Large-Scale MPC: Scaling Private Iris Code Uniqueness Checks to Millions of Users

Remco Bloemen[2], Daniel Kales[1], Philipp Sippl[2] and Roman Walch[1]

[1] TACEO, Graz, Austria
lastname@taceo.io
[2] Worldcoin Foundation

Abstract. In this work we tackle privacy concerns in biometric verification systems that typically require server-side processing of sensitive data (e.g., fingerprints and Iris Codes). Concretely, we design a solution that allows us to query whether a given Iris Code is similar to one contained in a given database, while all queries and datasets are being protected using secure multiparty computation (MPC). Addressing the substantial performance demands of operational systems like World ID and aid distributions by the Red Cross, we propose new protocols to improve performance by more than three orders of magnitude compared to the recent state-of-the-art system Janus (S&P 24). Our final protocol can achieve a throughput of over a million Iris Code comparisons per second on a single CPU core, while protecting the privacy of both the query and database Iris Codes. We additionally investigate GPU acceleration for some building blocks of our protocol, which results in further speedups of over 38x compared to the respective multi-threaded CPU implementation.

Keywords: MPC, Iris Codes, uniqueness, privacy, World ID

1 Introduction

Decentralized blockchains very fundamentally rely on the existence of a sybil-resistant mechanism for their consensus. While the mechanisms applied in this context (e.g. Proof-of-Stake or Proof-of-Work) make it very hard to accumulate a large share of representative power, they do not enforce that each participant owns *exactly* one share of equal size. However, there are many practical applications, in which it is of utmost importance to ensure that users can only sign up *exactly* once for a given service.

For example, Worldcoin's World ID[1] is a privacy-preserving *Proof-of-Personhood*, which requires that only real humans are able to sign up exactly once and are then able to reuse this proof for other applications. Another very relevant example is aid distribution by, e.g., the Red Cross [ELS+24]. While everyone should be able to apply for the aid program, people should be prevented from signing up multiple times, since available resources are usually scarce and should be used to help as many people as possible.

In both examples, the *uniqueness* property is enforced by storing biometric information, such as so-called Iris Codes derived from the human iris, in a database on a central server. When new users sign up, their biometric information is captured and some sort of liveness check is executed to ensure it is an authentic scan; afterwards their biometric information is sent to this server which checks whether it is close to any other sample in the database. If no match was found, the user is allowed to sign up and their biometric information is added to the database.

[1] https://worldcoin.org/world-id

While this simple protocol ensures uniqueness, it requires a single server to collect and store information derived from biometric data which may pose privacy concerns to some. Leakage of such a database could become even more critical when this biometric database is linked to other datasets. Furthermore, in such cases the database holders may potentially misuse the dataset to, e.g., discriminate against specific ethnicities,[2] and it becomes a prime target for attackers who might try to steal the data.

In this paper we propose an efficient and scalable protocol to ensure uniqueness of biometric information, concretely Iris Codes, in a distributed database. The database, thereby, is protected by secure multiparty computation (MPC), a cryptographic technique which allows multiple parties to compute functions on private data without revealing their private inputs to each other. Concretely, the database is split amongst multiple parties, such that no party on their own learns anything about the contained biometric information. Furthermore, new Iris Codes are also shared using MPC, protecting their privacy as well.

While MPC solves the privacy issues of the uniqueness service above, trivially applying it to any use case comes with a significant performance loss (see, e.g., [ELS+24]). Even more critical for real-world deployment are the performance requirements of large-scale systems, such as the World ID infrastructure which we focus on, where the overall system needs to scale to a database of tens of millions of Iris Codes while supporting a throughput of 10 queries per second. Looking at these numbers we can already draw two conclusions: (i) None of the related work comes close to achieving these requirements (see also next section). Hence, we have to come up with more efficient approaches and protocols, while also leveraging all available hardware, such as GPU's. (ii) The final solution will inevitably involve having a cluster of computing nodes for each MPC party. Hence, the main goal of this work is making the protocol as efficient as possible to reduce the required number of cluster nodes and, therefore, the overall cost.

1.1 Related Work

Many privacy-preserving solutions for authentication using biometric information have been introduced in the literature [BCP13, ITK13, JW99, MPR20, Tam16, YY13, ZZX+18], with the recently published Janus [ELS+24] (S&P 2024) being the most similar one to our solution. In Janus, the authors propose privacy-preserving protocols to prevent recipients from registering to aid distribution process multiple times by using biometric information. In essence, their deduplication protocol is very similar to the one employed by Worldcoin, i.e., using similarity measurements, such as the hamming distance, to determine whether biometric data of a new participant matches already registered ones. However, while we use a 3-party MPC solution (see later sections) to protect new and existing biometric data, they compare three approaches based on 2-party garbled circuits [Yao86], somewhat homomorphic encryption (SHE) [Gen09], and trusted execution environments (TEEs). While they acknowledge that the security guarantees of TEEs are broken on a regular basis by software and hardware based side channel attacks (see, e.g., [LSG+18, BMW+18], and https://sgx.fail/), their single-threaded TEE solution outperforms SHE and MPC and achieves deduplication of a database with 8000 users in 50 ms. Furthermore, they show that their SHE solution outperforms their MPC-based one while achieving a high deduplication runtime of 40 s for small databases with 8000 users. As we show in Section 6, we significantly outperform Janus by several orders of magnitude. Our solution (even without GPU acceleration) can achieve single-threaded throughput of 1 million Iris Codes per second, while supporting Iris Codes with larger precision. Consequently, we believe our protocol can be used to significantly speed up the deduplication check for aid distribution as well. Furthermore, as described in [ELS+24], the non-collusion assumption between the three computing nodes in our design can easily be achieved in the context of aid

[2] https://www.hrw.org/news/2022/03/30/new-evidence-biometric-data-systems-imperil-afghans

distribution by the International Committee of the Red Cross (ICRC).

Only a handful of papers consider GPU acceleration for MPC computations. Early works focus on implementation of generic, basic protocols [FN13, HMSG13]. More recent works focus on machine learning-related inference or training use-cases, or building blocks for it [WWP22, MLS+20, TKTW21, JSH+24]. Our work follows on a similar line, focusing on accelerating dot-product operations in MPC.

2 MPC Background

Secure multiparty computation (MPC) allows multiple parties to compute a function on their combined private input data without leaking these inputs to each other. In this paper we focus on secret sharing based MPC protocols [Sha79, DPSZ12, MR18]. In these protocols, private data is shared amongst n parties, such that each party holds a share which on its own does not leak any information about the original dataset. Given enough shares, however, the parties can reconstruct the data. Furthermore, the parties can use the shares to compute functions, which in turn yield shares of the result. *Linear* operations, such as adding shares or multiplying them with constants, can directly be computed on the shares, whereas multiplications of two shared values require the parties to exchange some form of randomized information. Consequently, the more multiplications are present in the circuit one wants to compute, the more data needs to be exchanged by the parties and thus, the number of multiplications in the circuit is a large part of the cost metric of MPC protocols and should be reduced for efficiency.

Many different flavours of MPC protocols exist, such as protocols based on additive secret sharing (e.g., SPDZ [DPSZ12]), replicated secret sharing (e.g., ABY3 [MR18], Swift [KPPS21], Fantastic Four [DEK21]), Shamir's secret sharing [Sha79], and Yao's garbled circuits [Yao86]. These protocols differ in the general sharing techniques, supported number of computing parties, tolerated number of malicious parties, and general security model.

Due to the high performance requirements of the World ID infrastructure we focus on protocols which require a *honest majority* among the parties in this work (see Section 4.1). Furthermore, similar to related work (e.g., [ELS+24]) we focus on the *semi-honest* security model.[3] Finally, since MPC protocols usually suffer from high communication complexity, and the final solution will inevitably run in server farms with low-latency network connections, we will focus on reducing the total communication at the cost of more communication rounds.

2.1 Notation

Throughout the paper we depict a secret sharing of a value $x \in \mathbb{Z}_t$ as $[x]$. Furthermore, we denote with $\vec{x} \odot \vec{y}$ the element-wise multiplication (Hadamard product) between two vectors $\vec{x}, \vec{y} \in \mathbb{Z}_t^\ell$.

3 The Iris Code Membership Protocol deployed by World ID

In this work, we focus on the uniqueness services deployed by Worldcoin for their World ID infrastructure, which is why we use Iris Codes [Dau02] as the biometric data of choice. Iris Codes are derived from images of human eyes and are considered as a highly accurate source of biometric information. However, since some parts of the eyes may be obstructed by, e.g., eyelashes, some bits of the Iris Codes should be excluded from matching protocols.

[3] All protocols in this paper can be adapted to provide malicious security as well. See, e.g., the report contained in https://github.com/TaceoLabs/worldcoin-experiments for some preliminary experiments.

Thus, Iris Codes are usually accompanied by a mask which removes faulty bits before matching. In the Worldcoin infrastructure, dedicated iris capture stations (called *orb*s) are used to create both the Iris Codes and the corresponding mask, while ensuring the liveness property of the user. Matching itself is done by computing a normalized hamming distance, which is then compared to a threshold.

The main uniqueness check protocol, as given in Algorithm 1, calculates whether a new Iris Code $\vec{c} \in \mathbb{F}_2^l$ with the mask $\vec{m} \in \mathbb{F}_2^l$ is similar (calculated via the hamming distance) to any Iris Code in a given database. In the currently deployed system the Iris Code size l is given as $l = 12800$. Furthermore, the deployed iris check involves additional protection against false negatives. First, each Iris Code is rotated $r = 31$ times by a small offset and each rotation is individually checked against the database. Second, codes for both irises per person are checked against the database. While Algorithm 1 does not explicitly consider these two additional checks, they can simply be modelled by executing it $2r$ times.

Algorithm 1 The Iris Code Membership Protocol without MPC. It checks whether the Iris Code $\vec{c}$, under 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: $\vec{c} \in \mathbb{F}_2^l, \vec{m} \in \mathbb{F}_2^l, C_{\texttt{DB}} \in \mathbb{F}_2^{s \times l}, M_{\texttt{DB}} \in \mathbb{F}_2^{s \times l}$
Output: `true` if $\vec{c}$ is similar to an entry in the DB, `false` otherwise.

```
for i in 0..s do
    m'⃗ ← m⃗ ∧ M_DB[i]                          ▷ Combine masks.
    ml ← CountOnes(m'⃗)
    hd ← CountOnes((c⃗ ⊕ C_DB[i]) ∧ m'⃗)          ▷ Hamming distance.
    if hd/ml < MATCH_RATIO then
        return true                             ▷ Iris is similar!
return false                                    ▷ No match found.
```

In the next sections, we explore different methods for translating Algorithm 1 to the MPC setting. First, in Section 4 we focus on the case where only the Iris Codes will be protected, whereas the accompanying masks are assumed to be known in plain by all computing parties. Then, in Section 5 we extend the solution to also protect the masks. Finally, we give some benchmarks in Section 6.

4 First MPC Protocol: Protecting the Iris Codes

In the first version of the MPC protocol, we aim to protect the Iris Codes while the corresponding masks are allowed to stay public. Thus, both the new Iris Code $\vec{c}$ should be protected from the computing parties, as well as the Iris Codes in the database $C_{\texttt{DB}}$. The masks $\vec{m}$ and $M_{\texttt{DB}}$ are assumed to be known by the parties. This section also serves as a stepping stone for the version of the protocol in Section 5, where masks are also protected.

4.1 Efficient Hamming Distance Calculation

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., an operation in $\mathbb{F}_2$, followed by counting the ones to get a result in a larger ring $\mathbb{Z}_t$. 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 l. 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 at least

linear in the length l. However, if we have the precondition that the input vectors are already shared over a larger ring $\mathbb{Z}_t$ instead of $\mathbb{F}_2$, we can rewrite the hamming distance calculation to

$$\mathtt{hd}(\vec{a}, \vec{b}) = \sum_i a_i + \sum_i b_i - 2 \cdot \langle a_i, b_i \rangle \,. \tag{1}$$

This reduces the calculation of the hamming distance to two sums which can be computed without party interaction in secret-sharing based 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 efficient dot-product evaluations. Protocols that have a honest-majority security assumption (e.g., Shamir secret sharing [Sha79] and replicated sharing protocols such as ABY3 [MR18]) can support dot-products that require communication which is independent of the size of the vectors, allowing us to efficiently realize the hamming distance calculation in MPC. This optimization alone reduces communication from $\approx 3\,\mathrm{GB}$ when using binary circuits to just $2\,\mathrm{MB}$ for $l = 12800, t = 2^{16}$ and a database size of $s = 1\,000\,000$.

4.1.1 Masked Bitvectors: Reducing the Workload

When looking at the equation $\mathtt{hd} \leftarrow \mathtt{CountOnes}((\vec{c} \oplus C_{\mathtt{DB}}[i]) \wedge \vec{m}')$, i.e., $\mathtt{hd}$ is calculated on bitvectors and masks, one can apply a further optimization by changing the representation of masked Iris Codes.

First, we introduce a new type, dubbed *masked bit*, consisting of three states $\{\mathtt{T}, \mathtt{U}, \mathtt{F}\}$, where $\mathtt{U}$ indicates that the mask bit is not set, $\mathtt{F}$ indicates a set mask bit while the code bit is not set, and $\mathtt{T}$ indicates both mask and code bits are set. We represent this new type as $(\mathtt{T}, \mathtt{U}, \mathtt{F}) = (-1, 0, 1)$. One can translate a bit $a \in \mathbb{F}_2$ with the mask $m_a \in \mathbb{F}_2$ to the masked bit representation $a' \in \{-1, 0, 1\}$ by computing $a' = m_a - 2 \cdot (a \wedge m_a)$ in a larger ring (using the canonical embedding of signed integers in such a ring).

By using this representation, one can rewrite the comparison $\mathtt{hd} < \mathtt{MATCH\_RATIO} \cdot \mathtt{ml}$ to a version using masked bitvectors, i.e.,

$$\langle c', C'_{\mathtt{DB}}[i] \rangle > (1 - 2 \cdot \mathtt{MATCH\_RATIO}) \cdot \mathtt{ml} \,,$$

removing two sums from the computation, improving performance. We refer to Appendix B for more details. Furthermore, only the left-hand-side of the resulting comparison, i.e., the dot product, depends on shared values, the right-hand-side can be calculated from public values only.

4.2 Efficient Comparison

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 in the chosen signed integer representation (see Lemma 1), a core building block is an MSB-extraction protocol. This subprotocol requires to change the sharing type of additive shares over $\mathbb{Z}_t$ to boolean shares over $\mathbb{F}_2^{\lceil \log_2(t) \rceil}$. 3-party replicated sharing protocols, such as ABY3 [MR18], are amongst the most efficient MPC protocols for converting between shares over bits and larger rings.

Accumulating the Resulting Bits. If one wants to prevent leaking which Iris Code in the database was similar to the query, one can accumulate all resulting comparison bits by setting them to true if at least one comparison was true. This operation is equivalent to many boolean OR operations. In MPC, one can compute $x \vee y = x \oplus y \oplus (x \wedge y)$. To reduce the number of communication rounds, we evaluate the accumulation of all bits in a binary tree.

4.3 MPC Protocols

After discussing the subprotocols in the previous sections we give the final MPC algorithm in Algorithm 2. The workflow is the following: The orb station which creates a new Iris Code $\vec{c} \in \mathbb{F}_2^l$ with mask $\vec{m} \in \mathbb{F}_2^l$ secret shares each bit of $\vec{c}$ (using the masked representation of Section 4.1.1) over a larger ring $\mathbb{Z}_t$ to the MPC nodes while sending the mask in plain. Then, the MPC nodes evaluate Algorithm 2 to produce a shared result bit, which is true if the new Iris Code is similar to an existing one. This bit is then reconstructed at a chosen output node.

Algorithm 2 The Iris Code Membership Protocol with MPC. It checks, whether the secret-shared Iris Code $[\vec{c}]$ (represented as masked bitvector), under the public mask $\vec{m}$ is similar to any secret-shared Iris Code in the database $[C_{\texttt{DB}}]$ (represented as masked bitvector) under public masks $M_{\texttt{DB}}$. l is the size of the Iris Codes in bits, s is the number of codes in the database.

Input: $[\vec{c}] \in \mathbb{Z}_t^l, \vec{m} \in \mathbb{F}_2^l, [C_{\texttt{DB}}] \in \mathbb{Z}_t^{s \times l}, M_{\texttt{DB}} \in \mathbb{F}_2^{s \times l}$
Output: Sharing of `true` if $\vec{c}$ is similar to an entry in the DB, `false` otherwise.

$[\vec{b}] \leftarrow [\vec{0}] \in \mathbb{F}_2^s$
for i in $0..s$ **do**
  $\texttt{ml} \leftarrow \texttt{CountOnes}(\vec{m} \wedge M_{\texttt{DB}}[i])$ ▷ Combine masks.
  $[\texttt{hd}] \leftarrow \langle [\vec{c}], [C_{\texttt{DB}}[i]] \rangle$ ▷ Hamming distance.
  $[b[i]] \leftarrow \texttt{MSB}((1 - 2 \cdot \texttt{MATCH\_RATIO}) \cdot \texttt{ml} - [\texttt{hd}])$ ▷ Comparison
return $\texttt{OrTree}([\vec{b}])$ ▷ Aggregate the resulting bits

4.3.1 Baseline: Semi-honest ABY3

Since 3-party replicated honest majority protocols, such as ABY3 [MR18] provide both an efficient dot product, as well as efficient arithmetic to binary conversion for secret shares, we will use the semi-honest variant of ABY3 as a baseline for further discussions. We refer to Appendix C for a small Introduction to ABY3. Importantly, computing a dot-product requires the same amount of communication as a single multiplication and requires a CPU workload of only two plain dot-products in our use case.

Sharing Ring and MSB-extraction. As mentioned in Section 4.2, we will use the efficient arithmetic-to-binary conversion of ABY3 to evaluate the comparison $[b[i]] \leftarrow \texttt{MSB}((1 - 2 \cdot \texttt{MATCH\_RATIO}) \cdot \texttt{ml} - [\texttt{hd}])$ in Algorithm 2. Consequently, we need to ensure that we choose the bitsize k of the ring $\mathbb{Z}_{2^k}$ used for sharing to be large enough, such that the subtraction $(1 - 2 \cdot \texttt{MATCH\_RATIO}) \cdot \texttt{ml} - [\texttt{hd}]$ does not produce overflows in the canonical representation of signed integers in the ring.

Lemma 1. *The comparison of* $\texttt{hd} > f \cdot \texttt{ml}$ *for a real number* $f \in [0, 1]$, $\texttt{hd}$ *being the dot product of two masked bitvectors of size* l*, and* $\texttt{ml}$ *being the dot product of two bitvectors of size* l*, is equivalent to* $\texttt{MSB}(\lceil f \cdot \texttt{ml} \rceil - \texttt{hd})$ *when calculated over* $\mathbb{Z}_t$*, if* $l < \frac{t}{4}$ *and* $l < t - 2^{\lceil \log_2(t) \rceil - 1}$.

For a proof of Lemma 1 we refer to Appendix A.1. Using Lemma 1, we choose $\mathbb{Z}_t$ with $t = 2^{16}$ as the sharing ring, since both conditions are fulfilled for $l = 12800$.

To extract the MSB of a share $[x] \in \mathbb{Z}_{2^{16}}$, we transform it to boolean shares over $\mathbb{F}_2^{16}$. In ABY3 this is done by interpreting the share $[x] = (x_1, x_2, x_3)$ (where $x = x_1 + x_2 + x_3 \mod 2^{16}$) as three trivial boolean shares and adding them via a 16-bit binary adder circuit in MPC, which implicitly reduces the result mod 2^{16}. We give the algorithmic description of the MSB-extraction procedure in Appendix E. When using a ripple-carry

adder (Algorithm 7), which requires the least amount of AND gates, MSB extraction requires 29 AND gates in 15 communication rounds.

4.3.2 Shamir Sharing: Reducing the database and dot products

As discussed in Appendix C, an ABY3 share consists of two additive shares. Consequently, since one has to share each bit of the Iris Code as sharing over $\mathbb{Z}_{2^{16}}$, the database increases from $s \cdot l$ bits to $2 \cdot s \cdot 16 \cdot l = 32 \cdot s \cdot l$ bits. Furthermore, for each MPC dot product two plain dot products have to be calculated. Since the dot products will dominate the runtime on an CPU for large databases, this is an undesirable property.

Luckily, one can counteract these disadvantages when switching to Shamir secret sharing [Sha79]. In Shamir sharing, secrets are shared as points on random polynomials of degree d, where the secret itself is located in the constant term of the polynomial. The secrets can be reconstructed from the shares by, e.g., Lagrange interpolation on $d+1$ shares, which is a linear operation. Similar to additive sharing, linear operations can be performed on the shares without requiring party interaction and without changing the degree d of the underlying sharing polynomial. While multiplications can be implemented as trivial multiplications of the shares, they increase the degree of the underlying polynomial to $2 \cdot d$, i.e., twice as many parties are needed for reconstruction. Consequently, multiplications are usually followed by a degree reduction step, which requires party interaction. However, similar to ABY3, one does not immediately have to reduce the degree after each multiplication in a dot-product evaluation. Consequently, a dot product can be implemented with communication being equivalent to that of a single multiplication in Shamir secret sharing.

One difference between using ABY3 and Shamir sharing, however, is that one cannot instantiate Shamir secret sharing over $\mathbb{Z}_{2^{16}}$ directly. Consequently, we will use a 16-bit prime field $\mathbb{F}_p$ with $p = 2^{16} - 17$, since it suffices to fulfill Lemma 1.

Iris Membership using Shamir Sharing. When using Shamir secret sharing instead of ABY3 the workflow changes as follows: First, the orb shares each bit of a new Iris Code as a Shamir share with degree of one to the MPC nodes. Then, the parties compute the dot products (with delayed modular reduction for performance), requiring just one plain dot product instead of two as in ABY3. The resulting dot products correspond to degree-2 sharings, i.e., all three parties are required for reconstruction. These degree-2 sharings, however, can be transformed to additive sharings (mod p) by multiplying the shares with the corresponding Lagrange coefficient. We transform the shares to replicated sharings by sending the share of party i to the next party $i+1$ (including re-randomization), allowing us to utilize efficient conversion protocols from ABY3 in later steps. Finally, since a Shamir share is of size $\lceil \log_2 p \rceil$ bits, the database size of the shared Iris Codes reduces from $32 \cdot s \cdot l$ using ABY3 over $\mathbb{Z}_{2^{16}}$ to $16 \cdot s \cdot l$ using Shamir Sharing with $p = 2^{16} - 17$.

MSB-extraction mod p. While it is easy to transform Shamir shares to replicated additive shares (mod p), MSB extraction becomes slightly more complicated. The strategy used in ABY3 when operating over $\mathbb{Z}_{2^k}$ does not directly work over prime fields, since k-bit addition circuits implicitly operate mod 2^k. Hence, we need to simulate a mod p reduction when using the addition circuit strategy. Since ABY3 needs to add three values in binary circuits, we potentially need to subtract p twice to get the reduced result. Thus, we will use a different strategy for starting the bitextract routine to directly get two binary shares which are already reduced mod p. Consequently, we only need to conditionally subtract p at most once in binary circuits to get a reduced result.

The strategy is as follows. First, each party produces a random binary zero-share r_i which can be done without interaction. Then, party 2 locally adds $(x_1 + x_2 \mod p)$

and sets its new share of $x_{1,2}$ to $(x_1 + x_2 \mod p) \oplus r_2$.[4] Then, $[a]^B = (r_1, (x_1 + x_2 \mod p) \wedge r_2, r_3)$ is a valid binary share of $x_1 + x_2 \mod p$. Thus, the share can be converted to a replicated share by resharing, i.e., each party sends its share to the next party. Furthermore, $[b]^B = (0, 0, x_2)$ is a valid binary share of x_2 for which the parties can produce a replicated sharing without interaction.

The parties then continue to add the two k-bit values $[a]^B$ and $[b]^B$ using a binary addition circuit to produce a $(k+1)$-bit value, which requires k bits of communication in k rounds using a ripple-carry adder. Then, we continue by computing a subtraction circuit, subtracting p from the $(k+1)$-bit result to get the k-th bit and an overflow bit. This circuit requires $k+1$ AND gates in $k+1$ communication rounds. Finally, to get the real MSB, we calculate a multiplexer circuit choosing between the k-th bit of the result of the addition circuit and the result of the subtraction circuit depending on the overflow bit. This multiplexer requires 1 AND gate in 1 round.

Thus, MSB-extraction using replicated sharing mod p requires $3 \cdot k + 2$ AND gates in $2 \cdot k + 3$ communication rounds. With $k = 16$, this results in 50 AND gates in 35 communication rounds. The full algorithm is given in Algorithm 12 in Appendix E.

5 Second MPC Protocol: Protecting the Codes and Masks

The goal of the second MPC protocol is to extend Algorithm 2 to also protect the masks $\vec{m} \in \mathbb{F}_2^l$ and $M_{\texttt{DB}} \in \mathbb{F}_2^{s \times l}$. This triggers two changes: First, the mask bits also need to be shared over a larger ring $\mathbb{Z}_t$ and $\texttt{ml}$ can be calculated by a dot product. Second, since $\texttt{ml}$ is unknown to the computing parties, they cannot trivially multiply it with a real value f and round the result. The second change is the reason why we need to approximate $(1 - 2 \cdot \texttt{MATCH\_RATIO}) \approx \frac{a}{b}$ with $a \leq b$ and $a, b \in \mathbb{Z}_{t'}$. Then, while ensuring that the operations do not overflow, $(1 - 2 \cdot \texttt{MATCH\_RATIO}) \cdot \texttt{ml} < \texttt{hd}$ can be calculated as $\texttt{MSB}(a \cdot [\texttt{ml}] - b \cdot [\texttt{hd}])$. We depict the resulting algorithm in Algorithm 3.

Algorithm 3 The Iris Code Membership Protocol with MPC. It checks, whether the secret-shared Iris Code $[\vec{c}]$ (represented as masked bitvector), under the shared mask $[\vec{m}]$ is similar to any secret-shared iris in the database $[C_{\texttt{DB}}]$ (represented as masked bitvector) under shared masks $[M_{\texttt{DB}}]$. l is the size of the Iris Codes in bits, s is the number of codes in the database.

Input: $[\vec{c}] \in \mathbb{Z}_{t_1}^l, [\vec{m}] \in \mathbb{Z}_{t_2}^l, [C_{\texttt{DB}}] \in \mathbb{Z}_{t_1}^{s \times l}, [M_{\texttt{DB}}] \in \mathbb{Z}_{t_2}^{s \times l}$
Output: Sharing of `true` if $\vec{c}$ is similar to an entry in the DB, `false` otherwise.

```
[b⃗] ← [0⃗] ∈ F_2^s
for i in 0..s do
    [ml] ← ⟨[m⃗], [M_DB[i]]⟩                    ▷ Combine masks.
    [hd] ← ⟨[c⃗], [C_DB[i]]⟩                    ▷ Hamming distance.
    [ml'] ← lift_ml([ml])      ▷ Some lifting to a larger ring to prevent overflows
    [hd'] ← lift_hd([hd])      ▷ Some lifting to a larger ring to prevent overflows
    [b[i]] ← MSB(a · [ml'] − b · [hd'])        ▷ Comparison
return OrTree([b⃗])                            ▷ Aggregate the resulting bits
```

Due to changing the comparison operation, we have to adapt Lemma 1 to Lemma 2 for correct bounds, such that the MSB extraction is equivalent to the comparison. The proof can be found in Appendix A.2.

Lemma 2. *The comparison of* $\texttt{hd} > \frac{a}{b} \cdot \texttt{ml}$ *for* $a, b \in \mathbb{Z}_{t'}$, $a \leq b$, $\texttt{hd}$ *being the dot product of two masked bitvectors of size* l*, and* $\texttt{ml}$ *being the dot product of two bitvectors of size* l*, is equivalent to* $\texttt{MSB}(a \cdot \texttt{ml} - b \cdot \texttt{hd})$ *when calculated over* $\mathbb{Z}_t$*, if* $b \cdot l < \frac{t}{4}$ *and* $b \cdot l < t - 2^{\lceil \log_2(t) \rceil - 1}$.

[4] This optimization only works if the parties are semi-honest, which is the setting we focus on here.

5.1 Lifting vs. Larger Dot-Product

Algorithm 3 immediately shows the potential for trading off the cost of the dot product with the MSB extraction algorithm. First, if we compute the dot products over a smaller ring/field, e.g., a 16-bit modulus t, we have to lift (in MPC) the results to a larger space (e.g., 36-bit to allow 20-bits of precision for a and b) such that the multiplications with a, b do not overflow and the MSB extraction can be used for the comparison. Second, if we directly compute the dot products in a larger ring/field, we do not have to compute the lifting in MPC at the cost of more expensive dot product computations.

Thus, we will first propose some efficient lifting algorithms. First, observe that lifting a multiplication is free if we change the resulting ring accordingly. Let $[x]_t$ be an additive share mod t, then the multiplication (without modular reduction) $s \cdot [x]_t = [s \cdot x]_{st}$ becomes a valid sharing of $s \cdot x \mod st$ (see Algorithm 13). Thus, when approximating $(1 - 2 \cdot \texttt{MATCH\_RATIO}) \approx \frac{a}{b}$ with $b = 2^m$ being a power-of-two, lifting $\texttt{hd}$ to the ring $\mathbb{Z}_{2^{k+m}}$ for ABY3 and $\mathbb{Z}_{p \cdot 2^m}$ for Shamir can be done without interaction. Consequently, $\texttt{hd}$ can always be computed in the smaller ring/field without performance loss.

Lifting in Power-of-Two-Rings. Since a is different from $b = 2^m$, we have to use a different method for lifting $[\texttt{ml}]_{2^n}$ to $[\texttt{ml}]_{2^{m+n}}$, which will require communication. Our approach involves the following steps. First, we directly interpret the sharing of $\texttt{ml}$ as shares over $\mathbb{Z}_{2^{m+n}}$. This result represents $\texttt{ml}_1 + \texttt{ml}_2 + \texttt{ml}_3$ without reduction modulo 2^n, which is why we need to perform it manually. We do this by extracting the n-th and $(n+1)$-th bit (when interpreting the LSB as the 0-bit) of this value, transforming them into a sharing over $\mathbb{Z}_{2^{n+m}}$ using a bit-injection algorithm to get $[b_n]_{2^{m+n}}$ and $[b_{n+1}]_{2^{n+m}}$ and calculating $[\texttt{ml}]_{2^{m+n}} = [\texttt{ml}']_{2^{n+m}} - 2^n \cdot [b_n]_{2^{m+n}} - 2^{n+1} \cdot [b_{n+1}]_{2^{m+n}}$. Furthermore, one can optimize the bit-injection step to inject into a smaller ring and use Algorithm 13 to perform the multiplication with 2^n to save some bits of communications. The resulting algorithm can be found in Algorithm 14 in Appendix E.

Lifting in Prime Fields. When $\texttt{ml}$ is shared over $\mathbb{F}_p$ instead of $\mathbb{Z}_2^n$, the protocol needs to be slightly adapted. Since we lift $b \cdot \texttt{hd}$ via Algorithm 13 the target ring is $\mathbb{Z}_{p \cdot 2^m}$.

For lifting $[\texttt{ml}]_p$, we again first locally interpret the shares as shares over $\mathbb{Z}_{p \cdot 2^m}$, which is a share of the result of combining the shares without modular reduction. Thus, we again need to correct the share by evaluating $[\texttt{ml'}]_{p \cdot 2^m} - p \cdot [\neg o_1]_{p \cdot 2^m} - p \cdot [\neg o_1 \wedge \neg o_2]_{p \cdot 2^m}$, where o_1 and o_2 are overflow bits. To get the overflow bits, we locally calculate $[\texttt{ml'}]_{p \cdot 2^m} - p$ and $[\texttt{ml'}]_{p \cdot 2^m} - 2 \cdot p$, and extract the $(n+1)$-th bit (when interpreting the LSB as the 0-bit). Both of these extractions can be performed in parallel, whereas the negation of boolean values can either simply be calculated as $1 - x$ without any interaction between the parties, or the negation can directly be injected by switching the inputs m_0, m_1 in Algorithm 17. Since $p < 2^n \Leftrightarrow 2p < 2^{n+1}$ and we are extracting the overflow bits in the ring $\mathbb{Z}_{2^{n+2}}$, we can reduce this statement to $[\texttt{ml'}]_{p \cdot 2^m} - p \cdot [\neg o_1]_{p \cdot 2^m} - p \cdot [\neg o_2]_{p \cdot 2^m}$.

For the bit-injection, we can use the same optimization as in the previous paragraph, i.e., we bit-inject $\neg o_i$ over $\mathbb{Z}_{2^m}$ and use Algorithm 13 to get to $[p \cdot \neg o_i]_{p \cdot 2^m}$. The final algorithm is depicted in Algorithm 15.

Ring Choices. Using Lemma 2, we can set the ring $\mathbb{Z}_t$ in which we calculate the comparison as MSB extraction. Since $l = 12800$, and $b = 2^m$ for the efficient lifting of $\texttt{hd}$ (where m is chosen for enough accuracy of the approximation of the threshold via $\frac{a}{b}$), we can choose $t = 2^{16+m}$ for ABY3 and $t = p \cdot 2^m$ for Shamir with $p = 2^{16} - 17$, since our tests show that the conditions of Lemma 2 are fulfilled for reasonable $m \leq 100$. Furthermore, when we do not choose to use the efficient lifting algorithm using $b = 2^m$ at the cost of having to compute both dot products in the larger ring we choose $t = 2^{16+k}$ for ABY3 and a $(16 + k)$-bit prime for Shamir, where k is the bit-precision of a, b.

5.2 Final MSB Extraction

Similar to Section 4, we can use Algorithm 10 (in Appendix E) to extract the MSB when the modulus is a power-of-two, and Algorithm 12 for other moduli.

Consequently, for k-bit moduli, this step requires $2k-3$ bits of communication in $k-1$ rounds for the ABY3 instantiation, and $3k$ bits of communication in $2k+1$ communication rounds for the Shamir one. If the modulus is $p \cdot 2^m$, we can optimize the circuit to reduce the communication to $3k-m$ bits in $2k-m+1$ communication rounds.

6 Benchmarks

In this section we give some benchmarks for our proposed protocols.[5] Thereby, we focus on comparing different instantiations of the two main parts of the protocol, i.e., the dot product evaluation as well as the comparisons via the MSB-extraction in different rings. On a high-level, these two subprotocols have completely different performance characteristics. First, the dot products have larger memory requirements, are more expensive on the CPU, and do not require network interaction except for the resharing in one communication round in the end. On the other hand, the comparisons (including lifting in MPC), have more communication rounds with comparably small CPU requirements in between.

In the next sections, we focus on different precisions of the a/b approximation of the threshold. First, when setting $b = 2^m = 2^{20}$ we have to set a to be of 20 bit as well for accuracy. Thus, the ring for comparison will extend from 16-bit to 36. For choosing $b \neq 2^m$ we set both a and b to be 16-bit values, thus the ring for comparison is of size 32.

6.1 Dot-Products

The evaluation of the dot-products in MPC is the computationally most expensive part of the protocol. After this first phase, the input vectors of length 12800 get reduced to a single value, which is then used in the comparison phase. We implemented CPU and GPU variants of the dot product and benchmarked the throughput of these variants on various large-scale instances that can be rented from AWS. We evaluate the CPU implementation on Graviton3 instances, and the GPU implementation on both NVIDIA A100s and H100s.

CPU Implementation. The Neoverse1 cores used in Graviton3 instances are based on the ARMv8.4 architecture and support the 16-bit SVE `UDOT` instruction, which can be used to efficiently compute the dot product in 16-bit rings as well as 16-bit prime-fields, since the accumulation takes place in a 64-bit register and modular reduction can be deferred.

For larger rings and fields, no dedicated dot-product instructions exist and an implementation of the dot-product utilizing both SVE and scalar instructions is used instead. We give the performance of the CPU dot product in Table 1. Note that these numbers only concern the inner loop and do not include effects of the cache hierarchy. We therefore expect that the performance of a full production solution will be slightly lower.

GPU Implementation. Even though our proposed protocol for matching Iris Codes has nothing to do with deep learning, we can still leverage the capabilities of modern GPUs to compute the pairwise hamming distances of Iris Codes through large matrix multiplications, the prime use-case for GPU acceleration. Unfortunately, the General Matrix Multiplications (GEMM) algorithms in cuBLAS are not directly applicable to our problem, since the only configuration with suitable precision (at least for the 16 bit setting) would be FP64[6]. The only integer data type supported in cuBLAS is `int8` for the operands

[5] The full implementation will be open-sourced in the future.

[6] https://docs.nvidia.com/cuda/cublas/#cublasgemmex

and `int32` for accumulation of the result. However, by applying decomposition of the operands into smaller data types [TKTW21, WWP22], we are able to use the `int8` Tensor operations (taking care of any deviations due to the use of signed integers with custom CUDA kernels). Notably, for the replicated sharing setting, in which we operate in a 16 bit or 32 bit unsigned integer ring, one only needs 3 or 10 individual `int8 GEMM` operations respectively, since the higher order cross-terms cancel out in the modular reduction. In the Shamir setting, the multiplications have to performed in a prime field and therefore calculation of all 4/16 individual `int8` GEMM operations is required. The benchmarks were conducted using `cuBLAS` and the Rust library `cudarc`[7], and similar to the CPU version, only contains the matrix multiplication and no other parts of the protocol. The results are depicted in Table 1.

Table 1: Performance of dot-product implementations on a 64-core Graviton3 instance vs. Nvidia A100 and H100 GPUs, measured in millions of length 12800 dot-products/s. $\mathbb{F}$`32b` denotes a 32-bit prime field, $\mathbb{Z}$`32b` a 32-bit ring. Calculations in rings take into account the additional dot product required by the replicated sharing.

	$\mathbb{Z}$`16b`	$\mathbb{F}$`16b`	$\mathbb{Z}$`32b`	$\mathbb{F}$`32b`	$\mathbb{Z}$`36b`	$\mathbb{F}$`36b`
Graviton3	49	129	24	21	10	18
A100	1364	2139	496	589	-	-
H100	3146	4960	1162	1395	-	-

Comparing the performance numbers in Table 1 we see that GPUs excel at workloads like dot-products and vastly outperform even CPUs with dedicated dot-product instructions. However, the main metric that is of interest for deployment is the cost-normalized throughput of the instances, and so we compare the 3 different implementations in Figure 1, based on the monthly upfront cost of `hpc7g.16xlarge`, `p4d.24xlarge` and `p5.48xlarge` instances on AWS. Still, the GPU implementation is the most cost-effective solution for the evaluation of the dot-products.

6.2 Comparisons via MSB-Extraction

The second phase of the protocol differs significantly from the dot-product evaluation, since the MPC protocols for the threshold comparison require many communication rounds. We benchmark and compare eight different settings: First, we distinguish between knowing the masks in *plain* (Section 4) and the protocols from Section 5. For the latter, we compare the cases where: (i) both `hd` and `ml` are computed over 16-bit and we lift both values to a 36-bit sharing (using $b = 2^{20}$, dubbed *MPCLift*), (ii) we compute `ml` over 36-bit such that we only have to lift `hd` with $b = 2^{20}$, which can be done for free (*ConstLift*), (iii) we compute both `ml` and `hd` over 32-bit (*NoLift*) and use 16-bit a, b approximations.

Finally, for these four settings we compare sharings in a power-of-two-ring ($\mathbb{Z}$) and with sharings mod p ($\mathbb{F}$). Furthermore, we give the benchmarks for accumulating all results into one shared output bit via an Or-tree evaluation. The benchmarks are given in Table 2 for a batch sizes of 100k `hd` $<$ `ml` $\cdot$ `MATCH_RATIO` evaluations in parallel. For more batch sizes we refer to Appendix D. All benchmarks are obtained on a machine with an AMD Ryzen 9 7950X CPU (4.5 GHz), where each party is executed on an individual thread and the parties are connected via a network connection on the same local host. While a real-world deployment would imply a different networking setup, the performance requirements imply that any cluster of MPC nodes would likely be connected via multiple high-speed networking links in the same data-center, which should not significantly impact the performance of the protocol.

[7] https://github.com/coreylowman/cudarc

Table 2: Comparison of the different protocols for doing the $\texttt{hd} < \texttt{ml} \cdot \texttt{MATCH\_RATIO}$ comparison 100k times in parallel in MPC. The parties are run on a single thread each on the same CPU, with localhost networking. Data gives the communication as amount of kB send per party. Results are averaged over 1000 executions.

Protocol	Runtime ms	Throughput M el/s	Comm kB	Runtime ms	Throughput M el/s	Comm kB
	$\mathbb{Z}$			$\mathbb{F}$		
Plain Mask	1.85	54.1	362	2.60	38.5	625
MPCLift	9.96	10.0	2 238	12.38	8.1	2 925
ConstLift	4.38	22.8	863	5.48	18.2	1 100
NoLift	2.55	39.2	763	3.45	29.0	1 200
Or Tree	0.26	384.6	12	0.26	384.6	12

6.3 Discussion

As already discussed in Section 5.1, the protocol allows for a tradeoff between the work done in the dot-product phase and the comparison phase. Looking at CPU dot-product performance in Table 1 first, we can see that 16-bit rings and fields are much more efficient than 32-bit ones. Furthermore, using Shamir sharing for this step offers additional performance benefits, since the multiplication boils down to a single multiplication with deferred reduction, whereas we need to perform a second dot-product when using the replicated sharing. The ladder doubles the cost for replicated sharing, while also introducing additional memory pressure.

On the other hand, the MPC lifting step required for the comparison phase adds additional overhead, as the throughput of the comparison phase is nearly 4x as large when values are already in the larger field (see Table 2). Additionally, when starting in a prime field, some of the building blocks get more complex due to the need of performing the modular reduction in a binary circuit (see Section 5.1), which manifests in lower throughput and more communication required for $\mathbb{F}$ compared to $\mathbb{Z}$. Finally, the Or-tree evaluation is a very lightweight operation with negligible overhead compared to the rest of the protocol.

When considering the GPU numbers, we can observe that the current MSB-extraction phase is the bottleneck of the protocol, as a single full 64-core Graviton instance is not sufficient to keep up with the throughput of the GPU-accelerated dot-product phase for the 16-bit field (observe that Table 2 reports single-threaded performance). We believe that the current implementation of the comparison phase can be optimized further and are actively working on further speedups.

Finally, regarding the communication overhead, we see that the strategy using MPC lifting requires to send 22/29 bytes of communication per comparison for the ring/field case, respectively. As a single core can produce a throughput of 10/8 million comparisons in the ring/field case, this amounts to 223/236 MB/s bandwidth requirement per core. For a full-scale production deployment managing to process 10 queries per second against a database of 10 million Iris Codes (considering 2 irises per user and also 31 rotations per iris query) the production cluster would require $10M \cdot 10 \cdot 2 \cdot 31 \cdot 29.2$ bytes/s = 181 GB/s of total bandwidth, further highlighting the scale of the problem.

7 Conclusion

In this work we have shown that by utilizing efficient dot-products in *honest-majority* MPC protocols we can improve the performance of secure Iris Code matching significantly. The resulting party-local dot-product workload is also a prime candidate for GPU acceleration, leading to even further speedups. We utilize ideas from ABY3 [MR18] and similar protocols

to switch representations of the data between different rings and (binary) fields to optimize the performance of the threshold comparison compared to naive approaches and thus significantly outperform previous works.

Comparing directly to Janus [ELS+24], we see that even when using a only a single CPU core for the setting of Shamir sharing over $\mathbb{F}_{2^{16}-17}$, the dot-product implementation followed by the `MPCLift` comparison strategy allows us to compare a query Iris Code against over 1 million entries in the database per second. In comparison to Janus, their SHE-Janus solution using somewhat homomorphic encryption manages to compare two query Iris Codes against a database of 8 000 Iris Codes in 40 seconds, which is a throughput of 400 comparisons per second. This is a factor of 2 500 less throughput than our solution, even without considering GPU acceleration. Furthermore, the used Iris Codes are only of length 2048, while we consider Iris Codes of length 12800, allowing for much better accuracy in the matching process. Comparing to the SMC-Janus solution, which is based on garbled circuits, the performance is even worse, with a throughput of about 50 comparisons per second and a communication of 1 GB for comparing two Iris Codes against a database of 1 000 Iris Codes (where our protocol would only communicate a few tens of kilobytes). The main reason for the performance difference to Janus is the fact that our protocol can utilize both the computational efficiency of the dot-product operation as well as its communication efficiency in honest-majority MPC protocols, which is not possible when using garbled circuits. As a consequence, MPC-based solutions are more competitive than solutions based on (somewhat) homomorphic encryption, since both solutions have communication which is independent to the size of the iris codes, while SHE traditionally introduces significantly larger overhead on CPUs.

Future Work. The current implementation is intended mostly as a proof-of-concept prototype and for a full production deployment several issues have to be considered and solved. First, the current performance of GPU-accelerated dot-products has much higher throughput than the threshold comparison phase. Improving the performance of that phase using multithreading and more efficient streaming networking operations is an ongoing engineering effort. We further also investigate whether GPUs could be used for the comparison phase, however, the communication requirements of that phase make this part not as well suited for GPUs as the dot-product step. The full-scale production deployment of this protocol is being actively worked on and we are confident that the current design is sufficient to handle the real-world performance demands.

Acknowledgment

This work was partially supported by the Human Collective Grant Program – Wave 0 by the Worldcoin Foundation.[8]

References

[BCP13] Julien Bringer, Hervé Chabanne, and Alain Patey. Privacy-preserving biometric identification using secure multiparty computation: An overview and recent trends. *IEEE Signal Process. Mag.*, 30(2):42–52, 2013.

[BMW+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium*, pages 991–1008. USENIX Association, 2018.

[8] https://worldcoin.org/community-grants

[Dau02] John Daugman. How iris recognition works. In *ICIP (1)*, pages 33–36. IEEE, 2002.

[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*, pages 2183–2200. USENIX Association, 2021.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

[ELS+24] Kasra Edalatnejad, Wouter Lueks, Justinas Sukaitis, Vincent Graf Narbel, Massimo Marelli, and Carmela Troncoso. Janus: Safe biometric deduplication for humanitarian aid distribution. In *SP*, pages 115–115. IEEE, 2024.

[FN13] Tore Kasper Frederiksen and Jesper Buus Nielsen. Fast and maliciously secure two-party computation using the GPU. In Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*, volume 7954 of *Lecture Notes in Computer Science*, pages 339–356. Springer, 2013.

[Gen09] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, USA, 2009.

[HMSG13] Nathaniel Husted, Steven A. Myers, Abhi Shelat, and Paul Grubbs. GPU and CPU parallelization of honest-but-curious secure two-party computation. In Charles N. Payne Jr., editor, *Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA, December 9-13, 2013*, pages 169–178. ACM, 2013.

[ITK13] Yadigar N. Imamverdiyev, Andrew Beng Jin Teoh, and Jaihie Kim. Biometric cryptosystem based on discretized fingerprint texture descriptors. *Expert Syst. Appl.*, 40(5):1888–1901, 2013.

[JSH+24] Wuxuan Jiang, Xiangjun Song, Shenbai Hong, Haijun Zhang, Wenxin Liu, Bo Zhao, Wei Xu, and Yi Li. Spin: An efficient secure computation framework with gpu acceleration, 2024.

[JW99] Ari Juels and Martin Wattenberg. A fuzzy commitment scheme. In *CCS*, pages 28–36. ACM, 1999.

[KPPS21] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. SWIFT: super-fast and robust privacy-preserving machine learning. In *USENIX Security Symposium*, pages 2651–2668. USENIX Association, 2021.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*, pages 973–990. USENIX Association, 2018.

[MLS+20] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2505–2522. USENIX Association, 2020.

[MPR20] Mahesh Kumar Morampudi, Munaga V. N. K. Prasad, and U. S. N. Raju. Privacy-preserving iris authentication using fully homomorphic encryption. *Multim. Tools Appl.*, 79(27-28):19215–19237, 2020.

[MR18] Payman Mohassel and Peter Rindal. Aby^3: A mixed protocol framework for machine learning. In *CCS*, pages 35–52. ACM, 2018.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[Tam16] Benjamin Tams. Unlinkable minutiae-based fuzzy vault for multiple fingerprints. *IET Biom.*, 5(3):170–180, 2016.

[TKTW21] Sijun Tan, Brian Knott, Yuan Tian, and David J. Wu. Cryptgpu: Fast privacy-preserving machine learning on the GPU. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1021–1038. IEEE, 2021.

[WWP22] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. Piranha: A GPU platform for secure computation. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 827–844. USENIX Association, 2022.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167. IEEE Computer Society, 1986.

[YY13] Jiawei Yuan and Shucheng Yu. Efficient privacy-preserving biometric identification in cloud computing. In *INFOCOM*, pages 2652–2660. IEEE, 2013.

[ZZX+18] Liehuang Zhu, Chuan Zhang, Chang Xu, Ximeng Liu, and Cheng Huang. An efficient and privacy-preserving biometric identification scheme in cloud computing. *IEEE Access*, 6:19025–19033, 2018.

A Missing Proofs

In this section we give the proofs omitted from the main body of the paper.

A.1 Proof of Lemma 1

Proof. `ml` is the dot product of two l-bit vectors, whereas `hd` is calculated as the dot product of two masked bitvectors of size l. Consequently, we can bound the ranges $\texttt{ml} \in [0, l]$ and $\texttt{hd} \in [-l, l]$. Furthermore, since f is a real number between 0 and 1, $f \cdot \texttt{ml}$ has the same bounds as `ml`. Consequently $\lceil f \cdot \texttt{ml} \rceil - \texttt{hd} \in [-l, 2l]$. Now, we need to ensure that $\mathbb{Z}_t$ is big enough to represent this ranges in signed integers, i.e., $l < \frac{t}{2}$ for the negative range as well as $2l < \frac{t}{2}$ for the positive one. Together, we have $l < \frac{t}{4}$. Furthermore, we need to ensure that a positive value $x \in [0, 2l]$ leads to a MSB of 0, which is the case if $2l < 2^{\lceil \log_2(t) \rceil - 1}$ which is already ensured if $l < \frac{t}{4}$. Finally, negative values $x \in [-l, 0)$ need to have the MSB set, which is the case if $l < t - 2^{\lceil \log_2(t) \rceil - 1}$. □

A.2 Proof of Lemma 2

Proof. `ml` is the dot product of two l-bit vectors, whereas `hd` is calculated as the dot product of two masked bitvectors of size l. Consequently, we can bound the ranges $\texttt{ml} \in [0, l]$ and $\texttt{hd} \in [-l, l]$. Consequently, $a \cdot \texttt{ml}$ is in the range $[0, a \cdot l]$ and $b \cdot \texttt{ml}$ is in the range $[-b \cdot l, b \cdot l]$. Thus, $a \cdot \texttt{ml} - b \cdot \texttt{hd} \in [-b \cdot l, (a + b) \cdot l]$. Now, we need to ensure that $\mathbb{Z}_t$ is big enough to represent this ranges in signed integers, i.e., $b \cdot l < \frac{t}{2}$ for the negative range as well as $(a + b) \cdot l < \frac{t}{2}$ for the positive one. Since $a \cdot l \geq 0$, the latter statement implies the first one. Furthermore, we need to ensure that a positive value $x \in [0, (a + b) \cdot l]$ leads to a MSB of 0, which is the case if $(a + b) \cdot l < 2^{\lceil \log_2(t) \rceil - 1}$ which is already ensured if $(a + b) \cdot l < \frac{t}{2}$. Since $a \leq b$ we have $(a + b) \cdot l \leq 2 \cdot b \cdot l$, thus $b \cdot l < \frac{t}{4}$ implies $(a + b) \cdot l < \frac{t}{2}$. Finally, negative values $x \in [-b \cdot l, 0)$ need to have the MSB set, which is the case if $b \cdot l < t - 2^{\lceil \log_2(t) \rceil - 1}$. □

B Details on Masked Bitvectors

We extend logical operations, such as AND and XOR operations, such that they produce `U` at the output if at least one input is `U`. Consequently, an XOR operation between two bits that considers their masks is equal to a multiplication: Let $a, b \in \mathbb{F}_2$ be bits, $m_a, m_b \in \mathbb{F}_2$ be their masks, and $a', b' \in \{-1, 0, 1\}$ be their masked representation, then $a' \cdot b'$ corresponds to $(a \oplus b) \wedge m_a \wedge m_b$. In the original bit-representation, the `CountOnes` operation counts how many bits are set. In the masked bit representation, this operation corresponds to counting how many elements are in the state `T`. Observe that $a'^2 = a' \cdot a'$ results in 1 if the masked input bit a' is either `T` or `F`, i.e., it extracts the mask from the masked value. Using this, one can calculate `CountOnes` on a masked bitvector $\vec{a}'$ as

$$\texttt{CountOnes}(\vec{a}') = \tfrac{1}{2} \cdot \texttt{Sum}(\vec{a}' \odot \vec{a}' - \vec{a}')\,, \tag{2}$$

where $\odot$ is an element-wise multiplication. Looking closer, the subtraction of $\vec{a}'$ from $\vec{a}' \odot \vec{a}'$ removes all `F` states, while adding the `T` states a second time, which gets normalized by the multiplication with 1/2.

Consequently, the hamming distance $\texttt{CountOnes}((\vec{a} \oplus \vec{b}) \wedge \vec{m}_a \wedge \vec{m}_b)$ can be calculated as

$$\begin{aligned}\texttt{CountOnes}(\vec{a}' \odot \vec{b}') &= \tfrac{1}{2} \cdot \texttt{Sum}(\vec{a}' \odot \vec{a}' \odot \vec{b}' \odot \vec{b}' - \vec{a}' \odot \vec{b}') \\ &= \tfrac{1}{2} \cdot (\texttt{Sum}(\vec{a}' \odot \vec{a}' \odot \vec{b}' \odot \vec{b}') - \texttt{Sum}(\vec{a}' \odot \vec{b}')) \\ &= \tfrac{1}{2} \cdot (\texttt{CountOnes}(\vec{m}_a \wedge \vec{m}_b) - \sum_i a'_i \cdot b'_i)\,. \end{aligned} \tag{3}$$

Since one can use $\vec{a}' \odot \vec{a}'$ to extract the mask $\vec{m}_a$ as bitvector, and since an AND gate between bits can be represented by a simple multiplication, $\texttt{Sum}(\vec{a}' \odot \vec{a}' \odot \vec{b}' \odot \vec{b}')$ reduces to $\texttt{CountOnes}(\vec{m}_a \wedge \vec{m}_b)$. Consequently, using masked bitvectors allows us to remove two sums from the hamming distance calculation in Equation (1).

To simplify the comparison operation $\texttt{hd} < \texttt{MATCH\_RATIO} \cdot \texttt{ml}$ from Algorithm 1, where $\texttt{hd} = \texttt{CountOnes}((\vec{c} \oplus C_{\texttt{DB}}[i]) \wedge \vec{m} \wedge M_{\texttt{DB}}[i])$ and $\texttt{ml} = \texttt{CountOnes}(\vec{m} \wedge M_{\texttt{DB}}[i])$, one can rewrite it using the masked bit representation:

$$\begin{aligned} & \texttt{hd} < \texttt{MATCH\_RATIO} \cdot \texttt{ml} \\ \Leftrightarrow\ & \texttt{CountOnes}((\vec{c} \oplus C_{\texttt{DB}}[i]) \wedge \vec{m} \wedge M_{\texttt{DB}}[i]) < \texttt{MATCH\_RATIO} \cdot \texttt{ml} \\ \Leftrightarrow\ & \frac{1}{2} \cdot (\texttt{ml} - \sum_j c'_j \cdot C'_{\texttt{DB}}[i]_j) < \texttt{MATCH\_RATIO} \cdot \texttt{ml} \\ \Leftrightarrow\ & \langle c', C'_{\texttt{DB}}[i] \rangle > (1 - 2 \cdot \texttt{MATCH\_RATIO}) \cdot \texttt{ml}. \end{aligned}$$

C Introduction on ABY3

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}) (wrapping using $(i \mod 3) + 1$). 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, they can compute an additive share of the result without communication. Namely

$$z_i = x_i \cdot y_i + x_{i-1} \cdot y_i + x_i \cdot y_{i-1} + r_i\,, \tag{4}$$

where r_i is a freshly generated random share of 0 required for re-randomization which can be produced without communication in ABY3 [MR18]. 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 re-establish the replication after each individual multiplication $[x_i] \cdot [y_i]$, but can perform the summation on the additive shares and finally re-share the result. Consequently, a dot product requires the same amount of communication as a multiplication in ABY3, i.e., it is independent of the length of the vectors $|[\vec{x}]| = |[\vec{y}]|$.

Observe that the multiplication can be rewritten as

$$\begin{aligned} z_i &= x_i \cdot y_i + x_{i-1} \cdot y_i + x_i \cdot y_{i-1} + r_i \\ &= (x_i + x_{i-1}) \cdot (y_i + y_{i-1}) - x_{i-1} \cdot y_{i-1} + r_i\,. \end{aligned} \tag{5}$$

Thus, since we directly use the input shares in dot-products, we can do the following optimization. First, one can pre-process the share (x_i, x_{i-1}) as $(x'_i, x'_{i-1}) = (x_i + x_{i-1}, x_{i-1})$. Then, multiplication reduces to two plain multiplications plus re-randomization:

$$\begin{aligned} z_i &= x_i \cdot y_i + x_{i-1} \cdot y_i + x_i \cdot y_{i-1} + r_i \\ &= x'_i \cdot y'_i - x'_{i-1} \cdot y'_{i-1} + r_i \,. \end{aligned} \tag{6}$$

Consequently, a dot-product of shared values reduces to two plain dot-products and resharing including re-randomization.

D Additional Benchmarks and Figures

Table 3 gives the benchmarks from Section 6.2 with more different batch sizes.

Table 3: Comparison of the different protocols for doing the $\texttt{hd} < \texttt{ml} \cdot \texttt{MATCH\_RATIO}$ comparison in MPC. The parties are run on a single thread each on the same CPU, with localhost networking. Data gives the communication as amount of kB send per party. Batch Size indicates the number of comparisons computed. Results are averaged over 1000 executions.

Protocol	Runtime ms	Throughput M el/s	Comm kB	Runtime ms	Throughput M el/s	Comm kB
	$\mathbb{Z}$			$\mathbb{F}$		
	Batch Size = $50k$					
Plain Mask	0.95	52.6	181	1.38	36.2	313
MPCLift	4.99	10.0	1 119	5.90	8.5	1 463
ConstLift	1.61	31.1	431	2.58	19.4	550
NoLift	1.54	32.5	381	2.06	24.3	600
Or Tree	0.21	238.1	6	0.21	238.1	6
	Batch Size = $100k$					
Plain Mask	1.85	54.1	362	2.60	38.5	625
MPCLift	9.96	10.0	2 238	12.38	8.1	2 925
ConstLift	4.38	22.8	863	5.48	18.2	1 100
NoLift	2.55	39.2	763	3.45	29.0	1 200
Or Tree	0.26	384.6	12	0.26	384.6	12
	Batch Size = $250k$					
Plain Mask	4.94	50.6	906	7.04	35.5	1 563
MPCLift	24.30	10.3	5 594	31.36	8.0	7 313
ConstLift	10.98	22.8	2 156	15.06	16.6	2 750
NoLift	6.82	36.7	1 906	9.38	26.7	3 000
Or Tree	0.35	714.3	31	0.35	714.3	31
	Batch Size = $500k$					
Plain Mask	10.24	48.8	1 813	14.08	35.5	3 125
MPCLift	49.18	10.2	11 188	58.80	8.5	14 625
ConstLift	22.51	22.2	4 313	29.66	16.9	5 500
NoLift	16.71	29.9	3 813	18.71	26.7	6 000
Or Tree	0.38	1315.8	62	0.38	1315.8	62

E Missing Algorithms

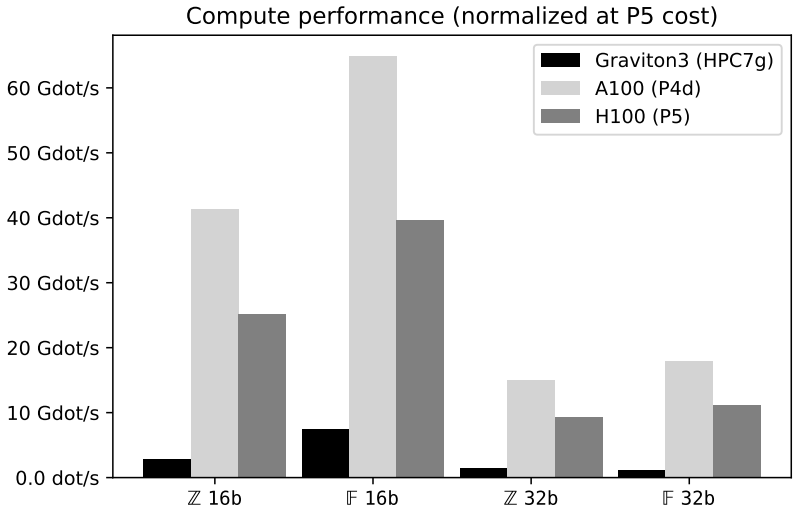

Figure 1: Cost-normalized throughput of the different dot-product implementations.

Algorithm 4 $\mathsf{InpLocal}(x, i, 2^k) \to ([x]_{2^k})$.

Input: x is known to parties $i, i+1$
Parties $i, i+1$ interpret x as an element of $\mathbb{Z}_{2^k}$.
Party i defines its replicated shares as $x_i = x$ and $x_{i-1} = 0$.
Party $i+1$ defines its replicated shares as $x_{i+1} = 0$ and $x_i = x$.
Party $i+2$ defines its replicated shares as $x_{i+2} = 0$ and $x_{i+1} = 0$.
return $[x]_{2^k}$

Algorithm 5 $\mathsf{ShareSplit}([x]_{2^k}) \to ([x_1[i]]_2, [x_2[i]]_2, [x_3[i]]_2)_{i\in[0,k)}$.

Parties $i, i+1$ interpret x_i as an element of $(\mathbb{Z}_2)^k$, i.e., as a vector of k bits.
$[x_1[j]]_2 \leftarrow \mathsf{InpLocal}(x_1[j], 1, 2)$ for all $j \in [0, k)$.
$[x_2[j]]_2 \leftarrow \mathsf{InpLocal}(x_2[j], 2, 2)$ for all $j \in [0, k)$.
$[x_3[j]]_2 \leftarrow \mathsf{InpLocal}(x_3[j], 3, 2)$ for all $j \in [0, k)$.
return $([x_1[i]]_2, [x_2[i]]_2, [x_3[i]]_2)_{i\in[0,k)}$

Algorithm 6 $\mathsf{FA}([a]_2, [b]_2, [c]_2) \to ([s]_2, [o]_2)$. This algorithm computes a full adder with inputs a, b, carry input c, carry output o and result s.

$[t_1]_2 \leftarrow [a]_2 + [c]_2$
$[t_2]_2 \leftarrow [b]_2 + [c]_2$
$[s]_2 \leftarrow [t_1]_2 + [b]_2$
$[o]_2 \leftarrow [t_1]_2 \cdot [t_2]_2 + [c]_2$
return $([s]_2, [o]_2)$

Algorithm 7 $\mathsf{BinAdd}(([a[m-1]]_2, \ldots, [a[0]]_2), ([b[m-1]]_2, \ldots, [b[0]]_2)) \to ([c[m]]_2, \ldots, [c[0]]_2)$. Binary addition of two values (ripple-carry adder).

$[s[0]]_2 = [a[0]]_2 + [b[0]]_2$
$[c]_2 = [a[0]]_2 \cdot [b[0]]_2$
for i in $1..m$ **do**
    $([c]_2, [s[i]]_2) \leftarrow \mathsf{FA}([a[i]]_2, [b[i]]_2, [c[i]]_2)$
return $([c]_2, [s[m-1]]_2, \ldots, [s[0]]_2)$

Algorithm 8 $\mathsf{FS}([a]_2, [b]_2, [c]_2) \to ([s]_2, [o]_2)$. This algorithm computes a full subber with inputs a, b, carry input c, carry output o and result s.

$[t_1]_2 \leftarrow [a]_2 + [b]_2$
$[t_2]_2 \leftarrow [b]_2 + [c]_2$
$[s]_2 \leftarrow [t_1]_2 + [c]_2$
$[o]_2 \leftarrow [t_1]_2 \cdot [t_2]_2 + [c]_2$
return $([s]_2, [o]_2)$

Algorithm 9 $\mathsf{BinSub}(([a[m-1]]_2, \ldots, [a[0]]_2), ([b[m-1]]_2, \ldots, [b[0]]_2)) \to ([c[m-1]]_2, \ldots, [c[0]]_2)$. Binary subtraction of two values (ripple-borrow subtractor).

$[s[0]]_2 = [a[0]]_2 + [b[0]]_2$
$[c]_2 = [s[0]]_2 \cdot [b[0]]_2$
for i in $1..m$ **do**
    $([c]_2, [s[i]]_2) \leftarrow \mathsf{FS}([a[i]]_2, [b[i]]_2, [c[i]]_2)$
return $([s[m-1]]_2, \ldots, [s[0]]_2)$

Algorithm 10 $\mathsf{BitExtract}([x]_{2^k}, I) \to ([x[i]]_2)_{i \in I}$. This algorithm extracts the bits with indices in I from the shared value x.

$([x_1[i]]_2, [x_2[i]]_2, [x_3[i]]_2)_{i \in [0,k)} \leftarrow \mathsf{ShareSplit}([x]_{2^k})$.
$m \leftarrow \max i \in I$
$([c[i]]_2, [s[i]]_2) \leftarrow \mathsf{FA}([x_1[i]]_2, [x_2[i]]_2, [x_3[i]]_2)$ for all $i \in [0, m]$.
$([r[m]]_2, \ldots [r[0]]_2) \leftarrow \mathsf{BinAdd}(([c[m-1]]_2, \ldots, [c[0]]_2, 0), ([s[m]]_2, \ldots, [s[0]]_2))$.
return $([r[i]]_2)_{i \in I}$

Algorithm 11 $\mathsf{CMux}([c]_2, [a]_2, [b]_2) \to [d]_2$.

$[d]_2 \leftarrow [b]_2 + [c]_2 \cdot ([a]_2 - [b]_2)$
return $[d]_2$

Algorithm 12 $\mathsf{BitExtractP}([x]_p, I) \to ([x[i]]_2)_{i \in I}$. This algorithm extracts the bits with indices in I from the shared value x, for a sharing that is not over a ring of the form 2^k. Let $\ell = \lceil \log_2(p) \rceil$.

Party 2 computes $x_{1,2} \leftarrow x_1 + x_2 \mod p$. Let $x_{1,2}[i]$ represent its i-th bit.
$[x_{1,2}[i]]_2 \leftarrow \mathsf{Input}([x_{1,2}[i]], 2, 2)$ for all $i \in [0, \ell)$.
$[x_3[i]]_p \leftarrow \mathsf{InpLocal}(x_3[i], 3, 2)$ for all $i \in [0, \ell)$.
$([c[\ell]]_2, \ldots [c[0]]_2) \leftarrow \mathsf{BinAdd}(([x_{1,2}[\ell-1]]_2, \ldots, [x_{1,2}[0]]_2), ([x_3[\ell-1]]_2, \ldots, [x_3[0]]_2))$.
$([d[\ell]]_2, \ldots [d[0]]_2) \leftarrow \mathsf{BinSub}(([c[\ell]]_2, \ldots, [c[0]]_2), (0, p[\ell-1], \ldots, p[0]))$.
$[r[i]]_2 \leftarrow \mathsf{CMux}(d[\ell]_2, [c[i]]_2, [d[i]]_2)$, for all $i \in I$.
return $([r[i]]_2)_{i \in I}$

Algorithm 13 $\mathsf{ConstLift}([a]_c, d) \to [ac]_{c \cdot d}$.

Party i interprets a_i as an element of $\mathbb{Z}_{c \cdot d}$ and computes $b_i = a_i \cdot d$.
return $[b]_{c \cdot d}$

Algorithm 14 $\mathsf{Lift}([x]_{2^k}, 2^{k+m}) \to [x]_{2^{k+m}}$.

Parties $i, i+1$ interpret x_i as an element of $\mathbb{Z}_{2^{k+m}}$
$[x_1]_{2^{k+m}} \leftarrow \mathsf{InpLocal}(x_1, 1, 2^{m+k})$.
$[x_2]_{2^{k+m}} \leftarrow \mathsf{InpLocal}(x_2, 2, 2^{m+k})$.
$[x_3]_{2^{k+m}} \leftarrow \mathsf{InpLocal}(x_3, 3, 2^{m+k})$.
$[x_1 + x_2 + x_3]_{2^{k+m}} \leftarrow [x_1]_{2^{k+m}} + [x_2]_{2^{k+m}} + [x_3]_{2^{k+m}}$.
$([a]_2, [b]_2) \leftarrow \mathsf{BitExtract}([x_1 + x_2 + x_3]_{2^{k+m}}, \{k+1, k\})$.
$[a]_{2^{m-1}} \leftarrow \mathsf{BitInject}([a]_2, 2^{m-1})$.
$[b]_{2^m} \leftarrow \mathsf{BitInject}([b]_2, 2^m)$.
$[2^{k+1}a]_{2^{k+m}} \leftarrow \mathsf{ConstLift}([a]_{2^{m-1}}, 2^{k+1})$.
$[2^k b]_{2^{k+m}} \leftarrow \mathsf{ConstLift}([b]_{2^m}, 2^k)$.
return $[x_1 + x_2 + x_3]_{2^{k+m}} - [2^{k+1}a]_{2^{k+m}} - [2^k b]_{2^{k+m}}$

Algorithm 15 $\mathsf{LiftP}([x]_p, p2^m) \to [x]_{p2^m}$. Let $\ell = \lceil \log_2(p) \rceil$.

Parties $i, i+1$ interpret x_i as an element of $\mathbb{Z}_{p2^m}$
$[x_1]_{p2^m} \leftarrow \mathsf{InpLocal}(x_1, 1, p2^k)$.
$[x_2]_{p2^m} \leftarrow \mathsf{InpLocal}(x_2, 2, p2^k)$.
$[x_3]_{p2^m} \leftarrow \mathsf{InpLocal}(x_3, 3, p2^k)$.
$[x_1 + x_2 + x_3]_{p2^m} \leftarrow [x_1]_{p2^m} + [x_2]_{p2^m} + [x_3]_{p2^m}$.
$([a]_2) \leftarrow \neg\mathsf{BitExtract}([x_1 + x_2 + x_3]_{p2^{k+m}} - p, \{\ell + 1\})$.
$([b]_2) \leftarrow \neg\mathsf{BitExtract}([x_1 + x_2 + x_3]_{p2^{k+m}} - 2p, \{\ell + 1\})$.
$[a]_{2^m} \leftarrow \mathsf{BitInject}([a]_2, 2^m)$.
$[b]_{2^m} \leftarrow \mathsf{BitInject}([b]_2, 2^m)$.
$[pa]_{p2^m} \leftarrow \mathsf{ConstLift}([a]_{2^m}, p)$.
$[pb]_{p2^m} \leftarrow \mathsf{ConstLift}([b]_{2^m}, p)$.
return $[x_1 + x_2 + x_3]_{p2^m} - [pa]_{p2^m} - [pb]_{p2^m}$

Algorithm 16 3-Party OT: $\mathsf{3OT}((m_0, m_1), c, c) \to (\bot, m_c, \bot)$. (Party 1 is the OT sender, Party 2 is the OT receiver, Party 3 is the OT helper. Let $m_i \in \mathbb{Z}_{2^k}$.) Based on the 3-party OT in [MR18].

Input: Party i holds $\mathsf{seed}_i, \mathsf{seed}_{i-1}$
Parties $1, 3$ generate $w_0, w_1 \leftarrow\!\$\ \mathsf{RNG.Gen}(\mathsf{seed}_3, k)$
Party 1: $k_0 \leftarrow w_0 \oplus m_0$, $k_1 \leftarrow w_1 \oplus m_1$
Party 1 sends k_0, k_1 to Party 2
Party 3 sends w_c to Party 2
Party 2: $m_c \leftarrow w_c \oplus k_c$ and outputs m_c

Algorithm 17 $\mathsf{BitInject}([x]_2, 2^k) \to [x]_{2^k}$. Based on [MR18].

Input: Party i holds $\mathsf{seed}_i, \mathsf{seed}_{i-1}$
Parties $1, 2$ generate $c_1 \leftarrow\!\$\ \mathsf{RNG.Gen}(\mathsf{seed}_1, k)$
Parties $1, 3$ generate $c_3 \leftarrow\!\$\ \mathsf{RNG.Gen}(\mathsf{seed}_3, k)$
Party 1: $m_i \leftarrow (i \oplus x_1 \oplus x_3) - c_1 - c_3$ for $i \in \{0, 1\}$.
Execute $\mathsf{3OT}((m_0, m_1), x_2, x_2) \to (\bot, m_{x_2}, \bot)$
Party 2 sets $c_2 \leftarrow m_{x_2}$ and sends c_2 to Party 3.
return $[c]_{2^k}$