Experiments for Decentralized Iris Code Membership Protocols using MPC

TACEO

January 2024

1 Introduction

The goal of the project is to construct an efficient MPC protocol computing an iris code membership check on an MPC-shared databases. Concretely, a database containing known iris code, each consisting of multiple thousand bits (approximately 12800 at the time of writing), is shared amongst three computing parties. Furthermore, each iris code is connected to a mask of the same size, which is known by all parties.

Another party, dubbed the orb, is able to create a iris code (and mask) and sends it in a shared form to the computing parties. The parties then check if the iris code is already present in the database by computing a similarity check between the iris code and each element in the database. This check is implemented as comparing the hamming distance to a given threshold. Finally, all comparisons are aggregated into one output bit, which is 0 if the new iris code is not found in the database, 1 otherwise.

We depict the desired matching algorithm (without MPC) in Algorithm 1.

Algorithm 1 The Iris Code Membership Protocol without MPC. It checks, whether the iris code $\vec{c}$, und the mask $\vec{m}$ is similar to any iris in the database $C_{\texttt{DB}}$ under masks $M_{\texttt{DB}}$. l is the size of the iris codes in bits, s is the number of codes in the database.

```
Input: $\vec{c}, \vec{m} \in \mathbb{F}_2^l, C_{\texttt{DB}}, M_{\texttt{DB}} \in \mathbb{F}_2^{s \times l}$
Output: true if $\vec{c}$ is similar to an entry in the DB, false otherwise.
  for $i$ in $0..s$ do
    $\vec{m}' \leftarrow \vec{m} \wedge M_{\texttt{DB}}[i]$                ▷ Combine masks.
    ml ← CountOnes($\vec{m}'$)
    if ml < MASK_THRESHOLD then
      return abort                          ▷ Not enough iris bits present...
    end if
    $\vec{c}' \leftarrow (\vec{c} \oplus C_{\texttt{DB}}[i]) \wedge \vec{m}'$
    hd ← CountOnes($\vec{c}'$)                 ▷ Hamming distance.
    if hd < MATCH_RATIO · ml then
      return true                                   ▷ Iris is similar!
    end if
  end for
  return false                                    ▷ No match found.
```

2 Iris Code Membership Protocol

In this section, we summarize the core building blocks of the protocol and the challenges involved in realizing them in MPC.

2.1 Core Building Block: Efficient Dot Product

One of the core operations of the Iris Code Membership protocol is the calculation of the hamming distance of the two binary iris code vectors. The main issue with this computation in MPC is that the hamming distance requires an XOR operation, i.e., MPC over $\mathbb{F}_2$, followed by counting the ones to get a result in $\mathbb{Z}_{2^k}$. Finally, after the comparison with the threshold, the result will be a boolean value again.

Calculating the hamming weight of a binary vector in MPC in a trivial fashion would thus require communication that is linear in the length of the vector. This comes from either translating the shared bits to arithmetic shares which requires communication for each bit, or from counting the ones in a binary circuit, which also requires communication linear in the bitsize. However, if we have the precondition that the input vectors are already shared over a larger ring $\mathbb{Z}_{2^k}$ instead of $\mathbb{F}_2$, we can employ the following strategy:

$$\mathsf{hd}(\vec{a}, \vec{b}) = \sum_i a_i - 2 \sum_i a_i \cdot b_i + \sum_i b_i \,.$$

This reduces the calculation of the hamming distance to two sums (which can be computed without party interaction in most MPC protocols), as well as a dot-product of two vectors. The calculation of this dot-product dominates the complexity of the hamming-distance operation, and we therefore want to use MPC protocols that support a very efficient dot-product operation. In general, protocols that have a honest-majority security assumption can support dot-products that require communication which is independent of the size of the vectors, which is optimal. The protocols we discuss in Section 3 all support such efficient dot-products.

2.2 Core Building Block: MSB Extraction

Since a comparison $a < b$ is equal to an MSB-extraction $\mathtt{msb}(a - b)$ (if the sizes of a, b are chosen to not produce an overflow), a core building block is an MSB-extraction protocol. This subprotocol requires to change the sharing type of additive shares over $\mathbb{Z}_{2^k}$ to boolean shares over $\mathbb{F}_2^k$. In the targeted honest-majority MPC protocols (Section 3) this is usually done by interpreting each additive share $x_i \in \mathbb{Z}_{2^k}$ of $[x] = (x_1, x_2, x_3)$ as trivial boolean shares. In other words, the shares are translated to $[x_1]^B = (x_1, 0, 0)$, $[x_2]^B = (0, x_2, 0)$, and $[x_3]^B = (0, 0, x_3)$. Finally, these three shares are combined to the target share by computing $[x_1]^B + [x_2]^B + [x_3]^B$ using a binary adder circuit in MPC. Thereby, one can use the following optimization: If one first computes a full adder $\mathtt{FA}([x_1[i]]^B, [x_2[i]]^B, [x_3[i]]^B) = ([c[i]]^B, [s[i]]^B)$ for each bit i (k And gates and depth of only one), the final result can be obtained by adding $2 \cdot [c]^B + [s]^B$. This drastically reduces the total number of communication rounds required for the binary circuit. Finally, since the use case operates on a huge amount of data, k being small, and reducing communication complexity being the primary optimization point, we opt for using a ripple-carry-adder instead of a depth-optimized carry-lookahead adder for computing $2 \cdot [c]^B + [s]^B$.

2.3 Core Building Block: OR-tree

After all hamming distances are compared to the thresholds, one needs to accumulate all resulting bits. Since the result should be zero iff all bits are zero, the accumulation is equivalent to many boolean OR operations. In MPC, one can compute an $x \vee y = x \oplus y \oplus (x \wedge y)$. To reduce the number of communication rounds, we evaluate the accumulation of all bits in a binary tree.

3 MPC Protocols

To efficiently evaluate the use case we opt for 3-party, ring-based, honest majority protocols, which use replicated secret sharing. This choice is crucial, since they provide a dot-product subprotocol with communication being independent to the size of the vectors, they do not operate over prime fields hence computational overhead is minimized, and they provide efficient protocols to switch between arithmetic sharings over $\mathbb{Z}_{2^k}$ and binary sharings over $\mathbb{F}_2^k$. The protocols we discuss achieve either semi-honest security, or malicious security with abort against one corrupted party.

3.1 Assumptions

To achieve a sufficiently fast performance for this specific use case, we make two assumptions:

- **Honest majority** of parties, i.e., only t of n parties are allowed to collude or deviate of the protocol, with $t < n/2$. In our concrete case (we are working with 3 parties), the protocols are secure if at most one party is cheating and no two parties are colluding.
- The iris **masks** are allowed to be **known** to all of the involved MPC parties. This holds for the masks of the iris codes which area already present in the database, as well as for the mask for the new iris code sent by the orb.

Note that both assumptions are essential to constructing an MPC protocol where the communication is **independent of the length of the iris code**, saving a factor of 12800 in many parts of the communication.

3.2 Semi-honest ABY3: Our baseline protocol

The ABY3 [MR18] protocol is the baseline for many 3-party, replicated-ring-based MPC protocols and its semi-honest variant is amongst the most efficient semi-honest MPC protocols to date. Thus, we will implement and use it for the semi-honest version of the use case.

In ABY3, arithmetic values $x \in \mathbb{Z}_{2^k}$ are shared additively, such that $\texttt{Share}(x) = [x] = (x_1, x_2, x_3)$ and $x = \sum x_i$. Then each party i gets as its share the values (x_i, x_{i-1}) (where the indices are taken $\mod 3$). Since additive sharing is used, linear operations, such as addition, subtraction, and multiplications with constants, can be performed on the shares without the parties having to communicate with each other.

Multiplications $[z] = [x] \cdot [y]$, on the other hand, can not be computed purely without communication. However, since each party has 2 additive shares, one can compute an additive share of the result without communication. Namely

$$z_i = x_i \cdot y_i + x_{i-1} \cdot y_i + x_i \cdot y_{i-1} + r_i,$$

where r_i is a freshly generated random share of 0 required for re-randomization which can be produced without communication in ABY3. To translate the additive share z_i to a replicated share (z_i, z_{i-1}) it suffices that party i sends its share to party $i + 1$.

When computing a dot-product $[z] = \langle[\vec{x}], [\vec{y}]\rangle = \sum[x_i] \cdot [y_i]$, one does not have to reshare after each individual multiplication $[x_i] \cdot [y_i]$, but can perform the summation on additive shares and reshare just the result. Consequently, a dot product requires the same amount of communication as a multiplication in ABY3, i.e., it is independent to the length of the vectors $|[\vec{x}]| = |[\vec{y}]|$.

3.3 Malicious Security

Many publications following ABY3's blueprint have been proposed in the literatures, which mainly differ on how to get malicious security in different security models. We discuss the main methods in this section, limiting ourselves to the 3-party setting and malicious security with abort.

3.3.1 Joint Message Passing

Since ABY3 is a replicated secret sharing protocol, each party has access to two out of three shares. Consequently, whenever one of the two shares is transmitted to another party (e.g., when reconstructing the output from the shares), two parties potentially can send the missing share. As example, when the value $[x] = (x_1, x_2, x_3)$ needs to be reconstructed, party i already has access to the shares x_i and x_{i-1} and can potentially send x_i to party $i - 1$ or x_{i-1} to party $i + 1$. Thus, when each party sends both shares to the correct other parties, the parties can check whether the received shares match and abort otherwise. Furthermore, one can delay sending one out of the two shares to the end of the protocol and only send a hash of all messages to reduce communication.

This simple and powerful technique allows to detect malicious behavior for all linear operations performed on the shares. However, it does not work for multiplications and dot products, since they involve a subprotocol translating the replicated share to just an additive one. So only one party knows the resulting share and the other parties cannot be used to verify correctness. Thus, multiplications need an additional way of checking correctness, which we investigate in the next sections.

3.3.2 Distributed Zero Knowledge Proofs

The (asymptotically) most efficient way in terms of communication complexity to verify correctness of multiplications and dot products is via distributed zero knowledge proofs [BGIN19], as employed by, e.g., the SWIFT [KPPS21] MPC protocol.

This method involves performing the semi-honest multiplication (and dot product) protocol of ABY3, and in the end, creating a zero-knowledge proof to prove to the other parties that you were following the protocol honestly. The advantages of the distributed zero knowledge proof is, that the two verifiers together know all involved data (due to replicated secret sharing) and can thus verify the proof more efficiently compared to non-distributed zero knowledge proof systems.

The main drawback of this method is, that the zero knowledge proof requires a large amount of polynomial interpolations, hence its computational overhead is large. Using this method in the use case would thus lead to a huge runtime increase compared to the semi-honest protocol, which is why we do not recommend to use it.

Remark 1. *In our implementation of the SWIFT protocol, we use the zero knowledge proofs for dot products and AND gate evaluations. A more efficient technique would be to just use it for the dot products, while leveraging cut-and-choose (Section 3.3.4) for verifying AND gates, since the computational overhead is significantly smaller. However, the computational overhead of the dot product proof is still highly expensive, hence, our recommendation of using different protocols still stands.*

3.3.3 Triple Sacrifice over Larger Rings

Another approach to verify the correctness of a semi-honest multiplication $[z] = [x] \cdot [y]$ is by performing an additional, correlated multiplication $[z'] = [x] \cdot [y']$ and *sacrifice* it to verify the validity of the first multiplication [ADEN21]. This sacrifice can be packed for many multiplications and requires to sample a random coin after all multiplications are done plus an additional communication round which requires to transmit a message with the size being linear in the number of multiplications.

This approach has the following drawbacks. First, to provide a statistical security of σ bits, one needs to perform both multiplications in a larger ring $\mathbb{Z}_{2^{k+\sigma}}$ increasing communication complexity. Second, when computing dot products, the size of the communication again depends on the size of the vectors resulting in infeasibly huge communication when applied to our use case. While the size-dependant communication could be outsourced to an input-independent offline phase, there exist more efficient verification protocols, which is why we do not recommend to use triple sacrificing in this use case.

Remark 2. *In our implementation, dubbed Malicious ABY3, we use triple sacrificing for both dot products and AND gates. This, however, requires at least a ring size of 41 bits for enough statistical security, increasing communication complexity per AND gate by a factor of at least 80 compared to semi-honest ABY3. Thus, using a different protocol, e.g., cut-and-choose (Section 3.3.4), for verifying AND gates is the preferred choice. However, replacing the AND gate verification does not compensate for the size-dependant dot products, which is why we still would not recommend this protocol in our use case.*

3.3.4 Triple Verification using Cut-And-Choose

Since triple sacrificing (Section 3.3.3) is not well suited for verifying AND gates, a different method is required for verifying their correctness. One such approach is called cut-and-choose [ABF+17]. This approach builds on the fact that many AND gates can be verified in a batched manner efficiently by having the same amount of correct precomputed shared AND gate triples $[z]^B = [x]^B \wedge [y]^B \in \mathbb{F}_2$.

To get N correct triples, one needs to precompute $N \cdot B + (B - 1) \cdot C$ semi-honest AND triples. These triples are then split into B buckets, where the first bucket consists of N

triples, while the remaining ones consist of $N + C$ ones. Now all buckets, except the first one, are pseudorandomly permuted, requiring to jointly sample a random permutation. Then, the first C triples of each bucket (except the first one) are published and checked for correctness. Finally, the first bucket is checked for correctness by sacrificing the remaining buckets. This results in the first bucket being correct shared AND triples with a soundness error of $\frac{1}{N^{B-1}}$. Furthermore, when permuting the N precomputed triples *after* all semi-honest AND gates are computed during protocol execution, one less bucket is required, i.e., the soundness error becomes $\frac{1}{N^{B}}$. Consequently, when pre-computing more than 2^{20} AND gates (which is easily the case for the data sizes of this use case), only one additional bucket is required for enough statistical security, leading to a communication increase of just a factor of 2. Finally, in the triple verification protocol (performed during triple generation and the final triple verification), 2 bits per AND gate are communicated, leading to a total communication overhead of factor 6 per AND gate.

Due to its efficiency in checking correctness of AND gates, we recommend using it in combination with the joint message passing protocol (Section 3.3.1) to get malicious security.

3.3.5 SPDZWise MACs

Another method to extend ABY3 to malicious security is by using authenticated sharing, similar to the SPDZ [DPSZ12] protocol. In this setting, each party has a share of a global MAC key α, while the actual value of α remains secret. The parties then use this key to authenticate each shared value $[x]$ with a MAC $[\delta_x] = [\alpha] \cdot [x]$. Each operation, such as addition and multiplication, of the semi-honest version of the protocol is then extended to also perform operations on the MACs such that the invariant $[\delta_x] = [\alpha] \cdot [x]$ is always fulfilled for each share. Furthermore, the result of each multiplication and dot-product, as well as values which are used as outputs of the protocol, are stored for a batched MAC-check protocol, in which the consistency of all MACs are checked, requiring the communication of one element and a hash value.

The advantage of MAC-based authentication is that it preserves the desired dot-product property of communication size being independent to the size of the vectors. However, each value is extended by a MAC, resulting in doubling the required computations and communications.

Furthermore, SPDZ was proposed to work over prime fields $\mathbb{F}_p$. However, to get σ bits of statistical security in rings $\mathbb{Z}_{2^k}$ one has to operate over a larger ring $\mathbb{Z}_{2^{k+\sigma}}$ [ADEN21; DEK21], further increasing communication complexity.

However, since the communication of dot products is independent to their input size, the overhead of the MAC and the larger ring is significantly smaller for our use case compared to the techniques discussed in this section. Hence, we recommend to use authentication MACs for the arithmetic parts of the use case. For the binary part though, where we do not have dot products, the communication overhead (similar to Section 3.3.3) is too large, which is why we do not recommend MACs there.

3.4 Protocol Choices

For the semi-honest version of the protocol, we choose semi-honest ABY3, since it is amongst the fastest MPC protocols in this setting.

For the malicious setting, as outlined in this section, we recommend using SPDZWise MACs (Section 3.3.5) for the arithmetic part of the protocol, while using cut-and-choose (Section 3.3.4) for the binary parts. Concretely, the workflow is the following. When the protocol is started, the parties together produce enough precomputed AND triples using cut-and-choose which are in the end used to verify the correctness of all AND gates.

Then, each party gets the shares of the inputs shared over $\mathbb{Z}_{2^{k+\sigma}}$. The next step is authenticating the inputs by multiplying them with the MAC-key, requiring the communication of n ring elements, where n is the input size (i.e., the bit length of the iris code in the use case).

Then, the parties perform the protocol using SPDZWise until the MSB-extraction subprotocol needs to be evaluated. Since we are switching protocols at this point, this step is similar to the output protocol of SPDZWise, hence we perform the MAC-check. Afterwards, the shares are reduced from $\mathbb{Z}_{2^{k+\sigma}}$ to $\mathbb{Z}_{2^k}$, and the MSB-extraction subprotocol followed by the OR-tree for producing one bit as the output, is executed using the joint message passing protocol and semi-honest AND gates. Finally, before the resulting bit is opened, all semi-honest AND gates are verified using the precomputed AND triples.

4 Benchmarks

In this section, we give some benchmarks for our protocols, concretely a plain evaluation baseline, a semi-honest ABY3, as well as a SPDZWise protocol (with cut-and-choose for binary computations) according to Section 3.4. All benchmarks where performed on machines with an *AMD EPYC 7543* CPU (32 threads, 2795 MHz) and 64 GB RAM. For all benchmark, we compute only whether an iris code is present in a database and do not additionally perform the same computation on rotations of the input code.

First, in Table 1 we perform benchmarks when each computing party is an individual server located in the same city. These servers are connected to each other with a network speed of 6.6 Gbit/s, and a latency of 1.5 ms. We perform these benchmarks for an iris code size of 12800 bits and a database size of 100 000 iris codes.

Next, we continue by varying the database size and give benchmarks for 1 000 000 iris codes in Table 2. Since we cannot benchmark SPDZWise for this database size due to the database not fitting into 64 GB of RAM, we benchmark this protocol for a smaller size of 10 000 in Table 3 to get an idea of scaling.

After varying the database sizes, we next give benchmarks for smaller iris code sizes of only 1024 bits. These benchmarks are given for a database size of 100 000 iris codes and can be found in Table 4.

Finally, we vary the network speed and give benchmarks for 100 000 iris codes of size 12800 bits in Table 5 when executing the protocol when the servers are located in different cities. Concretely, one server is located in Paris, one in Amsterdam, and the third one in Warsaw. Their slowest connection has a network speed of 477 Mbit/s, and a latency of 32 ms.

Table 1: Runtime and communication (in data sent per party) comparison of different MPC protocols, averaged over 3 runs. All parties run on different machines in the same city (network speed 6.6 Gbit/s, latency 1.5 ms). Benchmarks for code size $|\texttt{iris}| = 12800$ bits and database of size $|\texttt{DB}| = 100\,000$ iris codes. Chunk Size refers to number of dot-products and MSB extraction operations batched per communication round.

Protocol	Threads	Chunk Size	Malicious	Runtime	Data
				s	MB
Plain	1	-	-	0.134	-
ABY3	1	128	✗	10.460	1.048
ABY3	1	1 024	✗	4.665	0.649
ABY3	1	10 240	✗	3.084	0.598
SPDZWise	1	10 240	✓	7.834	4.643
ABY3	2	10 240	✗	1.660	0.598
ABY3	4	10 240	✗	0.942	
ABY3	8	10 240	✗	0.572	
ABY3	16	10 240	✗	0.414	
ABY3	32	10 240	✗	0.360	
ABY3	32	100 096	✗	0.234	
SPDZWise	32	10 240	✓	1.159	4.643
SPDZWise	32	100 096	✓	1.034	

As can be seen Table 5, network speed plays a huge role in the resulting performance. When sending smaller packages (i.e., small chunk sizes in the table) more communication rounds are required, which when having a slow latency, dominate the overall performance of the use case.

5 Open Problems and Bottlenecks

In this section, we discuss some of the bottlenecks in our implementation, give some ideas on how to improve them and touch on other areas that could be improved in future work.

5.1 Size of the Shared Iris Database.

Due to the requirements for the efficient dot-product operation, the iris database needs to be shared over a ring of size 2^k for some k, such that the summations do not overflow the ring. For the example parameters of an iris code length of 12800, this means that k is at least 14, and for simplicity we choose $k = 16$. Furthermore, every part has access to two out of three shares of the ring elements due to using replicated secret sharing, hence each party stores two ring elements of size $k = 16$ for each bit in an iris code. This expansion factor of 32 is a big reason for the slowdowns compared to the plain execution of the protocol, and also limits the loading of the full database into memory. As an example, the database for 1 million iris codes would require $2 \cdot 16 \cdot 12800 \cdot 1\,000\,000$ bits $= 51$ GB of memory.

Table 2: Runtime and communication (in data sent per party) comparison of different MPC protocols, averaged over 3 runs. All parties run on different machines in the same city (network speed 6.6 Gbit/s, latency 1.5 ms). Benchmarks for code size $|\texttt{iris}| = 12800$ bits and database of size $|\texttt{DB}| = 1\,000\,000$ iris codes. Chunk Size refers to number of dot-products and MSB extraction operations batched per communication round.

Protocol	Threads	Chunk Size	Malicious	Runtime	Data
				s	MB
Plain	1	-	-	1.343	-
ABY3	1	10 240	✗	30.801	5.925
ABY3	32	10 240	✗	3.386	5.925
ABY3	32	100 096	✗	2.566	

Table 3: Runtime and communication (in data sent per party) comparison of different MPC protocols, averaged over 3 runs. All parties run on different machines in the same city (network speed 6.6 Gbit/s, latency 1.5 ms). Benchmarks for code size $|\texttt{iris}| = 12800$ bits and database of size $|\texttt{DB}| = 10\,000$ iris codes. Chunk Size refers to number of dot-products and MSB extraction operations batched per communication round.

Protocol	Threads	Chunk Size	Malicious	Runtime	Data
				s	MB
SPDZWise	1	10 240	✓	0.847	0.825
SPDZWise	32	10 240	✓	0.178	

5.2 Multithreading

Currently, multithreading is only enabled for the computationally intensive parts of the protocol. Further optimizations could be achieved by also allowing the communication of the protocol to be multithreaded. However, especially for protocols with malicious security, the communication ordering is crucial for consistency checks, which makes multithreaded communication more difficult to implement. Implementing this could bring the multithreaded scaling closer to linear, however, the effects of memory contention are already noticeable for a large number of threads in the current implementation, so we do not expect to reach fully linear speedups from multithreading.

5.3 SPDZWise MACs and the effects on the Database.

Since we recommend to use SPDZWise MACs (Section 3.3.5), which operate over larger rings $\mathbb{Z}_{2^{k+\sigma}}$, the shared database gets further increased. Concretely, to get at least 40 bits of statistical security, each bit of the iris codes is stored as at least 108 bits (two shares of 54 bits), where we choose 128 bits for simplicity. Furthermore, to accommodate for the MACs, the database size has to be doubled again.

Alternatively, one can skip storing the MACs at the cost of computing them in a setup phase at the cost of communicating $64 \cdot |\texttt{iris\_code}| \cdot |\texttt{db}|$ bits. As an example, the database for 100k iris codes would require $2 \cdot 128 \cdot 12800 \cdot 100\,000$ bits $= 40.9$ GB of memory.

Table 4: Runtime and communication (in data sent per party) comparison of different MPC protocols, averaged over 3 runs. All parties run on different machines in the same city (network speed 6.6 Gbit/s, latency 1.5 ms). Benchmarks for code size $|\texttt{iris}| = 1024$ bits and database of size $|\texttt{DB}| = 100\,000$ iris codes. Chunk Size refers to number of dot-products and MSB extraction operations batched per communication round.

Protocol	Threads	Chunk Size	Malicious	Runtime	Data
				s	MB
Plain	1	-	-	0.023	-
ABY3	1	10 240	✗	0.392	0.598
SPDZWise	1	10 240	✓	1.101	4.643
ABY3	32	10 240	✗	0.149	0.598
SPDZWise	32	10 240	✓	0.548	4.643

5.4 Renewal of MACs.

To further strengthen security and prevent parties from cheating by knowing the MAC keys we propose periodically renewing the MAC keys (maybe once a month). This entails recomputing the MACs for each share in the database, requiring to communicate $64 \cdot |\texttt{iris\_code}| \cdot |\texttt{db}|$ bits.

5.5 Inputs are Bits shared over Larger Rings.

For efficiency we let the orbs share each bit of the input iris code over $\mathbb{Z}_{2^k}$ instead of $\mathbb{F}_2$. The reason for that is that the orb would be able to share wrong bits anyways, hence it has to be trusted to follow the protocol already. Second, the computing parties need to store the shares over $\mathbb{Z}_{2^k}$ for efficiency anyways. If this is undesired, an extension of our protocol would allow to share the inputs as bits. The workflow changes as follows. First, the input is interpreted as three trivial sharings $[x_1], [x_2], [x_2]$ in $\mathbb{Z}_{2^{k+\sigma}}$ (similar to Section 2.2) and authenticated using the shares of the MAC. Then, the three shares are combined using two arithmetic XOR gates, i.e., by computing $[y] = [x_1] + [x_2] - [x_1] \cdot [x_2]$ and $[z] = [x_3] + [y] - [x_3] \cdot [y]$. This results in authenticated shared bits over $\mathbb{Z}_{2^{k+\sigma}}$, as required for the rest of the protocol.

5.6 Precomputing Triples.

In Section 4, we benchmark the whole SPDZWise protocol, including generating triples to verify the AND gates. This triple generation, however, can be performed independently to the actual protocol execution. Thus, when computing servers do not have to respond to an actual query, they can precompute many of those triples, removing the runtime and communication from the actual execution. This can lead to a significant runtime increase, since triple generation requires bit-permutations which can not be parallelized using multithreading. Since this permutation is also part of verifying semi-honest AND gates at the end of the protocol, another future research direction could be to find a parallelizable permutation, similar to discussed in [ABF+17].

Table 5: Runtime and communication (in data sent per party) comparison of different MPC protocols, averaged over 3 runs. All parties run on different machines in different cities (slowest network speed 477 Mbit/s, highest latency 32 ms). Benchmarks for code size $|\texttt{iris}| = 12800$ bits and database of size $|\texttt{DB}| = 100\,000$ iris codes. Chunk Size refers to number of dot-products and MSB extraction operations batched per communication round.

Protocol	Threads	Chunk Size	Malicious	Runtime s	Data MB
Plain	1	-	-	0.134	-
ABY3	1	10 240	✗	5.294	0.598
ABY3	1	100 096	✗	3.575	
SPDZWise	1	10 240	✓	11.071	4.643
SPDZWise	1	100 096	✓	9.238	
ABY3	32	10 240	✗	2.543	0.598
ABY3	32	100 096	✗	1.142	
SPDZWise	32	10 240	✓	4.099	4.643
SPDZWise	32	100 096	✓	2.155	

5.7 Different Adder Circuits for MSB-extraction.

As can be seen in Table 5, in network connection with high latency the runtime is dominated by the communication rounds. Hence, replacing the ripple-carry adder with depth-optimized carry-lookahead header, which require fewer communication rounds, but larger total communication, can lead to significant performance gains in these networks.

5.8 Using solely protocols over binary fields.

The main optimizations used in the discussed protocols makes the communication independent to the size of the iris code by using arithmetic operations over $\mathbb{Z}_{2^k}$ instead of binary operations over $\mathbb{F}_2$. However, this comes at the cost of a large expansion factor of 16 for datatypes, which makes memory size and contention a potential bottleneck. Thus, future work could investigate if this tradeoff is worth it if the network conditions are very good (e.g., when the servers are located in the same data region or data center). There, one could use the full Gbps bandwidths of the network to facilitate protocols that operate over $\mathbb{F}_2$, but do not have the dot-product optimization.

5.9 Dedicated SIMD implementation.

While modern compilers are pretty good at autovectorization, a dedicated SIMD implementation could offer further speedups. The nightly version of Rust has support for the `portable-simd` API, which can serve as an underlying datatype for our shares. However, this API is still in development and not yet stable, so we did not use it in our implementation, but this could be investigated in the future.

References

[ABF+17] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. “Optimized Honest-Majority MPC for Malicious Adversaries - Breaking the 1 Billion-Gate Per Second Barrier”. In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2017, pp. 843–862.

[ADEN21] Mark Abspoel, Anders P. K. Dalskov, Daniel Escudero, and Ariel Nof. “An Efficient Passive-to-Active Compiler for Honest-Majority MPC over Rings”. In: *ACNS (2)*. Vol. 12727. LNCS. Springer, 2021, pp. 122–152.

[BGIN19] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. “Practical Fully Secure Three-Party Computation via Sublinear Distributed Zero-Knowledge Proofs”. In: *CCS*. ACM, 2019, pp. 869–886.

[DEK21] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. “Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security”. In: *USENIX Security Symposium*. USENIX Association, 2021, pp. 2183–2200.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. “Multiparty Computation from Somewhat Homomorphic Encryption”. In: *CRYPTO*. Vol. 7417. LNCS. Springer, 2012, pp. 643–662.

[KPPS21] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. “SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning”. In: *USENIX Security Symposium*. USENIX Association, 2021, pp. 2651–2668.

[MR18] Payman Mohassel and Peter Rindal. “ABY^3: A Mixed Protocol Framework for Machine Learning”. In: *CCS*. ACM, 2018, pp. 35–52.