**Funshade: Function Secret Sharing for Two-Party Secure Thresholded Distance Evaluation**

Alberto Ibarrondo  
Copper.co  
Sophia Antipolis, France  
ibarrondo@eurecom.fr

Hervé Chabanne  
Idemia & Telecom Paris  
Paris, France

Melek Önen  
EURECOM  
Sophia Antipolis, France

**Abstract**

We propose a novel privacy-preserving, two-party computation of various distance metrics (e.g., Hamming distance, Scalar Product) followed by a comparison with a fixed threshold, which is known as one of the most useful and popular building blocks for many different applications including machine learning, biometric matching, etc. Our solution builds upon recent advances in function secret sharing and makes use of an optimized version of arithmetic secret sharing. Thanks to this combination, our new solution named Funshade is the first to require only one round of communication and two ring elements of communication in the online phase, outperforming all prior state-of-the-art schemes while relying on lightweight cryptographic primitives. Lastly, we implement our solution from scratch in portable C and expose it in Python, testifying its high performance by running secure biometric identification against a database of 1 million records in ~10 seconds with full correctness and 32-bit precision, without parallelization.

**Keywords**

Function Secret Sharing, Secure Two Party Computation, Scalar Product, Hamming Distance

1 INTRODUCTION

The computation of privacy-preserving distance metrics $d_{\text{dist}}(x, y)$ between two vectors $x, y$ followed by a comparison with a threshold $\theta$ is a very popular building block in many applications in need of privacy protection, including machine learning (e.g., k-nearest neighbors [68]), linear regression [35]), biometrics (e.g., biometric authentication [46, 57], biometric identification [32]) etc.

The literature counts many solutions based on various cryptographic techniques that allow computation over sensitive data while preserving its privacy: Secure Multiparty Computation (MPC) (garbled circuits [65], secret sharing (SS) [39, 60]) to split the distance computation across multiple entities [19, 29, 33], Fully Homomorphic Encryption (FHE) [23, 34, 36] supporting addition and multiplication between ciphertexts [4, 7, 22], and Functional Encryption (FE) [2, 9] as a public-key encryption scheme that supports evaluation of scalar products when decrypting the ciphertexts [3, 10].

However, not all operations are born equal. While linear operations are widely covered by all the privacy-preserving techniques, the protection of non-linear operations including the comparison to a threshold $\theta$ is much harder to attain. Computing this non-linear operation with most MPC primitives is often communication intensive (e.g., [29, 64]) both in terms of communication size and in number of rounds; FHE-based techniques must resort to computation-intensive algorithms [24, 42]; and FE-based techniques are limited to linear function evaluations. Luckily, recent solutions [11, 14, 58] show a considerable improvement to securely realize the comparison to $\theta$ by resorting to Function Secret Sharing (FSS) primitives.

In [15], the authors study the computation of distance metrics. They propose GSHADE, a decomposition of each metric into a combination of local single-input functions and a cross-product, and preserve the privacy of these operations via Oblivious Transfer [54]. We draw inspiration from the family of distance metrics covered in GSHADE. Integrating FSS-based threshold comparison primitives from [11] with an optimized version of Secret Sharing [53] in a two-party computation (2PC) protocol to perform privacy-preserving distance metric computations with a subsequent comparison to $\theta$.

To summarize our contributions, our solution:

- requires just one round of communication in the online phase, lowering the communication costs with respect to the two-round state-of-the-art solutions from AriaNN [58] and Boyle et al. [11] by merging the communication required for the scalar product with that of the comparison to $\theta$;
- sends two ring elements only in the online phase, reducing the communication size of previous solutions by a factor of 2 (for input vectors with $l$ elements),
- features 100% correctness in the comparison result, as opposed to [58],
- is implemented and open-sourced in a portable C module with lightweight wrapping to high-level languages such as Python, Rust or Golang,
- is extensively tested, showcasing its high efficiency and correctness in a variety of scenarios (LAN/WAN, Authentication/Identification), requiring less than 300ms for realistic biometric identification against 5000 identities with full 32-bit precision, and ~10s against 1 million identities.

The paper is outlined as follows. Section 2 describes the preliminaries, the distance metrics we consider in this work and some applications. Section 3 details the proposed solution, including its security analysis. Section 4 tackles the implementation and experiments. Section 5 addresses previous work and positions our contribution, wrapping up with the conclusions in Section 6.
2 PRELIMINARIES

Notation
We use bold letters to denote vectors (e.g., \( \mathbf{x} \), \( \mathbf{y} \)) and plain letters (e.g., \( a \), \( k \)) for scalars. \( x^{(i)} \) denotes the \( i \)th element of vector \( x \). For convenience we omit the \( (i) \) subscript in elements-wise additions and multiplications of the form \( \Sigma [x^{(i)} + b^{(i)} + \ldots] \). We write \( \mathbf{a} \cdot \mathbf{b} = c \) to denote the element-wise multiplication of two vectors where \( c^{(i)} = a^{(i)} b^{(i)} \), and \( \mathbf{a}^T \mathbf{b} \) to denote the inner (scalar) product between two vectors.

\( P_0, P_1 \) denote the two computing parties in the 2PC paradigm. We generalize behavior common to these two computing parties by resorting to \( P_j \), where \( j \in \{0, 1\} \). Similarly, we reserve \( R_{\text{setup}} \) to indicate an entity in our scenario playing a role with a certain description, e.g., \( R_{\text{setup}} \) for the entity in charge of the setup, \( R_{\text{ins}} \) for the entity holding the input vector \( x \). We write \( P_j \geq R_{\text{descr}} \) to denote that party \( P_j \) takes on a role \( R_{\text{descr}} \). We use \( q \leftarrow 4 \) to set a local variable \( q \) to 4, and \( P_a : \text{Send} q \Rightarrow P_b \) for party \( a \) sending value \( q \) to party \( b \). We note \( U(5) \) as the uniform random distribution in the set \( S \), and write \( r \leftarrow U(5) \) to indicate sampling that distribution and assigning the sample to \( r \). We employ \( 1_{x \in A} \) to denote the indicator function (e.g., \( 1_{x > 5} = 1 \iff x > 5 \)):

\[
1_{x \in A} = 1_{A}(x) = \begin{cases} 
1 & \text{if } x \in A, \\
0 & \text{if } x \notin A.
\end{cases}
\]

As a special case of indicator function, the unit step function is defined as \( 1_{x \in \mathbb{Z}} = 1_{x \geq 0} \). We implicitly consider a two’s complement encoding to map between signed and unsigned \( n \)-bit integers, a bijective mapping between \([−2^{n−1}, 2^{n−1}−1] \) and \([0, 2^n−1] \) by applying \( \text{mod} 2^n \), where the interval of negative values \([−2^{n−1}, −1] \) is mapped to the upper half of the unsigned interval \([2^{n−1}, 2^n−1] \). As such, the unit step function for \( n \)-bit integers corresponds to \( 1_{x \in \mathbb{Z}_{−}} = 1_{0 < x < 2^{n−1}−1.} \)

We write \( (x) \) to indicate that value \( x \) is arithmetically secret shared into shares \((x_0, x_1)\) among computing parties \((P_0, P_1)\) such that \( P_0 \) holds the share \( x_0 \) and \( P_1 \) holds the share \( x_1 \). Likewise, we write \( (x) \) to indicate that value \( x \) is II-secret shared (Section 2.1.2) into three shares \((\Delta_x, \delta_{\text{desc}}, \delta_x1)\), where both parties \((P_0, P_1)\) hold \( \Delta_x \) and each party \( P_j \) holds \( \delta_x \).

2.1 Multi-Party Computation
Secure multi-party computation (or MPC) [6, 21, 39, 65] allows two or more parties to compute any mathematical function on private inputs without revealing anything but the output of the function. Typically, MPC is instantiated in the preprocessing model, where specially crafted randomness is generated in an offline input-independent phase from either a trusted dealer or via an offline interaction, and then this randomness is used in the online phase to compute the function, once the inputs are known. This two-phase approach yields considerable performance benefits. Some examples of this correlated randomness include Beaver multiplication triples [5] and garbled circuit preprocessing [29, 65].

When used to evaluate circuits based on only binary or only arithmetic interactions, MPC protocols present very fast online execution. However, applications such as biometrics or machine learning require a combination of linear operations (additions and multiplications over a large ring) and non-linear operations such as integer comparison or truncation. The cost of blindly implementing these two types of operations with only one MPC circuit type can be prohibitively high. To address this, many works have tackled mixed-mode MPC to provide efficient conversions between arithmetic and binary domains, supporting both linear and non-linear operations [19, 29, 50, 53]. Yet, these conversions often entail a hefty communication overhead in the online phase.

In line with the TinyTable protocol [27] to secret share truth tables in a succinct manner, Boyle et al. propose a very promising approach [11, 14] based on Function Secret Sharing (FSS) [12, 13]. Offering the same online communication and round complexity for non-linear function evaluations as for pure arithmetic computations in arithmetic-only circuits, FSS relies on fast symmetric cryptography primitives to also yield fast online evaluation.

The present work will benefit from standard arithmetic secret sharing techniques [5], more evolved secret sharing techniques emanating from research in mixed-mode operations [53] and modern FSS approaches [11] to achieve a lightweight and highly communication efficient biometric matching protocol. As such, we now delve into the details of these techniques.

2.1.1 Additive Secret Sharing. Secret sharing is a cryptographic primitive that allows a secret \( x \) to be shared among \( n \) parties, such that any \( t \) of them can reconstruct the secret. The secret sharing scheme is defined by \( k \), the number of parties, and threshold \( t \), minimum number of parties required to reconstruct the secret. In the domain of two-party computation (2PC), the number of parties is \( n = 2 \) and the threshold is \( t = 2 \). This work focuses on 2PC arithmetic secret sharing in rings (shortened to SS for convenience), where a secret value \( x \) is split into two random shares \( x_0 \) and \( x_1 \) such that \( x = x_0 + x_1 \mod N \), with \( N \) being the ring size. The shares are distributed to the two computing parties such that party \( P_j \) receives the share \( x_j \). With this sharing scheme, the two parties can perform local addition/subtraction of two secret shares. Additionally, parties can resort to Beaver’s multiplication triples [5] to perform multiplication at the cost of one round of communication:

\[
\text{SS. add: } \text{Online}(P_0, P_1): \langle x + y \rangle = \langle x \rangle ± \langle y \rangle
\]

\[
\text{SS. mult: Online}(R_{\text{setup}}): \langle a \rangle , \langle b \rangle \sim U(2^n) \quad (c) \leftarrow (a \cdot b)
\]

\[
\text{Send}(a_j, b_j) \Rightarrow P_j
\]

\[
\text{Online}(P_0, P_1): \langle x 

\cdot y \rangle = \langle b \rangle (x-a) + \langle a \rangle (y-b) + \langle c \rangle + \langle x-a \rangle(y-b)
\]

(1)

At the end of the computation, the resulting secret shared value can be reconstructed by sending both shares to a chosen party \( P_{\text{res}} \), to add the two shares together and reconstruct the result. We work with \( N = 2^n \) for values of \( n \in [8, 16, 32, 64] \) to benefit from a considerable speedup when dealing with \( n \)-bit modular arithmetic thanks to native integer types present in modern computers.

Of special interest for this work, computing a scalar product \( x^T y = \sum_{i=1}^t x^{(i)} \cdot y^{(i)} \) with SS requires sending 2 terms per multiplication, for a total of \( 2t \) values sent.

2.1.2 II-Secret Sharing. Originally inspired by ASTRA [20] in the 3PC scenario, ABY2.0 [53] introduced a novel way to perform
additive secret sharing in 2PC, where a value $x$ is split into three random shares $(\Delta_x, \delta_x, \delta_1)$ such that $\Delta_x = x + \delta_0 + \delta_1 \mod N$. The $\delta$-shares $\delta_x$ are distributed to each computing party $P_j$ forming an arithmetic secret sharing $(\delta_x)$ of $\Delta_x = \delta_0 + \delta_1$ while the $\Delta$-share $\Delta_x$ is held by both parties at once. We name this sharing scheme as II-secret sharing due to the "horizontally" mutual $\Delta$-share and the two "vertically" separated $\delta$-shares, and denote the II-sharing of value $x$ as $\langle x \rangle$. The II-sharing scheme allows local addition/subtraction, and multiplication at the cost of one round of communication. The essential difference with respect to the SS scheme is that the $\delta$-shares can be precomputed (leaving only the $\Delta$-share to be determined in the online phase), and thus carry extra correlation that was not possible with standard SS. The main arithmetic operations in IISS are defined as follows:

IISS.add: Online($P_0, P_1$): $\langle x \pm y \rangle = \langle x \rangle \pm \langle y \rangle$

IISS.mult: Offline($R_{setup}$): $(\langle \delta_x \rangle, \langle \delta_y \rangle, \langle \delta_\beta \rangle) \sim U_{\{\pm2\}}^N$

$$\langle \delta_{xy} \rangle \leftarrow \langle \delta_x \cdot \delta_y \rangle$$

Send $(\delta_{xyj}, \delta_xj, \delta_yj) \Rightarrow P_j$

Online($P_0, P_1$): $(x \cdot y) \equiv (z) \leftarrow j : \Delta_x \Delta_y - \Delta_x \langle \delta_y \rangle - \Delta_y \langle \delta_x \rangle + \langle \delta_\beta \rangle$

$$\langle \Delta_\beta \rangle \equiv (z) \pm \langle \delta_\beta \rangle$$

Send $(\Delta_\beta) \Rightarrow P_{1-j}$

$$\langle z \rangle \equiv \langle \Delta_\beta + \Delta_{21} \rangle \pm \langle \delta_\beta \rangle$$

(2)

Crucially, the online phase of the II-sharing multiplication first computes a local arithmetic sharing of the result, and then uses one round of communication to convert the result back into II-shares. As promptly explained in [53], this moves the communication from the multiplication inputs to the multiplication outputs, which yields significant advantages in terms of communication size for operations such as the scalar product: computing a scalar product $x^T y = \sum_{i=1}^n x(i) \cdot y(i)$ with IISS requires sending 2 values only for the entire operation, thus reducing the communication size by a factor of 1 with respect to SS.

2.1.3 Function Secret Sharing. A 2PC Functional Secret Sharing (FSS) scheme [12, 13] for a function family $F$ splits a function $f \in F$ into two additive shares $(f_0, f_1)$, such that each $f_j$ hides $f$ and $f_j(x) = f(x)$ for every input $x$. Beyond trivial solutions such as secret-sharing the truth-table of $f$, FSS schemes seek succinct descriptions of $f_j$ (function keys $k_0, k_1$) with efficient online execution. Since both function shares must be evaluated on the same value $x$, this value must be made public to both computing parties $P_j$. To maintain input data privacy, a random mask $r$ is added to the secret input $x$, so that the opened value $\hat{x} = x + r$ completely hides $x$ before using it as input to the FSS evaluation. In order to obtain full correctness on the function evaluation with respect to $f(x)$, the class of functions $F$ is restricted to $f(x) = f(x + r)$, where the mask is known by the dealer and used for the key generation.

For addition and multiplication gates over a ring $\mathbb{Z}_n$, the FSS gates correspond to Beaver’s protocol [5]. A much more interesting case arises in [11, 14], where non-linear operations including zero-test, integer comparison or bit decomposition are efficiently constructed using a small number of invocations of FSS primitives. Luckily, these FSS gates make a black-box use of any secure pseudorandom generator (PRG), yielding short keys and fast implementations based on AES.

Grounded on the MPC preprocessing model, a FSS gate is composed of two algorithms:

- Gen$(1^\lambda, f) \rightarrow (k_0, k_1)$ is a PPT key generation algorithm that, given the security parameter $\lambda$ and the description of a function $f : \mathbb{Z} \rightarrow \mathbb{Z}$, outputs a pair of function keys $(k_0, k_1)$ containing the descriptions for $f_0, f_1$ and the input mask shares $r_0, r_1$ respectively.

- Eval$(j, k_j, x) \rightarrow f(x)$ is a polynomial-time deterministic algorithm that, given the party index $j$, the function key $k_j$ and the masked input $x$ outputs an additive share $f_j(x)$, such that $f_0(x) + f_1(x) = f(x)$.

As central building block of many FSS gates, we recall the concept of Distributed Comparison Function (DCF) (Section 3 of [11]) to be a function $f_{\alpha, \beta}(x)$ outputting $\beta$ if $x > \alpha$ and zero otherwise. Built on top of two evaluations of DCF, [11] later proposes a FSS gate for Interval Containment (IC) computing $f_{\alpha, \beta}(x) = 1_{x \in [p, q]}$ (Section 4.1 of [11]). To compute the unit step function of a signed integer, it suffices to employ their construction (detailed in Figure 3 of [11]) setting $p = 0$ and $q = 2^n - 1$, obtaining $1_{p < x < q}$. For convenience, we detail this FSS gate instantiation in Protocols 1 (key generation) and 2 (evaluation), keeping the DCF calls to the original protocol in [11].

Algorithm 1: FSS.Gen$^{IC} (\lambda, n, r) \rightarrow k_0^{IC}, k_1^{IC}$

Inputs: $\lambda$: computational security parameter.

$r$: Mask for the input to the function.

Output: $(k_0, k_1)$: preprocessing keys, to send to $P_0, P_1$ respectively.

$(\delta_\alpha, \delta_\beta)$ - shares of input vectors, to send to $P_{inx}, P_{iny}$ (input owners) respectively.

Note: All arithmetic operations ($+,-,\cdot$) are defined in $\mathbb{Z}_n$, thus their results are susceptible to "overflow" due to modular reduction.

Define the interval $[p, q]$ for sign extraction:

1. $p \leftarrow 0$; $q \leftarrow 2^n - 1$

2. Generate a DCF for $\gamma$, an arbitrary value above the two interval limits:

3. $(k_{\gamma, 0}, k_{\gamma, 1}) \leftarrow$ FSS.Gen$^{IC} (\lambda^2, \gamma + r, 1, U_{\{\pm2\}})$

4. Generate the correction terms$^2$ to fix overflows:

5. $c_0 \sim U_{\{\pm2\}}$; $c_1 \leftarrow c - c_0$

6. $k_0^{IC} \leftarrow (k_{\gamma, 0}, c_0)$; $k_1^{IC} \leftarrow (k_{\gamma, 1}, c_1)$

7. return $k_0^{IC}, k_1^{IC}$

$^2$The correction terms test three standard overflow cases and one corner case. The standard case terms test if $q > r$ overflows $(1_{p < x \leq q})$, if $q + r + 1$ overflows $(1_{p + x > q + r + 1})$.\[\text{Note that ABY} 2.0 [33] refers to arithmetic secret sharing as } [-] \text{-sharing and II-secret sharing as } \langle \cdot \rangle \text{-sharing.} \]
Table 1: Reformulation of the distance metrics into a composition of local evaluations of $f_{local}$ and the cross product $f_{cp} \cdot x^T y$

<table>
<thead>
<tr>
<th>Distance Metric</th>
<th>Formula</th>
<th>$f_{local}(x)$ + $f_{local}(y)$ + $f_{cp} \cdot x^T y$</th>
<th>$f_{local}(v)$</th>
<th>$f_{cp}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar/Inner Product</td>
<td>$\sum x^{(i)} \cdot y^{(i)}$</td>
<td>$0 + 0 + 1 \sum (x^{(i)} \cdot y^{(i)})$</td>
<td>$0$</td>
<td>$1$</td>
</tr>
<tr>
<td>Hamming Distance</td>
<td>$\sum x^{(i)} \oplus y^{(i)}$</td>
<td>$\sum (x^{(i)})^2 + \sum (y^{(i)})^2 - 2 \sum (x^{(i)} \cdot y^{(i)})$</td>
<td>$\sum (v)^2$</td>
<td>$-2$</td>
</tr>
<tr>
<td>Squared Euclidean</td>
<td>$\sum (x^{(i)} - y^{(i)})^2$</td>
<td>$\sum (x^{(i)})^2 + \sum (y^{(i)})^2 - 2 \sum (x^{(i)} \cdot y^{(i)})$</td>
<td>$\sum (v)^2$</td>
<td>$-2$</td>
</tr>
<tr>
<td>Squared Mahalanobis</td>
<td>$(x - y)^T M(x - y)$</td>
<td>$x^T Mx + y^T My - 2(x^T M) \cdot y$</td>
<td>$(v^T Mv)$</td>
<td>$-2$</td>
</tr>
</tbody>
</table>

Algorithm 2: FSS.EvalIC($j, k_j, \hat{x} \rightarrow o_0, o_1$

Inputs: $j$: The party number, $j \in \{0, 1\}$
$k_j$: The key for $P_j$, composed of a DCF key for $\gamma$ and a correction share $c_j$
$\hat{x}$: Masked public input, result of reconstructing $x + r$.

Output: $o_0, o_1$: Additive secret shares of $1 \in \{0, 0\}$.

Define the interval $[p, q]$ for sign extraction:
1. $p \leftarrow 0$; $q \leftarrow 2^n - 1$

Deserialize key and obtain local overflow term $\eta$:
2. $(k_{j}, c_j) \leftarrow k_j$
3. $\eta \leftarrow 1_{x > p} - 1_{x > q + 1}$

Evaluate the DCF with two inputs and compute result:
4. $o_j \leftarrow \text{FSS.EvalIC}(j, k_{j}, 1, \hat{x} = 1)$
5. $o_i \leftarrow \text{FSS.EvalIC}(j, k_{j}, 1, \hat{x} = q = 2)$
6. return $o_j \leftarrow j \cdot \eta - o_j^f + o_j^c + c_j$

2.1.4 On security guarantees. This work focuses on 2PC with security against a semi-honest adversary non-adaptively corrupting at most one computing party. Also referred to as Honest-but-Curious, the computing parties $P_j$ are to follow the protocol faithfully, while a party corrupted by the adversary will try to extract as much information as possible from his computation.

Employing simulation based security proofs [17, 38], previous works have proven SS and ISS to be perfectly information theoretic secure against computationally unbounded semi-honest adversaries [29, 53]. In contrast, FSS schemes rely on the security of the underlying PRG to be proven computationally secure against time bounded adversaries [11].

2.2 Thresholded Distance Metrics and Applications

Inspired by GSHADE [15], we introduce the thresholded distance metrics that we seek to protect in this work alongside motivating real-world applications:

- **Scalar Product**: $f_{sp}(x, y) = x^T y = \sum_{i=1}^{n} x^{(i)} \cdot y^{(i)}$ is a common distance metric in face recognition where $x, y \in \mathbb{R}^n$ are two vectors of the same dimension.
- **Hamming Distance**: $f_{hd}(x, y) = \sum_{i=1}^{n} (x^{(i)} \oplus y^{(i)})$ is a distance metric frequently used in information theory and computer science to measure the distance between two bit-strings. Besides its interest in iris and fingerprint recognition, it is the base of the perceptual hashing technique [49] used in image comparison, with applications ranging from image watermarking [31] to detection of Child Sexual Abuse Material (CSAM)[25].
- **Squared Euclidean Distance**: $f_{sed}(x, y) = \sum_{i=1}^{n} (x^{(i)} - y^{(i)})^2$ is a distance metric used in many machine learning applications, such as clustering [48]. It is also used in the context of face recognition [37].
- **Squared Mahalanobis Distance**: $f_{md}(x, y) = (x - y)^T M(x - y)$ is a distance metric used in many machine learning applications, such as clustering [48] and recognition of hand shape/key/strokes/signatures [15].

3 OUR SOLUTION

We now describe our solution for a lightweight and efficient 2PC distance metric with comparison, requiring a single round of communication and two ring elements in the online phase.

3.1 Distance Metrics

We start off by writing the generic function we wish to protect:

$$f(\text{dist}, \theta, x, y) = 1_{\text{dist}(x, y) > \theta} = \begin{cases} 1 & \text{if } \text{dist}(x, y) > \theta, \\ 0 & \text{if } \text{dist}(x, y) < \theta. \end{cases} \quad (3)$$

To adapt to 2PC, we reformulate the distance metrics $f_{dist}$ from Section 2.2 as

$$z = f_{dist}(x, y) = f_{local}(x) + f_{local}(y) + f_{cp} \cdot \sum_{i=1}^{2n} (x^{(i)} \cdot y^{(i)}) \quad (4)$$

where $f_{local}$ is a function that can be computed locally by each input data holder, and $f_{cp}$ is the “cross product” constant factor that applies to the scalar product evaluation present in all the metrics. Using this blueprint, we rewrite all the distance metrics in Table 1.

We remark that the Hamming Distance can be reformulated as the Squared Euclidean Distance as long as the input vectors are composed of binary values $x^{(i)}, y^{(i)} \in \{0, 1\} \forall i$, since the boolean XOR operation between two binary values can be rewritten in the arithmetic domain as $x^{(i)} \oplus y^{(i)} = (x^{(i)} - y^{(i)})^2$, the square of its difference.

3.2 The Two-Party Computation Scenario

This work is set in a context of two party computation (2PC), where two non-colluding parties ($P_0, P_1$) are in charge of performing the secure computation. We argue the security of our protocols assuming these two parties behave in a semi-honest manner, following the protocol steps faithfully while attempting to extract as much
information as possible from the process. In this context, Funshade guarantees privacy of all the input data against a semi-honest adversary corrupting either of the two parties (see Section 3.6 for a detailed security analysis).

Our work is set in the MPC with preprocessing model, leveraging off a setup/offline phase in order to optimize the cost of the online (input-dependent) phase. Much like for other ZPC scenarios [19, 29, 53, 55], our protocols also require several additional roles to be filled in a complete solution:

- $R_{\text{setup}}$: The setup covers the generation of preprocessing material during the offline phase, and its distribution to the computing parties involved in the online phase.
- $R_{\text{in}_x}, R_{\text{in}_y}$: The input data holders, with access to the input vectors $x$ and $y$ respectively. These vectors are to be shared with the computing parties either during the offline phase if available beforehand, or at the beginning of the online phase. By convention, and following Equation 3, we employ $x$ for the live template and $y/Y$ for the reference template(s).
- $R_{\text{res}}$: Receives the shares of the secure computation and reconstructs the result.

A role can be performed by more than one party at the same time, e.g., $P_0, P_1$ can jointly perform the role $R_{\text{setup}}$ (more in Section 3.5). A party can carry out multiple roles as well.

### 3.3 Sketching the Solution

With the different parties and roles in place, we are now ready to sketch our solution. In a nutshell, we combine II-sharing to locally compute a scalar product, with the FSS gate for interval containment from [11] with full correctness.

The key insight driving our design stems from the intermediate SS state in the II-sharing multiplication (Equation 2). By providing II-shared input vectors to the computing parties $P_j$, we can locally obtain the SS shares of the element-wise multiplication, and perform local cumulative addition to obtain shares of the scalar product result. Compared to the pure SS approach, we no longer need a round of communication to reconstruct the intermediate values $x \leftarrow a$ and $y \leftarrow b$ masked by Beaver triples (Mult. in Equation 1). As pointed out in ABY2.0 [53], the communication in a FSS multiplication gate happens at the output wires, as opposed to SS multiplication gates where the round of communication is tied to the input wires.

The subsequent FSS gate for interval containment requires a publicly reconstructed input held by both parties, which, to preserve the input data privacy, must be masked prior to its reconstruction (in line with previous FSS-based works [11, 14, 58]). Crucially, the masking of the private input via local shares addition followed by its reconstruction (at the cost of one round of communication) happens at the input wire of the FSS gate.

All we have left is to put together the two pieces of the puzzle. We can skip the II-sharing reconstruction and instead add the input mask directly to the scalar product output, and then reconstruct this masked value to serve as public input for the FSS interval containment gate. Figure 1 depicts our idea applied to the scalar product metric.

To obtain the other metrics we would have each input data holder $R_{\text{in}_x}, R_{\text{in}_y}$ run $f_{\text{local}}$ on its input and secret share this result with the computing parties to add it to the output of the scalar product. In addition to that, both parties would multiply the shares of the scalar product result with the corresponding public value $f_{\text{p}}$, resulting in the correct distance metric evaluation $z = f_{\text{dist}}(x, y)$.

To keep the threshold $\theta$ hidden from the computing parties (and known only by $R_{\text{setup}}$), we subtract the value of $\theta$ from the additive random mask $r$ during the offline/setup phase to get $r_\theta$, and then compute $z_\theta = z - \theta$.

### 3.4 Protocol Specification

Embracing this combination of FSS for the locally computed scalar product and FSS for the comparison to $\theta$, we can now outline each of the protocols that compose the full solution:

1. **Funshade.Setup** (Protocol 3): generation of the correlated randomness required for the scalar product multiplications, as well as the keys for the interval containment, and distribution of all the preprocessing material.

#### Protocol 3  Funshade.Setup($l$, $n$, $\lambda$, $\theta$) → $k_0, k_1, (\delta_x), (\delta_y)$

**Players:** $R_{\text{setup}}$

**Input:**
- $l$: length of the input vectors.
- $n$: number of bits for the secret sharing ring $\mathbb{Z}_{2^n}$.
- $\lambda$: security parameter.
- $\theta$: threshold for the comparison $\in \mathbb{Z}_{2^n}$.

**Output:**
- $k_0, k_1$: preprocessing keys, sent to $P_0, P_1$ respectively.
- $\langle \delta_x \rangle$, $\langle \delta_y \rangle$: $\delta$-shares of input vectors, sent to $P_{\text{in}_x}, P_{\text{in}_y}$ (input owners) resp.

**Note:** All arithmetic operations (+, −, ×) are defined in $\mathbb{Z}_{2^n}$.

**Beaver Triples for II-sharing scalar product:**

$$
\begin{align*}
\delta_{xy} &\sim \mathcal{U}[\mathbb{Z}_{2^n}] \\
\delta_{xy} &\leftarrow \langle \delta_x \rangle \cdot \langle \delta_y \rangle  \\
\langle \delta_{xy} \rangle &\equiv \langle \delta_x \rangle \cdot \langle \delta_y \rangle \\
\langle r \rangle &\equiv \langle r_0 \rangle \cdot \langle r_1 \rangle  \\
\langle r_\theta \rangle &\equiv \langle r_0 \rangle \cdot \langle r_1 \rangle - \theta
\end{align*}
$$

**FSS interval containment:**

$$
\begin{align*}
k_0^\text{IC}, k_1^\text{IC} &\leftarrow \text{FSS.Gen}^{\text{IC}}(\lambda, n, r) \\
k_j &\equiv \langle \delta_x \rangle, \langle \delta_y \rangle, \langle \delta_{xy} \rangle, \langle r_\theta \rangle, k_j^\text{IC}, j \in \{0, 1\}
\end{align*}
$$

**Dealing the preprocessing material:**

$$
\begin{align*}
\text{SEND} & k_0 \leftarrow P_0, \quad (\langle \delta_x \rangle + \langle \delta_x \rangle) \Rightarrow R_{\text{in}_x} \\
& k_1 \leftarrow P_1, \quad (\langle \delta_y \rangle + \langle \delta_y \rangle) \Rightarrow R_{\text{in}_y}
\end{align*}
$$

(2) **Funshade.Share** (Protocol 4): $R_{\text{in}_x}, R_{\text{in}_y}$ prepare the II-shares of their corresponding inputs using the correlated randomness and then send these shares to $P_0, P_1$.

(3) **Funshade.Eval** (Protocol 5): $P_0, P_1$ engage in an online protocol upon acquiring the II-shares of both inputs, using local multiplication and addition to compute the scalar product, and then evaluate the interval containment FSS scheme to determine whether the result is below the threshold $\theta$.

(4) **Funshade.Result** (Protocol 6): $P_0, P_1$ send the arithmetic shares of the result to $R_{\text{res}}$ for its reconstruction.
This is a DRAFT of the final version. DOI and page numbers are subject to change.

3.5 Applications and Practical Considerations

We can employ our solution in a variety of use-cases, from biometric verification to CSAM detection. In this section we apply Funshade to a scenario of biometric authentication for access control.

There is a Gate allowing access to a flight/train/concert with a small computer and a camera, and a Biometric Provider (BP) operating a server that holds the database of registered/enrolled users. A user corresponds to a biometric template belonging to an enrolled user, and the server to a freshly captured biometric template. This setting naturally includes two computing parties \{Gate, BP\} ≡ \{P₀, P₁\}, each of them playing specific roles, with Gate ⊇ \(R_{in_x}, R_{res}\) and BP ⊇ \(R_{in_y}\). The solution is split into two phases:

### Enrollment (offline):

1. **R\(_{setup}\)** carries out the generation of preprocessing material and distributes it to BP and Gate.
2. **BP** (\(R_{in_y}\)) enrolls an user by collecting its biometric template \(y\), and then secret shares it with Gate (\(P₀\)).

### Online phase

3. A user attempts to cross the Gate. Gate (\(R_{in_x}\)) gets and secret shares his live template with BP (\(P₁\)).

---

![Figure 1: Overview of Funshade primitives](image)

**Protocol 4** Funshade.Learn(\(\sigma, \delta_0\)) → \(\Delta_0, (d_0)\)

**Players:** \(R_{in_a}\), holding the input vector \(\sigma\) (where \(\sigma \in \{x, y\}\)).

**Input:** \(\sigma\): input vector \(\sigma \in \mathbb{Z}^n\) _\(_2^\tau\) held by \(P_{in_a}\).

**Output:** \(\Delta_0\): \(\delta\)-shares of vector \(\sigma\) distributed to both \(P₀\) and \(P₁\).

\(d_0\): Arithmetic shares of the local computation \(f_{local}(\sigma)\).

1. **\(\Delta_0\) := \(\sigma + \delta_0\)**
2. **\(d_0\) := \(f_{local}(\sigma)\)**; \(d₀) \equiv (d₀₀, d₀₁) \equiv (\sim \mathcal{U}(\mathbb{Z}_q), d₀ - d₀₀)\)
3. **Send** (\(\Delta_0, d_0\)) \(\Rightarrow P₀\); (\(\Delta_0, d_0\)) \(\Rightarrow P₁\)

**Protocol 5** Funshade.Eval\(j, \Delta_x, \Delta_y, (d_x), (d_y), (k_1)\) → \(o\)

**Players:** \(P_j\), \(j \in \{0, 1\}\) computing parties.

**Input:** \(\Delta_x, \Delta_y\): \(\delta\)-shares of \(\{x, y\}\); (II-shared inputs \(x, y\)) held by both \(P₀\) and \(P₁\).

\(d_x\), \(d_y\): Arithmetic shares of locally computed single-input terms \(f_{local}(x), f_{local}(y)\) of \(f_{dist}(x, y)\).

\(k_1\): preprocessing key from Funshade.Setup containing: \(\delta_x, \delta_y\): \(\delta\)-shares of \(\delta\)-shared input vectors \(x, y\), \(\delta_{xy}\): arithmetic shares of Beaver triple s.t. \(\delta_{xy} = (\delta_x, \delta_y)\), \(r_0\): arithmetic shares of \(s\) input mask \(r\) minus threshold \(\theta\), \(k^IC\): FSS key for the IC gate of [11].

**Output:** \(\langle o\rangle\): arithmetic shares of the result \(o := \sum_{j=0}^{1} \delta_{xy} \cdot \delta_j\).

**Note:** All steps apply to both computing parties \(P_j, \ j \in \{0, 1\}\). All arithmetic operations (+, −, ∙) are defined in \(\mathbb{Z}_q\).

**II-sharing based scalar product:**

1. **\(\hat{z}_0\) := \(\hat{z}_0 + \hat{z}_1\)**
2. **Return** \(\hat{z}_0\)

Identification (Online):

3. A user attempts to cross the Gate. Gate (\(R_{in_x}\)) gets and secret shares his live template with BP (\(P₁\)).
Figure 2: Diagram of a privacy-preserving biometric authentication solution for flight boarding.

Protocol 6: Funshade\[Result(\varnothing)\] → \varnothing

Players: P, j ∈ {0, 1} computing parties, R\text{res} result holder.
Input: (\varnothing): secret shares o_0, o_1 ∈ \mathbb{Z}_{2^n} of the result o held by P_0, P_1.
Output: o: Output value.

1: \(R_j\): Send o_j → R\text{res}.
2: R\text{res}: o ← (o_0 + o_1)

4. Gate (P_0) and BP (P_1) jointly perform Protocol 5, incurring in one round of communication where each party sends its share of the distance evaluation to the other.

5. The final result is sent to Gate (R\text{res}), who accepts or rejects the user accordingly.

This instantiation can be parallelized for different reference inputs \(y^{(k)}\) in cases where the reference database contains more than one record, e.g., biometric identification against a database of multiple subjects, CSAM detection against a large database of image hashes. In these cases, the individual index secret shares outputs \(o_j^{(k)}\) can be locally summed up to yield a single value as output. Additionally, these use-cases normally gather their reference databases ahead of time, allowing for early deployment of the \(Y\)-shares of \(y^{(k)}\).

Realizing the setup \(R_{\text{setup}}\). In line with previous work on semi-honest MPC [8, 26, 45, 56, 64], fresh randomness is generated for each 1:1 verification in the offline phase and used only once in the online phase, as the security of our protocols would weaken if we were to reuse randomness/masks. Protocols in this work are presented in the preprocessing model, where P_0, P_1 receive correlated randomness from a trusted dealer taking the role of R\text{setup}. Protocols within this model can be converted to protocols in the standard model by:

(a) Resorting to trusted hardware [52]. The role R\text{setup} can be emulated within a trusted execution environment inside one of the computing parties, such as Intel SGX or ARM TrustZone [51].
(b) A semi-honest 3PC setting with honest-majority [63]: A third party P_2 can be added to the protocol to enact R\text{setup} during the offline phase and remain dormant in the online phase.
(c) Pure 2PC [11]: R\text{setup} can be jointly emulated by the two parties P_0, P_1 via a small-scale two-party secure protocol.

For the biometric authentication solution illustrated in Figure 2 we favor the 2PC approach, resorting to generic 2PC techniques for the FSS gate key generation (Appendix A.2 of [11]), and either Oblivious Transfer (OT) or Homomorphic Encryption (HE) for the preprocessing of the FISS scalar product (Section 3.1.3 of [53]).

3.6 Security Analysis

We consider security against a Honest-but-Curious adversary \(\mathcal{A}\) that corrupts up to one of the two computing parties \(P_j\). We consider a static corruption model where the adversary must choose which participant to corrupt before the execution of the computations. This is a standard security model in previous MPC frameworks [11, 19, 29, 50, 53, 58]. Under this threat model, we define and later prove the security and correctness of our constructions.

We employ the standard real-world-ideal world paradigm, providing the simulation for the case of a corrupt \(P_j\). The ideal world simulation contains an additional trusted party that receives all the inputs from P_0, P_1, computes the ideal functionality correctly and sends the corresponding results back to P_0, P_1. Conversely, the real world simulation executes the protocol as described in the Funshade algorithms in the presence of \(\mathcal{A}\).

Our security proof works in the \(\mathcal{F}_{\text{Funshade,setup}}\)-hybrid model where \(\mathcal{F}_{\text{Funshade,setup}}\) represents the ideal functionality corresponding to protocol Funshade\[setup\].

Definition 1 (Security of Funshade). For each \(j \in \{0, 1\}\), there is a PPT algorithm \(S\) (simulator) such that \(\forall o \in \mathbb{Z}_{2^n}, \forall x, y \in \mathbb{Z}_n^l\) and every function \(f_{\text{dist}}(x, y) : \mathbb{Z}_{2^n}^l \rightarrow \mathbb{Z}_n\). From Table 1, \(S\) realizes the ideal functionality \(\mathcal{F}_{\text{th,−dist}}\), such that its behavior is computationally...
indistinguishable from a real world execution of protocols 4-5-6 in the presence of a semi-honest adversary $\mathcal{A}$.

Ideal Functionality $F_{th\text{-dist}}$

$F_{th\text{-dist}}$ interacts with the parties $P_0, P_1$ and the adversary $S$ and is parametrized by a publicly known function $f_{dist}(x, y)$ and a threshold $\theta$.

- **Inputs**: $F_{th\text{-dist}}$ receives the inputs $\Delta x, \Delta y, \delta x_j, \delta y_j$ from the computing parties $P_0, P_1$.

- **Computation**: $F_{th\text{-dist}}$ reconstructs $x = \Delta x - (\delta x_0 + \delta x_j)$ and $y = \Delta y - (\delta y_0 + \delta y_j)$, computes $z = f_{dist}(x, y)$ and $o = 1_{z \geq \theta}$.

- **Output**: Sends $o, j$ to $P_{res}$.

**Theorem 1.** In the $F_{\text{FUSHADE-Setu}}\text{-hybrid}$ model, protocols 4-5-6 (online phase) securely realize the functionality $F_{th\text{-dist}}$.

**Proof.** The semi-honest adversary corrupts $P_j$ during the sequential execution of protocols 4-5-6. For this case, $S$ executes the setup phase honestly on behalf of $P_{1-j}$ (in case of interactive setup), and will simulate the entire circuit evaluation, assuming the circuit-inputs of $P_{1-j}$ to be 0 in the $\text{FUSHADE}$. Result protocol, $S$ adjusts the shares of $(o)$ on behalf of $P_{1-j}$ so that $A$ sees the same transcript as in the real-world protocol.

- **FUSHADE.Setup**: For the offline phase, we consider it as an ideal functionality $F_{\text{FUSHADE-Setu}}$, which generates the required FSS preprocessing keys and $\delta$-shares. Since we make only black-box access to $\text{FUSHADE.setup}$, its simulation follows from the security of the underlying primitive used to instantiate it (OT or HE for the ISS preprocessing material stemming from setupMULT of [53]), generic 2PC for the FSS keys following Appendix A.2 of [11]. Alternatively, the Setup can be delegated into a third party, emulated with trusted hardware inside $P_0|P_1$ or with an independent semi-honest party (as explained in Section 3.5).

- **FUSHADE.Share**: We generalize the behavior of $S$ for both inputs $o \in \{x, y\}$, for the instances where $P_j$ is the owner of the values (e.g., $P_j \supseteq R_{in,y}$), $S$ has to do nothing since $\mathcal{A}$ is not receiving any messages. $S$ receives $\Delta y_j$ from $A$ on behalf of $P_{1-j}$, for the instances where $P_{1-j}$ is the owner, $S$ sets $\sigma = 0$ and performs the protocol steps honestly.

- **FUSHADE.Eval**: During the online phase, $S$ follows the protocol steps honestly using the data obtained from the setup phase. The scalar product requires $l$ local additions (non-interactive and thus they don’t need to be simulated) and a subsequent reconstruction of $(\hat{z}_0)$ as $\hat{z}_0 = \hat{z}_0 + \hat{z}_0$ that behaves just like $\text{FUSHADE}.\text{Result}$ and serves as input to the FSS IC gate. For the FSS IC gate, we resort to the Simulation-based security of [11] (Definition 2) to argue computational indistinguishability of the ideal and real world executions, hiding the information of $r$ contained in $k_0$ and $k_1$ from $\mathcal{A}$.

- **FUSHADE.Result**: To reconstruct a value $(o)$, $S$ is given the output $o$, which is the output of $\mathcal{A}$. Using $o$ and the share $o_{1-j}$ corresponding to $P_{1-j}$, $S$ computes $o_j = o - o_{1-j}$ and sends this to $\mathcal{A}$ on behalf of $P_{1-j}$. $S$ receives $o_j$ from $\mathcal{A}$ on behalf of $P_{1-j}$.

**Definition 2 (Correctness of FUSHADE).** For every threshold $\theta \in \mathbb{Z}_{2^n}^+$, every pair of input vectors $x, y \in \mathbb{Z}_n$ and every function $f_{dist}(x, y) : \mathbb{Z}_n^l \rightarrow \mathbb{Z}_n$ from Table 1,

\[
\begin{align*}
\text{if } (k_0, k_1, (\delta x), (\delta y)) &\leftarrow \text{FUSHADE.Gen}(l, n, \lambda, \theta) \\
\text{and } (\Delta x, (\Delta x, (\delta x_0 + \delta x_j), \Delta y, (\delta y_0 + \delta y_j)) &\leftarrow \text{FUSHADE.Share}(y, (\delta y_j)) \\
\text{then } &\Pr[\text{FUSHADE.Eval}(0, \Delta x, \Delta y, dx_0, dy_0, k_0) \\
&+ \text{FUSHADE.Eval}(1, \Delta x, \Delta y, dx_j, dy_j, k_1) \\
&= 1_{f_{dist}(x, y) \geq \theta} = 1].
\end{align*}
\]

**Theorem 2.** Jointly, protocol 3 (offline), and protocols 4-5-6 (online), realize the function $f_{dist}(\theta, x, y) = 1_{f_{dist}(x,y) \geq \theta}$ correctly.

**Proof.** We first decompose the II-sharing based scalar product (step 1 of Protocol 5) for the joint result of the two computing parties $\hat{z}_0$ in Equation 6,

\[
\begin{align*}
\hat{z}_0 &= \hat{z}_0 + \hat{z}_1 = (r_0 + r_1) + (d_0 + d_x + d_y + f_{dist}(x, y)) \\
&= r_0 + d_x + d_y + f_{dist}(x, y) \\
&= r_0 + d_x + d_y + f_{dist}(x, y) \\
&= r_0 + d_x + d_y + f_{dist}(x, y) \\
&= r_0 + d_x + d_y + f_{dist}(x, y)
\end{align*}
\]

where we group all the SS shares and reconstruct their original values, replace $\hat{z}_0$ by the corresponding values (from definitions in protocol 1), group the II-shares of $x$ and $y$ to later reconstruct their values, and finally make use of Equation 4.

With the public input $\hat{z}$ sorted out, we analyze the Interval Containment evaluation with output reconstruction in Equation 7,

\[
\begin{align*}
\sigma &= \sigma_1 + \sigma_2 = \text{FSS.EvalIC}(0, k_0^{IC}, \hat{z}_0) + \text{FSS.EvalIC}(1, k_1^{IC}, \hat{z}_0) \\
&= \text{FSS.EvalIC}(0, \text{FSS.GenIC}(\lambda, n, r)) + \text{FSS.EvalIC}(1, \text{FSS.GenIC}(\lambda, n, r)) \\
&+ \text{FSS.EvalIC}(\lambda, n, r) f_{dist}(x, y) = 1_{\hat{z}_0 \leq \hat{z}_1} = 1_{f_{dist}(x, y) \geq \theta}
\end{align*}
\]

where we resort to Theorem 3 of [11] to argue that the two protocols (FSS.GenIC $(\lambda, n, r)$, FSS.EvalIC $(j, k_j^{IC}, \hat{z}_j)$) constitute an FSS gate correctly realizing $f_{dist}(x, y) = 1_{\hat{z}_0 \leq \hat{z}_1}$. Then, following Definition 2 (Correctness) of [11], we can argue that

\[
\Pr[\text{FSS.EvalIC}(0, k_0^{IC}, \hat{z}_0) + \text{FSS.EvalIC}(1, k_1^{IC}, \hat{z}_0) = 1_{\hat{z}_0 \leq \hat{z}_1}] = 1_{f_{dist}(x, y) \geq \theta}
\]

equating the FSS gate output to $1_{\hat{z}_0 \leq \hat{z}_1}$, the unit step function. □

\*\*\*There are several notation adaptations to align with [11]. Our mask $r$ is written as $r''$ in Figure 3 of [11] depicting the FSS IC gate. We set the parameters $p = 0$ and $q = 2^{-m-1} - 1$ to define the interval containing all positive integers $1_{\hat{z}_0 \leq \hat{z}_1} = \chi_{\mathbb{Z}_{2^n}^+}(\hat{z}_0)$ is a function that belongs (per definition of IC gate in Section 4 of [11]) to the family of functions $\mathcal{F}_{\mathbb{Z},\mathbb{Z}}$ referenced in Theorem 3 of [11].
This is a DRAFT of the final version. DOI and page numbers are subject to change.

**FUNSHADE: Function Secret Sharing for Two-Party Secure Thresholded Distance Evaluation**

**Proceedings on Privacy Enhancing Technologies 2023(4)**

### Table 2: Communication size, computation time and latency (in both LAN and WAN settings) for the different steps of **FUNSHADE** applied to biometrics, with \( n = 32 \) and \( \lambda = l = 128 \), in two scenarios: authentication (\( K = 1 \)) and identification (\( K = 5000 \)).

<table>
<thead>
<tr>
<th>Phase/Step</th>
<th>Authentication (K=1)</th>
<th>Identification (K=5000)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Comm.</td>
<td>Computation</td>
</tr>
<tr>
<td>Offline</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1) Setup</td>
<td>5.7 KB</td>
<td>33.63 ( \mu s )</td>
</tr>
<tr>
<td>2) Share(( y ))</td>
<td>512 B</td>
<td>0.038 ( \mu s )</td>
</tr>
<tr>
<td>Total</td>
<td>6.2 KB</td>
<td>33.67 ( \mu s )</td>
</tr>
<tr>
<td>Online</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3) Share(x)</td>
<td>512 B</td>
<td>0.038 ( \mu s )</td>
</tr>
<tr>
<td>4) Eval (SP)</td>
<td>8 B</td>
<td>0.19 ( \mu s )</td>
</tr>
<tr>
<td>5) Eval (IC)</td>
<td>-</td>
<td>9.2 ( \mu s )</td>
</tr>
<tr>
<td>5) Result</td>
<td>4 B</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>524 B</td>
<td>9.43 ( \mu s )</td>
</tr>
</tbody>
</table>

### 4 EXPERIMENTS

#### 4.1 Implementation and Environment

We implement **FUNSHADE** in a compact⁵, portable⁶ and standalone⁷ C module, and open source it⁸. We also provide a lightweight wrapper to Python by virtue of Cython. We instantiate the PRG function \( G \) (employed in the DCF gate) with a Miyaguchi-Pieneel one-way compression function over an AES block cipher, an extended variant of Matyas-Meyer-Oseas function used in previous works [58]. We concatenate several fixed key block ciphers to achieve the desired output length.

#### 4.2 Results

To assess the efficiency of our constructions, we test the execution of the full set of protocols from Figure 2 on a single core with an Intel(R) Core(TM) i7-7800X CPU, limiting the RAM consumption to up to 8GB, averaging measurements over at least 10 runs. For the intra and inter-protocol communication we employ a secure channel with capacity of 100MBps and consider two scenarios:

- A LAN setting with 10ms of transmission latency.
- A WAN setting with 70ms of transmission latency.

We time the execution of the FSS primitives, summarizing the results in Table 3. We remark a x5 speedup in the PRG function \( G \) by resorting to AES-NI CPU instructions, and therefore employ it for all the FSS primitives. As expected, the cost of both \( G_{\text{gen}} \) and \( G_{\text{eval}} \) algorithms increase linearly with \( n \), the bit size of the ring \( \mathbb{Z}_{2^n} \) where the operations take place.

Subsequently, we assess the communication costs for each step of our solution using the Scalar Product (SP) as metric⁹, and we list them in Table 4. The size of the offline phase’s output can be interpreted as the total preprocessing. Overall, we highlight the extremely low communication size in the online phase, of just \( 2Kn \) bits (2 ring elements for each of the \( K \) distance evaluations).

---

Footnotes:

⁵ Around 1000 Lines of Code for the core implementation.

⁶ We employ only generic integer types, C arrays and plain C89 instructions (supported by every C compiler) to integrate smoothly with higher-level languages such as Python, Rust or Golang.

⁷ The only optional dependency is 11bstdcom for secure randomness generation.

⁸ Our code is available at https://github.com/ibarrond/funshade.

⁹ We employ only generic integer types, C arrays and plain C89 instructions (supported by every C compiler) to integrate smoothly with higher-level languages such as Python, Rust or Golang.

¹⁰ Note that the evaluation of other distance metrics would require adding just \( n \) bits (one ring element, corresponding to a share of \( d_{x,y} \)) to the sharing of \( x \) and \( y \).

¹¹ Feature extractors with similar characteristics can be obtained from https://github.com/deepinsight/insightface/wiki/Model-Zoo

### Table 3: Timings (ns) for the execution of FSS primitives over \( n \)-bit inputs, for \( \lambda = 128 \). The PRG function \( G_{\text{gini}} \) implements a standalone AES block, whereas \( G_{\text{gini}} \) does so with faster AES-NI CPU instructions. All the FSS primitives use \( G_{\text{gini}} \) internally.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Input Size</th>
<th>8-bit</th>
<th>16-bit</th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>( G_{\text{gen}} )</td>
<td>162</td>
<td>162</td>
<td>202</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( G_{\text{gini}} )</td>
<td>871</td>
<td>871</td>
<td>1149</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSS.Gen(_{\text{gen}})</td>
<td>4306</td>
<td>6818</td>
<td>11521</td>
<td>26720</td>
<td></td>
</tr>
<tr>
<td>FSS.Eval(_{\text{gen}})</td>
<td>1207</td>
<td>2372</td>
<td>4700</td>
<td>11974</td>
<td></td>
</tr>
<tr>
<td>FSS.Gen(_{\text{gini}})</td>
<td>4974</td>
<td>7387</td>
<td>12287</td>
<td>28461</td>
<td></td>
</tr>
<tr>
<td>FSS.Eval(_{\text{gini}})</td>
<td>2409</td>
<td>4727</td>
<td>9296</td>
<td>23872</td>
<td></td>
</tr>
</tbody>
</table>

### Table 4: Communication costs of the **FUNSHADE** protocols. \( K \) is the number of reference vectors (\( K = 1 \) for authentication, \( K > 1 \) for identification), \( l \) is the vector length, \( n \) is the bit size of the vector elements, and \( \lambda \) is the security parameter.

<table>
<thead>
<tr>
<th>Phase / Step</th>
<th># rounds</th>
<th>Comm. size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offline</td>
<td>1) Setup</td>
<td>((+1)^l) ( Kn(2l+9l+4n+10)+2Kn )</td>
</tr>
<tr>
<td></td>
<td>2) Share(( y_k ))</td>
<td>1 ( Knl )</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>((+1)^l) ( Kn(2l+9l+4n+10)+2Kn )</td>
</tr>
<tr>
<td>Online</td>
<td>3) Share(x)</td>
<td>1 ( ml )</td>
</tr>
<tr>
<td></td>
<td>4) Eval (SP)</td>
<td>1 ( 2Kn )</td>
</tr>
<tr>
<td></td>
<td>5) Eval (IC)</td>
<td>0 ( - )</td>
</tr>
<tr>
<td></td>
<td>5) Result</td>
<td>1 ( 2n )</td>
</tr>
<tr>
<td>Total</td>
<td>3</td>
<td>( n(l+2K+2) )</td>
</tr>
</tbody>
</table>

Focusing on the biometrics use-case, we resort to the Labelled Faces in the Wild (LFW) dataset [40] (13233 faces from more than 5000 identities) and extract biometric templates of length \( l = 128 \) with an ArcFace-based [30] feature extractor¹¹. We set \( n = 32 \) to ensure that the protocols perfectly mimic the plaintext operations and thus obtain the exact same results, without any drop in accuracy. We report in Table 2 the results for two scenarios:

- **Authentication**, a 1:1 verification against a single reference.
- **Identification**, a 1:K verification of the live template with a set of \( K \) references, employing a subset (\( K = 5000 \)) of the identities in the LFW dataset, enough to be of use for the applications mentioned beforehand (e.g., access control for a flight/train).
As shown in Table 2, the computation costs of our solution for authentication are negligible with respect to the communication latency. The total online latency for the identification scenario is more balanced in the LAN setting, whereas communication still dominates in the WAN setting. In any case, the total online latency amounts to less than 300ms including input sharing and output reconstruction, as well as its low-interaction lightweight communication makes FUNSHADE an ideal solution for privacy-preserving biometric verification in real time due to its low interaction.

Last but not least, we test our protocols with randomized input vectors of varying length \( l \) corresponding to typical template sizes of modern biometric feature extractors [30], employing bit-sizes \( n \) corresponding to common CPU integer types to benefit from cheap modular operations. We record the total online latency of all these experiments in the identification scenario for increasing references \( K \). Figure 3 displays these results and shows how the vector size \( l \) has a much lower impact in the latency than \( n \), hinting that applications that require lower numerical precision (e.g., \( n = 16 \)) will be noticeably faster. Figure 4 shows the ratio of the communication latency to the total online latency. We observe that communication is the main bottleneck up until \( K \approx 1000 \) in the LAN setting, and \( K \approx 10000 \) in WAN. While this suggests that applications with higher requirements of \( K \) (e.g., nation-wide identification) should consider additional optimizations for the local computation (e.g., using multiple cores for parallelization or even resorting to a GPU), our non-parallelized solution requires around 10s to identify an individual against a database of 1 million records for \( n = 32 \).

To wrap up, we verify the 100% correctness of these experiments as long as natural overflows \((-2^{n-1} \leq z \leq 2^{n-1} - 1\) for signed integers, \(0 \leq z \leq 2^n - 1\) for unsigned integers) are avoided.
5 PREVIOUS WORK

Distance metric evaluations, specially for Hamming Distance and Scalar Products, range among the most typical applications of privacy-preserving computation techniques. Consequently, a wide variety of previous work in MPC, FHE and FE have dealt with some form of it.

The Multi Party Computation field includes a plethora of works covering distance metric evaluations. All the frameworks for privacy preserving neural networks cover scalar-product-based matrix multiplications often followed by ReLU activations [8, 26, 45, 56, 64], covering a mixture of Garbled Circuits, Secret Sharing and their conversions. Secure hamming distance evaluation has motivated work such as [16] based on Oblivious Transfer, with its generalization to multiple metrics in [15]. Mixed-mode protocols have also tackled distance evaluations [29, 50, 53]. However, the majority of these solutions incur in a considerable communication cost to perform comparison. More recently, solutions based on FSS [11, 14, 58] have shown promising results, leading to this work.

In the field of Homomorphic Encryption, the biometrics use-case has led to a variety of approaches, including [4, 47] for hamming distance or [66] for scalar product. However, these approaches do not include comparison to a threshold, and often rely on costly cryptographic primitives that make them slow.

Since the advent of Functional Encryption [9], scalar product and hamming distance have been the most suitable candidates to study. Inner Product Encryption (IPE) started off with selective security in [1], already envisioning biometric use-cases, and reaching full security with [28] and [61]. [44] applied FE to biometric authentication with hamming distance and to nearest-neighbor search on encrypted data; [46] employs IPE for hamming-weight based matchings of real-world iris templates. [43] and [41] are the latest iterations of privacy-preserving scalar product techniques based on FE, demonstrating performances in the order of hundreds of ms for vectors of 128 values. While FE does not require an extra operation after the "evaluation" to retrieve the result, these schemes scale polynomially with the input vector length (thus are unsuitable for very large vectors), and their computation does not include comparison to a threshold. To include it, one must resort to techniques such as Threshold Predicate Encryption [67]. There also exist techniques in the literature not resorting to these three main fields, such as [68] with a custom scheme, or [59] with Identity Based Encryption.

To position Funshade in the literature, we compare the costs of the online phase of our solution with that of selected previous works in Table 5. Funshade is the first work in the 2PC setting requiring one single round of communication to evaluate $\sum_{i=1}^{l} 1_{x_i \neq y_i}$ while also presenting the lowest communication size of 2 ring elements. An additional side-by-side comparison with AriaNN [58] is provided in Appendix A.

On the importance of the threshold comparison in privacy-preserving distance metrics. The security provided by our construction, and that of all privacy-preserving techniques in general (MPC, FHE, FE), does not prevent the reconstructed outputs $o = f(x, y)$ from revealing information about the inputs $x, y$. Indeed, $Pr_{cis}$ can leverage on his knowledge about the function being computed and attempt to extract information about the inputs from the outputs by inverting the function being computed $Leak(x, y) \leftarrow Leak(f^{-1}(o))$. Labeled as "input leakage" in previous works [41], this leakage affects the practical privacy of real-world deployments of privacy-preserving solutions. Applications using distance metric calculations as one of many building blocks (e.g., Machine Learning) might be more naturally protected thanks to the complexity of the function (beware!

<table>
<thead>
<tr>
<th>Work</th>
<th>Type</th>
<th>Rounds of communication</th>
<th>Ring elements in communication</th>
<th>Correctness</th>
<th>Online Computation Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>AriaNN [58]</td>
<td>2PC SS: Arith., FSS</td>
<td>$2(1+1)$</td>
<td>$4l + 4$</td>
<td>N</td>
<td>SS scalar product, FSS Comparison (1 DCF)</td>
</tr>
<tr>
<td>Boyle et. al. [11]</td>
<td>2PC SS: Arith., FSS</td>
<td>$2(1+1)$</td>
<td>$4l + 4$</td>
<td>Y</td>
<td>SS scalar product, FSS IC gate (2 DCF)</td>
</tr>
<tr>
<td>ABY [29]</td>
<td>2PC SS: Boolean&amp;Arith, GC</td>
<td>$3(1+2+0)$</td>
<td>$\gg 6l$</td>
<td>Y</td>
<td>Arith. to Yao conversion, GC evaluation</td>
</tr>
<tr>
<td>ABY2.0 [53]</td>
<td>2PC SS: Boolean&amp;Arith.</td>
<td>$5(1+1+3)$</td>
<td>$\gg 2$</td>
<td>Y</td>
<td>IISS scalar product, Arith. to Boolean conversion, BitExtraction</td>
</tr>
<tr>
<td>GSHADE [15] (only scalar prod.)</td>
<td>2PC OT</td>
<td>2</td>
<td>$&gt; 2l$</td>
<td>Y</td>
<td>correlated OTs.</td>
</tr>
<tr>
<td>CryptFlow2 [55]</td>
<td>2PC SS: Arith., OT</td>
<td>5</td>
<td>$&gt; (128 + 14)l$</td>
<td>Y</td>
<td>Linear layer (1-dim weights), dReLU</td>
</tr>
<tr>
<td>Falcon [64]</td>
<td>3PC Replicated SS: Arith.</td>
<td>$8(1+7)$</td>
<td>$&gt; 6$</td>
<td>Y</td>
<td>MatMult with 1-dim matrices, Private Compare</td>
</tr>
<tr>
<td><strong>Funshade (ours)</strong></td>
<td>2PC IISS: Arith., FSS</td>
<td>1</td>
<td>2</td>
<td>Y</td>
<td>IISS scalar product, FSS IC gate (2 DCF)</td>
</tr>
</tbody>
</table>
black-box model extraction attacks are real [62], yet applications requiring only one distance metric evaluation (e.g., biometric matching, CSAM detection) are much more sensitive to this leakage, since these distance metrics are linear functions and thus easily invertible.

While solutions exist to add controlled noise to the input (e.g., Differential Privacy in [18]), the most straightforward method to reduce this leakage is to output the least information possible. For applications like biometric matching and CSAM detection, one-bit outputs suffice to determine whether there is a match or not, and hence performing the comparison in a privacy-preserving manner reduces considerably the input leakage of the construction. As such, FHE and FE-based solutions without privacy-preserving threshold comparison are more risky to apply in real-world scenarios than threshold-enabled solutions that MPC (ours included) offers out of the shelf.

6 CONCLUSIONS

In this work we presented FUNSHADE, a novel 2PC privacy-preserving solution of various distance metrics (e.g., Hamming distance, Scalar Product) followed by threshold comparison. We build our protocols upon IISS, a version of arithmetic secret sharing optimized for the secure evaluation of scalar products, and function secret sharing with 100% correctness for comparison. Thanks to this, FUNSHADE proposes the first solution in the 2PC literature requiring one single round of communication in the online phase while outperforming all previous works in online communication size (two ring elements), all while relying on lightweight cryptographic primitives. We implement our solution from scratch in a portable C module, and showcase its extreme efficiency by achieving secure biometric identification against 5000 records in less than 300ms with 32-bit precision, and against 1 million records in ~10s.

ACKNOWLEDGMENTS

This work has been partially supported by the 3IA Côte d’Azur program (ANR19-P3IA-0002).

REFERENCES


[3] Manuel Barbosa, Dario Catalano, Azam Soleimanian, and Bogdan Warinschi. 2019. Efficient function-hiding functional encryption: From inner-products to secret sharing with 100% correctness for comparison. Thanks to this, FUNSHADE proposes the first solution in the 2PC literature requiring one single round of communication in the online phase while outperforming all previous works in online communication size (two ring elements), all while relying on lightweight cryptographic primitives. We implement our solution from scratch in a portable C module, and showcase its extreme efficiency by achieving secure biometric identification against 5000 records in less than 300ms with 32-bit precision, and against 1 million records in ~10s.


IEEE/CVF conference on computer vision and pattern recognition. 4690–4699.


This is a DRAFT of the final version. DOI and page numbers are subject to change.
### Arithmetic Secret Sharing (x):

\[ x = x_0 + x_1 \]

**setup**

- \( P_0 \) \( c_0, a_0, b_0 \)
- \( P_1 \) \( c_1, a_1, b_1 \)

**Secret sharing of x**

- \( P_0 \) \( x_0 \)
- \( P_1 \) \( x_1 \)

**Beaver** Mult. Triples

- \( P_0 \) \( \langle e \rangle = \langle a \cdot b \rangle \)
- \( P_1 \) \( \langle c \rangle = \langle x \cdot y \rangle \)

**Scalar Product z = \( x^T y \)**

- \( P_0 \) \( d_0, a_0, b_0, k_0, x_0, y_0, r_{\theta_0} \)
- \( P_1 \) \( d_1, a_1, b_1, k_1, x_1, y_1, r_{\theta_1} \)

**& Comparison: z ≥ \( \theta \)**

- \( \delta_{xy} = \langle \delta_x \cdot \delta_y \rangle \)
- \( \delta_{xy} = \langle \delta_x, \delta_y, r_{\theta_0}, r_{\theta_1} \rangle \)

### \( \Pi \) Secret Sharing (\( \langle x \rangle \)):

\[ x = \Delta_x - (\delta_{x0} + \delta_{x1}) \]

**setup**

- \( P_0 \) \( \Delta_x \)
- \( P_1 \) \( \Delta_x, \delta_{x0}, \delta_{x1} \)

**Online phase**

- \( \delta_{xy} = \langle \delta_{xy}, \delta_{x0}, \delta_{y0} \rangle \)
- \( \delta_{xy} = \langle \delta_{xy}, \delta_{x1}, \delta_{y1}, r_{\theta_0}, r_{\theta_1} \rangle \)

**Either**

- \( \langle \delta \rangle = \langle z \geq \theta \rangle \)
- \( \forall x, y, \forall \theta \)

**Fig. 5**: Side-by-side comparison between AriaNN and Funshade (ours)

---

This is a DRAFT of the final version. DOI and page numbers are subject to change.

Proceedings on Privacy Enhancing Technologies 2023(4) Ibarrondo et al.