work. Our choice of cryptographic schemes is influenced by the practical requirements of computational genomics, where the size of genomic data is detrimental to performance. Having that said, adding support for other cryptographic schemes is possible, and it is part of future work.

2.1.1 Secure Multiparty Computation

Secure multiparty computation enables computing on top of private datasets distributed between multiple non-trusting data proprietors, without disclosing any sensitive information, apart from the final outcome of the study. There are many variants of SMC [52, 73, 89] and, driven by the design decisions in the field of computational genomics [13, 33], we opted for *secret sharing* scheme, where the input is divided between the computing parties such that no single party can obtain any meaningful information from its data share, but the computation on top of such, distributed data

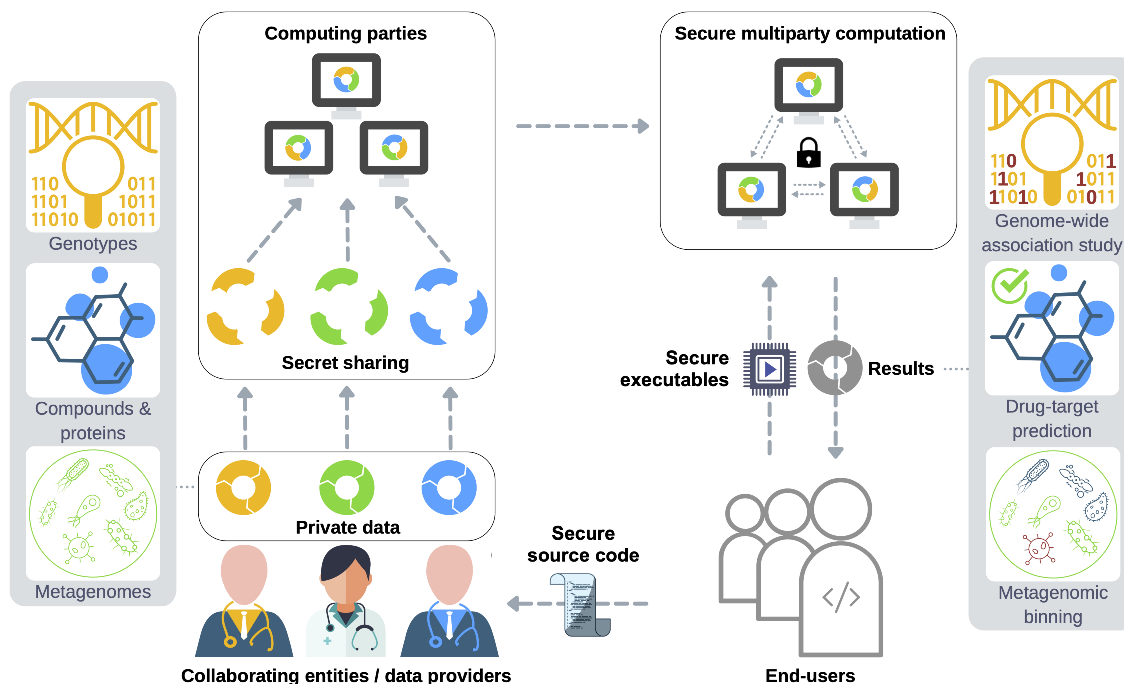


Figure 2.1: Overview of *Secure Multiparty Computation* and its typical use-case. Secure Multiparty Computation enables collaborative analysis of sensitive private data, such as patient genomes and proprietary pharmaceutical datasets, without disclosing the data to anyone other than the respective *data providers*.

is still possible. Specifically, we employed an *additive secret sharing* scheme [1] where each data holder splits each of their private data value x into n shares $x_1, x_2, \ldots x_n$, where n is the number of computing parties, such that $x = \sum_{i=1}^{n} x_i$ (modulo a predetermined modulus). The data holder then shares each x_i with CP_i respectively for $i = \{1, \ldots, n\}$. These shares satisfy the key property that each share individually is indistinguishable from a uniformly random number, thus hiding the underlying secret. Computing parties will then execute an interactive protocol in which they perform calculations over the respective secret-shared data. Subroutines defined for individual operations (e.g. multiplication) provide theoretical guarantees that the messages exchanged during the computation do not reveal any information about any party's private data. Once the computation is complete, the result is revealed and made available to the end-users by combining the final shares.

The above additive secret sharing scheme maintains the two invariants: (i) no

[1] There are other variants of secret sharing, too, such as *multiplicative secret sharing* and *Shamir's secret sharing*. Additive secret sharing, however, has been predominantly employed in computational genomics due to its superior performance in fixed-point arithmetic.

computing party can infer any information about the private data that they do not already have access to; and (ii) the shares are constrained to have a sum (modular addition) equal to the secret value they collectively represent. Because of this constrained data representation, performing computation over the private data (which are not directly accessible), e.g. multiplying two secret numbers, requires the use of special operations which typically require the parties to interact and exchange messages (see Section 3.4 for details). Note that in Sequre, we adopt an efficient server-aided model of SMC, in which one computing party plays the role of a *trusted dealer* who does not receive any private data (even secret shares) and only generates correlated random numbers in a preprocessing step to be shared with other computing parties to facilitate the main protocol. While this weakens the security model by requiring the trusted dealer not to collude with any other party, the resulting performance gain is significant, which is critical for large-scale genomic data.

The security of an SMC framework is typically expressed in terms of the power of the anticipated adversary (e.g. a rogue computing party). Common models include *honest-but-curious* and *malicious* adversaries. An honest-but-curious party will faithfully follow the computational protocol as given but may attempt to perform additional analysis based on the observed data during the protocol to infer information about the private data. A malicious party is additionally allowed to deviate from the prescribed computational protocol, potentially sending fabricated information in order to extract private information. Based on the type of adversary addressed by the chosen security model, SMC operations need to be revised accordingly to provide adequate protection. In general, stronger schemes that guard against malicious parties tend to be more computationally expensive. We note again that, in principle, allowing the presence of a trusted dealer and adopting an honest-but-curious threat model has been standard practice in genomic applications [33, 34]. Having said that, extending our SMC scheme to support a stronger threat model without a trusted dealer or even with the malicious security model, as presented in earlier work [34], is also possible and is part of future work.

Some SMC frameworks support only a limited number of computing parties (e.g. garbled circuits are for two parties). Frameworks that can be instantiated with more computing parties arguably offer stronger security since compromising privacy based on secret shared data typically requires compromising all (or the majority of) the computing parties.

Another aspect worth noting is that additive secret-sharing-based frameworks can

be instantiated with different types of finite algebraic structures required for modular arithmetic. Two popular choices are a finite field (integers modulo a prime; $\mathbb{Z}_p$) or a ring of integers with a power-of-two modulus ($\mathbb{Z}_{2^k}$). Recently, $\mathbb{Z}_{2^k}$ rings are becoming more widely adopted due to their superior performance on modern computer architectures leveraging native integer operations. However, $\mathbb{Z}_{2^k}$ rings support a more limited set of operations as modular inverse generally does not exist, which some operations require. Lastly, fractional numbers are commonly represented using fixed-point numbers, incurring some level of precision loss, which can be controlled with the choice of modulus size.

Developing efficient SMC protocols based on different secret-sharing schemes and security assumptions is an active area of research. The following subsections provide an insight into some essential operations in additive secret-sharing schemes.

Notation

We adopted a standard SMC notation from previous work [34, 41, 52]. While we support an arbitrary number of computing parties (CPs) and study participants (SPs, also known as *data holders*), we will here assume without the loss of generality to have only three computing parties (CP_0, CP_1, and CP_2), or to be more precise, two standard computing parties (CP_1, CP_2) and an auxiliary computing party (CP_0; also known as trusted dealer). We will denote each party's data with the angled brackets: $\langle a, b, c, d \rangle$ means that a, b, c, and d are accessible by CP_1, CP_2, CP_0, and SP, respectively. We will adhere to a convention that $\langle a, b \rangle \equiv \langle a, b, \perp, \perp \rangle$ and $\langle a, b, c \rangle \equiv \langle a, b, c, \perp \rangle$, where $\perp$ denotes an empty slot. Further, a pair of secret shares $\langle x_1, x_2 \rangle$ of x will be often shortened as $[x]$. Also, unless explicitly noted, each line of pseudocode is executed on all computing parties in parallel, and each routine operates on top of an algebraic structure S, which can interchangeably be the Galois field $\mathbb{Z}_p$ or a finite $\mathbb{Z}_{2^k}$ ring.

Revealing

The procedure for revealing the secretly shared value in additive secret-sharing schemes follows the opposite steps from the encryption: a value is revealed by adding up the secretly shared values together. In other words, $x = x_1 + x_2$ (modulo a field/ring characteristic).

Addition

Adding the two numbers in additive secret-sharing-based schemes is a straightforward procedure. Namely, for the sum of two secret-shared numbers $[x]$ and $[y]$ we just add the respective shares together: $[x + y] = \langle x_1 + y_1, x_2 + y_2 \rangle$. Note that revealing the obtained shares reveals the value of $x + y$. However, when adding a public value a to a secret-shared number, it suffices to add it to one and only one of the shares: $[x] + a = \langle x_1 + a, x_2 \rangle$. For detailed SMC pseudocodes for the procedures above, see protocols 3 and 4 in Cho et al. [34].

Multiplication and Beaver Partitions

Multiplying a secret-shared number $[x]$ with a public number a is straightforward in an additive secret-sharing scheme. It suffices to multiply each share of x with the public number: $[ax] = \langle ax_1, ax_2 \rangle$.

On the other hand, multiplying the two secret-shared values is not as easy. Note that if the shares at each computing party are simply multiplied together, we obtain an incorrect result: $x_1 y_1 + x_2 y_2 \neq xy$. This problem can be avoided by using the *Beaver multiplication triplets* [10]. In this method, to compute the shares of a product $[xy]$, we first sample and secretly share the three random numbers $a \in S$, $b \in S$, and $c = ab \in S$ and then reveal the value of $x - a$ and $y - b$ to finally compute $(x-a)(y-b) + (x-a)[b] + (y-b)[a] + [c]$. Revealing the last value yields $(x-a)(y-b) + (x-a)b + (y-b)a + ab$ which is equal to xy.

Here, however, we implement an improved variant of Beaver triplets multiplication that introduces a concept of *Beaver partitions* and leverages a trusted dealer and the streams of pseudo-random generators to reduce the online communication between the computing parties. Specifically, Beaver partitions of a secret-shared number $[x]$ are defined as the values $\langle x - r, x - r \rangle$ and $\langle r_1, r_2, r \rangle$, where $r \in S$ is randomly sampled by a trusted dealer and secretly shared to the computing parties. To multiply the two secret-shared numbers $[x]$ and $[y]$, it suffices to multiply their Beaver partitions in a cross-over fashion: $[xy] = (x - r_x)[r_y] + (y - r_y)[r_x] + [r_x r_y] + (x - r_x)(y - r_y)$, where $(x - r_x)(y - r_y)$ is added publicly and $r_x r_y$ is computed at trusted dealer and secretly shared to other computing parties. For details, see supplementary notes 3 and 7 in Cho et al. [34].

Generalized Polynomial Evaluation

The generalized polynomial form

$$P_m(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{m} c_i \prod_{j=1}^{n} x_j^{p_{ij}} \ , \ c_i, x_j, p_{ij} \in \mathbb{R}$$

can be evaluated with a minimal number of online communication rounds in our SMC scheme [34]. It suffices to compute the Beaver partitions $x_j - r_j, [r_j]$ for each variable x_j only once and then compute the public coefficients in the expansion of the polynomial terms $x_j^{p_{ij}} = ((x_j - r_j) + [r_j])^{p_{ij}}$ offline—within each computing party—to evaluate the polynomial. However, the expansion of the offline terms is exponential with respect to the polynomial degree and can be impractical if the polynomial degree is too high. In this case, a simple series of traditional multiplication routines often performs better.

2.1.2 Homomorphic Encryption

A cryptographic scheme (or cryptosystem) is usually defined as a tuple of a key generator that samples a random key from a keyspace $\mathcal{K}$, an encryption function $E : \mathcal{M} \to \mathcal{C}$, and a decryption function $D : \mathcal{C} \to \mathcal{M}$, where $\mathcal{M}$ and $\mathcal{C}$ are the message space and the cipher space, respectively. Encryption and description functions are usually defined for a specific key $k \in \mathcal{K}$ and denoted as E_k and D_k, respectively [87]. Further, a function $f : X \to Y$ is said to be homomorphic with respect to operator $\circ$ whenever $f(x \circ y) = f(x) \circ f(y)$, for all $x \in X$ and $y \in Y$. Any cryptographic scheme whose encryption function is homomorphic with respect to some operator, such as addition (i.e. $E_k(x + y) = E_k(x) + E_k(y)$) or multiplication (i.e. $E_k(xy) = E_k(x)E_k(y)$), is considered to be a *Homomorphic Encryption* (HE) scheme[2]. A number of cryptosystems, such as RSA [130], ElGamal [48], and Paillier [122], offer this property for exactly one operator (either addition or multiplication)[3]. These cryptosystems are known as *Partially* Homomorphic Encryption schemes, and they are of limited computational power on top of encrypted data. The first generic cryptographic scheme that offered the encryption function homomorphic under both addition and multi-

[2]Note that, since D_k is inverse of E_k, homomorphism of E_k implies the homomorphism of D_k for a given operator.

[3]Note that secret sharing, as an encryption function is, in principle, homomorphic with respect to both addition and multiplication, but it is restricted to distributed setups only.

plication was introduced by Gentry [65] in 2009. This scheme essentially enabled arbitrary computation on top of encrypted data and is considered to be the first *Fully* Homomorphic Encryption scheme (FHE). In this thesis, any reference to Homomorphic Encryption refers to Fully Homomorphic Encryption. There are multiple variants of FHE [21,22,31], and here, we adhere to the Cheon-Kim-Kim-Song (CKKS) scheme [31]. 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 $t = \mathcal{N}/2$ complex numbers. Roughly speaking, a vector $\mathbf{z} \in \mathbb{C}^t$ is encoded as polynomial $p \in \mathbb{Z}_q[X]/(X^{\mathcal{N}} + 1)$, such that $z_i = p(\xi^{2i-1})$ where ξ is a primitive root of -1. Security is then based on the ring learning with errors (RLWE) problem [107], in which it is considered hard to find the value of $s \in \mathbb{Z}_q[X]/(X^{\mathcal{N}}+1)$, given the list of pairs $(a_i, a_i s + e_i)$, for $a_i \in \mathbb{Z}_q[X]/(X^{\mathcal{N}} + 1)$, $e_i \leftarrow \mathbb{Z}_q[X]/(X^{\mathcal{N}} + 1)$, and $i > 0$. 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[4]. 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.

Encryption and Decryption

The encoding $m \in \mathbb{Z}_q[X]/(X^{\mathcal{N}} + 1)$ of input vector $\mathbf{z} \in \mathbb{C}$ is encrypted under the encryption function $E_p : \mathbb{Z}_q[X]/(X^{\mathcal{N}} + 1) \to (\mathbb{Z}_q[X]/(X^{\mathcal{N}} + 1))^2$ as $c = E_p(m) = (m - as + e, a) = (c_0, c_1) \in (\mathbb{Z}_q[X]/(X^{\mathcal{N}} + 1))^2$, where $a, s, e \leftarrow \mathbb{Z}_q[X]/(X^{\mathcal{N}} + 1)$ and e encodes some small noise. Polynomial s is considered to be a secret key, while the pair of polynomials $p = (-as + e, a)$ is a public key.

Decryption $D : (\mathbb{Z}_q[X]/(X^{\mathcal{N}} + 1))^2 \to \mathbb{Z}_q[X]/(X^{\mathcal{N}} + 1)$ is then simply defined as $m = D(c) = D((c_0, c_1)) = c_0 + c_1 s$.

[4]Up to 7× times faster in our experiments; see Section 4.8.3).

Addition and Multiplication

Adding two ciphertexts $c' = (c'_0, c'_1)$ and $c'' = (c''_0, c''_1)$ is straightforward since $D(c') + D(c'') = c'_0 + c''_0 + (c'_1 + c''_1)s = D(c' + c'')$.

Multiplication is a bit more complex since $D(c')D(c'') = D(c'_0c''_0, c'_1c''_0 + c'_0c''_1) + c'_1c''_0s^2$. Finding $d'_0, d'_1 \in (\mathbb{Z}_q[X]/(X^{\mathcal{N}} + 1))^2$ such that $D(c'_0c''_0, c'_1c''_0 + c'_0c''_1) + c'_1c''_0s^2 = D(d'_0, d'_1)$ is part of *relinearization* process mentioned above and is an integral part of RLWE-based schemes.

2.1.3 Multiparty Homomorphic Encryption

Most RLWE-based HE schemes allow for the decryption key (secret) to be distributed between multiple computing parties. This way, parties can collaborate together by collectively generating a distributed secret key, encrypting their private data via a shared public key, and sharing the encrypted data for downstream computing. This concept has been only recently introduced as a distributed HE scheme [117] and has later been extended to a hybrid scheme allowing to switch from data encrypted in HE under a public key to secret-shared data in SMC [32]. The decryption key is shared among the parties via SMC, 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. In our case, the secret key is distributed additively between the computing parties, reflecting our additive secret-sharing scheme and facilitating switching between MHE and additive secret-sharing-based SMC.

Independent computing is especially beneficial since, unlike in secret-sharing-based SMC, the workload can be divided between the computing parties. This allows MHE to scale linearly with respect to the number of parties and size of data, which results in superior performance when compared to SMC in mass collaborations and large-scale data (the interactive protocols of SMC scale quadratically with respect to number of parties and data size). On the other hand, interactive protocols can be used to replace the expensive HE bootstrapping operation and to convert HE ciphertexts into additive shares for SMC operations [32] in the use cases where SMC is of superior performance (usually when the collaboration or data size is small). Lastly, unlike secret-sharing, MHE allows offloading parts of computation offline, on non-encrypted data, to each computing party, should the application context allow it. MHE can, therefore, be used to divide operations across HE and SMC while leveraging the

strengths of both schemes and performing well on large-scale datasets.

2.2 Compiler Preliminaries: Codon Framework

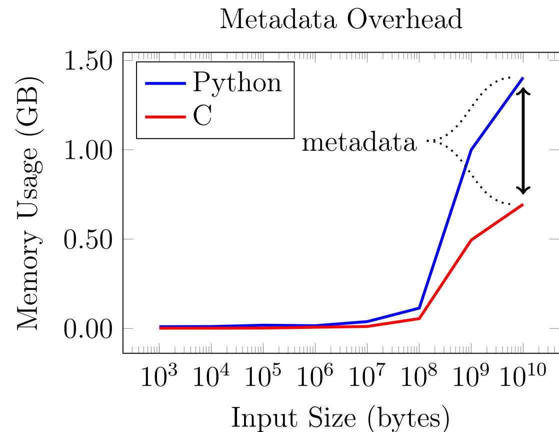


Figure 2.2: Object metadata overhead in Python, using C as a reference.

To enable Pythonic syntax and custom code optimizations in our solutions, we used Codon[5], a novel solution to bring high-performance DSLs to the Python user community, by building a flexible development framework on top of an optimized Pythonic base [141]. Codon is a full language and compiler that borrows Python 3's syntax and (with some limitations) its semantics and library features but compiles to native machine code with zero runtime and metadata overhead, allowing it to rival C/C++ in performance. To this end, Codon leverages a specialized bidirectional type-checking algorithm and novel intermediate representation (IR) to enable optional domain-specific extensions—both in the language's syntax (front-end) and in compiler optimizations (back-end)—via a plugin system.

Unlike other performance-oriented Python implementations (such as PyPy [19], Numba [4] or Nuitka [74]), Codon operates completely independently of the standard Python runtime and interpreter and is built from the ground up as a standalone system. As a result, Codon is able to achieve vastly superior performance and overcome issues such as the infamous global interpreter lock, which most other Python implementations retain. While Codon (at the time of writing) is not yet a drop-in replacement for standard Python, it circumvents the hurdle of having to learn an entirely new language or ecosystem, and still nonetheless allows for substantial code reuse from existing Python programs.

A critical difference between Codon and other optimized Python implementations is that Codon does not leverage just-in-time (JIT) compilation, but rather compiles full programs ahead of time. Consequently, Codon does not need to retain any runtime type information whatsoever, unlike JIT compilers, which still require such metadata

[5]This section is adapted from our recently published work [141]

to accommodate non-JIT'ed code. This metadata overhead poses a challenge in a number of domains, including computational genomics, when dealing with a large number of small objects, each of which carries per-object overhead. For example, many genomics applications need to store and manipulate billions of small sequences ("ACGT" strings) concurrently [138]. Figure 2.2 shows the effect of metadata overhead on such an application; the plot shows memory usages for the Python and C implementations of the "k-nucleotide" benchmark from *The Computer Language Benchmarks Game* suite of benchmarks [69], for various input sizes. This benchmark involves counting occurrences of various sequences in a large dataset. As the input size increases, so too does the overhead of the Python implementation—Codon, on the other hand, does not have any of Python's metadata overhead, allowing it to be applied effectively in such domains.

Codon is built through two main components:

- *Localized type system with delayed instantiation* (LTS-DI) which combines existing techniques commonly used to type check strongly-typed languages, such as *bidirectional type inference*, *parametric polymorphism*, *monomorphization* and *static expressions*, to statically type check dynamic Python syntax.

- *Bidirectional intermediate representation* where a new class of IRs called *bidirectional IRs* is added. Here, the compilation does not follow a linear path after parsing, but can return to the type-checking stage during IR passes to generate new specialized IR nodes.

2.2.1 Localized Type System With Delayed Instantiation

Codon employs a new type system, *localized type system with delayed instantiation (LTS-DI)*, which builds on top of the classical Hindley-Milner-Damas bidirectional type inference algorithm [40, 78, 113] used in Standard ML and many other functional languages. LTS-DI combines several ideas from RPython [5] and Starkiller [132] (e.g. monomorphization, localized typing approach), and

```
def foo():
  a = []    # a: List[α]
  b = None  # b: Optional[β]
  # At this point, α and β are unbound and
  # cannot be resolved in the scope of foo.
```

```
def foo(x):
  x.append(1)
a = []  # a:       List[α]
foo(a)  # foo(a): Function[[β], η]
        # unify:   β ←List[α]
# At this point, type of a is still unbound α,
# and thus Codon cannot instantiate foo.
```

Figure 2.3: Examples of cases that cannot be type-checked by Codon.

glues them to a Standard ML-like backend to achieve more comprehensive static type checking of Pythonic code.

However, LTS-DI makes some notable departures from the Standard ML type system in order to be compatible with Python's duck typing approach:

Delayed instantiation The complete type of function's body, as well as its return type, will be inferred by the type checking algorithm solely from the provided argument types at the time of realization, and not of definition (recall that Standard ML will infer the most general type of function at the definition time). In other words, LTS-DI will delay the instantiation (and type checking) of a function until all function parameters become bound. This technique, when combined with monomorphization and generic functions (in other words, parametric let-polymorphism), allows LTS-DI to faithfully simulate Python's runtime duck typing at compile time.

Localization LTS-DI will treat each function block as an independent type checking unit with its own typing context. Thus, each function block must be locally and independently resolvable by the type system without knowing anything about other blocks' context. For example, as seen in the top snippet of Figure 2.3, the type of `a` cannot be inferred from the scope of `foo` alone, and as such will produce a compiler error. Because the outermost scope is treated as a function itself, the type of `a` also cannot be inferred from the top-level alone in the bottom snippet of Figure 2.3 (the fact that `foo` can realize `a` does not help, as each function context is independent of other contexts).

Static evaluation A ML-like type system such as LTS-DI can sometimes overeagerly reject a valid Python program. For example, a common Python pattern is to dynamically check the type of an expression via `isinstance`, and to proceed to different blocks of code for different types, often within an `if-else` statement. A similar strategy is also used in conjunction with `hasattr`. For example, both `isinstance` and `hasattr`, as well as many other methods, can and are resolved at compile time. Thus, if an `if`-statement's condition is a static expression, it will be type checked and compiled *only if* the expression evaluates to true at compile time. Thus the compiler can opt out of compiling blocks that fail a `hasattr` check

```
(* OCaml *)
let f g a b = (g a, g b) in
  let g x = x in
    f g 1 "a"   (* error *)
```

```
# Python
def f(g, a, b): return g(a), g(b)
def g(x): return x
f(g, 1, "a")  # compiles
```

Figure 2.4: Program that cannot be type-checked by standard Hindley-Milner type inference algorithms, but *can* by LTS-DI. Since LTS-DI delays function instantiation, it supports multiple applications of `f` on different argument types, unlike OCaml.

(Figure 2.5).[6] Codon also allows static expressions to instantiate types and functions: for example, the n-bit integer type can be expressed as `Int[N: Static[int]]`.

These departures and extensions—or in some cases, restrictions—of Standard ML's type system allow LTS-DI much greater flexibility in dealing with generic functions. Most importantly, LTS-DI treats generic functions as *bound types*, and instantiates them as unbound types only during their application. In other words, if a generic function is passed as an argument to a function `foo`, it can be instantiated differently depending on the supplied arguments within the local context of `foo`, unlike in ML where the first instantiation determines the type of the function variable. A concrete example of this distinction is shown in Figure 2.4. These differences from Standard ML together allow better compatibility with Python, by enabling more general lambda support, as well as support for *returning* generic functions—a necessary requirement for implementing Python's decorators (Figure 2.5).

2.2.2 Intermediate Representation

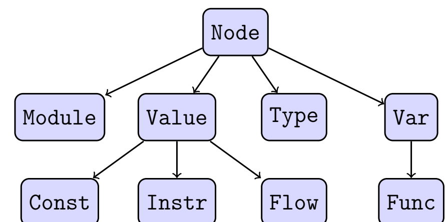


Figure 2.6: CIR hierarchy

Codon introduces a custom intermediate representation: a Codon Intermediate Representation (CIR), which is radically simpler than the standard IRs such as abstract syntax trees (ASTs). Despite this simplicity, CIR maintains most of the source's semantic information and facilitates "progressive lowering," enabling optimization at multiple layers of abstraction similar to other IRs [98, 156]. Optimizations that are more convenient at a given layer of abstraction are able to proceed before further lowering.

[6]Note that, unlike many other languages, Codon automatically detects static expressions and does not require manual annotations.

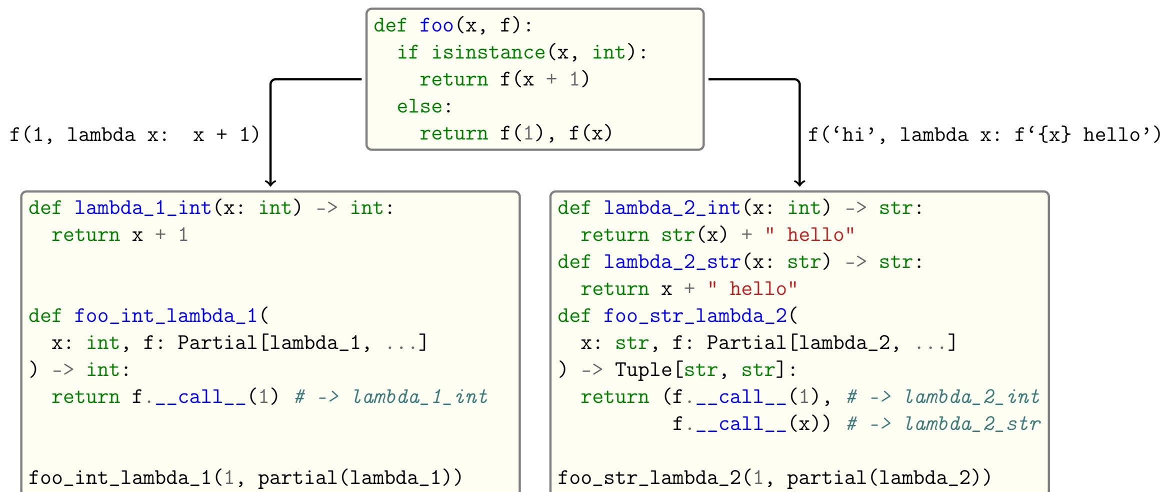


Figure 2.5: Example of monomorphization and static evaluation in Codon. By combining these two, Codon can support many common Pythonic constructs, like `isinstance` type checking, generic functions that return different types on different invocations, and so on.

High-Level Design

CIR is a value-based IR inspired in part by LLVM IR [97], where a structure similar to single static assignment (SSA) form is employed, making a distinction between *values*, which are assigned once, and *variables*, which are conceptually similar to memory locations and can be modified repeatedly.

Unlike LLVM, however, the IR initially represents control flow using explicit nodes called *flows*, allowing for a close structural correspondence with the source code. Importantly, this makes optimizations and transformations that depend on precise notions of control-flow much easier to implement. One key example is the `for` flow: in Pythonic languages, the `for x in range(y)` pattern is exceedingly common; maintaining explicit loops allows Codon to easily recognize this pattern rather than having to decipher a maze of branches, as is done in lower-level IRs like LLVM IR.

Bidirectional Intermediate Representations

Traditional compilation pipelines are linear in their data flow: source code is parsed into an AST, usually converted to an IR, optimized, and finally converted to machine code. Codon introduces the concept of a *bidirectional IR*, wherein IR passes are able to return to the type checking stage to generate new IR nodes and specializations not present in the source program. Among the benefits of a bidirectional IR are:

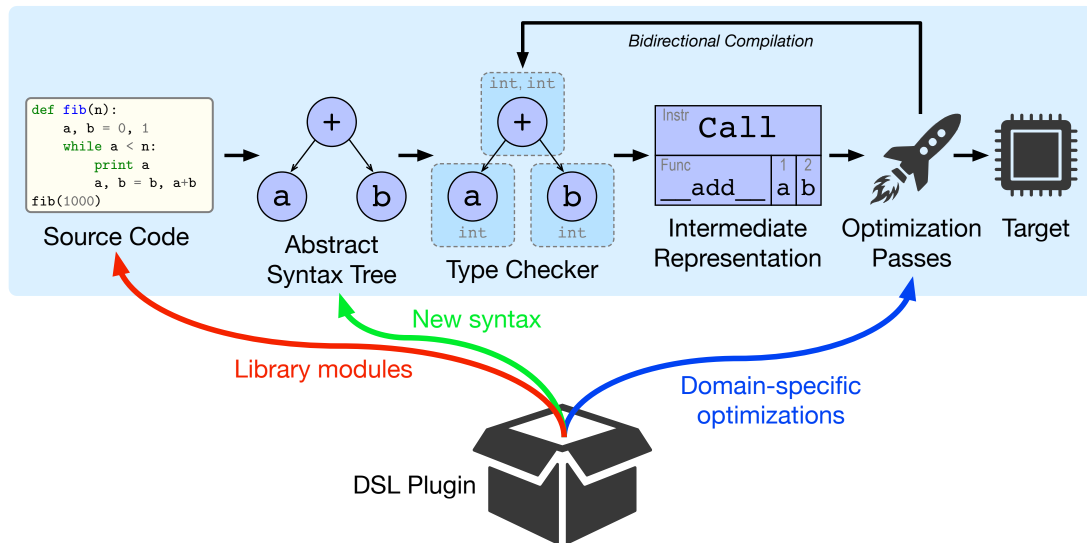


Figure 2.7: Codon's compilation pipeline. Compilation proceeds at first in a linear fashion, where source code is parsed into an abstract syntax tree (AST), on which type checking is performed to generate an intermediate representation (IR). Unlike other compilation frameworks, however, Codon's is *bidirectional*, and IR optimizations can return to the type-checking stage to generate new IR nodes and specializations not found in the original program, which is required for several key optimizations we present in this work. The framework is "domain-extensible", and a "DSL plugin" consists of library modules, syntax extensions, and domain-specific IR passes.

- *Large portions of complex IR optimizations can be implemented in Codon.* and *new instantiations of user- or library-defined data types can be generated on demand.* For example, an optimization that requires the use of Codon/Python dictionaries can instantiate the `Dict` type for the appropriate key and value types. Instantiating types or functions is a non-trivial process that requires a full re-invocation of the type checker due to cascading realizations, specializations and so on.

- *The IR can take full advantage of Codon's intricate type system.* By the same token, IR passes can themselves be generic, using Codon's expressive type system to operate on a variety of types.

The ability to instantiate new types during CIR passes is critical in our work. For example, creating a tuple `(x, y)` from given CIR values `x` and `y` requires instantiating a new tuple type `Tuple[X,Y]` (where the uppercase identifiers indicate types), which in turn requires instantiating new tuple operators for equality and inequality checking,

iteration, hashing and so on. Codon, however, makes this process seamless by calling back to the type-checker.

Passes and Transformations

```
class AddFolder : public OperatorPass {
  void handle(CallInstr *v) {
    auto *f = util::getFunc(v->getCallee());
    if (!f || f->getUnmangledName() != "__add__") return;
    auto *lhs = cast<IntConst>(v->front());
    auto *rhs = cast<IntConst>(v->back());
    if (lhs && rhs) {
      auto sum = lhs->getVal() + rhs->getVal();
      v->replaceAll(v->getModule()->getInt(sum));
    }
  }
};
```

Figure 2.8: Simple integer addition constant folder pass in Codon IR. This pass recognizes expressions of the form `<int> + <int>` (where `<int>` is a constant integer) and replaces them with the correct sum.

CIR provides a comprehensive analysis and transformation infrastructure: users write passes using various CIR builtin utility classes and register them with a `PassManager`, which is responsible for scheduling execution and ensuring that any required analyses are present. Figure 2.8 showcases a simple addition constant folding optimization that utilizes the `OperatorPass` helper, a utility pass that visits each node in an IR module automatically.

More complex passes can make use of CIR's bidirectionality and re-invoke the type checker to obtain new CIR types, functions, and methods, an example of which is shown in Figure 2.9. In this example, calls of the function `foo` are searched for, and a call to `validate` on `foo`'s argument and its output is inserted after each. As both functions are generic, the type checker is re-invoked to generate three new, unique `validate` instantiations. Instantiating new types and functions requires handling possible specializations and realizing other types and functions (e.g. the `==` operator method `__eq__` must be realized in the process of realizing `validate` in the example), as well as caching realizations for future use to avoid a blowup in code size.

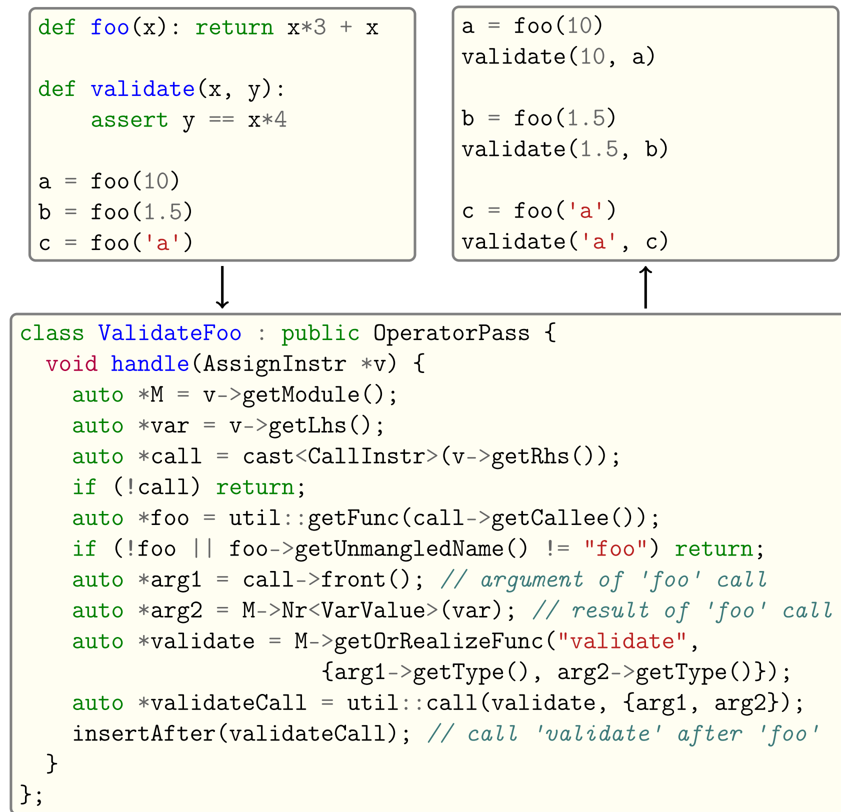


Figure 2.9: Example of bidirectional compilation in Codon IR. The simple pass shown in the bottom box searches for calls of function `foo`, and inserts after each a call to `validate`, which takes `foo`'s argument as well as its output and verifies the result. Both functions are generic and can take as an argument any type that can be multiplied by an integer, so the type checker is re-invoked to generate three distinct `validate` instantiations for the example code in the top left box, producing code equivalent to that in the top right box.

Extensibility

Due to the framework's flexibility and bidirectional IR, as well as the overall expressiveness of Python's syntax, a large fraction of the DSL implementation effort can be deferred to the Codon source. This has the benefit of making Codon DSLs and frameworks intrinsically interoperable—so long as their standard libraries compile, disparate DSLs can be used together seamlessly. Along these lines, Codon offers a modular approach for incorporating new IR passes and syntax, which can be packaged as dynamic libraries and Codon source files. At compile time, the Codon compiler can load the plugin, registering the DSL's elements.

2.3 Related Work

Framework	E2E security	Distr. trust	Distr. workload	Versatility	Matrix ops.	High-level	Full-stack opt.	AoT compile	Type check
NSFF	✗	~	✓	✓	✓	✓	✓	✓	✓
SMC (MP-SPDZ / MPyC)	✓	✓	✗	✓	✓	✓	✓/ ✗	✓/ ✗	✓/ ✗
HE (HEFac. / EVA)	✓	✗	✗	✗	✗	✓	✗	✗	✗
Sequre & Shechi	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 2.1: Survey of the current strategies for non-secure and secure distributed computing. NSFF stands for the non-secure federated framework. End-to-end security 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.

Non-secure federated frameworks (NSFFs) [54, 62, 115, 120] 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 [111,164]. 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.

Several HE-based compilers have been proposed [68, 151] to facilitate the development of applications on encrypted datasets. They target tasks such as machine learning inference [15–17, 25] or general analytics [36, 44, 150] and are typically built on top of HE schemes [22, 31] 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 [36,43] optimizes circuit evaluation to reduce ciphertext maintenance overhead, while HECO [150]

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.

The first comprehensive survey on SMC frameworks [73] measured the expressiveness, accessibility, level of security, and functionality of 11 different frameworks [18, 46, 51, 56, 105, 116, 129, 146, 153, 159, 160] across three simple applications. Today, there are more than 23 available SMC frameworks [2, 3, 9, 11, 18, 24, 26, 45, 46, 51, 56, 76, 89, 92, 105, 116, 129, 136, 146, 153, 159, 160, 166], each offering a different SMC flavour regarding accessibility, security, functionality, or performance. SMC compilers and frameworks [73] typically offer more versatile computations compared to HE but require interactions between computing parties. In SMC, the input data is secret-shared among multiple computing parties that interact to compute on top of 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 [136] and MP-SPDZ [89] transform high-level Pythonic code into SMC-enabled applications. The latter also automatically optimizes the underlying source code at compile-time. However, despite these and other developments in the SMC domain [73], 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 [117] (MHE) have been proposed for specific applications such as machine learning [60, 134, 163], PCA [58] and genomics [32]. 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 distribute the workload and leverage efficient local operations while ensuring security by encrypting any shared data and emulating a centralized execution by aggregating local intermediate results when needed. 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, developers manually make these decisions through an iterative and repetitive process, often overlooking performance and security implications and leading to convoluted codebases.

Chapter 3

Sequre

Privacy concerns present a key hurdle in genomic data-sharing efforts. As mentioned above, genomic data leaks are not only irreversible because one's genetic sequence cannot be changed, but their potential harm also extends to the genetic relatives of the individuals whose data is leaked. Also, traditional approaches to privacy protection, such as de-identification and access control, as described in policies such as the Health Insurance Portability and Accountability Act (HIPAA) in the U.S.A. and the Personal Information Protection and Electronic Documents Act (PIPEDA) in Canada, provide limited guidance for responsible sharing of genomic data due to inability to fully de-identify such data. Furthermore, novel privacy attack surfaces continue to be discovered, exacerbating these concerns [50, 80, 135, 142].

Recent advances in privacy-enhancing technologies offer a promising approach for mitigating the privacy concerns associated with data sharing [13,33]. These technologies generally enable computation on private data—in an encrypted form—without disclosing the sensitive information to anyone involved. A prominent such approach is secure multiparty computation (SMC) [52], which distributes the private data to multiple computing parties in a form that does not reveal any sensitive information to either party but allows all parties to interactively carry out the desired computation without revealing the underlying data. As the private data is kept confidential throughout the analysis, this approach allows private data held by multiple parties to be securely and jointly leveraged without disclosing the raw data. Recent studies have demonstrated the practical applicability of SMC for a range of computational genomics and biomedical research workflows [34, 61, 77, 82, 83, 86].

However, the practical application of SMC has been stymied by the high cost of developing efficient SMC protocols with minimal computational overhead. The

distributed nature of SMC implies that each data operation, such as the multiplication of two secret numbers, needs to be performed in a coordinated manner across different parties, increasing the complexity of the computation compared to its non-secure counterpart. This overhead can make even simple algorithms many orders of magnitude (e.g., 100× or more) slower than their non-secure counterparts [128]. Furthermore, existing non-secure pipelines cannot be easily ported to secure environments since SMC frameworks typically require (i) a near-complete reimplementation of existing algorithms using only low-level SMC routines and (ii) a manual optimization of the algorithms to improve SMC performance. Such optimization can obfuscate the original intent of the code and sacrifice readability and maintainability, thus making the subsequent development and code reviews (needed for security assurance and compliance) tedious and prone to oversight. These limitations are in part because the existing SMC frameworks (e.g. [166]) are implemented as custom libraries of low-level SMC operations that cannot be efficiently composed and optimized [140], or as domain-specific languages with *ad hoc* syntax and limited expressiveness that increase the difficulty of SMC pipeline development and maintenance [73].

We introduce Sequre, a Python-like, high-performance domain-specific framework for developing secure SMC algorithms. Sequre uses standard Python syntax and semantics to ease the development of secure pipelines and the transformation of existing code into SMC equivalents. Its novel optimizations utilize Codon's *intermediate representation* (CIR) of the Python source code (i.e., a logical representation of a program's execution flow that can be statically analyzed) to remove unnecessary computation and select the best SMC routines and optimization approaches for each computational step. As a result, Sequre enables high-performance and simple codebases that do not require extensive SMC-related modifications.

We demonstrate Sequre's performance and usability by employing it to implement two genomics pipelines, namely the genome-wide association study (GWAS) [34] and metagenomic binning [106, 124], and one pharmacology pipeline, namely the drug-target interaction (DTI) inference [77]. To our knowledge, a secure SMC protocol for metagenomic binning has not been previously developed. We implemented each pipeline in only 80–160 lines of high-level Python code, the functionality of which is equivalent to the original algorithm. We achieved up to 7× reduction in code length compared to the existing state-of-the-art pipelines. Furthermore, the overall execution time of these pipelines was reduced by 3–4×, and the network utilization was also 17% lower. Where possible, we also compared Sequre to an existing Python-based

SMC framework, PySyft (SyMPC) [166], and showed that various machine learning tasks can be performed 2× faster while providing stronger security guarantees. We also micro-benchmarked Sequre's performance and compared it to ten existing SMC frameworks using a standard benchmark suite [73]. Sequre achieved the best runtime performance in most cases while being one of the most accessible frameworks and offering, for the most part, stronger security guarantees.

We expect Sequre to enable practitioners without expertise in SMC and cryptography to easily write efficient SMC algorithms for various genomics workflows. Also, while Sequre has been tested exclusively on large-scale genomic applications, it applies to non-genomic fields as well (see Chapter 5). Furthermore, the improved readability and usability of Sequre programs can simplify the sharing and maintenance of secure workflows. Thus, Sequre could facilitate the use of secure computation technologies and, as a result, broaden data sharing and collaboration efforts in biomedicine.[1]

3.1 Sequre at a Glance

Sequre is a high-performance, Secure Multiparty Computing compiler framework consisting of an SMC library and a set of domain-specific compile-time transformation and optimization passes for detecting and optimizing SMC operations in the source code. It follows the fundamental principle that the compiler can optimize the code automatically through custom analyses and compile-time optimizations that utilize domain-specific knowledge. As mentioned in previous chapter, this principle has been successfully applied to languages and optimization toolkits in various domains, including GPU computing [118], image processing [126], deep learning [8], tensor computing [91], parallel computing [101], and recently, bioinformatics [138, 140]. Secure SMC procedures—i.e., blocks of code meant to operate on securely shared data in a distributed fashion—are annotated via the "`@sequre`" decorator. Code annotated with this decorator is automatically converted to an SMC routine by transforming each operation to the SMC equivalent implemented in Sequre's standard library. This library supports common arithmetic, Boolean and linear algebra operations, and shares the same semantics as Python's NumPy library [72]. The transformed source code is then statically type-checked and transformed to Codon's intermediate representation (CIR), a starting point for all further analysis and optimization passes, where custom

[1]This chapter was recently published in Genome Biology [144, 145].

SMC-related optimizations are applied to reduce network utilization and runtime performance. Optimized IR is then translated to LLVM IR and subsequently handled by the LLVM framework [97] that applies additional general-purpose performance optimizations and facilitates the final machine code generation. The final result is a highly optimized executable that a set of computing parties can deploy to perform the desired computation on the private data, as well as the original high-level source code that can be easily understood by the involved entities.

Sequre's compiler pipeline and automated optimization techniques are illustrated in Figure 3.1. For example, the compiler will expand a series of expressions into a polynomial and invoke the optimized SMC routine for polynomial evaluation to minimize the network overhead. If the expanded polynomial is highly complex, the compiler will perform a static code analysis to find hotspots where auxiliary SMC computation can be cached—a procedure that would otherwise require significant manual intervention. Additionally, Sequre looks for specific algebraic patterns (e.g., secure matrix and fixed-point arithmetics) and substitutes them with SMC-efficient alternatives.

3.2 Sequre's SMC Framework

Sequre uses additive secret sharing-based SMC [52], which represents each data value as an element in a finite algebraic structure. This structure is typically a finite (Galois) field or a $\mathbb{Z}_{2^k}$ ring. While $\mathbb{Z}_{2^k}$ rings tend to have better performance due to native integer operations, they support a limited range of arithmetic operations [38] (e.g., protocols that require a modular inverse are not supported). Existing SMC frameworks typically provide support only for a single algebraic structure. In contrast, Sequre supports both and is able to convert between different representations to achieve better performance (section 3.2.1).

There are many flavours of SMC that differ in the desired security and performance guarantees, often trading off one for the other. Sequre uses additive (arithmetic) secret sharing with a trusted dealer under an honest-but-curious security model [52], which we view as a balanced option achieving both practical efficiency and a meaningful level of security. Our framework builds upon the SMC framework used by the prior work on secure GWAS [34], which combines a variety of key SMC building blocks from the literature (e.g. for fixed-point arithmetic and comparison protocols) into a unified SMC library. Sequre supports joint computation among any number of computing

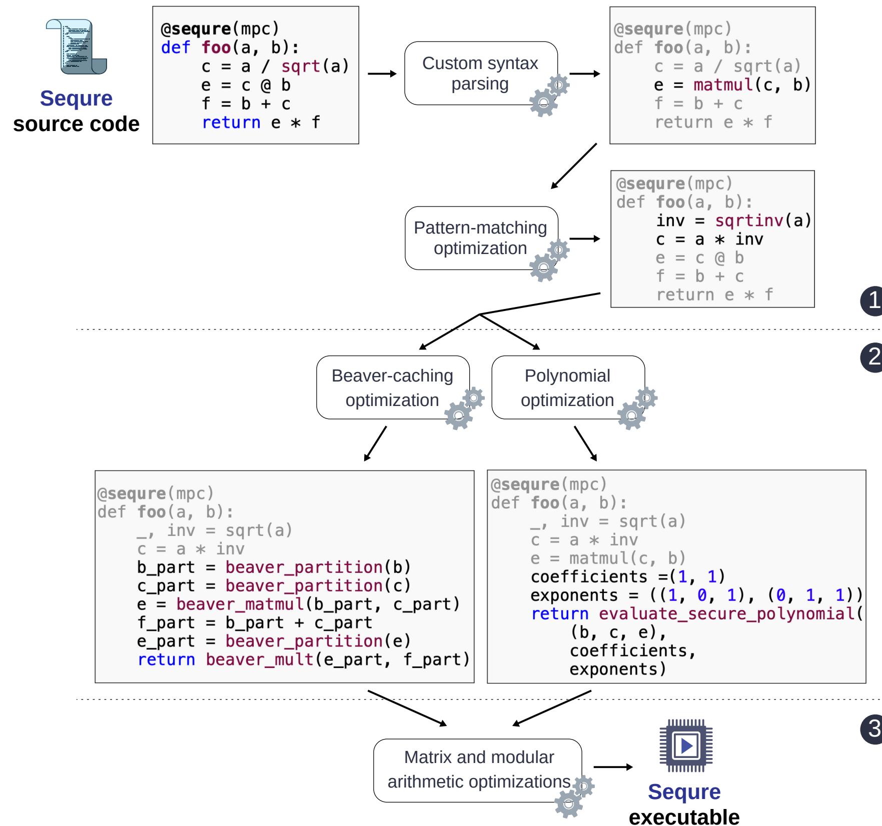


Figure 3.1: Sequre's automatic compiler optimization workflow. Sequre transforms an analysis pipeline written in standard Python language into equivalent secure SMC programs through a set of compiler analysis and optimization modules. These optimizations include: (1) replacing common code patterns with more efficient equivalents under the SMC setting; (2) restructuring arithmetic expressions to minimize redundant computation and the network overhead incurred by the *Beaver partitioning* operations; and (3) applying a set of low-level performance optimizations, such as faster modulus and matrix operations.

parties (at least two, not including the trusted dealer). It remains secure against arbitrary collusion among parties as long as the trusted dealer and at least one other party (participating in additive secret sharing) remain honest. Since Sequre provides a general framework for SMC implementation and optimization, it allows end-users to extend and adjust the existing SMC protocols, as well as to implement novel SMC protocols that could provide different security guarantees if desired.

3.2.1 SMC-specific Optimizations

Sequre performs five compile-time SMC-specific optimizations: two network load optimizations, one code generation optimization, and two low-level performance optimizations. All of them are automatically invoked via the custom `@sequre` decorator (Figure 3.1).

Network Optimizations

Sequre provides two network optimizations to reduce the communication rounds and the overall network bandwidth in the secure multiplication routine. This routine, by default, uses a generalized form of the Beaver multiplication triples [10], which were originally devised for the secure multiplication of two elements but later generalized for computing higher order polynomials [34]. Such computation necessitates constructing the so-called *Beaver partitions* of the secretly shared data beforehand.

A naïve implementation of this procedure calculates fresh Beaver partitions in each multiplication for each variable. However, the Beaver partitions of the past variables that have not been modified can be reused, which can significantly improve the performance of the overall pipeline (Beaver partitioning is expensive both in terms of network utilization and computational overhead) [34]. Such reuses are typically implemented manually, and require developers to carefully inspect the code and avoid redundant partitions when needed. While this manual optimization can significantly reduce the runtime of the protocol, it complicates the development process and makes the underlying code complex and less readable. For example, an optimized secure SMC implementation of QR factorization or a simple linear regression [34] can become $10\times$ longer than the non-secure program implementing the same algorithm due to manual optimizations. Sequre addresses this problem by automatically tracking the multiplication operations and finding the places where Beaver partitions can be reused through static code analysis methods described below.

Beaver caching optimization As mentioned earlier, the SMC framework of Sequre requires Beaver partitions of the input variables before each multiplication. Once generated, these partitions can be cached and reused in subsequent multiplications as long as the variable remains unchanged. Furthermore, some operations, like addition and public scalar multiplication, are invariant to Beaver partitioning (i.e., when adding two numbers, it is enough to add the corresponding partitions to obtain the

partitions of the sum). Hence the partitions of the sums can be propagated and reused in subsequent multiplications, thus avoiding redundant computation and communication across different multiplications (Figure 3.1). Sequre automates the partition reuse by statically analyzing arithmetic expressions that operate on secretly shared variables. Generated partitions are cached and reused by traversing the binary expression tree for each target expression, and by labelling the redundant sub-expressions, identified either directly or through propagation. For a set of expressions that share the same variable, the variable is partitioned in only one expression; other expressions in the same set reuse the cached partitions. Sequre also tracks changes to variables and invalidates a cached partition whenever a change occurs.

To be more specific, our variant of secure multiplication $[x][y]$ necessitates obtaining Beaver partitions $(x - r_1, [r_1])$ and $(y - r_2, [r_2])$ beforehand. Note that the parts of Beaver partitions are known to all computing parties, while the actual values of x and y remain hidden as $[r_1] \in S$ and $[r_2] \in S$ are randomly generated and secret-shared between the parties. It suffices to compute the Beaver partitions only once for each unique secret share (i.e., a variable) and reuse the partitions in the subsequent multiplications. Partitions will be invalidated if a new value is assigned to the partitioned variable (e.g., `a = 5` will invalidate all partitions of the variable `a`), and the variable will be re-partitioned if it is utilized as a multiplicative factor in the subsequent expressions.

```
...
a = b * c
# a should be immediately partitioned here
d = a + 1
e = a + 2
# Partitioning e and d here could have been avoided
# if a was partitioned immediately after it was instantiated above
f = e * d
...
x = m * n
# However, x should not be partitioned here as it would be unnecessary
return x
```

Figure 3.2: Beaver caching example.

Constructing the binary expression tree and its subsequent static compile-time analysis is implemented as Codon's Intermediate Representation pass in over 500 lines of code. The propagation and utilization of partitions during runtime are implemented as a part of the Codon library.

Polynomial optimization Arithmetic expressions that contain operations that rely on Beaver multiplications (multiplication, addition, and exponentiation) can often be represented in a generalized polynomial form:

$$f_m(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{m} c_i \prod_{j=1}^{n} x_j^{p_{ij}} \ , \ c_i, x_j \in \mathbb{R}, p_{ij} \in \mathbb{N}_0.$$

Certain types of these polynomials (e.g., a low-degree polynomial) can be efficiently evaluated by the generalized Beaver partitioning approach [34], where the partitions of the input variables are calculated only once. However, manually formulating such polynomials from the existing expressions is a cumbersome task that often requires large-scale code changes. Even when the polynomials are identified, this procedure is hard to implement manually. To address this problem, Sequre automatically transforms a block of expressions to a polynomial at compile-time and generates the secure generalized polynomial evaluation procedure on top of it (Figure 3.1). This way, Sequre minimizes the overall network overhead for evaluating a set of expressions. Note that while our polynomial evaluation routine minimizes the rounds of communication, it introduces an offline performance overhead that can grow exponentially with the degree of the polynomial. For this reason, Sequre limits the degree of polynomials for this optimization and resorts to the Beaver caching technique to further divide the expression into smaller components if the full expansion is deemed to be infeasible.

Similar to Beaver caching analysis, Sequre extracts polynomials from the binary expression tree of the code block by recursively expanding its multiplication and exponentiation nodes (Figure 3.3) to reach the form of a generalized polynomial. The polynomial coefficients and exponents are then parsed from the expanded binary expression tree and passed to the polynomial evaluation method. Binary expression tree analysis, transformation, parsing, and scaffolding for the polynomial evaluation method are done at the compile-time and implemented as a CIR pass in over 800 lines of C/C++ code. Secure polynomial evaluation, as a part of Sequre's standard library, is then executed at runtime.

Multiple Algebraic Structures

Enabling the SMC protocols to operate on top of $\mathbb{Z}_{2^k}$ rings instead of Galois fields is a common practice because the rings are generally faster on modern computer architectures [38]. Sequre supports running most of its standard library routines in

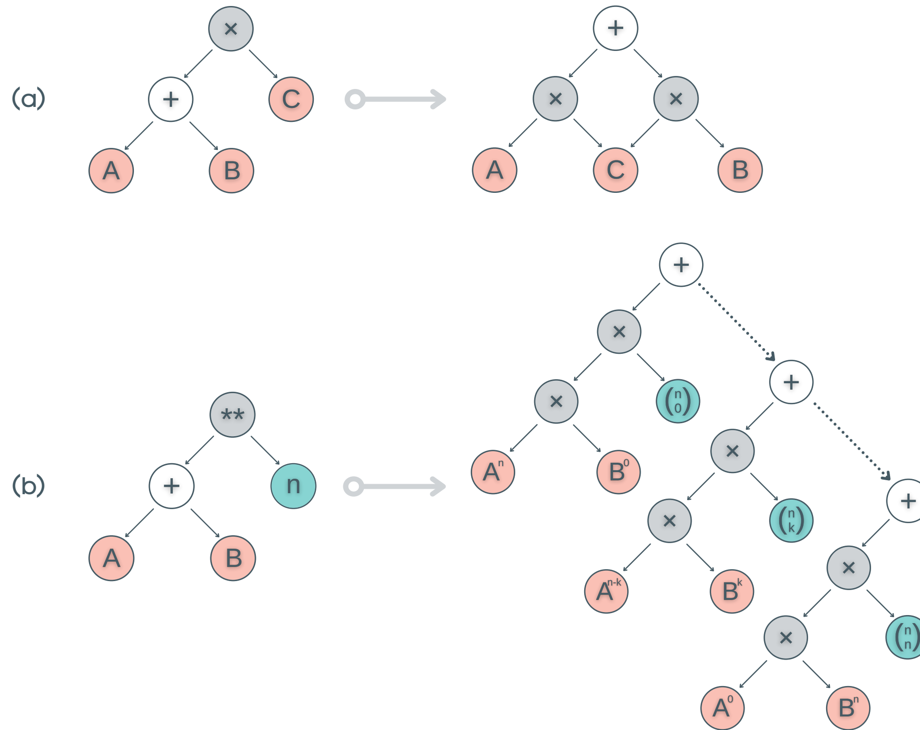


Figure 3.3: Binary expression tree term expansion for (a) multiplication and (b) exponentiation. The two transformations will be applied to the nodes recursively until the structure of the tree reaches the shape of a generalized polynomial.

both algebraic structures (see Figure 3.4).

Generalized beaver triplets	On field (Z_p)	On ring (Z_{2^k})
Multiplication	✓	✓
Exponentiation	✓	✓
Polynomial evaluation	✓	✓
Table lookup	✓	✗

Fixed-point arithmetic	On field (Z_p)	On ring (Z_{2^k})
Truncation	✓	✓
Division	✓	⇄
Square root	✓	⇄

Oblivious data structures	On field (Z_p)	On ring (Z_{2^k})
Oblivious array	✓	✓
Oblivious dictionary	✓	✗

"Boolean's"	On field (Z_p)	On ring (Z_{2^k})
Bitwise operations	✓	✗
Comparison	✓	⇄

Linear algebra	On field (Z_p)	On ring (Z_{2^k})
Householder	✓	✓
QR factorization	✓	✓
Tridiagonalization	✓	✓
Eigen decomposition	✓	✓
Orthonormal basis	✓	✓

✓ - Fully supported
✗ - Not supported
⇄ - Supported by resorting to fields

Figure 3.4: Finite Galois fields and $\mathbb{Z}_{2^k}$ rings support in Sequre.

However, some SMC protocols such as fixed-point value comparisons, division, and

square root calculation—all operating on top of the fixed-point values—exclusively work with finite Galois fields in Sequre. Hence, when operating on top of a $\mathbb{Z}_{2^k}$ ring, Sequre needs to internally switch to a finite Galois field when calling such procedures. This switch is not free, because the difference between the sizes of the two algebraic structures is publicly subtracted from the input and then publicly added to the output of each procedure. If the sum of secret shares does not overflow the predefined size of the algebraic structure, the switch will change the accuracy of the procedure. The change in accuracy is equal to $\tau = (p - r)/2^f$, where p, r, and f are the size of a field, ring and the fractional portion of the fixed-point number, respectively. Sequre's default fixed-point arithmetic setup evaluates this value to $\tau = 1/2^{32}$. Note that the comparison between the two fixed point values a and b will yield incorrect results on $\mathbb{Z}_{2^k}$ rings if the difference between a and b is within the $(0, \tau]$ half-range. Finally, the size of f is configurable in Sequre, which can coerce τ to be arbitrarily small and hence minimize the practical chance of the error. Hence, setting the size of the finite field to be as similar as possible to the size of the ring, and increasing the fractional part of the fixed-point numbers—something that Sequre does by default—is needed for the improved accuracy (note that the sizes of the field and the ring must differ since the only case where Z_p is equivalent to Z_{2^k} is when $p = 2$—a value too impractical for arithmetic-circuits based SMC setups). Note that the procedures that are sensitive to any errors (e.g., table lookup that underpins the Sequre's implementation of oblivious dictionaries, or bitwise operations) operate only on finite fields in Sequre.

Code Generation and Performance Optimizations

Sequre takes advantage of the specific nature of SMC algebraic operations to introduce domain-specific optimizations that can improve elementary operations such as division and square root. For example, the standard iterative technique for calculating a square root via SMC (based on Goldschmidt's algorithm) additionally outputs the inverse square root as a by-product. Thus, an efficient way of dividing by the square root of some number b (e.g. normalizing a vector by its norm) is to multiply the numerator by the inverse square root of b, as opposed to invoking both square root and division operations [34]. Sequre compiler identifies this and other similar patterns and replaces the expression with an equivalent that can be more efficiently evaluated under SMC (Figure 3.1; Figure 3.5). In addition, Sequre tracks the diagonal matrices in the expressions using a vector representation, and replaces the matrix op-

erations involving them with efficient vector-based operations that avoid unnecessary calculations involving the off-diagonal zeros. Sequre also optimizes compute-intensive operations like matrix multiplication by using a specialized, LLVM-optimized version of the Strassen algorithm.

```
a = b / sqrt(c)
# Transformed to:
... a = b * sqrt_inverse(c)

a = fixed_point_share / 3.4
# Transformed to:
... a = fixed_point_share * to_fixed_point(1 / 3.4)
... a = truncate(a)

a = non_fixed_point_share * 3.4
# Transformed to:
... a = to_fixed_point(non_fixed_point_number) * to_fixed_point(3.4)
... # There is no need for truncation

a = fixed_point_share * 3
# No transformations needed.
```

Figure 3.5: Pattern-matching transformations.

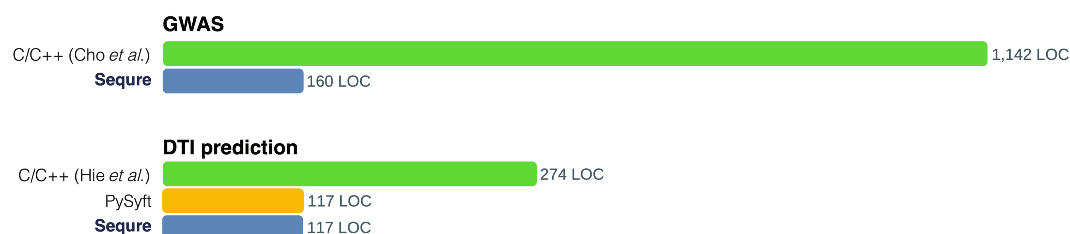
We also note that the modulo operator is one of the most commonly used operators in secret sharing-based SMC protocols. Unfortunately, LLVM ships with a generic implementation of the modulo operation, which often offers substandard performance. For this reason, Sequre introduces a new and efficient calculation of fixed modulus operations (as the modulus remains fixed in SMC protocols), yielding up to 40% of performance improvement over the default LLVM modulo operator (Figure 3.6).

3.3 Applications

In the following, we demonstrate three applications of Sequre in different domains, including medical genetics (GWAS), pharmacology (drug-target interaction prediction), and metagenomics (taxonomic binning). The first two are reimplementations of recently published SMC solutions [34, 77], while the last (metagenomic binning [106, 124]) illustrates a novel SMC implementation of a common biomedical task from scratch.

A typical secret-sharing-based SMC pipeline currently consists of three general stages: (i) data holders prepare the data locally and share it with computing parties

A Code size (smaller is better)



B Runtime speedup (higher is better)

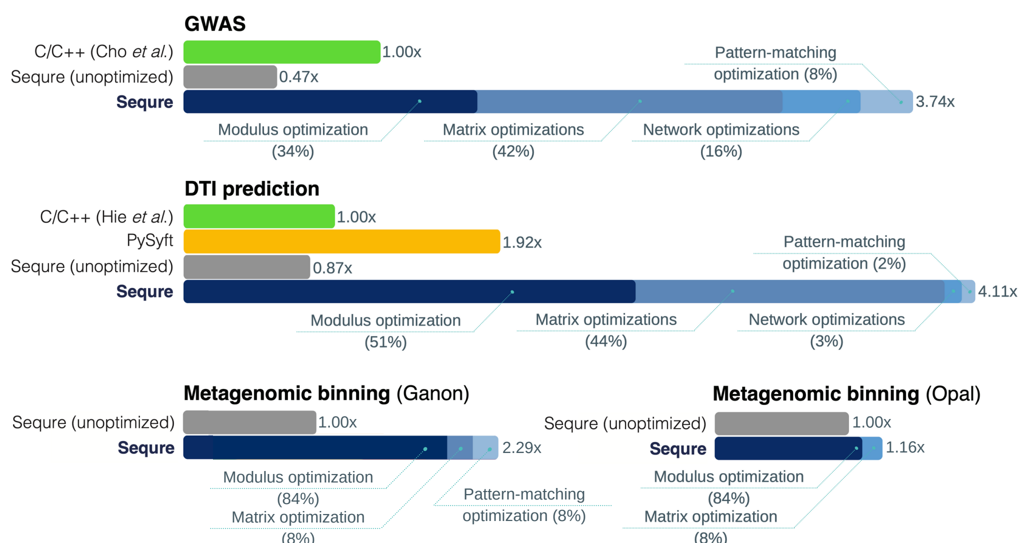


Figure 3.6: Sequre's usability and performance improvements in three biomedical applications. Sequre's automated code transformation and optimizations reduce (**A**) code complexity (number of the lines of code [LOC] in the implementation) and (**B**) runtime (execution time in seconds). We show Sequre's improvements in these metrics in three applications: genome-wide association studies (GWAS), drug-target interaction (DTI) prediction, and metagenomic binning. For metagenomic binning, we consider two recent algorithms, Ganon [124] and Opal [106]. We implemented the analysis pipeline for each application in Sequre and compared it with the version without the compiler optimizations, as well as implementations in existing frameworks where applicable (C/C++ and PySyft). Note that the C/C++ baselines refer to the recently published, manually optimized SMC implementations of GWAS [34] and DTI prediction [77]. There is no prior SMC implementation for metagenomic binning. Contributions of individual optimization modules in Sequre are shown in different colors within a bar. Sequre generates high-performance SMC programs while allowing them to be easily and compactly written in standard Python language.

over secure channels, (ii) computing parties combine the data from data holders and execute the desired pipeline on top of it in a secure manner, and (iii) results are broadcasted back to the end-users and data holders either publicly or privately, where they are reconstructed locally after pooling the outputs from all computing parties. Computation in the first and third stages is local and private, while the second stage is executed in a distributed manner over multiple computing parties.

Sequre adds support for the online distributed computing of a second stage. For local (offline) computing, it internally shifts to either Seq [139] for bioinformatics-specific computation or raw Codon for general-purpose tasks.

Our benchmarks measure the following aspects for each application: (i) *expressiveness* as the number of lines needed for implementation (lines of code, LOC) with all the comments, logging, line breaks, and blank lines removed, (ii) *network utilization* as the number of bytes exchanged between the computing parties during the execution, and (iii) *runtime* as the number of seconds needed for execution. The offline pre-processing and secret-sharing are of negligible runtime in our threat model and scale of data and were excluded from the total runtime in all experiments. Where possible, we also keep track of the network-related and SMC-specific parameters, such as the number of Beaver partitions (see Section 2.1.1), which measure the effectiveness of our network-related compile-time optimizations. Furthermore, as secret-sharing-based SMC paradigms operate within a finite algebraic structure such as the Galois field or $\mathbb{Z}_{2^k}$ ring, the size of the algebraic structure directly impacts the performance and security of the deployed protocols—the larger fields or rings increase the security at the cost of the performance (see Section 3.2.1). Sequre defaults to 192-bit long structures.

For each application, we ran Sequre in two modes: one with the optimizations disabled (see Section 3.4 for their overview), and the other with the optimizations enabled. To ensure fairness, all benchmarks were also conducted in a single-threaded mode and, where possible, coerced to use the same security constraints. Finally, the benchmarks were executed in two network environments: on a single machine, with the network latency reduced to a minimum, and within a local-area network (Section 3.5).

3.3.1 Secure Genome-Wide Association Studies

One of the first practical demonstrations of secure computation in genomics was for genome-wide association studies (GWAS) [14, 34, 61, 86]. GWAS aims to identify genetic variants that are statistically correlated with phenotypes of interest (e.g., biological traits or disease status). For example, Cho et al. [34] introduced an SMC solution written in more than 1,000 lines of carefully optimized C/C++ code, encompassing all the standard steps of a GWAS: quality control filtering (to control missing rates, allele frequencies, and Hardy-Weinberg equilibrium), population stratification analysis through principal component analysis (PCA), and linear regression-based association tests. Despite extensive optimizations, this pipeline was estimated to require 80 days to securely perform GWAS on a million individuals and half a million single-nucleotide polymorphisms (SNPs), illustrating the overhead of SMC for complex operations such as GWAS.

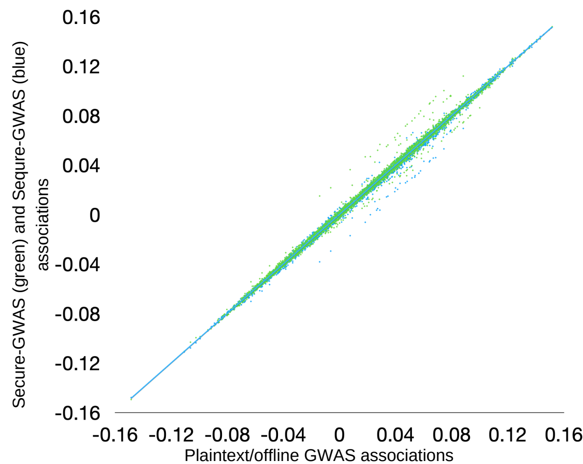


Figure 3.7: Accuracy comparison of Secure-GWAS (C++; green) and Sequre-GWAS (Sequre; blue) against the offline GWAS implementation (diagonal). The error mean is 9.3×10^{-4} for Secure-GWAS and 8.5×10^{-4} for Sequre-GWAS.

We developed Sequre-GWAS, a reimplementation of the aforementioned SMC-based GWAS pipeline [34] in Sequre. Our implementation consisted of only 160 lines of high-level Python code, representing over 7× reduction in code length. We observed a 3.7× decrease in the overall runtime of GWAS with a comparable network utilization on the lung cancer dataset from [34]. Further, we observed consistent speedup factors for varying dataset dimensions. Based on this, we estimate a runtime of 3 weeks for a million-individual study using Sequre-GWAS, in contrast to the nearly 3 months reported in the original publication. The analysis results retain the same accuracy even after the transformations and optimizations are automatically applied to the pipeline by the Sequre compiler.

The breakdown of the performance improvement achieved by Sequre is shown in

Figure 3.6. SMC-related optimizations, such as caching the intermediate results of secure multiplications and adjusting the precedence of operators to be executed in an SMC-friendly manner—both manually optimized in prior work—are automatically performed by our compiler. Not only Sequre reproduced all such optimizations in the original code, it also found 4% more hotspots in the original codebase that could be further optimized. These optimizations alone, together with modular arithmetic optimizations, improved the overall GWAS runtime by 1.64×. Finally, automatic conversion from the finite (Galois) field-based to the ring-based SMC protocols resulted in an additional 2.3× speed-up.

In population stratification analysis, a specific randomized principal component analysis (PCA) is employed for better performance [34]. It utilizes several secure SMC variants of linear algebra routines, such as QR factorization and eigendecomposition—all of which are fully reimplemented in 66 lines of Sequre code—6× fewer than the original C/C++ codebase. These procedures are individually benchmarked in Table 3.1. The specifics on the randomized PCA and the Cochran-Armitage trend test employed in the third step are available in Cho et al. [34] (protocols 32 and 33 in Supplementary Note 9).

Table 3.1: Expressiveness, runtime, and network statistics for the original Secure-GWAS (C/C++) implementation (bottom), together with its linear algebra subroutines (top), and their Sequre reimplementations. GWAS was ran on top of a lung cancer dataset reduced to 3,000 individuals and 30,000 SNPs [34]. Sequre is presented with and without the compile-time optimizations enabled (i.e. network, pattern-matching, matrix and modulus optimizations). Runtime is given in the hh:mm:ss format. Bandwith is expressed in megabytes (MB).

			Runtime		Network	
		LOC	On field	On ring	Bandwidth	Beaver partitions
Lin. alg.	C/C++	395	0:00:59	N/A	455	1.31 mil.
	Sequre (w/o opt)	66	0:01:20	0:00:57	499	1.97 mil.
	Sequre (w/ opt)	66	**0:00:44**	**0:00:37**	**378**	**0.96 mil.**
GWAS	C/C++	1,142	3:21:08	N/A	10,254	2.41 mil.
	Sequre (w/o opt)	117	7:09:02	2:26:07	151,309	3.61 mil.
	Sequre (w/ opt)	117	**2:02:22**	**0:53:43**	**9,887**	**2.34 mil.**

GWAS was benchmarked on top of a genotype matrix of 3,000 individuals and 30,000 SNPs, with the top five principal components being selected in the population

Table 3.2: Runtimes (hh:mm::ss) and speed-up of Sequre over C/C++ baseline measured on top of a lung cancer dataset for different number of individuals and SNPs. Note that the speed-up is preserved as the dataset size increases.

Indiv.	SNPs	Secure-GWAS	Sequre-GWAS	Speed-up
1,000	10,000	0:20:16	0:05:07	3.95×
	20,000	0:40:20	0:10:13	3.93×
	30,000	1:00:42	0:14:54	4.07×
2,000	10,000	0:40:20	0:09:48	4.11×
	20,000	1:14:52	0:19:41	3.80×
3,000	10,000	0:55:49	14:45	3.78×

stratification analysis and ten covariates added to it before the trend test. Sequre needed 70,225 fewer Beaver partitions than the baseline implementation. As a result, the total network consumption was reduced by 4%. The overall speedup (3.7×) when compared to the original implementation is primarily due to Sequre's modulus and matrix optimizations (Section 3.2.1), and shifting to $\mathbb{Z}_{2^k}$ ring instead of the Galois field. Also, some portions of the pipeline, such as the Cochran-Armitage test, benefited more than the rest from these optimizations: the network utilization was reduced 28×, while the runtime was reduced 2× when compared to the non-optimized counterpart. The runtime improvement is projected to be more pronounced in the wide-area network setups because our local test environments had low network latency. Finally, the overall accuracy of Sequre-GWAS is preserved or even marginally improved when compared against the original C++ implementation (Figure 3.8). More precisely, when compared to the offline GWAS implementation (Figure 3.7), the error mean (i.e., the mean of the difference in accuracy over all points) is 8.5×10^{-4} for Sequre-GWAS—a marginal improvement over the 9.3×10^{-4} that the original C++ implementation achieves. These marginal improvements are observable regardless of the SMC settings used: a Sequre version on rings with 20-bit field precision (theoretically, the least "accurate" mode as compared to the original implementation) achieves better accuracy (8.8×10^{-4}) than the "ideal" 30-bit C++ version on Galois fields (9.2×10^{-4}). To our understanding, this showcases no impact of SMC setting on accuracy at such small precision scales.

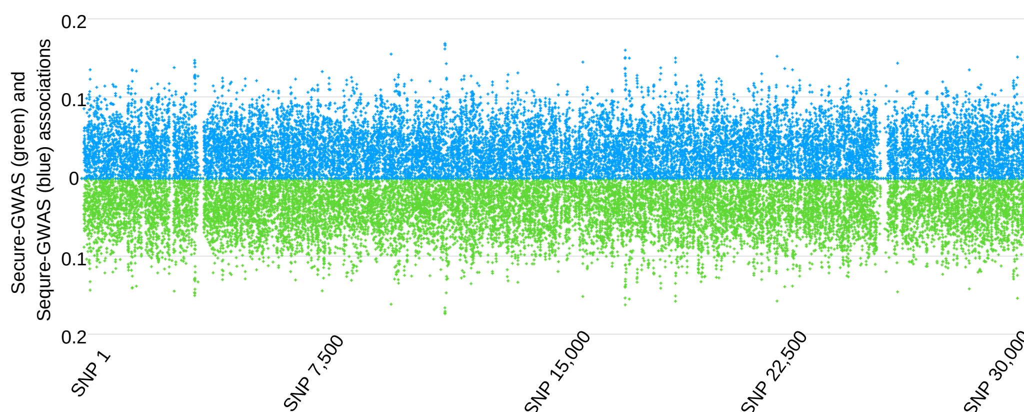


Figure 3.8: GWAS accuracy comparison between Secure-GWAS (C++; green) and Sequre-GWAS (Sequre; blue) on lung cancer dataset for 30,000 SNPs. Associations accuracy (χ^2 value on y-axis) was preserved in Sequre-GWAS. The two set of results are presented in a mirrored fashion.

3.3.2 Secure Drug-Target Interaction Prediction

Another promising application of secure computation is drug-target interaction (DTI) prediction. The goal of DTI prediction is to uncover novel interactions between drug molecules and putative protein targets. The inference method takes a set of compounds and a set of targets as input and outputs the probability of interaction between new compound-target pairs. While the feature representations vary across the existing methods, the inference is increasingly performed using neural networks [77, 99, 147, 155].

Unfortunately, many existing pharmacological datasets are held by labs and commercial companies that are unable to share data due to intellectual property concerns. This problem was recently addressed by a privacy-preserving DTI prediction pipeline that protects the individual training datasets from the involved entities [77]. The pipeline begins with the data preprocessing step, where the chemical compounds and proteins are encoded as feature vectors by the data holders and securely shared with the untrusted computing parties. The second step of the pipeline, i.e. secure model training, uses these features to securely train a neural network using SMC. The existing method for model training takes up to 4 days for one million drug-target pairs [77]. After the training, DTI inference continues in a similar fashion: the data owners encode their drug-target pairs locally and securely evaluate the pre-trained neural network on the new inputs to obtain the predictions.

To demonstrate that machine learning tasks, such as neural network training, can

be efficiently developed using our framework, we reimplemented the DTI training and prediction pipeline in Sequre (Sequre-DTI). We observed improvements similar to the GWAS task: 2–3× reduction in code length (from 274 to 117 lines of code), more than 4× faster execution times, and more than 34% reduction in network usage (Figure 3.6). For example, 4 days of training time could be reduced to less than a day. The impact of individual optimizations in Sequre-DTI followed the same patterns as in the GWAS example. We also implemented the DTI inference procedure in SyMPC (PySyft) [166], an existing Python framework specialized in deep learning SMC pipelines. While Sequre-DTI and the PySyft implementation were comparable in both accuracy and code length, the former was more than 2× faster (Figure 3.6). This improvement is despite the fact that SyMPC employs function secret sharing (more restrictive but theoretically faster than Sequre's approach) and 64-bit data types that incur smaller CPU and network overhead than Sequre's 128-bit data types [2].

Previous work (i.e. Secure-DTI) uses binary encoding of the chemical compounds and proteins as features. The simplified molecular-input line-entry system (SMILES) representation of the chemical compounds from STITCH and DrugBank datasets was converted to an extended connectivity fingerprint with diameter 4 (ECFP4) via JChem Base (version 17.28.0, 2017, ChemAxon, `http://www.chemaxon.com`) to obtain a binary feature vector from $\{0,1\}^{1024}$ for each compound [77]. Similarly, each protein was encoded as a one-hot binary vector corresponding to its Pfam family. There are 5,879 unique Pfam families for the protein identifiers in the STITCH dataset and 1,129 in the DrugBank dataset, yielding feature vectors of the same respective lengths. To simplify the use case and preserve fixed input size for the downstream neural network, the two representations are not combined (i.e. only one representation is used as a protein feature). Then, the corresponding feature vectors of each drug-target pair were concatenated and fed into an inference model: a neural network with a configurable number of layers and neurons. Rectified linear units (ReLUs) were used as activation functions, and a hinge loss was used as a target loss function. Finally, a mini-batch gradient descent was employed over a predefined number of epochs (further details are available in the supplementary notes of Hie et al. [77]). Our solution adopts the same workflow.

We used the protein identifiers from the STITCH dataset, resulting in input

[2]The size of integer was reduced to 128 bits in DTI to match the size of the state-of-the-art solution to which we compare [77].

feature vectors from $\{0, 1\}^{6,903}$. We also employed a neural network with one hidden layer of a hundred neurons with a zero dropout and 50 epochs of mini-batch gradient descent. We compared the original C/C++ implementation with our Sequre reimplementation, as well as the SyMPC/PySyft reimplementation of the same pipeline [166]. Table 3.3 shows the improvements of the Sequre version when compared to the C/C++ baseline and SyMPC/PySyft. The improvement trends and the breakdown of the improvement factors are similar to those observed in Sequre-GWAS.

Table 3.3: Expressiveness, runtime, and network statistics for the drug-target inference (DTI) implementations in C/C++, PySyft, and Sequre. Sequre is presented with and without the compile-time optimizations enabled (i.e. network, pattern-matching, matrix and modulus optimizations). SyMPC/PySyft employs a different SMC protocol that requires no Beaver partitions. Runtime is given in the hh:mm:ss format. Bandwith is expressed in megabytes (MB).

		Runtime		Network		
	LOC	On field	On ring	Bandwidth	Beaver part.	Accuracy
C/C++	274	0:09:18	N/A	3,436	1.04 mil.	0.90 ± 0.05
SyMPC/PySyft	117	N/A	0:04:40	**1,816**	N/A	0.90 ± 0.05
Sequre (w/o opt)	117	0:10:54	0:02:46	2,846	1.04 mil.	0.90 ± 0.05
Sequre (w/ opt)	117	**0:03:59**	**0:02:16**	2,651	**1.04 mil.**	0.90 ± 0.05

Also, we observed that SyMPC/PySyfy has better network utilization than Sequre (Table 3.3). This is because it uses 64-bit integers for SMC operations, compared to Sequre's 128 bits. However, smaller integers offer lower statistical security (the attacker observes 64-bit values instead of 128-bit). Still, Sequre is nearly two times faster than SyMPC. In terms of accuracy—calculated as the sum of true positive and true negative results divided by the total number of test cases—we observed the same trends in each framework, varying from 0.85 to 0.95 accuracy, depending on the initial random weights. Finally, we would like to note that we were unable to run any version of SyMPC or PySyft in a full network (LAN) setup; all versions at the time of writing offer only a proof-of-concept setup that can only be run on a single machine.

3.3.3 Secure Metagenomic Binning

To demonstrate that new analysis pipelines can be easily developed from scratch with Sequre, we turn to metagenomic binning, a central task in the analysis of microbiomes

for which practical SMC algorithms have not yet been developed. The goal of this task is to identify and quantify the organisms present in a sequenced metagenomic sample by classifying reads and assigning them to reference genomes, resulting in a "metagenomic profile" of the sample that can be used for various health-related tasks [49, 55, 108, 148]. Despite the need for a service where the user can upload their samples and receive the binning results with respect to the existing reference datasets (which may be proprietary), privacy risks have limited the utility of such services. For example, a sequenced metagenomic sample may include the host's genetic sequence, as well as viral sequences potentially indicating the infection status of the host [57]. In some cases, even the metagenomic profile alone can identify the individual and disclose information about the host's behaviour and environment [53].

There exists a wide variety of metagenomic binning methods and pipelines developed in the non-secure context [112]. Here, we used Sequre to implement two state-of-the-art solutions: Ganon [124], based on Bloom filters, and Opal [106], based on locality-sensitive hashing and machine learning classifiers. We refer to our SMC implementations of Ganon and Opal as Sequre-Ganon and Sequre-Opal, respectively.

Given a sample including reads to be classified, Sequre-Ganon extracts sequence features from the reads, and secretly shares them with the computing parties. The features are then securely queried against the index in the form of an interleaved Bloom filter [39]. The query results—the probability of a read belonging to a given bin (or reference)—are used to determine the most likely classification for each read. This procedure is implemented in a secure manner using oblivious array data structures [90], efficiently supported by Sequre. On the other hand, Sequre-Opal uses machine learning classifiers (logistic regression or support vector machines) for the same task. It also begins by encoding the reads as feature vectors, this time through Gallager coding [81], and secretly sharing the data with computing parties. We implemented SMC versions of both training and inference steps that use binary classifiers with hinge loss and stochastic gradient descent optimization.

Despite the complexity of both algorithms, Sequre-Ganon and Sequre-Opal are implemented as compact high-level Python programs in 113 and 80 lines of code, respectively. The code did not include any manual SMC-related optimization, thus being effectively identical to the non-secure counterparts. We evaluated our methods on the Opal benchmark dataset [106], which includes 10 reference bacterial genomes sequenced at 15$\times$ coverage as well as classification test data including 10,000 reads of length 65 bp. Sequre-Ganon took 18.5 hours to perform the classification—the

task that otherwise takes less than 10 seconds in an offline, non-secure setting, while Sequre-Opal took 3 hours for both training and classification. As a reference, the non-secure Opal run terminates in less than 10 minutes. The large runtime difference is due to Sequre-Ganon's use of Bloom filters, which incurs a considerable overhead in the SMC setting: privately answering a query takes $O(n)$ time, where n is the size of the data structure, compared to $O(1)$ time needed in the non-secure setting for the same task. The algorithm of Sequre-Opal is less affected by the algorithmic complexity changes induced by SMC protocols. Nonetheless, we note that Sequre still optimized the naïve SMC implementations of both methods. The compiler optimizations made Sequre-Ganon 2.29× faster, and Sequre-Opal 1.16× faster. Finally, we note that Sequre-Ganon and Ganon had identical accuracy as expected. This was also the case for Sequre-Opal and Opal based on a comparable choice of models and parameters.

Ganon uses k-mer index in the form of an interleaved Bloom filter (IBF) [39] for read classification. For each read, its k-mers are queried against the index, and the read is assigned to the bin with the highest count of matching k-mers. Reads with insufficient matches are filtered out via the k-mer counting (q-gram) lemma [85]. As the index is public during the inference, it is built through the original offline algorithm. An IBF is a matrix whose columns correspond to the Bloom filters for each available bin. That is, the i-th row of an IBF corresponds to the vector of i-th entries of the available Bloom filters. This design allows more efficient joint queries of the index as follows. For querying a k-mer t, for each available hash function h_i Ganon computes its hash $t_i = h_i(t)$ and obtains the t_i-th row of the IBF matrix. Then it performs an element-wise logical conjuction between the obtained rows to get the final result—the bit-vector in which a non-zero value at position j means that t belongs to a j-th bin in the index. Our secure implementation follows the same procedure with minor differences over the original implementation. First, the IBF is encoded as a list of integers, such that j-th element of the i-th row in the original IBF corresponds to the j-th bit of the i-th integer in the new IBF. The hashes $h_i(t)$ are computed for each k-mer t in offline fashion and then secretly shared between the computing nodes. This enables us to query the encoded IBF directly via an oblivious array getter (Section 3.4) in order to access the secretly shared value of $\text{IBF}[h_i(t)]$ for each secretly shared hash value. Finally, to compute the bit-wise logical conjuction between the $\text{IBF}[h_i(t)]$, we use the secure bit-decomposition protocol [41]—a part of the Sequre standard library—together with the secure element-wise multiplication of the bit-decomposed values. The result is a secretly shared bit-vector that contains

the same information as its offline counterpart, i.e. the list of bins that contain the target k-mer.

We evaluated Sequre-Ganon on the sample dataset from Opal [106] that contains 10 bacterial metagenomes. We built an index from these metagenomes. This dataset also comes with 10,000 microbiome labeled reads of length 65 that can be used for evaluation classification. The size of each Bloom filter in the index was 4,801,571 with the "technical number" of bins equal to 64 (note that the actual number of bins—10 in our case—differs from the technical number; see [124] for details). The k-mer size k was set to 19, and the number of hash functions in the Bloom filter was 4.

The second pipeline—Opal [106]—encodes reads as features through Gallager encoding [81] and uses support vector machine (SVM) classifier from the Vowpal-Wabbit library [95] for training and classification. Since the exact reimplementation of Vowpal-Wabbit and its various parameters is outside of the scope of this work, and since the performance of different classification algorithms is orthogonal to the problem itself, we instead used a stochastic gradient descent-based binary classification with hinge loss and L_2 regularizer—built in 13 lines of code—as a variant of a linear SVM. We also modified Opal to use the same binary classification algorithm for fair comparison. It is important to note that, unlike Ganon, Opal trains the classification model from the reads themselves, and not from the reference genomes. Thus, different coverage of the training set yields different classification model. Thus we evaluated Sequre-Opal with two training sets: $0.1\times$-deep and $15\times$-deep training sets simulated from the 10 reference metagenomes without errors.

Table 3.4 shows the improvements of Sequre-based implementations over their non-optimized counterparts. For reference, the performance and accuracy of the original non-secure implementations are also provided. In the case of Sequre-Ganon, the accuracy remains the same as the algorithms are identical between the non-secure and secure implementations. The accuracy is measured as the percentage of correct matches within the test set. The same holds for Sequre-Opal when compared to the Opal with linear SVM. However, when compared to Opal that uses Vowpal-Wabbit for classification, the accuracy varies and is either $3\times$ better or $1.4\times$ worse, depending on the depth of coverage.

Table 3.4: Expressiveness, runtime, network and accuracy stats for secure metagenomic binning implementations on top of 10 microbiomes. Opal was evaluated in two offline setups: the one based on Vowpal Wabbit (VWabbit) and the other based on custom Python implementation of linear SVM (PythonSVM). Ganon's offline implementation was done in the Seq language [139]. Sequre is presented with and without the enablement of compile-time optimizations (i.e., network, pattern-matching, matrix, and modulus optimizations). Runtime is given in hh:mm:ss format. Online bandwidth is expressed in MB. Beaver partitions are expressed in millions.

		LOC	Runtime (s)	Bandwidth	Beaver part.	Accuracy
Opal[1]	VWabbit	264	0:00:09	N/A	N/A	0.102
	PythonSVM	150	0:00:04	N/A	N/A	0.316
	Sequre (w/o opt)	113	0:01:16	370	2.68	0.316
	Sequre (w/ opt)	113	0:01:08	229	1.93	0.316
Opal[2]	VWabbit	264	0:09:07	N/A	N/A	0.760
	PythonSVM	150	0:09:44	N/A	N/A	0.540
	Sequre (w/o opt)	113	3:36:46	55,035	402.22	0.540
	Sequre (w/ opt)	113	3:07:05	33,941	290.49	0.540
Ganon	Seq-lang	80	0:00:07	N/A	N/A	0.872
	Sequre (w/o opt)	80	42:30:18	96,141	55.88	0.872
	Sequre (w/ opt)	80	18:29:23	81,186	55.72	0.872

[a]Opal with 0.1× coverage simulated reads from 10 microbiomes.
[b]Opal with 15× coverage simulated reads from 10 microbiomes.

3.3.4 Sequre and Other SMC Frameworks

Many other SMC frameworks have been introduced in recent years [2,9,18,45,46,89, 92,128,129,136,166] (see [73] for a survey of existing SMC frameworks). Each framework offers some novelty concerning security, expressiveness or performance. However, most of the available frameworks are hard to apply to computational genomics [13,33] due to the large scale of genomic datasets. Sequre emphasizes practicality and optimizes expressiveness and performance through various compile-time optimizations. It operates under what we view as middle-ground security constraints, providing a rigorous notion of security based on the properties of secret sharing while introducing additional requirements, such as an auxiliary party known as *trusted dealer* that generates correlated randomness to be used in the main protocol to accelerate the computation. This party, again, does not interfere with any of the private data (i.e. it never receives any data) but only shares the random data necessary for blind-

ing the intermediate revealed values. This model is also known as the server-aided model of SMC, and it requires collaborating entities to identify an independent, trustworthy actor to assume this role. However, many bioinformatics applications with large-scale datasets currently necessitate this modification to achieve practical performance. Nevertheless, disabling a trusted dealer at the expense of performance can be done and is part of future work. Additionally, as discussed in Cho et al. [34], the existing SMC scheme can be strengthened to support malicious SPDZ protocol [42] to allow both malicious and semi-honest adversaries, which relaxes the security assumptions by allowing the parties to deviate from the protocol at the expense of having worse performance. This also is part of future work.

Out of the available SMC frameworks, we selected and benchmarked ten mature and actively maintained SMC frameworks that are comparable to Sequre. We evaluated usability (as measured by code length), overall runtime and network utilization. It should be noted that not all frameworks operate under the same computational model and security constraints. For example, four evaluated frameworks use garbled circuits which support only two parties and the evaluation of Boolean circuits [52], while the other six operate under the same or, in some cases, slightly weaker models (e.g., t-out-of-n secret sharing where $t < \frac{n}{2}$) compared to Sequre. For comparison, we used the closest SMC paradigm and parameter settings across the tools to the extent possible.

In terms of usability, we found Sequre to be 3$\times$ more expressive on average, measured by the number of lines of code required to implement the benchmark (fewer means more expressive). Sequre is also one of the few frameworks that do not require learning a new language or framework: a single-line decorator is sufficient for Sequre to convert a normal Python code into its secure equivalent. Performance-wise, Sequre was on average 100$\times$ faster than the other frameworks (despite excluding the four outlier test cases where Sequre was from 1,600$\times$ to 32,941$\times$ faster than its counterparts). More precisely, Sequre was up to 250$\times$ faster in 9 test instances based on similar security models, and up to 1117$\times$ faster in 8 test instances involving comparison with a different SMC paradigm (garbled circuits). Even when compared with frameworks with more limited security guarantees aimed at faster performance, Sequre was 3–9$\times$ faster over 9 test instances. The only cases where Sequre was slower (2.5–5.5$\times$) were the four comparisons that evaluated the performance of oblivious data structures (e.g. secure array access), for which more efficient, specialized routines have been implemented in existing frameworks. Note that, given the modularity of Sequre, more

efficient subroutines such as these can be continuously integrated to further improve performance while maintaining expressiveness. The performance comparison between Sequre and ten other frameworks is provided in Table 3.5. We also note that because Sequre includes Seq as a plugin, it also provides a wide range of domain-specific features and routines for processing genomic datasets (e.g., sequence operations), which can be seamlessly integrated with the SMC portions of the analysis pipeline.

We adopted three benchmarks from the survey [73] (Table 3.5) and slightly altered them for better scalability:

- `mult3`: a simple multiplication and addition of three numbers;
- `innerprod`: an inner product between two vectors containing 100,000 elements; and
- `xtabs`: a cross-table aggregation (joining the two tables by key attributes and computing the sum of the values).

Each evaluated framework offers some novelty concerning the security, expressiveness or performance. Except for ABY, each framework utilizes pseudo-random generators with shared seeds to reduce the communication rounds between computing parties, just like Sequre. We found that other frameworks, excluding Jiff, do not employ a trusted dealer. Here is the overview of the evaluated frameworks and their results:

ABY is a two-party SMC framework embedded in C/C++ with semi-honest computing parties. It supports garbled circuits and conversion between them and other secret-sharing protocols. ABY is generally considered to be a fast but less expressive framework. Compared to Sequre, ABY is up to 6× less expressive on average in the selected benchmarks. It is also slower than Sequre: from 4× slower (`xtabs`) to 1,700× slower (`mult3`).

EMP is an SMC framework based on garbled circuits and embedded in C/C++. Sequre is roughly 6× more expressive on average. It is approximately 21× faster in `mult3` and 276× faster than EMP in `innerprod`. However, EMP is 5.5× faster than Sequre in `xtabs` due to the more advanced implementation of oblivious data structures.

Frigate is a C-like domain-specific language for SMC based on binary circuits. Unfortunately, we were unable to successfully run the benchmarks with the instructions

provided by the survey. Nevertheless, Sequre needed on average 4× fewer lines of code for the three benchmarks.

Jiff is a secret sharing and honest-majority-based SMC framework implemented in JavaScript for use in web applications. It required 4× more lines of code than Sequre to implement the three benchmarks. Sequre was also 12× faster in `mult3`. Unfortunately, the code in the survey was obsolete and could not be evaluated for the last two benchmarks (`innerprod` and `xtabs`).

MP-SPDZ is an SMC framework focused on providing support for various SMC variants. It comes with a custom compiler framework akin to Sequre that includes a novel Python-like domain-specific language. The source code is compiled to byte-code and executed on a custom virtual machine. However, the existing compiler pipeline is still in a proof-of-concept shape. It is implemented in Python, and it employs a set of static compile-time optimizations, such as loop unrolling and software prefetching to reduce network latency. We benchmarked an honest-majority non-malicious replicated secret sharing variant of SMC, optimized via pseudo-random zero secret sharing [7], in MP-SPDZ—one of the fastest variants supported by MP-SPDZ. This variant also leverages the pseudo-random generators to reduce the network communication on secret sharing. Computing on rings is relatively efficient in MP-SPDZ with the 2.5× speed-up over Sequre in the `xtabs`. However, it is still up to 9× slower than Sequre in the first two (`mult3` and `innerprod`) benchmarks, and 1.4× slower on field in the third benchmark, despite using a weaker and generally faster SMC scheme.

MPyC is a Python library for SMC based on Shamir's secret sharing with semi-honest computing parties. Being a Python library, its syntax and semantics are most similar to that of Sequre; however, it does not possess compile-time optimization capabilities. It has the same code complexity in `mult3` and `xtabs` benchmarks. Still, it required 2× more lines to implement the `innerprod` benchmark. Performance-wise, it was 9× and 14× slower in the `mult3` and `xtabs` benchmark respectively, and up to 250× slower when computing the inner product in the `innerprod` benchmark.

Obliv-C is a garbled-circuits-based SMC with semi-honest computing parties embedded in C. Sequre is up to 1000× faster in the `mult3` and more than 1100× faster in

innerprod benchmarks. In xtabs, despite leveraging a dynamic programming approach (unlike Sequre and other frameworks that use a straightforward brute-force algorithm), it was 2× slower than Sequre (and needed nearly 10× larger codebase for its implementation).

Oblivm is a garbled-circuits-based SMC with semi-honest computing parties embedded in Java. Sequre was significantly faster: from 30× in xtabs and 1,600× in mult3, to 32,900× in innerprod.

Picco is an SMC framework based on Shamir's secret sharing with semi-honest adversaries and implemented in C. While Picco was only 50% less expressive than Sequre, we were unable to run these benchmarks in practice.

Sharemind is an additive three-party SMC framework based on secret sharing with semi-honest adversaries and an honest majority setup. Sharemind provides its custom domain-specific language for SMC operations. The full version of the framework is not publicly available. However, prototyping the simple SMC programs is enabled through the use of their free SDK (https://sharemind-sdk.github.io/). Sequre was faster in all three benchmarks (28× in mult3, 1.33× in innerprod, and 50× faster in xtabs) while maintaining the same level of expressiveness, despite Sharemind's support of only the 64-bit $\mathbb{Z}_{2^k}$ rings.

Table 3.5: A cross-comparison between Sequre and ten state-of-the-art SMC frameworks. Frameworks were benchmarked for expressiveness (in terms of lines of code (LOC)) and runtime over multiple SMC setups. Some variants are not supported (marked with ⊥), while some could not be evaluated (marked with N/B) for the provided code samples.

			Runtime (ms)				
	Framework	LOC	128bit $\mathbb{Z}_p$	128bit $\mathbb{Z}_{2^k}$	64bit $\mathbb{Z}_p$	64bit $\mathbb{Z}_{2^k}$	GC[1]
mult3	ABY	20	⊥	170	⊥	170	⊥
	EMP	25	⊥	⊥	⊥	⊥	**2.1**
	Frigate[2]	19	⊥	⊥	⊥	⊥	N/B
	Jiff	20	⊥	⊥	1.2	⊥	⊥
	MP-SPDZ	**4**	1.0	0.9	0.7	0.6	⊥
	MPyC	8	⊥	⊥	0.9	⊥	⊥
	Obliv-C	11	⊥	⊥	⊥	⊥	100
	Oblivm	10	⊥	⊥	⊥	⊥	160
	Picco[2]	6	N/B	N/B	N/B	N/B	⊥
	Sharemind	**4**	⊥	⊥	⊥	2.8	⊥
	Sequre	**4**	**0.2**	**0.1**	–	–	⊥
innerprod	ABY	30	⊥	520	⊥	900	⊥
	EMP	28	⊥	⊥	⊥	⊥	**4,700**
	Frigate[2]	18	⊥	⊥	⊥	⊥	N/B
	Jiff[2]	20	⊥	⊥	N/B	⊥	⊥
	MP-SPDZ	7	78	45	77	44	⊥
	MPyC	7	⊥	⊥	4,200	⊥	⊥
	Obliv-C	13	⊥	⊥	⊥	⊥	19,000
	Oblivm	21	⊥	⊥	⊥	⊥	560,000
	Picco[2]	6	N/B	N/B	N/B	N/B	⊥
	Sharemind	**4**	⊥	⊥	⊥	20	⊥
	Sequre	**4**	**24**	**17**	–	–	⊥
xtabs	ABY	50	⊥	210	⊥	200	⊥
	EMP	25	⊥	⊥	⊥	⊥	**9**
	Frigate[2]	45	⊥	⊥	⊥	⊥	N/B
	Jiff[2]	25	⊥	⊥	N/B	⊥	⊥
	MP-SPDZ	24	70	**20**	40	15	⊥
	MPyC	**9**	⊥	⊥	700	⊥	⊥
	Obliv-C[3]	140	⊥	⊥	⊥	⊥	100
	Oblivm	44	⊥	⊥	⊥	⊥	1,500
	Picco[2]	19	N/B	N/B	N/B	N/B	⊥
	Sharemind	15	⊥	⊥	⊥	2,500	⊥
	Sequre	**9**	**50**	95	–	–	⊥

[a] GC: garbled circuits.

[b] Obsolete, or unable to run.

[c] Unlike other solutions that employ brute-force strategy, Obliv-C uses a dynamic programming approach.

3.3.5 Local-area Network Environment

So far, all the benchmarks were evaluated in a simulated network environment (with `AF_UNIX` sockets) on a single machine. Here, we evaluate the four main applications (GWAS, DTI, Opal, and Ganon) in a local-area network environment, where each computing party is deployed to a different machine.

Table 3.6: Runtime and network stats for secure GWAS, secure DTI, secure Opal, and secure Ganon. Sequre is presented with and without the enablement of compile-time optimizations (i.e., network, pattern-matching, matrix, and modulus optimizations). Note that we were unable to set-up and run SyMPC/PySyft on a LAN network. Runtime is given in hh:mm:ss format. Online bandwidth is expressed in MB. Beaver partitions are expressed in millions.

		Runtime	Bandwidth	Beaver partitions
GWAS	C/C++	2:43:10	10,254	2.41
	Sequre (w/o opt)	6:46:46	151,309	3.61
	Sequre (w/ opt)	0:48:40	9,886	2.34
DTI	C/C++	0:10:16	683	1.04
	SyMPC (w/o opt)	N/A	N/A	N/A
	Sequre (w/o opt)	0:13:17	567	1.04
	Sequre (w/ opt)	0:06:08	510	1.04
Opal	Sequre (w/o opt)	19:26:40	55,035	402.21
	Sequre (w/ opt)	15:54:22	33,941	290.49
Gan.	Sequre (w/o opt)	44:39:08	96,141	55.88
	Sequre (w/ opt)	20:30:52	81,186	55.72

3.4 Sequre's Standard Library

This section provides an overview of the SMC routines included in Sequre's standard library. Most of the procedures are adopted from external sources such as Cho et al. [34]; for more details, see the accompanying supplementary materials therein.

Sequre's standard library includes 24 algebraic protocols from Cho et al. [34]: secret sharing and revealing routines, private and public addition and multiplication, private fixed-point arithmetic, private division and square root calculation, private exponentiation and polynomial evaluation, private and public bitwise operators, private

and public comparison operators, and private linear algebra routines (householder transformation, QR decomposition, tridiagonalization, eigendecomposition and orthonormal basis calculation). It also includes naïve oblivious array getter [90] and oblivious dictionary with private indexing, private bit decomposition and bitwise addition [41]), private principal components analysis, linear support vector machine, regression module, and neural networks, some of which are explained below, while others later throughout the text.

Oblivious Data Structures

Data structures with secret indexing—also known as *oblivious data structures*—enable secret indexing of the secret data, where both the data and the indices by which the data is accessed are secretly shared. For example, the traditional secret-shared array has its values shared between the computing parties, but the indexing of the array is still public (i.e. we can still access the i-th element of a secretly shared array $[x]$ as $[x][i]$ at each computing party). In an oblivious array, the index i is also secretly shared, so there is a need to allow array access via the $[i]$ at each party (i.e. $[x][[i]]$). Sequre implements a naïve oblivious array getter [90]. Additionally, it supports a private dictionary with secret indexing, where both the keys and values of the dictionary can be private, by extending a table lookup routine from Cho et al. [34] to implement a secure getter.

Algorithm 1 Oblivious dictionary construction
INPUT: $D = \{k_i \rightarrow v_i\}_1^n$: a public dictionary given as a key-value mapping
OUTPUT: $[D] \in S^n$: a private dictionary represented as a private array of Lagrange coefficients

1: $c_1, c_2, \ldots, c_n \leftarrow \textsc{LagrangeInterpolation}(D)$
2: $[D] \leftarrow \textsc{SecretShare}(c_1, c_2, \ldots, c_n)$

Algorithm 2 Oblivious dictionary getter
INPUT: $[D] \in S^n$: a private dictionary represented as a private array of Lagrange coefficients
$[k] \in S$: a Secret-shared key to query the dictionary with
OUTPUT: $[v] \in S$: a secret-shared value that corresponds to k in D

1: $[k^2], \ldots, [k^{n-1}] \leftarrow \textsc{Powers}([k], n-1)$
2: $[v] \leftarrow [D] \cdot (1, [k], [k^2], \ldots, [k^{n-1}])$

Note that this approach makes the implementation of the dictionary setter difficult because we have to re-construct the dictionary whenever a new key-value pair is added to it.

Shared Tensor and Supported Operations

Sequre operates on top of tensors of arbitrary dimension. The SharedTensor class that implements them stores the secret additive share and auxiliary data, such as Beaver partitions, as n-dimensional arrays at each computing party. Furthermore, the compile-time optimizations apply only to SharedTensor expressions. Table 3.7 presents all secure operations supported for SharedTensor operands.

Table 3.7: Supported SharedTensor operations in Sequre. x and y are shared tensors. Some operations are supported only for 2-dimensional (matrices) or 1-dimensional (vectors) shared tensors.

Secure operation	Example usage
Element-wise addition / subtraction	x + y; x - y
Element-wise multiplication	x * y
Element-wise exponentiation	x ** c (c is a constant)
Element-wise division	x / y
Element-wise comparisons	x == y; x != y; x > y; x < y; x >= y; x <= y
Element-wise equare root	sqrt(x)
Dot product / Matrix multiplication	dot(x, y); matmul(x, y)
Element-wise absolute value	abs(x)
Max/min element (vector only)	max(x); min(x)
Argmax/argmin element (vector only)	argmax(x); argmin(x)
Householder transformation (matrix only)	householder(x)
QR factorization (matrix only)	qr_fact_square(x)
Tridiagonalization (matrix only)	tridiag(x)
Eigen decomposition (matrix only)	eigen_decomp(x)
Orthonormal basis (matrix only)	orthonormal_basis(x)
Element-wise bit decomposition	bit_decomposition(x, ...)
Element-wise bit-wise addition	bit_add(x, y)
Oblivious getter	oblivious_get
Polynomial evaluation	evaluate_poly

3.5 Benchmark and Hardware Setup

For benchmarking secure GWAS, secure DTI inference, and secure metagenomic binning, we used the test-case scenarios and benchmark configurations from [34], [77], and [106] respectively. For comparing Sequre against other frameworks, we adopted benchmarks from [73]: `mult3` for a series of secure multiplications, `innerprod` for inner product between two private vectors, and `xtabs` for a cross-tabular aggregation of oblivious arrays. The first two benchmarks were slightly adapted for scalability by extending a series of multiplications into a series of additions and multiplication and by setting the vectors' length in the second benchmark to be $100,000$ instead of the original 10. For the sake of precise measurements, all benchmarks were executed on a single machine where each party (computing or collaborating node) was run as a separate process. Network communication was established through inter-process communication sockets (`AF_UNIX`). Additionally, GWAS, DTI, Opal, and Ganon where evaluated on a local-area network with a different machine for each computing party. Single-machine results for GWAS were evaluated on Intel® Xeon® Platinum 8260 CPU at 2.40 GHz with 192 logical cores and 1 TB of RAM. All the other benchmarks, including the local-area network runs, were evaluated on multiple Intel® Core™ i7-8700 CPU at 3.20 GHz with 12 logical cores and 60 GB of RAM.

Lastly, Table 3.8 enlists all the tools, versions and the links to datasets used for benchmarks.

Table 3.8: List of tools, versions, and datasets used in each application (Secure-GWAS, Secure-DTI, Ganon, Opal) and SMC frameworks benchmarked against Sequre. GWAS lung cancer dataset was sampled into first 3,000 individuals and 30,000 SNPs. For tools that do not use a versioning scheme, the shortened commit hash of the version used is included.

	Tool	Version	Dataset
GWAS	Clang	14.0.0	Lung cancer dataset (accession: phs000716.v1.p1)
	GMP	6.2.1	
	NTL	10.3.0	
DTI	Clang	14.0.0	Reduced STITCH dataset: `https://bit.ly/3AuhaPn`
	GMP	6.2.1	
	NTL	10.3.0	
	Python	3.8.11	
	Syft	0.5.3	
	SyMPC	0.5.0	
Opal	Python	3.6.13	Opal dataset: `http://giant.csail.mit.edu/opal/data.tar.bz2`
	VowpalWabbit	8.11.0	
Ganon	Seq	0.10.1	Opal dataset: `http://giant.csail.mit.edu/opal/data.tar.bz2`
	Clang	14.0.0	
	SeqAn	3.1.0	
SMC frameworks	ABY	08baa85	N/A
	EMP	0.2.3	
	Frigate	4ef001b	
	Jiff	8ea565d	
	MP-SPDZ	0.1.5	
	MPyC	0.8	
	SyMPC	0.5.0	
	Obliv-C	2bacf04	
	Oblivm	50ed0fb	
	Picco	ee85c91	
	Sharemind	2017.12	
	Sequre	0.0.1	

Chapter 4

Shechi

Analyzing large and diverse datasets distributed among many 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 [63, 79] that restrict the sharing of sensitive data, have resulted in the creation of data silos that significantly hamper data analytics [152]. 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 [14, 32, 58–61, 93, 114, 163]. However, these methods come with several challenges that prevent their widespread adoption. One major hurdle is the difficulty in developing efficient, scalable, and secure applications capable of leveraging these methods for secure data analysis distributed between many parties. They demand expert-level proficiency in cryptography and require careful engineering efforts to achieve computing at scale. There are HE and SMC libraries, frameworks and compilers that aim to abstract the cryptographic complexity away from the user to improve expressiveness and simplify the development of secure applications [17, 43, 44, 46, 68, 73, 144, 146, 150, 151]. However, they often exclusively target either HE or SMC and lack support for promising hybrid frameworks such as MHE, which proliferate at mass collaboration and large data scales[1]. Many of these tools provide only limited optimization of high-level code be-

[1] To be more specific, MHE scales better than additive-secret-sharing-based SMC for all operations that require interaction between the computing parties (e.g. multiplication) and is generally worse

yond the use of efficient low-level primitives and still require significant cryptographic expertise to develop large-scale applications. Secure multiparty computation offers a provably secure means of protecting and sharing private data between multiple non-trusting parties. However, while it enables practical runtimes for applications that operate on large-scale data, it still comes with one important drawback that renders it impractical for use cases that involve many study participants. Namely, it requires network communication between all study participants, and for this reason, it scales quadratically with respect to the number of collaborating parties. On the other hand, while the computation overhead associated with the Homomorphic Encryption primitives often leads to impractical runtimes on large datasets, its multiparty variant (i.e. MHE) amortizes this cost by enabling computation on local, non-encrypted data at each party and independent, parallel computing between the parties. However, although 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, MHE 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 [58]. 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 is an extension of Sequre and it incorporates all of its functionalities. This extension is natural since the MHE scheme can be converted to Sequre's additive secret-sharing scheme and vice-

for operations that require no network communication. However, there are only a few such operations (addition and multiplication by a public value) in additive secret sharing, while in MHE, only matrix multiplication and collective bootstrapping require interaction between the parties.

versa [32]. Shechi enables end users 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. 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 [72] 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 [35] 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 performant, secure applications easily and constitutes an important step for the design and adoption of secure distributed solutions.

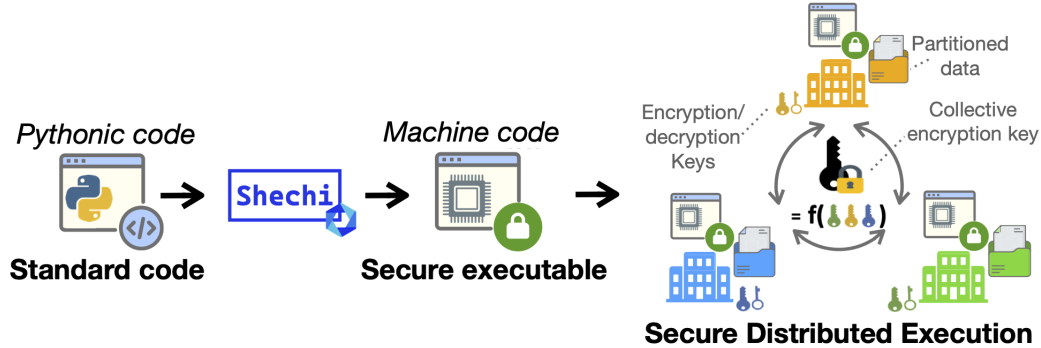


Figure 4.1: Shechi automatically translates standard centralized Pythonic code into a secure distributed workflow to be executed in a multiparty setting.

4.1 Problem Statement

We address the following problem: how to automatically apply a high-level Pythonic program to 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.[2] The

Code:

```
@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()
```

Input:

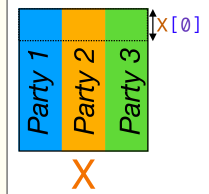


Figure 4.2: Example of high-level Pythonic code (left; forward pass of QR decomposition) to be securely executed on the input `X` split vertically among 3 parties (right).

[2]Our goal is to, ultimately, enable writing otherwise complex, secure workflows on large datasets distributed between many parties as a simple Pythonic code that abstracts away all security aspects and automatically optimizes the performance bottlenecks of the newly written applications. While hiding some of the optimizations behind a high-level syntax may seem to hinder the users' manual interventions (i.e. disallowing them to apply their custom optimizations or further improve the existing ones), that is not the case. Users can easily extend any part of our system and apply custom solutions even to the very low-level operators (our system is modular and easily extensible). Most importantly, however, regarding our automatic code optimizations, we automate only the common practices already manually employed by MHE developers. While automating these practices is

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.

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. 4.2 where the user defines a forward pass of QR decomposition [6]—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.

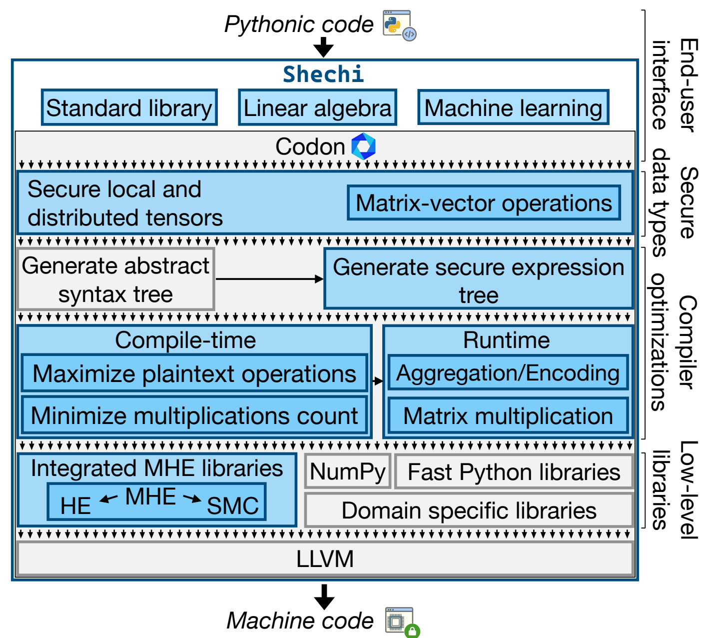


Figure 4.3: Shechi's overall structure. Shechi's novel components are highlighted in blue.

indeed a hard problem (likely an NP-hard problem), it is safe to say that the automatic solution is expected to produce a better result than a human intervention (the search space of the problem is exponential with respect to the number of operators). For this reason, we can state that our code optimizations do not sacrifice flexibility compared to the manually optimized code.

4.2 Shechi at a Glance

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. 4.1) while maintaining data confidentiality. For example, user can securely execute our lead example in Fig. 4.2 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.

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. 4.3, and we detail its components in the following sections.

4.3 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. 4.1). Shechi enables users to run the Pythonic procedure (`fqr`) from Fig. 4.2 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. 4.2), 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.

For efficient MHE operations, Shechi introduces specific optimizations (compiler passes step in Fig. 4.3) 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, like Sequre, 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 [150]. Thus, while Shechi aims to be a drop-in replacement for the existing Python pipelines, users still need to account for these differences.

4.4 Shechi's Secure Data Types

In a distributed scenario, multiple parties hold the input data (e.g., `X` in Fig. 4.2) in cleartext. In SMC contexts, such data is apt only for a limited amount of local, non-encrypted computation. After secret-sharing, data usually remains encrypted until the very end of the computation. However, computations on distributed, non-encrypted data can also result in only 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. 4.4). 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.4.4) and targeted optimizations (Section 4.5).

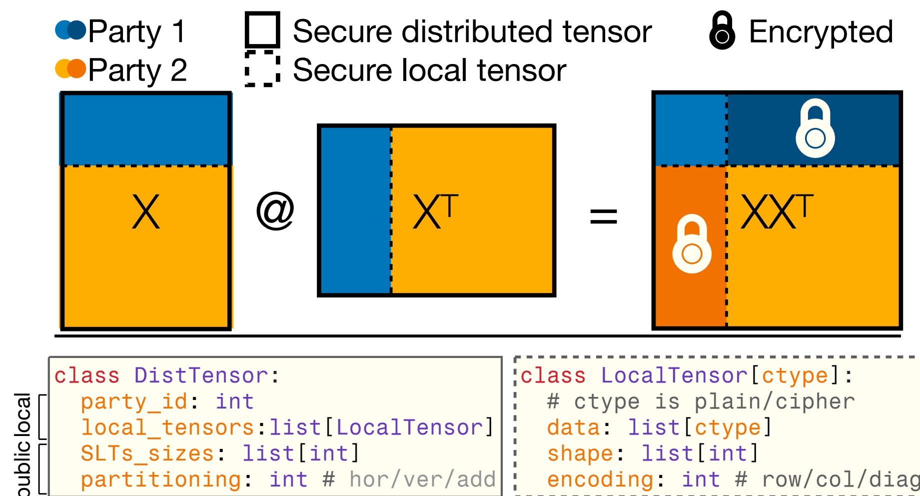


Figure 4.4: Secure distributed tensor example and definition. Matrix multiplication $X@X^\top$ is split among two parties. The data held locally at each party (bright parts) are used to compute the majority of the result in cleartext. Only a fraction of the data needs to be encrypted (parts with a lock). 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.

4.4.1 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. 4.4 is a local tensor. Similar to conventional tensor implementations like NumPy [72], Shechi keeps the n-dimensional data in a single, contiguous vector and lazily tracks the data

dimensions (*shape* attribute in *LocalTensor class* in Fig. 4.4). This approach 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 (Section 4.5).

4.4.2 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. 4.4). 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. 4.4 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. 4.4 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. 4.6 (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 (Section 4.5).

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 (Section 4.5.3).

4.5 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.

4.5.1 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. 4.2) is shown in Figs. 4.5 and 4.6.

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. 4.5). This tree encapsulates each expression operating on top of secure data and is used for the subsequent analysis and optimization of such expressions.

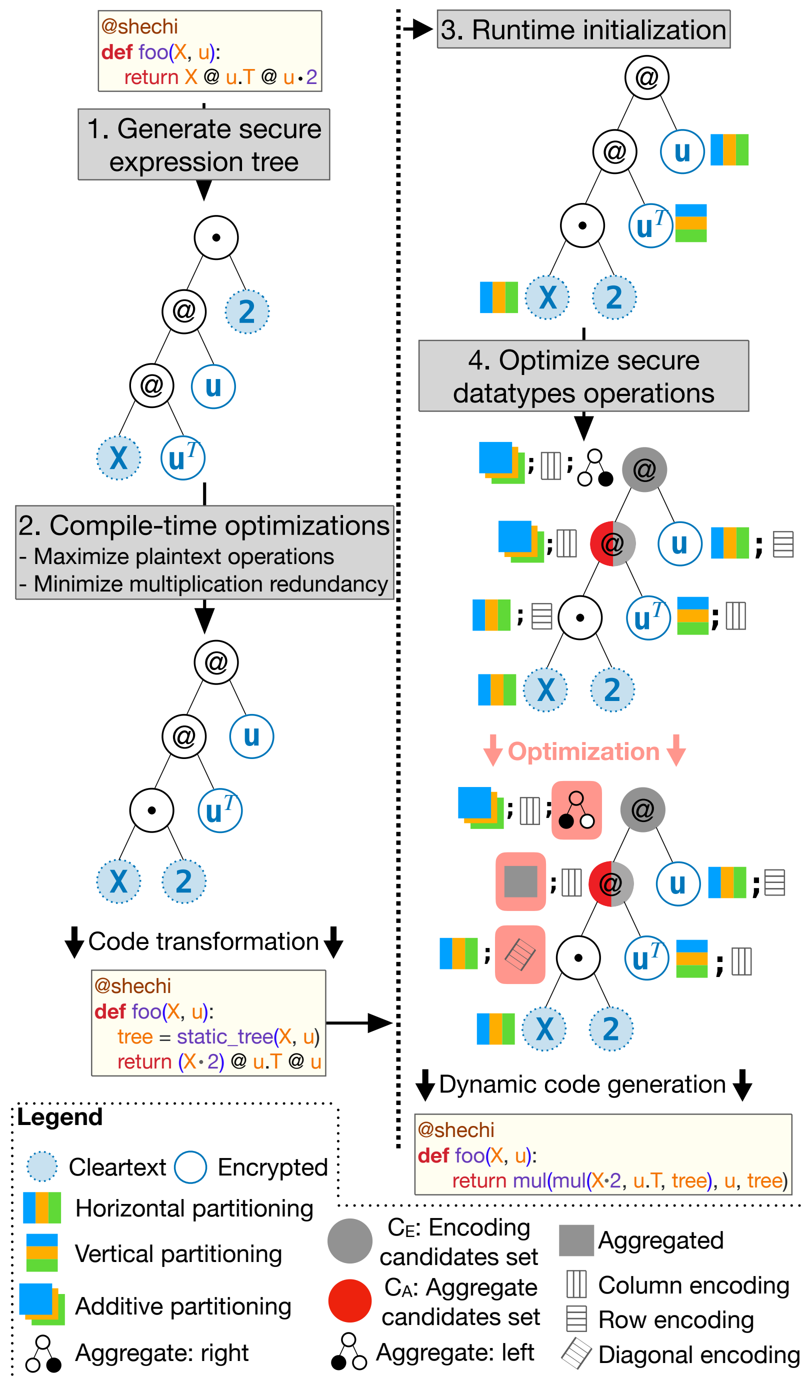


Figure 4.5: Shechi's code analysis and optimization passes. Shechi begins by generating a secure expression tree (Step 1). Optimizations on this tree can be either static (left) or dynamic (right). Compile-time (static) optimizations rewrite expressions for optimal MHE performance (Step 2). Runtime (dynamic) 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 is further explained in Fig. 4.6. 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).

Each leaf node in the tree corresponds to an operand, while inner nodes correspond to arithmetic operations. 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. 4.5). 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 the code snippet in Fig. 4.5). 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.

4.5.2 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. 4.2) 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). Some operations, such as advanced slicing or matrix multiplications, 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. Matrix multiplications, however, are of special importance since their aggregation choice also determines the partitioning of the result, impacting the performance of all downstream operations. For example, a subsequent multiplication of the result of `X @ u.T` with `u` (Step 2 in Fig.4.6) is executed either on a single partitioned matrix with the left aggregation strategy (Fig.4.6; *Aggregate:* ***left***), or on a full matrix shared additively at each party (Fig.4.6; *Aggregate:* ***right***). In this case, the latter is less efficient. Similarly, the choice of encoding

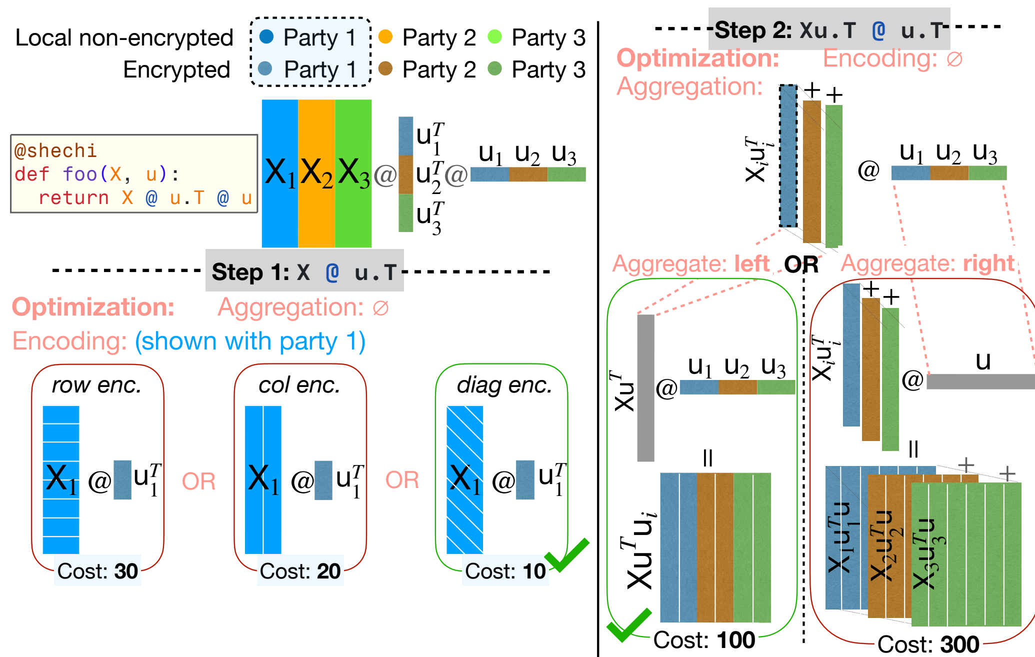


Figure 4.6: Aggregation & encoding Optimization. The performance of multiplication `X @ u.T @ u` depends on the choice of encoding of the non-encrypted operand (`X`) and the choice of aggregation strategy. In this example, Shechi first calculates `X @ u.T` (*Step 1*). As the shares of `X` are non-encrypted, 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.

for an unencrypted secure local tensor determines the encoding of the output and the cost of all subsequent operations. 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. 4.6).

Finding the optimal aggregation and encoding strategies is a combinatorial problem that can significantly impact a solution's runtime, especially in complex workflows with large-scale inputs (Section 4.7.4). Shechi finds the best aggregation and encoding strategy (Step 4 in Fig. 4.5; Fig.4.6) 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. 4.5). It then pinpoints all multiplication nodes in the tree for which one secure distributed operand needs to be aggregated (C_A; red in Fig. 4.5) 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. 4.5). 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 4.1). 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 3 in Section 4.8.1. 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 [12,28,43,133]. Without using the heuristic approach, an exhaustive search for both aggregation and encoding strategies would result in the overall complexity of $\mathcal{O}(2^{|C_A|} \cdot 3^{|C_E|})$, 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 $\mathcal{O}(c \cdot |C_A| \cdot 3^{|C_E|})$ 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 [70], 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. 4.2), 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. 4.5).

Minimizing multiplication redundancy. In MHE, as in standard HE, multiplications are at least one order of magnitude slower than additions (Tab. 4.1). 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 [71].

4.5.3 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 [58, 84]. 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 $\mathcal{O}(a \cdot c \cdot log_2(b))$ rotations, whereas a diagonally-encoded matrix requires $\mathcal{O}(a\sqrt{b})$ rotations [58]. The output encodings also vary based on the input encodings. For example, in Fig. 4.6 (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 (Section 4.5.2; Step 4 and *Dynamic code generation* in Fig. 4.5; Fig. 4.6). In addition to previously established matrix encodings and matrix multiplications [58, 84], 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 Section 4.8.2.

Automatic instantiation of MHE. Shechi adopts default parameters ensuring 128-bit security for MHE (Section 4.8.3), striking a balance across diverse applications. Unlike existing HE solutions that are limited by the absence of a practical bootstrapping routine, thus requiring parameter selection to be tightly coupled with the specific circuit under evaluation (Section 2.3), Shechi primarily bases this choice on balancing precision with performance. Users can easily override these parameters (e.g., by using Shechi's simple syntax to run a cleartext version of their application on local data to find the required level of precision).

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 [32, 58]. 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 (Section 4.9).

Optimizing ciphertext maintenance. The optimal placement of rescaling, relinearization, and bootstrapping operations (Section 2.3) is a difficult problem in general [12, 28, 43, 133]. For instance, optimal relinearization has been proven to be NP-hard [28, 43]. 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).

4.6 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 (Section 4.7), 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 [60,96,117] in Codon [141]. In addition to preserving existing performance optimizations from Lattigo, such as fast matrix multiplication via number theoretic transform [30], 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.

4.7 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.

4.7.1 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, which is the least favourable setup for Shechi. The scalability benchmark shows how our solution's performance improves as the number of parties and input data dimensions increase.

4.7.2 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 [89] and Sequre for additive secret-share SMC; HEFactory [75] and EVA [43] for HE; and Lattigo [96, 137] and SEAL [29] libraries for low-level MHE and HE, respectively. We also provide comparison against MPyC [136] and HECO [150] (whose front-end is currently deprecated) in Section 4.10.

Micro-Benchmarks

We demonstrate Shechi's low-level performance through micro-benchmarks in Table 4.1, evaluating basic algebraic operations on encrypted vectors using default parameters and a 128-bit security level for all tools (Section 4.8.3).

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

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 4.1: 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".

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 are performant enough and, at the same time, support MHE operations.

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 Section 4.7.2 and Section 4.7.3.

Basic Workflows

We implemented two applications commonly used to benchmark HE compilers [25,43, 150]—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. 4.7.

HE implementations use the recommended default parameters (Section 4.8.3). 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. 4.7, 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.

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`) [109], and genome-wide association study (`gwas`) [32]. 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 [94], 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

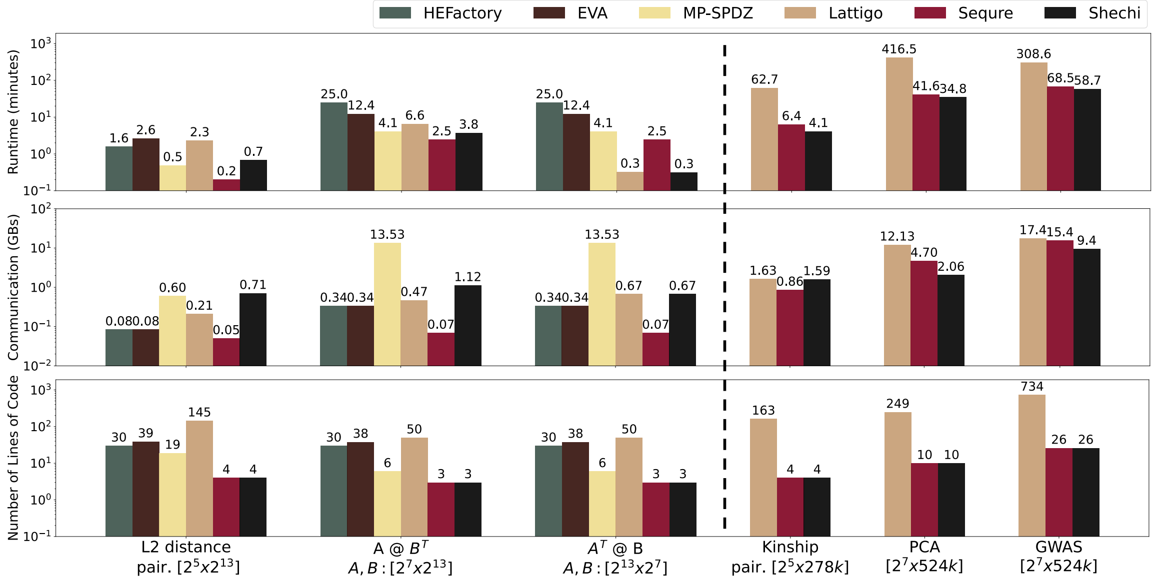


Figure 4.7: Runtime, communication and expressiveness comparison between Shechi and existing approaches. For distributed approaches, the input data are evenly and horizontally split among 2 parties. Communication is measured as the maximum number of bytes any party sends. Expressiveness is measured as the number of lines of code without comments, blank lines, and debug statements. For reference, offline, non-secure runtimes of Kinship, PCA, and GWAS were 0.34s, 118s, and 167s, respectively.

due to their low-level nature, as well as the scale of the input data and required computational depth. We refer to Section 4.8.3 for the detailed description of these workflows.

Shechi achieves on-par or better performance as existing manually-optimized secure solutions [32, 58] while improving expressiveness by up to two orders of magnitude. In Fig. 4.7, we notably observe a 15× speed improvement in `kinship` and 6× network traffic reduction in `pca` experiments when compared against the manually optimized version in Lattigo. This is notably due to Shechi's encoding optimization (Section 4.5.2), 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 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 (Section 4.5) to encode and optimize computation on top of wide matrices that are inherently abundant in this application. Complete code examples are shown in Section C.

4.7.3 Scaling

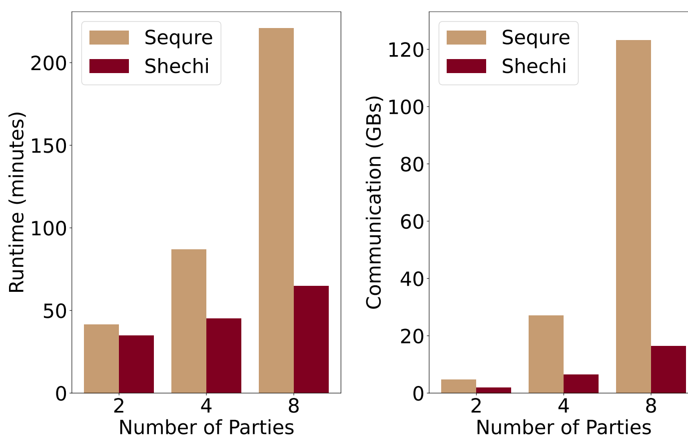


Figure 4.8: Shechi's scaling with the number of parties and samples. In this experiment, PCA is executed for the increasing number of parties and individuals, with a fixed number of 64 samples per party.

We show in Fig. 4.8 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. 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.

4.7.4 Necessity of Shechi's Optimizations

Shechi applies a set of compile-time and runtime optimizations to translate standard code into efficient distributed executions. The overall runtime of simple expression `a @ b.T * (d + c) * 2`, where `a`, `b`, `c`, and `d` are matrices of shapes (32×8192) for `a` and `b`, and (32×32) for `c` and `d` is shown in Figure 4.9 (right). The overall runtime is mostly impacted by Aggregation & Encoding optimization, which selects the most efficient matrix encoding and aggregation at each step of the algorithm (section 4.5). Whether any of the three optimizations (Aggregation & Encoding, plaintext prioritization, or multiplication redundancy removal) is applied is denoted by the presence of the respective coloured circle below each column. Disabled Aggregation & Encoding 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), and follows a fixed aggregation choice. Note that another common use case, where the developer relies on a single encoding, yields impractical runtimes in most of our applications. Shechi automatically chooses from a set of three encodings and various matrix multiplication methods, both of which are easily extensible. Even though we employed an ablation study on a small example, we note that we observed the same behaviour across all of our applications. For example, Figure 4.9 (left) demonstrates the performance gain achieved using Aggregation & Encoding optimization for executing PCA on the lung cancer dataset with an increasing number of samples. Lastly, we note that Shechi's search for the most efficient aggregation and encoding strategy takes less than one second in our experiments and is negligible compared to the overall runtime.

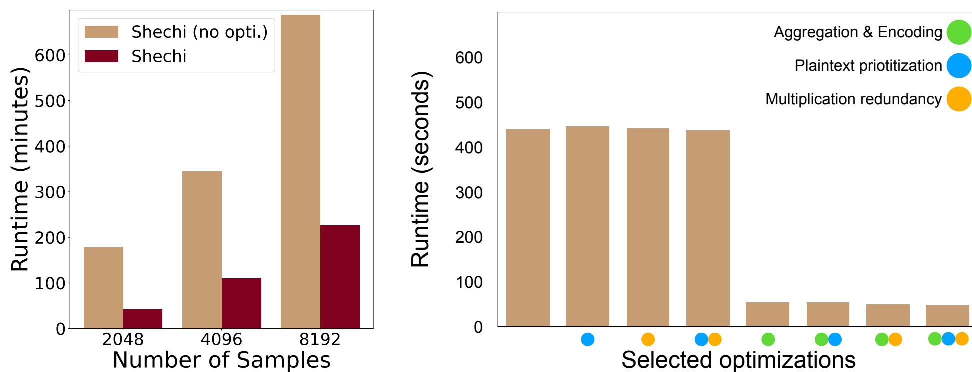


Figure 4.9: Shechi's ablation study evaluated for PCA workflow (left) and a simple application (right). Shechi's runtime worsens up to 10× with optimizations turned off.

4.7.5 Accuracy, Network, and Memory Usage

The security of CKKS, the HE scheme used in Shechi, 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 [37, 100]. In Fig. 4.10, we illustrate that Shechi is accurate 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× and up to 108× (with delays between 12 and 62 ms) when compared to our LAN setting, Shechi runtime increases only up to 2.2× 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.

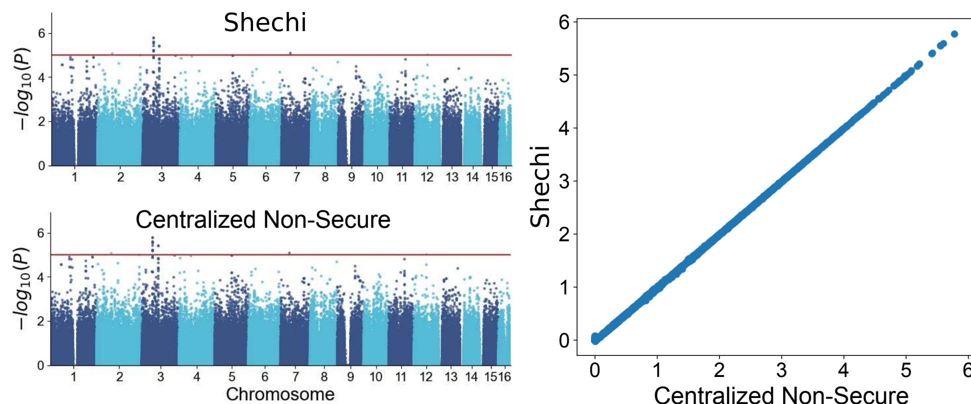


Figure 4.10: Shechi's secure distributed execution of GWAS produces results comparable to centralized non-secure execution, yielding similar association statistics, represented as the negative logarithm base-10 of the p-values.

4.8 Implementation Details

Here, we provide more details on our compile-time and runtime optimizations, cryptographic details, and complex workflows.

4.8.1 Aggregation and Encoding Optimization

Algorithm 3 describes 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.

4.8.2 Novel Matrix Multiplication Method

In addition to previously established matrix encodings and matrix multiplications [58, 84], 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 $0 \leq i \leq \min(a, b)$ and $0 \leq j \leq \max(a, b)$, instead of $d_i[j] = A[(i+j) \bmod a, j]$ with $0 \leq i \leq a$ and $0 \leq j \leq b$, in standard diagonal encoding. Multiplying a row-wise encoded matrix by a matrix with this encoding requires $\mathcal{O}(a \cdot \min(b, c) + log_2 \max(a, b))$ rotations and $\mathcal{O}(a \cdot \min(b, c))$ multiplications, or even just $\mathcal{O}(\min(a, b) \cdot \min(b, c))$ 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.

4.8.3 Cryptographic Details & Parameters

We used parameters that ensure 128-bit security level across all frameworks. For HE, we used parameters enabling 8192 slots with a default scale of 2^{34} for all applications and HE frameworks. Additionally, in HE, each rotation value requires a different rotation key. There are multiple strategies to solve this issue: (i) generate a predefined

Algorithm 3 Aggregation and Encoding Optimization
INPUT:
tree, *aggr_candidates*, *enc_candidates* - tree with bottom-top greedy approach and list of candidate nodes for aggregation and encoding strategies; c: parameter for hill climbing heuristic
OUTPUT: updated *tree*

```
1: cur_cost ← tree.cost() //estimated cost of tree eval.
2: for i ∈ 0,c (overlined) do
3:     tree_modified ← False
4:     for ni ∈ agg_candidate do
5:         new_tree ← tree.flip_aggr(ni) //change aggr. operand
6:         new_tree.resolve_tree() //update tree complement
       //start exhaustive search for encoding
7:         min_enc ← new_tree.encoding
8:         min_cost ← new_tree.cost()
9:         for j = 0 to 3^|enc_candidates| − 1 do
10:            new_tree.set_encoding(j)
11:            bet_cost ← new_tree.cost()
12:            if bet_cost < min_cost then
13:                min_cost ← bet_cost
14:                min_encoding ← new_tree.encoding
15:            end if
16:        end for
17:        if min_cost < cur_cost then
18:            tree_modified ← True
19:            cur_cost ← min_cost
20:            tree ← 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
```

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 Se-

qure) 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 most similar to that of Shechi's SMC module. 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 $t = 2$.

4.8.4 Complex Workflows Details

Principal Component Analysis (PCA)

PCA is a standard and widely-used dimensionality reduction algorithm that identifies a small set of orthogonal directions, or principal components (PCs), that capture the most variance in a high-dimensional dataset. We reimplemented the recent secure distributed variant by [58], which uses the randomized PCA algorithm [70]. We executed a joint PCA on 128 individuals split among two parties with $524,288$ features and relied on the randomized PCA [70] 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)

Identifying related individuals in genomic datasets is crucial in genomic workflows and has societal benefits, such as reuniting lost relatives [1,64]. Running this analysis across distributed databases has the potential to enhance privacy and expand service reach. We implemented a recent method [109] 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 [32] 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).

4.9 Users' Choices and 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 (Section 2.3). 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 [60], using an interval and degree parametrized by the user. Previous works [58,60] 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.

4.10 More Comparison Against Other Frameworks

We also compared against MPyC [136] and HECO [150]. 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

Solution / Runtime [ms]	encrypt or secret_share	add_ const	add	mult_ const	mult	rotate	decrypt or reveal
MPyC (SMC)	3.02	2.86	4.98	27,878.03	28,629.94	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 4.2: 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.

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 4.2).

HECO, on the other hand, is an HE compiler that translates high-level Pythonic code to high-performance executables through MLIR [98] 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 4.2)), and we plan to extend and test it on larger benchmarks after their high-level operations are enabled.

Chapter 5

Beyond Computational Genomics: Secure MICE algorithm

Electronic Health Records (EHR) have been routinely collected by healthcare providers across the US and extensively used for research purposes. Similarly, claims data from insurance companies are often used in population-based clinical research. Normally, data are stored and managed within the institutions that collect and own them. Storing data locally is generally more feasible logistically, more cost-friendly, and easier for the data-owning entity to access, control, and manage the data. More importantly, local storage helps ensure data sovereignty, and maintain data privacy and security that comply with data protection regulations such as the Health Insurance Portability and Accountability Act (HIPAA) [110, 119]. Strong privacy protection helps build confidence in researchers, patients, and other stakeholders to encourage research collaborations in trustworthy medical AI [165].

Through collaborative research, EHRs and claims data from institutions across diverse geographical locations can form a larger and potentially more representative sample of the US population that could yield more reliable and generalizable research findings [88]. Leveraging distributed data in healthcare research can be particularly beneficial for certain marginalized or underserved minority groups because it allows institutions with very small minority populations to borrow information from others [67, 127, 161, 162]. Several large-scale distributed health data networks (DHDN) have been established to facilitate collaborations across multiple institutions. For example, the Sentinel Initiative by the U.S. Food and Drug Administration (FDA) is an effort to monitor the safety of FDA-regulated medical products. The Sentinel

can get data from more than a dozen partners including academic medical centers, healthcare systems, and health insurance companies. These data partners collect data in routine operations and maintain control of their own data [23, 125]. Another example is the Patient-Centered Scalable National Network for Effectiveness Research (pSCANNER), a national research infrastructure containing data from 13 sites emphasizing comparative effectiveness research [121]. Similarly, data are stored, owned, and governed by each one of the pSCANNER sites without a central data repository.

To enjoy the aforementioned benefits brought by distributed health data, conventional machine learning (ML) methods would require researchers to first "bring data to computation," transmitting individual patient data from the remote sites to a central data repository and performing centralized ML on aggregated data. However, this is not always permitted for legal reasons or for data privacy and security concerns. In addition, operating data centers that are large enough for centralized storage and computation is financially and logistically challenging, and the consequences are serious if the large data center experiences system failure or data breach [33]. These restrictions and limitations have motivated a broad class of modern distributed ML algorithms that "bring computation to data" [131]. Distributed ML has allowed researchers to take advantage of distributed storage and computational infrastructure and resources, which reduces, if not eliminates, the need for large data centers for EHRs. It also minimizes the need to share sensitive, protected health information, complying with legal requirements and improving the privacy and security of healthcare data [102, 157].

Missing data problems are prevalent in real-world EHRs and claims data; therefore, DHDNs as well [27, 154]. Failure to properly account for missing data will lead to biased inference and prediction results [27, 66, 149]. Recent studies further show that missingness in health data tends to harm minority groups disproportionally, exacerbating health inequities and disparities [66]. Complete case analysis that excludes observations with missing values is a valid approach if data is missing completely at random (MCAR) [149]. However, data that are missing at random (MAR) or missing not at random (MNAR) need to be properly imputed to recover unbiased analysis results [149]. Multiple imputation (MI) is a popular imputation technique that, in general, works by replacing missing values with predicted values multiple times and combining the analysis results acquired from these imputed datasets. Now, within DHDNs that comprise data from multiple institutions, the missing data problems could potentially be more complex due to various heterogeneities between the

institutions.

However, compared to the large body of literature on distributed ML algorithms, distributed MI that can handle missing data problems in DHDNs has not received as much attention. In principle, MI algorithms rely on various statistical ML models to impute missing observations. Therefore, distributed MI algorithms should enjoy the same benefits mentioned earlier as model-based distributed analysis. In addition, data sources with either small sample sizes or a small number of observed values due to high missing rates can borrow information from other data sources. To our knowledge, several distributed MI algorithms designed for MAR data have been proposed and have been shown to outperform MI conducted independently at each site [27]. Further, a distributed MI algorithm for MNAR data demonstrating superiority over independent MI algorithms has been developed [103]. However, these approaches are not provably secure as they reveal intermediate results (such as the Gramian matrix of each local dataset) that can leak private information. For example, whenever the number of individuals is less or equal to the number of training features at some party, the private data can be entirely reconstructed from its revealed Gramian matrix.

Multiple imputation (MI) addresses the uncertainty of the single imputation by probabilistically imputing data multiple times before conducting a study. The study is then done independently over each imputed dataset, and the results are combined via Rubin's rules, usually in the form of an aggregate statistic of the underlying studies' coefficients (Figure 5.1) [104]. Whenever more than one variable in the initial dataset is incomplete during a single imputation, data is imputed iteratively, one variable at a time, while re-using the complete data from the previously imputed variables. This procedure is called multiple imputation with chained equations (MICE). We used Sequre and Shechi to implement SMC and MHE variants of MICE. Thus, for the first time, we offer a provably secure imputation of the missing data in distributed electronic health records that reveals only the final analysis result. For secure statistical analyses, on top of such incomplete health data, we implemented a secure regression module.

5.1 Methods

We consider a typical distributed use case where the incomplete training data is horizontally divided between the parties (i.e., each party contributes with a different number of individuals and the same number of training features). We note, however,

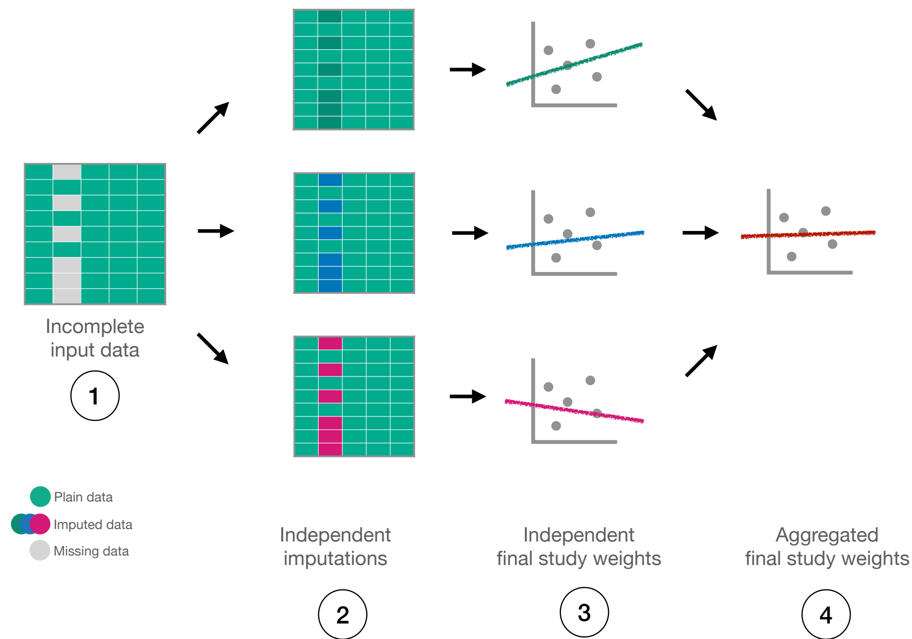


Figure 5.1: Multiple imputation. The missing data is independently imputed multiple times to address the uncertainty of imputation. Then, a set of independent studies is done on top of imputed datasets, and their parameters are combined using Rubin's rules to produce a final study. For example, if the final study involves doing a linear regression on top of a dataset, then multiple linear regression models will be independently trained and their coefficients combined, usually through some aggregation, into a final linear regression model.

that our solution is also applicable to other distribution types, such as vertical or even additive, where the sum of private data partitions forms the complete dataset. We enabled two variants of secure distributed MICE, one implemented using Secure Multiparty Computation (SMC-MICE; Algorithm 4; Figure 5.2) and the other using Multiparty Homomorphic Encryption (MHE-MICE; Algorithm 6; Figure 5.3). The former is suitable for small data scales (approximately less than $300,000$ individuals) and a number of computing parties, while the latter scales better with the increase of data size or number of parties. Both schemes enable computation on top of encrypted, distributed data without revealing any meaningful information to the study participants or data owners.

Our solution, in both schemes, works conceptually as follows. The training data is first encrypted and pooled together from multiple data owners before being imputed multiple times using linear regression with error (drawn from $\mathcal{N}(0, 0.01)$) for imputing continuous variables or logistic regression for categorical variables. On top of each

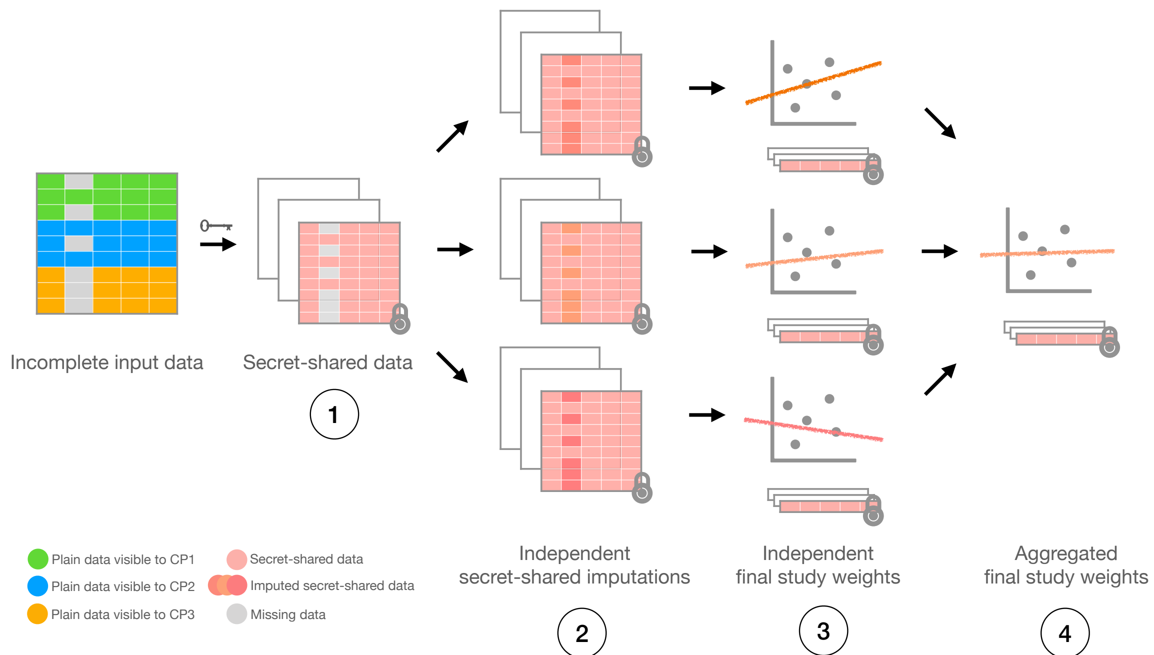


Figure 5.2: Multiple imputation via SMC. The input data is first secret-shared and then imputed and analyzed in SMC context. Each independent study produces secret-shared coefficients that are averaged together without decryption. The result is a final, secure linear regression model that allows inference on top of encrypted data without revealing any meaningful information.

imputed dataset, an independent linear regression is trained as a part of a final study, and the arithmetic average of the resulting models' coefficients is used as Rubin's rules to produce the final linear regression model. We utilized a mini-batched gradient descent with a pre-defined step size and a number of epochs for both linear and logistic regression against the mean-squared and categorical cross-entropy loss, respectively. Additionally, we used a closed-form solution if the number of features is relatively small (less than 4 in our implementations) in linear regression. Each step, in both SMC and MHE variants of MICE, is done on top of the encrypted data without revealing any meaningful information to the parties.

In SMC-MICE, the computing parties first secret share the incomplete training data ($[\mathbf{X}]$) and the training labels ($[\mathbf{y}]$). Additionally, each party provides the *missigness* mask for its data partition (i.e., the matrix of $0, 1$ entries where 0 denotes missing entry). The missingness masks are pooled into a single matrix ($\mathbf{M}$) that remains public throughout execution. The incomplete training data is then imputed multiple times using an SMC implementation of the two aforementioned regression models (i.e. linear for continuous and logistic for categorical variables). Each im-

puted, secret-shared dataset is used to train an independent final analysis model, which in our case again, is an SMC variant of linear regression. The secret shares of the models' weights are then pooled and averaged together to form the final regression model. The details of the SMC-MICE algorithm are provided in Algorithm 4 and Algorithm 5. The former provides a general overview, while the latter gives an insight into a single imputation procedure where, for each incomplete variable, a different SMC regression model is used to infer the missing data.

Algorithm 4 Regression analysis via multiple imputation using SMC
INPUT:
$[\mathbf{X}] \in \mathbb{Z}_p^{m \times n}$: secret shared incomplete training data
$[\mathbf{y}] \in \mathbb{Z}_p^{m}$: secret shared training labels
$\mathbf{M} \in \{0,1\}^{m \times n}$: public missing data mask
$\mathcal{M}_{im}$: SMC imputation model
$\mathcal{M}_{f}$: SMC final analysis model
k: public number of multiple imputations
OUTPUT:
$[\mathbf{c}] \in \mathbb{Z}_p^{n+1}$: secret shared final analysis model coefficients

1: **procedure** SMC_MICE_ANALYSIS($[\mathbf{X}], [\mathbf{y}], \mathbf{M}, \mathcal{M}_{im}, \mathcal{M}_f, k$)
2: $[\mathbf{C}] \leftarrow [\mathbf{0}] \in \mathbb{Z}_p^{k \times (n+1)}$ ▷ Secret shared zeros
3: **for** $j = \overline{0, \ldots \texttt{k}}$ **do**
4: `smc_mice_impute(`$\mathcal{M}_{im}, [\mathbf{X}], \mathbf{M}$`)`
5: `smc_fit(`$\mathcal{M}_f, [\mathbf{X}], [\mathbf{y}]$`)`
6: $[\mathbf{C}]_j \leftarrow$ `get_coeffs(`$\mathcal{M}_f$`)`
7: **end for**
8: $[\mathbf{c}] \leftarrow$ `smc_rubin(`$[\mathbf{C}]$`)`
9: **return** $[\mathbf{c}]$
10: **end procedure**

The MHE-MICE is conceptually the same as its SMC counterpart (see Algorithm 6 and Algorithm 7). The main difference is in the input data format and the implementation of elementary matrix algebra operations. Namely, the input data to MHE-MICE is initially kept local, non-encrypted at each party, and only encrypted and shared when needed throughout the computation. This, for example, enables the pre-processing steps in the imputation algorithm (Algorithm 7) to be done independently at each party on top of local, non-encrypted data. Generally, any element-wise operation, such as addition, subtraction, and multiplication, is computed in the same manner—independently at each party—as long as the partition sizes of the operands are aligned between the parties.

Algorithm 5 Imputation algorithm via chained equations via SMC

INPUT:

$\mathcal{M}_{im}$: imputation model

$[\mathbf{D}] \in \mathbb{Z}_p^{m \times n}$: secret shared incomplete training data

$\mathbf{M} \in \{0,1\}^{m \times n}$: missing data mask

```
1: procedure SMC_MICE_IMPUTE(M_im, [D], M)
2:     n ← len([D]^⊤)
3:     for j = 0,...n do
4:         C ← [D]_{M=1}                                  ▷ Filter only complete data
5:         [X] ← [C]_{:,k≠j}
6:         [y] ← [C]_{:,j}
7:         smc_fit(M_im, [X], [y])
8:         ε ← N(0, 0.01)                      ▷ Draw error from normal distribution
9:         [ŷ] ← smc_predict(M_im, ([D] · M)_{:,k≠j}, ε)
10:        [D]_{:,j} ← [ŷ]
11:        M_{:,j} ← 1 ∈ R^{m×1}
12:    end for
13: end procedure
```

Algorithm 6 Regression analysis via multiple imputation using MHE

INPUT:

$X \in \mathbb{R}^{m_i \times n}$: incomplete training data partition held locally at i-th party

$y \in \mathbb{R}^{m_i}$: training labels partition held locally at i-th party

$M \in \{0,1\}^{m_i \times n}$: missing data mask held locally at i-th party

$\mathcal{M}_{im}$: MHE imputation model

$\mathcal{M}_f$: MHE final analysis model

k: public number of multiple imputations

N: number of CKKS slots

$\mathcal{C}$: CKKS ciphertexts space (i.e., $(\mathbb{Z}_p[X]/(X+1))^2$)

OUTPUT:

$\mathbf{c} \in \mathcal{C}^{\lceil n/N \rceil}$: aggregated final analysis model coefficients

```
1: procedure MHE_MICE_ANALYSIS(X, y, M, M_im, M_f, k)
2:     C ← 0 ∈ C^{k×⌈(n+1)/N⌉}                              ▷ CKKS encrypted zeros
3:     for j = 0,...k do
4:         mhe_mice_impute(M_im, X, M)
5:         mhe_fit(M_f, X, y)
6:         C_j ← get_coeffs(M_f)
7:     end for
8:     c ← mhe_rubin(C)
9:     return c
10: end procedure
```

Algorithm 7 Imputation algorithm via chained equations using MHE

INPUT:

$\mathcal{M}_{im}$: imputation model

$D \in \mathbb{R}^{m_i \times n}$: incomplete training data partition held locally at i-th party

$M \in \{0,1\}^{m_i \times n}$: missing data mask held locally at i-th party

$\mathcal{C}$: CKKS ciphertexts space (i.e., $(\mathbb{Z}_p[X]/(X+1))^2$)

1: **procedure** MHE_MICE_IMPUTE($\mathcal{M}_{im}, D, M$)
2: $\quad n \leftarrow$ `len`($D^\top$)
3: $\quad$ **for** $j = \overline{0, \ldots n}$ **do**
4: $\quad\quad C \leftarrow D_{M=1}$ ▷ Filter only complete data at each party
5: $\quad\quad X \leftarrow C_{:,k \neq j}$
6: $\quad\quad y \leftarrow C_{:,j}$
7: $\quad\quad$ `mhe_fit`($\mathcal{M}_{im}, X, y$)
8: $\quad\quad \epsilon \leftarrow \mathcal{N}(0, 0.01)$ ▷ Draw error from normal distribution
9: $\quad\quad \hat{y} \leftarrow$ `mhe_predict`($\mathcal{M}_{im}, (D \cdot M)_{:,k \neq j}, \epsilon$) ▷ Local partition of imputed column
10: $\quad\quad D_{:,j} \leftarrow \hat{y}$
11: $\quad\quad M_{:,j} \leftarrow \mathbf{1} \in \mathbb{R}^{m_i \times 1}$
12: $\quad$ **end for**
13: **end procedure**

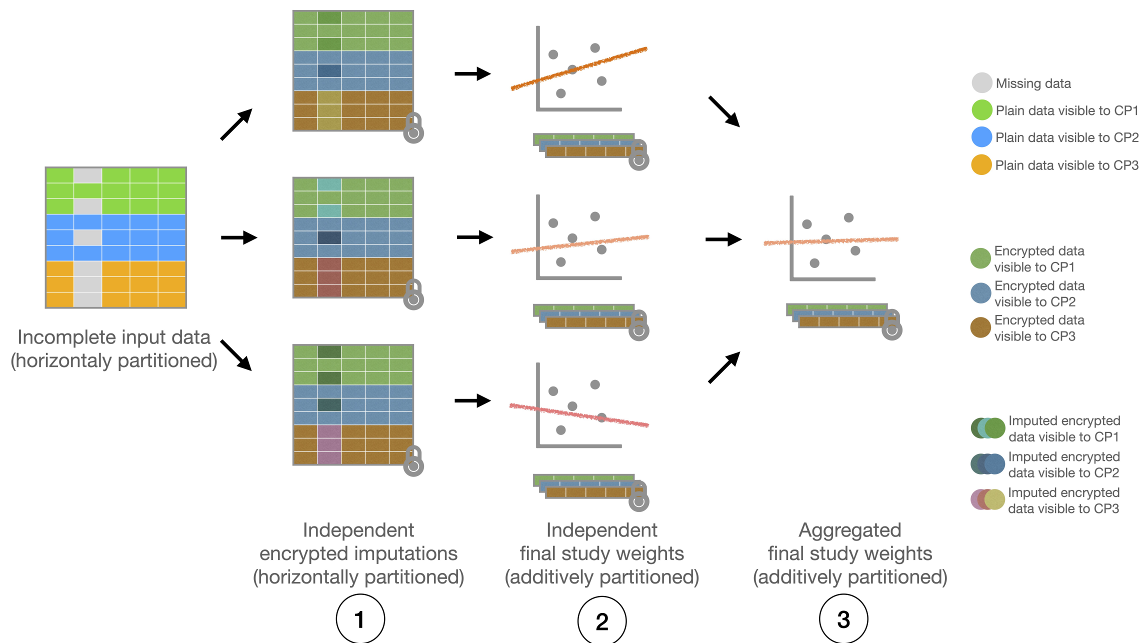


Figure 5.3: Multiple imputation via MHE. The input data is distributed between the parties and kept in a non-encrypted form, only to be encrypted when needed during the imputation and final analysis. The procedure also benefits from independent, parallel computation on top of local data partitions. This scheme, however, is suitable only for large-scale datasets due to the performance overhead incurred by the underlying, expensive cryptographic scheme that is inherently scalable with respect to data size and the number of computing parties.

The SMC implementation of linear regression uses Beaver triplets [10] to enable secure multiplication and computes the secret shares of weights in an otherwise classical manner (see Algorithm 8 for details), using only simple operations such as addition and subtraction (together with multiplication) that are efficient in our security scheme. Also, each additive operation is computed independently at each party without network overhead. For logistic regression, we implemented SMC variants of Chebyshev interpolation to support sigmoid and natural logarithms that are otherwise hard to compute in SMC.

The MHE-based regression analysis follows the same conceptual logic as its SMC counterpart. The only difference is the shape of the input data: it is non-encrypted, kept locally at each party, and only encrypted when needed throughout the computation. Moreover, computing the invariants in linear regression (i.e., the Gramian matrix $\tilde{X}^\top \times \tilde{X}$ and $\tilde{X}^\top \times y$) is also done independently at each party since the product of vertically and horizontally partitioned matrices is an additively partitioned

Algorithm 8 Linear regression via SMC

INPUT:

$[\mathbf{X}] \in \mathbb{Z}_p^{m \times n}$: secret shared training data

$[\mathbf{y}] \in \mathbb{Z}_p^{m \times 1}$: secret shared training labels

$\mathcal{M}$: linear regression model that stores initial, secret shared weights ($[\mathbf{w}_\mathcal{M}] \in \mathbb{Z}_p^{(n+1) \times 1}$), number of training epochs ($e_\mathcal{M} \in \mathbb{N}$), and step size ($\eta_\mathcal{M} \in \mathbb{R}$)

1: **procedure** SMC_FIT($\mathcal{M}, [\mathbf{X}], [\mathbf{y}]$)
2: $[\tilde{\mathbf{X}}] \leftarrow (\mathbf{X} \| \mathbf{1})$ ▷ Append bias column
3: $[\mathbf{C}] \leftarrow [\tilde{\mathbf{X}}]^\top \times [\tilde{\mathbf{X}}]$
4: $[\mathbf{R}] \leftarrow [\tilde{\mathbf{X}}]^\top \times [\mathbf{y}]$
5: **if** `len(`$[\tilde{\mathbf{X}}]^\top$`)` < 4 **then** ▷ Closed-form solution
6: $[\mathbf{w}_\mathcal{M}] \leftarrow [\mathbf{C}]^{-1} \times [\mathbf{R}]$
7: **else** ▷ Batched gradient descent
8: **for** $j = \overline{0, \ldots e_\mathcal{M}}$ **do**
9: $[\mathbf{w}_\mathcal{M}] \leftarrow [\mathbf{w}_\mathcal{M}] + ([\mathbf{R}] - [\mathbf{C}] \times [\mathbf{w}_\mathcal{M}]) \cdot \eta_\mathcal{M}$
10: **end for**
11: **end if**
12: **end procedure**

Algorithm 9 Logistic regression via SMC

INPUT:

$[\mathbf{X}] \in \mathbb{Z}_p^{m \times n}$: secret shared training data

$[\mathbf{y}] \in \mathbb{Z}_p^{m \times 1}$: secret shared training labels

$\mathcal{M}$: logistic regression model that stores initial weights ($\mathbf{w}_\mathcal{M} \in \mathbb{Z}_p^{(n+1) \times 1}$), number of training epochs ($e_\mathcal{M} \in \mathbb{N}$), and step size ($\eta_\mathcal{M} \in \mathbb{R}$)

1: **procedure** SMC_FIT($\mathcal{M}, [\mathbf{X}], [\mathbf{y}]$)
2: $[\tilde{\mathbf{X}}] \leftarrow (\mathbf{X} \| \mathbf{1})$ ▷ Append bias column
3: **for** $j = \overline{0, \ldots e_\mathcal{M}}$ **do**
4: $[\mathbf{A}] \leftarrow \sigma_{cheby}([\tilde{\mathbf{X}}] \times [\mathbf{w}_\mathcal{M}], (0, 1))$
5: $[\mathbf{w}_\mathcal{M}] \leftarrow [\mathbf{w}_\mathcal{M}] - [\tilde{\mathbf{X}}]^\top \times ([\mathbf{A}] - [\mathbf{y}]) \cdot \eta_\mathcal{M}$
6: **end for**
7: **end procedure**

matrix with each partition being a product of corresponding non-encrypted local shares. Some operations, however, require one of the operands to be *aggregated* (i.e., encrypted and shared among the parties) beforehand. For example, matrix multiplication of two additively partitioned matrices requires at least one operand to be aggregated beforehand. The result is then obtained by multiplying each additive share with the aggregated counterpart independently at each party. The aggregation strategy (i.e., deciding whether to aggregate the first or the second operand) directly impacts the partitioning of the result and the performance of all downstream operations. For example, aggregating the weights $[\mathbf{w}_{\mathcal{M}}]$ instead of training data $\tilde{X}$ in logistic regression in Algorithm 11 would result in a completely different algorithm downstream. Shechi's automatic selection of optimal aggregation strategy consistently aggregates $\tilde{X}$ first to avoid aggregating $[\mathbf{w}_{\mathcal{M}}]$ multiple times within the loop body. Moreover, multiplying vertically partitioned against the horizontally partitioned matrix, as well as two additively partitioned matrices, are the only two matrix multiplication instances encountered in our implementation of linear and logistic regression.

We implemented SMC- and MHE-MICE in Sequre and Shechi in less than 550 lines of high-level Pythonic code.

5.2 Results

We adopted the experiment setup from the previous work [27], which includes four simulations and one real-data study. Each study follows the same pattern. First, the incomplete dataset of a different number of individuals and variables, such as demographic and disease information, is encrypted and pooled together from multiple study participants. The missing data in the pooled dataset is then imputed multiple times to form several independent complete datasets, on top of which different linear regression models are trained as part of the *final analysis*. The trained models' weights are then combined via Rubin's rules to produce a final linear regression model that is used to assess performance. Each step of the study is done on encrypted data without revealing any meaningful information to the study participants apart from the final analysis output. The accuracy of the final linear regression is measured as a *mean absolute difference* and a *standard deviation of the absolute difference* between the predicted outcome and the ground truth. We also measured the bias $\|\mathbb{E}\mathbf{\Theta} - \tilde{\mathbf{\Theta}}\|_2$, standard deviation $\sqrt{\mathbb{E}\|\mathbf{\Theta} - \mathbb{E}\mathbf{\Theta}\|_2^2}$, and the mean-squared error $\sqrt{\mathbb{E}\|\mathbf{\Theta} - \tilde{\mathbf{\Theta}}\|_2^2}$ of

Algorithm 10 Linear regression via MHE

INPUT:

N: number of CKKS slots

$\mathcal{C}$: CKKS ciphertexts space (i.e., $(\mathbb{Z}_p[X]/(X+1))^2$)

$X \in \mathbb{R}^{m_i \times n}$: training data partition held locally at i-th party

$y \in \mathbb{R}^{m_i \times 1}$: training labels partition held locally at i-th party

$\mathcal{M}$: linear regression model that stores initial, aggregated weights ($[\mathbf{w}_\mathcal{M}] \in \mathcal{C}^{\lceil (n+1)/N \rceil \times 1}$, number of training epochs ($e_\mathcal{M} \in \mathbb{N}$), and step size ($\eta_\mathcal{M} \in \mathbb{R}$)

1: **procedure** MHE_FIT($\mathcal{M}, X, y$)
2: $\quad \tilde{X} \leftarrow (X \| \mathbf{1})$ ▷ Append bias column locally at each party
3: $\quad [C] \leftarrow \tilde{X}^\top \times \tilde{X}$ ▷ Additively shared local partitions $\tilde{X}^\top \times \tilde{X}$
4: $\quad [R] \leftarrow \tilde{X}^\top \times y$ ▷ Additively shared local partitions of $\tilde{X}^\top \times y$
5: $\quad$ **if** `len`($\tilde{X}^\top$) < 4 **then** ▷ Closed-form solution
6: $\quad\quad \mathbf{R} \leftarrow$ `aggregate`($[R]$)
7: $\quad\quad [\mathbf{w}_\mathcal{M}] \leftarrow ([C])^{-1} \times \mathbf{R}$
8: $\quad$ **else** ▷ Batched gradient descent
9: $\quad\quad$ **for** $j = \overline{0, \ldots e_\mathcal{M}}$ **do**
10: $\quad\quad\quad \mathbf{w}_\mathcal{M} \leftarrow$ `aggregate`($[\mathbf{w}_\mathcal{M}]$)
11: $\quad\quad\quad [\mathbf{w}_\mathcal{M}] \leftarrow [\mathbf{w}_\mathcal{M}] + ([R] - [C] \times \mathbf{w}_\mathcal{M}) \cdot \eta_\mathcal{M}$
12: $\quad\quad$ **end for**
13: $\quad$ **end if**
14: **end procedure**

Algorithm 11 Logistic regression via MHE

INPUT:

N: number of CKKS slots

$\mathcal{C}$: CKKS ciphertexts space (i.e., $(\mathbb{Z}_p[X]/(X+1))^2$)

$X \in \mathbb{R}^{m_i \times n}$: training data partition held locally at i-th party

$y \in \mathbb{R}^{m_i \times 1}$: training labels partition held locally at i-th

$\mathcal{M}$: logistic regression model that stores initial, aggregated weights ($[\mathbf{w}_\mathcal{M}] \in \mathcal{C}^{\lceil (n+1)/N \rceil \times 1}$, number of training epochs ($e_\mathcal{M} \in \mathbb{N}$), and step size ($\eta_\mathcal{M} \in \mathbb{R}$)

1: **procedure** SMC_FIT($\mathcal{M}, X, y$)
2: $\quad \tilde{X} \leftarrow (X \| \mathbf{1})$ ▷ Append bias column locally at each party
3: $\quad \tilde{\mathbf{X}} \leftarrow$ `aggregate`($\tilde{X}$)
4: $\quad \tilde{\mathbf{X}}^\top \leftarrow$ `aggregate`($\tilde{X}^\top$)
5: $\quad$ **for** $j = \overline{0, \ldots e_\mathcal{M}}$ **do**
6: $\quad\quad [\mathbf{P}] \leftarrow \tilde{\mathbf{X}} \times [\mathbf{w}_\mathcal{M}]$
7: $\quad\quad [\mathbf{A}] \leftarrow \sigma_{cheby}([\mathbf{P}], (0, 1))$
8: $\quad\quad [\mathbf{w}_\mathcal{M}] \leftarrow [\mathbf{w}_\mathcal{M}] - \tilde{\mathbf{X}}^\top \times ([\mathbf{A}] - y) \cdot \eta_\mathcal{M}$
9: $\quad$ **end for**
10: **end procedure**

the linear regression weights $\mathbf{\Theta}$ and their ground truth $\tilde{\mathbf{\Theta}}$ in the final analysis. To assess the quality of imputation alone, we additionally measured the mean absolute difference and standard deviation between imputed datasets and their ground truth in the simulation studies with incomplete continuous variables, and accuracy and area-under-curve (AUC) for the ones with incomplete binary variables. This assessment is not possible in the real-data study, however, because the ground truth is unknown. Finally, we also measured the runtime and network overhead where applicable.

We implemented two secure solutions for MICE, one based on Secure Multiparty Computation and the other on Multiparty Homomorphic Encryption. Additionally, we implemented two non-secure solutions to compare against. The first one is a raw Python implementation of MICE using off-the-shelf linear and logistic regression for imputation and final analysis, and the second one is an off-the-shelf MICE algorithm from Python's `sklearn` library. There is no clear winner between the two non-secure solutions, but the latter is generally expected to have better accuracy while the former has a better performance. The Python-based solutions are tested in an offline, non-secure context on top of plain, non-encrypted data, while the secure solutions are tested in a secure distributed setup, on top of encrypted data, with two computing parties aided by a *trusted dealer*.

The first simulation is conducted on top of ten variables drawn from a normal distribution $\mathcal{N}(0, 1)$, and one variable made incomplete uniformly at random with the missingness rate of 30%. The second simulation is the same, with the incomplete variable being a binary variable drawn from a Bernoulli distribution $\mathcal{B}(1, 0.5)$ instead. The third and fourth simulation studies have only two variables, X_1 and X_2, with the second variable drawn from a uniform distribution $\mathcal{U}(-3, 3)$ and the first either from a normal distribution $\mathcal{N}(0.2 - 0.5X_2, 1)$ in the third simulation, or from a Bernoulli distribution $\mathcal{B}(1 + e^{-0.2+0.5X_1})$ in the fourth simulation, with the missingness rate of approximately 50%. Each simulation is benchmarked for a different number of individuals (500 and $5,000$ for our experiments). For a real-data study, the data from Georgia Coverdell Acute Stroke Registry (GCASR) is used with 15 out of 203 selected variables (five continuous and ten binary) and $68,287$ patients. Each continuous variable and seven binary variables are incomplete, and the missingness rate ranges between 0.035% and 53.84%. The outcome variable (i.e., the ground truth of final regression analysis) in each simulation study is obtained as $Y = \mathbf{\Theta}_0 + \sum_i X_i \mathbf{\Theta}_i + \epsilon$, where X_i are the variables; $\mathbf{\Theta}_i$ the ground truth linear regression weights (set to $\mathbf{1}$ in our experiments), and ϵ is drawn from $\mathcal{N}(0, (\mathbf{\Theta}_0 + \sum_i X_i \mathbf{\Theta}_i)/100)$. The outcome

variable is computed on a complete dataset, before removing the missing data. In a real-data study, an *arrival-to-computed tomography time* is used as an outcome variable. In each study, five multiple imputations are used (i.e., each dataset is imputed five times and five independent linear regression models are trained as a part of a final analysis). The simulation and real-data studies are independently benchmarked 100 and 5 times, respectively.

The imputation and the final study quality of secure solutions are on par or slightly better than the offline solutions in all simulation studies. We note that our goal was not to improve the existing MICE algorithms but to design their secure equivalents with on-par accuracy and performance for the first time. The imputation accuracy is slightly worse (< 0.006) only in studies where a categorical variable is imputed (Table 5.3 and Table 5.5) due to approximation algorithms (Chebyshev approximation) employed in secure variants of logistic regression. Similarly, the quality of the final study is only fractionally worse (0.001 - 0.063) in secure solutions—the offset that can be further accounted for approximation errors that are unavoidable in secure technologies.

To further assess the quality of our imputation algorithms, we measured a *number of discrepancies* [27] with respect to an off-shelf MICE algorithm from `sklearn` library as a base algorithm. In short, a variable in the final linear regression study has a discrepancy between the two MICE algorithms (target and base algorithm) if and only if its statistical significance is less or equal to 0.05 in the base algorithm and either its statistical significance in the target algorithm is larger than 0.05 or its weights in the two algorithms have the opposite signs. The smaller number of discrepancies indicates similar imputation quality between the two algorithms. In our measurements, we observed only one discrepancy in the offline Python implementation of MICE in the real-data study. We measured no discrepancies in any other solution across all studies. Counting the number of discrepancies is particularly useful when there is no ground truth to measure the quality of imputation against, such as in the real-data study in Table 5.1. Further, the accuracy noise that stems from underlying approximation algorithms and approximation errors inherent in secure technologies is more obvious in the final analysis of the large-scale, real-data study (Table 5.1). However, the discrepancy analysis shows that our imputation algorithms are accurate even at this data scale. Moreover, apart from the MHE variant, which is inherently slow for this data scale, the runtimes of SMC solutions are also comparable or even faster for the smaller number of individuals than the non-secure algorithms. This

is an important practical result since secure solutions are generally known to incur large performance overhead. Lastly, the stable runtime of cca. $2,000$ seconds in MHE solutions across all studies demonstrates the scalability of this solution. In our experience, the SMC solutions become slower than the MHE counterparts only when the number of individuals surpasses $300,000$.

Lastly, all experiments were done on a single 12-core Intel Core i7-8700 CPU at 3.20GHz and 64 GB of RAM. To simulate a multiparty setup, the UNIX sockets were used to connect multiple processes—each process corresponding to a separate computing party.

Table 5.1: Real-data scenario: 5 continuous and 10 binary incomplete (missing) variables, 5 random runs, missing rate ranging from 0.035% to 53.84%.

	Final analysis			Performance	
Technology	$\lvert\hat{\mathbf{y}} - \mathbf{y}\rvert$ (μ)	$\lvert\hat{\mathbf{y}} - \mathbf{y}\rvert$ (σ)	Discr.	Time (s)	Net (MB)
Python	0.306	0.333	1	23.185	N/A
PyMICE	0.305	0.332	N/A	98.716	N/A
SMC-MICE	0.632	0.764	0	12.108	712
MHE-MICE	0.629	0.765	0	2,017	19,264

Table 5.2: Scenario 1: Single continuous incomplete (missing) variable, 9 continuous complete variables, 100 random runs, 30% missing rate

		Final analysis						Imputation		Performance	
	Technology	$\mathbf{\Theta}$ bias	$\mathbf{\Theta}$ SD	$\mathbf{\Theta}$ rMSE	$\lvert\hat{\mathbf{y}}-\mathbf{y}\rvert$ (μ)	$\lvert\hat{\mathbf{y}}-\mathbf{y}\rvert$ (σ)	Discr.	$\lvert\hat{\mathbf{y}}-\mathbf{y}\rvert$ (μ)	$\lvert\hat{\mathbf{y}}-\mathbf{y}\rvert$ (σ)	Time (s)	Net (MB)
500 inds.	Python	0.045	$4.87 \cdot 10^{-4}$	0.045	0.041	0.034	0	0.496	0.073	0.061	N/A
	PyMICE	0.042	$1.57 \cdot 10^{-16}$	0.042	0.040	0.032	N/A	0.498	0.034	0.223	N/A
	SMC-MICE	0.045	$4.18 \cdot 10^{-4}$	0.045	0.041	0.034	0	0.496	0.073	0.090	4.737
	MHE-MICE	0.065	$3.91 \cdot 10^{-4}$	0.065	0.044	0.035	0	0.496	0.071	1,658	19,834
5000 inds.	Python	0.025	$4.00 \cdot 10^{-16}$	0.025	0.031	0.027	0	0.814	0.601	0.149	N/A
	PyMICE	0.019	$4.15 \cdot 10^{-16}$	0.019	0.027	0.025	N/A	0.813	0.600	0.205	N/A
	SMC-MICE	0.019	$1.15 \cdot 10^{-4}$	0.019	0.029	0.026	0	0.813	0.601	0.536	44.641
	MHE-MICE	0.019	$1.13 \cdot 10^{-4}$	0.019	0.029	0.026	0	0.813	0.601	2,129	32,839

Table 5.3: Scenario 2: Single binary incomplete (missing) variable, 9 continuous complete variables, 100 random runs, 30% missing rate

		Final analysis						Imputation	Performance		
	Technology	$\mathbf{\Theta}$ bias	$\mathbf{\Theta}$ SD	$\mathbf{\Theta}$ rMSE	$\lvert\hat{\mathbf{y}}-\mathbf{y}\rvert$ (μ)	$\lvert\hat{\mathbf{y}}-\mathbf{y}\rvert$ (σ)	Discr.	Accuracy	AUC	Time (s)	Net (MB)
500 inds.	Python	0.268	$5.09 \cdot 10^{-16}$	0.268	0.141	0.092	0	0.466	0.477	0.063	N/A
	PyMICE	0.035	$4.00 \cdot 10^{-16}$	0.035	0.040	0.031	N/A	0.460	0.500	0.225	N/A
	SMC-MICE	0.052	$3.12 \cdot 10^{-10}$	0.052	0.049	0.038	0	0.466	0.432	0.335	14.019
	MHE-MICE	0.053	$2.91 \cdot 10^{-7}$	0.053	0.050	0.038	0	0.466	0.432	1,753	17,061
5000 inds.	Python	0.263	$4.71 \cdot 10^{-16}$	0.263	0.134	0.078	0	0.508	0.509	0.143	N/A
	PyMICE	0.011	$4.84 \cdot 10^{-16}$	0.011	0.024	0.025	N/A	0.510	0.496	0.212	N/A
	SMC-MICE	0.012	$3.30 \cdot 10^{-10}$	0.012	0.026	0.026	0	0.508	0.475	3.052	137.398
	MHE-MICE	0.012	$8.31 \cdot 10^{-7}$	0.012	0.026	0.026	0	0.508	0.475	2,321	27,038

Table 5.4: Scenario 3: Single continuous incomplete (missing) variable, single continuous complete variable, 100 random runs, 50% missing rate

	Technology	Final analysis						Imputation		Performance	
		Θ bias	Θ SD	Θ rMSE	$\lvert\hat{\mathbf{y}}-\mathbf{y}\rvert$ (μ)	$\lvert\hat{\mathbf{y}}-\mathbf{y}\rvert$ (σ)	Discr.	$\lvert\hat{\mathbf{y}}-\mathbf{y}\rvert$ (μ)	$\lvert\hat{\mathbf{y}}-\mathbf{y}\rvert$ (σ)	Time (s)	Net (MB)
500 inds.	Python	0.014	$5.24 \cdot 10^{-4}$	0.014	0.028	0.019	0	0.392	0.238	0.023	N/A
	PyMICE	0.014	$2.22 \cdot 10^{-16}$	0.014	0.028	0.018	N/A	0.393	0.235	0.046	N/A
	SMC-MICE	0.013	$2.35 \cdot 10^{-3}$	0.013	0.028	0.018	0	0.392	0.238	0.038	1.686
	MHE-MICE	0.016	$2.40 \cdot 10^{-3}$	0.016	0.031	0.021	0	0.388	0.248	285.9	9,629
5000 inds.	Python	0.002	$3.09 \cdot 10^{-4}$	0.002	0.021	0.013	0	0.494	0.074	0.086	N/A
	PyMICE	0.015	0.0	0.015	0.021	0.014	N/A	0.494	0.072	0.031	N/A
	SMC-MICE	0.002	$4.06 \cdot 10^{-2}$	0.040	0.040	0.019	0	0.494	0.074	0.206	16.176
	MHE-MICE	0.069	$2.40 \cdot 10^{-2}$	0.069	0.051	0.027	0	0.494	0.087	1,910	86,628

Table 5.5: Scenario 4: Single binary incomplete (missing) variable, single continuous complete variable, 100 random runs, 50% missing rate

	Technology	Final analysis						Imputation	Performance		
		Θ bias	Θ SD	Θ rMSE	$\lvert\hat{\mathbf{y}}-\mathbf{y}\rvert$ (μ)	$\lvert\hat{\mathbf{y}}-\mathbf{y}\rvert$ (σ)	Discr.	Accuracy	AUC	Time (s)	Net (MB)
500 inds.	Python	0.378	0.0	0.378	0.19	0.026	0	0.636	0.466	0.024	N/A
	PyMICE	0.010	$2.48 \cdot 10^{-16}$	0.010	0.019	0.015	N/A	0.632	0.631	0.047	N/A
	SMC-MICE	0.012	$3.85 \cdot 10^{-3}$	0.012	0.023	0.015	0	0.632	0.719	0.231	8.359
	MHE-MICE	0.028	$2.50 \cdot 10^{-3}$	0.028	0.042	0.026	0	0.640	0.719	470.4	8,298
5000 inds.	Python	0.348	$2.48 \cdot 10^{-16}$	0.348	0.184	0.121	0	0.529	0.510	0.081	N/A
	PyMICE	0.004	0.0	0.004	0.024	0.014	N/A	0.525	0.500	0.031	N/A
	SMC-MICE	0.060	$1.08 \cdot 10^{-1}$	0.123	0.105	0.045	0	0.518	0.467	2.016	86.63
	MHE-MICE	0.082	$4.97 \cdot 10^{-3}$	0.083	0.087	0.038	0	0.523	0.467	1,737	60,778

Chapter 6

Conclusion

We have introduced Sequre, a performant and user-friendly framework for building Secure Multiparty Computing applications, and Shechi, the first framework that combines Secure Multiparty Computation and Fully Homomorphic Encryption to enable high-performance secure computing without sacrificing readability and maintainability.

Sequre introduces a compiler that transforms a high-level Python script into a secure MPC program while applying a variety of sophisticated code optimizations without manual intervention. This allows practitioners without the expertise in MPC to develop and use efficient MPC software. Our results on diverse applications demonstrate the usability of Sequre as well as its state-of-the-art performance, often outperforming carefully optimized published tools from prior works. We note that there are inherent limitations to what can be achieved by the automated compiler optimization of Sequre. For example, the performance difference between the two Sequre tools for metagenomic binning (Sequre-Ganon and Sequre-Opal) illustrates how the performance of a program pivotally depends on the underlying algorithmic choices. To an extent, the user still needs to remain engaged in exploring different implementation strategies in order to obtain the most efficient tool for the desired task. The fact that Sequre allows the user to program in Python without any special consideration for MPC greatly simplifies and accelerates this development process. Providing a library of high-level routines that are commonly used in biomedical analyses (e.g. basic statistical models) may further reduce the user's burden on the algorithmic side and, thus, is a meaningful direction for future work. Sequre can easily be extended to incorporate novel MPC protocols, frameworks, and optimization techniques. Its approach can also be used to target other privacy-enhancing technologies such as

Homomorphic Encryption or hardware-based Trusted Execution Environment (TEE) technologies, which present unique challenges. Finally, practitioners in other fields beyond biomedicine can use Sequre to develop secure data analysis pipelines.

Shechi 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 Sequre and other 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 where mass collaboration or large data scales are needed and allow users in various domains to conduct collaborative studies that would otherwise be impossible or impractical due to privacy concerns.

Finally, we use Sequre and Shechi to enable provably secure statistical studies on top of private, incomplete distributed datasets while maintaining data privacy. Specifically, we used Secure Multiparty Computation (SMC) and Multiparty Homomorphic Encryption (MHE) to implement a secure distributed variant of *principal component analysis (PCA)*, *genome-wide association study (GWAS)*, *drug-target interaction inference (DTI)*, *metagenomic binning*, *kinship inference (KING)*, and *multiple imputation with chain equations (MICE)*. Our solutions proved to have an on-par accuracy with the standard, non-secure, and centralized equivalents and offer superior performance to their previous secure implementations.

Bibliography

[1] 23andMe. https://www.23andme.com/, (11.2023).

[2] Cocsku Acay, Rolph Recto, Joshua Gancher, Andrew C Myers, and Elaine Shi. Viaduct: An Extensible, Optimizing Compiler for Secure Distributed Programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 740–755, New York, NY, USA, 2021. Association for Computing Machinery.

[3] Kinan Dak Albab, Rawane Issa, Andrei Lapets, Peter Flockhart, Lucy Qin, and Ira Globus-Harris. Tutorial: Deploying secure multi-party computation on the web using JIFF. *Proceedings - 2019 IEEE Secure Development, SecDev 2019*, page 3, 2019.

[4] Anaconda. Numba, 2018. https://numba.pydata.org/.

[5] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. Rpython: A step towards reconciling dynamically and statically typed oo languages. In *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, page 53–64, New York, NY, USA, 2007. Association for Computing Machinery.

[6] E Anderson, Zhaojun Bai, and J Dongarra. Generalized qr factorization and its applications. *Linear Algebra and its Applications*, 162:243–271, 1992.

[7] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 805–817, New York, NY, USA, 2016. Association for Computing Machinery.

[8] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 193–205. IEEE Press, 2019.

[9] Yuyan Bao, Kirshanthan Sundararajah, Raghav Malik, Qianchuan Ye, Christopher Wagner, Nouraldin Jaber, Fei Wang, Mohammad Hassan Ameri, Donghang Lu, Alexander Seto, Benjamin Delaware, Roopsha Samanta, Aniket Kate, Christina Garman, Jeremiah Blocki, Pierre-David Letourneau, Benoit Meister, Jonathan Springer, Tiark Rompf, and Milind Kulkarni. HACCLE: Metaprogramming for Secure Multi-Party Computation. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2021, pages 130–143, New York, NY, USA, 2021. Association for Computing Machinery.

[10] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '91, page 420–432, Berlin, Heidelberg, 1991. Springer-Verlag.

[11] Assaf Ben-David, Noam Nisan, and Benny Pinkas. Fairplaymp: a system for secure multi-party computation. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 257–266, 2008.

[12] Fabrice Benhamouda, Tancrède Lepoint, Claire Mathieu, and Hang Zhou. Optimization of bootstrapping in circuits. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2423–2433. SIAM, 2017.

[13] Bonnie Berger and Hyunghoon Cho. Emerging technologies towards enhancing privacy in genomic data sharing. *Genome Biology*, 20(1):128, 2019.

[14] Marcelo Blatt, Alexander Gusev, Yuriy Polyakov, and Shafi Goldwasser. Secure large-scale genome-wide association studies using homomorphic encryption. *Proceedings of the National Academy of Sciences*, 117(21):11608–11613, 2020.

[15] Fabian Boemer, Rosario Cammarota, Daniel Demmler, Thomas Schneider, and Hossein Yalame. 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.

[16] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. 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.

[17] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. 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.

[18] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security*, pages 192–206. Springer, 2008.

[19] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM.

[20] Luca Bonomi, Yingxiang Huang, and Lucila Ohno-Machado. Privacy challenges and research opportunities for genomic data sharing. *Nature Genetics*, 52(7):646–654, 2020.

[21] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Proceedings of the 32nd Annual Cryptology Conference on Advances in Cryptology — CRYPTO 2012 - Volume 7417*, page 868–886, Berlin, Heidelberg, 2012. Springer-Verlag.

[22] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, page 309–325, New York, NY, USA, 2012. Association for Computing Machinery.

[23] Jeffrey S Brown, Aaron B Mendelsohn, Young Hee Nam, Judith C Maro, Noelle M Cocoros, Carla Rodriguez-Watson, Catherine M Lockhart, Richard Platt, Robert Ball, Gerald J Dal Pan, et al. The US food and drug administration sentinel system: a national resource for a learning health system. *Journal of the American Medical Informatics Association*, 29(12):2191–2200, 2022.

[24] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. Hycc: Compilation of hybrid protocols for practical secure computation. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 847–861, 2018.

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

[26] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. Ezpc: programmable, efficient, and scalable secure two-party computation for machine learning. *Cryptology ePrint Archive*, 2017.

[27] Changgee Chang, Yi Deng, Xiaoqian Jiang, and Qi Long. Multiple imputation for analysis of incomplete data in distributed health data networks. *Nature communications*, 11(1):5467, 2020.

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

[29] Hao Chen, Kim Laine, and Rachel Player. Simple encrypted arithmetic library - seal v2.1. Cryptology ePrint Archive, Paper 2017/224, 2017. `https://eprint.iacr.org/2017/224`.

[30] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full RNS variant of approximate homomorphic encryption. In *Selected Areas in Cryptography–SAC 2018: 25th International Conference, Calgary, AB, Canada, August 15–17, 2018, Revised Selected Papers 25*, pages 347–368. Springer, 2019.

[31] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Proceedings of the*

International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT), 2017.

[32] Hyunghoon Cho, David Froelicher, Jeffrey Chen, Manaswitha Edupalli, Apostolos Pyrgelis, Juan R Troncoso-Pastoriza, Jean-Pierre Hubaux, and Bonnie Berger. Secure and Federated Genome-Wide Association Studies for Biobank-Scale Datasets. *bioRxiv*, pages 2022–11, 2022. [doi:10.1101/2022.11.30.518537].

[33] Hyunghoon Cho, David Froelicher, Natnatee Dokmai, Anupama Nandi, Shuvom Sadhuka, Matthew M. Hong, and Bonnie Berger. Privacy-enhancing technologies in biomedical data science. *Annual Review of Biomedical Data Science*, 7(Volume 7, 2024):317–343, 2024.

[34] Hyunghoon Cho, David J. Wu, and Bonnie Berger. Secure genome-wide association analysis using multiparty computation. *Nature Biotechnology*, 36(6):547–551, Jul 2018.

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

[36] Sangeeta Chowdhary, Wei Dai, Kim Laine, and Olli Saarikivi. 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.

[37] Anamaria Costache, Benjamin R Curtis, Erin Hales, Sean Murphy, Tabitha Ogilvie, and Rachel Player. On the Precision Loss in Approximate HE. *ePrint*, 2022.

[38] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for Dishonest Majority. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 769–798, Cham, 2018. Springer International Publishing.

[39] Temesgen Hailemariam Dadi, Enrico Siragusa, Vitor C Piro, Andreas Andrusch, Enrico Seiler, Bernhard Y Renard, and Knut Reinert. DREAM-Yara: an exact read mapper for very large databases with short update time. *Bioinformatics*, 34(17):i766–i772, 09 2018.

[40] Luis Damas. Type assignment in programming languages. *KB thesis scanning project 2015*, 1984.

[41] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *Theory of Cryptography*, pages 285–304, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[42] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, pages 643–662, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[43] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. Eva: An encrypted vector arithmetic language and compiler for efficient homomorphic computation. In *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*, pages 546–561, 2020.

[44] Roshan Dathathri, Kim Laine, Olli Saarikivi, Kristin Lauter, Hao Chen, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. Chet: An optimizing compiler for fully-homomorphic neural-network inferencing. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 142–156, 6 2019.

[45] Daniel Demmler, Stefan Katzenbeisser, Thomas Schneider, Tom Schuster, and Christian Weinert. Improved Circuit Compilation for Hybrid MPC via Compiler Intermediate Representation. *IACR Cryptol. ePrint Arch.*, 2021:521, 2021.

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

[47] Cynthia Dwork. Differential Privacy. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming*, pages 1–12, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[48] Taher Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[49] Alison R Erickson, Brandi L Cantarel, Regina Lamendella, Youssef Darzi, Emmanuel F Mongodin, Chongle Pan, Manesh Shah, Jonas Halfvarson, Curt Tysk, Bernard Henrissat, et al. Integrated metagenomics/metaproteomics reveals human host-microbiota signatures of crohn's disease. *PloS one*, 7(11):e49138, 2012.

[50] Yaniv Erlich, Tal Shor, Itsik Pe'er, and Shai Carmi. Identity inference of genomic data using long-range familial searches. *Science*, 362(6415):690–694, 2018.

[51] Aly A. et al. Scale and mamba documentation, 2018. `https://homes.esat.kuleuven.be/~nsmart/SCALE/`.

[52] David Evans, Vladimir Kolesnikov, and Mike Rosulek. *A Pragmatic Introduction to Secure Multi-Party Computation*. Now Foundations and Trends, 2018.

[53] Noah Fierer, Christian L. Lauber, Nick Zhou, Daniel McDonald, Elizabeth K. Costello, and Rob Knight. Forensic identification using skin bacterial communities. *Proceedings of the National Academy of Sciences*, 107(14):6477–6481, 2010.

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

[55] Kevin J Forsberg, Alejandro Reyes, Bin Wang, Elizabeth M Selleck, Morten OA Sommer, and Gautam Dantas. The shared antibiotic resistome of soil bacteria and human pathogens. *Science*, 337(6098):1107–1111, 2012.

[56] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Cbmc-gc: An ANSI C compiler for secure two-party computations. In Albert Cohen, editor, *Compiler Construction*, pages 244–249, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[57] Eric A. Franzosa, Katherine Huang, James F. Meadow, Dirk Gevers, Katherine P. Lemon, Brendan J. M. Bohannan, and Curtis Huttenhower. Identifying

personal microbiomes using metagenomic codes. *Proceedings of the National Academy of Sciences*, 112(22):E2930–E2938, 2015.

[58] David Froelicher, Hyunghoon Cho, Manaswitha Edupalli, Joao Sa Sousa, Jean-Philippe Bossuat, Apostolos Pyrgelis, Juan R. Troncoso-Pastoriza, Bonnie Berger, and Jean-Pierre Hubaux. Scalable and privacy-preserving federated principal component analysis. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 888–905, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[59] David Froelicher, Patricia Egger, João Sá Sousa, Jean Louis Raisaro, Zhicong Huang, Christian Vincent Mouchet, Bryan Alexander Ford, and Jean-Pierre Hubaux. Unlynx: a decentralized system for privacy-conscious data sharing. *Proceedings on Privacy Enhancing Technologies (PoPETS)*, 2017(4):232–250, 2017.

[60] David Froelicher, Juan R Troncoso-Pastoriza, Apostolos Pyrgelis, Sinem Sav, Joao Sa Sousa, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. Scalable privacy-preserving distributed learning. In *Proceedings on Privacy Enhancing Technologies Symposium*, volume 2, pages 323–347, 2021.

[61] David Froelicher, Juan R Troncoso-Pastoriza, Jean Louis Raisaro, Michel A Cuendet, Joao Sa Sousa, Hyunghoon Cho, Bonnie Berger, Jacques Fellay, and Jean-Pierre Hubaux. Truly privacy-preserving federated analytics for precision medicine with multiparty homomorphic encryption. *Nature communications*, 12(1):5910, oct 2021.

[62] Amadou Gaye, Yannick Marcon, Julia Isaeva, Philippe LaFlamme, Andrew Turner, Elinor M Jones, Joel Minion, Andrew W Boyd, Christopher J Newby, Marja-Liisa Nuotio, et al. DataSHIELD: Taking the Analysis to the Data, not the Data to the Analysis. *International journal of epidemiology*, 43(6):1929–1944, 2014.

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

[64] Comprehensive solutions for genetic genealogy and family tree search. `https://www.gedmatch.com/`, (11.2023).

[65] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, page 169–178, New York, NY, USA, 2009. Association for Computing Machinery.

[66] Emily Getzen, Lyle Ungar, Danielle Mowery, Xiaoqian Jiang, and Qi Long. Mining for equitable health: Assessing the impact of missing data in electronic health records. *Journal of biomedical informatics*, 139:104269, 2023.

[67] Milena A Gianfrancesco, Suzanne Tamang, Jinoos Yazdany, and Gabriela Schmajuk. Potential biases in machine learning algorithms using electronic health record data. *JAMA internal medicine*, 178(11):1544–1547, 2018.

[68] Charles Gouert, Dimitris Mouris, and Nektarios Tsoutsos. Sok: New insights into fully homomorphic encryption libraries via standardized benchmarks. *Proceedings on privacy enhancing technologies*, 2023.

[69] Isaac Gouy. The computer language benchmarks game. `https://benchmarksgame-team.pages.debian.net/benchmarksgame/`.

[70] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions. *SIAM review*, 53(2):217–288, 2011.

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

[72] Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern'andez del R'ıo, Mark Wiebe, Pearu Peterson, Pierre G'erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

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

[74] K Hayen. Nuitka, 2012.

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

[76] Tim Heldmann, Thomas Schneider, Oleksandr Tkachenko, Christian Weinert, and Hossein Yalame. Llvm-based circuit compilation for practical secure computation. In Kazue Sako and Nils Ole Tippenhauer, editors, *Applied Cryptography and Network Security*, pages 99–121, Cham, 2021. Springer International Publishing.

[77] Brian Hie, Hyunghoon Cho, and Bonnie Berger. Realizing private and practical pharmacological collaboration. *Science*, 362(6412):347–350, 2018.

[78] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

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

[80] Nils Homer, Szabolcs Szelinger, Margot Redman, David Duggan, Waibhav Tembe, Jill Muehling, John V. Pearson, Dietrich A. Stephan, Stanley F. Nelson, and David W. Craig. Resolving individuals contributing trace amounts of dna to highly complex mixtures using high-density snp genotyping microarrays. *PLOS Genetics*, 4(8):1–9, 08 2008.

[81] M. Horstein. Review of 'low-density parity-check codes' (gallager, r. g.; 1963). *IEEE Trans. Inf. Theor.*, 10(2):172, sep 2006.

[82] Karthik A Jagadeesh, David J Wu, Johannes A Birgmeier, Dan Boneh, and Gill Bejerano. Deriving genomic diagnoses without revealing patient genomes. *Science (New York, N.Y.)*, 357(6352):692–695, aug 2017.

[83] Somesh Jha, Louis Kruger, and Vitaly Shmatikov. Towards Practical Privacy for Genomic Computation. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 216–230, USA, 2008. IEEE Computer Society.

[84] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. Secure Outsourced Matrix Computation and Application to Neural Networks. In *Proceedings of the 2018 ACM SIGSAC CCS*, pages 1209–1222, 2018.

[85] Petteri Jokinen and Esko Ukkonen. Two algorithms for approxmate string matching in static texts. In Andrzej Tarlecki, editor, *Mathematical Foundations of Computer Science 1991*, pages 240–248, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[86] Liina Kamm, Dan Bogdanov, Sven Laur, and Jaak Vilo. A New Way to Protect Privacy in Large-Scale Genome-Wide Association Studies. *Bioinformatics*, 29(7):886–893, apr 2013.

[87] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC Cryptography and Network Security Series. CRC Press, 2020.

[88] Amit Kaushal, Russ Altman, and Curt Langlotz. Geographic distribution of us cohorts used to train deep learning algorithms. *Jama*, 324(12):1212–1213, 2020.

[89] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1575–1590, 2020.

[90] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for mpc. Cryptology ePrint Archive, Report 2014/137, 2014. `https://ia.cr/2014/137`.

[91] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. Taco: A tool to generate tensor algebra kernels. In *Proc. IEEE/ACM Automated Software Engineering*, pages 943–948. IEEE, 2017.

[92] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. Crypten: Secure multi-party computation meets machine learning. *Advances in Neural Information Processing Systems*, 34, 2021.

[93] Can Kockan, Kaiyuan Zhu, Natnatee Dokmai, Nikolai Karpov, M Oguzhan Kulekci, David P Woodruff, and S Cenk Sahinalp. Sketching algorithms for genomic data analysis and querying in a secure enclave. *Nature methods*, 17(3):295–301, 2020.

[94] Qing Lan, Chao A Hsiung, Keitaro Matsuo, Yun-Chul Hong, Adeline Seow, Zhaoming Wang, H Dean Hosgood, Kexin Chen, Jiu-Cun Wang, Nilanjan Chatterjee, Wei Hu, Maria Pik Wong, Wei Zheng, Neil Caporaso, Jae Yong Park, Chien-Jen Chen, Yeul Hong Kim, Young Tae Kim, Maria Teresa Landi, Hongbing Shen, Charles Lawrence, Laurie Burdett, Meredith Yeager, Jeffrey Yuenger, Kevin B Jacobs, I-Shou Chang, Tetsuya Mitsudomi, Hee Nam Kim, Gee-Chen Chang, Bryan A Bassig, Margaret Tucker, Fusheng Wei, Zhihua Yin, Chen Wu, She-Juan An, Biyun Qian, Victor Ho Fun Lee, Daru Lu, Jianjun Liu, Hyo-Sung Jeon, Chin-Fu Hsiao, Jae Sook Sung, Jin Hee Kim, Yu-Tang Gao, Ying-Huang Tsai, Yoo Jin Jung, Huan Guo, Zhibin Hu, Amy Hutchinson, Wen-Chang Wang, Robert Klein, Charles C Chung, In-Jae Oh, Kuan-Yu Chen, Sonja I Berndt, Xingzhou He, Wei Wu, Jiang Chang, Xu-Chao Zhang, Ming-Shyan Huang, Hong Zheng, Junwen Wang, Xueying Zhao, Yuqing Li, Jin Eun Choi, Wu-Chou Su, Kyong Hwa Park, Sook Whan Sung, Xiao-Ou Shu, Yuh-Min Chen, Li Liu, Chang Hyun Kang, Lingmin Hu, Chung-Hsing Chen, William Pao, Young-Chul Kim, Tsung-Ying Yang, Jun Xu, Peng Guan, Wen Tan, Jian Su, Chih-Liang Wang, Haixin Li, Alan Dart Loon Sihoe, Zhenhong Zhao, Ying Chen, Yi Young Choi, Jen-Yu Hung, Jun Suk Kim, Ho-Il Yoon, Qiuyin Cai, Chien-Chung Lin, In Kyu Park, Ping Xu, Jing Dong, Christopher Kim, Qincheng He, Reury-Perng Perng, Takashi Kohno, Sun-Seog Kweon, Chih-Yi Chen, Roel Vermeulen, Junjie Wu, Wei-Yen Lim, Kun-Chieh Chen, Wong-Ho Chow, Bu-Tian Ji, John K C Chan, Minjie Chu, Yao-Jen Li, Jun Yokota, Jihua Li, Hongyan Chen, Yong-Bing Xiang, Chong-Jen Yu, Hideo Kunitoh, Guoping Wu, Li Jin, Yen-Li Lo, Kouya Shiraishi, Ying-Hsiang Chen, Hsien-Chih Lin, Tangchun Wu, Yi-Long Wu, Pan-Chyr Yang, Baosen Zhou, Min-Ho Shin, Joseph F Fraumeni, Dongxin Lin, Stephen J Chanock, and Nathaniel Rothman. Genome-wide association analysis identifies new lung cancer susceptibility loci in never-smoking women in Asia. *Nature Genetics*, 44(12):1330–1335, 2012.

[95] J. Langford, L. Li, and A. Strehl. Vowpal wabbit: Your go-to interactive machine learning library, 2007.

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

[97] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program

analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, Palo Alto, California, 2004.

[98] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021.

[99] Ingoo Lee, Jongsoo Keum, and Hojung Nam. Deepconv-dti: Prediction of drug-target interactions via deep learning with convolution on protein sequences. *PLoS computational biology*, 15(6):e1007129, 2019.

[100] Yongwoo Lee, Joon-Woo Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and HyungChul Kang. High-Precision Bootstrapping for Approximate Homomorphic Encryption by Error Variance Minimization. In *Eurocrypt*, 2022.

[101] William Mitchell Leiserson. *Defining scalable high performance programming with DEF*. PhD thesis, Massachusetts Institute of Technology, 2020.

[102] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. Federated learning: Challenges, methods, and future directions. *IEEE signal processing magazine*, 37(3):50–60, 2020.

[103] Yi Lian, Xiaoqian Jiang, and Qi Long. Federated multiple imputation for variables that are missing not at random in distributed electronic health records. *medRxiv*, 2024.

[104] Roderick J. A. Little and Donald B. Rubin. Statistical analysis with missing data, third edition. *Wiley Series in Probability and Statistics*, 2019.

[105] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. *Proceedings - IEEE Symposium on Security and Privacy*, 2015-July:359–376, 2015.

[106] Yunan Luo, Yun William Yu, Jianyang Zeng, Bonnie Berger, and Jian Peng. Metagenomic binning through low-density hashing. *Bioinformatics (Oxford, England)*, 35(2):219–226, jan 2019.

[107] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On Ideal Lattices and Learning with Errors over Rings. In *EUROCRYPT*, 2010.

[108] Derrick F MacFabe. Short-chain fatty acid fermentation products of the gut microbiome: implications in autism spectrum disorders. *Microbial ecology in health and disease*, 23(1):19260, 2012.

[109] Ani Manichaikul, Josyf C. Mychaleckyj, Stephen S. Rich, Kathy Daly, Michèle Sale, and Wei-Min Chen. Robust relationship inference in genome-wide association studies. *Bioinformatics*, 26(22):2867–2873, 10 2010.

[110] Deven McGraw. Building public trust in uses of health insurance portability and accountability act de-identified data. *Journal of the American Medical Informatics Association*, 20(1):29–34, 2013.

[111] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. Exploiting unintended feature leakage in collaborative learning. In *Proceedings of IEEE Symposium on Security and Privacy (SP)*, pages 691–706, 2019.

[112] Fernando Meyer, Adrian Fritz, Zhi-Luo Deng, David Koslicki, Till Robin Lesker, Alexey Gurevich, Gary Robertson, Mohammed Alser, Dmitry Antipov, Francesco Beghini, Denis Bertrand, Jaqueline J. Brito, C. Titus Brown, Jan Buchmann, Aydin Buluç, Bo Chen, Rayan Chikhi, Philip T. L. C. Clausen, Alexandru Cristian, Piotr Wojciech Dabrowski, Aaron E. Darling, Rob Egan, Eleazar Eskin, Evangelos Georganas, Eugene Goltsman, Melissa A. Gray, Lars Hestbjerg Hansen, Steven Hofmeyr, Pingqin Huang, Luiz Irber, Huijue Jia, Tue Sparholt Jørgensen, Silas D. Kieser, Terje Klemetsen, Axel Kola, Mikhail Kolmogorov, Anton Korobeynikov, Jason Kwan, Nathan LaPierre, Claire Lemaitre, Chenhao Li, Antoine Limasset, Fabio Malcher-Miranda, Serghei Mangul, Vanessa R. Marcelino, Camille Marchet, Pierre Marijon, Dmitry Meleshko, Daniel R. Mende, Alessio Milanese, Niranjan Nagarajan, Jakob Nissen, Sergey Nurk, Leonid Oliker, Lucas Paoli, Pierre Peterlongo, Vitor C. Piro, Jacob S. Porter, Simon Rasmussen, Evan R. Rees, Knut Reinert, Bernhard Renard, Espen Mikal Robertsen, Gail L. Rosen, Hans-Joachim Ruscheweyh, Varuni Sarwal, Nicola Segata, Enrico Seiler, Lizhen Shi, Fengzhu Sun, Shinichi Sunagawa, Søren Johannes Sørensen, Ashleigh Thomas, Chengxuan Tong, Mirko Trajkovski, Julien Tremblay, Gherman Uritskiy, Riccardo Vicedomini, Zhengyang

Wang, Ziye Wang, Zhong Wang, Andrew Warren, Nils Peder Willassen, Katherine Yelick, Ronghui You, Georg Zeller, Zhengqiao Zhao, Shanfeng Zhu, Jie Zhu, Ruben Garrido-Oter, Petra Gastmeier, Stephane Hacquard, Susanne Häußler, Ariane Khaledi, Friederike Maechler, Fantin Mesny, Simona Radutoiu, Paul Schulze-Lefert, Nathiana Smit, Till Strowig, Andreas Bremges, Alexander Sczyrba, and Alice Carolyn McHardy. Critical Assessment of Metagenome Interpretation: the second round of challenges. *Nature Methods*, 2022.

[113] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

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

[115] Arturo Moncada-Torres, Frank Martin, Melle Sieswerda, Johan van Soest, and Gijs Geleijnse. VANTAGE6: an open source priVAcy preserviNg federaTed leArninG infrastructurE for Secure Insight eXchange. In *AMIA Annual Symposium Proceedings*, pages 870–877, 2020.

[116] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin Butler, and Patrick Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. *Proceedings - 2016 IEEE European Symposium on Security and Privacy, EURO S and P 2016*, pages 112–127, 2016.

[117] Christian Mouchet, Juan R. Troncoso-pastoriza, Jean-Philippe Bossuat, and Jean Pierre Hubaux. Multiparty homomorphic encryption from ring-learning-with-errors. In *Proceedings on Privacy Enhancing Technologies Symposium*, 2021.

[118] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, March 2008.

[119] Rachel Nosowsky and Thomas J Giordano. The health insurance portability and accountability act of 1996 (HIPAA) privacy rule: implications for clinical research. *Annu. Rev. Med.*, 57(1):575–590, 2006.

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

[121] Lucila Ohno-Machado, Zia Agha, Douglas S Bell, Lisa Dahm, Michele E Day, Jason N Doctor, Davera Gabriel, Maninder K Kahlon, Katherine K Kim, Michael Hogarth, et al. pscanner: Patient-centered scalable national network for effectiveness research. *Journal of the American Medical Informatics Association*, 21(4):621–626, 2014.

[122] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, pages 223–238, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[123] David Pellow, Itzik Mizrahi, and Ron Shamir. PlasClass improves plasmid sequence classification. *PLoS Computational Biology*, 16(4):1–9, 2020.

[124] Vitor C Piro, Temesgen H Dadi, Enrico Seiler, Knut Reinert, and Bernhard Y Renard. ganon: precise metagenomics classification against large and up-to-date sets of reference sequences. *Bioinformatics*, 36(Supplement_1):i12–i20, 07 2020.

[125] Richard Platt, Jeffrey S Brown, Melissa Robb, Mark McClellan, Robert Ball, Michael D Nguyen, and Rachel E Sherman. The fda sentinel initiative—an evolving national resource. *N Engl J Med*, 379(22):2091–2093, 2018.

[126] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[127] Alvin Rajkomar, Michaela Hardt, Michael D Howell, Greg Corrado, and Marshall H Chin. Ensuring fairness in machine learning to advance health equity. *Annals of internal medicine*, 169(12):866–872, 2018.

[128] Jaak Randmets. *Programming languages for secure multi-party computation application development*. PhD thesis, PhD Thesis, University of Tartu, 2017.

[129] Aseem Rastogi, Matthew A Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *2014 IEEE Symposium on Security and Privacy*, pages 655–670. IEEE, 2014.

[130] Ron L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[131] Adam Sadilek, Luyang Liu, Dung Nguyen, Methun Kamruzzaman, Stylianos Serghiou, Benjamin Rader, Alex Ingerman, Stefan Mellem, Peter Kairouz, Elaine O Nsoesie, et al. Privacy-first health research with federated learning. *NPJ digital medicine*, 4(1):132, 2021.

[132] Michael Salib. *Starkiller: A static type inferencer and compiler for Python*. PhD thesis, Massachusetts Institute of Technology, 2004.

[133] Hiroki Sato, Akira Umayabara, Yu Ishimaki, and Hayato Yamana. Poster: Loop circuit optimization with bootstrapping over fully homomorphic encryption. In *2 nd IEEE European Symp. on Security and Privacy*, 2017.

[134] Sinem Sav, Apostolos Pyrgelis, Juan R Troncoso-Pastoriza, David Froelicher, Jean-Philippe Bossuat, Joao Sa Sousa, and Jean-Pierre Hubaux. POSEIDON: Privacy-Preserving Federated Neural Network Learning. *NDSS*, 2021.

[135] Eric E Schadt, Sangsoon Woo, and Ke Hao. Bayesian method to predict individual snp genotypes from gene expression data. *Nature Genetics*, 44(5):603–608, 2012.

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

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

[138] Ariya Shajii, Ibrahim Numanagić, Riyadh Baghdadi, Bonnie Berger, and Saman Amarasinghe. Seq: A high-performance language for bioinformatics. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[139] Ariya Shajii, Ibrahim Numanagić, Riyadh Baghdadi, Bonnie Berger, and Saman Amarasinghe. Seq: a high-performance language for bioinformatics. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[140] Ariya Shajii, Ibrahim Numanagić, Alexander T Leighton, Haley Greenyer, Saman Amarasinghe, and Bonnie Berger. A Python-based programming language for high-performance computational genomics. *Nature Biotechnology*, 39(9):1062–1064, 2021.

[141] Ariya Shajii, Gabriel Ramirez, Haris Smajlović, Jessica Ray, Bonnie Berger, Saman Amarasinghe, and Ibrahim Numanagić. Codon: A compiler for high-performance pythonic applications and dsls. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, CC 2023, page 191–202, New York, NY, USA, 2023. Association for Computing Machinery.

[142] Suyash S Shringarpure and Carlos D Bustamante. Privacy risks from genomic data-sharing beacons. *The American Journal of Human Genetics*, 97(5):631–646, 2015.

[143] Montgomery Slatkin. Linkage disequilibrium — understanding the evolutionary past and mapping the medical future. *Nature Reviews Genetics*, 9(6):477–485, 2008.

[144] Haris Smajlović, Ariya Shajii, Bonnie Berger, Hyunghoon Cho, and Ibrahim Numanagić. Sequre: a high-performance framework for secure multiparty computation enables biomedical data sharing. *Genome Biology*, 24(1):5, 2023.

[145] Haris Smajlović, Ariya Shajii, Bonnie Berger, Hyunghoon Cho, and Ibrahim Numanagić. Sequre: a high-performance framework for rapid development of secure bioinformatics pipelines. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 164–165, 2022.

[146] Ebrahim M. Songhori, Siam U. Hussain, Ahmad Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly compressed and scalable sequential Garbled Circuits. *Proceedings - IEEE Symposium on Security and Privacy*, 2015-July:411–428, 2015.

[147] Kai Tian, Mingyu Shao, Yang Wang, Jihong Guan, and Shuigeng Zhou. Boosting compound-protein interaction prediction by deep learning. *Methods*, 110:64–72, 2016.

[148] Peter J Turnbaugh and Jeffrey I Gordon. The core gut microbiome, energy balance and obesity. *The Journal of physiology*, 587(17):4153–4158, 2009.

[149] Stef Van Buuren. *Flexible imputation of missing data*. CRC press, 2018.

[150] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. HECO: Fully homomorphic encryption compiler. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4715–4732, 2023.

[151] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. Sok: Fully homomorphic encryption compilers. *Proceedings - IEEE Symposium on Security and Privacy*, 2021-May:1092–1108, 1 2021.

[152] Zhiyu Wan, James W Hazel, Ellen Wright Clayton, Yevgeniy Vorobeychik, Murat Kantarcioglu, and Bradley A Malin. Sociotechnical safeguards for genomic data privacy. *Nature Reviews Genetics*, pages 1–17, 2022.

[153] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. `https://github.com/emp-toolkit`, 2016.

[154] Brian J Wells, Kevin M Chagin, Amy S Nowacki, and Michael W Kattan. Strategies for handling missing data in electronic health record derived data. *Egems*, 1(3), 2013.

[155] Ming Wen, Zhimin Zhang, Shaoyu Niu, Haozhi Sha, Ruihan Yang, Yonghuan Yun, and Hongmei Lu. Deep-learning-based drug–target interaction prediction. *Journal of Proteome Research*, 16(4):1401–1409, 2017. PMID: 28264154.

[156] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. Suif: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not.*, 29(12):31–37, December 1994.

[157] Jie Xu, Benjamin S Glicksberg, Chang Su, Peter Walker, Jiang Bian, and Fei Wang. Federated learning for healthcare informatics. *Journal of healthcare informatics research*, 5:1–19, 2021.

[158] 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.

[159] Samee Zahur and David Evans. Obliv-C: A Language for Extensible Data-Oblivious Computation. *Cryptology ePrint Archive*, 2015.

[160] Yihua Zhang, Aaron Steele, and Marina Blanton. Picco: A general-purpose compiler for private distributed computation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, page 813–826, New York, NY, USA, 2013. Association for Computing Machinery.

[161] Yiliang Zhang and Qi Long. Assessing fairness in the presence of missing data. *Advances in neural information processing systems*, 34:16007–16019, 2021.

[162] Yiliang Zhang and Qi Long. Fairness in missing data imputation. *arXiv preprint arXiv:2110.12002*, 2021.

[163] Wenting Zheng, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Helen: Maliciously Secure Coopetitive Learning for Linear Models. In *IEEE S&P*, 2019.

[164] Ligeng Zhu, Zhijian Liu, and Song Han. Deep leakage from gradients. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, volume 32, 2019.

[165] Roberto V Zicari, James Brusseau, Stig Nikolaj Blomberg, Helle Collatz Christensen, Megan Coffee, Marianna B Ganapini, Sara Gerke, Thomas Krendl Gilbert, Eleanore Hickman, Elisabeth Hildt, et al. On assessing trustworthy ai in healthcare. machine learning as a supportive tool to recognize cardiac arrest in emergency calls. *Frontiers in Human Dynamics*, 3:673104, 2021.

[166] Alexander Ziller, Andrew Trask, Antonio Lopardo, Benjamin Szymkow, Bobby Wagner, Emma Bluemke, Jean-Mickael Nounahon, Jonathan Passerat-Palmbach, Kritika Prakash, Nick Rose, and Others. PySyft: A Library for Easy Federated Learning. In *Federated Learning Systems*, pages 111–139. Springer, 2021.

Appendix A

A Short, Practical Guide to Sequre

We illustrate the usage of Sequre by implementing a secure version of PlassClass [123], a binary classification tool for distinguishing whether a sequence originates from a plasmid sequence or a chromosomal segment. It is based on a binary classification with k-mer counts as feature variables. This example describes how to securely perform the training of the classification model in an SMC environment. Secure evaluation of the trained model on private data for inference can be achieved in a similar manner.

A.1 Data Pre-processing and Secret Sharing

Each data owner prepares the private data to be securely shared with the computing parties. PlassClass begins by counting the number of k-mers (for a predefined k) over a given set of chromosomal and plasmid sequences. These counts will be used as the classifier features. Each data owner can execute this step locally, as it does not require any coordination with other parties. Because Sequre inherits all the functionalities of the Seq language [139] for high-performance bioinformatics pipelines [140], we can use such functionality to quickly implement the k-mer counting in only 10 lines of code (Figure A.1). Once the classifier features are constructed, we can proceed with *secret sharing*—the process of dividing the private data among the untrusted computing parties without disclosing any private information to each party. We do so by calling Sequre's secret sharing routine (Figure A.2).

Sequre's secret sharing protocol defaults to additive secret sharing [52], which means that one needs to add all shares together to reveal the underlying data. Each data owner executes the secret sharing routine locally to construct the shares. The

```
features = zeros(len(labels), 2 ** 10).to_int()  # k-mer length is 5

def update(label_idx, kmer):
  features[label_idx][int(min(kmer, ~kmer).as_int())] += 1

for fasta_path, label in zip(fastas, labels):
  for seq in seqs(FASTA(fasta_path, fai=False)):
    for kmer in seq.kmers[Kmer[5]](1):
      update(label, kmer)
print("Data preprocessing done!")
```

Figure A.1: PlassClass preprocessing. Note that the original implementation of PlassClass required more than 150 lines of Python code.

```
from sequre import secret_share
secret_share(features, labels)
```

Figure A.2: Sequre's secret sharing routine.

constructed shares are then distributed to the computing parties via secure channels. Finally, each client compiles and runs the secret-sharing procedure (Figure A.3).

```
$ sequre client.seq

Setting up Sequre ...
Compiling client.seq ...
Field size: 170141183460469231731687303715884105727
Ring size: 170141183460469231731687303715884105728

Data preprocessing done!
Connected at 127.0.0.1:9090!
Connected at 127.0.0.1:9091!
Connected at 127.0.0.1:9092!
Secret sharing done!
```

Figure A.3: Client compile and run instruction.

A.2 Configuring the Network

Sequre defaults to a `localhost` address. However, each owner (or a *client*) can reconfigure network addresses for the SMC computing parties and the SMC trusted dealer within `dsl/settings.seq` file in Sequre's main directory. Note that this step requires all data holders to agree on the same configuration—Sequre terminates the execution if a mismatch is found between the configurations.

Also, we will assume that the servers used by computing parties are properly deployed and configured (the specifics of deployment are discussed at the end of this section). The following code listing shows an example configuration for a local network of two computing parties running on the same machine:

```
# IPs
TRUSTED_DEALER = '127.0.0.1'  # Trusted dealer / Auxiliary party
COMPUTING_PARTIES = [
  '127.0.0.1',  # 1st computing party
  '127.0.0.1'   # 2nd computing party
]
```

Figure A.4: Network configuration.

A.3 Secure Training

Once the data is secretly shared, we can initialize secure training. We will use a linear support vector machine (SVM) classifier for binary classification of sequences. For simplicity, we will optimize regularized hinge loss via the stochastic gradient descent algorithm for our linear SVM. The secure implementation of this procedure in Sequre does not differ much from the straightforward linear SVM pseudocode (see Section 4 for details):

```
from sequre import dot, zeros_like

mpc, (features, labels) = pool_shares()

@sequre
def lsvm(mpc, X, Y, eta, epochs, l2):
  w = zeros_like(X[0]) + 1
  b = zeros_like(Y[0]) + 1

  for i in range(epochs):
    for feature_vector, label in zip(X, Y):
      z = dot(mpc, feature_vector, w) - b
      # Backward pass
      grad_b = label * ((1 - z * label) > 0)
      grad_w = w * l2 * 2 - feature_vector * grad_b
      w = w - grad_w * eta
      b = b - grad_b * eta

  return w, b
```

Figure A.5: Linear SVM training.

In the above code, note how the `lsvm` procedure is decorated by a `@sequre` decorator. This decorator signals to the compiler that the code within the function needs to be executed in an SMC environment. Specifically, it needs to be executed using a fixed set of protocols provided by the `mpc` instance (the first argument of the `lsvm` procedure) that contains the instantiated SMC environment. This environment uses the same settings that data holders used during the secure sharing.

The training procedure can be used as follows:

```
from sequre import pool_shares
mpc, (features, labels) = pool_shares()

print("Training the linear SVM.")

weights = lsvm(
  mpc, features, labels.flatten(),
  eta=0.01, epochs=10, l2=0.01
)
print(f"First 10 weights at CP{mpc.pid}: {weights[:10].print(mpc)}")
mpc.done()
```

Figure A.6: PlassClass training in Sequre.

This code will gather the secretly shared data, instantiate the SMC environment (line 2), train a binary classification model on the private data (line 6), reveal the trained weights to the end-users (line 10), and finally notify the data holders and the end-users that the training is complete (line 11). Note that the above binary classification routine is actually a part of Sequre's standard library and can be easily used in any pipeline through a simple import statement.

A.4 Complete SMC Implementation of PlassClass

Here are the final 42 lines of code for our secure SMC version of PlassClass: 15 lines of code for the secret sharing (Figure A.7) program and 27 lines for the training (Figure A.8). Note that the same pipeline would be hundreds of lines of code long if SMC-related optimizations had to be done by hand, even if the multiplication and comparison procedures were provided by Sequre's standard library. Finally, note the original non-secure PlassClass implementation includes over 300 lines of code in Python.

```
from bio import seqs, Kmer, FASTA
from sequre import secret_share, zeros

features = zeros(len(labels), 2 ** 10).to_int()  # k-mer length is 5

def update(label_idx, kmer):
  features[label_idx][int(min(kmer, ~kmer).as_int())] += 1

for fasta_path, label in zip(fastas, labels):
  for seq in seqs(FASTA(fasta_path, fai=False)):
    for kmer in seq.kmers[Kmer[5]](1):
      update(label, kmer)
print("Data preprocessing done!")

secret_share(features, [labels])
```

Figure A.7: PlassClass client call.

A.5 Deployment

So far, we have covered the client's perspective of deploying the secure pipeline (i.e., configuring the network, secret sharing and pipeline execution). Currently S,equre assumes that the participating computing parties are deployed and accessible on the network. The secure pipeline (`server.seq` in our example) needs to be sent to the computing parties for execution after the code review is completed (expected to be performed by the data owner or a third party responsible for ensuring security compliance). Users can choose between compiling the source code privately ahead of time and deploying the executable(s) to the servers or deploying the source code to the servers and using just-in-time compilation for execution. The exact method of deploying the code to the servers is currently left to the users. Once the code is deployed, each computing party executes it independently. Upon detecting the secure code blocks (e.g., `import sequre` or a `@sequre` decorator), the compiler will prepend the necessary SMC setup procedures to the codebase. This will allow servers to set up secure communication channels between each other as well as the pseudo-random generators needed for reducing the network bandwidth upon execution. Finally, after the environment is set, secure computing will commence. The output on each computing node should roughly resemble the following output:

```
from sequre import sequre, pool_shares
from sequre import dot, zeros_like

mpc, (features, labels) = pool_shares()

@sequre
def lsvm(mpc, X, Y, eta, epochs, l2):
  w = zeros_like(X[0]) + 1
  b = zeros_like(Y[0]) + 1

  for i in range(epochs):
    for feature_vector, label in zip(X, Y):
      z = dot(mpc, feature_vector, w) - b
      # Backward pass
      grad_b = label * ((1 - z * label) > 0)
      grad_w = w * l2 * 2 - feature_vector * grad_b
      w = w - grad_w * eta
      b = b - grad_b * eta

  return w, b

print("Training the linear SVM.")

weights, bias = lsvm(mpc, features, labels.flatten(),
                     eta=0.01, epochs=10, l2=0.01)
print(f"First 10 weights at CP{mpc.pid}: {weights[:10].print(mpc)}")
mpc.done()
```

Figure A.8: PlassClass server call.

```
$ sequre server.seq 1

Setting up Sequre ...
Compiling server.seq ...
Field size: 170141183460469231731687303715884105727
Ring size: 170141183460469231731687303715884105728

Connected at 127.0.0.1:9001
Listening at 0.0.0.0:9003
Listening at 0.0.0.0:9091
Initialized MPC at CP1

Training the linear SVM.
Epoch: 1/10
Epoch: 2/10
Epoch: 3/10
Epoch: 4/10
Epoch: 5/10
Epoch: 6/10
Epoch: 7/10
Epoch: 8/10
Epoch: 9/10
Epoch: 10/10
First 10 weights at CP1:
    [-7.62409, 5.62368, -1.00295, 0.983992, 9.47111,
     4.6789, 6.42332, 4.0334, 1.43668, 0.998597]
```

Figure A.9: PlassClass training output from the first computing party.

Appendix B

Other Applications: Machine Learning Module

Our machine learning module consists of dimensionality reduction algorithms like principle component analysis, regression algorithms, support vector machines, and a neural network module. Most of these algorithms have been already applied in our applications above. We used PCA to build GWAS, linear SVMs to support metagenomic binning, neural networks for the DTI algorithm, and regression analysis in Secure MICE. Most of these algorithms have already been explained in the respective sections. Here, we provide additional details about our SVM and Neural Network modules.

B.1 Linear Support Vector Machines

The linear SVM from the implementation of metagenomic binning (Section 3.3) in Sequre is also a part of Sequre's and Shechi's standard library. It is a binary classification algorithm that uses stochastic gradient descent to minimize the regularized Hinge loss. To be more specific, we minimize the loss $l \in \mathbb{R}$ for $L : \mathbb{R}^n \to \mathbb{R}$ such that

$$l = L(\mathbf{w}, b) = \lambda \|\mathbf{w}\|^2 + max(0, 1 - t(\mathbf{w}^T \cdot \mathbf{x} - b)))$$

where $\mathbf{w} \in \mathbf{R}^n$, $\mathbf{x} \in \mathbf{R}^n$, $b \in \mathbf{R}$, and and $t \in \{0, 1\}$ are respectively the regression weights vector, input feature vector, the bias, and the truth value.

We need to minimize l with respect to $\mathbf{w}$ and b via stochastic gradient descent. Thus we need to iteratively translate $\mathbf{w}$ and b by a predefined step size $\eta \in \mathbb{R}$ in the

negative direction of a gradient of $L(\mathbf{w}, b)$ as follows:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta \nabla_{\mathbf{w}_k} L(\mathbf{w}_k, b)$$

$$b_{k+1} = b_k - \eta \nabla_{b_k} L(\mathbf{w}_k, b)$$

where $\mathbf{w}_0$ and b_0 can be picked at random. Note that

$$\nabla_{\mathbf{w}} L(\mathbf{w}, b) = 2\lambda \mathbf{w} - t \cdot \left[(1 - t(\mathbf{w}^T \cdot \mathbf{x} - b)) > 0\right] \cdot \mathbf{x}$$

$$\nabla_b L(\mathbf{w}, b) = t \cdot \left[(1 - t(\mathbf{w}^T \cdot \mathbf{x} - b)) > 0\right].$$

The pseudocode for the described procedure is provided in Algorithm 12. The secure SMC variant of the same algorithm is provided in Algorithm 13.

Algorithm 12 Linear SVM training
INPUT:
features $\in \mathbb{R}^{m \times n}$: features matrix
labels $\in \mathbb{R}^m$: list of labels
eta$(\eta) \in \mathbb{R}$: step size
epochs $\in \mathbb{Z}$: number of epochs
lambda$(\lambda) \in \mathbb{R}$: L_2 regularization factor
OUTPUT:
$w \in \mathbb{R}^n$ and $b \in \mathbb{R}$: a weights vector and a bias that minimize the Hinge loss of the classifier output

---

1: $\mathbf{w} \leftarrow (1, 1, \ldots 1) \in \mathbb{R}^n$
2: $b \leftarrow 1$
3: **for** $i \in \overline{0, \text{epochs}}$ **do**
4: **for** $\mathbf{x} \in \text{features} \wedge t \in \text{labels}$ **do**
5: $y \leftarrow \mathbf{w} \cdot \mathbf{x} - b$
6: $\mathbf{w} \leftarrow \mathbf{w} \cdot (1 - 2\lambda\eta) + \eta \cdot t \cdot ((1 - t \cdot y) > 0) \cdot \mathbf{x}$
7: $b \leftarrow b - \eta \cdot t \cdot ((1 - t \cdot y) > 0)$
8: **end for**
9: **end for**

Figure B.1 shows that Sequre source code does not differ much from the Algorithm 12.

B.2 Neural Networks

```
from sequre import dot, zeros_like

mpc, (features, labels) = pool_shares()

@sequre
def lsvm(mpc, X, Y, eta, epochs, l2):
  w = zeros_like(X[0]) + 1
  b = zeros_like(Y[0]) + 1

  for i in range(epochs):
    for feature_vector, label in zip(X, Y):
      z = dot(mpc, feature_vector, w) - b
      # Backward pass
      grad_b = label * ((1 - z * label) > 0)
      grad_w = w * l2 * 2 - feature_vector * grad_b
      w = w - grad_w * eta
      b = b - grad_b * eta

  return w, b
```

Figure B.1: Linear SVM training in Sequre/Shechi.

```
from sequre import dot

@sequre
def lsvm_predict(mpc, x, w, b):
  return dot(mpc, x, w) - b
```

Figure B.2: Linear SVM inference in Sequre/Shechi.

Sequre/Shechi includes a neural network module that implements the popular Keras's [35] API to facilitate secure machine learning workflows on distributed data. This module 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. B.3. Shechi was able to perform training of privacy-preserving credit score evaluation [158] for approximately 100K individuals with 16 features in a practical runtime of 7 hours. Using a similar codebase, we also evaluated drug-target interaction inference [77] from section 3.3.2 using MHE within the same time span. 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 in MHE contexts and will require further advances in cryptographic techniques to be incorporated into our MHE framework.

Algorithm 13 Linear SVM training in SMC

INPUT:

$[\mathsf{features}] \in \mathbb{Z}_p^{m \times n}$ - secret shared features matrix

$[\mathsf{labels}] \in \mathbb{Z}_p^{m}$ - secret shared list of labels

$\mathsf{eta} \in \mathbb{Z}_p$ - public step size

$\mathsf{epochs} \in \mathbb{Z}$ - public number of epochs

$\mathsf{lambda} \in \mathbb{Z}_p$ - L_2 regularization factor

OUTPUT:

$[\mathsf{w}] \in \mathbb{Z}_p^n$ and $[\mathsf{b}] \in \mathbb{Z}_p$ - secret shared weights vector and a bias that minimize the Hinge loss of the classifier output

1: $[\mathbf{w}] \leftarrow \text{secret_share}((1, 1, \ldots, 1))$
2: $[b] \leftarrow \text{secret_share}(1)$
3: **for** $i \in \overline{0, \mathsf{epochs}}$ **do**
4: **for** $[\mathbf{x}] \in [\mathsf{features}] \wedge [t] \in [\mathsf{labels}]$ **do**
5: $[y] \leftarrow \text{beaver_dot_product}([\mathbf{w}], [\mathbf{x}])$
6: $[y] \leftarrow \text{truncate}([y]) - b$
7: $[c] \leftarrow \eta \cdot [t]$
8: $[c] \leftarrow \text{truncate}([c])$
9: $[c_1] \leftarrow (1 - y)$
10: $[s] \leftarrow \text{is_positive}(c_1)$
11: $[c] \leftarrow [c] \cdot [s]$
12: $[\mathbf{v}] \leftarrow \text{beaver_multiply}([\mathbf{x}], [c])$
13: $[\mathbf{v}] \leftarrow \text{truncate}([\mathbf{v}])$
14: $[\mathbf{w}] \leftarrow [\mathbf{w}] \cdot (1 - 2\lambda\eta) + [\mathbf{v}]$
15: $[b] \leftarrow [b] - [c]$
16: **end for**
17: **end for**

```
@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 B.3: Sequre/Shechi's Keras-like neural networks interface enables simple implementation of network training and inference for privacy-preserving credit score evaluation.

Appendix C

Selected Code Listings

```
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)
```

Figure C.1: Shechi's instantiation for secure execution on partitioned data.

```
from shechi import *
from numpy import ones_like

@shechi
def king(mhe, data):
    dot = data * data @ ones(
        data.shape, dtype=float).T
    distance = (dot + dot.T) - data @ data.T * 2
    het_inv = ones_like(data) / data.count(
        1, axis=1)
    return distance * maximum(
        mhe, het_inv, het_inv.T)
```

Figure C.2: Kinship computation between all samples (i.e., rows) in a partitioned matrix.

```
from shechi import *

@shechi
def qr(data):
    v_cache = []
    Q = A.zeros() + A.I
    for i in range(len(A)):
        v = data[0].copy()
        v[0] += v.norm() + v[0].sign()
        v /= v.norm()
        A = (A - (A @ v.T @ v) * 2)[1:, 1:]
        v_cache.append(v)
    for i in range(len(Q) - 1, -1, -1):
        Qsub = Q[:, i:]
        Q[:, i:] = Qsub - (
            Qsub @ v_cache[i].T @ v_cache[i]) * 2
    return Q
```

Figure C.3: QR decomposition.

```
from shechi import *
from lin_alg import qr, eigen_decomp

@shechi
def randomized_pca(mhe, data, pi, pow_it, rho):
    p = pi @ data
    for _ in range(pow_it):
        p = p @ data.T
        r = qr(p)
        p = r @ data
    z = p @ data.T
    z_cov = z @ z.T
    u = z_cov.via_mpc(
        lambda x: eigen_decomp(mhe, x)[0][:rho])
    return u @ z
```

Figure C.4: Shechi-PCA.

```
from shechi import *
from pca import randomized_pca
from lin_alg import qr, dot

@shechi
def cochran_armitage(mhe, V, pheno, dosage):
    pheno_int = pheno.expand_dims()
    pheno_float = pheno_int.to_fp()
    dosage_float = dosage.astype(float)
    p_hat = pheno_float - pheno_int @ V.T @ V
    sp = p_hat.sum(axis=1)[0]
    sx = ((1 - V.T.sum(axis=0
        ).expand_dims() @ V) @ dosage_float)[0]
    spp = dot(mhe, p_hat[0], axis=0)
    sxp = (p_hat @ dosage_float)[0]
    sxx = (dosage * dosage).sum(
        axis=0).to_fp() - dot(
            mhe, V @ dosage_float, axis=0)
    norm_sp = sp / len(pheno)
    numer = sxp - sx * norm_sp
    denom = (sxx - sx * sx / len(pheno)) * (
        spp - sp * norm_sp)
    return numer / sqrt(mhe, denom)

@shechi
def psa(mhe, dosage, cov, rho, pow_it):
    pi = random_sketch(rho, dosage.shape)
    components = randomized_pca(
        mhe, data, pi, pow_it, rho)
    components.extend(cov.T)
    return qr(components)

@shechi
def gwas(mhe, dosage, cov, pheno, rho, pow_it):
    components = psa(
        mhe, dosage, cov, rho, pow_it)
    return cochran_armitage(
        mhe, components, pheno, dosage)
```

Figure C.5: Shechi-GWAS.