Fabian Gruber, BSc

riscMPC: General-Purpose Multi-Party Computation from RISC-V Assembly

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisors

Christian Rechberger, Univ.-Prof. Dipl.-Ing. Dr.techn.
Fabian Schmid, Dipl.-Ing.
Shibam Mukherjee, Dipl.-Ing.

Institute of Applied Information Processing and Communications

Graz, September 2024

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date, Signature

Acknowledgements

I want to take this opportunity to thank my supervisors, especially Fabian Schmid, who answered countless questions and gave me valuable feedback during the last year.

Furthermore, I want to thank Franco Nieddu, who inspired me to take on this topic and helped me when I was stuck with a particular problem or design choice during development.

Finally, I want to thank everyone who supported me during my studies and while working on this thesis.

Abstract

With rising privacy concerns, the use of Privacy Enhancing Technologies (PETs) grows in importance. One such technology is Multi-Party Computation (MPC), which allows multiple parties to perform computations on sensitive data without revealing it. Often, the lack of PETs is not due to a lack of interest but rather due to their complexity and performance cost. This proves to be a bottleneck in their widespread adoption.

This thesis describes riscMPC, a RISC-V Virtual Machine (VM) for general-purpose MPC. The riscMPC VM forms the core of a Rust library crate that handles inputs and outputs. riscMPC takes arbitrary RISC-V assembly functions as input and executes the program in a network with two parties. This allows the use of any programming language that compiles to RISC-V. Parties can input public or secret values as function arguments via registers or as data in memory. Inputs that are passed via registers adhere to the RISC-V calling convention. Larger inputs, such as arrays, are initialized in memory and passed as pointers. Public inputs are used in plain, and secret inputs are secret-shared. During program execution, protocols are chosen based on whether instruction arguments are secret or public.

As a result, riscMPC can be used as a drop-in replacement for almost any public function, and the computation can be run via MPC. The program execution depends entirely on the input data. This makes riscMPC ideal for rapid development and easy to use, even for non-experts.

Keywords. Mutlti-Party Computation · RISC-V · Virtual Machine · Rust

Kurzfassung

Mit zunehmenden Datenschutzbedenken gewinnt der Einsatz von Privacy Enhancing Technologies (PETs) an Bedeutung. Eine dieser Technologien ist Multi-Party Computation (MPC), welche es mehreren Parteien ermöglicht, Berechnungen mit sensiblen Daten durchzuführen, ohne diese preiszugeben. Oft liegt der Mangel an PETs nicht an fehlendem Interesse, sondern an ihrer Komplexität und Performancekosten. Dies erweist sich als Hindernis für ihre weit verbreitete Einführung.

Diese Arbeit beschreibt riscMPC, eine RISC-V Virtual Machine (VM) für allgemeine MPC Anwendungen. Die riscMPC VM bildet den Hauptteil einer Rust Bibliothek, welche für Eingaben und Ausgaben verantwortlich ist. riscMPC nimmt beliebige RISC-V Assembly-Funktionen als Eingabe und führt das Programm in einem Netzwerk mit zwei Parteien aus. Dies ermöglicht die Nutzung jeder Programmiersprache, die zu RISC-V kompiliert werden kann. Parteien können öffentliche oder geheime Werte als Funktionsargumente über Register oder als Daten im Speicher eingeben. Eingaben, die über Register übergeben werden, entsprechen der RISC-V-Aufrufkonvention. Größere Eingaben, wie Arrays, werden im Speicher initialisiert und als Pointer übergeben. Öffentliche Eingaben werden im Klartext verwendet, und geheime Eingaben werden geheim geteilt. Während der Ausführung eines Programmes werden Protokolle basierend darauf gewählt, ob die Instruktionsargumente geheim oder öffentlich sind.

Folglich kann riscMPC als Ersatz für fast jede öffentliche Funktion verwendet werden und die Berechnung mit MPC durchführen. Die Programmausführung hängt vollständig von den Eingabedaten ab. Dies macht riscMPC ideal für schnelle Entwicklung und einfach zu bedienen, selbst für Benutzer und Benutzerinnen ohne Fachkenntnisse.

Schlagwörter. Mutlti-Party Computation · RISC-V · Virtual Machine · Rust

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

MPC	Multi-Party Computation
PETs	Privacy Enhancing Technologies
OT	Oblivious Transfer
RISC	Reduced Instruction Set Computer
CISC	Complex Instruction Set Computer
ISA	Instruction Set Architecture
PSI	Private Set Intersection
ML	Machine Learning
VM	Virtual Machine
SIMD	Single Instruction Multiple Data
DH	Diffie-Hellman
AES	Advanced Encryption Standard
NN	Neural Network
AEAD	Authenticated Encryption with Associated Data
PPML	Privacy-Preserving Machine Learning
GDPR	General Data Protection Regulation
ALSZ	Asharov-Lindell-Schneider-Zohner
ECC	Elliptic Curve Cryptography
MNIST	Modified National Institute of Standards and Technology
SPN	Substitution Permutation Network

Chapter 1

Introduction

Today, and even more so in the future, more and more private data is stored somewhere on some server of some company or government. Often, these datasets only consist of a small subset of all the personal information required to run the provided service. However, when entities need to collaborate and share user data, concerning amounts of personal information can be aggregated in one place. In addition to users, some service providers also want to keep their data (such as machine learning models) secret. So, neither end wants to or is allowed to (e.g., due to the EU General Data Protection Regulation (GDPR) [EC23]) share their data at all.

Privacy Enhancing Technologies (PETs) such as Multi-Party Computation (MPC) are used to allow computation on accumulated personal information without sacrificing privacy. MPC uses secret-sharing techniques to split secret values into multiple shares, such that no information can be gained without knowledge of sufficient shares [Sha79]. Parties follow different protocols to perform operations on their shares. The final result can only be learned by combining the required amount of shares. Multiple parties can perform operations such as addition and multiplication on secret-shared values. Different protocols and security guarantees exist for different numbers of parties [Eva+18].

MPC is used to preserve privacy in statistics, machine learning, and many more use cases. With MPC, the average salary of employees from different companies can be computed without leaking the salaries of individual people. In Privacy-Preserving Machine Learning (PPML), neither the model provider nor the user has to share secret data with the other. Another common use case for MPC is calculating the Private Set Intersection (PSI) of two data sets. Normally, to get the intersection of two sets, one needs to know both sets. However, MPC allows us to perform comparisons in secret. Therefore, no party learns anything about the other set other than the intersection.

Most MPC frameworks or libraries provide a set of functions and data types or interpret a custom language to perform MPC operations [Kel20]. Often, implementing applications using these frameworks requires a lot of effort and expert knowledge. riscMPC tries to make MPC more approachable by allowing the use of any language (assuming it can be compiled to RISC-V) for both plain and secret-shared computation.

Outline. This thesis is structured as follows: In Chapter 2, we introduce the relevant background information. Chapter 3 gives insight into the implementation and challenges of riscMPC. Chapter 4 shows the results of this thesis. Finally, in Chapter 5, we conclude with a discussion of our findings and directions for future work.

Chapter 2

Background

In this chapter, we will look at the necessary background information about the building blocks of MPC and different MPC protocols. Additionally, we will give a basic introduction to RISC-V.

2.1 Multi-Party Computation

Multi-Party Computation (MPC) is a cryptographic technique that facilitates secure computation between mutually untrusting parties. The parties jointly perform computation over their inputs without revealing them to each other. In the end, they only learn the result of the computation and nothing about the secret inputs of other parties. It was first introduced by Yao in 1982 as a solution to Yao's Millionaires' Problem [Yao82]. In this problem, two millionaires want to find out who the wealthiest of the two is without revealing their actual wealth to anyone.

Since then, much research has been done in this area, and many different protocols have been proposed [DSZ15, GMW19, MR18, Pat+21]. These protocols can be categorized by different attributes.

- **Number of parties:** Some protocols, such as ABY2 and ABY3, only support two or three parties. Others support an arbitrary amount of parties. Limiting the number of parties enables the use of specialized protocols that have higher performance compared to protocols that support an arbitrary number of parties.
- **Security assumptions:** Often, adversaries are assumed to be semi-honest or malicious. A semi-honest (honest but curious) adversary will follow the protocol but will use all the information they have to try to gain access to secret inputs. Malicious adversaries, on the other hand, can deviate from the protocol in arbitrary ways. Semi-honest protocols typically offer higher performance compared to others that have to account for malicious adversaries.
- **Number of corrupted parties:** Some protocols' security guarantees only hold if there is an honest majority; others can even cope with a dishonest majority.

Protection against a dishonest majority can come with a considerable performance cost. This is one reason why protocols with an honest majority are very popular.

2.1.1 Arithmetic Secret Sharing

Arithmetic secret sharing shares a value $x \in \mathbb{Z}_Q$ where $\mathbb{Z}_Q$ denotes the ring of integers $\bmod Q$, where $Q = 2^l$, for parties $p \in \mathcal{P}$. A ring is a set with two binary operations, addition $(+)$ and multiplication $(\cdot)$. $[x]$ is the arithmetic sharing of x and $[x]_p \in \mathbb{Z}_Q$ is the share belonging to party $p \in \mathcal{P}$. The shares $[x]$ are constructed, such that $x = \sum_{p\in\mathcal{P}} [x]_p \bmod Q$, i.e., their sum modulo Q is equal to x. The last share is computed as $[x]_n = \sum_{i=0}^{n-1} [x]_i - x \bmod Q$, such that $x = \sum_{i=0}^{n-1} [x]_i + [x]_n \bmod Q$. For each party $p \in \mathcal{P}$, except the last party, a random value is drawn from $\mathbb{Z}_Q$.

Addition: To add two secret-shared values $[x]$ and $[y]$, such that $z = x + y$, each party $p \in \mathcal{P}$ locally computes $[z]_p = [x]_p + [y]_p$. When revealing the shares $[z]$, we compute $z = \sum_{i=0}^{n} z_i \bmod Q = \sum_{i=0}^{n} [x]_i + [y]_i \bmod Q = x + y$. Addition of a secret-shared value $[x]$ and a public value y can be implemented by having one party p adding y to its share $[x]_p$, such that $z = \sum_{i=0}^{n} [x]_i + y \bmod Q$.

Multiplication: Multiplication of a secret-shared value $[x]$ with a public value y can be computed locally by each party by computing $[z]_p = [x]_p \cdot y$. Upon revealing $[z]$, we get $\sum_{i=0}^{n} [x]_i \cdot y = y \cdot \sum_{i=0}^{n} [x]_i = xy$. Unfortunately, the multiplication of two secret-shared values $[x]$ and $[y]$ cannot be done as easily. Simply multiplying the shares locally for each party does not work.

$$z = \sum_{i=0}^{n} [z]_i = \sum_{i=0}^{n} [x]_i \cdot [y]_i \neq \sum_{i=0}^{n} [x]_i \cdot \sum_{i=0}^{n} [y]_i \tag{1}$$

Instead, we can use Beaver triples proposed by Donald Beaver [Bea92]. Beaver triples (also called multiplication triples) are three values $\in \mathbb{Z}_Q$, such that $c = a \cdot b$. More specifically, we use additive shares $[a], [b], [c]$ where no party can know the public values a, b, c. Given a multiplication triple $[a], [b], [c]$, parties compute $[d] = [x] - [a]$ and $[e] = [y] - [b]$. Then they reveal the shares $[d]$ and $[e]$ and compute $[z] = d \cdot e + d \cdot [b] + e \cdot [a]$. After revealing $[z]$, we get $z = x \cdot y$.

The proof that this trick works is as follows:

$$
\begin{aligned}
[z] &= d \cdot e + d \cdot [b] + e \cdot [a] + [c] \\
z &= de + d \sum_{i=0}^{n} [b]_i + e \sum_{i=0}^{n} [a]_i + \sum_{i=0}^{n} [c]_i \\
&= de + db + ea + c \\
&= (x-a)(y-b) + (x-a)b + (y-b)a + c \\
&= xy - xb - ya + ab + xb - ab + ya - ba + ab \\
&= xy
\end{aligned} \tag{2}
$$

Revealing $[d]$ and $[e]$ does not leak any information about the shares $[x]$ and $[y]$ because $[a]$ and $[b]$ are used to hide the secret values. Additionally, since both d and e are public values, adding de must only be done by one party. The multiplication triples can be generated in an offline preprocessing phase and later used in the online phase. Per multiplication of two secret-shared values, one Beaver triple is needed. It is crucial that a Beaver triple is not used more than once. Otherwise, an attacker could learn previous secret values. ■

2.1.2 Binary Secret Sharing

Binary secret sharing is similar to arithmetic secret sharing, but instead of working in the ring $\mathbb{Z}_Q$, it uses the binary field (ring where Q is prime) $\mathbb{Z}_2$. In binary secret sharing, $\langle x \rangle$ denotes the shares of a value x by sharing all its bits in $\mathbb{Z}_2$. The shares $\langle x \rangle$ are constructed such that $x = \bigoplus_{p \in \mathcal{P}} \langle x \rangle_p$, i.e., XOR of the shares is equal to x.

XOR: To XOR two binary secret shares $\langle x \rangle$ and $\langle y \rangle$, such that $z = x \oplus y$, each party $p \in \mathcal{P}$ locally computes $\langle z \rangle_p = \langle x \rangle_p \oplus \langle y \rangle_p$. When revealing the shares, we compute $z = \bigoplus_{i=0}^{n} \langle x \rangle_i \oplus \langle y \rangle_i$. XOR of a secret-shared value $\langle x \rangle$ and a public value y is done by having one party p XOR y and its share $\langle x \rangle_p$, such that $z = \bigoplus_{i=0}^{n} \langle x \rangle_i \oplus y$.

AND: AND of a binary secret-shared value $\langle x \rangle$ with a public value y can be computed locally by each party by computing $\langle z \rangle_p = \langle x \rangle_p \wedge y$. Upon revealing $\langle z \rangle$, we get $\bigoplus_{i=0}^{n} \langle x \rangle_i \wedge y = y \wedge \bigoplus_{i=0}^{n} \langle x \rangle_i = x \wedge y$. When computing the AND of two binary secret shares $\langle x \rangle$ and $\langle y \rangle$, we face the same issue as with the multiplication of two arithmetic shares. Fortunately, we can again use the Beaver trick and compute $z = x \wedge y$ analogous to $x \cdot y$. Instead of using a multiplication triple of arithmetic shares, we now use an AND triple of binary shares $\langle a \rangle, \langle b \rangle, \langle c \rangle$ such that $c = a \wedge b$. The result can be computed as $\langle z \rangle = (d \wedge e) \oplus (d \wedge \langle b \rangle) \oplus (e \wedge \langle a \rangle)$.

2.1.3 Share Conversions

Some protocols, such as ABY2 and ABY3, support conversion between different sharing types. In the case of ABY, conversions from arithmetic (A) to binary (B) and to Yao

(Y) are possible [DSZ15, MR18, Pat+21]. Such mixed protocol frameworks often perform better because they can perform more operations, such as XOR with binary shares or ADD with arithmetic shares, instead of using other more expensive procedures. The following sections briefly explain share conversions between arithmetic and binary for a two-party setting. More information about conversions from and to Yao can be found in [DSZ15] and will not be covered further.

Arithmetic to Binary: A2B share conversation can be done with a binary addition circuit. Both parties binary secret-share their shares $[x]_i$ for $i \in \{0,1\}$. Then, they use $\langle [x]_0 \rangle$ and $\langle [x]_1 \rangle$ as inputs to a binary adder. The result of this addition is $\langle x \rangle$. Each A2B conversion needs 13 AND gates in the optimized 64-bit adder circuit described in [DSZ15, Kno+21]. One AND gate in the beginning and two for each of the six masks used in the circuit. The binary AND triples can be generated in the setup phase. More information on the conversions can be found here [DSZ15].

Binary to Arithmetic: A technique similar to the multiplication triple generation can be used to perform A2B conversion in a performant way. We perform Oblivious Transfer (OT) for each bit to obliviously transfer two additively correlated values by a power of two. In more detail, the sender (party 0) randomly chooses $r_i \underset{R}{\in} \{0,1\}^l$ for each OT of the l-bit value (where l is the bit width of the modulus). The inputs (s_{i0}, s_{i1}) are defined as $s_{i0} = \left(1 - \langle x \rangle_0[i]\right) \cdot 2^i - r_i$ and $s_{i1} = \langle x \rangle_0[i] \cdot 2^i - r_i$. The receiver (party 1) chooses one of these values based on the bits values of his share $\langle x \rangle_1[i]$. By adding all of the random values r_i, the sender obtains $[x]_0 = \sum_{i=1}^{l} r_i$. The receiver adds up all s_{i*} values, which results in $x - [x]_0$ [DSZ15].

Communication: As shown in Table 1, converting from arithmetic to binary shares incurs $2l\kappa + l$ bits of communication and is executed over two rounds. Where κ is the symmetric security parameter used in OT as number of base-OTs and the field size in Elliptic Curve Cryptography (ECC). To convert from binary back to arithmetic shares, a communication of $2l\kappa$ bits is necessary [DSZ15].

Conversion	**Setup**	**Online**	
	Comm (bits)	Comm (bits)	Rounds
A2B	$4l\kappa$	$2l\kappa + l$	2
B2A	–	$2l\kappa$	2

Table 1: Communication cost of A2B and B2A conversion in setup and online phase. The numbers are given for l-bit values. κ is the symmetric security parameter. [DSZ15]

2.1.4 Fixed-Point Encoding

Additive secret sharing (and other sharing schemes) only works on integer arithmetic in a ring $\mathbb{Z}_{2^l}$ or a finite field $\mathbb{F}_p$. But for some use cases, such as PPML, we would like to use real numbers $\mathbb{R}$. However, a real number $\widetilde{x} \in \mathbb{R}$ cannot be secret-shared in the same way as an integer $x \in \mathbb{Z}_{2^l}$. Instead, we can approximate $\widetilde{x}$ by encoding it as a fixed-point number by multiplying it with some scaling factor (2^k where k is the precision in bits) and rounding it to get whole integers. This means that we lose some precision compared to float-point numbers, but we now have integers that can be secret-shared. This way of embedding real numbers in integers is called fixed-point embedding.

$$x = \lfloor \widetilde{x} \cdot 2^k \rfloor \tag{3}$$

Addition and subtraction with secret-shared fixed-point numbers work analogous to secret-shared integers. However, in multiplication, we end up with a scale of 2^{2k} because both values were multiplied with 2^k. Therefore, we need to divide the product by 2^k to get rid of the additional factor.

In all operations with a secret-shared fixed-point number and a public float, the public value also has to be embedded.

2.1.5 Use Cases

In this section, we will take a look at two important use cases of MPC.

Private Set Intersection: PSI is a common application of MPC. Two or more parties want to find the intersection of their datasets without revealing any information other than the intersection itself. This can be achieved using general-purpose MPC or specialized protocols [Bay+21].

One application for PSI is Private Contact Discovery, where users and service providers privately compare their lists of contacts. Users only learn which of their contacts also use the service and learn nothing about other users that are not in their contacts. Service providers only learn what other users are in a certain user's list of contacts but learn nothing about contacts who don't use the service [Dem+18]. For more details, see Section 2.5.

Mean of Salaries: One well-known use case of MPC is the secret computation of the mean salary of multiple parties. Often, due to legislation or other factors, employers are not able to share salary data. To gather important statistics such as mean salaries, MPC can be used to perform the mean calculation in secret [Lap+18].

Privacy-preserving Machine Learning: PPML has been gaining more and more interest over the past years. With the increasing application of Machine Learning (ML) to various problems, concerns about user privacy cannot be ignored. Furthermore, ML model providers often spend considerable amounts of time and resources on model training and are not willing to share their models just to respect the privacy of users. The two mutually untrusting parties that want to perform computation on their private inputs is a classic MPC setting. The user inputs the data that the model should evaluate, and the service provider inputs the ML model. The evaluation is performed with MPC protocols, and only the output of the model is revealed to the user [Kno+21]. For more details, see Section 2.7.

2.2 Oblivious Transfer

Oblivious Transfer (OT) is a cryptographic primitive that allows one party, known as the sender, to transfer one of n inputs to another party, known as the receiver. The key characteristic of oblivious transfer is that the sender remains oblivious to the choice made by the receiver and that the receiver learns nothing about the message that was not chosen. This makes it a powerful tool in secure MPC and privacy-preserving protocols.

One example of an OT protocol is the semi-honest 1-out-of-2 Oblivious Transfer introduced by Naor and Pinkas in 1999 [NP99]. In this protocol, the sender has two inputs (m_0 and m_1), and the receiver wants to obtain one of them without the sender learning which one was chosen. Furthermore, even if Bob is malicious and tries to obtain both m_0 and m_1, he should gain no information about the other message he did not choose, and Alice should not be able to determine Bob's choice.

Oblivious transfer protocols have applications in secure two-party computation, where privacy is crucial, such as in secure voting systems or secure auctions. They provide a means for multiple parties to jointly compute a function over their inputs while keeping those inputs private from each other.

2.2.1 Chou Orlandi (Simplest) OT

Tung Chou and Claudio Orlandi proposed their 1-out-of-2 OT protocol based on the Diffie-Hellman (DH) protocol in 2015 [CO15]. The protocol starts identical to DH, where both parties (Alice and Bob) sample a random value from a group $\mathbb{G}$ with a generator g. Alice, who acts as the sender, then sends $A = g^a$ to Bob, who acts as the receiver. Bob then computes B as a function of his choice bit c: if $c = 0$: Bob computes $B = g^b$ else if $c = 1$: Bob computes $B = Ag^b$. After sending B to Alice, both parties derive keys from the previously exchanged values. Alice derives $k_0 = H(B^a)$ and $k_1 = H\left(\left(\frac{B}{A}\right)^a\right)$, while Bob derives $k_c = H\left(A^b\right)$ corresponding to his choice bit. Finally, Alice uses the two keys to

encrypt the two messages M_0 and M_1 and sends them to Bob. Bob can only decrypt the message M_c corresponding to his choice bit, and Alice learns nothing about Bob's choice [CO15].

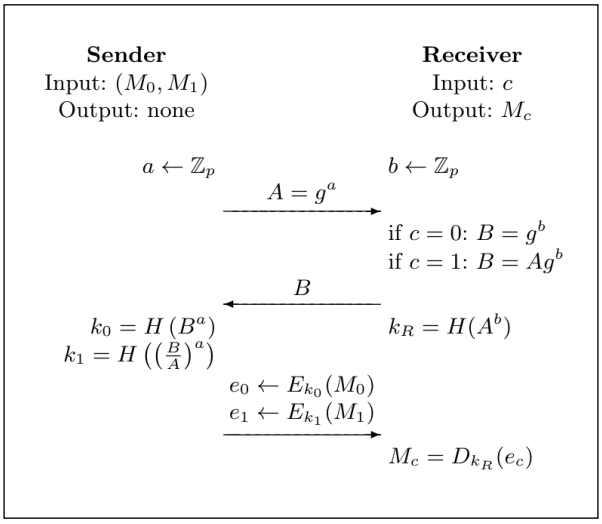


Figure 1: Simplest OT protocol [CO15]

2.2.2 OT Extensions

Many use cases for OT require a very large number of OTs. While state-of-the-art OT protocols such as Chou Orlandi OT can achieve up to $10,000$ 1-out-of-2 OTs per second [Ash+13, CO15], typical applications may need orders of magnitude more.

- The Advanced Encryption Standard (AES) circuit has $\approx 10,000$ AND gates and requires $20,000$ semi-honest or $\approx 2^{20}$ malicious OTs.
- The PSI circuit (Sort-Compare-Shuffle) has 2^{25} AND gates and requires 2^{26} semi-honest or $\approx 2^{30}$ malicious OTs [Ash+13].

To handle such high numbers of OTs, OT extensions [Ash+13, Bea96, Ish+03] can be used. OT extension protocols first perform a small number of base-OTs that can be extended to many OTs using only symmetric cryptographic operations. This can be thought of as a hybrid encryption scheme where, instead of using asymmetric encryption on large messages, which would be expensive, we only asymmetrically encrypt a key that is then used for symmetric encryption of the message. With the number of base-OTs being as low as 80 or 128, OT extension protocols are crucial for performance-critical applications.

2.2.3 Asharov-Lindell-Schneider-Zohner OT

The Asharov-Lindell-Schneider-Zohner (ALSZ) OT protocol [Ash+13] is an improved version of the IKNP protocol [Ish+03].

On a high level, ALSZ work as follows:

In the initial OT phase, the sender P_S creates a random vector $s \in \{0,1\}^l$. The receiver P_R chooses l (= number of baseOTs) pairs of seeds (k_i^0, k_i^1) where each seed is of size κ (symmetric security parameter). The parties then perform the base-OTs with s as choice bits and the seed pairs as input. The seeds, together with a pseudorandom generator G, are used to generate a bit matrix T.

In the OT extension phase, the P_R uses T to compute u_i and sends it to P_S, while P_S uses T to compute q_i. P_S then constructs the bit matrix Q. Together with a hash function H, P_S constructs m pairs (y_j^0, y_j^1) for each $j \in 0..m$ and sends them to P_R. P_R then computes the outputs using the choice bits r.

Figure 2 shows a detailed explanation of the ALSZ OT extension protocol.

Optimized semi-honest secure OT protocol

- **Input of** P_S: m pairs (x_j^0, x_j^1) of n-bit strings, $1 \leq j \leq m$.
- **Input of** P_R: m choice bits $r = (r_1, ..., r_m)$.
- **Common input**: Symmetric security parameter κ and number of base-OTs $l = \kappa$.
- **Cryptographic primitives**: The parties use a pseudorandom generator $G : \{0,1\}^\kappa \to \{0,1\}^m$ and a correlation-robust hash function $H : [m] \times \{0,1\}^l \to \{0,1\}^n$.

1) *Initial OT phase*:

   a) P_S initializes a random vector $s = (s_1, ..., s_l) \in \{0,1\}^l$ and P_R chooses l paris of seeds k_i^0, k_i^1 each of size κ.
   b) The parties use a $l \times$ OT protocol, where P_S acts as the receiver with input s and P_R acts as the sender with inputs (k_i^0, k_i^1) for every $1 \leq i \leq l$.

   For every $1 \leq i \leq l$, let $t^i = G(k_i^0)$. Let $T = [t^1 \mid ... \mid t^l]$ denote the $m \times l$ bit matrix where its ith column is t^i for $1 \leq i \leq l$. Let t_j denote the jth row of T for $1 \leq j \leq m$.

2) *OT Extension phase*:

   a) P_R computes $t^i = G(k_i^0)$ and $u^i = t^i \oplus G(k_i^1) \oplus r$, and sends u^i to P_S for every $1 \leq i \leq l$.
   b) For every $1 \leq i \leq l$, P_S defines $q^i = (s_i \cdot u^i) \oplus G(k_i^{s_i})$. (Note that $q^i = (s_i \cdot r) \oplus t^i$.)
   c) Let $Q = [q^1 \mid ... \mid q^l]$ denote the $m \times l$ bit matrix where its ith column is q^i. Let q_j denote the jth row of the matrix Q. (Note that $q^i = (s_i \cdot r) \oplus t^i$ and $q_j = (r_j \cdot s) \oplus t_j$.)
   d) P_S sends (y_j^0, y_j^1) for every $1 \leq j \leq m$, where:

$$y_j^0 = x_j^0 \oplus H(j, q_j) \quad \text{and} \quad y_j^1 = x_j^1 \oplus H(j, q_j \oplus s)$$

   e) For $1 \leq j \leq m$, P_R computes $x_j = y_j^{r_j} \oplus H(j, t_j)$.

3) **Output**: P_R outputs $(x_1, ..., x_m)$; P_S has no output.

Figure 2: Protocol for ALSZ optimized semi-honest secure OT [Ash+13]

2.3 RISC-V

RISC-V [WK17] is a free and open standard Instruction Set Architecture (ISA) based on the Reduced Instruction Set Computer (RISC) architecture design. It was initially designed for research and education, and unlike most other ISA, it is provided under a

royalty-free open-source license. In contrast to Complex Instruction Set Computer (CISC) architectures, RISC focuses on simple and fewer instructions instead of many complex instructions. Instructions for RISC architectures can be executed within one clock cycle. In CISC architecture, complex instructions often need many cycles. Another difference between the two is that RISC separates `LOAD` and `STORE` instructions from, e.g., arithmetic instructions such as `MUL`. This can be a very important advantage if following instructions can reuse the same values already present in the registers. CISC architectures would potentially have to re-load the data from memory. One downside of the limited number of instructions of RISC architectures is the increased size of programs because many instructions need to be used instead of one more complex instruction. However, the fewer and simpler instructions are a lot easier to decode for the CPU, so less space and power are needed for decoding circuits [Jam95].

Figure 3: RISC-V logo [WK17]

2.3.1 Overview

The RISC-V Instruction Set Architecture (ISA) is defined as a base integer ISA that can be extended to add new functionalities. These extensions provide additional support for, e.g., 64-bit architectures, multiplication/division, or floating-point numbers. The base ISA uses fixed-length 32-bit instructions, which are 32-bit aligned. Like x86, RISC-V also uses a little-endian memory system.

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Figure 4: RISC-V register names and descriptions [WK17]

In the base ISA, there are 31 general-purpose registers x1-x31 for integers. The register x0 is hardwired to 0. The "F/D" extension adds 32 additional registers f0-f31 for floating-point numbers. In RV32, the registers are 32 bits wide; in RV64, they are 64 bits wide. Additionally, there is a register for the program counter that holds the address of the current instruction.

The base ISA has four core instruction formats (R/I/S/U), which are all 32-bit long and four-byte aligned. These four formats are used to encode different instructions, some with two source registers and some with immediate values. RISC-V keeps the source registers (rs1, rs2) and the destination register (rd) in the same place to simplify the decoding of instructions.

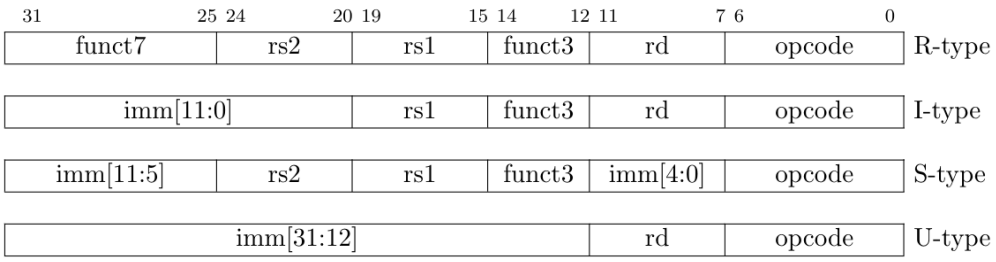


Figure 5: Base ISA instruction formats [WK17]

Some instructions have reserved space for immediate values. These immediates are encoded directly into the instructions. This approach is used for small and constant values that are known at compile time.

Base RISC-V consists of integer computational instructions such as `ADD` (addition), `SUB` (subtraction), `ADDI` (addition with immediate), `AND` (binary AND), `OR` (binary OR), and `XOR` (binary XOR); control transfer instructions such as `JAL` (jump and link), `BEQ` (branch if equal), `BNE` (branch if not equal), `BLT` (branch if less than), and `BGE` (branch if greater or equal); load and store instructions such as `LB` (load byte), `SB` (store byte), `LW` (load word), and `SW` (store word); memory operations, control/status register instructions, environment call, and breakpoints.

2.3.2 Extensions

The RISC-V ISA is designed to be very extendable. In addition to the base integer instruction sets RV32I, RV64I, and RV128I with 32-bit, 64-bit, and 128-bit wide integers/ registers, respectively, there exist different extensions listed in Table 2.

Extension	Description
"M"	Integer Multiplication and Division
"A"	Atomic Instructions
"F"	Single-Precision Floating-Point
"D"	Double-Precision Floating-Point
"Q"	Quad-Precision Floating-Point
"L"	Decimal Floating-Point
"C"	Compressed Instructions
"B"	Bit Manipulation
"J"	Dynamically Translated Languages
"T"	Transactional Memory
"P"	Packed-SIMD Instructions
"V"	Vector Operations

Table 2: RISC-V Extensions

2.3.3 Pseudo Instructions

To improve the development experience of RISC-V assembly, frequently used sequences of instructions can be represented as pseudo instructions. The compiler replaces these pseudo instructions with their actual sequence of instructions. The `NOP` (no-op) pseudo instruction is nothing more than `ADDI x0, x0, 0`, which adds 0 to the `x0` registers. Since the `x0` register is hardwired to the value 0, this instruction will have no effect. Instead of having an additional instruction for "branch if $>$" and "branch if $\leq$" there are pseudo instructions that simply use the existing instruction `BLT rs1, rs2, offset` and `BGE rs1, rs2, offset` and switch the register operands to invert the result. Pseudo instructions add more convenience and readability to RISC-V assembly without increasing the decoding in hardware.

2.4 Rust Programming Language

Rust is a multi-paradigm systems programming language introduced by Mozilla in 2010. It focuses on performance, type-safety, memory-safety, and concurrency. Unlike other memory-safe languages such as JavaScript or Go, Rust does not rely on a garbage collector. Instead, it uses the "borrow checker" to track lifetimes and references during compile time. The "borrow checker" enforces that a piece of data only ever has one owner. To temporarily give access to this data, it can be "borrowed". In Rust, mutability is opt-in,

which means that by default, all variables and references are immutable. The "borrow checker" makes sure that there is at most one mutable reference to every piece of data, and if a mutable reference exists, no other immutable references are allowed.

In addition to "borrowing", ownership of data can also be transferred by "moving". Rust also takes some ideas from functional programming, such as higher-order functions and algebraic data types (e.g. `Option` type).

2.4.1 Syntax & Features

The Rust type system gives us structs with attached methods that are used to build more complex types and functionality. In Rust, enums are a lot more powerful than in other languages. Each enum variant can contain additional data (e.g., `Option` enum where the `Some` variant holds a generic type `T`). Similar to interfaces in other languages, Rust `Traits` are used to define shared behavior.

Listing 1 below shows a "Hello, World!" program in Rust. It uses the `fn` keyword and the `println!` macro.

```
1 fn main() {
2     println!("Hello, World!");
3 }
```

Listing 1: Rust Hello World program

Listing 2 shows the immutability by default concept of Rust. Only variables that are declared as mutable can be mutated.

```
1 fn main() {
2     let foo = 42;
3     foo += 1; // this will not compile because foo is not mutable!
4     let mut bar = 10;
5     bar = 2 // this compiles, because bar is declared with mut
6 }
```

Listing 2: Rust mutability

Listing 3 shows the `Option` type. This type is an enum with the variants `Some(T)` and `None`. An alternative solution would be to use the `Result` enum with variants `Ok(T)` and `Err(E)`. The contained error type can have multiple variants or more details.

```rust
fn div(x: u64, y: u64) -> Option<u64> {
    if y != 0 {
        Some(x / y)
    } else {
        None
    }
}
```

Listing 3: Rust safe division using `Option` type

Listing 4 shows the rules imposed by the "borrow checker". If the two references are used later in the code and not optimized away by the compiler, the following code will not compile.

```rust
fn main() {
    let mut foo = 42;
    let foo_ref = &foo;
    let foo_mut_ref = &mut foo; // will not compile because foo is
already borrowed as immutable
}
```

Listing 4: Rust "borrow checker" example

2.5 Private Set Intersection

Private Set Intersection (PSI) is a cryptographic protocol that allows two parties (A and B) to compute the intersection ($B \cap B$) of their datasets while all their data, other than the intersection, stay hidden from the other party.

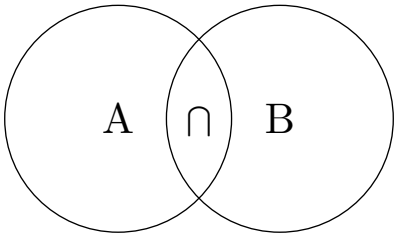


Figure 6: Private Set Intersection

PSI has a wide variety of applications, and a lot of time has been invested in its research. It is used to preserve privacy and keep datasets private in medical analysis, contact discovery, social networks, and many more cases. In addition to the classic definition of PSI, there are many other variants for different cases [MAL23].

- **MP-PSI**: Extends the PSI problem to multiple parties.
- **TH-PSI**: In Threshold PSI, the intersection only gets revealed when its size hits a certain threshold.
- **CA-PSI**: Cardinality PSI only outputs the size of the intersection.
- **SH-PSI**: Size-Hiding PSI ensures that the set sizes also stay hidden.

There exist many different protocols for PSI. Some implementations use MPC, homomorphic encryption, OT, or hash-based techniques. Different protocols work better balanced (both sets are similar in size) or unbalanced (one set is significantly larger than the other) sets [Bay+21, MAL23].

Early PSI protocols were often too computationally expensive and slow, especially for large datasets. Making PSI protocols more usable for large datasets continues to be an active research topic.

2.6 Ascon

ASCON [Dob+21] is a suite of lightweight authenticated encryption and hashing functions. The NIST Lightweight Cryptography competition (2019-2023) selected it as the new standard for lightweight cryptography. All variants of ASCON operate on a 320-bit state and update it in multiple rounds. The permutation in these rounds is based on five 64-bit words (x_{0-4}) and only uses bitwise functions (AND, NOT, XOR) and bit-rotations. This makes ASCON an excellent choice for lightweight devices and also allows it to work with MPC protocols that support binary operations such as ABY3.

2.6.1 Ascon Permutation

All Ascon variants use the same lightweight permutation. The permutations p^a and p^b iteratively apply the Substitution Permutation Network (SPN)-based round transformation $a = 12$ times and $b \in \{6, 8\}$ times. Each round consists of three steps that operate on the 320-bit state [Dob+21].

1) **Addition of Round Constants**: XOR x_2 with 1-byte round specific constant.
2) **Nonlinear Substitution Layer**: Apply the 5-bit S-box to each of the 64 bits in x_{0-4} in parallel.

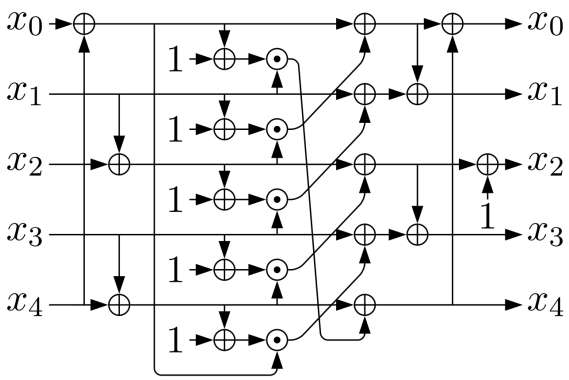


Figure 7: Ascon S-Box [Dob+21]

3) **Linear Diffusion Layer**: XOR each word with rotated copies.

$$\begin{aligned} x_0 &\coloneqq x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28) \\ x_1 &\coloneqq x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39) \\ x_2 &\coloneqq x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6) \\ x_3 &\coloneqq x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17) \\ x_4 &\coloneqq x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41) \end{aligned} \tag{4}$$

2.6.2 Ascon AEAD

Authenticated Encryption with Associated Data (AEAD) is a cryptographic scheme that ensures both the confidentiality and authenticity of data. AEAD integrates symmetric encryption and Message Authentication Code (MAC) functionalities to provide robust security guarantees.

In AEAD, the plaintext is encrypted, and both the ciphertext and additional data (which is not encrypted but is authenticated) are verified for integrity and authenticity. This additional data, known as Associated Data (AD), might include headers or other metadata that need to be authenticated but not encrypted.

The AEAD process typically involves the following steps:

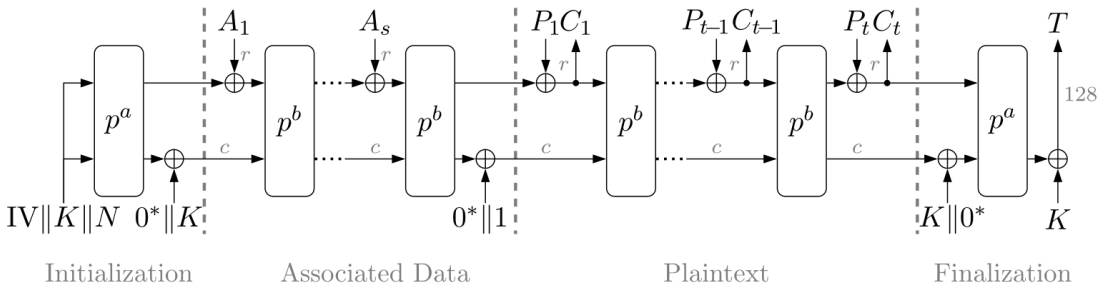
- **Encryption:** The plaintext is encrypted using a symmetric key to produce the ciphertext.
- **Authentication:** A MAC is computed over both the ciphertext and the associated data using the same key, ensuring that any tampering with either the ciphertext or the associated data can be detected.

The encryption procedure with Ascon-128 uses a 128-bit key K, 128-bit nonce N, associated data A, and the plaintext P to produce the ciphertext C and a 128-bit tag T. The associated data and plaintext are consumed in 64-bit blocks.

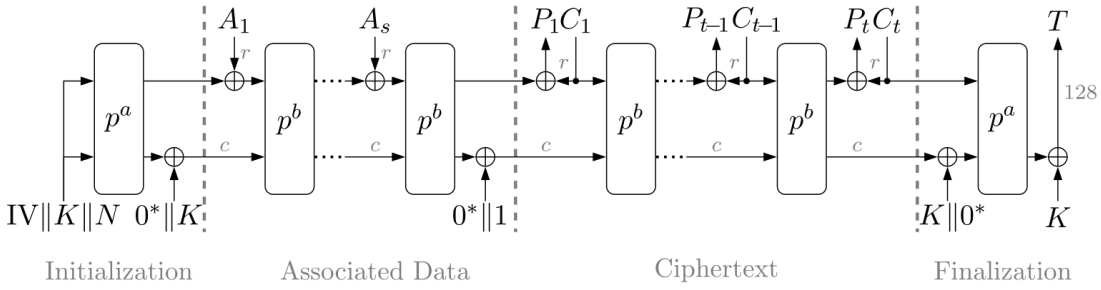
$$\mathcal{E}_{k,r,a,b}(K, N, A, P) = (C, T) \tag{5}$$

The decryption procedure with Ascon-128 uses a key K, nonce N, associated data A, the ciphertext C, and the tag T to produce either the plaintext P if the verification is successful or an error $\perp$ if the tag does not match.

$$\mathcal{D}_{k,r,a,b}(K, N, A, C, T) \in \{P, \perp\} \tag{6}$$



(a) AEAD Encryption



(b) AEAD Decryption

Figure 8: Ascon AEAD modes of operation [Dob+21]

2.6.3 Ascon Hash

The Ascon hash function is based on a sponge construction, a well-known design paradigm for hash functions that provides flexibility and strong security guarantees. Ascon hash involves an absorbing phase, where the input message is processed in 64-bit blocks, and a squeezing phase, where the final hash value is extracted.

Ascon can be instantiated as a hash or Extendable Output Function (XOF) to map the input to a fixed or arbitrary length. It takes the message M and an output length l as inputs and computes the 256-bit hash H as follows:

$$\mathcal{X}_{h,r,a}(M, l) = H \tag{7}$$

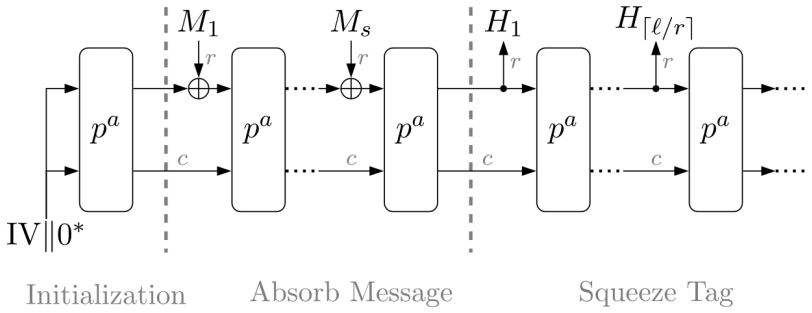


Figure 9: Ascon hash mode of operation [Dob+21]

2.7 Privacy Preserving Machine Learning

Machine Learning (ML) focuses on developing algorithms that enable computers to learn from data and make decisions based on learned parameters. There are many techniques that fall under the term ML, and one of the most common applications are Neural Networks (NNs) [Pop+09]. Traditional NNs are based on the Multilayer Perceptron (MLP) that uses multiple layers of interconnected neurons [Pop+09]. The connections between these neurons have weights and biases that are learned by training models on a large training dataset. This process can be very expensive and take a long time for large (millions or billions of parameters) models.

Since the training of large models is so expensive, it is only feasible for large corporations that let users rent their models. To keep the costly models private, often the end-user has to upload their inputs and trust that the model provider does not use their data for anything other than model evaluation. This is far from ideal. Many advances have been made to enable PPML where neither party has to reveal their model or inputs [MZ17, XBJ21]. This can be achieved by using PETs such as MPC or Homomorphic Encryption [XBJ21]. Given the large performance costs of PPML, it is used more for evaluation/ inference and not on training. The reason for this is that training is orders of magnitude more expensive, even in traditional ML [XBJ21].

2.7.1 MNIST Handwritten Digits

One of the most common applications of ML is image recognition, where models are trained to correctly label images. The Modified National Institute of Standards and Technology (MNIST) database [Den12] provides a widely used dataset of 28x28 grayscale images of handwritten digits (0 - 9).

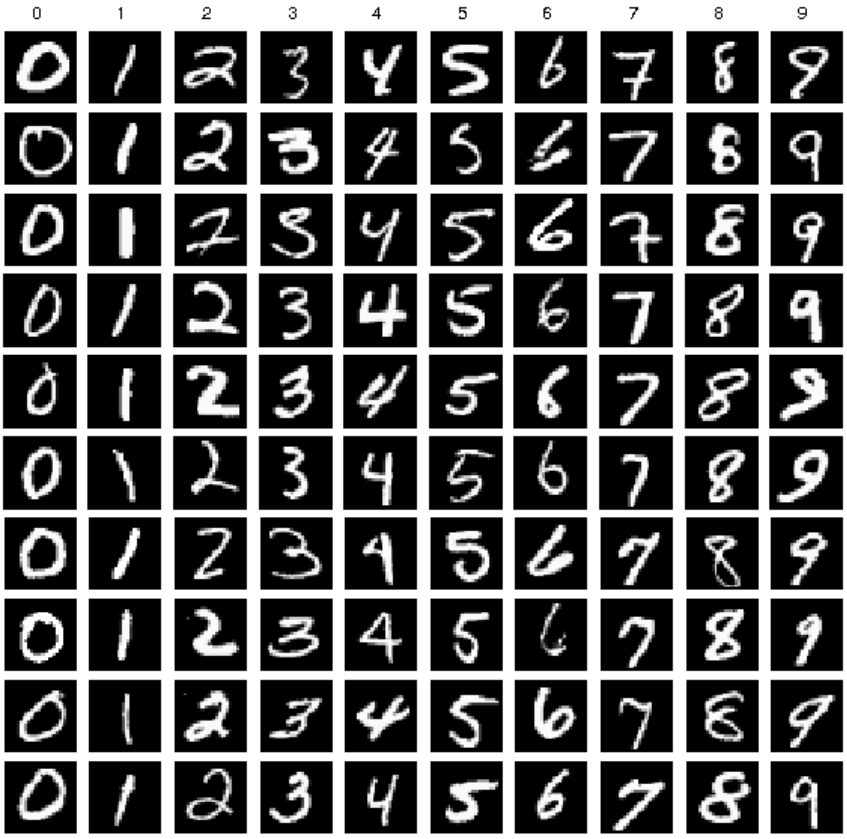


Figure 10: Examples of MNIST handwritten digits recognition dataset [Den12]

Since there are 28x28 pixels in the input images, the first layer (input layer) consists of 784 neurons. Because the images should be labeled as digits, the last layer (output layer) has 10 neurons. Between these two layers are one or more so-called hidden layers of varying sizes.

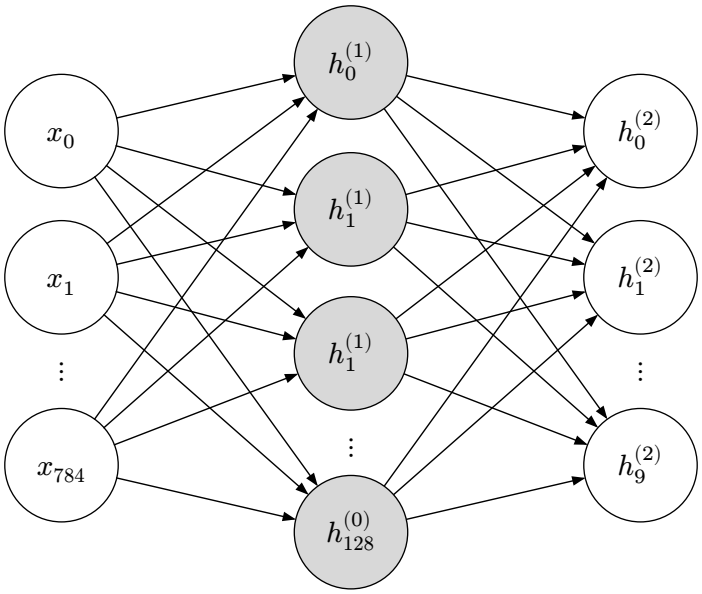


Figure 11: MNIST digit recognition model layout

A common setup is one hidden layer with 100 neurons. All neurons are fully connected with the next layer, and their output depends on the weights and bias of the connection. The output of a neuron (not in the input layer) can be computed as follows:

$$h_j^{(k)} = F\left(\sum_{i=1}^{n} w_{i,j}^{(k)} \cdot h_i^{(k-1)} + b_j^{(k)}\right) \tag{8}$$

The output value of a neuron j is computed by multiplying the output of the neurons in the previous layer $(k-1)$ with all the weights of the connections and adding the bias term. Additionally, an activation function F is applied. This calculation is repeated forward until the output layer is reached. The final outputs correspond to the label predictions.

Chapter 3

Implementation

In this chapter, we will look at the general setup of riscMPC, give insight into the implementation details of individual instructions, and show how to use and interact with the riscMPC library.

3.1 riscMPC Virtual Machine

We implemented riscMPC as a two-party protocol and assumed a semi-honest adversary. For most use cases of MPC, the semi-honest is sufficient because all parties have a shared interest in the result and can thus be assumed to follow the protocols. Often, the motivation to use MPC comes from regulations or the end user's desire for more privacy and not the need to cooperate with completely untrustable parties. Due to the two-party setup, we also have to assume an honest majority.

At the core of riscMPC is the riscMPC VM. Each party runs its VM, which executes the program instruction by instruction. The VM emulates parts of a 64-bit RISC-V CPU. Each VM has access to the 32 general-purpose 64-bit integer registers, a program counter, and virtual memory. To execute an instruction, its operands are fetched from registers or memory.

riscMPC currently supports the RV64 base integer instruction set and the ''M" extension for multiplication and division. Where applicable, instructions can operate on public or secret-shared values. This means that, e.g., `x10, x10, 1` will work differently depending on whether the value stored in `x10` is public or a secret share. Computations based on secret-shared values will always return a secret-shared value, i.e., all values derived from secret values will also be secret. In contrast to other MPC frameworks, there is no way to deliberately reveal a secret-shared value during execution. Instead, one would separate the program into multiple functions such that intermediate results can be returned and revealed and again passed as input into the next function.

We implemented a proof-of-concept riscMPC VM using the Rust programming language [SC23]. Rust not only offers a powerful type system and strong security guarantees but is

also blazingly fast. Most of riscMPC's data types make use of Rust's fat enums, which offer enum variants that can hold additional data. Deserialized RISC-V assembly instructions, arithmetic/binary shares, and public/secret values are all implemented using fat enums. Enums let us match over the different cases and handle all the different variants of instructions, public, binary secret-shared, or arithmetic secret-shared values.

Some RISC-V instructions such as ADD, ADDI, and MUL can directly be mapped to basic MPC operations. Other instructions, such as branches (comparisons), need different MPC protocols.

3.1.1 Instructions

In this section, we show implementation details of all relevant RISC-V assembly instructions (see Section 2.3). All instructions differentiate between public and secret-shared input values. If necessary, the secret-shared values are converted from arithmetic shares to binary shares or vice versa. The list below gives short explanations of the important instructions and describes how they are implemented using MPC operations.

- **ADD, SUB, ADDI:** The ADD instruction takes two source registers, `rs1` and `rs2`, and stores the addition of the two values in the destination register `rd`. Analogously, the SUB instruction subtracts the value in `rs2` from the value in `rs1` and stores the result in `rd`. The ADDI instruction, on the other hand, has an immediate value encoded in the instruction and stores the result of the value in `rs1` plus immediate in `rd`. With public values in the source registers, ADD, SUB, and ADDI perform simple addition, subtraction, and addition with immediate in the ring $\mathbb{Z}_Q$.

  For ADD and SUB, if one of the parameters is a secret-shared value and the other is public, party 0 adds or subtracts the public value to/from its share, and party 1 does nothing. Party 0 adds the public value y to its share $[x]_0$, and party 1 does nothing. ADDI will always be an addition with a public value because the immediate is encoded in the instruction itself and, therefore, known by both parties. In the case where both inputs are secret-shared, both parties add their shares of $[x]$ and $[y]$. The result $[z]$ is stored in the destination register.

- **MUL:** The MUL instruction takes two source registers, `rs1` and `rs2`, and stores the product of the two values in the destination register `rd`. With public values as input, MUL performs multiplication in the ring $\mathbb{Z}_Q$. With one secret-shared input, all parties multiply their share $[x]_i$ with the public multiplier y. Given two secret inputs, the parties perform Beaver multiplication as described in Section 2.1.1.

- **DIV, REM:** The DIV instruction takes two source registers and stores the result of the value in `rs1` divided by the value in `rs2` in the destination register `rd`. REM calculates the remainder of an integer division of `rs1` and `rs2` and stores it in `rd`.

  To implement integer division of a secret value by a public value, we used a simplified version of the probabilistic truncation used in [Kno+21]. Instead of computing the wrap count, we assume it to be one and perform the truncation of a secret share $[x]$ as follows:

  $$\frac{x}{l} = [y] - \frac{2^{64}}{l} \text{ where } [y] = \frac{[x]}{l} \tag{9}$$

  Due to the integer division of the shares, the result of this operation can have an error of ± 1. If the combination of the resulting shares did not wrap around, the result will be completely wrong. This will be the case if the secret input is larger than the first randomly generated share.

- **LD, SD:** The LD instruction takes a base address from the source register `rs1` and an immediate offset value and loads the 64-bit value from base + offset into the destination register `rd`. The SD instruction takes a base address from `rs2` and an immediate offset value and stores the 64-bit value from `rs2` at address base + offset. Note that the values in the base address registers must be public values because riscMPC does not support secret-shared addresses.

  Since riscMPC stores shares as 64-bit values, loading, e.g., the lower 32-bit value and opening it, will not correspond to the lower 32 bits of the secret input. By converting values to binary shares we could load and store smaller width integers in the 64-bit shares. Doing so would significantly impact the performance of memory accesses. Therefore, we chose to only support full-width integers.

- **SLL, SRL, SRA, SLA, SLLI, SRLI, SLAI, SRAI:** The SLL instruction takes two source registers, `rs1` and r2, and stores `rs1` logically left shifted by `rs2` in the destination register `rd`. Other shift instructions work analogously. Immediate versions of these instructions shift the value in `rs1` by the amount specified in the instruction. The shift amount in `rs2` must not be a secret value. All shift instructions can be performed both for public and secret values. Both parties shift secret-shared values (arithmetic or binary). Therefore, the revealed value is also shifted by the same amount.

- **BLT, BGE, BEQ, BNE:** The BLT instruction takes two source registers, `rs1` and `rs2`, and a label. It compares the values stored in `rs1` and `rs2`, and if `rs1` is less than `rs2`, it updates the program counter so that execution continues at the specified label. Branch instructions (i.e., comparisons) use standard comparison operations for

public values. For one or two secret-shared inputs, comparisons are more involved. We use a protocol similar to [Kno+21] for comparisons of secret-shared values. To get $[x] < [y]$, the parties first compute $[x] - [y]$. To get the sign-bit of this subtraction, we perform A2B conversion as described in Section 2.1.3 and right-shift the binary shares by 63.

After opening this binary share, both parties learn $x < y$ but do not learn anything else about x or y. Using $x < y$, we can compute all other comparisons, $x > y$ is just $x < y$ with swapped inputs, $x \geq y$ and $x \leq y$ are just the negations of $x < y$, and $x > y$. Finally, to get $x == y$ we compute $x < y$ and $x \geq y$ to get $\neg <$ and $\leq$ [DSZ15, Kno+21].

- **SLT:** Similar to the branch instructions, the SLT instruction performs the comparison in the same way, but instead of opening the value to perform the conditional jump, we keep the value secret-shared. Instead of revealing the sign-bit of $[x] - [y]$, we keep it secret and convert it to a binary share. The other comparisons are derived as follows [Kno+21]:

$$\begin{aligned} [x > y] &= [y < x] \\ [x \geq y] &= 1 - [x < y] \\ [x \leq y] &= 1 - [y < x] \\ [x == y] &= [x \leq y] - [x < y] \\ [x \neq y] &= 1 - [x == y] \end{aligned} \tag{10}$$

- **XOR, XORI:** The XOR instruction takes two source registers, `rs1` and `rs2`, and stores the logical `XOR` of the values in `rs1` and `rs2` in the destination register `rd`. The immediate version of this instruction calculates the `XOR` of the value in `rs1` and the immediate value. For public inputs, XOR and XORI perform $\oplus$ with two values or with an immediate. Given one binary secret-shared input $\langle x \rangle$ and a public value y, party 0 computes $\langle x \rangle_0 \oplus y$, and party 1 does nothing. For two secret-shared values $\langle x \rangle$ and $\langle y \rangle$, both parties compute $\langle x \rangle \oplus \langle y \rangle$.

- **AND, ANDI:** The AND instruction takes two source registers, `rs1` and `rs2`, and stores the logical `AND` of the values in `rs1` and `rs2` in the destination register `rd`. The immediate version of this instruction calculates the logical `AND` of the value in `rs1` and the immediate value. For public inputs, AND and ANDI perform $\wedge$ with two values or with an immediate. Given one binary secret-shared input $\langle x \rangle$ and a public value y, all parties compute $\langle x \rangle_i \wedge y$. For two secret-shared inputs, we use an and-triple and employ the Beaver trick explained in Section 2.1.1.

- **OR, ORI:** The OR instruction takes two source registers, `rs1` and `rs2`, and stores the logical `OR` of the values in `rs1` and `rs2` in the destination register `rd`. The

immediate version of this instruction calculates the logical `OR` of the value in `rs1` and the immediate value. For public inputs, OR and ORI perform $\vee$ with two values or with an immediate. Given the protocols for XOR and AND, we can compute $\langle x \rangle \vee \langle y \rangle$ as $(\langle x \rangle \wedge \langle y \rangle) \oplus \langle x \rangle \oplus \langle y \rangle$

- **NOT:** The NOT instruction takes one source operand `rs1` and stores the bit-wise negation in the destination register `rd`. Given a binary secret-shared input $\langle x \rangle$, party 0 computes the negation as $\langle \neg x \rangle_0 = \langle x \rangle_0 \oplus -1$, and party 1 does nothing.

- **FADD, FSUB:** The FADD instruction takes two source registers, `rs1` and `rs2`, and stores the sum of both values in the destination register `rd`. When working with real numbers that are fixed-point encoded, addition and subtraction work identically to the addition and subtraction of integers. Given one fixed-point encoded arithmetic share $[x]$ and a public floating-point number $\widetilde{y} \in \mathbb{R}$, the public value $\widetilde{y}$ is first encoded as $y = \lfloor \widetilde{y} \cdot 2^k \rfloor$.

- **FMUL:** The FMUL instruction takes two source registers, `rs1` and `rs2`, and stores the product of both values in the destination register `rd`. When working with real numbers that are fixed-point encoded, multiplication works almost identically to the multiplication of integers. The only difference is that we need to truncate the result of the multiplication afterward (see Section 2.1.4).

  We implemented a probabilistic truncation described by [MZ17].

$$\text{truncate}([x]) = \begin{cases} [x]_0 \gg k & \text{if party} = 0 \\ 2^l - \left(\left(2^l - [x]_1\right) \gg k\right) & \text{if party} = 1 \end{cases} \tag{11}$$

  The result of this truncation is at most off by one with a high probability. In the field $\mathbb{Z}_{2^l}$, let $x \in [0, 2^{l_x}] \cup [2^{l_x}, 2^l)$ where $l > l_x + 1$, the result of a truncation by $l_{\text{trunc}} \leq l_x$ bits is either correct or off by ± 1 with probability $1 - 2^{l_x + 1 - l}$

- **FDIV:** The FDIV instruction takes two source registers, `rs1` and `rs2`, and stores the result in the value in `rs1` divided by the value in `rs2` in the destination register `rd`.

- **FLD, FSD:** The FLD and FSD instructions work identically to the LD and SD instructions.

- **FLT, FLE, FEQ:** The FLT, FLE, and FEQ instructions work identically to their integer equivalents instructions.

- **FMIN, FMAX:** The FMIN and FMAX instructions take two source registers, `rs1` and `rs2`, and store the minimum or the maximum of the two, respectively, in the destination register `rd`. To compute the minimum of two values, we can use a similar

approach as described in [Pat+21]. The two parties secretly compute the LT protocol to get $\text{lt} = \langle x < y \rangle$, a binary sharing of `rs1` < `rs2`. Since this value represents either 1 (if $x < y$ is true) or 0 (if $x < y$ is false), we can convert it to an arithmetic share $[\text{lt}] = \text{a2b}(\langle \text{lt} \rangle)$ and use it to compute min and max as follows:

$$\begin{aligned} \min(x, y) &= \text{lt} \cdot (x - y) + y \\ \max(x, y) &= \text{lt} \cdot (y - x) + x \end{aligned} \tag{12}$$

- **FSGNJ, FSGNJN, FSGNJX:** The FSGNJ, FSGNJN, and FSGNJX instructions take two source operands, `rs1` and `rs2`, and store `rs1` with the sign of `rs2`, ¬ sign of `rs2`, or the ⊕ of both sign bits, respectively, in the destination register `rd`. For normal floating-point values, the sign is determined by the sign-bit (0 for positive, 1 for negative). To secretly evaluate the sign of fixed-point encoded value, we compute $\text{sign}([x]) = 2 \cdot [x > 0] - 1$ [Kno+21]. We can the use $\text{sign}([x])$ to compute $\text{abs}([x]) = [x] \cdot \text{sign}([x])$.

  With these two primitives we can recreate the sign injection instructions on secret-shared values. For FSGNJ we compute the absolute value $[\text{abs}] = \text{abs}([x])$ and the sign $[\text{sign}] = \text{b2a}(\text{sign}([y]))$. We then compute the new value as $[\text{abs}] \cdot [\text{sign}]$ and store it in `rd`. For FSGNJN, we have to negate the sign using $[\neg\ \text{sign}] = 0 - [\text{sign}]$, and for FSGNJX, we compute the sign of $[x]$ and $[y]$, and use $\langle \text{sign\_x} \rangle \oplus \langle \text{sing\_y} \rangle$ as the new sign.

- **FNEG:** The FNEG instruction takes one source operand `rs1` and stores the value with its sign negated in the destination register `rd`. To secretly negate the sign of a fixed-point encoded value, we compute $[\neg x] = -1 \cdot [x]$

- **FABS:** The FABS instruction takes one source operand `rs1` and stores the absolute value in the destination register `rd`. We can the use $\text{sign}([x])$ to compute $\text{abs}([x]) = [x] \cdot \text{sign}([x])$.

- **FSQRT:** The FSQRT instruction takes one source operand `rs1` and stores the square root of the value in the destination register `rd`. Many functions, such as computing the square root, are very expensive, using only addition, multiplication, and division (truncation). However, we can use numerical approximations to compute these functions in a faster but less accurate way.

  To approximate the square root, we use Newton's method, as explained in [Kno+21]. Calculating the square root directly is quite inefficient because it includes division by a secret value. Instead, we use Newton's method to compute the inverse square root:

$$y_{n+1} = \frac{1}{2} y_n (3 - x y_n^2) \tag{13}$$

Then, compute the square root of x by multiplying x with the inverse square root: $\sqrt{x} = x\frac{1}{\sqrt{x}}$. To get a reasonably accurate approximation, we use three iterations for Newton's method. Additionally, we need to estimate the initial value of y_0. We use the approach of [Kno+21] and set $y_0 = \exp\left(-\left(\frac{x}{2} + 0.2\right)\right) \cdot 2.2 + 0.2 - \frac{x}{1024}$. Where exp is approximated using eight iterations ($n = 3$) of:

$$\exp(x) = \lim_{n \to \infty} \left(1 + \frac{x}{2^n}\right)^{2^n} \tag{14}$$

- **VMUL, VAND:** The VMUL and VAND instructions take two source operands, vs1 and vs2, and perform vectorized multiplication or binary AND, respectively. The vector lengths can be configured using the VSETVL instruction. The VLE and VSE instructions are used to load data from and into the vector registers.

  Since multiplication and binary AND of secret values come with communication overhead because we need to open the two values d and e (see Section 2.1), it is a lot more efficient to transfer these values at once rather than individually. For a large number of multiplications, this can significantly improve performance, as can be seen in Figure 21.

3.2 Data Types

Integer registers (also called x-registers) contain `Integer` enums. This enum has the two variants `Public(u64)` and `Secret(Share)`. The `Public` variant just contains the public 64-bit integer. The `Secret` variant contains the `Share` enum to differentiate between `Arithmetic(u64)` and `Binary(u64)` shares. Floating-point registers (also called f-registers) contain `Float` enums. This enum has the two variants `Public(f64)` and `Secret(u64)`. The `Public` variant just contains the public 64-bit float. The `Secret` variant contains a share of the fixed-point encoded float.

```
1 enum Integer {
2     Secret(Share),
3     Public(u64),
4 }
```

Listing 5: `Integer` enum used to represent public and secret integers

```
enum Float{
    Secret(u64),
    Public(f64),
}
```

Listing 6: `Float` enum used to represent public and secret floating-point numbers

```
enum Share {
    Arithmetic(u64),
    Binary(u64),
}
```

Listing 7: `Share` enum used to represent arithmetic and binary shares

```
enum Value {
    Integer(Integer),
    Float(Float),
}
```

Listing 8: `Value` enum used to represent integers and floats in memory

The 32 base integer registers are implemented as an array of `Integer` enums. To read and write to a register, the `XRegister` enum is used to index this array. Similarly, the `FRegister` enum is used to access the 32 floating-point registers. To emulate the full 64-bit address range, the memory is implemented using a map with the address as key and `Value` as value. Elements in this map are required to be 64-bit aligned, so addresses 0x0 - 0x7 are all within the first entry.

3.3 Application Program Interface

In this section, we describe how to use the riscMPC library API and present a PSI example.

3.3.1 Channel

The API of riscMPC exposes a `Channel` trait that users can implement to use different communications channels. Any implementation of `Channel` has to implement the `send()` and `recv()` methods. We provide a `TcpChannel` and a `ThreadChannel`.

```
fn TcpChannel::new(id: usize, addr: SocketAddr) -> Result<TcpChannel> {}
```

Listing 9: `TcpChannel` instantiation method

With the associated new() method of `TcpChannel`, we instantiate a communication channel. This function takes in the party id $\in \{0, 1\}$ and IP address + port. Party 0 passes its own IP + desired port; party 1 passes the IP + port of party 0.

3.3.2 Party

With `Channel`, a generic `PartyBuilder` can be used to instantiate a party. We use the builder pattern to give the user a clean and simple way to get a `Party` instance with the following `PartyBuilder` functions (Listing 10, Listing 11, Listing 12, Listing 15).

```
// instantiate new builder
fn PartyBuilder::new(id: usize, ch: C) -> Result<PartyBuilder<C>> {}
```

Listing 10: `PartyBuilder` builder pattern instantiation

To input public or secret values, the `register_u64()` and `register_f64()` can be used to input a `Integer` or `Float` enum into registers. The `register_shared_u64()` and `register_shared_f64()` functions are used to pass pre-shared secret values. By passing a `Integer::Public(u64)` or `Float::Public(f64)` variant, a public value gets stored at the specified register. By passing a `Integer::Secret(Share)` variant, the value first gets secret-shared, and then, each party stores its share at the specified location. To use secret-shared real numbers, the `f64` value needs to be embedded into a `u64` (see Section 2.1.4). For a definition of all functions used to set registers, see Listing 11.

```
// set integer register to public or secret `Integer`
fn PartyBuilder::register_u64(
    mut self,
    register: XRegister,
    integer: Integer
) -> PartyBuilder<C> {}

// set integer register to pre-shared secret `Integer`
fn PartyBuilder::register_shared_u64(
    mut self,
    register: XRegister,
    integer: Integer
) -> PartyBuilder<C> {}

// set integer register to public or secret `Float`
fn PartyBuilder::register_f64(
    mut self,
    register: FRegister,
    float: Float
) -> PartyBuilder<C> {}

// set integer register to pre-shared secret `Float`
fn PartyBuilder::register_shared_f64(
    mut self,
    register: FRegister,
    float: Float
) -> PartyBuilder<C> {}
```

Listing 11: Builder pattern setters for public or secret registers

To input public or secret values, the `address_u64()` and `address_f64()` can be used to input a `Integer` or `Float` enum into memory. The `address_shared_u64()` and `address_shared_u64()` functions are used to pass pre-shared secret values. For a definition of all functions used to set memory addresses, see Listing 12.

```
// write public or secret `Integer` to address
fn PartyBuilder::address_u64(
    mut self,
    address: Address,
    integer: Integer
) -> Result<PartyBuilder<C>> {}

// write pre-shared secret `Integer` to address
fn PartyBuilder::address_shared_u64(
    mut self,
    address: Address,
    integer: Integer
) -> Result<PartyBuilder<C>> {}

// write public or secret `Float` to address
fn PartyBuilder::address_f64(mut self,
    address: Address,
    float: Float
) -> Result<PartyBuilder<C>> {}

// write pre-shared secret `Float` to address
fn PartyBuilder::address_shared_f64(
    mut self,
    address: Address,
    float: Float
) -> Result<PartyBuilder<C>> {}
```

Listing 12: Builder pattern setters for public or secret addresses

To input vectors of public or secret values, the `address_range_u64()` and `address_range_f64()` can be used to input a `Vec<Integer>` or `Vec<Float>` into memory. The `address_range_shared_u64()` and `address_range_shared_u64()` functions are used to pass pre-shared vectors of secret values. These functions take a base address and store each element of the passed vector at a 64-bit offset. For a definition of all functions used to set memory address ranges, see Listing 13.

```
// write vector of public or secret `Integer` to address
fn PartyBuilder::address_range_u64(
    mut self,
    address: Address,
    integers: Vec<Integer>
) -> Result<PartyBuilder<C>> {}

// write vector of pre-shared secret `Integer` to address
fn PartyBuilder::address_range_shared_u64(
    mut self,
    address: Address,
    integers: Vec<Integer>
) -> Result<PartyBuilder<C>> {}

// write vector of public or secret `Float` to address
fn PartyBuilder::address_range_f64(
    mut self,
    address: Address,
    floats: Vec<Float>
) -> Result<PartyBuilder<C>> {}

// write vector of pre-shared secret `Float` to address
fn PartyBuilder::address_range_shared_f64(
    mut self,
    address: Address,
    floats: Vec<Float>
) -> Result<PartyBuilder<C>> {}
```

Listing 13: Builder pattern setters for public or secret address ranges

If the program will contain multiplication or binary AND with secret-shared values, the methods `n_mul_triples()` and `n_and_triples()` need to be used to specify what and how many Beaver triples should be generated in the offline phase.

```
// set number of multiplication triples
fn PartyBuilder::n_mul_triples(mut self, n: u64) -> PartyBuilder<C> {}

// set number of and triples
fn PartyBuilder::n_and_triples(mut self, n: u64) -> PartyBuilder<C> {}
```

Listing 14: Builder pattern setters for Beaver triples

Finally, to create a `Party`, we use the `build()` method. Sharing inputs and offline Beaver triple generation happen during the final `build()` call.

```
// build party, share secret inputs, offline setup phase
fn PartyBuilder::build(mut self) -> Result<Party<C>> {}
```

Listing 15: `PartyBuilder` build function

Listing 16 shows how to use the builder pattern to create a party for the PSI example.

```
let set0 = BTreeSet::from([1, 2, 3]);
let set0_len = set.len() as u64;
let set1_len = set0_len;

let ch = TcpChannel::new(PARTY_0, "127.0.0.1:8000".parse()?)?;
let mut party = PartyBuilder::new(PARTY_0, ch)
    .register_u64(XRegister::x10, Integer::Public(set0_addr))
    .register_u64(XRegister::x11, Integer::Public(set0_len))
    .register_u64(XRegister::x12, Integer::Public(set1_addr))
    .register_u64(XRegister::x13, Integer::Public(set1_len))
    .register_u64(XRegister::x14, Integer::Public(inter_addr))
    .address_range_u64(
        set0_addr,
        set0.into_iter()
            .map(|x| Integer::Secret(Share::Arithmetic(x)))
            .collect(),
    )?
    // 2 lt per set element cmp
    .n_and_triples(CMP_AND_TRIPLES * 2 * (set0_len + set1_len))
    .build()?;
```

Listing 16: PSI example for party 0

3.3.3 Program Execution

With a `Party` instance, a user can now pass a vector of `Instruction` enum variations to the `Party::execute()` method.

```
fn Party::execute(&mut self, program: Program) -> Result<()> {}
```

Listing 17: `Party execute()` function

`Instruction` implements the `FromStr` trait, so a program can be directly constructed by instantiating enum variants or parsed from a string of RISC-V assembly instructions. We

wrap the instruction vector in a `Program` unit struct that also implements `FromStr`. This allows users to conveniently parse a new-line separated string of RISC-V instructions.

```
let program = "psi: ...".parse()?;
party.execute(&program)?;
```

Listing 18: Program execution

3.3.4 Open Results

After the program terminates, the `register_u64()` and `address_u64()` getter methods of a `Party` can be used to open secret-shared values. These functions reveal the secret-shared values to both parties. The functions with the suffix `_for` also take an `id` as a parameter to specify which party the secret-shared value should be revealed to. In the latter case, the specified party gets `Option::Some(u64)`, i.e., the open value, and the other party gets `Option::None`. If the address or register contains a public value, it just gets returned.

```
// read public or reveal secret `Integer` in register
fn Party::register_u64(&mut self, register: XRegister) -> Result<u64> {}

// read public or partially reveal secret `Integer` in register
fn Party::register_for_u64(
    &mut self,
    register: XRegister,
    id: usize
) -> Result<Option<u64>> {}

// read public or reveal secret `Float` in register
fn Party::register_f64(&mut self, register: FRegister) -> Result<f64> {}

// read public or partially reveal secret `Float` in register
fn Party::register_for_f64(
    &mut self,
    register: FRegister,
    id: usize
) -> Result<Option<f64>> {}
```

Listing 19: Getter methods for accessing public or secret outputs.

```
// read public or reveal secret `Integer` at address
fn Party::address_u64(&mut self, address: Address) -> Result<u64> {}

// read public or partially reveal secret `Integer` at address
fn Party::address_for_u64(
    &mut self,
    address: Address,
    id: usize
) -> Result<Option<u64>> {}

// read public or reveal secret `Float` at address
fn Party::address_f64(&mut self, address: Address) -> Result<f64> {}

// read public or partially reveal secret `Float` at address
fn Party::address_for_f64(
    &mut self,
    address: Address,
    id: usize
) -> Result<Option<f64>> {}
```

Listing 20: Getter methods for accessing public or secret outputs.

```
// read public or reveal vector of secret `Integer` at address
fn Party::address_range_u64(
    &mut self,
    range: Range<Address>
) -> Result<Vec<f64>> {}

// read public or partially reveal vector of secret `Integer` at address
fn Party::address_range_for_u64(
    &mut self,
    range: Range<Address>,
    id: usize
) -> Result<Option<Vec<f64>>> {}

// read public or reveal vector of secret `Integer` at address
fn Party::address_range_f64(&mut self,
    range: Range<Address>
) -> Result<Vec<f64>> {}

// read public or partially reveal vector of secret `Float` at address
fn Party::address_range_for_f64(&mut self,
    range: Range<Address>,
    id: usize
) -> Result<Option<Vec<f64>>> {}
```

Listing 21: Getter methods for accessing public or secret outputs.

We now use these functions to open the result of the secret addition used above. Additionally, we also open a value from memory.

```
// get intersection length
let len = party.register_u64(XRegister::x10)?;
// open intersection address range
let intersection = party.address_range_u64(
    inter_addr..inter_addr + U64_BYTES * len
)?;
```

Listing 22: Revealing of outputs

3.4 Oblivious Transfer Implementation

The OT implementation is based on [Gal]. We use OT to generate Beaver triples and for share conversion.

3.4.1 Simplest OT

We use Simplest OT (see Section 2.2.1) to convert binary shares to arithmetic shares (see Section 2.1.3). The number of OTs in B2A conversion depends on bit width $l = 64$ of the shares. Therefore, Simplest OT is a better choice than the ALSZ OT extension protocl with 128 base-OTs.

Our implementation uses the KangarooTwelve [Ber+18] hash function. We chose K12 because, in our testing, it offered better performance for the small input sizes of 128 bits used in Simplest OT.

3.4.2 OT Extension

To generate large amounts of Beaver triples, we use the ALSZ (see Section 2.2.3) OT extension protocol. As shown Figure 14, OT extension is working as intended and clearly outperforms Simplest OT when it comes to large numbers of OTs. We use the implementation of the protocol described in Figure 2 in [Gal] with some small changes and simplifications. The ALSZ protocol uses matrix transpose operations that can be optimized by using Single Instruction Multiple Data (SIMD) instructions.

Chapter 4

Evaluation

In this chapter, we evaluate the usability and performance of riscMPC. We compare it to other MPC frameworks and show benchmark results.

4.1 Practical Examples

This section shows the riscMPC setup for different practical examples.

4.1.1 Private Set Intersection

In this section, we show a practical application of riscMPC to solve Private Set Intersection (PSI).

```
unsafe fn psi(set0: &[u64], set1: &[u64], inter: &mut [u64]) -> u64 {
    let (mut i, mut j, mut k) = (0, 0, 0);
    while i < set0.len() && j < set1.len() {
        // order of comparisons avoids computing equality check
        if set0[i] < set1[j] {
            i += 1;
        } else if set0[i] > set1[j] {
            j += 1;
        } else {
            *inter.get_unchecked_mut(k) = set0[i];
            i += 1;
            j += 1;
            k += 1;
        }
    }
    k as u64
}
```

Listing 23: Ordered set intersection

To simplify the functions, we use Rust's unsafe keyword and unchecked functions to eliminate some bounds checks on arrays. To compile a rust program to RISC-V assembly, we need to specify the target architecture when invoking the compiler (`rustc`). We do this by passing the command line argument `--target riscv64gc-unknown-linux-gnu`. Additionally, we also pass `-C opt-level=z` to enable all optimizations. We use the compiler explorer (https://godbolt.org) to compile the rust function shown in Listing 23. After compilation, we get the RISC-V assembly instructions, which can be seen in Listing 24.

```
1  psi:
2          li      a6, 0
3          li      a7, 0
4          li      t1, 0
5  .LBB0_1:
6          slli    t0, a7, 3
7          add     t0, t0, a2
8          slli    t2, t1, 3
9          add     t2, t2, a0
10 .LBB0_2:
11         sltu    t3, a7, a3
12         sltu    a5, t1, a1
13         and     a5, a5, t3
14         beqz    a5, .LBB0_8
15         ld      t3, 0(t2)
16         ld      a5, 0(t0)
17         bgeu    t3, a5, .LBB0_5
18         addi    t1, t1, 1
19         addi    t2, t2, 8
20         j       .LBB0_2
21 .LBB0_5:
22         bgeu    a5, t3, .LBB0_7
23         addi    a7, a7, 1
24         j       .LBB0_1
25 .LBB0_7:
26         addi    t1, t1, 1
27         addi    a7, a7, 1
28         slli    a5, a6, 3
29         add     a5, a5, a4
30         sd      t3, 0(a5)
31         addi    a6, a6, 1
32         j       .LBB0_1
33 .LBB0_8:
34         mv      a0, a6
35         ret
```

Listing 24: RISC-V assembly of ordered set intersection

Now, all that is left is to set up the two parties' registers and memory with the correct values. According to the RISC-V calling convention, registers x10-17 are used for function arguments, and register x10 will hold the return value. Both parties pass the same arguments (`set0_address, set0_len, set1_address, set1_len, intersection_address`) as `Integer::Public()` via registers `x10-14`. Finally, the individual elements in the sets are input as `Integer::Secret(Share::Arithmetic())` to tell riscMPC that it has to secret-share them using arithmetic secret sharing (see Listing 16).

After execution is finished, the parties read the return value from x10 and then open the memory range `intersection_address..intersection_address + 8 * intersection_len` (see Listing 22). The major advantage that riscMPC offers here is that effectively, all the parties had to do was specify which data is public and which is supposed to be secret-shared. No code has to be changed to transform the set intersection function to PSI. In riscMPC, execution is determined by data.

4.1.2 Mean of Salaries

In this section, we show a well-known use case of MPC that is used to secretly compute mean salary.

```
1  pub fn mean(salaries0: &[u64], salaries1: &[u64]) -> u64 {
2      let mut sum = 0;
3      for x in salaries0 {
4          sum += x;
5      }
6      for x in salaries1 {
7          sum += x;
8      }
9      sum / (salaries0.len() + salaries1.len()) as u64
10 }
```

Listing 25: Compute the mean of two groups of salaries

With the generated assembly, we instantiate the parties as follows:

```rust
let mut party = PartyBuilder::new(PARTY_0, ch)
    .register_u64(XRegister::x10, Integer::Public(sal0_addr))
    .register_u64(XRegister::x11, Integer::Public(sal0_len))
    .register_u64(XRegister::x12, Integer::Public(sal1_addr))
    .register_u64(XRegister::x13, Integer::Public(sal1_len))
    .address_range_u64(
        sal0_addr,
        sal0.iter().map(|x| Integer::Secret(Share::Arithmetic(*x)))
            .collect(),
    )?
    .build()?;
// execute mean function
party.execute(&program)?;
// open mean in return value register
let mean = party.register_u64(XRegister::x10)?;
```

Listing 26: Using riscMPC for secret computation of mean salary

Because the salaries are secret inputs, the sum is computed using secret addition (see Section 2.1). Finally, the mean is calculated by dividing the secret-shared sum by the public number of data points.

4.1.3 Ascon Hash

For some use cases, two parties have to compute a hash of secret-shared data. For this purpose, we can use riscMPC to compute the Ascon hash function in MPC.

To compute the Ascon round permutation in riscMPC, we simply compile the round function from Rust to RISC-V assembly and set the state to be binary secret-shared.

```rust
/// Ascon's round function
#[no_mangle]
pub fn round(x: [u64; 5], c: u64) -> [u64; 5] {
    // S-box layer
    let x0 = x[0] ^ x[4];
    let x2 = x[2] ^ x[1] ^ c; // with round constant
    let x4 = x[4] ^ x[3];

    let tx0 = x0 ^ (!x[1] & x2);
    let tx1 = x[1] ^ (!x2 & x[3]);
    let tx2 = x2 ^ (!x[3] & x4);
    let tx3 = x[3] ^ (!x4 & x0);
    let tx4 = x4 ^ (!x0 & x[1]);
    let tx1 = tx1 ^ tx0;
    let tx3 = tx3 ^ tx2;
    let tx0 = tx0 ^ tx4;

    // linear layer
    let x0 = tx0 ^ tx0.rotate_right(9);
    let x1 = tx1 ^ tx1.rotate_right(22);
    let x2 = tx2 ^ tx2.rotate_right(5);
    let x3 = tx3 ^ tx3.rotate_right(7);
    let x4 = tx4 ^ tx4.rotate_right(34);
    [
        tx0 ^ x0.rotate_right(19),
        tx1 ^ x1.rotate_right(39),
        !(tx2 ^ x2.rotate_right(1)),
        tx3 ^ x3.rotate_right(10),
        tx4 ^ x4.rotate_right(7),
    ]
}
```

Listing 27: Rust implementation of the Ascon round function

For one round, we need to perform multiple secret XORs, NOTs, and rotations. All of these can be done locally by each party. However, we also need to compute five secret ANDs per round. Therefore, Ascon with a secret state needs five binary Beaver triples per round.

Listing 28 shows a simple hash implementation that uses the Ascon state. The state contains an array of five `u64`s and is updated by applying the round function 12 times.

Since riscMPC is limited to 64-bit values, we cannot directly use bytes as input. Instead, our Ascon hash implementation expects the input to be an array of blocks (`&[u64]`) with the correct padding already applied.

```rust
#[no_mangle]
fn ascon_hash(input: &[u64], output: &mut [u64; 4]) {
    let mut state = State::new();
    for block in input {
        state.x[0] ^= block;
        state.permute_12();
    }

    output[0] = state.x[0];
    state.permute_12();
    output[1] = state.x[0];
    state.permute_12();
    output[2] = state.x[0];
    state.permute_12();
    output[3] = state.x[0];
}
```

Listing 28: Rust implementation of the Ascon hash function

With this setup, the two parties can concatenate their inputs and pass them to the hash function. No party learns anything about the input of the other except the length.

4.1.4 Ascon AEAD

Another use case is data encryption in MPC. The two parties already possess a pre-shared key that none of them knows in plain. Then, one party can provide a message to encrypt with the secret-shared key without the other party learning the contents of the message. To decrypt the ciphertext, both parties again need to cooperate and provide their shares of the encryption key.

```rust
#[no_mangle]
fn process_encrypt_inplace(state: &mut State, message: &mut [u64]) {
    for block in message {
        state.x[0] ^= *block;
        *block = state.x[0];
        state.permute_6();
    }

    state.x[0] ^= pad(0);
}

#[no_mangle]
fn process_final(state: &mut State, key: [u64; 2]) -> [u64; 2] {
    state.x[1] ^= key[0];
    state.x[2] ^= key[1];

    // permute_12 and apply key
    state.permute_12();
    state.x[3] ^= key[0];
    state.x[4] ^= key[1];

    // get tag from state
    [state.x[3], state.x[4]]
}

/// encrypt the message with the given key, nonce, ad and return tag
#[no_mangle]
fn encrypt_inplace(
    key: [u64; 2],
    nonce: [u64; 2],
    message: &mut [u64],
    associated_data: &[u64],
) -> [u64; 2] {
    let mut state = State::new(
        [IV, key[0], key[1], nonce[0], nonce[1]]
    );
    state.permute_12();
    state.x[3] ^= key[0];
    state.x[4] ^= key[1];

    process_associated_data(&mut state, associated_data);
    process_encrypt_inplace(&mut state, message);
    process_final(&mut state, key)
}
```

Listing 29: Rust implementation of Ascon AEAD

As shown in the `encrypt_inplace()` function (see Listing 29), the AEAD encryption function takes a 128-bit key and nonce, a mutable reference to the plaintext, and associated data. The fix-sized arrays for key and nonce are turned into simple pointers by the compiler, whereas plaintext and additional data turn into pointer and length parameters. With that information, we can set up the function arguments/memory and execute the program with the specified entry point (see Listing 30).

```
let mut party = PartyBuilder::new(PARTY_0, ch0)
  .register_u64(XRegister::x11, Integer::Public(key_addr))
  .register_u64(XRegister::x12, Integer::Public(nonce_addr))
  .register_u64(XRegister::x13, Integer::Public(pt_addr))
  .register_u64(XRegister::x14, Integer::Public(pt_len))
  .register_u64(XRegister::x15, Integer::Public(ad_addr))
  .register_u64(XRegister::x16, Integer::Public(ad_len))
  .address_range_shared_u64(
      key_addr,
      key.iter().map(|x| Integer::Secret(Share::Binary(*x))).collect(),
  )?
  .address_range_u64(
      nonce_addr,
      nonce.iter().map(|x| Integer::Secret(Share::Binary(*x)))
          .collect(),
  )?
  .n_and_triples(
      15 * 12 * 2 + 15 * 6 * pt_len +
      15 * 6 * (ad_len + 1) * (ad_len > 0) as u64
  )
  .build()?;
party.execute(
    &program.parse::<Program>()?.with_entry("encrypt_inplace")?
)?;
party.address_range_u64_for(
    pt_addr..pt_addr + pt_len * U64_BYTES, PARTY_1
)?;
party.address_range_u64_for(
    key_addr..key_addr + key_len * U64_BYTES, PARTY_1
)?;
```

Listing 30: riscMPC setup of Ascon AEAD for party 0

```
let mut party = PartyBuilder::new(PARTY_1, ch1)
  .register_u64(XRegister::x11, Integer::Public(key_addr))
  .register_u64(XRegister::x12, Integer::Public(nonce_addr))
  .register_u64(XRegister::x13, Integer::Public(pt_addr))
  .register_u64(XRegister::x14, Integer::Public(pt_len))
  .register_u64(XRegister::x15, Integer::Public(ad_addr))
  .register_u64(XRegister::x16, Integer::Public(ad_len))
  .address_range_shared_u64(
      key_addr,
      key.iter().map(|x| Integer::Secret(Share::Binary(*x))).collect(),
  )?
  .address_range_u64(
      pt_addr,
      pt.iter().map(|x| Integer::Secret(Share::Binary(*x))).collect(),
  )?
  .n_and_triples(
      15 * 12 * 2 + 15 * 6 * pt_len +
      15 * 6 * (ad_len + 1) * (ad_len > 0) as u64
  )
  .build()?;
party.execute(
    &program.parse::<Program>()?.with_entry("encrypt_inplace")?
)?;
let ct = party.address_range_u64_for(
    pt_addr..pt_addr + pt_len * U64_BYTES, PARTY_1
)?;
let tag = party.address_range_u64_for(
    key_addr..key_addr + key_len * U64_BYTES, PARTY_1
)?;
```

Listing 31: riscMPC setup of Ascon AEAD for party 1

The setup for both parties uses the same values for public argument registers, such as pointers and lengths, but one party inputs key plus nonce as binary secret-shared values, while the other inputs the plaintext. To input the data as binary shares, the `Share::Binary()` variant is used.

After the program was successfully executed, we use the partial open functions to only reveal the ciphertext and tag to party 1 (the party that provides the plaintext).

In the same way, both parties can also perform AEAD decryption. For this case, the setup changes slightly. Party 1 inputs the ciphertext and tag, and we choose the decryption function as entry point. To verify the decrpytion process, the function returns `true` if the passed tag was valid, or `false` if it was not. This boolean value is represented as a “0” or “1” public integer that can be checked by both parties.

4.1.5 Vectorized Multiplication

To improve multiplication performance, many MPC frameworks perform multiplication in batches. This can significantly increase performance because it reduces the time that is wasted waiting on shares sent by other parties.

Using vectorized multiplication in riscMPC is as simple as loading the data from memory into the special vector registers and executing the VMUL instruction. As of today, this RISC-V extension is still in a draft state and has not yet been included in most toolchains. Therefore, it may be necessary to use bare assembly instructions instead of relying on the compiler for this optimization.

```
1 vsetvli  x0,a3,e64
2 vle.v    v0,a0
3 vle.v    v1,a1
4 vmul.vv  v0,v0,v1
5 vse.v    v0,a2
```

Listing 32: RISC-V vectorized multiplication

4.1.6 MNIST Digit Recognition

Spearheaded by the launch of ChatGPT [Ach+23] in 2022, even more development and research is being done in ML fields. Because of that, PPML (see Section 2.7 for more details) has become more and more important for respecting users' data privacy. However, with ML already being as expensive as it is, the added development complexity, cost, and performance impact of PPML is very undesirable for corporations.

riscMPC makes it possible to design models without putting much thought into MPC. To demonstrate this, we show a simple implementation of the well-known MNIST digit recognition ML example. We use a basic Rust implementation of the model evaluation and use riscMPC to have one party input the trained model and the other a 28x28 pixel image of a handwritten digit (see Figure 13). In the end, the model-providing party will have learned nothing about the input image, and the party that provided the evaluation image learned nothing about the model other than its layer sizes.

```rust
fn sigmoid_approx(x: f64) -> f64 {
    if x < -2.5 {
        0.0
    } else if x > 2.5 {
        1.0
    } else {
        0.5 + 0.2 * x
    }
}

#[no_mangle]
fn evaluate(
    image: &[f64; 784],
    w1: &[[f64; 784]; 128],
    b1: &[f64; 128],
    w2: &[[f64; 128]; 10],
    b2: &[f64; 10],
    output: &mut [f64; 10],
) {
    // hidden layer computations
    let mut hidden_layer: [f64; 128] = [0.0; 128];
    for i in 0..128 {
        let mut sum = b1[i];
        for j in 0..784 {
            sum += w1[i][j] * image[j];
        }
        hidden_layer[i] = sigmoid_approx(sum);
    }

    // output layer computations
    for i in 0..10 {
        let mut sum = b2[i];
        for j in 0..128 {
            sum += w2[i][j] * hidden_layer[j];
        }
        output[i] = sigmoid_approx(sum);
    }
}
```

Listing 33: Rust implementation of a simple evaluation function for a ML model

Listing 33 shows the evaluate function and a linear piecewise approximation of the sigmoid function. The sigmoid function is a widely used activation function and is defined as follows:

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)} \tag{15}$$

However, since the sigmoid function contains a division by a secret value (x is the output of each layer and depends on the secret weights and inputs), we have to use an approximation without a division by a secret value.

$$\text{sigmoid\_approx}(x) = \begin{cases} 0 & \text{if } x < -2.5 \\ 1 & \text{if } x > 2.5 \\ 0.5 + 0.2 \cdot x & \text{else} \end{cases} \tag{16}$$

Compared to the actual sigmoid function, this approximation is possible to compute in MPC since it is just two comparisons and a multiplication with a public value. Since we already have to tolerate some errors due to fixed-point encoding, the additional error introduced by this approximation is acceptable.

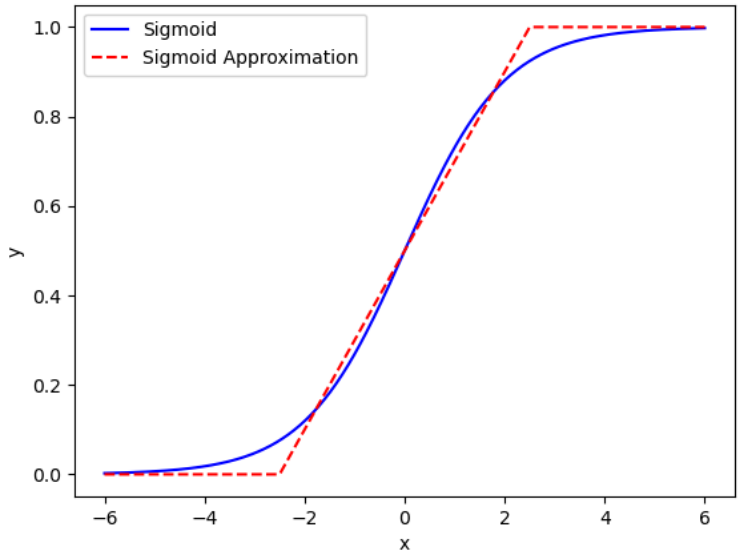


Figure 12: Linear piecewise approximation of the sigmoid function

Another commonly used activation fuction is the Rectified Linear Unit (ReLU) function.

$$\text{relu}(x) = \begin{cases} 0 \text{ if } x < 0 \\ x \text{ else} \end{cases} \tag{17}$$

The ReLU activation function is a lot simpler and can be easily used in riscMPC. Compiled to RISC-V assembly, it turns into `fmax.d fa0, fa0, fa1`, which can be computed using one

secret comparison (see Section 3.1.1). For our specific example of MNIST digit recognition, the sigmoid function approximation, although less performant, yielded better results.

With the Rust code compiled to assembly we can instantiate both parties and run the program. Party 0 (model provider) inputs the weights and biases (w1/b2, w2/b2), and party 1 a 784 (28x28) long array of normalized pixel values.

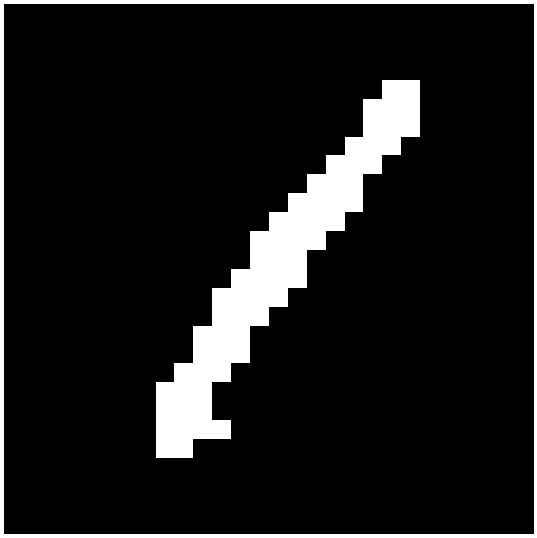


Figure 13: Input image

Listing 34 and Listing 35 show the riscMPC setup for the model-providing party and the image-providing party. These code examples also show the use of fixed-point encoding that is implemented using Rust traits for the basic `f64` and `u64` (embed/to_fixed_point respectively) types. The encoded values are input as `Float::Secret()` values, where instead of using the `Share` type, the contained `u64` always represents an arithmetic share of a fixed-point encoded float.

Similar to the AEAD example, the resulting predictions of the model are only revealed to the image-providing party. Because the assembly code contains multiple functions, we use `with_entry("evaluate")` to specify the program entry point.

```
let mut party = PartyBuilder::new(PARTY_0, ch0)
  .register_u64(XRegister::x10, Integer::Public(image_addr))
  .register_u64(XRegister::x11, Integer::Public(w1_addr))
  .register_u64(XRegister::x12, Integer::Public(b1_addr))
  .register_u64(XRegister::x13, Integer::Public(w2_addr))
  .register_u64(XRegister::x14, Integer::Public(b2_addr))
  .register_u64(XRegister::x15, Integer::Public(output_addr))
  .address_range_f64(
      w1_addr,
      w1.iter()
          .flatten()
          .map(|x| Float::Secret(x.embed().unwrap()))
          .collect(),
  )?
  .address_range_f64(
      b1_addr,
      b1.iter().map(|x| Float::Secret(x.embed().unwrap())).collect(),
  )?
  .address_range_f64(
      w2_addr,
      w2.iter()
          .flatten()
          .map(|x| Float::Secret(x.embed().unwrap()))
          .collect(),
  )?
  .address_range_f64(
      b2_addr,
      b2.iter().map(|x| Float::Secret(x.embed().unwrap())).collect(),
  )?
  .n_mul_triples(784 * 128 + 128 * 10)
  .n_and_triples(13 * (2 + 4) * (128 + 10))
  .build()?;
party.execute(&program.parse::<Program>()?.with_entry("evaluate")?)?;
party.address_range_f64_for(
  output_addr..output_addr + U64_BYTES * 10, PARTY_1
)?;
```

Listing 34: riscMPC MNIST digit recognition setup for party 0

```
let mut party = PartyBuilder::new(PARTY_1, ch1)
  .register_u64(XRegister::x10, Integer::Public(image_addr))
  .register_u64(XRegister::x11, Integer::Public(w1_addr))
  .register_u64(XRegister::x12, Integer::Public(b1_addr))
  .register_u64(XRegister::x13, Integer::Public(w2_addr))
  .register_u64(XRegister::x14, Integer::Public(b2_addr))
  .register_u64(XRegister::x15, Integer::Public(output_addr))
  .address_range_f64(
      image_addr,
      image.iter().map(|x| Float::Secret(x.embed().unwrap())).collect(),
  )?
  .n_mul_triples(784 * 128 + 128 * 10)
  .n_and_triples(13 * (2 + 4) * (128 + 10))
  .build()?;
party.execute(&program.parse::<Program>()?.with_entry("evaluate")?)?;
let predictions = party.address_range_f64_for(
  output_addr..output_addr + U64_BYTES * 10, PARTY_1
)?;
```

Listing 35: riscMPC MNIST digit recognition setup for party 1

Output for party 1, that provided the image to be labeled:

```
predictions = [0.0, 0.945, 0.112, 0.106, 0.0, 0.0, 0.0, 0.043, 0.060, 0.168]
```

The model labels this image correctly as a “1” with a probability of 94.5%.

4.2 Performance Benchmarks

We benchmarked core components of riscMPC and a concrete example of PSI. The benchmarks were run with both parties on the same PC with a i7-6700K 4C/8T @ 4.6GHz and 32GB RAM. Since both parties run on the same machine and in the same local network, latency is lower than in real-world applications. Currently, riscMPC only runs single-threaded.

4.2.1 Oblivious Transfer

During the online phase, riscMPC uses Chou Orlandi OT (see Section 2.2.1) to convert secret shares from arithmetic to binary (see Section 2.1.3). In the setup phase, we can make use of the ALSZ OT (see Section 2.2.3) extension protocol to generate Beaver triples. Because we may need to perform a large but known amount of OTs, we use ALSZ to extend a small number of base-OTs, significantly improving the triple generation compared to standard OT.

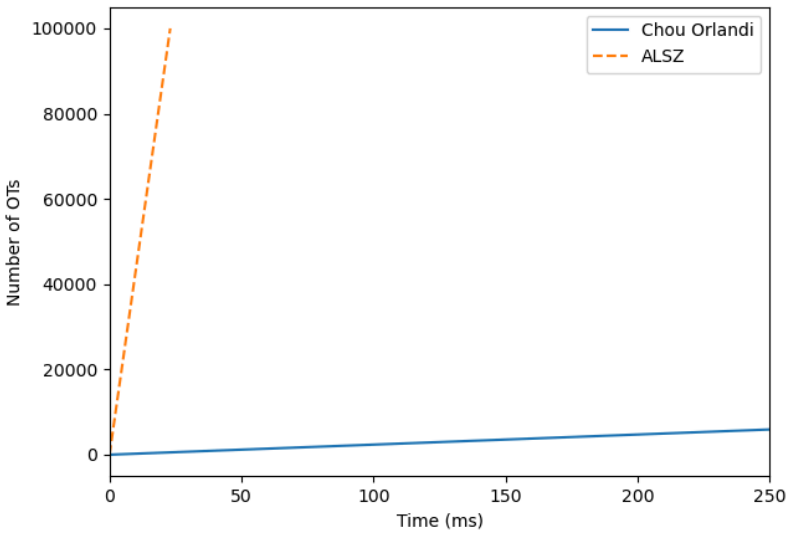


Figure 14: OT performance of Chou Orlandi vs. ALSZ

Figure 14 shows the performance difference between Chou Orlandi and ALSZ when performing a large number of OTs at once. Using Chou Orlandi OT it takes about 4.2s to perform 10^5 OTs. With ALSZ the same amount of OTs only takes 23ms.

4.2.2 Beaver Triple Generation

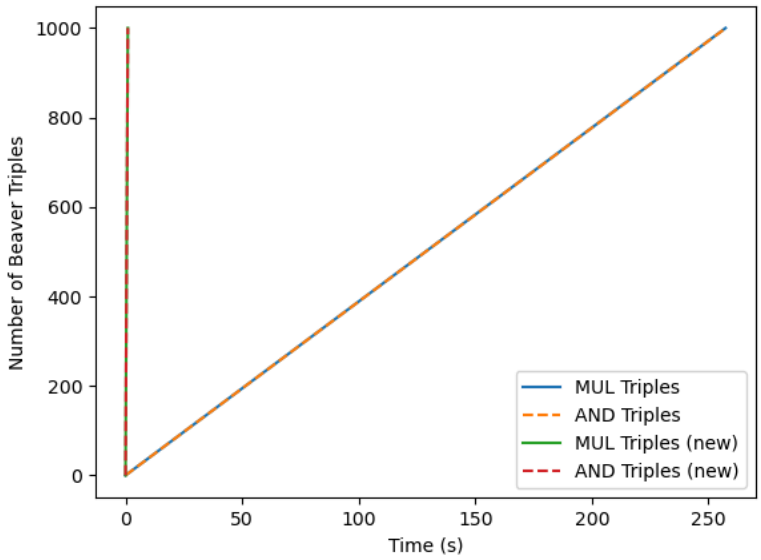


Figure 15: Beaver triple generation using OT. MUL and AND triple generation both use two rounds of 64 (once per bit) 1-out-of-2 OTs, thus their runtime is almost identical.

For the proof-of-concept implementation of riscMPC, we mainly focused on the performance of the online phase and the general ergonomics of the framework. As can be seen in Figure 15 above, it takes about 250 seconds to generate 1000 multiplication/AND Beaver triples. Using simple Naor-Pinkas OT without OT extensions [NP99], we create triples at a rate of 3.89 triples per second.

The updated and improved Beaver triple generation uses OT extension (see Section 2.2.2). With this new implementation, it only takes about 1 second to generate the same 1000 multiplication/AND triples as the basic version. This significant improvement in performance makes riscMPC a lot more feasible for actual use.

4.2.3 Share Conversions

During runtime, we often need to convert between arithmetic and binary shares (see Section 3.1.1 for more details on which instructions make use of share conversions). If we want to perform arithmetic (e.g., ADD, MUL, …) and binary (e.g., XOR, AND, …) on the same secret-share values, we choose one share type at the start and always convert to the other when necessary.

Arithemitc to Binary. The A2B conversion uses 13 binary AND triples that are generated in the setup phase. By offloading this cost to the setup phase, we can achieve

great online performance. Figure 16 shows the time distribution of 1000 conversions. A single A2B conversion takes about $166.53 \pm 14.06 \mu s$.

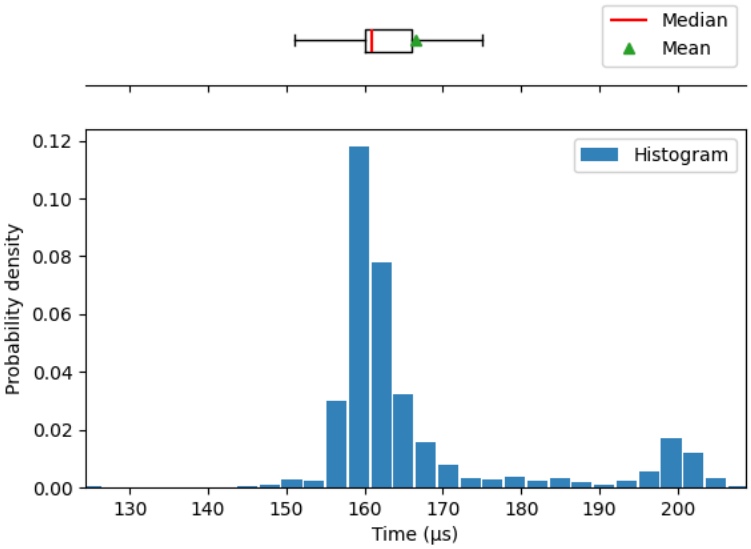


Figure 16: Arithmetic to binary share conversion performance

Binary to Arithmetic. The B2A conversion uses OT per bit of our 64-bit values. Compared to B2A conversion, this takes considerably more time, but no Beaver Triples are required. Figure 17 shows the time distribution of 1000 conversions. A single B2A conversion takes about $7.67 \pm 0.87 ms$.

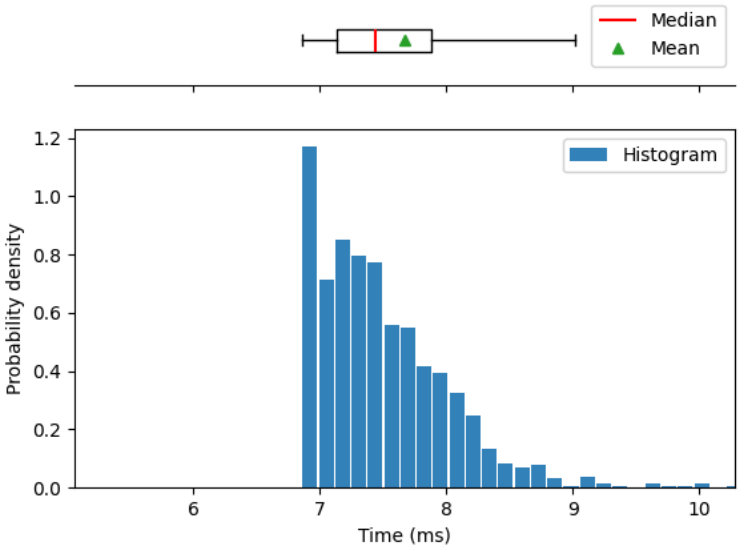


Figure 17: Binary to arithmetic share conversion performance

4.2.4 Private Set Intersection

There are many different specialized approaches to PSI. ABY's authors provide multiple implementations, which can be seen in the examples directory. Although very performant (see Figure 18), the provided examples require considerable effort and knowledge. Figure 18 shows that riscMPC can offer similar performance for set sizes < 2048 and acceptable performance for set sizes < 8192. All the while, riscMPC provides a far less complex user experience that is well-suited for rapid development.

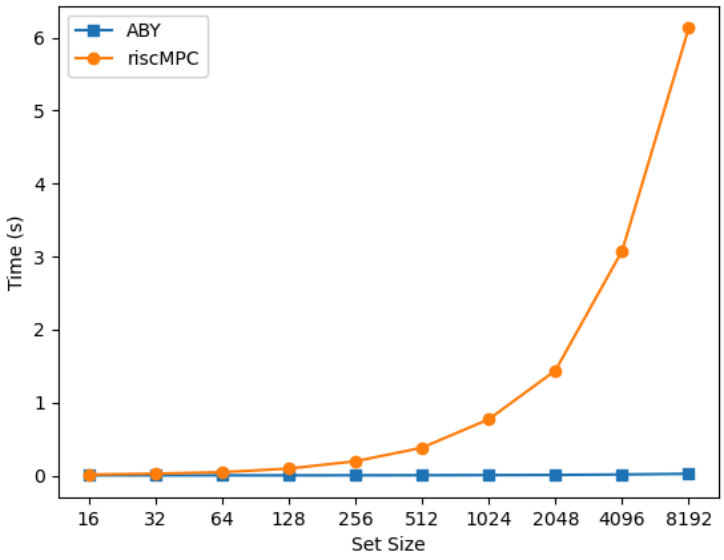


Figure 18: Comparison of PSI online phase runtime with different set sizes. ABY [DSZ15] vs. riscMPC (x-axis is scaled logarithmically)

Figure 18 above compares the PSI online phase of riscMPC and ABY. The performance differences can be attributed partly to the more specialized protocols and the use of multiple threads in ABY. The ABY example sorts the two input sets before computing the PSI, so we test riscMPC with a PSI program (see Listing 23) that assumes ordered sets and needs $\mathcal{O}(n+k)$ time (where n and k are the parties' set sizes).

	Setup (ms)	**Online (ms)**	**Total (ms)**
set size = 10			
ABY [DSZ15]	381.13	2.76	383.89
riscMPC	538.50	9.50	548.00
set size = 100			
ABY [DSZ15]	437.69	3.52	441.23
riscMPC	$5.39 \cdot 10^3$	83.00	$\approx 5.40 \cdot 10^3$
set size = 1000			
ABY [DSZ15]	833.19	7.26	840.45
riscMPC	$57.52 \cdot 10^3$	763.00	$\approx 58.20 \cdot 10^3$

Table 3: Execution times of setup and online phase, comparing PSI implemented with ABY vs riscMPC.

Table 3 shows the setup and online phase durations of ABY and riscMPC. The runtime of the setup and online phase depends on the set sizes and whether the sets are ordered or not. For ordered PSI, we need to compute the comparison chain `if a < b {...} else if a > b {...} else {...}` at most $n + k$ times. By doing the comparisons in this order, we avoid computing $a == b$ because we get it for free in the else branch. Additionally, if the first comparison is evaluated as true, we do not have to check the second comparison. For this benchmark, we chose both sets so that half of the elements overlap. This setup demonstrates a good average-case performance. In the worst case, we need $<$ and $>$, and thus $13 \cdot 2 \cdot \mathcal{O}(n + k)$ binary AND triples. Given the current basic Beaver triple generation in riscMPC, the setup phase performs very poorly. However, riscMPC is designed to be extensible and modular. Therefore, this part can easily be improved in the future.

4.2.5 Ascon Hash

In each round of Ascon, we need to perform 5 binary ANDs and 10 binary ORs (see Listing 27). For each AND/OR we need one binary Beaver triple. As shown in Listing 28, Ascon hash uses the round function 12 times (`State::permute_12()`) per input block (block size = 8 bytes) and an additional 3 times to extract the hash.

We calculate the total number of Beaver triples necessary as $15 \cdot 12 \cdot (3 + n)$. Using 100 input blocks, this results in 18,540 triples. After generating the Beaver triples in the offline phase, the online phase runtime mostly depends on the number of input blocks. With the exception of ANDs and ORs, both parties perform all other instructions locally. As can be seen in Figure 19, the runtime depends linearly on the number of blocks. It takes about 200ms to compute the hash of 100 secret input blocks.

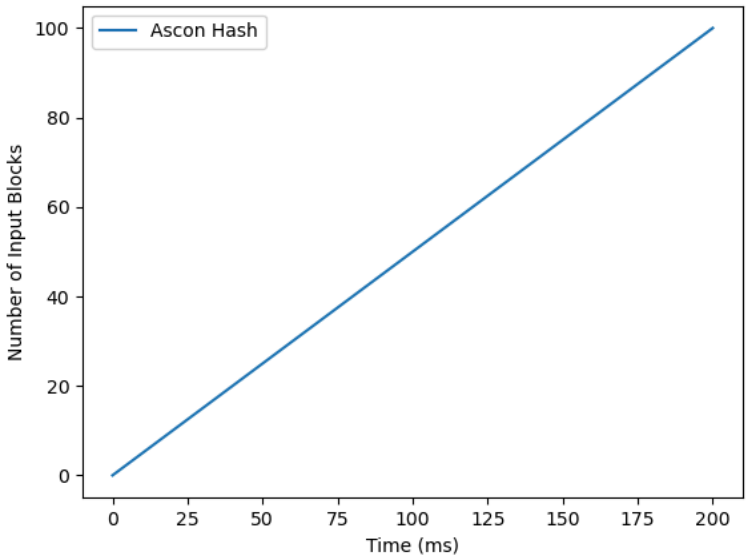


Figure 19: Online phase performance of Ascon hash computation on secret inputs

4.2.6 Ascon AEAD

As shown in Listing 29, Ascon AEAD encryption uses `State::permute_12()` once at the start and again in `process_final()`. To process additional, `State::permute_6()` is used $n + 1$ times if the associated data is not empty. Otherwise, the round function is not used at all. Finally, to process the plaintext input blocks, `State::permute_6()` is used once per block. Putting all of that together, we compute the total number of Beaver triples as $15 \cdot 12 \cdot 2 + 15 \cdot 6 \cdot n_{\text{pt}} + 15 \cdot 6 \cdot (n_{\text{ad}} + 1) \cdot (n_{\text{ad}} > 0)$. Using 100 plaintext blocks and no associated data, this results in 9,360 triples.

As can be seen in Figure 20, the runtime without associated data linearly depends on the number of pt blocks. With associated data, the runtime would depend linearly on $n_{\text{pt}} + n_{\text{ad}}$. For 100 blocks of plaintext and no AD, the encryption takes about 95ms.

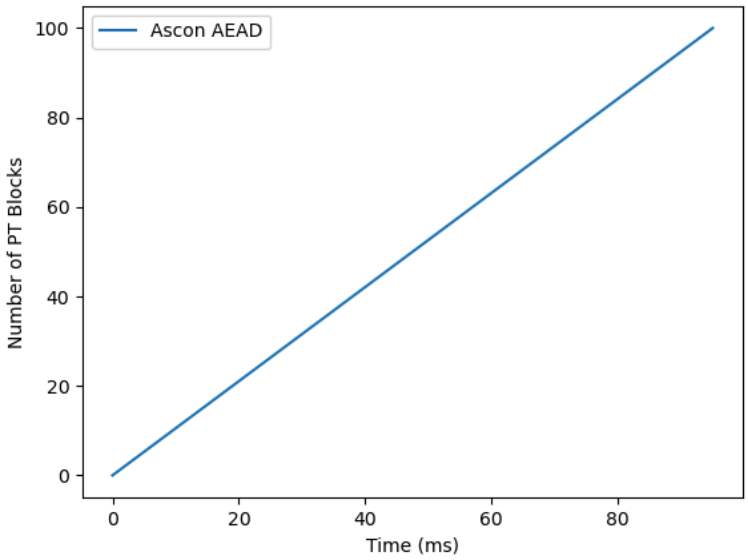


Figure 20: Online phase performance of Ascon AEAD computation on secret inputs

4.2.7 Vectorized Multiplication

By using the VMUL instruction instead of repeated MUL instructions, we can batch the secret multiplication to significantly increase performance. Even on a local network, we observed a big performance improvement. With the standard individual multiplication, it takes about 100ms to perform 10000 secret multiplication (with Beaver triples generated in the offline phase). If we use the VMUL instruction instead, the revealing of the intermediary values d and e is batched, and the same number of multiplications now only takes 4ms. This is a speedup of about 25×. In a real use case, where communication has much higher latency than a local network, the performance difference will be even greater.

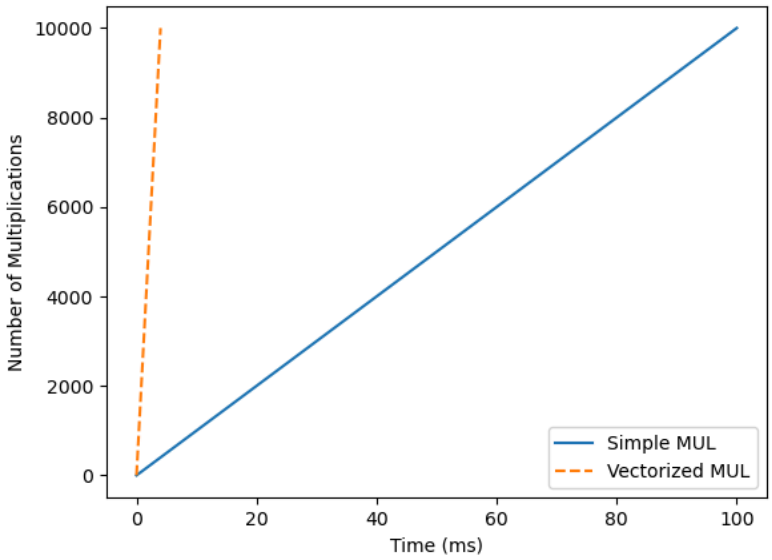


Figure 21: Comparison of simple secret multiplication vs vectorized multiplication

4.2.8 MNIST Digit Recognition

We tested the evaluation of a pre-trained model where the input image and the model are provided as secret inputs by different parties. With the standard 28x28 input images and our chosen layers (784 input, 128 hidden, 10 output neurons), evaluation takes about 2.86s. For the evaluation, we need a very large amount of multiplication triples. The exact amount can be computed with $784 \cdot 128 + 128 \cdot 10 = 101,632$, as it depends on the layer sizes. Additionally, we need binary triples for the branches in the sigmoid approximation. Here, we need at most $13 \cdot (2 + 4) \cdot (128 + 10) = 10,764$. 2 for both float comparisons and 4 for branch equal instruction that, together with the float comparisons, form the if statements.

Assuming we would use inline assembly or custom build tools, we could greatly increase the performance by using vectorized multiplication. With division by a secret value, we could also replace the linear approximation of the sigmoid function with the actual sigmoid function and just approximate the exponential function. Depending on how the division by a secret value is implemented, this could further increase performance.

4.2.9 Network Latency

Given the potentially large amount of communication between parties, network latency has a huge impact on MPC. In this section, we benchmark the impact of different amounts of latency on repeated multiplication. To artificially add latency to the connection in the local network, we used the following command: `sudo tc qdisc add dev lo root netem delay XXms`. As shown in Figure 22, even a small amount of latency results in a large drop in online phase performance. Without any artificial latency, 100 secret by secret multiplications take about 4ms. With just 10ms of latency, the same amount of multiplications takes about 4.49s (an increase of $\approx 1000\times$). Adding more latency linearly increases the drop in performance. With vectorized multiplication, the impact of network latency could be mitigated by a large amount.

Figure 22 and Figure 23 show the significant improvement that vectorized multiplications bring, especially in higher latency cases. Even with a latency of 50ms, vectorized multiplication far outperforms standard multiplicans with just 10ms of latency.

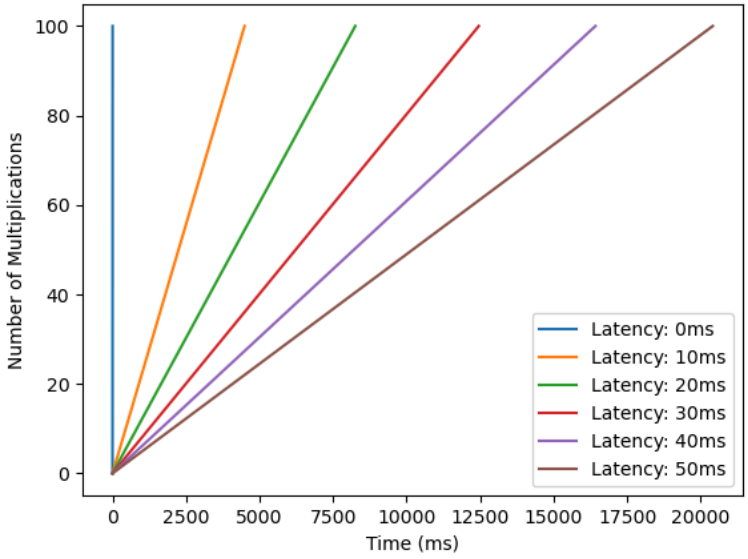


Figure 22: Comparison of latency values for online phase of repeated multiplications

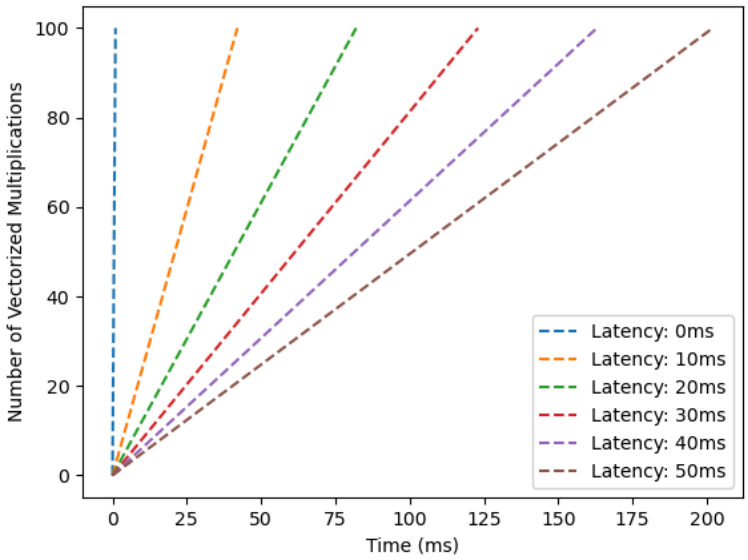


Figure 23: Comparison of latency values for online phase of vectorized multiplications

4.2.10 Network Communication

In this section, we measure the network communication for different operations. The measurements do not take TCP packet headers and other additional data from the network implementation into account. Instead, we measure data going through our Rust `TcpChannel` implementation.

Operation	#	**Setup**	**Online**
		Comm (KB)	Comm (KB)
MUL	100	725.07	6.32
A2B	100	9,125.07	81.32
B2A	100	–	475.07

Table 4: Network traffic measurements in kilobytes (KB) for offline and online phase

Table 4 shows that for multiplications and A2B conversions, a lot of communication is offloaded to the setup phase. By generating Beaver triples in the offline phase, the only communication in the online phase is the revealing of d and e (see Section 2.1.1). For B2A conversions, no setup phase is required, and all OTs happen in the online phase (see Section 2.1.3).

Chapter 5

Conclusion

In this thesis, we presented riscMPC, a new, data-driven approach to MPC frameworks using RISC-V assembly. We developed a proof-of-concept implementation for two-party semi-honest MPC and compared its usability and performance to current state-of-the-art MPC frameworks.

We showed in Chapter 4 that for PSI, riscMPC can be a suitable alternative to current frameworks such as ABY [DSZ15]. Performance stays within the same order of magnitude for set sizes of up to 1024. Set sizes at or below 8912 result in online phase run times that are still in the single-digit seconds. Additionally, we presented examples of using the Ascon hash and AEAD functions in riscMPC. Finally, we showed that with fixed-point encoding, riscMPC can also be used to perform ML tasks, such as handwritten digit recognition, without revealing the model or inputs.

Because riscMPC interprets RISC-V assembly, specialized implementations, such as ABY's PSI examples, are not feasible. But our general-purpose riscMPC framework considerably reduces the implementation overhead and complexity that might hold MPC back.

5.1 Future Work

To stay within the scope of this thesis, we had to make some trade-offs. Because we chose to implement riscMPC using Rust, we had to write most of it from scratch.

Program Limitations: Currently, system calls are not supported. This means that access to the file system or network is not possible. Additionally, calls into dynamically linked libraries are not supported. This also means that we cannot use heap allocations. The implications of these limitations remain to be seen, but during our research, most use cases were not affected. Some of these problems could be fixed by choosing one compiled language, such as Rust, for first-class support and handling its syscalls and calls to dynamically linked libraries in riscMPC itself.

Support for more types: Currently riscMPC only supports 64-bit integers (`u64`) and 64-bit floats (`f64`). Having support for other types like 8-bit, 16-bit, and 32-bit values would greatly increase compatibility. This could be done in an imperfect way where all the types are secret-shared in their own ring respective to their size. This allows computation between identical types but not with mixed types. This limits a lot of use cases that have different types, but there might be a better solution for this problem.

Caching Share Conversions: For some programs, a lot of time can be spent on share conversions. While there is not much we can do about that for secret-shared values that are mutated, values that either rarly or never change could implement software caches. The goal is to keep track of both arithmetic and binary shares of such values and use them where needed. When a value is mutated, it would be evicted from the cache.

Intermediary Share Opening: In the future, a new RISC-V extension could be introduced to support the revealing of shares with assembly instructions. Such instructions could look something like this: `open rd, rs1, imm` where the party's share is in register rs1, the immediate would indicate which party the value should be revealed to, and rd is the register to store the revealed value. This would solve the issue that riscMPC can only reveal secret-shared values after execution.

Branch Prediction: Branch prediction for branches with secret comparisons could be implemented to increase performance. This would be especially useful for branches with expensive multiplications/and instructions. The parties could perform the mul/and instruction and the comparison in parallel and discard the result if the branch prediction turns out to be wrong.

Bibliography

[CO15] T. Chou and C. Orlandi, "The simplest protocol for oblivious transfer," in *Progress in Cryptology–LATINCRYPT 2015: 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings 4*, 2015, pp. 40–58.

[Ash+13] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner, "More efficient oblivious transfer and extensions for faster secure computation," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 535–548.

[WK17] A. Waterman and R.-V. F. Krste Asanović, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2." [Online]. Available: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[Dob+21] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "Ascon v1. 2: Lightweight authenticated encryption and hashing," *Journal of Cryptology*, vol. 34, pp. 1–42, 2021.

[Den12] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[EC23] European Parliament and Council of the European Union, "Regulation (EU) 2016/679 of the European Parliament and of the Council." Accessed: Jul. 23, 2023. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679

[Sha79] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[Eva+18] D. Evans, V. Kolesnikov, M. Rosulek, and others, "A pragmatic introduction to secure multi-party computation," *Foundations and Trends® in Privacy and Security*, vol. 2, no. 2–3, pp. 70–246, 2018.

[Kel20] M. Keller, "MP-SPDZ: A versatile framework for multi-party computation," in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 2020, pp. 1575–1590.

[Yao82] A. C. Yao, "Protocols for secure computations," in *23rd annual symposium on foundations of computer science (sfcs 1982)*, 1982, pp. 160–164.

[DSZ15] D. Demmler, T. Schneider, and M. Zohner, "ABY-A framework for efficient mixed-protocol secure two-party computation.," in *NDSS*, 2015.

[GMW19] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game, or a completeness theorem for protocols with honest majority," *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*. pp. 307–328, 2019.

[MR18] P. Mohassel and P. Rindal, "ABY3: A Mixed Protocol Framework for Machine Learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, in CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 35–52. doi: 10.1145/3243734.3243760.

[Pat+21] A. Patra, T. Schneider, A. Suresh, and H. Yalame, "{ABY2. 0}: Improved {Mixed-Protocol} Secure {Two-Party} Computation", in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2165–2182.

[Bea92] D. Beaver, "Efficient multiparty protocols using circuit randomization," in *Advances in Cryptology—CRYPTO'91: Proceedings 11*, 1992, pp. 420–432.

[Kno+21] B. Knott, S. Venkataraman, A. Hannun, S. Sengupta, M. Ibrahim, and L. van der Maaten, "Crypten: Secure multi-party computation meets machine learning," *Advances in Neural Information Processing Systems*, vol. 34, pp. 4961–4973, 2021.

[Bay+21] A. Bay, Z. Erkin, J.-H. Hoepman, S. Samardjiska, and J. Vos, "Practical multi-party private set intersection protocols," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 1–15, 2021.

[Dem+18] D. Demmler, P. Rindal, M. Rosulek, and N. Trieu, "PIR-PSI: scaling private contact discovery," *Cryptology ePrint Archive*, 2018.

[Lap+18] A. Lapets *et al.*, "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*, in COMPASS '18. Menlo Park and San Jose, CA, USA: Association for Computing Machinery, 2018. doi: 10.1145/3209811.3212701.

[NP99] M. Naor and B. Pinkas, "Oblivious transfer and polynomial evaluation," in *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, 1999, pp. 245–254.

[Bea96] D. Beaver, "Correlated pseudorandomness and the complexity of private computations," in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, pp. 479–488.

[Ish+03] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank, "Extending oblivious transfers efficiently," in *Annual International Cryptology Conference*, 2003, pp. 145–161.

[Jam95] T. Jamil, "Risc versus cisc," *Ieee Potentials*, vol. 14, no. 3, pp. 13–16, 1995.

[MAL23] D. Morales, I. Agudo, and J. Lopez, "Private set intersection: A systematic literature review," *Computer Science Review*, vol. 49, p. 100567–100568, 2023, doi: https://doi.org/10.1016/j.cosrev.2023.100567.

[Pop+09] M.-C. Popescu, V. E. Balas, L. Perescu-Popescu, and N. Mastorakis, "Multilayer perceptron and neural networks," *WSEAS Transactions on Circuits and Systems*, vol. 8, no. 7, pp. 579–588, 2009.

[MZ17] P. Mohassel and Y. Zhang, "Secureml: A system for scalable privacy-preserving machine learning ," in *2017 IEEE symposium on security and privacy (SP)*, 2017, pp. 19–38.

[XBJ21] R. Xu, N. Baracaldo, and J. Joshi, "Privacy-preserving machine learning: Methods, challenges and directions," *arXiv preprint arXiv:2108.04417*, 2021.

[SC23] K. Steve and N. Carol, "The Rust Programming Language." [Online]. Available: https://doc.rust-lang.org/book/

[Gal] I. Galois, "swanky: A suite of rust libraries for secure computation." [Online]. Available: https://github.com/GaloisInc/swanky

[Ber+18] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, R. Van Keer, and B. Viguier, "K angaroo T welve: Fast Hashing Based on Keccak-p KECCAK-p," in *Applied Cryptography and Network Security: 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings 16*, 2018, pp. 400–418.

[Ach+23] J. Achiam *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.