FunshadeRS: Rust Implementation of the Privacy-Preserving Funshade Framework for Distributed Systems

Cintia Maja Bódi
Intitute of Network and Security
Johannes Kepler University Linz
Linz, Upper Austria, Austria
bodi@ins.jku.at

Rene Mayrhofer
Intitute of Network and Security
Johannes Kepler University Linz
Linz, Upper Austria, Austria
rm@ins.jku.at

Abstract

With biometric data increasingly being used for security, protecting this sensitive information is more important than ever. As a particular example, the Digidow project uses biometric authentication for real-world transactions such as opening doors. Digidow employs a decentralised structure in which the biometric template of each person is stored either with a chosen cloud provider or on a personal server. Since biometric data are stored and processed across potentially untrusted locations, comparison of the stored template with live data from a sensor requires a secure and privacy-preserving solution that protects this sensitive data even in the presence of potentially malicious parties.

Funshade provides a method to compare biometric data between parties without revealing the actual data itself, ensuring privacy and security. It is designed to enable secure biometric authentication through privacy-preserving protocols. In contrast to previous k-of-n bitwise comparison approaches, Funshade supports full threshold comparison over usual vector metrics, and is therefore applicable to state-of-the-art face embedding models. We implement a complete open source prototype in Rust, chosen for its strong memory-safety and performance features, to bridge the gap between modern face recognition pipelines and multi-party computation for decentralised architectures.

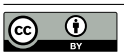
CCS Concepts

• **Security and privacy** → **Privacy protections**; **Cryptography**;
• **Computing methodologies** → **Distributed algorithms**.

Keywords

Cryptography, Biometrics, Authentication, Digital identity, Privacy, Multi-party computation

ACM Reference Format:
Cintia Maja Bódi and Rene Mayrhofer. 2025. FunshadeRS: Rust Implementation of the Privacy-Preserving Funshade Framework for Distributed Systems. In *Proceedings of the 2025 Workshop on Privacy in the Electronic Society (WPES '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3733802.3764061



WPES '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1898-4/25/10
https://doi.org/10.1145/3733802.3764061

1 Introduction

Biometric authentication is a technique used to verify an individual's identity based on their physical or behavioural characteristics, such as fingerprints, facial features, iris pattern, voice, or any other unique aspects of a person [31]. These traits are turned into data known as biometric data or embeddings, which are like digital versions of the characteristics, generated through mathematical calculations (in current state-of-the-art often implemented as machine learning models) and stored as a collection (often a simple vector) of numerical values. The verification process consists of comparing two sets of these embeddings with the help of a distance metric that checks how similar they are.

In the case where two distinct parties want to perform this comparison, a simple solution would require that one of them sends their input embedding (sensitive data) to the other. For this problem secure multi-party computation (MPC) can offer promising solutions. MPC eliminates the need for the input of the other party, instead they can perform the calculation collaboratively [11, 41]. This solution can keep their private data secret from each other.

During biometric authentication, the goal is to perform a threshold comparison in a way that preserves privacy, but using MPC primitives for this task is either communication or computation intensive [21]. Many solutions were proposed to address the efficiency problem with these techniques, including Funshade [21] which outperforms its counterparts and uses only one round of communication in the evaluation phase.

Our main contribution is to apply Funshade to a specific distributed systems architecture with practical real-world applicability, and to provide an open source prototype for privacy-first face authentication. We additionally perform a quantitative evaluation of our prototype implementation under realistic assumptions of face authentication use cases.

The structure of the paper is as follows. Section 2 provides a brief summary about the necessary background for Funshade and Section 3 introduces its roles and application both in a general way and applied to the Digidow scenario. Section 4 explores the Rust implementation of Funshade for which Section 5 offers an evaluation. Section 6 gives a review of the other practical solutions that focus on secure biometric matching. Finally, Section 7 concludes the paper and describes future improvements.

2 Preliminaries

The main idea behind Multi-Party Computation (MPC) is to establish a protocol that allows the calculation of a function using secret inputs from different participants [11]. In the end, it prevents any party from learning (at most) more than the output, and no

participant gains access to additional information about the input of the others beyond the final result. This ensures protection of the data from any involved parties seeking to act maliciously or not.

One useful method for MPC is Secret Sharing. It is used to split information into multiple pieces, called shares [24], which on their own do not hold meaningful information. These shares are distributed amongst the parties, who must cooperate to restore the secret. For example, see additive secret sharing: $s = s_0 + s_1$.

Funshade introduced an efficient and privacy-preserving way to perform biometric authentication. During this process, a distance or similarity metric will be calculated and then compared to a predefined threshold. This is the expression whose values it was meant to protect, where $\mathbf{x}$ and $\mathbf{y}$ are the embedding vectors and θ is the threshold.

$$d(\mathbf{x}, \mathbf{y}) \geq \theta$$

Led by the inspiration from GSHADE [6], the metric evaluation is divided into two parts to fit the two-party computation model: two local calculations and a factor of the remaining scalar product.

$$d(\mathbf{x}, \mathbf{y}) = d_{local}(\mathbf{x}) + d_{local}(\mathbf{y}) + d_{sp} \cdot \sum_{i=1}^{l} (x_i \cdot y_i)$$

In this form, the parties can individually calculate the local parts on their input, only the scalar product remained a problematic part.

To protect the inputs during multiplication, Funshade applies secret sharing, more concretely Beaver Multiplication Triples [2]:

$$a \cdot b = c$$

The triples are additively shared and act as kind of "helper" values to assist the parties perform the multiplication securely on additive shares. After receiving the shared triples, the inputs can be masked by them, which the parties exchange with each other. The result (shares) are determined by the following, where $j \in \{0; 1\}$ is the party identifier $d = x - a$ and $e = y - b$ are the added masked inputs:

$$z_j = d \cdot b_j + e \cdot a_j + c_j + j \cdot d \cdot e$$

In addition to these techniques in Funshade, the local evaluation results are secret shared, and even the distance evaluation is further protected by a randomised and secret shared mask value, $r = r_0 + r_1$. This combination of a common value (masked inputs) and shared values (local evaluation) is the secret sharing method (referred to as Π-Secret Sharing) from ABY2.0 [32].

2.1 Function Secret Sharing (FSS)

For the comparison of the distance evaluation with a threshold value, Funshade employs Function Secret Sharing [4, 5]. FSS is similar to other secret sharing methods with the difference that, instead of a value, it is designed to securely split a function into multiple shares. In the case of Funshade, this means that θ will be kept secret. To acquire the result, the shares are evaluated on the same input value; meanwhile, the secret value was not revealed to either party.

Function Secret Sharing is broken down into two phases:

- **Key Generation:** $Gen(1^\lambda, f : \mathbb{G}_{in} \to \mathbb{G}_{out}) \to (k_0, k_1)$, the key generation algorithm takes the given security parameter and the description of a function, producing a pair of function keys (k_0, k_1). The security parameter defines the length of the bit string generated by a pseudo-random number generator (PRNG), the larger the value, the harder it will be for the attacker to break the scheme.
- **Evaluation:** $Eval(j, k_j, x) \to z_j$ is the evaluation phase of the process. As inputs, it takes a party identifier, j, the evaluation key of the party, k_j, and a public input value, x. The output is an additive share of the final result, z_j.

One of the fundamental constructions of FSS is called Distributed Comparison Function (DCF) [4]. The core concept of DCF is a binary tree, where the bit representation of the input value stand as a route to a node. When the node is on the left side of the node that corresponds to the threshold route, the sum of the parties' output will be the specified value (e.g. β); otherwise it yields zero.

$$f : \{0;1\}^n \to \{0;\beta\} \quad f^{<}_{\alpha,\beta}(x) = \begin{cases} \beta & \text{if } 0 \leq x \text{ and } x < \alpha \\ 0 & \text{else} \end{cases}$$

The threshold value and two randomly generated seeds (one for each party) are the basis of the key generation process. At each step during setup, a seeded pseudo-random number generator creates two new leaves with a new seed, an integer, and a boolean value. At the end of each step, so-called "correction words" are generated to guide the additive shares towards a correct value over the course of evaluation. It is important to note, that a third party does not require to build up the whole tree, only the nodes on the input path are calculated both during setup and evaluation.

In Funshade, simply using DCF is not a suitable solution, as not the original value, but a masked one needs to be compared to a specified threshold. For this kind of operation, FSS Gates [4] can be used, more specifically an Interval Containment (IC) Gate. IC Gate performs a DCF key generation that takes a non-zero mask, a minimum, and a maximum value of an interval, followed by two evaluations. In this gate, the mask is used to generate the DCF keys; there is no threshold value. For comparison, θ can be "hidden" in the additive shares of the mask during setup, as the $d(\mathbf{x}, \mathbf{y}) \geq \theta$ comparison can be reformulated as $d(\mathbf{x}, \mathbf{y}) - \theta \geq 0$; in other words, the difference between the metric evaluation and the threshold must be positive.

In the end, the final calculation of the metric looks as follows:

$$z_j = r_{\theta_j} + d_{x_j} + d_{y_j} + f_{cp} \cdot \sum_{j=1}^{l} (j \cdot \Delta_x \Delta_y - \Delta_x \delta_{y_j} - \Delta_y \delta_{x_j} - \delta_{xy_j})$$

2.2 Digidow

In recent years, the application of biometric data for various purposes has become increasingly widespread. Unfortunately, the primary design to implement them uses centralised models [18]. These are convenient and efficient; however, since sensitive data reside in one hand, they pose a significant risk to privacy and security. For example, new modifications of the law in Hungary now authorise the use of face recognition during assemblies, demonstrations, and protests [19].

Digidow [7, 27] addresses the risk posed by centralised biometric storage. Rather than relying on single institutions for digital identity management, it offers a decentralised model.

There are three main participants in the Digidow model:

- **Personal Identity Agents (PIA):** The PIA is associated with an individual; the PIA stores the individual's digital identity cedentials and allows them to interact with services and control where to host their agent, ensuring their (biometric and other) identity data and transaction history remain under their control.
- **Sensor:** The Sensor is responsible for detecting characteristics of a person and translating it to a biometric template.
- **Verifier:** The Verifier checks the information provided by the PIA (signed digital credential attributes relevant to the interaction) and ensures that a trusted sensor was used. It will generally trigger a physical component, such as an electronic door lock.

2.3 Tor

The main goal of the Tor [15, 37] network is to maintain privacy during communication between the caller and the receiver over an open network, for example, between a user and a web server, even under the assumption of a global passive adversary

The caller sends a signal through several routers, called Onion Routers, which are volunteer-run servers. Each router maintains a connection to all the other routers in the network. Tor encrypts the data 3 times and routes the traffic through 3 routers, each decrypting a layer ("peeling off a layer").

Onion services are websites or applications that are only accessible within the Tor network. In this case, it does not transmit data *through* the network to an *outside service*, but *towards* an end-to-end encrypted service *within*. These services are reachable by their onion addresses that help Tor find and route traffic privately to them, without revealing their operators.

2.4 Arti

Arti [36] is a project that aims to reimplement Tor in Rust. The current implementation of Tor uses C, and the team behind it estimates that at least half of their tracked security vulnerabilities would have been impossible in Rust and many would have been unlikely [36]. They also claim that Rust enables faster development because of its expressiveness and strong guarantees.

At the moment, Arti is still in active development. It still contains APIs that are not stable, and there are likely to be bugs in the implementation. Despite this, Arti is available to everyone for testing and experimentation.

3 Application of Funshade for Embedding Comparison

Funshade distributes the computation between several roles, each responsible for a specific task. Multiple roles can be assigned to the same party but can also be different entities. The process can be separated into two phases: an offline phase and an online phase. The offline phase is more like a preparation phase where no actual input data is processed yet or is not available at all; its main purpose is to precompute certain values. The online phase takes place after the offline phase and begins when the actual input data is available. In MPC, the main point of the separation is to minimise the delay during the actual computation on the information, allowing computations to be made more efficient by preparing certain operations in advance.

The roles are as follows:

- R_{setup}: The party with the setup role covers the generation of the necessary values during the offline phase. It distributes the Beaver Triple shares between the input holders and the generated keys between the ones performing the evaluation.
- R_{in_x}, R_{in_y}: These roles are assigned to data holders with access to the input. Their responsibility is to prepare the Π-secret shares of their input and the local distance evaluation shares. When they are done, they forward to the evaluating parties. This computation can even take place during the offline phase (after setup) if the input vectors are already available.
- P_x, P_y: These parties are responsible for the evaluation part during the online phase. After receiving the shares of the local evaluation and the Π-shares from the input parties, they use their keys to calculate the additive shares of the masked distance. Then, after exchanging these shares, they can evaluate the IC Gate.
- R_{res}: This party receives the shares of the result and uses them to reconstruct the result.

The setup phase can be handled in three different ways: with a trusted hardware, a trusted third party, or with two parties that perform it collaboratively.

Funshade can be applied to distributed constructions, such as to a Digidow [7, 27] biometric authentication scenario. We will use Digidow as a specific use-case throughout the paper to illustrate how it works and improve understanding. In the ideal scenario, the PIA and the sensor perform the setup collaboratively and the roles are issued as follows:

- PIA: Setup R_{setup_y}, Input holder R_{in_y}, and Evaluator P_y.
- Camera (Sensor): Setup R_{setup_x}, Input holder R_{in_x}, and Evaluator P_x.
- Lock (Verifier): Result role R_{res}.

For the sake of an example, assume that Alice is authorised to access a secure room guarded by the Lock component, acting as the Verifier. Her PIA stores her biometric template that needs to be compared to the one detected by the Sensor, e.g., a Camera. Figure 1 shows the process step by step:

(1) In the offline phase, the PIA and Camera collaboratively perform the setup. Both parties compute the required materials. After finishing the process, the PIA and Camera own δ_j and a Funshade key k_j.
(2) Since the PIA already has the biometric template of Alice, y, it can precompute the outstanding values of the Π-share, Δ_y and the additive shares of the local distance function evaluation $d_{local}(\mathbf{y})$. The PIA sends Δ_y and d_{y_1} to the Camera and both are waiting for somebody to approach the door.
(3) When someone passes by, the online phase begins. The camera captures live biometric data of the person, x, and generates the corresponding shares that the PIA had done beforehand.
(4) The construction of additive shares for the distance evaluation begins as both parties possess the secret shares derived

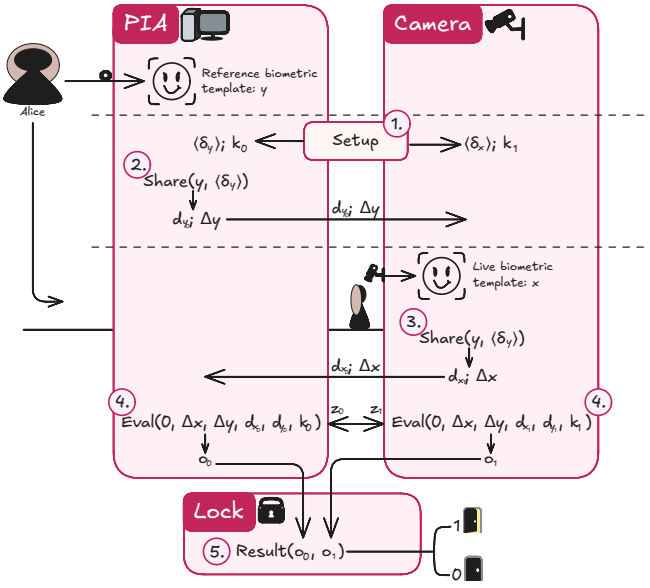


Figure 1: Application of Funshade in a Digidow scenario. Image is based on [21].

from the inputs **x** and **y**. At this point, they require a round of communication to produce the common value that represents the result of the masked distance calculation. To conclude the process, both evaluate the Interval Containment gate and send the values to the Verifier.

(5) The Lock combines the values and either grants or denies access to the person standing at the door. For example, if this person is Alice, the door will open.

4 Implementation of Funshade

For this project, Rust [35] was chosen due to its memory safety features. It is a modern, system-level programming language that emphasises safety, performance, and concurrency [23]. It has a distinct ownership system that prevents memory-related errors that are common sources of bugs. It was designed to solve many common problems found in older languages such as C and C++, such as dangling pointers and null pointer dereferences [3].

This section focusses on our Rust implementation of Funshade, for which the source code is available in Appendix A. As it stands now, only one of the scenarios is implemented, that is, when the setup happens with the help of a third-party. For DCF and basic functions, the original C project [20] created by Ibarrondo et al. was the inspiration, such as `setup`, `share`, `eval_scalar`, `eval_sign` and `res`. In the codebase, these can be found in specified modules (`fss/dcf`, `fss/ic` and `funshade`) to enhance navigation throughout the project.

During development, it was important to create the code in a way that is easily readable and maintainable. To achieve this, some of the tuples mentioned by Boyle et al. [4] are ordered in their own data structures together with their unique operations. For example, such a tuple in the paper is the $s^L||v^L||t^L||s^R||v^R||t^R \leftarrow G(s^{(i-1)})$ random tuple in each iteration, which corresponds to our `DcfNode` structure. Otherwise, small changes needed to be made in all three parts because of differences in programming languages.

4.1 Scaling

In the end of Section 2.1, it was mentioned that "the difference between the metric evaluation and the threshold must be positive". Looking at this from a practical point of view, this means that in a limited range of numbers, such as `i32`, overflows might occur. Without constraining the input values, this might lead to incorrect results.

The solution is to scale the input values to a proper range. Concerning the exact values, a test in the C implementation [20] provided a hint in `test_funshade.py` file:

```
Maximum value for vector elements. To avoid overflows:
2*log2(max\_el) + log2(l) <= 32 bits
```

To this end, scaling methods are implemented that are specifically tailored around the parameters given to Funshade. These functions were originally developed for testing, but as the code base grew and was extended with a `FunshadeSettings` data structure, they became available to the user. The only task of this structure is to hold onto data that are commonly used in different parts of the process, making it easier to pass around.

The maximum acceptable value in an input vector, as can be seen in the aforementioned hint, depends on the length of the vector and the specified number of bits. Furthermore, the distance metric used, the formula, also strongly influences this calculation. Think about multiplication and addition on the same number set; multiplication on the values approaches a limit much faster than addition. The limit in our case is the maximum integer value and the operations are specified by the metric. For example, for Scalar Product, the one used in the C implementation (or for Cosine Distance on normalised inputs), the maximum value can be calculated as follows:

$$
\begin{aligned}
l \cdot (m \cdot m) &\leq 2^n \\
l \cdot m^2 &\leq 2^n \\
\log_2 l + \log_2 m^2 &\leq n \\
\log_2 l + 2\log_2 m &\leq n \qquad \text{same as in the hint} \\
2\log_2 m &\leq n - \log_2 l \\
\log_2 m &\leq \frac{n - log_2 l}{2} \\
m &\leq 2^{\frac{n-log_2 l}{2}}
\end{aligned}
$$

The left-hand side expression comes from the formula of the Scalar Product, where pairs of values are multiplied and then added together. In this case, the maximum product can be achieved when the two values are equal to the maximum. The sum of these needs to avoid overflow, or in other words, remain under the maximum of the value range.

However, there is one problem with this equation. In an integer range, there is no way for a value to reach 2^n as it will overflow; it became the minimum value of the range and will produce incorrect result. During our tests on randomly generated data, this was very rare to happen when 32 bits were used, but on 8 bits, usually it failed around the 100th iteration at the latest. We corrected the calculation and tests using $<$ rather than $\leq$. In this way, the final

form of the above example is as follows:

$$m < 2^{\frac{n-log_2 l}{2}}$$

4.2 Distance Metrics

Ibarrondo et al. [21] have already provided a reformulation of some distance metrics in their paper. As in GSHADE [6], the metrics are separated into two local evaluations and a cross-product factor.

This interpretation is implemented by our `DistanceMetric` trait. In the initial phases, this was created as a data structure with an integer field and a `Box`-ed function. Re-implementing them as traits is deemed more "heapless", which is preferred in Rust and should always be the goal when using a heap is not necessary. This means that we do not create pointers to a `Box`-ed instance in a heap that points to a static function definition. Furthermore, it is more user-friendly in the sense that omitting `Box` seems more readable and gives the user more flexibility.

In this way, the user can implement the trait on their own structure, implement their preferred distance metric, and easily use additional data, for instance, a matrix in Squared Mahalanobis. Achieving the same with functions that have a static header is impossible as they can not take out values from their environment, but with closures is harder, more complex, and might cause problem, when parallelised,

The trait has an additional mandatory function, which is needed for calculating the maximum element of the input vector as they depend on the distance metric. The project provides two pre-implemented metric, the Scalar Product and the Squared Euclidean, which are the most common ones used for comparing face embeddings.

4.3 Party Structure

Our implementation includes common parties to provide users with ready-to-use interfaces. As the implemented scenario is the one with a third-party, the currently available parties are a) a setup party; b) one that receives the result; and c) the two input holders who also perform the evaluation (in Digidow terms, they are called the Sensor and the PIA).

The parties are implemented in the form of structures to be able to store the necessary data, for example the role-specific shares. In order to use them, the user needs to instantiate them and call one of their publicly reachable functions. Rather than a single static instance, each party maintains individual states for every connected party. The different sessions do not share the same shares and keys, as the generated key pairs cannot be reused in a connection to a different PIA. An example in Figure 2 shows that for each connection, the sensor needs to handle their key of the key-pair properly. Therefore, encapsulating the affiliated data, such as keys and shares, makes their management easier for the user.

Each of these objects offers various functionalities, including:

- Protocol initialisation (`init_protocol`): This method is responsible for exchanging configuration data between the parties, such as the `embedding_size`, ensuring that all participants work with the same parameters. This is an optional, but not a mandatory step.
- Protocol setup (`setup_protocol`): Perform the setup phase of the protocol.

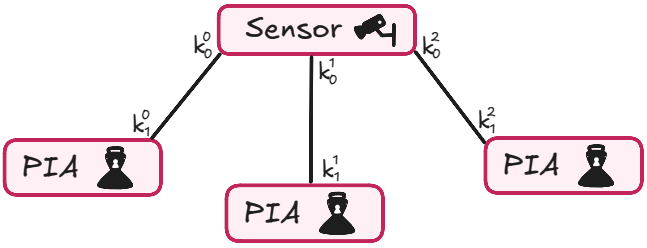


Figure 2: Multiple active sessions in a sensor.

- Abort protocol (`abort_protocol`): In cases where the computation cannot continue or an error is detected, this method can be invoked to stop the process.
- Handling messages (`handle_funshade_message`): Any message received from another party can be passed directly to the object, which will automatically process the message based on its type and role in the protocol.

The Sensor party has an essential role in initiating the execution. The `start_matching` method allows to trigger the protocol execution, when live data becomes available. This ensures that the computation does not begin until all the necessary conditions are met. The PIA stores its input data, the long-term biometric embedding, which can be updated as needed using `change_input`. Additionally, the PIA can use the `calculate_shares` method to precompute its shares before the protocol begins, reducing the overhead during the online phase.

For simplicity sake, users of our library implementation can only use these few public functions. In this way, the user does not need to handle each message themselves and know what to call when any data arrives, but instead the objects manage their internal logic during the protocol based on the message they received.

As there are no specific functions to call like `eval_distance`, the user might not know if there is a need to send a message to the other party or not. To mitigate this problem, the library takes on this responsibility by performing this when needed. Even if this task became managed by the library, our aim was still to provide this as flexibly as possible. Fortunately, Rust supports generic parameters, which makes it possible to store an object of arbitrary type, such as `TcpStream`. The parties require these for each connection, which in the implementation we marked as `CommunicationPartner` types, so it will know where to send. As these objects might behave differently, they still need a function pointer to a "send" function implemented by the user. For example, the send function using a raw TCP stream and another that uses HTTP will not be identical.

5 Evaluation

In this section, we present the performance measurement and communication overhead of our implementation under different conditions. The goal is to quantify (I) the runtime of the functions in Funshade [21], (II) the size of the messages that are exchanged during execution, and (III) how the performance scales with different embedding lengths. The experiments cover local TCP communication on the same machine and traffic routed via the Tor network to explore different deployment scenarios.

The benchmarks are created with a recent (2022) benchmarking crate, called Divan [39]. It was chosen because it is relatively easy to use and for its feature that enables benchmarking private methods. It is constantly under development/enhancements.

Along with the benchmarks, several unit tests and a few integration tests were written for every part of the project. In this way, we could identify errors in the code, such as the mistake with the relation sign during the maximum element calculation.

5.1 Experimental Setup

All experiments were performed on a machine with an Intel® Core™ i5-11400F x 12 processor and 16 GB of DDR4 memory, using Ubuntu 24.04.2 LTS 64-bit operation system. The implementation was compiled with `rustc` 1.88.0 (2025-06-23). For network measurements, we used WireShark 4.2.2 [40] to capture the total size of the messages.

Each experiment was repeated multiple times on a single thread, for which Divan automatically specified the exact values on the basis of the performance. This means that Divan repeated the whole benchmark method multiple times and in each it did the same for the small parts to be measured.

5.2 Runtime Performance

First we measured the performance of the function in Funshade and compared it to the original implementation (see Table 1). The benchmarks were run on a single thread with similar parameters as Ibarrondo et al. [21]; $n = 32$ number of bits, $l = 128$ number of embedding vector elements and $\lambda = 128$ security parameter (for pseudo-random generation in DCF).

When measuring the performance with network communication, we split the process into two parts: the online and offline phase. Figure 3 visualises which data are sent in the corresponding phases (and in what order):

- **Offline Phase**: The trusted third party (TTP) generates the triples $\langle \delta_x \rangle, \langle \delta_y \rangle$ along with the keys k_0, k_1 and sends them to the two parties. As the PIA, marked as P_1, already knows their input data, it masks the input vector, Δ_y, calculates the additive shares of the local distance function, d_y and sends the (d_{y_0}, Δ_y) tuples to the other.
- **Online Phase**: P_0 receives an embedding that marks the start of the online phase, which in Digidow terms means that the Sensor receives their live embedding x. It performs the same operations as the PIA did, calculates the shares d_{x_0} and d_{x_1}, masks the input, Δ_x, and sends them. Both compute their share of the distance evaluation, z_j, which they can restore with a round of communication. After performing the comparison, they send the result o_j to the party with the Result role.

Table 1: Protocol measurements of the C and Rust library.

	Funshade C Reported in [21]	Funshade C Measured	Funshade Rust*
Setup	33.63μs	138.3μs	16.2μs
Share	0.038μs	0.050μs	0.71μs
Eval Distance	0.19μs	0.26μs	0.048μs
Eval IC Gate	9.2μs	8.71μs	9.376μs
Result	–	–	0.001μs

*The median of 100 samples and 3200 iterations.

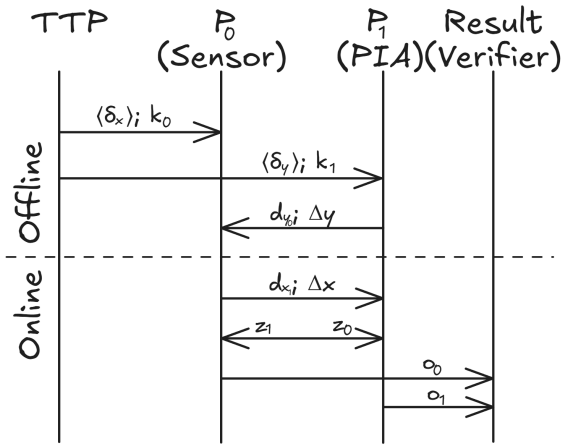


Figure 3: Setup with a trusted third-party. The P_1 party conforms to the PIA, while P_0 to the Sensor. The Verifier acts as the party who receives the results and in this setup, could potentially be the trusted third-party.

In Table 2 two communication settings were considered: local TCP connections on the same machine and over the Tor network [1]. As shown, the offline phase, more specifically the setup, accounts for most of the runtime. For the TCP case, we turned off Nagle's algorithm [30] to send segments as soon as possible; otherwise, some of the smaller messages would increase the cost by at least 80ms by default. When it is turned on, it waits for a sufficient amount of data to avoid sending frequent small packages. To implement the Tor use case, we used Arti with an HTTP client using the crate called Hyper [28]. The setup phase makes up the majority of the runtime in all three cases, even with the additional overhead caused by Tor. The last measurement we made has measured the whole execution; therefore, the table presents the benchmark result instead of the exact sum of the offline and online phase.

5.3 Message Sizes

In this section, we first estimate the theoretical size of the messages exchanged in the protocol. The following messages were implemented in the project and measured:

- `Setup`: Sent from either P_0 or P_1 to the third party, telling them to start the setup. Optionally, the party can perform the setup even from its own volition. This is an enum element, which is serialised as strings using the Serde crate [38]. Therefore, the size will be 7 bytes.

[1] For Tor, the whole process measurement was not performed, as the total duration is not meaningful, due to external events and asynchronous processing. The benchmark would include a loop that checks if the sensor can start the online phase or not, which could influence the result.

Table 2: Library benchmarks.

	Pure Computation	TCP	Tor Arti + Hyper
Offline Phase	23.34μs	216.2μs	4.15s
Online Phase (Sensor side)	17.37μs	88.42μs	3.08s
Whole	49.25μs	327.8μs	–

- `DeltaShare`: The `DeltaShares` and `Key` are counted as different messages, as in other scenarios they might be sent to two different parties. This separation ensures that the project is more easily extendible with other kinds of parties, for example, separate Sensor and Eval parties. They are sent to P_0 and P_1 to hide their input.
  These shares consist of two l length vectors with integer elements. The size in bits is calculated as $2 \cdot l \cdot n$.
- `Key`: The keys are sent to P_0 and P_1 for the evaluation phase. They include the Beaver Triples, the mask, correction term for the IC Gate (which would handle a possible overflow in the ranges) and the DCF key. This key consists of a seed, an integer, and an n-length correction word vector, each with a seed, an integer, and two boolean values (which are stored on 8 bits in Rust). In both cases, the length of the seeds is λ. All put together, the size can be calculated as follows: $[3 \cdot l \cdot n + n + [n + [\lambda + n \cdot (\lambda + n + 8 + 8) + n]]]$, where we mark the borders of the nested keys with [] brackets for clearance sake (see in Figure 4).
- `Start`: P_0 sends this message to P_1 to mark the beginning of the online phase. This message is also an enum element, whose size is 7 bytes.
- `DShare` or in other terms, the Π-secret shares: They wrap the shares of the local evaluation and the hidden embedding vector together. One is stored, and the other is sent to the other party. Therefore, the size is $l \cdot n + n$.
- `ZValue`: These are the values that the two parties exchange in one round during the online phase, the additive shares of the masked distance. They are an integer value; hence the size is n.
- `OValue`: The additive shares of the final result are marked as o by Ibarrondo et al.. Similarly to z, they are integers of size n.

Table 3 presents the message types that we implemented and their costs where the number of bits $n = 32$, the security parameter

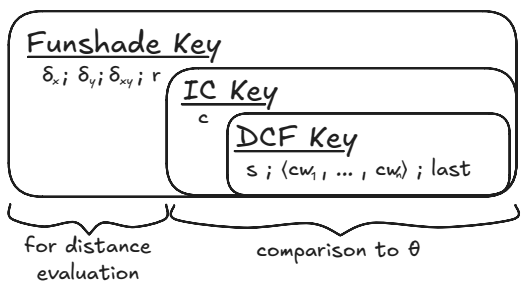


Figure 4: The nested keys.

Table 3: Size measurement using different communication methods. The TCP and Tor with JSON options could not give a constant size, due to the string conversion of the values. For example the number 66 will be smaller (2 B) in terms of storage than 166 (3 B). Consequently, we give an approximation of the observed sizes.

	Estimation	TCP JSON	Tor JSON	TCP Postcard
`Setup`	7 B	75 B	276 B	69 B
`DeltaShare`	4096 B	~11.3 KB	~11.34 KB	5.13 KB
`Key`	6876 B	~20.8 KB	~20.9 KB	8.43 KB
`Start`	7 B	75 B	276 B	69 B
`DShare`	2052 B	~5.7 KB	~5.75 KB	2.6 KB
`ZValue`	4 B	~289 B	~292 B	74 B
`OValue`	4 B	~296 B	~295 B	75 B

$\lambda = 128$ and the embedding length are $l = 512$, as most state-of-the-art papers about biometrics use embedding sizes of 512 elements [14, 18]. For TCP and Tor, we serialised the data to JSON format (and sent the data in that way). We also included measurements, where we changed the serialisation method to Postcard [29], which aims to be efficient and support as many Serde features as possible, therefore making the transition from Serde easier.

When using JSON, objects are converted to string format, which causes significant overhead. Although unreadable, converting the data into bytes instead of strings reduces the size of the messages. For example, `{"ZValue":-378342174}` is the the JSON formatted value z_j with an approximate size of 289B (including metadata). Meanwhile, the other method only takes 74B, but the value results in unreadable bytes: `09 8e b3 ed cb 0c`.

In Table 4, we present the total amount of the messages the parties store, send and receive. The values are based on the following calculations:

- **Store**: TTP stores the θ and the length of the embeddings, which are two integer numbers, while the Verifier only stores an integer, the `OValue`. The calculation for the PIA and the Sensor is the same, they need to preserve most of the values they receive along with their identifiers, the length of the embeddings and the inputs:
  $|j| + |l| + 4l$ + `|Key|` + `|DeltaS|` + `|DShare|` + `|ZValue|`.
- **Send**: The TTP sends two `Keys` and two `DeltaShares`, which are received by the PIA and the Sensor. They both send a `|DShare|`, a `|ZValue|` and an `|OValue|` messages, except the Sensor who additionally gives signal to the PIA to `Start` the calculation. The Verifier does not send anything.

Table 4: The total estimated stored, sent and received data sizes of different parties.

	TTP	Sensor, P_1	PIA, P_1	Verifier, P_{Result}
Store	8 B	14.73 KB		4 B
Send	21.43 KB	2.02 KB	2.01 KB	-
Receive	0 B or 7 B	12.72 KB	12.73 KB	8 B

Table 5: Estimated and measured message sizes for different embedding lengths.

Length (l)	`DeltaShare`	`Key`	`DShare`
1	8 B	744 B	8 B
2	16 B	756 B	12 B
4	32 B	780 B	20 B
8	64 B	828 B	36 B
16	128 B	924 B	68 B
32	256 B	1.1 KB	132 B
64	512 B	1.5 KB	260 B
128	1 KB	2.3 KB	516 B
256	2 KB	3.8 KB	1 KB
512	4.1 KB	6.87 KB	2.1 KB
1024	8.2 KB	13 KB	4.1 KB

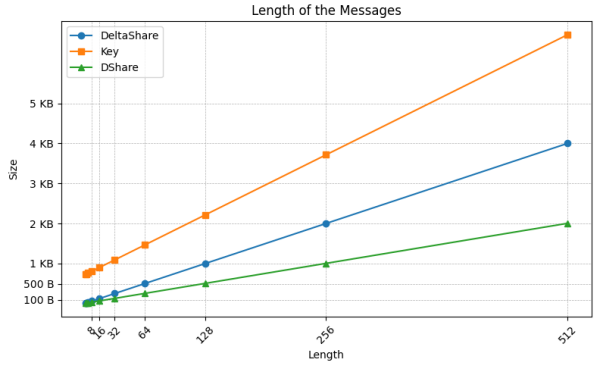


Figure 5: Growth of messages with variable length.

- **Receive**: The TTP may receive a `Setup` command from either of the parties. The sensor and PIA receives the `Key` and `DeltaShare` after setup with a `|DShare|` and `|ZValue|` from each other. The PIA additionally receives the `Start` message. The Verifier only gets the two final result.

Table 5 summarizes how the different embedding lengths influence the message sizes. We only measure those that are influenced by the length.

Figure 5 visualises the estimated size for each message. All of the messages grow linearly. `DShare` increases more slowly, while `Key` increases faster than `DeltaShare`.

5.4 Scalability

As the evaluation on both sides might occur at the same time, we built our benchmark in a way that it does not measure them twice. Following the recommendation of Dian, we measured the runtime over 100 samples and 3200 iterations, where the runtime grows proportional to the size of the embedding vectors.

We implemented Funshade in a way, to allow the user to specify the input bit size for DCF, in other words, they can modify the size of the tree. When using a reasonable value, such as 24 levels for input data elements stored in the 8 bit range as proposed by Hofer [18], the differences in overall performance are insignificant. The core computation still relies on the usual integer types, like `i32`, because performing wrapping operations after every calculation can cause overhead. However, in possible future implementations, it might matter more. For example, one of the two-party solutions for DCF construction [4, 16] requires building the whole tree in the setup phase. In these scenarios, the user can compromise between the value of the maximum element and the runtime based on their use cases.

6 Related Work

In this section, we show other solutions to biometric matching in distributed systems and introduce a few other possible solutions, where Funshade gains insights from: AriaNN, GSHADE, and ABY2.0.

AriaNN [33] focusses on secure neural network computations, minimising communication to two rounds in the online phase by combining Beaver Triples, additive secret sharing and FSS. Although AriaNN might appear more efficient due to single DCF evaluation, its two-round communication could introduce enough practical overhead [21].

GSHADE [6] computes distance metrics without revealing private data by splitting the computations into two local steps and a scalar product, solved by Oblivious Transfer in two rounds. Funshade adopts the GSHADE metric handling, but reduces communication to a single round.

ABY2.0 [32] introduces Π-Secret Sharing, allowing private input to be shared for distance calculations. Although efficient, the scalar product evaluation in ABY2.0 lacks constant-round communication, unlike Funshade.

These works inspire Funshade's design. AriaNN emphasises reducing communication rounds, GSHADE shows secure distance computation, and ABY2.0 offers secure joint computations. Funshade combines these benefits into a single-round protocol suitable for latency-sensitive applications like Digidow.

Luo et al. [26] proposed an anonymous biometric access control using homomorphic encryption to protect user data during encrypted similarity searches. However, such schemes involve extra communication rounds and computational overhead, as seen also in Ghostshell [8] and THRIVE [22].

SEMBA [1] combines multiple biometrics using SPDZ [13], performing computations without revealing the inputs. However, it requires two communication rounds per iteration, unlike the single-round design in Funshade.

Lee et al. [25] use homomorphic encryption and function-hiding inner product encryption (FFB-IPE) for privacy-preserving Hamming distance evaluation. Similar approaches in PassBio [42] and by Chun et al. [9] also rely on encrypted data, which adds computational overhead that is unsuitable for Digidow, where a malicious PIA or sensor could influence results more directly than in server-operated contexts.

Takahashi et al. [34] propose fuzzy private keys that tolerate biometric variability, though with limits on how often new keys can be issued if secrets are compromised.

7 Conclusion and Future Work

Our measurements confirm that the protocol achieves reasonable performance. The message sizes scale predictably with the embedding length and parameter choices, allowing informed tuning based on application requirements.

Although this implementation of Funshade achieves its initial goals, there are still numerous areas where it can be improved.

One of the major opportunities lies in the refinement of the setup phase, where the parties collaboratively compute the DCF keys. Boyle et al. already provided a solution for this problem [4, 16], which was further improved by Guo et al. in their Half-Tree paper [17]. To also address potential attacks from compromised parties, a similar solution that is applied in SPDZ [13] might be suitable, where they check honest execution with MAC codes [4].

From an implementation perspective, one addition could be to allow the user to choose their own PRNG or use the default AES-PRNG [12]. Furthermore, implementing the type state pattern [10] might improve the formal verification process of the library, which is outside the scope of this paper but an explicit target of future research to support deployment in regulated scenarios such as next iterations of the European Digital Identity Wallet (EUDIW).

Acknowledgments

This work has been carried out within the scope of Digidow, the Christian Doppler Laboratory for Private Digital Authentication in the Physical World. We gratefully acknowledge financial support by the Austrian Federal Ministry of Economy, Energy and Tourism, the National Foundation for Research, Technology and Development, the Christian Doppler Research Association, 3 Banken IT GmbH, ekey biometric systems GmbH, Kepler Universitätsklinikum GmbH, NXP Semiconductors Austria GmbH & Co KG, and Österreichische Staatsdruckerei GmbH.

References

[1] Mauro Barni, Giulia Droandi, Riccardo Lazzeretti, and Tommaso Pignata. 2019. SEMBA: secure multi-biometric authentication. *IET Biometrics* 8, 6 (2019), 411–421. doi:10.1049/iet-bmt.2018.5138

[2] Donald Beaver. 1992. Efficient Multiparty Protocols Using Circuit Randomization. In *Advances in Cryptology — CRYPTO '91 (LNCS, Vol. 576)*. Springer, Berlin, Heidelberg, 420–432. doi:10.1007/3-540-46766-1_34

[3] Jim Blandy, Jason Orendorff, and Leonora F. S. Tindall. 2021. *Programming Rust* (2 ed.). O'Reilly Media, Inc.

[4] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. 2020. Function Secret Sharing for Mixed-Mode and Fixed-Point Secure Computation. Cryptology ePrint Archive, Paper 2020/1392. https://eprint.iacr.org/2020/1392

[5] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2018. Function Secret Sharing: Improvements and Extensions. Cryptology ePrint Archive, Paper 2018/707. https://eprint.iacr.org/2018/707

[6] Julien Bringer, Herve Chabanne, Melanie Favre, Alain Patey, Thomas Schneider, and Michael Zohner. 2014. GSHADE: faster privacy-preserving distance computation and biometric identification. In *Proceedings of the 2nd ACM workshop on Information hiding and multimedia security* (Salzburg, Austria) *(IH&MMSec '14)*. ACM, 187–198. doi:10.1145/2600918.2600922

[7] CDL Digidow. 2024. *Digidow: Christian Doppler Laboratory for Private Digital Authentication in the Physical World.* Retrieved 2024-10-03 from https://www.digidow.eu/

[8] Jung Hee Cheon, HeeWon Chung, Myungsun Kim, and Kang-Won Lee. 2016. Ghostshell: Secure Biometric Authentication using Integrity-based Homomorphic Evaluations. Cryptology ePrint Archive, Paper 2016/484. https://eprint.iacr.org/2016/484

[9] Hu Chun, Yousef Elmehdwi, Feng Li, Prabir Bhattacharya, and Wei Jiang. 2014. Outsourceable two-party privacy-preserving biometric authentication. In *Proceedings of the 9th ACM symposium on Information, computer and communications security* (Kyoto, Japan) *(ASIA CCS '14)*. ACM, 401–412. doi:10.1145/2590296.2590343

[10] Cliff L. Biffle. 2019. *Cliffle: The Typestate Pattern in Rust.* Retrieved 2025-07-11 from https://cliffle.com/blog/rust-typestate/

[11] Ronald Cramer and Ivan Damgård. 2005. Multiparty Computation, an Introduction. In *Contemporary Cryptology*. Birkhäuser Basel, Basel, 41–87. doi:10.1007/3-7643-7394-6_2

[12] Morten Dahl and Dragoș Rotaru. 2024. *aes-prng - crates.io: Rust Package Registry.* Retrieved 2024-10-21 from https://crates.io/crates/aes-prng

[13] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. 2011. Multiparty Computation from Somewhat Homomorphic Encryption. Cryptology ePrint Archive, Paper 2011/535. https://eprint.iacr.org/2011/535

[14] Jiankang Deng, Jia Guo, Niannan Xue, and Stefanos Zafeiriou. 2019. ArcFace: Additive Angular Margin Loss for Deep Face Recognition. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (Long Beach, CA, USA). IEEE, 4685–4694. doi:10.1109/CVPR.2019.00482

[15] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. *Tor: The Second-Generation Onion Router.* Technical Report ADA465464. Defense Technical Information Center. doi:10.21236/ADA465464

[16] Jack Doerner and Abhi Shelat. 2017. Scaling ORAM for Secure Computation. Cryptology ePrint Archive, Paper 2017/827. https://eprint.iacr.org/2017/827

[17] Xiaojie Guo, Kang Yang, Xiao Wang, Wenhao Zhang, Xiang Xie, Jiang Zhang, and Zheli Liu. 2022. Half-Tree: Halving the Cost of Tree Expansion in COT and DPF. Cryptology ePrint Archive, Paper 2022/1431. https://eprint.iacr.org/2022/1431

[18] Philipp Hofer. 2024. *Enhancing Privacy-Preserving Biometric Authentication through Decentralization.* phdthesis. Johannes Kepler University, Linz. Retrieved 2024-10-31 from https://resolver.obvsg.at/urn:nbn:at:at-ubl:1-79283

[19] Hungarian Parliament. 2025. 2025. évi III. törvény a gyülekezési jogról szóló 2018. évi LV. törvény módosításáról. 6 pages. Retrieved 2025-06-25 from https://magyarkozlony.hu/dokumentumok/46d496a57e0e6ac221c4acecfa5cc0830415c746/megtekintes

[20] Alberto Ibarrondo. 2024. *Funshade.* Retrieved 2024-10-22 from https://github.com/ibarrond/funshade

[21] Alberto Ibarrondo, Hervé Chabanne, and Melek Önen. 2022. Funshade: Function Secret Sharing for Two-Party Secure Thresholded Distance Evaluation. Cryptology ePrint Archive, Paper 2022/1688. https://eprint.iacr.org/2022/1688

[22] Cagatay Karabat, Mehmet Sabir Kiraz, Hakan Erdogan, and Erkay Savas. 2015. THRIVE: threshold homomorphic encryption based secure and privacy preserving biometric verification system. *EURASIP Journal on Advances in Signal Processing* 2015, 1 (Aug. 2015), 71. doi:10.1186/s13634-015-0255-5

[23] Steve Klabnik and Carol Nichols. 2024. *The Rust Programming Language - The Rust Programming Language.* Retrieved 2024-10-28 from https://doc.rust-lang.org/book/title-page.html

[24] Stephan Krenn and Thomas Lorünser. 2023. *An Introduction to Secret Sharing* (1 ed.). Springer, Cham. doi:10.1007/978-3-031-28161-7

[25] Joohee Lee, Dongwoo Kim, Duhyeong Kim, Yongsoo Song, Junbum Shin, and Jung Hee Cheon. 2018. Instant Privacy-Preserving Biometric Authentication for Hamming Distance. Cryptology ePrint Archive, Paper 2018/1214. https://eprint.iacr.org/2018/1214

[26] Ying Luo, Sen-ching S. Cheung, and Shuiming Ye. 2009. Anonymous Biometric Access Control based on homomorphic encryption. In *2009 IEEE International Conference on Multimedia and Expo* (New York, NY, USA, 2009-06-01). IEEE, 1046–1049. doi:10.1109/ICME.2009.5202677

[27] René Mayrhofer, Michael Roland, Tobias Höller, Philipp Hofer, and Mario Lins. 2025. An Architecture for Distributed Digital Identities in the Physical World. arXiv:2508.10185 [cs.CR] https://arxiv.org/abs/2508.10185

[28] Sean McArthur. 2025. *hyper - crates.io: Rust Package Registry.* Retrieved 2025-06-16 from https://crates.io/crates/hyper

[29] James Munns. 2025. *postcard - crates.io: Rust Package Registry.* Retrieved 2025-07-03 from https://crates.io/crates/postcard

[30] John Nagle. 1984. *Congestion Control in IP/TCP Internetworks.* Request for Comments RFC 896. Internet Engineering Task Force. 9 pages. doi:10.17487/RFC0896

[31] Joseph Nelson, CPP. 2013. *Chapter 12 - Biometrics Characteristics* (4 ed.). Butterworth-Heinemann, 255–256. doi:10.1016/B978-0-12-415892-4.00012-2

[32] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2020. ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. https://eprint.iacr.org/2020/1225
[33] Théo Ryffel, Pierre Tholoniat, David Pointcheval, and Francis Bach. 2022. AriaNN: Low-Interaction Privacy-Preserving Deep Learning via Function Secret Sharing. *Proceedings on Privacy Enhancing Technologies* 2022 (Jan. 2022), 291–316. doi:10.2478/popets-2022-0015
[34] Kenta Takahashi, Takahiro Matsuda, Takao Murakami, Goichiro Hanaoka, and Masakatsu Nishigaki. 2019. Signature schemes with a fuzzy private key. *International Journal of Information Security* 18, 5 (Oct. 2019), 581–617. doi:10.1007/s10207-019-00428-z
[35] The Rust Team. 2024. *Rust Programming Language*. Retrieved 2024-10-28 from https://www.rust-lang.org/
[36] The Tor Project. 2020. *Arti*. https://tpo.pages.torproject.net/core/arti/
[37] The Tor Project. 2025. *The Tor Project | Privacy & Freedom Online*. Retrieved 2025-07-02 from https://torproject.org
[38] David Tolnay. 2024. *serde - crates.io: Rust Package Registry*. Retrieved 2024-10-21 from https://crates.io/crates/serde
[39] Nikolai Vazquez and Thom Chiovoloni. 2025. *divan - crates.io: Rust Package Registry*. Retrieved 2025-06-16 from https://crates.io/crates/divan
[40] Wireshark Foundation. 2025. *Wireshark*. https://www.wireshark.org/
[41] Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)* (Toronto, ON, Canada). IEEE, 162–167. doi:10.1109/SFCS.1986.25
[42] Kai Zhou and Jian Ren. 2018. PassBio: Privacy-Preserving User-Centric Biometric Authentication. *IEEE Transactions on Information Forensics and Security* 13, 12 (Dec. 2018), 3050–3063. doi:10.1109/TIFS.2018.2838540

A Source Code

The description aligns with the implementation that is attached to this PDF and can be downloaded by clicking on the following icon[2]:

 .

[2]Some PDF readers may not support this, e.g., the automatic PDF reader of the Opera browser.