ANALYSIS OF TWO VERSATILE MPC FRAMEWORKS MP-SPDZ AND MPYC

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF APPLIED MATHEMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

FATİH AYKURT

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
CRYPTOGRAPHY

DECEMBER 2023

Approval of the thesis:

ANALYSIS OF TWO VERSATILE MPC FRAMEWORKS MP-SPDZ AND MPYC

submitted by **FATİH AYKURT** in partial fulfillment of the requirements for the degree of **Master of Science in Cryptography Department, Middle East Technical University** by,

Prof. Dr. A. Sevtap Kestel
Dean, Graduate School of **Applied Mathematics** __________

Assoc. Prof. Dr. Oğuz Yayla
Head of Department, **Cryptography** __________

Assoc. Prof. Dr. Oğuz Yayla
Supervisor, **Cryptography, METU** __________

Examining Committee Members:

Prof. Dr. Zülfükar Saygı
Mathematics, TOBB ETU __________

Assoc. Prof. Dr. Oğuz Yayla
Cryptography, METU __________

Prof. Dr. Sedat Akleylek
Computer Engineering, Ondokuz Mayıs University __________

Assoc. Prof. Dr. Adnan Özsoy
Computer Engineering, Hacettepe University __________

Assist. Prof. Dr. Buket Özkaya
Cryptography, METU __________

Date: __________

Name, Last Name:	FATİH AYKURT
Signature :	

ABSTRACT

ANALYSIS OF TWO VERSATILE MPC FRAMEWORKS MP-SPDZ AND MPYC

AYKURT, FATİH
M.S., Department of Cryptography
Supervisor : Assoc. Prof. Dr. Oğuz Yayla

December 2023, 44 pages

Using secure multi-party computing protocols (MPC), a group of participants who distrust one another can securely compute any function of their shared secret inputs. Participants exchange these inputs in a manner similar to secret sharing, where each participant owns a portion of the input but is unable to independently reconstruct the complete information without collaborating with the other participants. This kind of computation is quite powerful and has many uses where data privacy is quite critical such as areas like government, business, and academia. MPC has grown from a subject of theoretical study to a technology being employed in industry, becoming effective enough to be deployed in practice with various algorithms implemented with MPC frameworks. In this study, two versatile MPC frameworks, MP-SPDZ and MPyC are analyzed. These frameworks' performances are compared by using algorithms execution times from basic operations to more complex structures like shuffle sort algorithm. Profiling results are also analyzed to reveal the bottleneck points of the algorithms where the time consumption increases drastically. To detect the critical parts easier, profiling results are visualized as dot graphs. Besides all these, in the MPyC framework, Sattolo shuffle algorithm is implemented and compared with the current modern version of Fisher-Yates algorithm.

Keywords: MPC, Profiling, Bottleneck, Benchmark

ÖZ

İKİ ÇOK YÖNLÜ MPC ÇERÇEVESİ MP-SPDZ VE MPYC'NİN ANALİZİ

AYKURT, FATİH
Yüksek Lisans, Kriptografi Bölümü
Tez Yöneticisi : Doç. Dr. Oğuz Yayla

Aralık 2023, 44 sayfa

Çok partili hesaplama(ÇPH), kişisel verilerini paylaşmak istemeyen partilerin, birbirleri arasında güvenli veri paylaşımı yapmasını sağlar. Katılımcılar verilerinin sadece belirli bir kısmını birbirleriyle paylaşır. Paylaşım sonucu her katılımcı verilerin sadece bir kısmına sahip olur ve diğer katılımcıların verilerini de kullanmadan tüm bilgiyi kendi başına elde edemez. Bu sayede katılımcıların kendi veri bütünlüğünü açığa çıkarmasına gerek kalmadan toplu veri paylaşımı yapılabilir. Bu tarz hesaplamalar oldukça verimlidir ve veri güvenliğinin ön planda olduğu devlet işleri, iş dünyası, akademi gibi birçok alanda kullanılabilir. ÇPH artık teorik bir çalışma olmaktan çıkmış, endüstride kullanılabilecek verimli bir teknoloji aracına dönüşmüştür. Bu çalışmada 2 tane etkili ÇPH algoritma çerçevesi, MP-SPDZ ve MPyC analiz edildi. Bu algoritma çerçevelerinin performansları basit operasyonlardan karıştır-sırala gibi karmaşık yapılara kadar farklı algoritmalar kullanılarak karşılaştırıldı. Algoritmalardaki zaman tüketiminin şiddetli bir şekilde arttığı boğum yerlerini ortaya çıkarmak için profil analizi yapıldı. Algoritmalardaki kritik kısımları daha kolay analiz edebilmek için profil analizi sonuçları nokta grafiği formatında görselleştirildi. Tüm bunların yanında MPyC çerçevesinde Sattolo karıştır algoritması implemente edildi ve güncel olarak kullanılan Fisher-Yates algoritmasının modern haliyle karşılaştırıldı.

Anahtar Kelimeler: MPC, Profil Çıkarma, Boğum Yeri, Performans Analizi

ACKNOWLEDGMENTS

I would like to express my very great appreciation to my thesis supervisor Assoc. Prof. Dr. Oguz Yayla for his patient guidance, enthusiastic encouragement and valuable advices during the development and preparation of this thesis. His willingness to give his time and to share his experiences has brightened my path.

Furthermore, I would like to thank my family and my friends for listening to my concerns, believing in me and their supports.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A group of participants that distrust each other can safely compute any function of their shared secret inputs using secure multi-party computation protocols (MPC). These inputs are shared among participants as a manner of secret sharing, where each person owns a portion of the input but cannot reconstruct the entire information independently without working with the other participants. MPC has a broad range of applications from secure financial transactions to healthcare. It allows participants to share their private data and compute jointly without learning nothing but just computation result. To adapt these functionalities to the real world, many MPC frameworks are developed. The utility and implementation of these frameworks were significantly improved by ground-breaking publications. Modern MPC frameworks like MP-SPDZ and MPyC, which provide versatile, high-performance options for secure multi-party computation, were made possible by these developments however the performances still need improvements.

There are various studies focus on performance comparison about MPC frameworks [22], [18], [26]. In these studies, execution times of basic functions and simple algorithms like inner product are used for benchmark. A bunch of ciphers are implemented and their execution time results are detailed for comparison of MPC frameworks. Reasons of the poor performances are explained without diving into sub functions just by analyzing execution times of algorithms.

Performance comparison and enhancement are always hot topic for all areas. One of the ways of the enhancement is revealing the bottleneck points of the algorithms and resolving the over time consumption. With using profiling tools, bottleneck points are detected of the algorithms by measuring the execution time of each call. Using profiling tools to enhance the algorithms by resolving the bottlenecks is a common method in recent studies [20], [8].

When performances of the frameworks are considered, MP-SPDZ is one of the best frameworks however it is not on the top if the usability has more priority. MPyC is better than MP-SPDZ for accessibility and usability so combining these features with improved performance on MPyC framework may increase the usage of it in real world applications. For the improvement, we try to find out the reasons behind poor performance with profiling analysis. In the literature, we didn't encounter any profiling analysis about these two MPC frameworks.

By combining execution times and profiling results, we aimed to show the performance difference of these two frameworks and reveal bottleneck points of the implemented algorithms if exists. Showing the statistics of all the function calls and their effects on the overall execution times is the motivation of this study.

1.1 Historical Process and Literature Review

Multi-Party Computation originated at the intersection of computer science, cryptography and concerns about privacy. The principle of secure multiparty computation was originally formalized in the early 1980s, which is when MPC frameworks began to evolve historically. This concept was first proposed in Andrew Yao's 1982 article "Protocols for Secure Computations," [45] which is where it all began. It has introduced as "Yao's Millionaires' Problem". The protocol allowes two millionaires to compare their wealth without revealing related data.

Following years, researchers developed effective algorithms for secure two-party computation, which were essential to secure e-commerce transactions and computations that preserved privacy. The research of Oded Goldreich and Charles Rackoff on "Secure Multi-Party Computation" [16] and the creation of cryptographic primitives like Oblivious Transfer and Zero-Knowledge Proofs are two examples. More practical and versatile MPC frameworks developments are inevitable. The evolution of MPC frameworks over time shows a progression from theoretical ideas to the useful, effective and adaptable frameworks present today.

There are several studies on MPC frameworks about their performance or usability comparison. These works are the basis our experiment. The study "General purpose compilers for secure multi-party computation"[18] compares eleven systems: EMP-toolkitç[41], Obliv-C[46], ObliVM[25], TinyGarble[37], SCALE-MAMBA[1], Wysteria[30], Sharemind[6], PICCO[47], ABY[12], Frigate[29] and CBMC-GC[14]. It focuses on usability by evaluating language expressibility, capabilities of the cryptographic back-end and accessibility to developers. Implementation of inner product is one of their test algorithms. The study "Performance Comparison of Two Generic MPC-frameworks with Symmetric Ciphers"[26] compares two frameworks MP-SPDZ and MPyC by implementing a bunch of Symmetric Ciphers like AES, ChaCha20, Trivium etc. It presents the benchmark results of basic operations. Then it shares the performance analysis of the ciphers by evaluating execution times without any profiling so any bottlenecks can't be revealed. The study "MP-SPDZ: A Versatile Framework for Multi-Party Computation"[22] focuses the comparison MP-SPDZ with other frameworks like ABY[12], Fresco[19], OblivC[46] etc. They calculated execution times of inner product implementations. They apply many protocols like Shamir[35], Malicious Shamir[7], Yao's Garbled Circuits[4] etc. Most of the cases, MP-SPDZ presents better performance. This research also focuses on only execution times without any further study like profiling.

While evaluating the performance of the frameworks, execution time analysis is very usefully however it is not enough to find out the reason of the performance difference and improve the poor performances. Profiling is the common method for this purpose. Code profiling enables us to analyze the algorithms and reveals whether there is a bottleneck that spends too much time or not. There are many studies that use profiling method to improve specific tasks in different fields. The study "Optimizing a medical image registration algorithm based on profiling data for real-time performance"[20] suggests improvements based on the dot graph visualization and detects performance bottlenecks on a complex algorithm with gprof profiling. The study "Parallel Collision Detection with OpenMP"[8] tries to improve the performance of a collision detection algorithm by analyzing its profiling data. They reduce the number of instructions with the help of gprof2dot and speed up the process.

1.2 Contribution of thesis

As mentioned above sections, there various studies about MPC frameworks. Benchmarks provide only execution time analysis. Their results don't present the behaviour inside the functions. Bottlenecks of the algorithms stay unknown. Our aim is taking these studies a step further. We present a new perspective for comparison of MPC frameworks. We used profiling tools to reveal the reasons of MPC frameworks poor performance in specific algorithms. We presented the statistics of every function call with cProfile tool. To analyze these data, we visualized them with gprof and gprof2dot tools. After the profiling results are obtained, a well known shuffle algorithm, Sattolo Shuffle is implemented in the MPyC to add a new usability feature to the framework.

1.3 Outline

In this thesis, two Multi Party Computation[24] frameworks, MP-SPDZ[22] and MPyC [34] are studied. Their performance and profiling characteristics are analyzed, bottleneck of the algorithms are focused. Sattolo shuffle algorithm implementation is presented.

Chapter 2 presents an introduction to MPC. Uses cases of MPC and one of the main protocols, Shamir Secret Sharing [35] are mentioned. Features and current studies about MPyC and MP-SPDZ are briefly described.

Chapter 3, we compute and compare the executions times of algorithms in both frameworks. Analysis of inner product, shuffle-sort algorithms and basic operations like addition and multiplication are our objective.

Chapter 4 demonstrates the profiling results of the frameworks. Using cProfiling tool, we focused algorithms bottlenecks to find whether there is a part that needs optimization or not. Using gProf and gProf2dot tools, execution paths are visualized. Sattolo shuffle implementation in MPyC is presented

CHAPTER 2

MULTI PARTY COMPUTATION

MPC can be thought of as a cryptographic technique that offers trusted-party functionality without the need for reciprocal trust—a trusted party that would accept secret inputs, perform a function, and deliver the output to the stakeholders. As a result, it is guaranteed that no participant learns anything through using the protocol that they couldn't have known from the output alone. For example, a computation will be applied to two secret values like comparison, addition etc. Firstly, these two secret values are divided into n piece where n is the party number. All pieces shared among parties and each party have n secret pieces. Shamir secret sharing [35] is a common method used for this secret sharing. None of the parties can't evaluate the main secret value without enough pieces. Desired computation is applied to all pieces without revealing the secret values.Finally secret values are reconstructed with computation results and MPC is completed. Lagrange interpolation[11] is one the common reconstruction method.

2.1 Real-World Applications of MPC

MPC has a broad range of applications. In secure financial transactions, it enables parties to jointly perform financial operations without revealing sensitive details, ensuring the privacy of transactions. In healthcare, it facilitates privacy-preserving analytic on patient data, allowing medical researchers to gain insights without compromising patient confidentiality [36]. Secure voting systems use MPC to ensure the integrity of the voting process while preserving anonymity. One of the real world example is Boston wage gap [23]. MPC is used in 2017 to calculate the salaries of 166,705 workers from 114 companies. Since companies weren't going to share their raw data because of privacy concerns, the usage of MPC was essential. The research results revealed that the gender wage gap in the Boston area is significantly greater than predicted. This is an impressive example of how MPC can be applied for the benefit of society.

2.2 Shamir's Secret Sharing

Shamir's Secret Sharing [35] is a fundamental cryptographic method that is essential to secure sharing of information and multi-party computation. The fundamental ideas and importance of Shamir's Secret Sharing system are explained in this subsection.

Shamir's Secret Sharing is a method for sharing a secret into a number of pieces that can only be merged when an agreed-upon amount of shares are combined. It was developed by Adi Shamir in 1979. Polynomial interpolation is the main idea. Consider a scenario in which shares are generated using the coordinates of a secret, which is represented as a point on a polynomial curve. On this curve, there is a point for each share. Polynomial interpolation is utilized to find the curve's equation, which reveals the secret, and a minimum number of shares equal to or exceeding a predefined threshold must be acquired in order to reconstruct the secret.

An adversary cannot discover the secret as long as they hold fewer shares than the required amount. Due to this, it is a useful technique in MPC where parties seek to execute computations simultaneously without revealing their private inputs. Shamir's technique also offers resilience against share defects or losses; even if some shares are corrupted or lost, the secret can still be rebuilt as long as the required number of valid shares is present.

2.3 MP-SPDZ

Multi-Party Secure Protocol for Data-Zero-Knowledge (MP-SPDZ) is a cutting-edge framework designed for secure multi-party computation. It emerges from the intersection of cryptography, computer science and distributed systems to address the critical issue of computing functions while preserving privacy among multiple parties. MP-SPDZ stands out due to its strong focus on efficiency and usability. This framework enables multiple parties to jointly compute a function over their private inputs without revealing any sensitive information to one another. MP-SPDZ's underlying principle is to break down computations into smaller operations then securely combine the results using cryptographic techniques.

2.3.1 Features and Architecture of MP-SPDZ

MP-SPDZ boasts several remarkable features that make it a robust choice for secure multi-party computation. It provides support for arithmetic operations over various data types including integers and floating-point numbers. Moreover, it excels in scalable protocols, which means it can handle computations involving numerous parties efficiently. Additionally, MP-SPDZ offers flexibility by accommodating different security models and threat scenarios. This adaptability makes it versatile for use in both research and practical implementations.

2.3.2 Architecture

The architecture of MP-SPDZ is structured around a client-server model. The clients represent the parties involved in the computation, while servers handle the heavy cryptographic computations. This separation of roles ensures a practical division of labor, with clients responsible for input and output handling and servers managing the secure computation. MP-SPDZ employs advanced cryptographic primitives like homomorphic encryption and secret sharing schemes to ensure data privacy. The system is designed to be modular, allowing for easy integration of new protocols and improvements, which is crucial for keeping pace with evolving security requirements. Overall, MP-SPDZ's architecture is a well-thought-out balance of efficiency, security and adaptability making it a prominent choice in the MPC landscape.

2.3.3 Use Cases of MP-SPDZ

MP-SPDZ finds practical application in various domains where secure computations are essential. One of its prominent use cases is in privacy-preserving data analysis, especially in situations where multiple parties need to collaborate without disclosing sensitive information. For example, healthcare institutions can use MP-SPDZ to jointly analyze patient data from different hospitals while ensuring that individual patient records remain confidential. Similarly, financial institutions can utilize MP-SPDZ to detect fraudulent activities across multiple banks without sharing customer transaction details.

Another significant application is in secure voting systems. MP-SPDZ can be employed to create a trustful and tamper-proof voting process where voters can cast their ballots privately and the final election results can be computed securely without revealing individual votes. This ensures the integrity of the voting process and protects against coercion or bribery.

Due to its high performance, MP-SPDZ is preferred in many different fields for secure computation. The study "Secure integer division with a private divisor"[38] offers a solution to secure integer division within a secret-sharing based MP-SPDZ framework. The study "SAFEFL: MPC-friendly Framework for Private and Robust Federated Learning"[15] aims to develop more efficient Federated learning (FL)[28] systems. They also used MP-SPDZ framework, which implements various MPC protocols. The study "RPM: Robust Anonymity at Scale" [27] presents a scalable anonymous communication protocol suite using MPC with the offline-online model. They have implemented their protocols using the MP-SPDZ.

2.4 MPyC

The MPyC Python package provides a user-friendly environment for building MPC protocols and is designed for secure computing tasks. Researchers and practitioners have both taken notice of MPyC, which was developed with a focus on simplicity and usability. By allowing many parties to work together on data analysis and processing while maintaining the privacy of their inputs, MPyC supports secure computations. To maintain privacy and security during computation, it makes use of modern cryptographic techniques including secret sharing and homomorphic encryption.

2.4.1 Features and Architecture of MPyC

The strength of MPyC depends in its extensive feature set and clear architectural design. The framework's high degree of adaptability makes it possible for users to quickly and efficiently perform a variety of secure compute tasks. Support of multiple cryptographic primitives, a high-level Python interface and compatibility with numerous secure computation protocols are a few of the important characteristics. Its architecture takes advantage of Python's power and simplicity, making it usable by both security professionals and beginners.

A trusted provider is in charge of starting the safe computation in MPyC's network of parties, each of which has its own secret data inputs. The framework ensures the privacy of these parties' inputs while allowing safe communication and computation between them. MPyC is a useful tool for researchers, developers and organizations looking to take advantage of secure multi-party computation since it offers a simple and expressive programming model that makes the construction of secure protocols easier.

2.4.2 Use Cases of MPyC

A variety of use cases in many different sectors are possible by MPyC's versatility. For instance, MPyC can be used in the healthcare industry for secure medical data analysis, enabling hospitals and academics to collaborate on patient data studies without revealing private data. In the field of finance, MPyC provides secure financial computations involving numerous stakeholders without revealing sensitive financial data, such as portfolio optimization or risk evaluation.Additionally, MPyC is utilized in secure machine learning, where parties can collaborate to train models on their individual private datasets.

Due to its usability and high compatibility, MPyC is preferable among different fields. The study "Differentially-Private Multi-Party Sketching for Large-Scale Statistics"[9] offers a solution to compute aggregate statistics on large amounts of sensitive data while protecting the privacy of individual users by using MPyC framework. The study "Privacy-Preserving Contrastive Explanations with Local Foil Trees"[39] provides a secure algorithm for machine learning models with implementations in MPyC.

CHAPTER 3

BENCHMARKS

Benchmark serves as a fundamental component of this research, enabling a performance analysis of MPC frameworks, MP-SPDZ and MPyC. The primary purposes of benchmark within this study are as follows:

1. **Performance Comparison:** The main goal of benchmark is to compare MP-SPDZ and MPyC performance across different relevant algorithms. We obtain significant understanding into how these frameworks differ to one another by giving both of them the same set of tasks and evaluating the time it takes for each task to be completed. This is crucial for ones who are looking to choose the best framework for particular MPC applications.
2. **Identifying Strengths and Weaknesses:** Benchmark enables us to identify the execution speed and resource efficiency of MP-SPDZ and MPyC. We can identify which framework performs best in specific tasks or scenarios through analyzing benchmark data and we can also identify potential spots may require optimization. Developers can efficiently focus their optimization efforts by analyzing what sections of the frameworks are the most significant contributions to total execution time.

From basic operations to complex algorithms, execution times are measured. Inner product algorithm, shuffle-sort algorithm and basic operations like multiplication, addition etc. are implemented and analyzed. Shamir secret sharing protocol is applied for all algorithms with 64-bit integers and MacBook Pro with M2 chip, 8-core CPU, 8GB of RAM is the hardware configuration.

3.1 Basic Operations

Basic operations form the basis of all algorithms. Their combinations provide complex functionalities. Following chapters introduce inner product and shuffle sort algorithms which consist of many basic functions. Analyzing the behaviour of basics helps to understand the complex structures. In our work, we calculated total execution times of 100 times of each basic operations, addition, multiplication, reduce addition and reduce multiplication for both MPyC and MP-SPDZ frameworks. Besides we added vector operations for MPyC. Table 3.1 shows how many seconds it takes to compute 100 basic operations of input length from 1 to 100.

Table 3.1: Basic Operations Execution Times

n	MP-SPDZ	MPyC	MPyC Vec
	map(operator.add, a, b)		vector_add(a, b)
1	<1	7	11
10	<1	14	24
100	<1	72	59
	map(operator. mul, a, b)		schur_prod(a, b)
1	7	32	34
10	7	86	54
100	9	455	119
	reduce(operator.add, a)		mpc.sum(a)
1	<1	<1	16
10	<1	68	23
100	<1	507	35
	reduce (operator mul, a)		mpc.prod(a)
1	<1	<1	22
10	38	180	106
100	366	1671	178

Functionality of the basic operations are:

- **Schur_prod**: The secure entry-wise multiplication of two vectors
- **Reduce operation**: Applies a function of two arguments cumulatively to the items of a sequence or iterable, from left to right, so as to reduce the iterable to a single value. For example:

$$reduce(lambda\, x, y:\ x + y,\ [1, 2, 3, 4, 5]) \quad calculates$$

$$r(((1 + 2) + 3) + 4) + 5)$$

In MP-SPDZ, addition time is very small, less than 1 millisecond. It increases with a rising n size for multiplication. The unvectorized computations grow linearly with vector size in MPyC. The vectorized operations scale more quickly, although only the sum scales sublinearly. The other operations observe a slowdown as the size of their input vectors increase.

Algorithm 1 shows a secure multiplication operation in MP-SPDZ. Firstly, the variables x and y values are assigned from an external file called "Input_0" and "Input_1". While receiving inputs, they are converted to secure types by "sint" operation. x and y variables become secure integers. Their values can't be read by an external user. Only the owners of input files know the values. Then secure multiplication is done with overloaded "*" operation and final result is obtained. Final result is also secure since it is the result of a secure process. To reveal the final result, it has to be converted from secure type to insecure one. By "reveal()" operation, result is converted from secure to insecure one and its readable by anyone. This process shows only secure operation steps. Secret sharing mechanism isn't included here. For that purpose, many protocols like Shamir Secret Sharing are used.

Algorithm 1 Secure Multiplication Algorithm in MP-SPDZ

```
  $x \leftarrow sint.get\_raw\_input\_from(Input\_0)$
  $y \leftarrow sint.get\_raw\_input\_from(Input\_1)$
  $result \leftarrow x * y$
  $print\_ln('\%s', result.reveal())$
```

Algorithm 2 shows a secure multiplication operation in MPyC. Firstly, insecure integers 5 and 7 are converted to secure integers by "SecInt()" operation. Then secure multiplication is done by overloaded "*" operation. To convert the secure result to insecure integer, mpc.run() operation is called and the result is readable by all users.

Algorithm 2 Secure Multiplication Algorithm in MPyC

```
$x \leftarrow mpc.SecInt(5)$
$y \leftarrow mpc.SecInt(7)$
$result \leftarrow x * y$
$final\_result \leftarrow mpc.run(result)$
print($final\_result$)
```

Evaluating basic operations execution times and using them for comparison approach is also used in the study "Performance Comparison of Two Generic MPC-frameworks with Symmetric Ciphers"[26] to compare MPyC and MP-SPDZ performance. Same basic operations execution times are compared. We want to ensure from our implementations by comparing our results with this study. The relation between size and execution time is similar. MPyC simple operations are worse than both MP-SPDZ and MPyC vectored operations however there is an improvement in the execution times. Compared to mentioned study, our calculated execution times almost 4 times less in the size of 100 in the addition and multiplication operations. In the mpc.sum operation, the results are opposite. Our execution time is 2 times greater. Other operations have similar times.

3.2 Inner Product

Compare to basic operations, more complex algorithms provide better understanding for the performance analysis. Inner product is the second analyzed algorithm. It takes 2 insecure inputs, converts them to secure ones. Then inner product of the inputs is calculated. It is converted back to insecure type and outputted. Figure 3.1 shows how many seconds it takes to compute an inner product of length from 10,000 to 1,000,000.

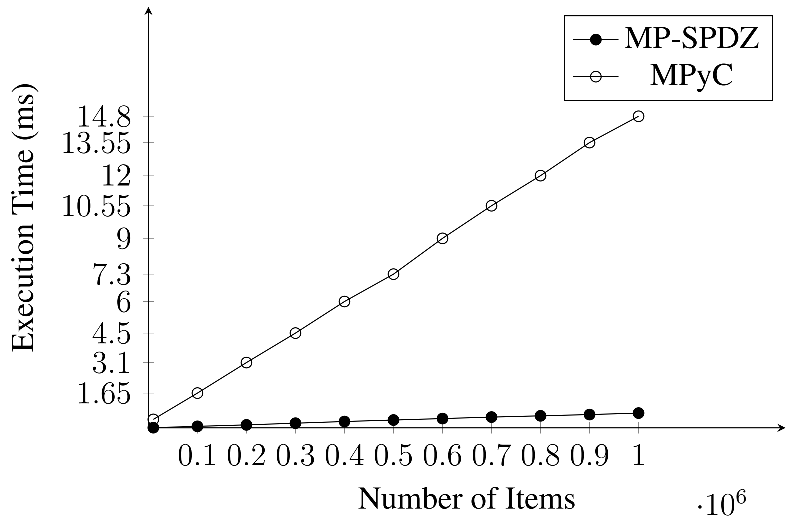


Figure 3.1: Inner Product Algorithm Execution Times in MPyC and MP-SPDZ

There is a linear relationship between time and length for both frameworks. In the basic operations, performance difference is obvious however when more complex structures are compared, the difference becomes more striking. MP-SPDZ is about 20 times faster than MPyC with a 1000000 input size and the difference increases with increasing size.

Inner product algorithm execution time analysis is used also in the study "MP-SPDZ: A Versatile Framework for Multi-Party Computation"[22]. This study compares MP-SPDZ with other MPC frameworks. They implemented the same algorithm and used its execution time for performance comparison however they focused only the results of the execution. Improvement of the poor performances isn't their field of study contrary to us. One of our motivation is to take this study a step further by profiling the algorithm and find out any possible bottleneck.

3.3 Shuffle-Sort

Shuffle and sort algorithms are key components of many larger secure computation protocols. Analysis of the shuffle-sort algorithm has a huge importance in our work. By evaluating this algorithms execution times in both framework, performance comparison becomes more meaningful and precise. Secure sorting protocols allow two (or more) participants to privately sort a list of n secret-shared[35] values without revealing any data about the underlying values to any of the participants. The algorithm inputs a set of insecure values and converts them to secure ones. To ensure the randomness, firstly shuffles the input set randomly. Then the secure set is sorted and outputted.

3.3.1 MP-SPDZ

We aimed to have an idea on the overall performance with benchmark results before applying the profiling tools. Firstly we analyzed the MP-SPDZ framework. Shuffle algorithm uses Waksman permutation network[40], [3]. Waksman permutation networks are built using “controlled-swap-gates” which take two inputs and a “control bit” that determines whether to swap the two inputs. In the second part, MP-SPDZ sort algorithm bases from radix sort. Its implementation and performance are detailed in the study "Oblivious Radix Sort: An Efficient Sorting Algorithm for Practical Secure Multi-party Computation[5]". The overall algorithm is implemented and execution time is evaluated. Figure 3.2 which shows how many seconds it takes to compute a shuffle-sort algorithm of length from 100 to 2100 in the MP-SPDZ framework.

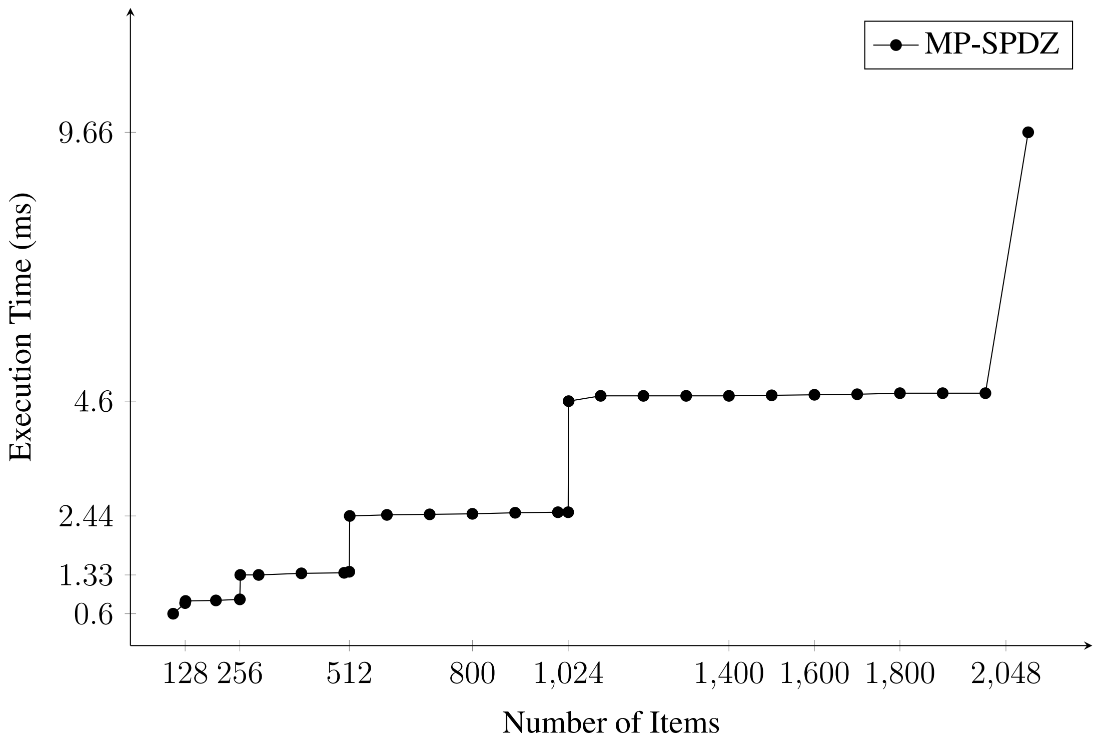


Figure 3.2: Shuffle-Sort Algorithm Execution Time In MP-SPDZ

According to the originated study[5], the proposed algorithm time complexity is O(nlogn). Time vs size of input graph should has a smooth increasing line however in our work, we obtained a non-smooth, stair like line. Figure 3.3 shows both expected and our results. Expected result data are taken from the originated study[5]. The reason of this behaviour is the shuffle implementation. Sort algorithm also uses shuffle and Waksman networks are the base of shuffle. Waksman network is a permutation network capable of n! permutation of its n input terminals to its n output terminals. The building blocks for this network are binary cells capable of permuting their two input terminals to their two output terminals. The complexity of shuffle increases by powers of two so with every power of 2 of input length, execution time almost doubles and until the next power, it almost stays the same. Without doubling, the results are almost matched with the expected ones.

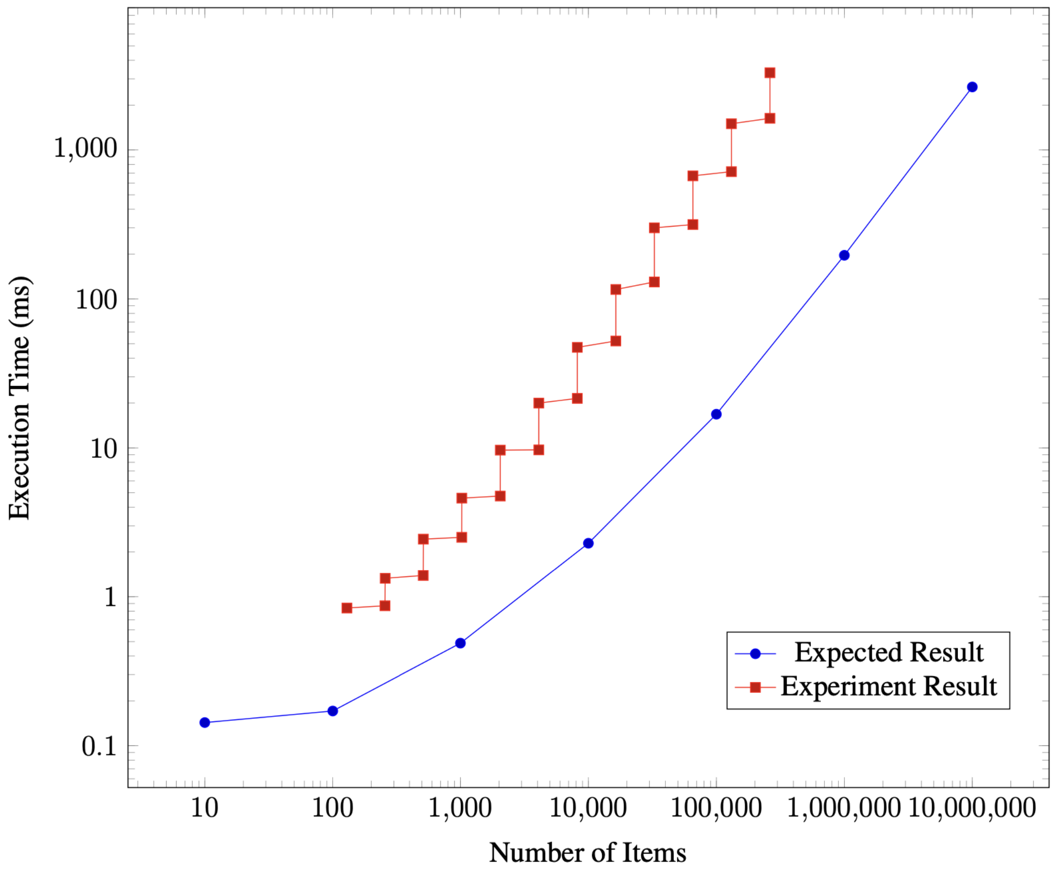


Figure 3.3: Shuffle-Sort Algorithm Execution Time In MP-SPDZ and Radix Sort

In the multi party computations, nonlinear behaviours may cause weaknesses. MP-SPDZ framework uses a nonlinear shuffle algorithm and this nonlinear behaviour creates a weakness. If the execution time result of different input lengths are known, the relation between input lengths may be estimated. For example, if the traitor gets two different execution time results, according to the ratio between them, the power difference of input lengths can be estimated. There are linear solutions for shuffle however their execution time performance is worse. The study "Efficient Secure Three-Party Sorting with Applications to Data Analysis and Heavy Hitters"[2] offers a perfectly linear approach however it is only described for honest majority and only for three parties. This nonlinear behaviour is an issue to be resolved for MP-SPDZ. A linear and higher performance shuffle algorithm is essential.

3.3.2 MPyC

Besides the MP-SPDZ analysis, for the performance comparison, we analyzed the same algorithm in the MPyC framework. Algorithm3 shows an example of shuffle-sort with input size 10. # parts are comment lines to show the output of variables.

Algorithm 3 Shuffle-Sort Algorithm in MPyC

```
$n \leftarrow 10$
$s \leftarrow [(-1)**i*(i+n//2)**2\ for\ i\ in\ range(n)]$
#s  [25, -36, 49, -64, 81, -100, 121, -144, 169, -196]
$secnum \leftarrow mpc.SecInt()$
$x \leftarrow list(map(secnum, s))$
$async\ with\ mpc$ :
    $mpc.random.shuffle(secnum, x)$
    $await\ mpc.output(x)$
    #output  [81, 49, -144, 169, -64, -100, 25, -36, -196, 121]
    $x \leftarrow mpc.sorted(x)$
    $await\ mpc.output(x)$
    #output  [-196, -144, -100, -64, -36, 25, 49, 81, 121, 169]
```

Before shuffle sort calls, input list has to be converted to secure integers for MPC. Secure conversion means that the values are distributed among parties with secret sharing protocols. Default party number is 3 and MPyC uses Shamir[35] secret sharing. "async" "await" structure is used since if the sort operation starts before the first output() operation finished, list x already started to be sorted before shufle result is outputted and output doesn't give true result. Figure 3.4 shows how many seconds it takes to compute a shuffle-sort algorithm of length from 100 to 6000 in the both frameworks. MPyC execution times have a linear relationship with n size since MPyC uses Batcher's odd-even mergesort[21]. This approach has a linear cost however its complexity is $O(nlog^2n)$ and its clearly seen from the Figure3.4.

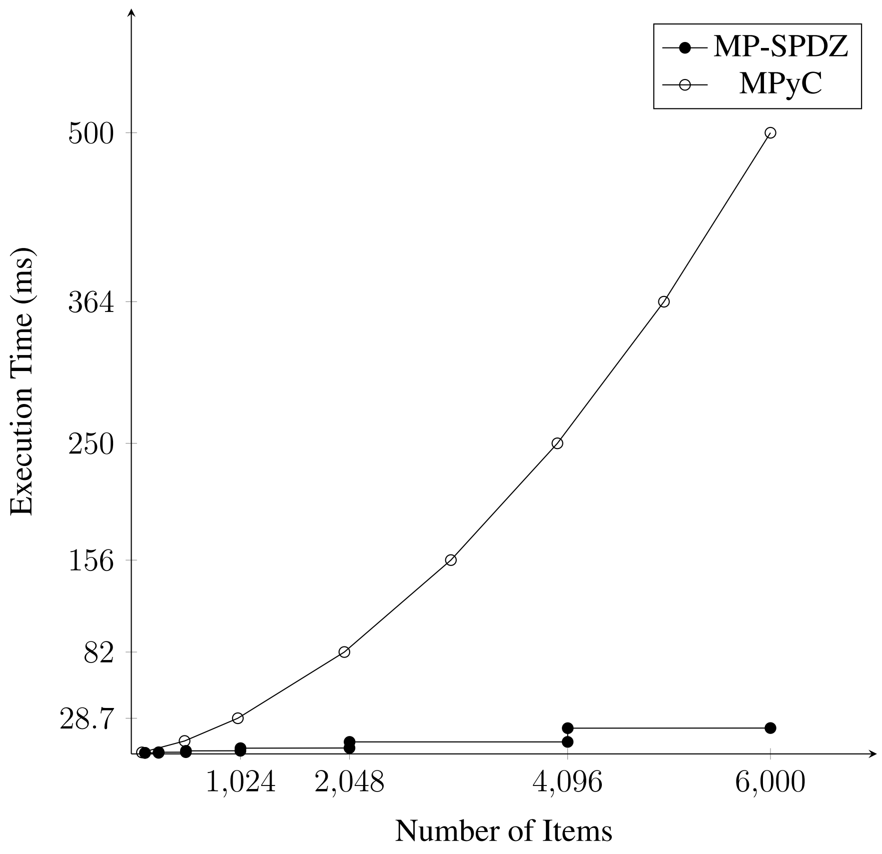


Figure 3.4: Shuffle-Sort Algorithm Execution Time In MP-SPDZ and MPyC

CHAPTER 4

PROFILING

Code profiling is a software analysis technique used to measure and analyze the performance characteristics of an algorithm during its execution. The aim is identifying bottlenecks for optimization. It involves set of statistics that describes how often and for how long various parts of the program are executed. This data helps developers and analysts identify areas in the code that may require optimization. There are several ways of profiling. In this work, cProfiling, gprof[17] and gprof2dot[13] tools are used. “cProfile” is a built-in profiler for Python. It allows you to measure the execution time of different parts of your Python code. Advantages of cProfiling as follows:

- It provides the overall execution time of the whole code
- It gives the time spent of each individual call. You may compare and determine which components require optimization using this.
- The number of times each function call is also presented
- Using pstats module, the useful data can be extracted. Then this data can be visualised for better analysis.

“gprof” is a profiling tool provided by the GNU Compiler Collection (GCC) for analyzing the performance programs. gprof2dot” is a Python script that converts the output of gprof into a graphical representation, typically in the form of a call graph or dot graph. In our study, using cProfile and pstats modules, profiling data of the code is extracted and saved as pstats file. This file holds all execution time statistics of the functions.Then pstats file is converted to a dot file with gprof module. For visualization of the data, gprof2dot module converts the dot file into a call graph to a png file. Call graph represents the code execution path from main function to basic operations like addition, multiplication. It also provides percentage of the execution times of all functions by comparing them.

In the Chapter 3, executions times of the algorithms and basic operations are shared separately. In this chapter, we will provide the profiling statistics of the algorithms by showing how often the basic operations are called and how much their percentages are in the execution time. This progress reveals the bottlenecks of the algorithms if exist. Applying the profiling methods to MPyC framework is more meaningful since it has worse performance than MP-SDPZ in all benchmark results. Following sections present our analysis about inner product and shuffle sort on MPyC. Basic operations profiling is nonsense since they are on the bottom of the algorithm. They forms the basis. There is not any further call after them. Nevertheless, we share the call graphs of the basic functions to support this claim. In the Figures 4.9, 4.10, 4.11, 4.12, 4.13 and 4.14 shows the graphs of basic calls. They don't have a significant call block. Most of the execution times are belong to input and output phases.

4.1 Inner Product

We analyzed Inner Product algorithm with profiling tools to get a deeper insight. In the beginning, to get the execution times of each function call, line profiling is used however only the functions which are located on the main page times are obtained. We are not able to see whether there are any bottleneck points in any of the functions. Then the method is switched to cProfiling. It allows us to see all function calls and their execution times.

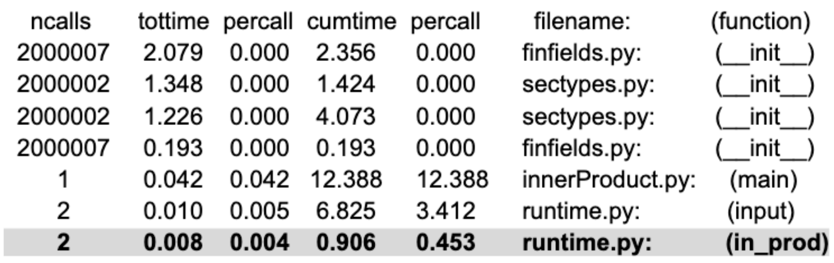
cProfile tool outputs are the followings:

- **ncalls**: The number of times a specific code block was called during the profiling
- **tottime**: Total time spent executing the specific code block, excluding the time

spent in its sub-functions.

- **percall**: Average time spent per call to the specific function. It's calculated by dividing the tottime by the ncalls.
- **cumtime**: Cumulative time spent in the specific function and all its sub-functions. It includes the time spent in the sub-functions.
- **percall**: Average cumulative time spent per call to the specific function. It's calculated by dividing the cumtime by the ncalls.
- **filename**: This indicates the name of the file where the function or code block is defined.

Figure 4.1 shows cProfiling results of the Inner Product algorithm in MPyC framework. The results are sorted according the tottime of the calls.



```
 ncalls   tottime  percall  cumtime  percall   filename:         (function)
2000007    2.079   0.000    2.356    0.000    finfields.py:      (__init__)
2000002    1.348   0.000    1.424    0.000    sectypes.py:       (__init__)
2000002    1.226   0.000    4.073    0.000    sectypes.py:       (__init__)
2000007    0.193   0.000    0.193    0.000    finfields.py:      (__init__)
      1    0.042   0.042   12.388   12.388    innerProduct.py:    (main)
      2    0.010   0.005    6.825    3.412    runtime.py:         (input)
      2    0.008   0.004    0.906    0.453    runtime.py:         (in_prod)
```

Figure 4.1: Profiling Results of Inner Product

Functionality of the calls are:

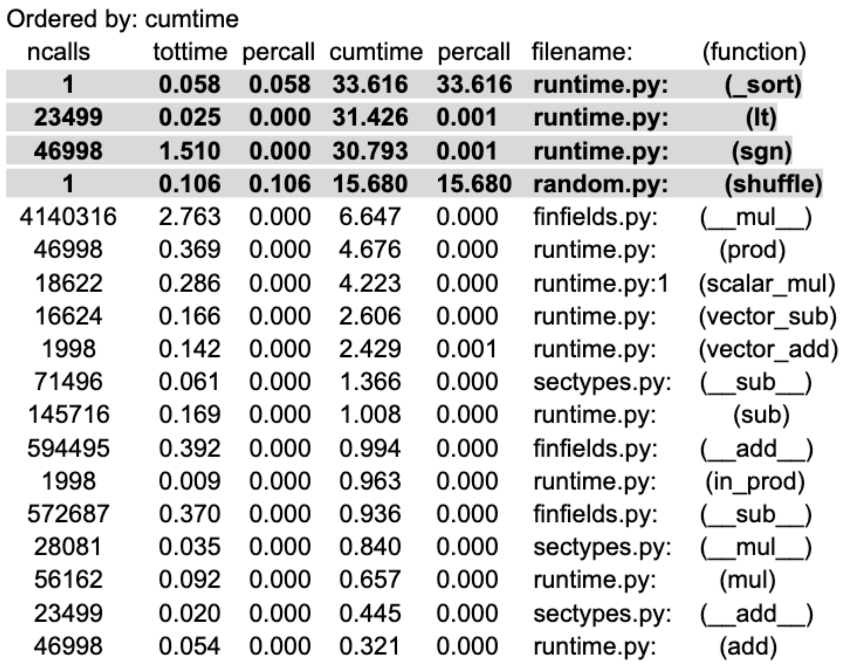
- **__init__()**: Convert the insecure inputs to secure ones
- **input()**: Input x to the computation
- **in_prod()**: Securely calculate the inner product of the inputs

The algorithm doesn't have any basic function call. Initialization calls have a dominance on the execution time. Almost %33 of the total time belongs to init functions. Init functions are mainly responsible for initialization of secure integers and initialization of shares. The functions assign secret values to integers to use them in the secret sharing phase. Secret values of shares for each party are also assigned for the computation part. The key call is "in_prod()" however it has just 8 milliseconds call time. Even with its subfuntions, its cumulative time is %7 of the total execution time. Figure4.8 supports these analysis. It tells us the call path of the algorithm from main function to basic operations. Blue painted blocks have feasible execution times. Other colors represent the increasing usage of time. Clearly seen that "in_prod()" call is on the side of blue blocks. In the bottom, init functions are colored green and light blue which represents the most effective parts in the whole calls. We didn't face with any bottlenecks in the analysis however optimization on the init functions may have an impressive effect on the total execution time.

4.2 Shuffle-Sort

The profiling analysis of shuffle sort algorithm is one of our main contributions. In the benchmark chapter, we showed the performance difference between frameworks for the shuffle sort algorithm. In addition to that analysis, with profiling results, we aimed to find out at least one of the reasons of MPyC poor performance. Compared to inner product, shuffle-sort has more complex inner structure. With this algorithm, we created a use case which consists of many basic operations. Too much function calls increase the probability of a bottleneck where the execution is stuck and spends redundant time. By analyzing the statistics of the calls, we aimed to reveal the bottlenecks if they exist. In the Figure 4.2 shows cProfiling results of a shuffle-sort algorithm of length 100,000 in MPyC. The results are sorted according to the cumulative time of the calls.

Sort() and shuffle() functions are the key calls. Their total execution time equals the overall execution time. Init functions don't have an important effect in contrast to inner product.



```
Ordered by: cumtime
 ncalls     tottime  percall  cumtime  percall  filename:       (function)
    1        0.058    0.058   33.616   33.616   runtime.py:      (_sort)
 23499       0.025    0.000   31.426   0.001    runtime.py:        (lt)
 46998       1.510    0.000   30.793   0.001    runtime.py:       (sgn)
    1        0.106    0.106   15.680   15.680   random.py:       (shuffle)
4140316      2.763    0.000    6.647   0.000    finfields.py:   (__mul__)
 46998       0.369    0.000    4.676   0.000    runtime.py:       (prod)
 18622       0.286    0.000    4.223   0.000    runtime.py:1    (scalar_mul)
 16624       0.166    0.000    2.606   0.000    runtime.py:     (vector_sub)
  1998       0.142    0.000    2.429   0.001    runtime.py:     (vector_add)
 71496       0.061    0.000    1.366   0.000    sectypes.py:    (__sub__)
145716       0.169    0.000    1.008   0.000    runtime.py:        (sub)
594495       0.392    0.000    0.994   0.000    finfields.py:   (__add__)
  1998       0.009    0.000    0.963   0.000    runtime.py:      (in_prod)
572687       0.370    0.000    0.936   0.000    finfields.py:   (__sub__)
 28081       0.035    0.000    0.840   0.000    sectypes.py:    (__mul__)
 56162       0.092    0.000    0.657   0.000    runtime.py:       (mul)
 23499       0.020    0.000    0.445   0.000    sectypes.py:    (__add__)
 46998       0.054    0.000    0.321   0.000    runtime.py:       (add)
```

Figure 4.2: Profiling Results of Shuffle-Sort are sorted according to cumulative time

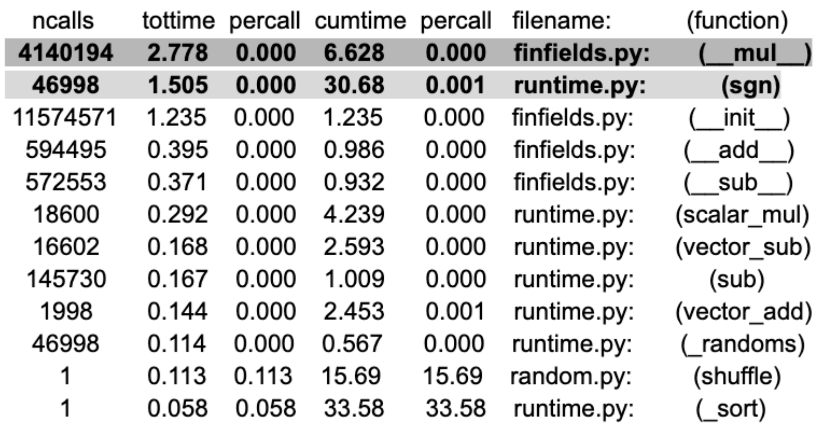
Functionality of the other calls are:

- **lt()**: Convert the insecure inputs to secure ones
- **sgn()**: returns the sign of the input.

$$return\ of\ sgn() = \begin{cases} -1, & \text{if } x \leq 0 \\ 0, & \text{if } x = 0 \\ 1, & \text{if } x \geq 0 \end{cases}$$

Besides the above functions, basic operations have an impressive role. Their cumulative time seems like they are the least important ones however Figure 4.3 shows that without adding sub functions, **__mul__**() function have the most execution time. Figure4.3 shows cProfiling results sorted according to the tottime of the calls.

Gray painted rows have the most cumulative time however their own execution times are less than basic operations. They don't have an impressive effect on the total time with own execution times. On the other hand, they have an impressive effect with their subfunctions. Figure 4.7 shows gprof2dot output of a shuffle-sort algorithm of length 100,000 in MPyC. When we follow the green and red blocks, in the bottom of the colored blocks, sgn() function call has the most weight compared with the same level calls. There is not any impressive call below sgn function in the call path. Most of the basic functions are the sub function of sgn(). It may not effect the total time by its own execution however with many basic calls, it have a huge effect. In the Figure4.2, almost %90 of the cumulative time belongs to sgn() and its subfunctions. Effort on the decreasing its execution time is meaningless however if the sub function call times is decreased, their total effect may be decreased and sgn() weight is reduced.



```
  ncalls    tottime  percall  cumtime  percall  filename:        (function)
  4140194    2.778    0.000    6.628    0.000   finfields.py:     (__mul__)
   46998     1.505    0.000    30.68    0.001   runtime.py:         (sgn)
 11574571    1.235    0.000    1.235    0.000   finfields.py:     (__init__)
  594495     0.395    0.000    0.986    0.000   finfields.py:     (__add__)
  572553     0.371    0.000    0.932    0.000   finfields.py:     (__sub__)
   18600     0.292    0.000    4.239    0.000   runtime.py:     (scalar_mul)
   16602     0.168    0.000    2.593    0.000   runtime.py:     (vector_sub)
  145730     0.167    0.000    1.009    0.000   runtime.py:         (sub)
   1998      0.144    0.000    2.453    0.001   runtime.py:     (vector_add)
   46998     0.114    0.000    0.567    0.000   runtime.py:      (_randoms)
     1       0.113    0.113    15.69    15.69   random.py:        (shuffle)
     1       0.058    0.058    33.58    33.58   runtime.py:        (_sort)
```

Figure 4.3: Profiling Results of Shuffle-Sort are sorted according to tottime

Profiling results show that shuffle call has a critical effect on the performance and we focused that part. Optimization in the shuffle call may decrease the execution time drastically. MPyC framework uses the modern version of Fisher–Yates[10] shuffle algorithm.The original version of the Fisher-Yates is an algorithm for generating a random permutation of a finite set. Each element is swapped with a random one from the same list so this process ensures a uniformly random arrangement of elements. The modern version swaps the elements with unswapped ones and this increases efficiency. In the result of these shuffles, the elements may stay in their original positions and also the sequence may not be shuffled with a $1 \div n!$ probability. In the Sattolo shuffle[42], elements are shuffled in a way that guarantees no element remains in its original position by creating a cycle of permutations.

Algorithm4 shows the Sattolo shuffle implementation in Python. Firstly, a random integer is obtained to get the value of a random index. In each iteration, this random value and the next value in the iteration are swapped.

Algorithm 4 Sattolo Shuffle in Python

```
def sattolo(x):
    $n \leftarrow len(x)$
    for i in range(n-1 , 0 , -1):
        $r \leftarrow random.randrange(0, i)$
        $x[i], x[r] \leftarrow x[r], x[i]$
return
```

For the insecure operations, swapping the elements in a list is straight forward since the values and indexes are known however for secure operations, the indexes are secret so swapping is a bit challenging. All elements in the list must be swapped randomly one by one from last element to first one. While swapping, none of the elements can stay in its original position since swap operation is done between the target element and rest of the list which are not swapped yet. Algorithm5 shows the Sattolo shuffle implementation in MPyC framework.

Algorithm 5 Sattolo Shuffle Implementation in MPyC

```
def shuffle(sectype, x):
    $n \leftarrow len(x)$
    for i in range(n-1 , 0 , -1):
        $u \leftarrow random\_unit\_vector(sectype\ ,\ i)$
        $x\_u \leftarrow runtime.in\_prod(x[\,:i]\ ,\ u)$
        $d \leftarrow runtime.scalar\_mul(x[i] - x\_u\ ,\ u)$
        $x[i] \leftarrow x\_u$
        $x[\,:i] \leftarrow runtime.vector\_add(x[\,:i]\ ,\ d)$
return
```

1. Firstly a random element has to be selected. To select an element randomly and securely, random_unit_vector() call is used. A unit vector is created with one of the cells is filled with 1 and rest of the elements are 0. The index which is determined randomly is secure so the position of the 1 is secret.

2. With the inner product operation between the actual list x and unit vector u, a random element is chosen from the list and assigned to x_u. This value will be

switched with x[i] in the following lines.

3. With scalar multiplication operation between (x[i]-x_u) and unit vector, a vector, d, is obtained that holds $x[i] - x\_u$ in its secret index. Then x_u value is assigned to i index of x vector. First swap is done.

4. After the vector addition with x and d vector, result of $x\_u + x[i] - x\_u = x[i]$ assigned to the secret index of x vector.

We have implemented Sattolo shuffle for both performance and usability since Sattolo shuffle is one the preferred algorithms in many studies[43]' [32],[31],[44],[33]. For the performance, the execution time is the same as modern version of Fisher-Yates shuffle. The figure 4.4 shows the execution times of original and Sattolo shuffle in MPyC.

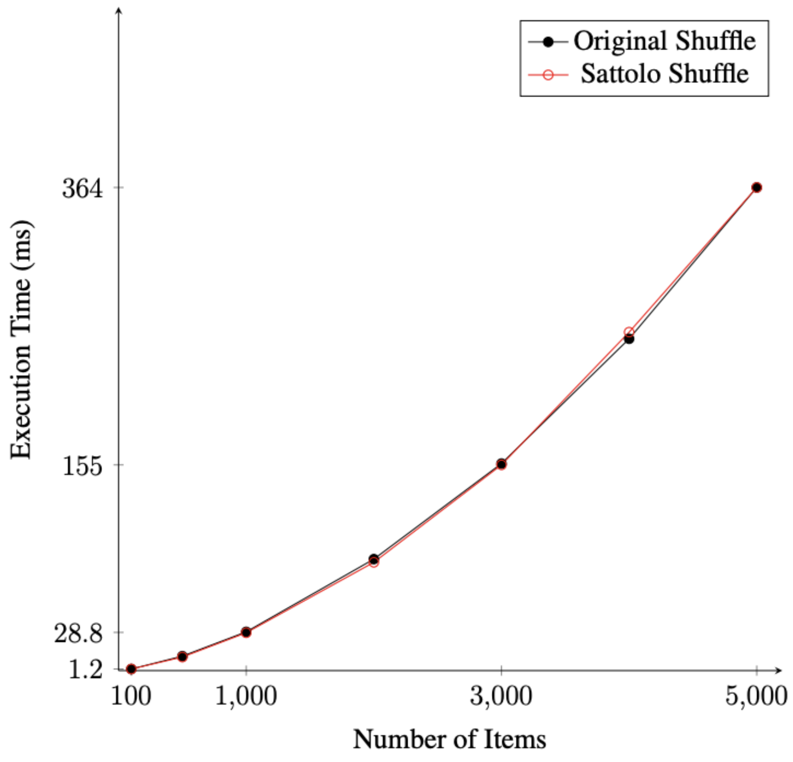
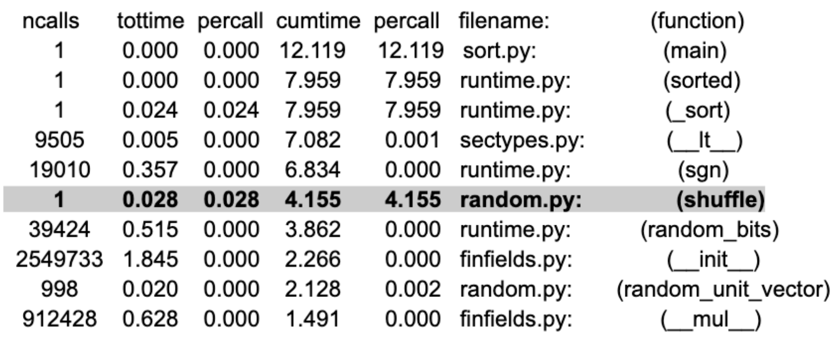


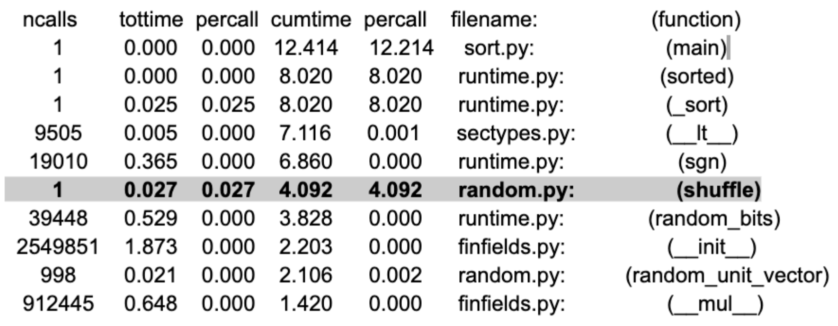
Figure 4.4: Original and Sattolo Shuffle-Sort Algorithm Execution Time in MPyC

Besides execution times, we obtained profiling results for deeper analysis. Main functions and their sub calls execution times and number of calls are presented in the Figure 4.5 for original Shuffle and in the Figure 4.6 for the Sattolo Shuffle. The call times and overall execution time are almost same. Performance enhancement isn't obtained however our implementation shows that without any performance concern, Sattolo Shuffle can be implemented easily and used for further studies.



ncalls	tottime	percall	cumtime	percall	filename:	(function)
1	0.000	0.000	12.119	12.119	sort.py:	(main)
1	0.000	0.000	7.959	7.959	runtime.py:	(sorted)
1	0.024	0.024	7.959	7.959	runtime.py:	(_sort)
9505	0.005	0.000	7.082	0.001	sectypes.py:	(__lt__)
19010	0.357	0.000	6.834	0.000	runtime.py:	(sgn)
1	**0.028**	**0.028**	**4.155**	**4.155**	**random.py:**	**(shuffle)**
39424	0.515	0.000	3.862	0.000	runtime.py:	(random_bits)
2549733	1.845	0.000	2.266	0.000	finfields.py:	(__init__)
998	0.020	0.000	2.128	0.002	random.py:	(random_unit_vector)
912428	0.628	0.000	1.491	0.000	finfields.py:	(__mul__)

Figure 4.5: Profiling Results of Original Shuffle-Sort are sorted according to cumtime



ncalls	tottime	percall	cumtime	percall	filename:	(function)
1	0.000	0.000	12.414	12.214	sort.py:	(main)
1	0.000	0.000	8.020	8.020	runtime.py:	(sorted)
1	0.025	0.025	8.020	8.020	runtime.py:	(_sort)
9505	0.005	0.000	7.116	0.001	sectypes.py:	(__lt__)
19010	0.365	0.000	6.860	0.000	runtime.py:	(sgn)
1	**0.027**	**0.027**	**4.092**	**4.092**	**random.py:**	**(shuffle)**
39448	0.529	0.000	3.828	0.000	runtime.py:	(random_bits)
2549851	1.873	0.000	2.203	0.000	finfields.py:	(__init__)
998	0.021	0.000	2.106	0.002	random.py:	(random_unit_vector)
912445	0.648	0.000	1.420	0.000	finfields.py:	(__mul__)

Figure 4.6: Profiling Results of Sattolo Shuffle-Sort are sorted according to cumtime

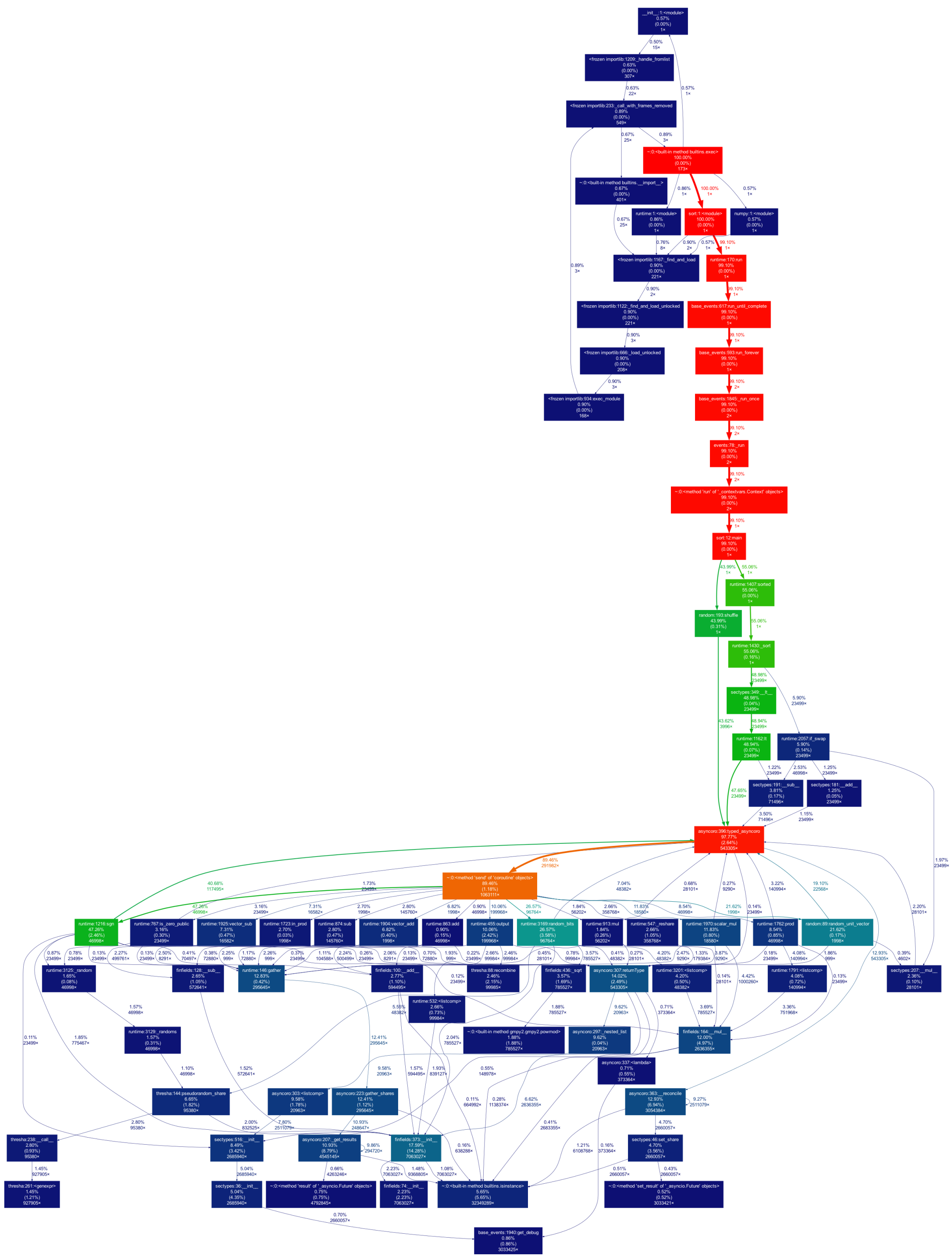


Figure 4.7: Dot Graph of Shuffle-Sort in MPyC

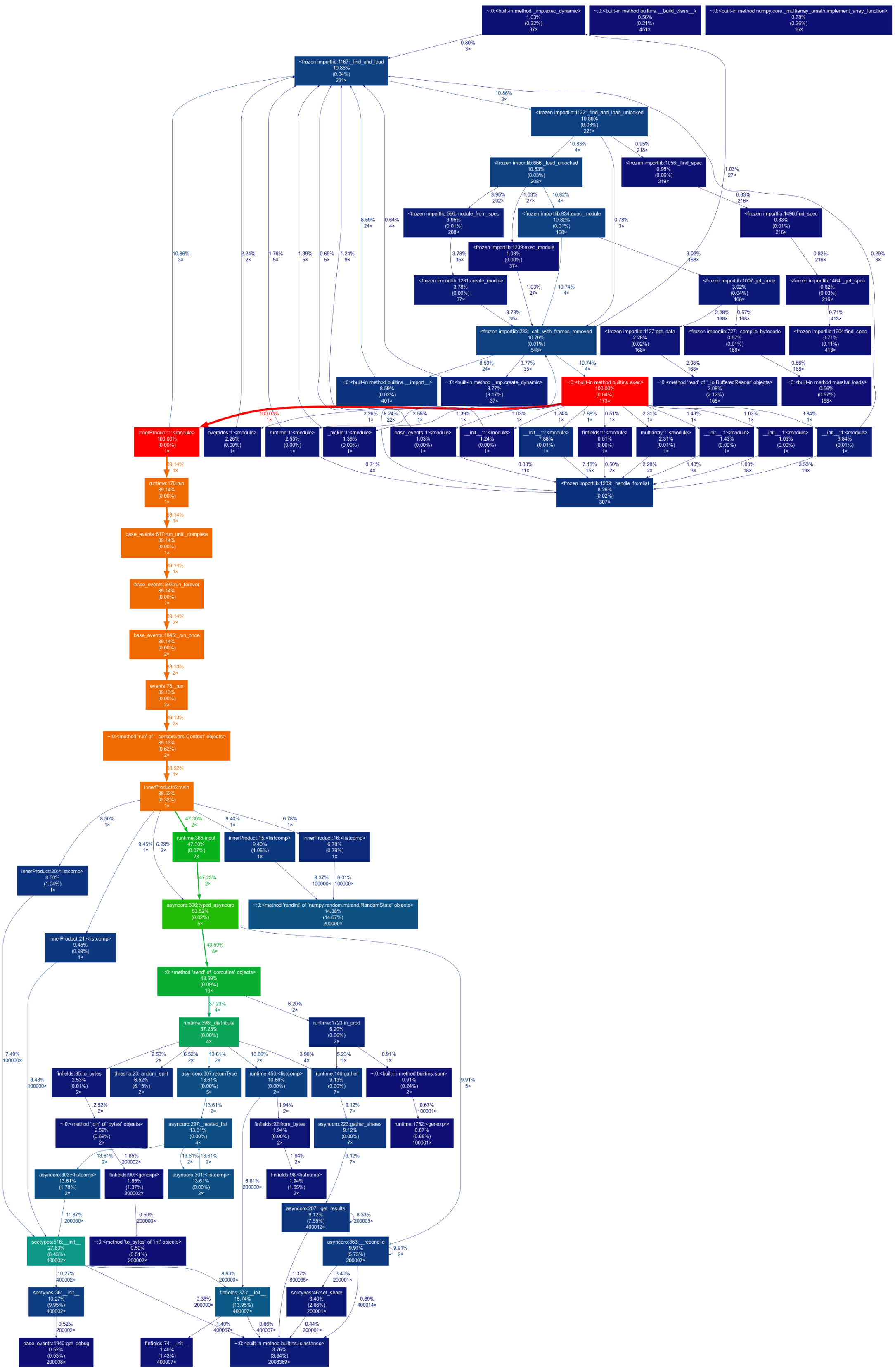

~:0:<built-in method _imp.exec_dynamic>
~:0:<built-in method builtins.__build_class__>
~:0:<built-in method numpy.core._multiarray_umath.implement_array_function>
<frozen importlib:1167:_find_and_load
<frozen importlib:1122:_find_and_load_unlocked
<frozen importlib:666:_load_unlocked
<frozen importlib:1056:_find_spec
<frozen importlib:566:module_from_spec
<frozen importlib:934:exec_module
<frozen importlib:1496:find_spec
<frozen importlib:1239:exec_module
<frozen importlib:1231:create_module
<frozen importlib:1007:get_code
<frozen importlib:1464:_get_spec
<frozen importlib:233:_call_with_frames_removed
<frozen importlib:1127:get_data
<frozen importlib:727:_compile_bytecode
<frozen importlib:1604:find_spec
~:0:<built-in method builtins.__import__>
~:0:<built-in method _imp.create_dynamic>
~:0:<built-in method builtins.exec>
~:0:<method 'read' of '_io.BufferedReader' objects>
~:0:<built-in method marshal.loads>
innerProduct:1:<module>
overrides:1:<module>
runtime:1:<module>
_pickle:1:<module>
base_events:1:<module>
__init__:1:<module>
finfields:1:<module>
multiarray:1:<module>
<frozen importlib:1209:_handle_fromlist
runtime:170:run
base_events:617:run_until_complete
base_events:593:run_forever
base_events:1845:_run_once
events:78:_run
~:0:<method 'run' of '_contextvars.Context' objects>
innerProduct:6:main
runtime:365:input
innerProduct:15:<listcomp>
innerProduct:16:<listcomp>
innerProduct:20:<listcomp>
asyncoro:396:typed_asyncoro
~:0:<method 'randint' of 'numpy.random.mtrand.RandomState' objects>
innerProduct:21:<listcomp>
~:0:<method 'send' of 'coroutine' objects>
runtime:398:_distribute
runtime:1723:in_prod
finfields:85:to_bytes
thresha:23:random_split
asyncoro:307:returnType
runtime:450:<listcomp>
runtime:146:gather
~:0:<built-in method builtins.sum>
~:0:<method 'join' of 'bytes' objects>
asyncoro:297:_nested_list
finfields:92:from_bytes
asyncoro:223:gather_shares
runtime:1752:<genexpr>
asyncoro:303:<listcomp>
finfields:90:<genexpr>
asyncoro:301:<listcomp>
finfields:98:<listcomp>
asyncoro:207:_get_results
sectypes:516:__init__
~:0:<method 'to_bytes' of 'int' objects>
asyncoro:363:__reconcile
sectypes:36:__init__
finfields:373:__init__
sectypes:46:set_share
base_events:1940:get_debug
finfields:74:__init__
~:0:<built-in method builtins.isinstance>

Figure 4.8: Dot Graph of Inner Product in MPyC

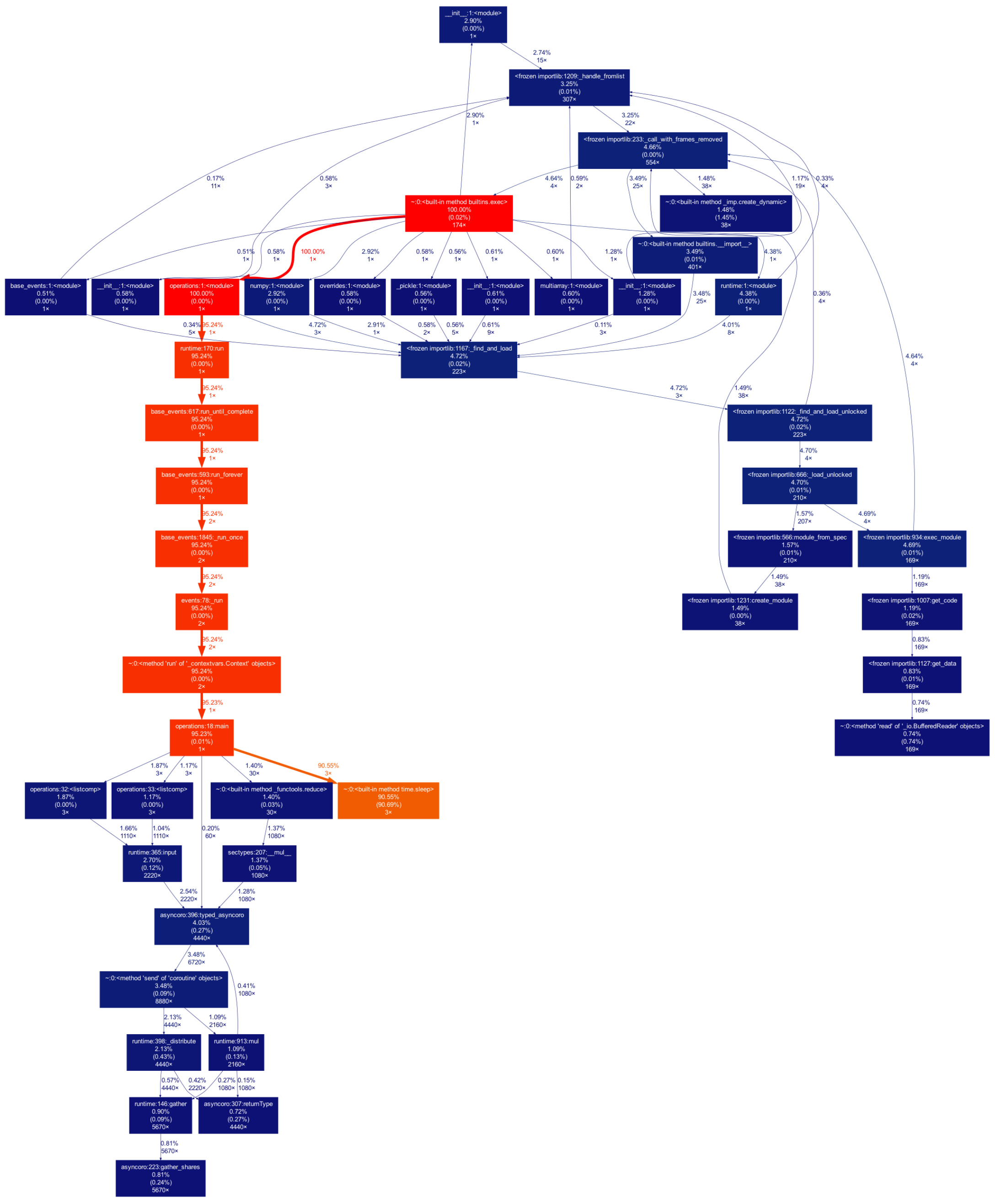

__init__:1:<module> 2.90% (0.00%) 1×
<frozen importlib:1209:_handle_fromlist 3.25% (0.01%) 307×
<frozen importlib:233:_call_with_frames_removed 4.66% (0.00%) 554×
~:0:<built-in method _imp.create_dynamic> 1.48% (1.45%) 38×
~:0:<built-in method builtins.exec> 100.00% (0.02%) 174×
~:0:<built-in method builtins.__import__> 3.49% (0.01%) 401×
base_events:1:<module> 0.51% (0.00%) 1×
__init__:1:<module> 0.58% (0.00%) 1×
operations:1:<module> 100.00% (0.00%) 1×
numpy:1:<module> 2.92% (0.00%) 1×
overrides:1:<module> 0.58% (0.00%) 1×
_pickle:1:<module> 0.56% (0.00%) 1×
__init__:1:<module> 0.61% (0.00%) 1×
multiarray:1:<module> 0.60% (0.00%) 1×
__init__:1:<module> 1.28% (0.00%) 1×
runtime:1:<module> 4.38% (0.00%) 1×
<frozen importlib:1167:_find_and_load 4.72% (0.02%) 223×
<frozen importlib:1122:_find_and_load_unlocked 4.72% (0.02%) 223×
<frozen importlib:666:_load_unlocked 4.70% (0.01%) 210×
<frozen importlib:566:module_from_spec 1.57% (0.01%) 210×
<frozen importlib:934:exec_module 4.69% (0.01%) 169×
<frozen importlib:1231:create_module 1.49% (0.00%) 38×
<frozen importlib:1007:get_code 1.19% (0.02%) 169×
<frozen importlib:1127:get_data 0.83% (0.01%) 169×
~:0:<method 'read' of '_io.BufferedReader' objects> 0.74% (0.74%) 169×
runtime:170:run 95.24% (0.00%) 1×
base_events:617:run_until_complete 95.24% (0.00%) 1×
base_events:593:run_forever 95.24% (0.00%) 1×
base_events:1845:_run_once 95.24% (0.00%) 2×
events:78:_run 95.24% (0.00%) 2×
~:0:<method 'run' of '_contextvars.Context' objects> 95.24% (0.00%) 2×
operations:18:main 95.23% (0.01%) 1×
operations:32:<listcomp> 1.87% (0.00%) 3×
operations:33:<listcomp> 1.17% (0.00%) 3×
~:0:<built-in method _functools.reduce> 1.40% (0.03%) 30×
~:0:<built-in method time.sleep> 90.55% (90.69%) 3×
runtime:365:input 2.70% (0.12%) 2220×
sectypes:207:__mul__ 1.37% (0.05%) 1080×
asyncoro:396:typed_asyncoro 4.03% (0.27%) 4440×
~:0:<method 'send' of 'coroutine' objects> 3.48% (0.09%) 8880×
runtime:398:_distribute 2.13% (0.43%) 4440×
runtime:913:mul 1.09% (0.13%) 2160×
runtime:146:gather 0.90% (0.09%) 5670×
asyncoro:307:returnType 0.72% (0.27%) 4440×
asyncoro:223:gather_shares 0.81% (0.24%) 5670×

Figure 4.9: Dot Graph of Reduce Multiplication in MPyC

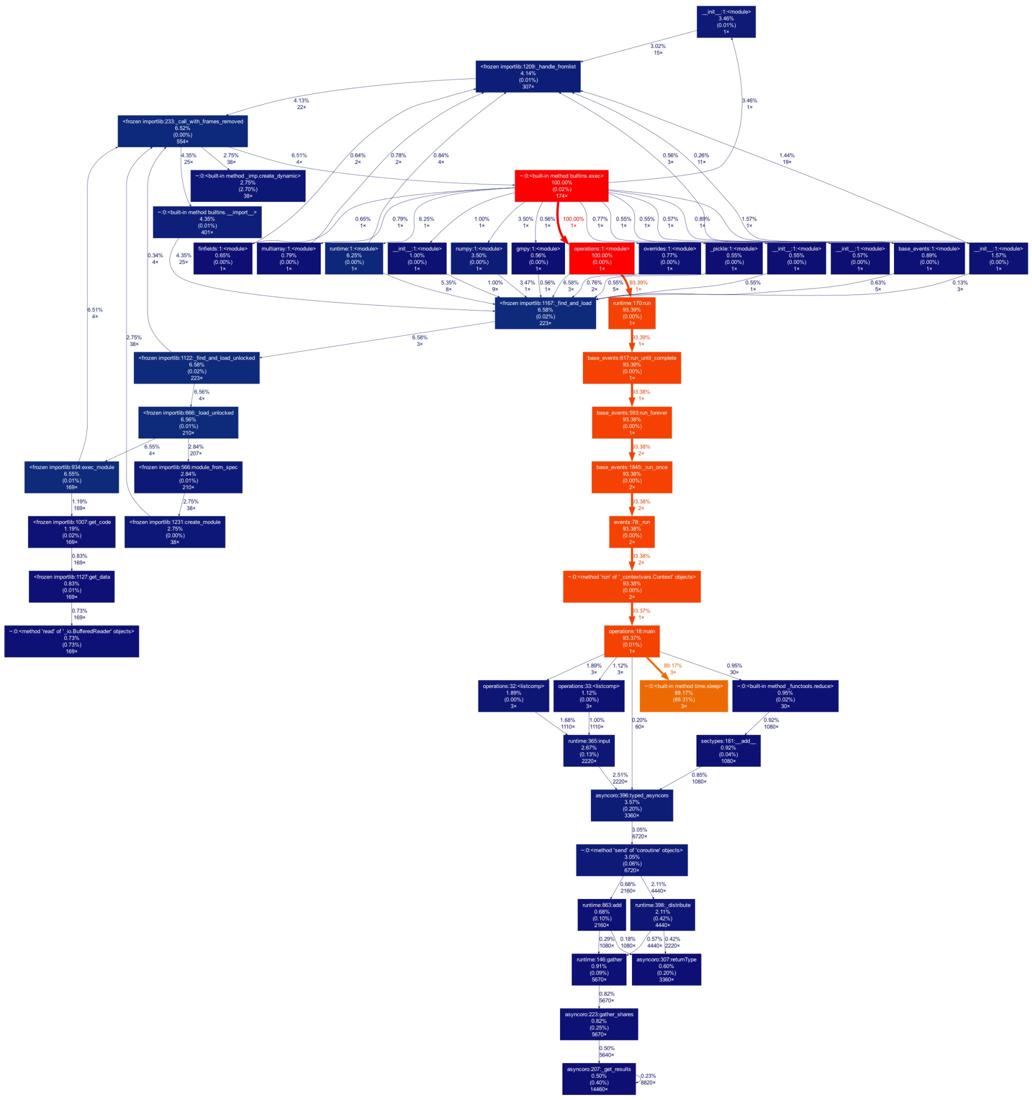
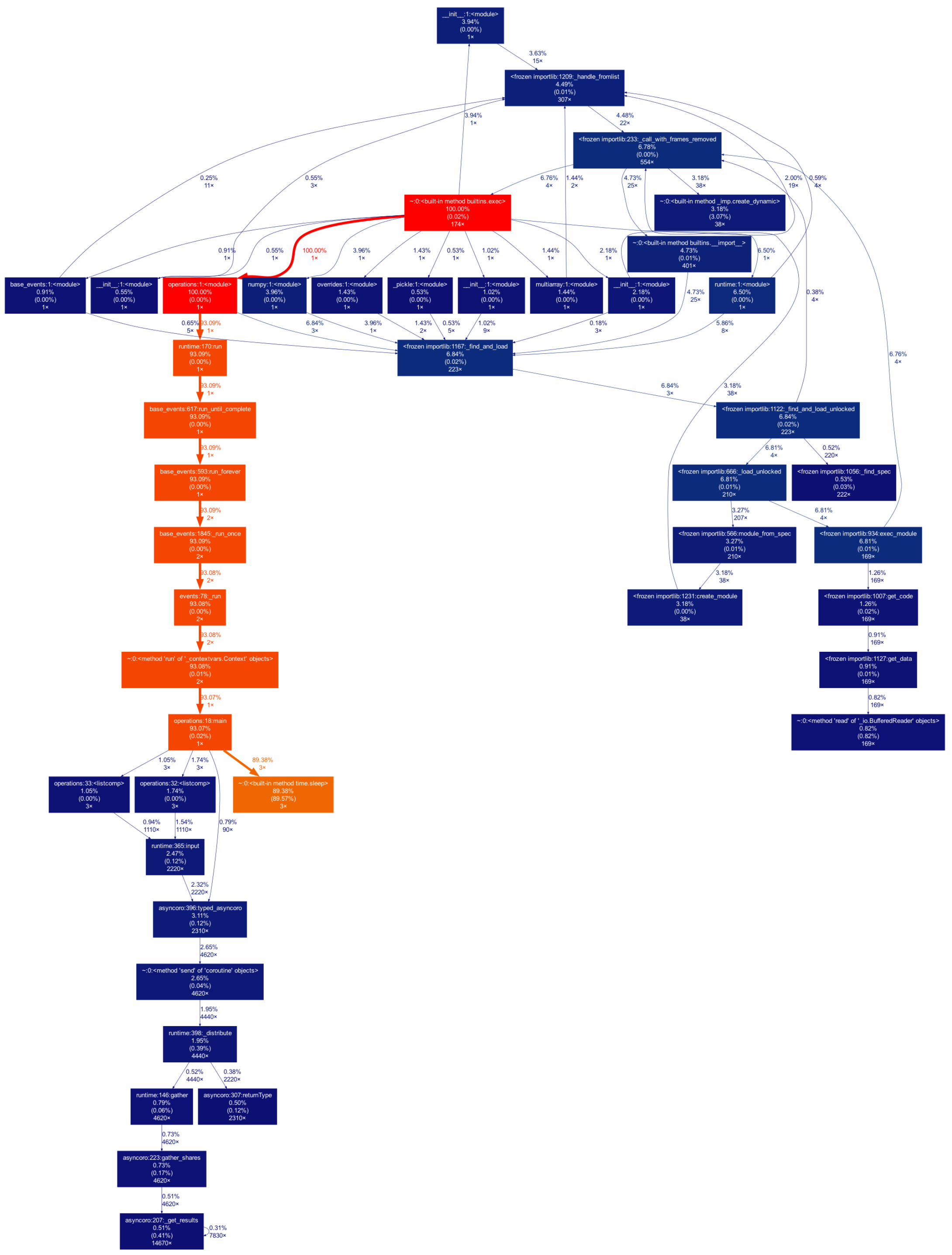


Figure 4.10: Dot Graph of Reduce Addition in MPyC


__init__:1:<module> 3.94% (0.00%) 1×
<frozen importlib:1209:_handle_fromlist 4.49% (0.01%) 307×
<frozen importlib:233:_call_with_frames_removed 6.78% (0.00%) 554×
~:0:<built-in method _imp.create_dynamic> 3.18% (3.07%) 38×
~:0:<built-in method builtins.exec> 100.00% (0.02%) 174×
~:0:<built-in method builtins.__import__> 4.73% (0.01%) 401×
base_events:1:<module> 0.91% (0.00%) 1×
__init__:1:<module> 0.55% (0.00%) 1×
operations:1:<module> 100.00% (0.00%) 1×
numpy:1:<module> 3.96% (0.00%) 1×
overrides:1:<module> 1.43% (0.00%) 1×
_pickle:1:<module> 0.53% (0.00%) 1×
__init__:1:<module> 1.02% (0.00%) 1×
multiarray:1:<module> 1.44% (0.00%) 1×
__init__:1:<module> 2.18% (0.00%) 1×
runtime:1:<module> 6.50% (0.00%) 1×
runtime:170:run 93.09% (0.00%) 1×
<frozen importlib:1167:_find_and_load 6.84% (0.02%) 223×
base_events:617:run_until_complete 93.09% (0.00%) 1×
<frozen importlib:1122:_find_and_load_unlocked 6.84% (0.02%) 223×
base_events:593:run_forever 93.09% (0.00%) 1×
<frozen importlib:666:_load_unlocked 6.81% (0.01%) 210×
<frozen importlib:1056:_find_spec 0.53% (0.03%) 222×
base_events:1845:_run_once 93.09% (0.00%) 2×
<frozen importlib:566:module_from_spec 3.27% (0.01%) 210×
<frozen importlib:934:exec_module 6.81% (0.01%) 169×
events:78:_run 93.08% (0.00%) 2×
<frozen importlib:1231:create_module 3.18% (0.00%) 38×
<frozen importlib:1007:get_code 1.26% (0.02%) 169×
~:0:<method 'run' of '_contextvars.Context' objects> 93.08% (0.01%) 2×
<frozen importlib:1127:get_data 0.91% (0.01%) 169×
~:0:<method 'read' of '_io.BufferedReader' objects> 0.82% (0.82%) 169×
operations:18:main 93.07% (0.02%) 1×
operations:33:<listcomp> 1.05% (0.00%) 3×
operations:32:<listcomp> 1.74% (0.00%) 3×
~:0:<built-in method time.sleep> 89.38% (89.57%) 3×
runtime:365:input 2.47% (0.12%) 2220×
asyncoro:396:typed_asyncoro 3.11% (0.12%) 2310×
~:0:<method 'send' of 'coroutine' objects> 2.65% (0.04%) 4620×
runtime:398:_distribute 1.95% (0.39%) 4440×
runtime:146:gather 0.79% (0.06%) 4620×
asyncoro:307:returnType 0.50% (0.12%) 2310×
asyncoro:223:gather_shares 0.73% (0.17%) 4620×
asyncoro:207:_get_results 0.51% (0.41%) 14670×
Figure 4.11: Dot Graph of Vector Addition in MPyC

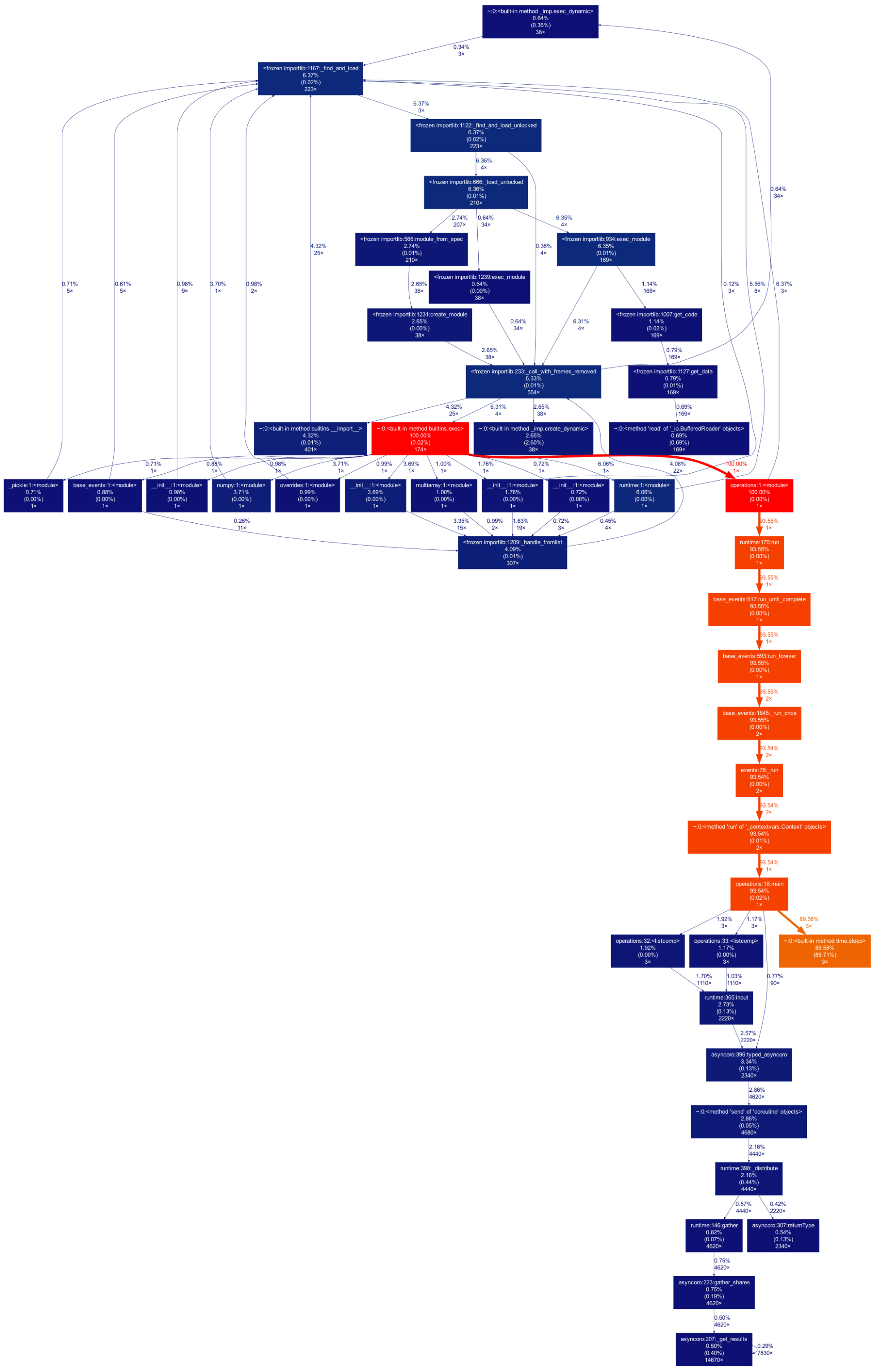

~:0:<built-in method _imp.exec_dynamic>
<frozen importlib:1167:_find_and_load>
<frozen importlib:1122:_find_and_load_unlocked>
<frozen importlib:666:_load_unlocked>
<frozen importlib:566:module_from_spec>
<frozen importlib:934:exec_module>
<frozen importlib:1239:exec_module>
<frozen importlib:1231:create_module>
<frozen importlib:1007:get_code>
<frozen importlib:233:_call_with_frames_removed>
<frozen importlib:1127:get_data>
~:0:<built-in method builtins.__import__>
~:0:<built-in method builtins.exec>
~:0:<built-in method _imp.create_dynamic>
~:0:<method 'read' of '_io.BufferedReader' objects>
_pickle:1:<module>
base_events:1:<module>
__init__:1:<module>
numpy:1:<module>
overrides:1:<module>
multiarray:1:<module>
runtime:1:<module>
operations:1:<module>
<frozen importlib:1209:_handle_fromlist>
runtime:170:run
base_events:617:run_until_complete
base_events:593:run_forever
base_events:1845:_run_once
events:78:_run
~:0:<method 'run' of '_contextvars.Context' objects>
operations:18:main
operations:32:<listcomp>
operations:33:<listcomp>
~:0:<built-in method time.sleep>
runtime:365:input
asyncoro:396:typed_asyncoro
~:0:<method 'send' of 'coroutine' objects>
runtime:398:_distribute
runtime:146:gather
asyncoro:307:returnType
asyncoro:223:gather_shares
asyncoro:207:_get_results

Figure 4.12: Dot Graph of Schur_Prod() in MPyC

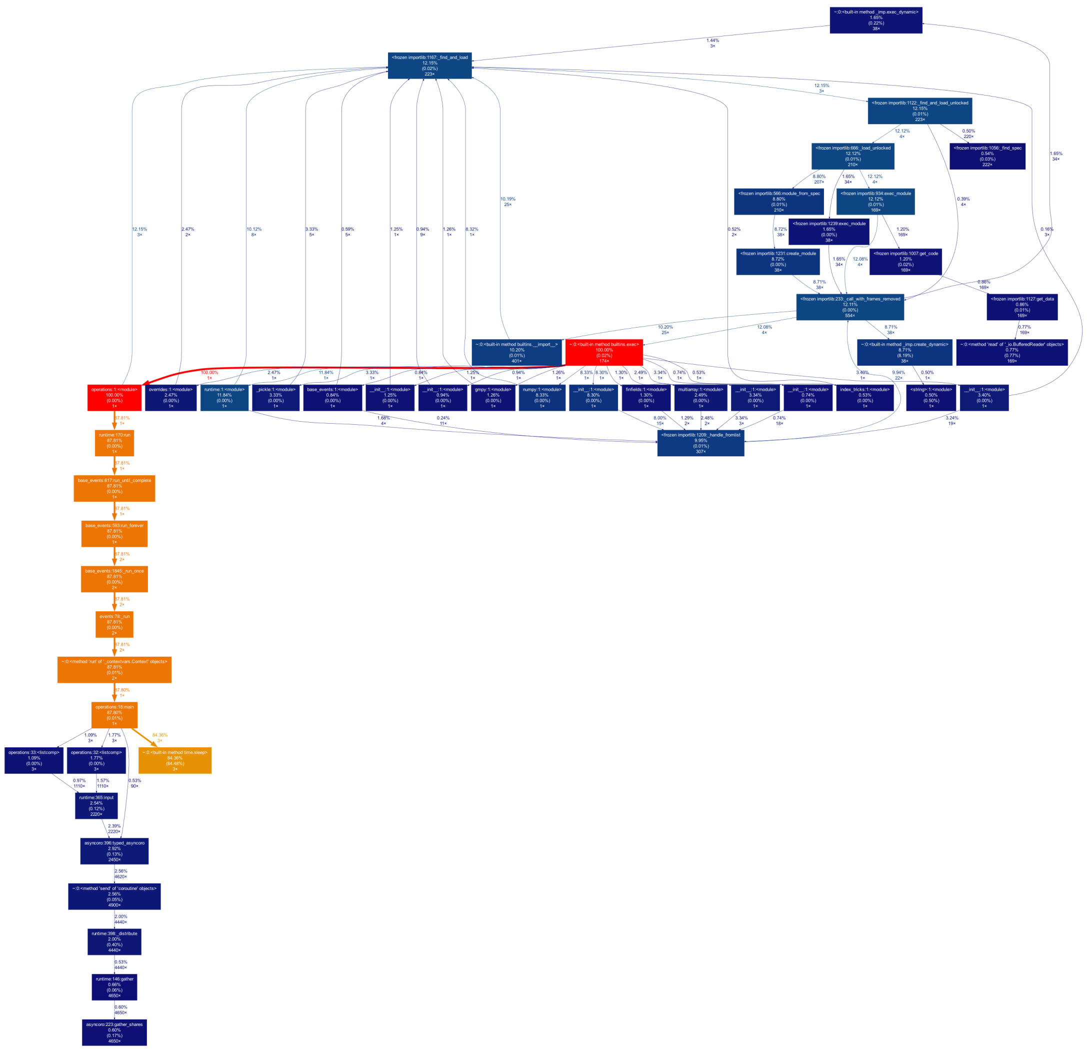


Figure 4.13: Dot Graph of mpc.Prod() in MPyC

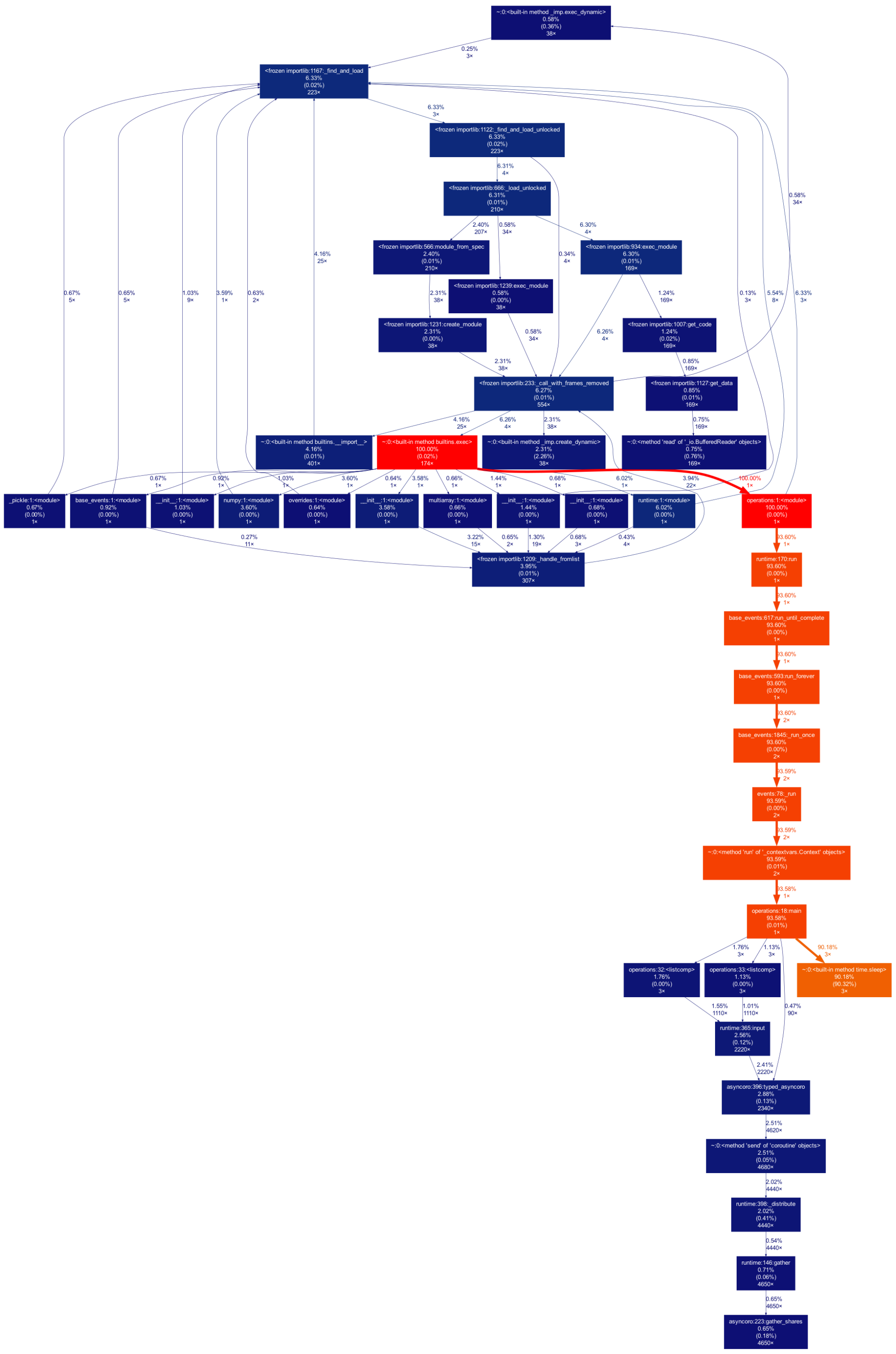

~:0:<built-in method _imp.exec_dynamic>
<frozen importlib:1167:_find_and_load
<frozen importlib:1122:_find_and_load_unlocked
<frozen importlib:666:_load_unlocked
<frozen importlib:566:module_from_spec
<frozen importlib:1239:exec_module
<frozen importlib:934:exec_module
<frozen importlib:1231:create_module
<frozen importlib:1007:get_code
<frozen importlib:233:_call_with_frames_removed
<frozen importlib:1127:get_data
~:0:<built-in method builtins.__import__>
~:0:<built-in method builtins.exec>
~:0:<built-in method _imp.create_dynamic>
~:0:<method 'read' of '_io.BufferedReader' objects>
_pickle:1:<module>
base_events:1:<module>
__init__:1:<module>
numpy:1:<module>
overrides:1:<module>
multiarray:1:<module>
runtime:1:<module>
operations:1:<module>
<frozen importlib:1209:_handle_fromlist
runtime:170:run
base_events:617:run_until_complete
base_events:593:run_forever
base_events:1845:_run_once
events:78:_run
~:0:<method 'run' of '_contextvars.Context' objects>
operations:18:main
operations:32:<listcomp>
operations:33:<listcomp>
~:0:<built-in method time.sleep>
runtime:365:input
asyncoro:396:typed_asyncoro
~:0:<method 'send' of 'coroutine' objects>
runtime:398:_distribute
runtime:146:gather
asyncoro:223:gather_shares

Figure 4.14: Dot Graph of mpc.Sum() in MPyC

CHAPTER 5

CONCLUSION

Thanks to advances in technology, algorithms are now more reliable and efficient. Nowadays people have the chance to use these algorithms to store and process their private information. Key objectives of the research is to analyze these algorithms especially the frameworks, and reveal the bottlenecks for optimization.

In this thesis, we focus on two Multi Party Computation frameworks MP-SPDZ and MPyC. Firstly, we give the background information about MPC. Its development process and features are presented. Then, detailed descriptions of the frameworks are given. Their features and related studies are introduced for better understanding of our goal. Later, we present our work. Performance comparison results of the frameworks are given. Benchmarks results are shown and the outstanding performance of MP-SPDZ is clearly seen in all implemented algorithms. A nonlinear pattern is obtained in the MP-SPDZ framework analysis because of its shuffle algorithm. In the shuffle-sort algorithm, in every power of 2, execution time is doubled and remained almost same until the next power of 2. This nonlinear behaviour is described as a weakness and defined as an open problem for future work.

Besides benchmarks, profiling analysis are done for a deeper insight. Bottlenecks of the algorithms are searched. We presented the profiling results and for better understanding, we visualised the execution paths by using gproof and gproof2dot tools. In the inner product algorithm analysis, we didn't face any bottlenecks in the key functions. Improvement of the init functions may have major effect on the optimization. In the shuffle-sort algorithm analysis in MPyC, shuffle part has the same importance with the sort part and we focused on shuffle. Sattolo shuffle is implemented instead of modern version of Fisher-Yates shuffle. Performance enhancement isn't achieved however its showed that Sattolo shuffle functionality can be used easily if needed with our implementation.

REFERENCES

[1] A. Aly, K. Cong, D. Cozzo, M. Keller, E. Orsini, D. Rotaru, O. Scherer, P. Scholl, N. P. Smart, T. Tanguy, et al., Scale–mamba v1. 14: Documentation, Documentation. pdf, 2021.

[2] G. Asharov, K. Hamada, D. Ikarashi, R. Kikuchi, A. Nof, B. Pinkas, K. Takahashi, and J. Tomida, Efficient secure three-party sorting with applications to data analysis and heavy hitters, Cryptology ePrint Archive, Paper 2022/1595, 2022.

[3] B. Beauquier and E. Darrot, On Arbitrary Waksman Networks and their Vulnerability, Technical Report RR-3788, INRIA, October 1999.

[4] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, Efficient garbling from a fixed-key blockcipher, Cryptology ePrint Archive, Paper 2013/426, 2013.

[5] D. Bogdanov, S. Laur, and R. Talviste, A practical analysis of oblivious sorting algorithms for secure multi-party computation, in K. Bernsmed and S. Fischer-Hübner, editors, *Secure IT Systems*, pp. 59–74, Springer International Publishing, Cham, 2014.

[6] D. Bogdanov, S. Laur, and J. Willemson, Sharemind: A framework for fast privacy-preserving computations, in *Computer Security-ESORICS 2008: 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings 13*, pp. 192–206, Springer, 2008.

[7] K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof, Fast large-scale honest-majority mpc for malicious adversaries, Cryptology ePrint Archive, Paper 2018/570, 2018.

[8] T.-H. Chien, J.-W. Lin, and R.-G. Chang, Parallel collision detection with openmp, Journal of Physics: Conference Series, 1069(1), p. 012180, aug 2018.

[9] S. G. Choi, D. Dachman-Soled, M. Kulkarni, and A. Yerukhimovich, Differentially-private multi-party sketching for large-scale statistics, Cryptology ePrint Archive, Paper 2020/029, 2020.

[10] W. contributors, Fisher–Yates shuffle, `https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle`, 2023, [Online; accessed 2-January-2024].

[11] R. Cramer, I. B. Damgård, and J. B. Nielsen, *Secure Multiparty Computation and Secret Sharing*, Cambridge University Press, 2015.

[12] D. Demmler, T. Schneider, and M. Zohner, Aby - a framework for efficient mixed-protocol secure two-party computation, 01 2015.

[13] D. Flater, Configuration of profiling tools for c/c++ applications under 64-bit linux, `https://doi.org/10.6028/NIST.TN.1790`, 03 2013, [Online; accessed 2-January-2024].

[14] M. Franz, A. Holzer, S. Katzenbeisser, C. Schallhart, and H. Veith, Cbmc-gc: an ansi c compiler for secure two-party computations, in *International Conference on Compiler Construction*, pp. 244–249, Springer, 2014.

[15] T. Gehlhar, F. Marx, T. Schneider, A. Suresh, T. Wehrle, and H. Yalame, Safefl: Mpc-friendly framework for private and robust federated learning, Cryptology ePrint Archive, Paper 2023/555, 2023.

[16] O. Goldreich, Secure multi-party computation, Manuscript. Preliminary Version, 03 1999.

[17] S. Graham and P. Kessler, Gprof: A call graph execution profiler, ACM SIGPLAN Notices, 17, 06 1982.

[18] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic, Sok: General purpose compilers for secure multi-party computation, in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 1220–1237, 2019.

[19] A. Institute, FRESCO - A FRamework for Efficient Secure Computation, `https://github.com/aicis/fresco`, 2023, [Online; accessed 2-January-2024].

[20] C. J. Gulo, A. Sementille, and J. Tavares, Optimizing a medical image registration algorithm based on profiling data for real-time performance, Multimedia Tools and Applications, 01 2022.

[21] K. V. Jönsson, G. Kreitz, and M. Uddin, Secure multi-party sorting and applications, Cryptology ePrint Archive, Paper 2011/122, 2011.

[22] M. Keller, Mp-spdz: A versatile framework for multi-party computation, Cryptology ePrint Archive, Paper 2020/521, 2020.

[23] A. Lapets, F. Jansen, K. D. Albab, R. Issa, L. Qin, M. Varia, and A. Bestavros, Accessible privacy-preserving web-based data analysis for assessing and addressing economic inequalities, in *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies*, COMPASS '18, Association for Computing Machinery, New York, NY, USA, 2018.

[24] Y. Lindell, Secure multiparty computation (mpc), Cryptology ePrint Archive, Paper 2020/300, 2020.

[25] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, Oblivm: A programming framework for secure computation, in *2015 IEEE Symposium on Security and Privacy*, pp. 359–376, 2015.

[26] T. Lorünser and F. Wohner, Performance comparison of two generic mpc-frameworks with symmetric ciphers, pp. 587–594, 07 2020.

[27] D. Lu and A. Kate, Rpm: Robust anonymity at scale, Cryptology ePrint Archive, Paper 2022/1037, 2022.

[28] P. M. Mammen, Federated learning: Opportunities and challenges, 2021.

[29] B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor, Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation, in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 112–127, IEEE, 2016.

[30] A. Rastogi, M. A. Hammer, and M. Hicks, Wysteria: A programming language for generic, mixed-mode multiparty computations, in *2014 IEEE Symposium on Security and Privacy*, pp. 655–670, 2014.

[31] S. Q. Ren, B. H. M. Tan, S. Sundaram, T. Wang, Y. Ng, V. Chang, and K. M. M. Aung, Secure searching on cloud storage enhanced by homomorphic indexing, Future Generation Computer Systems, 65, pp. 102–110, 2016.

[32] S. Santo and N. M. S. Iswari, Design and development of animal recognition application using gamification and sattolo shuffle algorithm on android platform, International Journal of New Media Technology, 4, pp. 46–53, 06 2017.

[33] A. Sarad and S. Srikanth, Improved interference diversity in multicellular ofdma systems, pp. 1 – 8, 02 2009.

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

[35] A. Shamir, How to share a secret, Commun. ACM, 22(11), p. 612–613, nov 1979.

[36] H. Smajlović, A. Shajii, B. Berger, H. Cho, and I. Numanagić, Sequre: a high-performance framework for rapid development of secure bioinformatics pipelines, 2022, pp. 164–165, 05 2022.

[37] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar, Tinygarble: Highly compressed and scalable sequential garbled circuits, in *2015 IEEE Symposium on Security and Privacy*, pp. 411–428, 2015.

[38] T. Veugen and M. Abspoel, Secure integer division with a private divisor., Proc. Priv. Enhancing Technol., 2021(4), pp. 339–349, 2021.

[39] T. Veugen, B. Kamphorst, and M. Marcus, Privacy-preserving contrastive explanations with local foil trees, Cryptography, 6(4), 2022.

[40] A. Waksman, A permutation network, J. ACM, 15(1), p. 159–163, jan 1968.

[41] X. Wang, A. J. Malozemoff, and J. Katz, EMP-toolkit: Efficient MultiParty computation toolkit, `https://github.com/emp-toolkit`, 2016, [Online; accessed 2-January-2024].

[42] M. C. Wilson and É. Fusy, Overview of sattolo's algorithm, 2005.

[43] L. Windheuser, C. Anneser, H. Zhang, T. Neumann, and A. Kemper, Adaptive compression for databases, 2024.

[44] M. Wongso and W. Istiono, Learn muay thai basic movement in virtual reality and sattolo shuffle algorithm, International Journal of Science, Technology amp; Management, 4(2), pp. 341–349, Mar. 2023.

[45] A. C. Yao, Protocols for secure computations, in *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pp. 160–164, 1982.

[46] S. Zahur and D. Evans, Obliv-c: A language for extensible data-oblivious computation, Cryptology ePrint Archive, Paper 2015/1153, 2015.

[47] Y. Zhang, A. Steele, and M. Blanton, Picco: A general-purpose compiler for private distributed computation, in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, p. 813–826, Association for Computing Machinery, New York, NY, USA, 2013.