JOHANNES KEPLER
UNIVERSITY LINZ

Author
Cintia Maja Bódi, BSc
12322483

Submission
Institute of
Networks and Security

First Supervisor
Univ.-Prof. DI Dr.
René Mayrhofer

Second Supervisor
Dr. **Gergely Kovásznai**

Assistant Thesis
Supervisor
Dr. **Philipp Hofer**

December 2024

Privacy-Preserving Biometric Matching via Secure Two-Party Computation

Master's Thesis

to confer the academic degree of

Diplom-Ingenieurin

in the Master's Program

Computer Science

JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at

Abstract

This thesis presents a detailed exploration of Funshade, a framework designed to enable secure biometric authentication through privacy-preserving protocols. With biometric data increasingly used for security, protecting this sensitive information is cardinal. Funshade provides a method for comparing biometric data between parties without revealing the actual data itself, ensuring privacy and security.

As a particular example, the Digidow project uses biometric authentication to determine if the access can be given to the individual detected by a sensor. Digidow employs a decentralized structure in which each person's biometric template is stored either with a chosen cloud provider or on a personal server. Since biometric data is stored across potentially untrusted locations, the comparison of the stored template with the live data from a sensor requires a secure and privacy-preserving solution that protects the data even in the presence of potentially malicious parties. For this task Funshade is viewed as a potential candidate, as the participants are able to keep the sensitive data private that is needed for the verification.

A prototype is implemented in Rust, chosen for its strong memory-safety and performance features. Throughout the thesis, challenges such as managing Rust's memory model, and optimizing cryptographic functions were addressed. Additionally, several areas for future improvement are identified. These enhancements aim to improve security, usability, and adaptability of the framework in diverse applications.

Contents

List of Figures

Listings

Chapter 1

Introduction

Let's assume that in a building, there is a very important room (e.g. server room) into which only a few selected people can enter. It is not the only room on the same floor, there are a lot of other ones as well, such as regular offices. Despite many people walking around this room, there is no security guard protecting the secrets inside; only a system is in their place. This system is responsible for allowing only the members of the selected group to enter. It operates with a camera, a lock, and an external third-party system that handles the group's biometric data. When the camera detects someone in the corridor, it uses biometric authentication to verify their identity. Based on the verification, it signals to the lock to either open or remain closed.

Biometric authentication is a method used to verify a person's identity based on their physical or behavioural characteristics, which can include fingerprints, facial features, iris pattern, voice, or any other unique aspects of a person [44]. These traits are turned into data known as biometric data or embeddings, which are like digital versions of the characteristics, generated using mathematical formulas and stored as a collection of numerical values.

Biometric authentication relies heavily on this kind of data for verification to check if someone is who they claim to be. Typically, the process consists of comparing two sets of these embeddings, the stored one and the individual's who seeks access to something at the time. This comparison is performed by algorithms that check how closely the two sets of numbers match, and if they are similar enough, access will be granted. The same way as a security guard would.

In theory, this method ensures a high level of accuracy and security, as the chances of two people having the same biometric representation are very unlikely. This whole thing seems nice and simple at first, it is just a trivial comparison. But, where is your data stored? To grant access, a comparison needs to happen between two embeddings, namely the detected and stored. Where should this happen?

Let's analyze the above-mentioned scenario in more detail. The camera is placed on the corridor in a way that anyone's face who gets close to the door is visible. There are no obstacles that can prevent it to read their face. Whenever a person comes into the sight of the camera, it calculates their biometric data. This data needs to be compared with another that is accessible through an external third-party system, not operated by the party to whom the camera belongs to. Since the camera, the lock and the external system are not physically connected they need to communicate over a network in order to exchange data during the authentication process. Neither of them can do the same things as the other: the camera can not open the door, the lock can; the external system can not detect a person, the camera can; they need to collaboratively perform their part in order to fulfill their bigger purpose. When systems have this type of architecture or relationship between each other they are called distributed systems.

Now that everything is set, for the first scenario assume that the comparison happens on the camera side (cf. Figure 1.1). In this system it means that when the camera detects somebody and calculates the data, it will receive the other set of embeddings from the external system through a secure encrypted channel. This data needs to be decrypted in order to handle it, but this also means that the data becomes public to this party. In other words, the camera will get to know every persons' biometric data and this can cause a security issue. If the camera is untrustworthy or should become compromised, then a malicious party could get their hands on the sensitive information of these people.

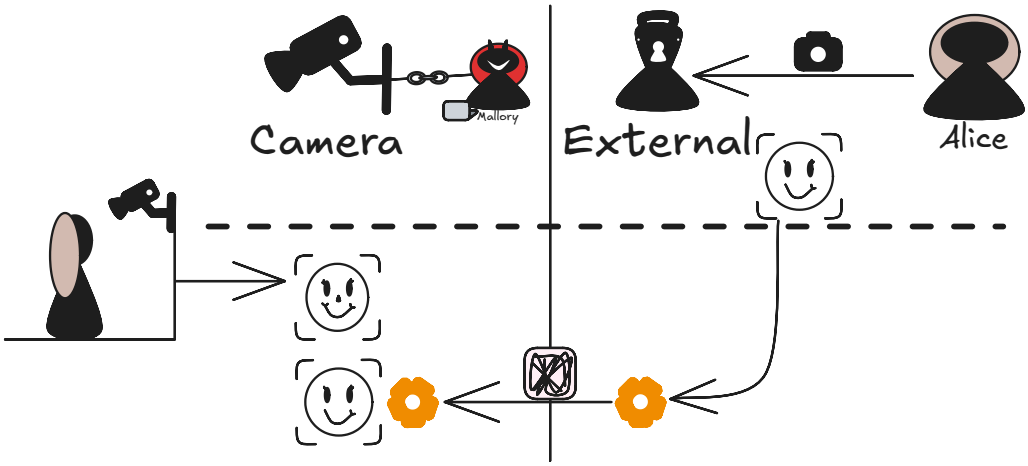


Figure 1.1: Stored data sent to the compromised camera.

In the second case, use the external system for the calculation (cf. Figure 1.2). The problem is similar to the previous one. The external system acquires the embedding of every person who wants to enter the room and also those one who are just passing-by. Should it receive or potentially store the data which it has nothing to do with? If the company whom the external system belongs to want it, they could monitor every person in that floor.

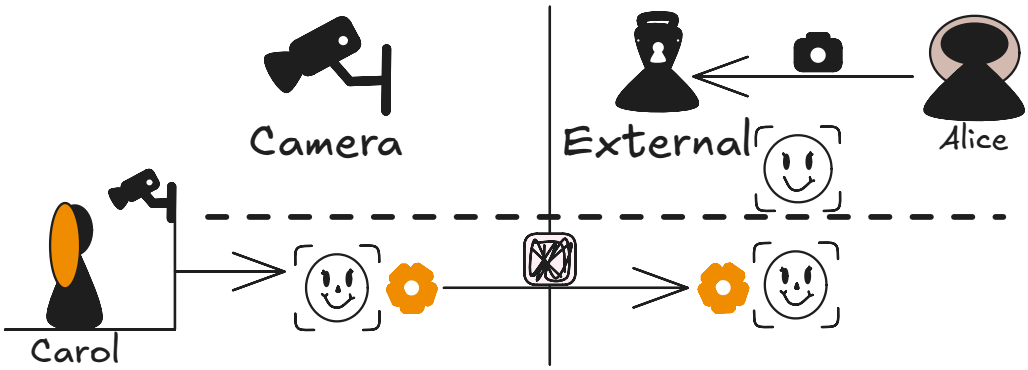


Figure 1.2: An office worker's data sent to the external system.

For this problem, secure multi-party computation (SMPC) offers promising solutions. It is an important and emerging area within cryptography that addresses the difficulty of performing collaborative calculations, while still ensuring the privacy of each party's inputs. This is especially valuable when sensitive data needs to be processed without exposing it to all of the involved par-

ties. The goal is to enable this kind of computation without revealing the input — the actual data — to one another.

Within this area, many solutions have already been developed [28] that deal with biometric matching based on various cryptographic techniques that allow computation over sensitive data. While these methods ensure privacy and security, in most cases it comes with some trade-offs. Some of them are communication-intensive, meaning they require a large volume of data to be exchanged, or they call for high number of communication rounds between the parties, increasing the complexity and latency of the interactions. Others are computation-insensitive and rely on such algorithms that demand significant processing power or time to perform, making them less convenient in applications where efficiency is an important aspect.

This thesis focuses on Funshade [29], a cryptographic solution that aims to strike a balance between security and efficiency. Funshade is designed to be computation-friendly and highly optimized for the evaluation phase of the process, where it achieves only a single round of communication between the two involved parties. This feature makes Funshade appealing for practical use, as it reduces the communication overhead while still maintaining the necessary level of security and privacy.

This thesis will explore the implementation of Funshade using Rust, a programming language well-know for its emphasis on safety, concurrency, and performance. By the application of Funshade and leveraging Rust's features — such as its strict memory management and type safety — this work aims to enhance the security of Digidow [41]. It will not only detail the mechanics and underlying principles of Funshade but also inspects key design and development choices, taking different use case scenarios into account.

By providing a secure and efficient implementation of Funshade, this thesis aims to contribute to the broader field of cryptographic research and offer a practical solution for privacy-preserving computations.

The structure of the thesis is as follows. Chapter 2 provides an introduction to the topic and the necessary background information for understanding the protocol and implementation. Chapter 3 offers a review of the other solutions focusing on secure biometric matching. Chapter 4 introduces the theoretical background of Funshade, explaining the cryptographic principles it relies on. In Chapter 5, the Rust programming language is explored, including an overview of its features relevant to the project. Chapter 6 delves into the actual implementation of Funshade using Rust, including the reasons behind the design choices made. Chapter 7 evaluates Funshade's performance through positive and negative test runs with concrete values, run-time benchmarking, and summaries of implemented unit tests. Finally, Chapter 8 concludes the thesis by summarizing the key findings, describes future improvements and suggests potential directions for future research.

1.1 Digidow

In recent years, the use of biometric data for various purposes has become increasingly widespread. In some countries, governments have implemented systems that monitor and track individuals using facial recognition and other technologies, while in others, people can pay for their bus ticket using biometric authentication.

These methods of identification provide the possibility to use them in public transport, payment and ticketing applications, or to cross country borders

without any physical identification or valet [41]. Aside from that, they are all using biometric authentication, there is a significant similarity between the previous examples, which is that all of them use a centralized database that resides in one hand (cf. Figure 1.3). Therefore, sensitive data controlled by a single entity carries the possibility of governmental overreach, potential misuse, or in the case of hacking, the attacker can gain all of them. In such scenarios, individuals have no control over how their data is used, collected, and stored, leading to a loss of privacy.

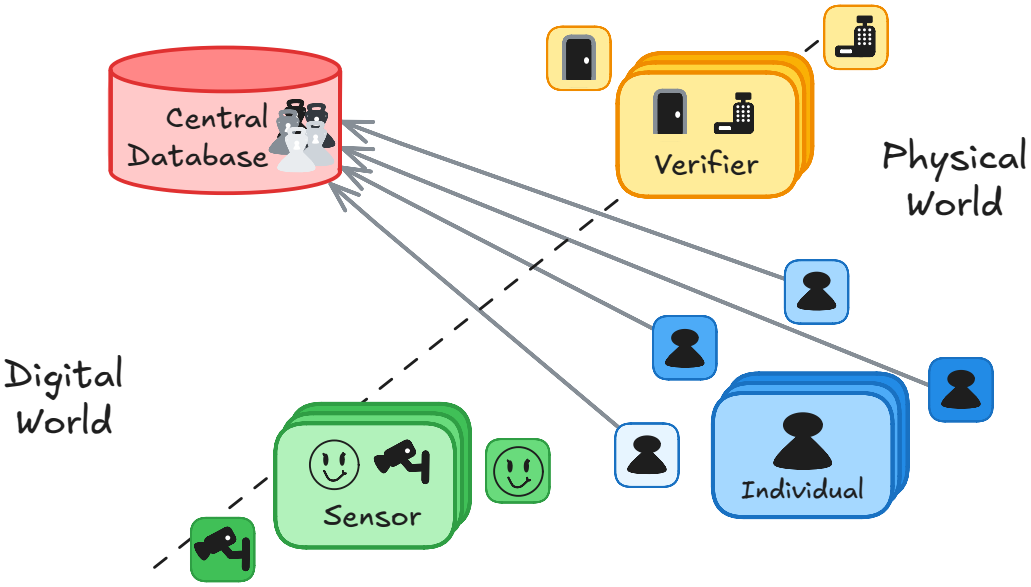


Figure 1.3: Individuals' data resides in a single, central database. Image is based on [42].

In this context, Digidow addresses the risk posed by centralized biometric storage. It offers a decentralized model for digital identity management, rather than relying on single institutions to store and manage the data. This is achieved by associating each individual with so-called Personal Identity Agents (PIA), a "*digital shadow of the person*" that makes it possible for them to interact with different services. By allowing the users to control their own agent, letting them decide where to install or host their agent, monitor and turning it off, it prevents single entities from being in direct charge of every person's data.

In addition to the PIA, there are two other participants in the process (cf. Figure 1.4): the Sensor and the Verifier. The Verifier checks the information provided by the PIA and ensures that a trusted sensor was used, but it can be also a physical component, such as a lock that opens the door. It also plays a crucial role in interacting with the physical world and executing actions based on the verification results. The Sensor is responsible for detecting characteristics of a person and translating it to a biometric template.

Unlike traditional systems that depend on centralized databases or physical ID documents, Digidow ensures that sensitive information remains under user control. This approach not only enhances privacy but also reduces the risk of data breaches and misuse by distributing trust across multiple parties rather than a single entity.

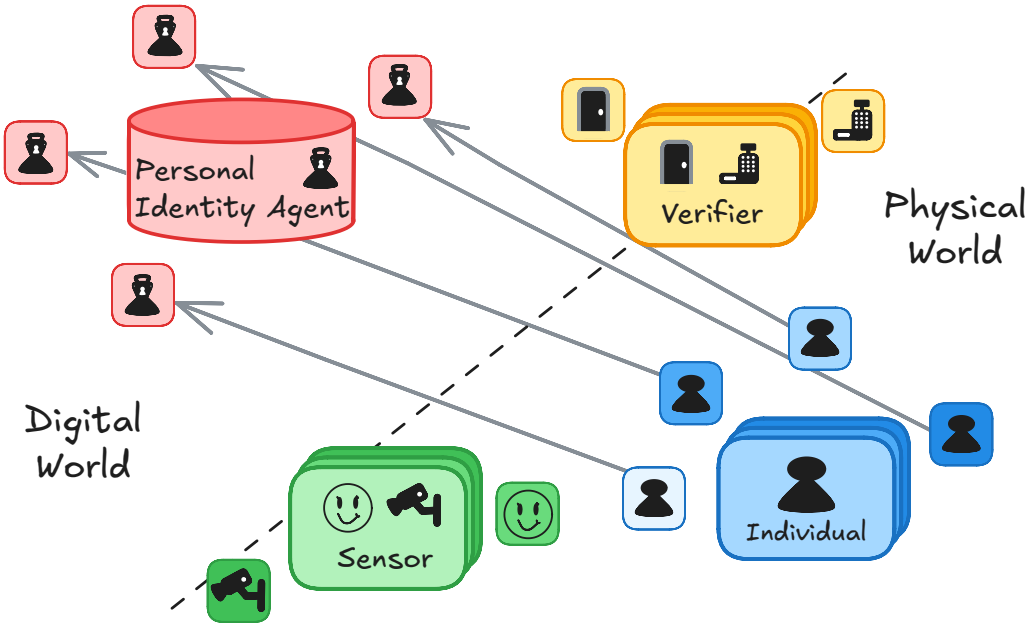


Figure 1.4: Individuals' data resides in their own Personal Identity Agent. Image is based on [42].

Chapter 2

Background

2.1 Biometric Authentication

Even though this work's main focus is not the biometric data itself or evaluating different face recognition and embedding calculation techniques, it needs to be mentioned for a deeper understanding of the topic.

Biometrics are measurable traits, that, due to their unique nature, can uniquely identify a person. These traits can be separated into two categories [44]: biological and behavioural. Biological traits include DNA, dental patterns, vein patterns, and perhaps the most well-known, fingerprints. The behavioural category involves aspects related to how individuals conduct themselves, such as speaking style, handwriting, and gait, the pattern of how an individual walks.

Using different mathematical formulas, neural networks, and techniques, the process of digitalizing these physical traits creates embeddings (cf. Figure 2.1). In most cases, this data represented as numerical vectors that can be used for calculations, allowing software and other systems to perform operations on them.

Traditional heuristic methods in biometric recognition relied on manually selected features, such as measuring the distance between the eyes or nose width [21]. However, modern neural networks learn more complex and abstract patterns directly from the data. The embeddings generated by them, capture even smaller details and are less dependent on predefined features [26].

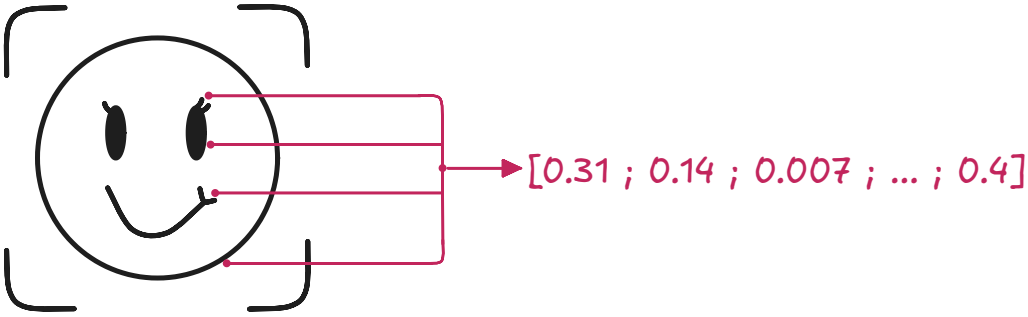


Figure 2.1: A simple visualization of embeddings.

During biometric authentication, two embeddings are needed to verify whether someone is who they claim to be. The process consists of a comparison of these vectors through computation that rates their similarity (cf. Figure 2.2). In simple terms, they assess how they relate to each other. This calculation is done by calculating distances (using distance metrics) or correlations [67]. The analysis allows the software or system to determine if there is a match or not, ensuring a reliable authentication process.

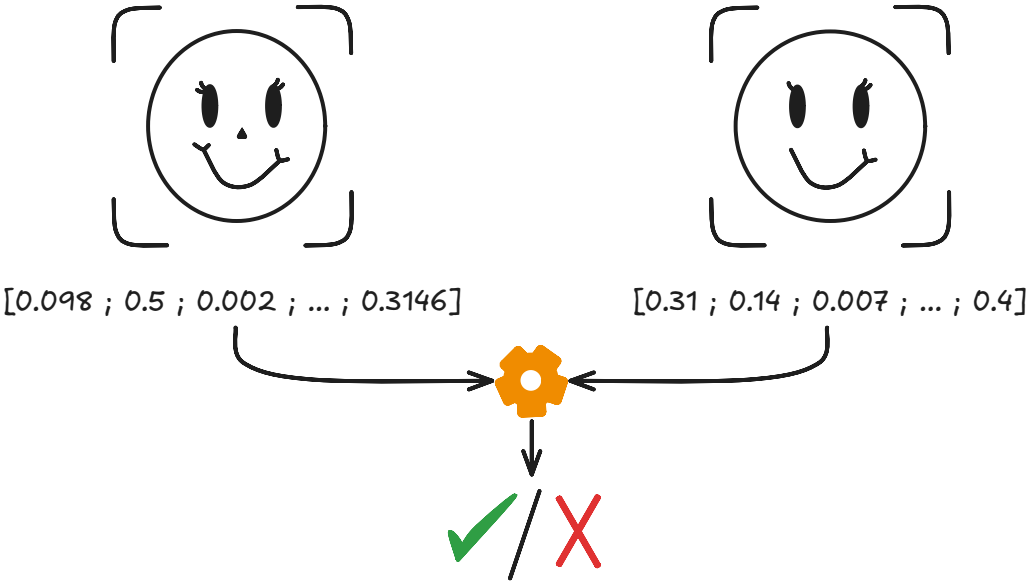


Figure 2.2: A simple visualization of biometric authentication.

2.2 Distance Metrics

Distance metrics, or also known as distance functions, are mathematical formulas used to quantify the dissimilarity between object points in a given space. In machine learning, data mining, and pattern recognition, distance measurement is a crucial part of different tasks, such as classification, clustering, and regression [60].

In most cases, a distance metric takes two data points — which in our case are vectors — and returns a single numerical value representing "how close" they are (cf. Figure 2.3). If this value is above a pre-defined threshold then the result will be a match [29], meaning the similarity between the two vectors considered high enough. Based on the dataset structure and the algorithm being used, the type of distance metric chosen will significantly affect the results [30].

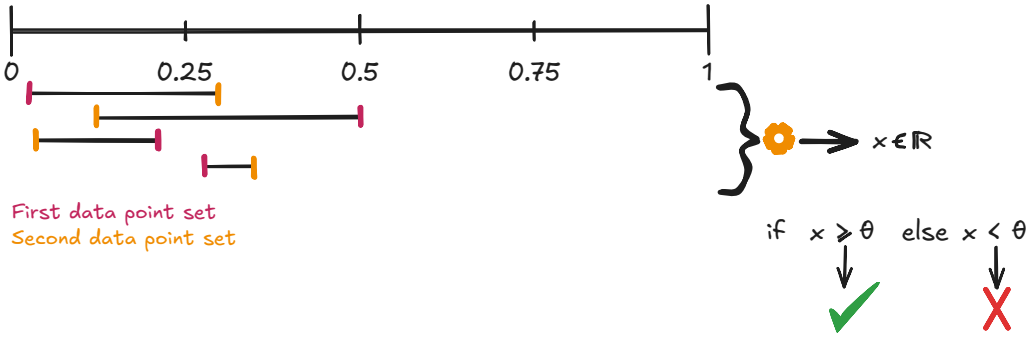


Figure 2.3: A simple visualization of distance metric.

Commonly used vector distance metrics are:

- **Euclidean Distance** [40]

  The Euclidean distance function measures the straight-line distance between two points (cf. Figure 2.4). The shortest possible path. It's not only usable in a 2D or 3D plane, but can be generalized to higher dimensions.

$$f_{ED}(x, y) = \sqrt{\sum_{j=1}^{n} (x_j - y_j)^2}$$

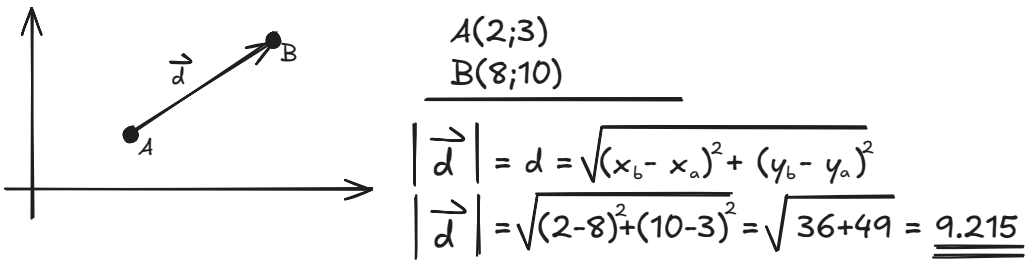


Figure 2.4: Euclidean Distance example.

- **Squared Euclidean Distance** [29]

  This is a slight optimization of the Euclidean Distance, achieved by dropping the calculation of the computationally intensive square root. It is used in many machine learning application, mostly in the context of clustering and face recognition.

$$f_{SED}(x, y) = \sum_{j=1}^{n} (x_j - y_j)^2$$

- **Hamming Distance** [7, 29]

  It was first used in information theory for error detection and error-correction codes to measure the error introduced by noise, but generally it is used to measure the distance between two bit-strings. The result is the minimum number of symbol changes needed to change from one to the other.

$$f_{HD} = \sum_{j=1}^{n} (x_j \oplus y_j)$$

2.3 Distributed Systems

As computer networks began to arise around 1960-70s, the concept of distributed systems started to take shape. Organizations grew, the computing needs increased, making it evident that these computational tasks needed to be divided and let different machines handle these pieces. This arising challenge paved the way for the development of Local-Area Networks (LANs) and Wide-Area Networks (WANs). In addition, other developments and events are

contributed to the evolution of distributed systems, such as the usage of personal computers became more general, the introduction of handheld devices and small computers with network connections, and the emergence of the Internet [66].

By definition "*a distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system*" [66]. In a distributed system the participating computing elements, called nodes, are/can be geographically separated. Although they operate independently, they work together to achieve a common goal. This network of computers is easily scalable, meaning that when a new device connects to this system, the overall architecture or the software does not need to be changed [32].

One challenging part of any definition about distributed systems is the phrase "*it appears as a single coherent system to the user*". This statement seems subjective, as it depends on the user's knowledge and perspective. More precisely, the system only hides the complexity of coordination from the user and manages the resources across the network without the direct involvement of the user [66].

Before looking at an example, it is important to mention another key aspect: fault-tolerance [32]. In a distributed system, many computers are connected to each other and any one could run into a problem. However, such problems should not halt the operation of the entire system. For instance, it would be a major issue if a student encountered an error while registering for courses and consequently the whole university system suddenly stops working. Therefore, it is the system's responsibility to handle these errors locally without impacting the other nodes.

As an example on how to deal with failures, we look at file sharing. In a torrent network, data is shared across multiple devices, with each device acting as both a client and a server. When somebody decides to download large files through this, like Linux distributions, they start a communication with multiple computers. The user retrieves chunks — small pieces of data — from these devices, ultimately each of them contributing to the final file. In this scenario, the user only needs to open a `.torrent` file in their preferred torrent client, while the process of acquiring and assembling the data happens automatically, without any further user interaction. Torrent exhibits the fault-tolerance characteristics of distributed systems: if one participant drops out from the process, the data will still be retrieved from someone else without an issue.

2.4 Cryptography

By definition "*Cryptography is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication*" [43].

Cryptography focus on securing communication in the presence of adversaries. It ensures that information can be transmitted in a way that allows the intended recipient to easily interpret it, while others face significant computational or time-based challenges in decoding the message. In short, cryptography provides the tools to guarantee the confidentiality and integrity of information [37].

The origins of cryptography can trace back to thousands of years to civilizations like Egypt, Greece, and Rome [59]. These initial methods often involved substituting letters to make messages unreadable and meaningless without the

right knowledge. One simple method is the "Caesar cipher", used by Julius Caesar, which shifts the alphabet by a fixed number, making it simple but effective at that time. Fast-forward to World War II, a famous example is the Germans' Enigma machine, which was broken by Alan Turing.

Over the centuries, cryptographic methods evolved, particularly driven by technological advances, these methods quickly shifted from "witty ideas" to complex mathematical algorithms [37]. One such mathematical algorithm is a public-key cryptographic technique, the RSA [53], named after its inventors, Rivest, Shamir and Adleman. RSA relies on the computational difficulty of factoring large prime numbers — around 100 digit — which makes the decryption extremely hard for unauthorized parties [43, 59].

For the further part of this thesis, we use a few key terms:

- **Hash Function**

  A hash function is a cryptographic primitive that transforms input data of arbitrary size into a fixed-size output, typically called a hash [37, 43]. It's used to verify data integrity since even a small change in the input creates completely different outputs. These functions are typically designed to work one-way, making it difficult to reverse original input from the hash. They are widely used in password storage, among other uses in integrity verification.

- **Offline and Online Phase**

  In secure multi-party computation protocols, computations are often split into two distinct phases [49]: the offline phase and the online phase. This separation allows computations to be made more efficient by preparing certain operations in advance.

  The offline phase is more like a preparation phase where no actual input data is processed yet, or it is not available at all. Its main purpose is to precompute certain values to speed up the subsequent online phase (e.g. generating Beaver triples in secret sharing, which is further discussed in Section 2.6.2).

  The online phase takes place after the offline phase and begins when the actual input data is available. The main point of the separation is to minimize the delay during the actual computation on the information. In the context of the example mentioned in Chapter 1, this is akin to standing at a door, waiting to be authenticated for entry. The quicker the process, the sooner access is granted.

- **Semi-Honest Party**

  There are different adversary models for describing how different so-called parties (e.g. nodes in a distributed system) behave. In secure multi-party computation, the term semi-honest describes parties that follow the protocol faithfully but attempt to extract as much information as possible from the computation [29]. These parties, also known as Honest-but-Curious, are not malicious in the sense of threatening integrity but still pose significant risks.

2.4.1 Password Authenticated Key Exchange (PAKE)

Password-authenticated key exchange (PAKE) is a major area of research in cryptographic protocols [5, 8, 25]. While user-chosen passwords are widely used, the users often pick weak ones, especially compared to cryptographic keys, making them vulnerable to theft, guessing, and brute-force attacks. Although measures can be implemented to limit online trial-and-error attempts, offline attacks remain a considerable threat. For instance, if an attacker gains access to the server, they can perform Rainbow Table attacks to crack the hashes.

PAKE protocols facilitate an interactive process between two parties (or a group) to establish cryptographic keys based on a shared password or its hash, while ensuring protection against malicious actors, even if messages are intercepted. The original work in this area, Encrypted Key Exchange (EKE) [5], was the first protocol that did not require each party's public key or digital certificates for authentication. Due to its simplicity, EKE paved the way for subsequent PAKE proposals over the following decades, many of which built on the original concept and some adapted it for client-server scenarios. EKE's consideration of users relying on short, easily memorable passwords also makes it advantageous in contexts where an additional cryptographic device for storing secret keys is unavailable [24].

Commonly, PAKE protocols can be categorized into two types: the balanced and the augmented schemes [24]. Balanced schemes assume the two communicating parties possess the same password information. This type is applicable for both client-client and client-server scenarios. However, in client-server situations, if the password from the server is stolen, it can be used to impersonate the client. To mitigate the risk, when the server is compromised, augmented schemes are tailored for client-server interactions. In these protocols, even if the attacker obtains verification files from the server, they cannot know the password without cracking it [8, 25].

After the participants agreed on a password, the process of Encrypted Key Exchange is as follows (cf. Figure 2.5) [5]:

1. One participant, A, generates a random number a and encrypts it using a symmetric cipher with the common password as a key, then sends it to the other party, B.
2. B decrypts the message using the same password.
3. After decryption, B generates a random value r, which serves as the common session key. They encrypt the value similarly to how A did, but first they use the value a and then the common password as keys, before sending it back to A.
4. A receives the message, then decrypts it. Now both of them knows the session key r.

If an attacker has control over the communication channel, they may resend old or stale messages. To prevent this, the protocol must implement safeguards, typically in the form of random challenges.

Despite its weaknesses, Encrypted Key Exchange, as the first PAKE protocol, demonstrated that the problem is solvable [24]. One of these protocols — and a more interesting one to mention in this thesis — is the Password Authenticated Key Exchange by Juggling, J-PAKE [25], where the two parties learn only 1-bit of information indicating whether the supplied passwords match (1) or not (0).

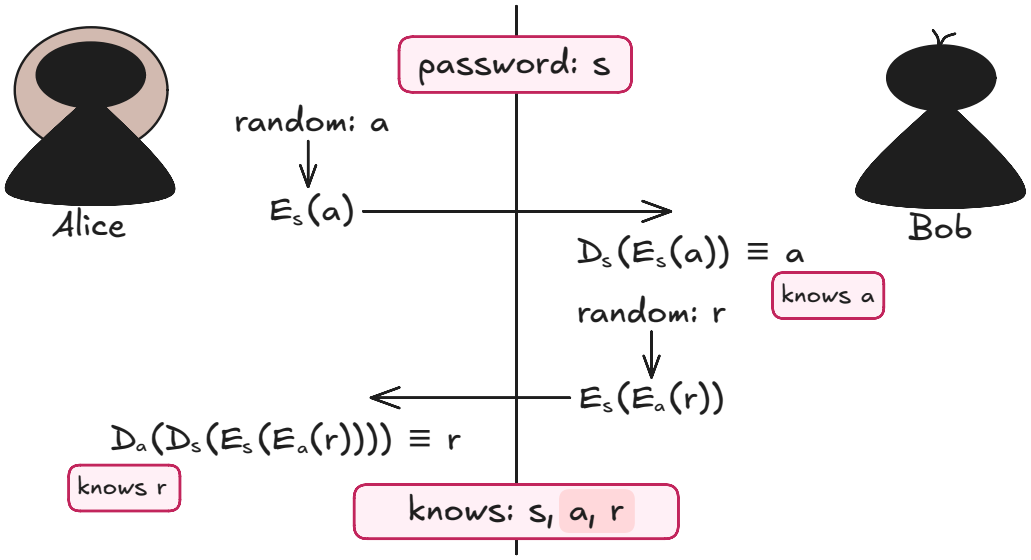


Figure 2.5: Encrypted Key Exchange.

In J-PAKE, participants Alice and Bob agree on a subgroup G of $\mathbb{Z}_q$ with order q, where q is the power of a prime number p, $\mathbb{Z}_q$ contains q number of elements and G is a subgroup of this field (e.g. 32-bit integer field: $q = p^m = 2^8 = 32$) [47]. They both know the (potentially weak) password $s \in [1, q-1]$ and a generator g that is able to generate a subgroup of G (cf. Figure 2.6).

Each party generates two random values: one from $\mathbb{Z}_q$,which may include 0, x_1 and x_3, and one from $\mathbb{Z}_q^*$, x_2 and x_4, where neither can be zero. Along with a knowledge proof (KP) [23] for the random values, they send g^x to the other. Using the knowledge proof, the parties verify that the other possesses the random values without learning the exact values. Following this verification, each party then compute:

$$\mathcal{A} = g^{(x_1+x_3+x_4)\cdot x_2\cdot s} \quad \text{and} \quad \mathcal{B} = g^{(x_1+x_2+x_3)\cdot x_4\cdot s}$$

For instance, in Alice side this can be expressed as:

$$\mathcal{A} = (g^{x_1} \cdot g^{x_3} \cdot g^{x_4})^{x_2\cdot s} = (g^{x_1+x_3+x_4})^{x_2\cdot s} = g^{(x_1+x_3+x_4)\cdot x_2\cdot s}$$

They exchange this data and calculate the common value, $\mathcal{K}$:

$$\mathcal{K} = \left(\frac{\mathcal{B}}{g^{x_2 x_4 s}}\right)^{x_2} = \frac{g^{(x_1+x_2+x_3)x_4 s}}{g^{x_2 x_4 s}} = g^{(x_1+x_2+x_3)x_4 s - x_2 x_4 s} = g^{(x_1+x_3)x_2\cdot x_4\cdot s}$$

A similar computation occurs on the side of Bob. With the same materials, they can derive a key using a hash function. The simplest way to verify that they share the same key is to encrypt a known value and exchange it, or use a random challenge [25].

This description might suggest that J-PAKE is suitable for biometric authentication. However, biometric templates are inherently variable, capturing slight differences each time due to environmental factors, changes in the individual (such as moisture on a finger), or differences in scanning devices. Consequently, biometric templates are never exactly the same, which makes algorithms that rely on exact matching unsuitable for biometric authentication. In

summary, the dynamic nature of biometric templates renders J-PAKE inappropriate for such purposes.

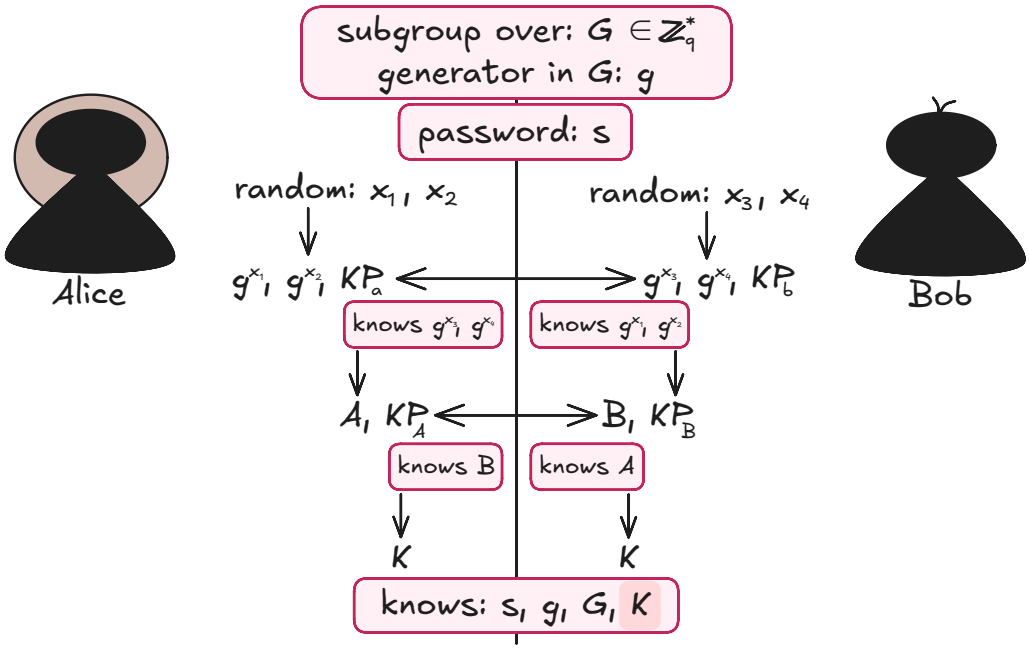


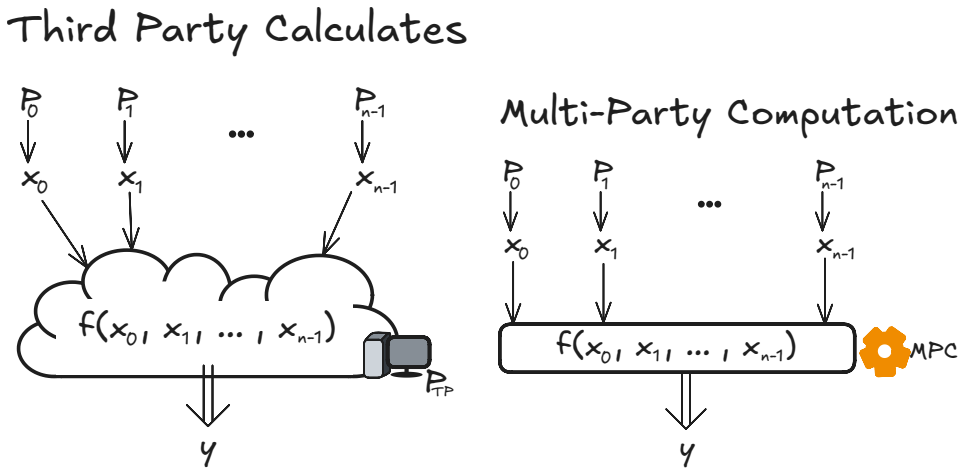
Figure 2.6: Password Authenticated Key Exchange by Juggling.

2.5 Multi-Party Computation

Secure Multi-Party Computation (MPC or SMPC) is a subfield of cryptography that enables multiple entities to collaboratively compute any mathematical function over their individual inputs, without revealing anything about them. This technique is especially valuable when sensitive information must be processed to compute a shared result, while no party discloses its data to the others. MPC finds applications in industries like finance, healthcare, government, and whenever confidentiality and data privacy are paramount [29].

The main idea behind MPC is to establish a protocol that allows the computation of a function, denoted as $f(x_0, x_1, \ldots x_{n-1}) = y$, using n inputs from different participants. These participants, labelled $P_0, P_1, \ldots P_{n-1}$, each holds a piece of information x_i, which they want to keep secret [15]. However, their inputs are necessary for the computation. Most of the time this is not easy to do due to legal and ethical constraints. For instance, when companies or official institutions are involved, they are bound by regulations that prohibit them from sharing sensitive data with an external party (P_{TP}) without explicit consent. From the point of reception the data will be in other hands, that can result in losing control over how it is used or further distributed (cf. Figure 2.7(a)).

MPC in this sense is different, in the end, it enables computation in a manner that prevents any party from learning (at most) more than the output (cf. Figure 2.7(b)). In other words, while the collaborative computation occurs, no participants gains access to any additional information about the others' input beyond the final result [15]. This ensures the protection of the data from any involved parties seeking to act maliciously or not.

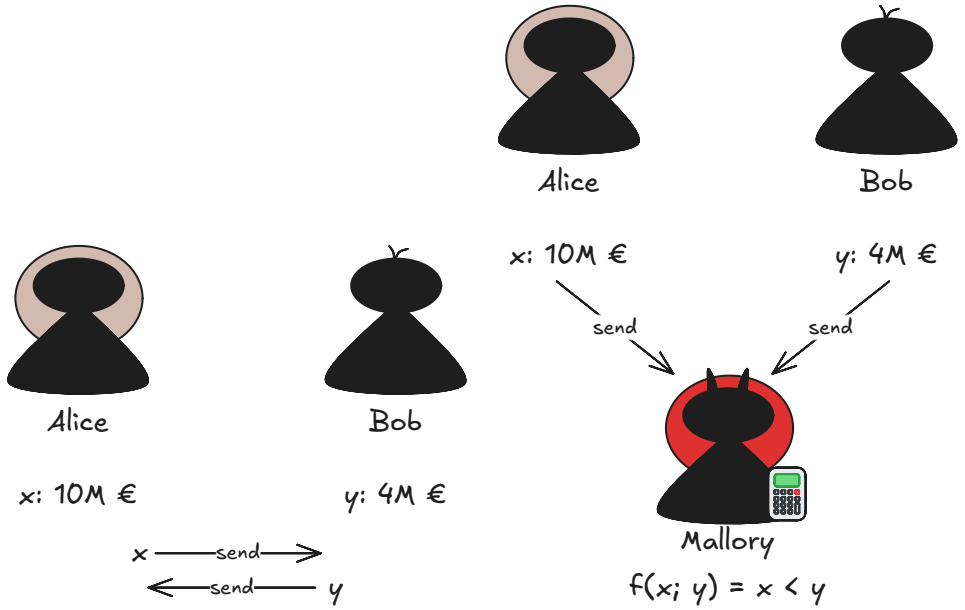


(a) A third party receives the data and calculates the result.

(b) The participants calculate the result using MPC.

Figure 2.7: Introduction to MPC.

Yao's Millionaire Problem [68] is a frequently mentioned example of what multi-party computation can solve. Let's assume that Alice and Bob are both millionaires, and they wish to compare their wealth without revealing their exact amounts to one another. At the time they meet, Alice has $x = 10\text{M}€$ and Bob has $y = 4\text{M}€$. They want to know who is wealthier, but they are afraid in the case they are significantly "poorer", they will be laughed at. The problem can be modelled by the function $f(x, y) = x \leqslant y$ which determines whether Alice has more or equal wealth compared to Bob.



(a) Alice and Bob send the value to each other.

(b) Alice and Bob send their values to Mallory.

Figure 2.8: Bad scenarios for Alice and Bob.

For a start, they can send their values to each other, but then both of the secrets will be publicly known to them, and they would rather keep it as a secret (cf. Figure 2.8(a)). Another solution could be to involve a third party, Mallory, who could compute the result for them. Alice and Bob would be happy,

they would know the answer from Mallory that Alice is richer, but neither of them would know by how much. However, this introduces a new issue: Mallory now possesses both secrets, which she could potentially misuse for her benefit (cf. Figure 2.8(b)).

Alice and Bob drop these solutions as they neither want to let the other know their secret nor want to let another third person take advantage of it. Fortunately, Alice learnt cryptography in school and suggests an alternative: Multi-Party Computation (cf. Figure 2.9). Alice only needs to choose a technique she likes, such as Homomorphic Encryption [22, 54] — that allows computations on encrypted data — or Secret Sharing that will be explained in the following sections.

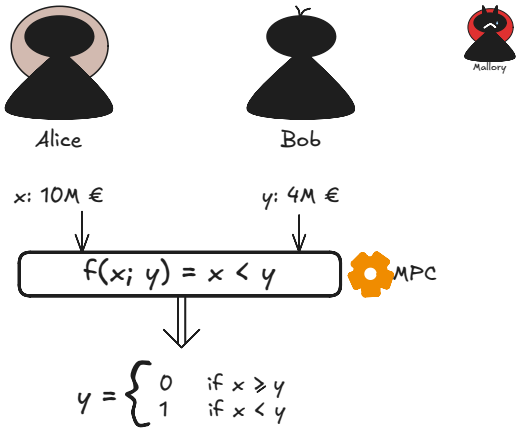


Figure 2.9: Alice and Bob's solution using MPC.

In a real world scenario, one example where Multi-Party Computation can be applied is when multiple companies want to compare performance metrics to gain industry insights, but without revealing their own numbers to competitors [15]. With this technology, companies can securely compute benchmarks while keeping their individual data confidential.

2.6 Secret Sharing

Secret Sharing is a cryptographic method used to split (sensitive) information, referred to as secret s, into multiple n pieces or shares. These shares are distributed among a group of participants in such a way that only authorized parties, upon collaboration, can reconstruct the secret, but nobody from outside. Importantly, no participant can access the complete secret or even other individual shares on their own. The shares in isolation hold no meaningful information about the secret, ensuring that if one party is compromised, the attacker gains no knowledge of the actual value [34].

The shares are distributed in a way that a specific number of participants, denoted as $k \leqslant n$, must cooperate to reconstruct the secret [16]. This design ensures that while the secret remains protected, it can still be recovered if some shares are lost or compromised. In the special case where $k = n$, all participants are required to reveal the secrets (cf. Figure 2.10).

In a situation where information must be protected and not trusted to a single individual or entity, Secret Sharing methods are essential. Without collaboration among authorized parties, the secret remains inaccessible — with the ex-

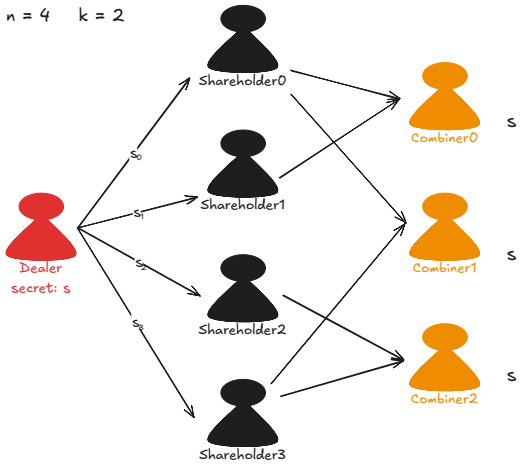


Figure 2.10: Secret Sharing visualization.

ception of the dealer who distributes the shares. These methods serve as building blocks for several cryptographic protocols, including Multi-Party Computation, Byzantine Agreement, Access Control, Attribute-Based Encryption, and Generalized Oblivious Transfer [34].

The following sections describes Additive Secret Sharing, which is a basic building-block for the more advanced schemes in question. Such schemes are Π-Secret Sharing and what it build on, the Beaver Triples.

2.6.1 Additive Secret Sharing

Additive Secret Sharing is one of the simplest forms of secret sharing, especially when the number of required participants equals the total number of parties $(k = n)$. In this scheme, the dealer selects $n - 1$ random numbers uniformly from a predefined field. These random numbers serve as the shares for the first $n - 1$ participants [36]:

$$s_i = r_i \quad i \in [0; n-1), r_i \sim \mathcal{U}$$

The share for the n-th participant is computed by subtracting the sum of the $n - 1$ random values from the original secret:

$$s_{n-1} = s - \sum_{0}^{n-2} s_i$$

The dealer, who initially holds the secret, distributes the shares to each participant. To reconstruct the secret, an authorized and trusted party can combine the values. In this additive method, the secret is recovered by summing the individual shares, thereby obtaining the original value (cf. Figure 2.11).

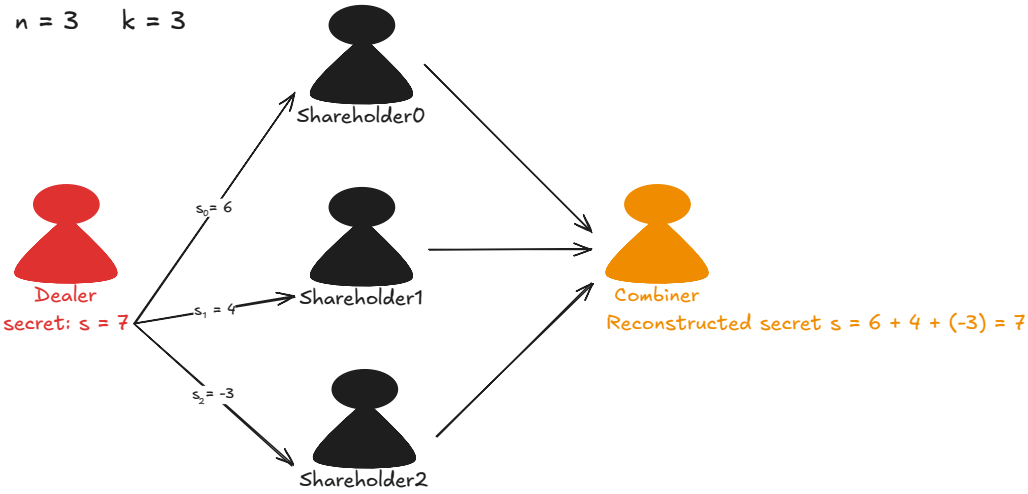


Figure 2.11: Additive Secret Sharing Example.

2.6.2 Beaver Triples

Beaver Triples, or Beaver Multiplication Triples [4], are a set of pre-computed values that enable secure multiplication over secret-shared inputs. The triples can be computed before the actual input is known, avoiding the real-time randomization during the online phase. These triples act as helper values, preserving the privacy of individual inputs while the parties collaboratively compute a product, requiring only one round of communication.

Beaver Triples consists of two random values, a and b, which are used to mask the secret-shared inputs x and y, respectively. To reconstruct the true result of the multiplication, an additional value $c = ab$ is required [50]. In the offline phase, before the inputs are available, the parties receive additive shares of these three values (a, b, and c). The actual computation begins only when the input shares (x and y) are provided or requested.

The process starts with each party computing masked versions of their input shares. Specifically, each party computes:

$$d_j = x_j - a_j \quad \text{and} \quad e_j = y_j - b_j$$

Next, the parties exchange their values to separately calculate the masked inputs d and e. After this exchange, they can compute their respective shares of the multiplication result, z_j:

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

The final result is obtained by summing the result shares from all parties.

Why can this work? To understand why Beaver Triples work, consider adding the shares together while recalling the definitions $c = ab$, $d = x - a$ and $e = y - b$. The resulting expression simplifies as follows:

$$
\begin{aligned}
z &= d \cdot b_0 + e \cdot a_0 + c_0 + 0 \cdot d \cdot e + d \cdot b_1 + e \cdot a_1 + c_1 + 1 \cdot d \cdot e \\
&= b_0(x - a) + a_0(y - b) + c_0 + b_1(x - a) + a_1(y - b) + c_1 + (x - a)(y - b) \\
&= xb_0 - ab_0 + ya_0 - a_0b + c_0 + xb_1 - ab_1 + ya_1 - a_1b + c_1 + xy - xb - ya + ab \\
&= xb_0 + xb_1 - ab_0 - ab_1 + ya_0 + ya_1 - a_0b - a_1b + c_0 + c_1 + xy - xb - ya + ab \\
&= x(b_0 + b_1) - a(b_0 + b_1) + y(a_0 + a_1) - b(a_0 + a_1) + c_0 + c_1 + xy - xb - ya + ab \\
&= xb - ab + ya - ab + c + xy - xb - ya + ab \\
&= xb - ab + ya - ab + ab + xy - xb - ya + ab \\
&= xy
\end{aligned}
$$

Figure 2.12 summarizes this in an example. Suppose that Carol has two values, $x = 5$ and $y = 7$. She would like to calculate $z = xy$, but she lacks the necessary computation power to do it. She seeks help from Alice and Bob, but wants to keep her values secret from them. Using Beaver Triples, Carol sends input shares and the pre-computed triples to Alice and Bob. After both of them go through the computation process and send their results back to Carol, she can simply add their results together to obtain the correct answer.

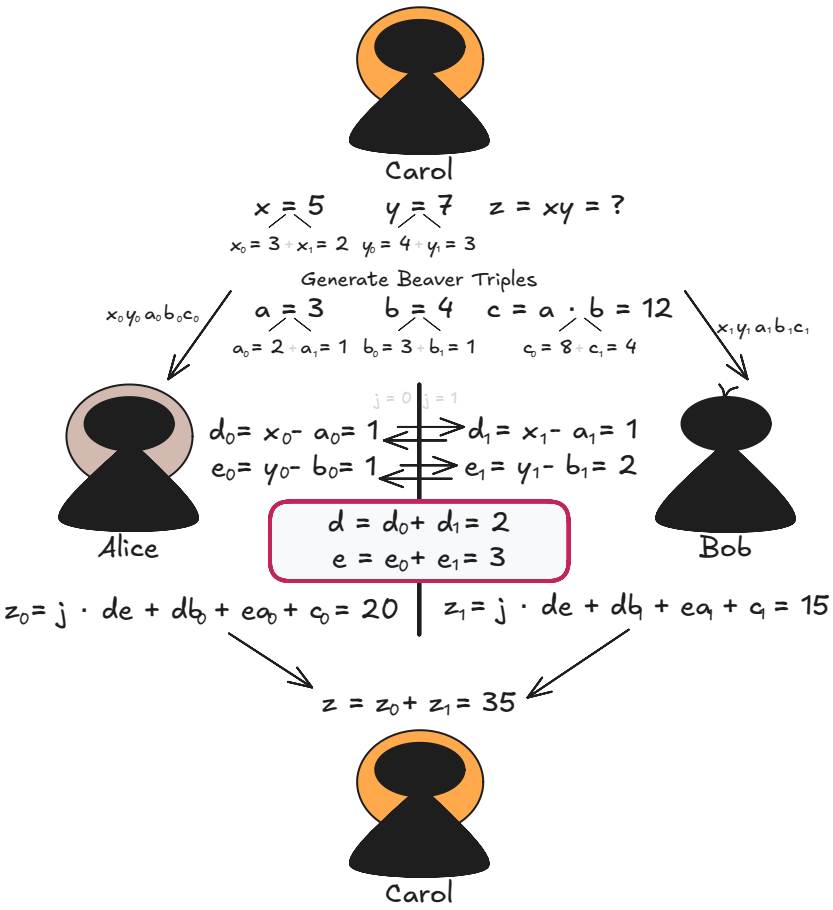


Figure 2.12: Beaver Triples Example.

2.6.3 Π-Secret Sharing

This secret sharing method was introduced in ABY2.0 [48], inspired by Beaver Triples to perform additive secret sharing in two-party computation (2PC). The Funshade team, in their paper, referred to it as Π-Secret Sharing to simplify their explanation (cf. Figure 2.13) and make it easier to reference [29].

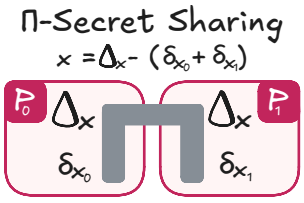


Figure 2.13: Π-Secret Sharing. Image is based on [29].

During the first phase, the parties prepare for the actual operations by generating additive shares of the input values in order to keep it a secret from each other (cf.Figure 2.14). Each party, P_i, generates a uniformly random value δ_{v_i}, and together, they sample the other party's share, $\delta_{v_{1-i}}$. These values behave as the additive share of the mask $\delta_v = \delta_{v_i} + \delta_{v_{1-i}}$ for each $v \in \{a, b\}, i \in \{0, 1\}$, assuming the two parties can securely compute it. The parties then calculate a masked version of the input:

$$\Delta v = v + \delta_v$$

The masked input is stored locally and shared with the other party [29].

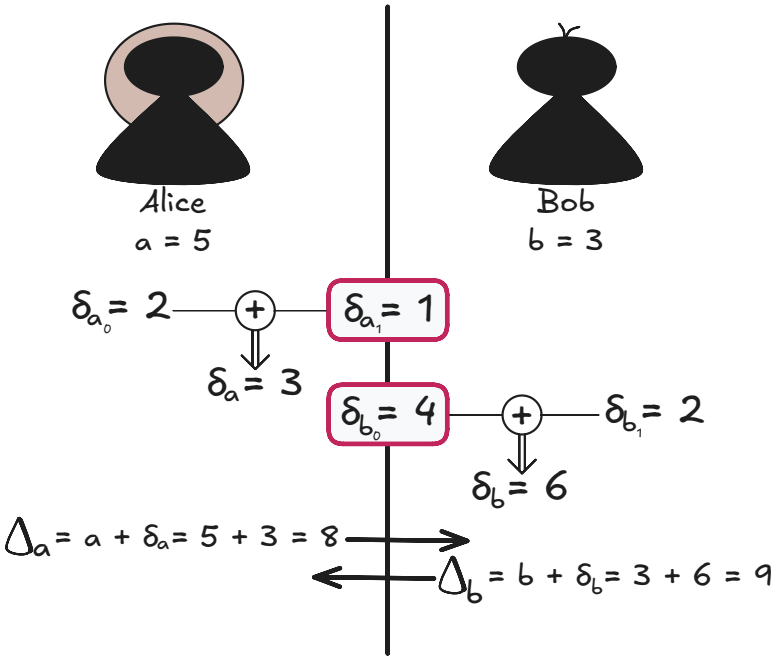


Figure 2.14: Π-Secret Sharing Initialization.

So how does this work when two numbers from two parties should be added? Let's assume Alice has a value $a = 5$ and Bob has the value $b = 3$. After they are finishing the first phase and calculating the shares, they exchange the masked

inputs Δa and Δb as visualized in Figure 2.14. Alice ends up with the shares δ_{a_0} and δ_{b_0}, while Bob has δ_{a_1} and δ_{b_1} [48, 65].

The original values can be reconstructed by subtracting the mask from the masked value: $v = \Delta v - \delta_v$. Therefore, to compute $a+b$, the following expression is evaluated:

$$a + b = (\Delta a - \delta_a) + (\Delta b - \delta_b)$$

The calculation of the result in this case is simple, both compute the sum of their local shares:

$$\delta_{z_i} = \delta_{a_i} + \delta_{b_i}$$

Finally, the result of the addition is reconstructed by subtracting the resulting shares from the sum of the masked inputs (cf. Figure 2.15):

$$z = a + b = \Delta z - \delta_z = \Delta a + \Delta b - \delta_z = \Delta a + \Delta b - \delta_{z_0} - \delta_{z_1}$$

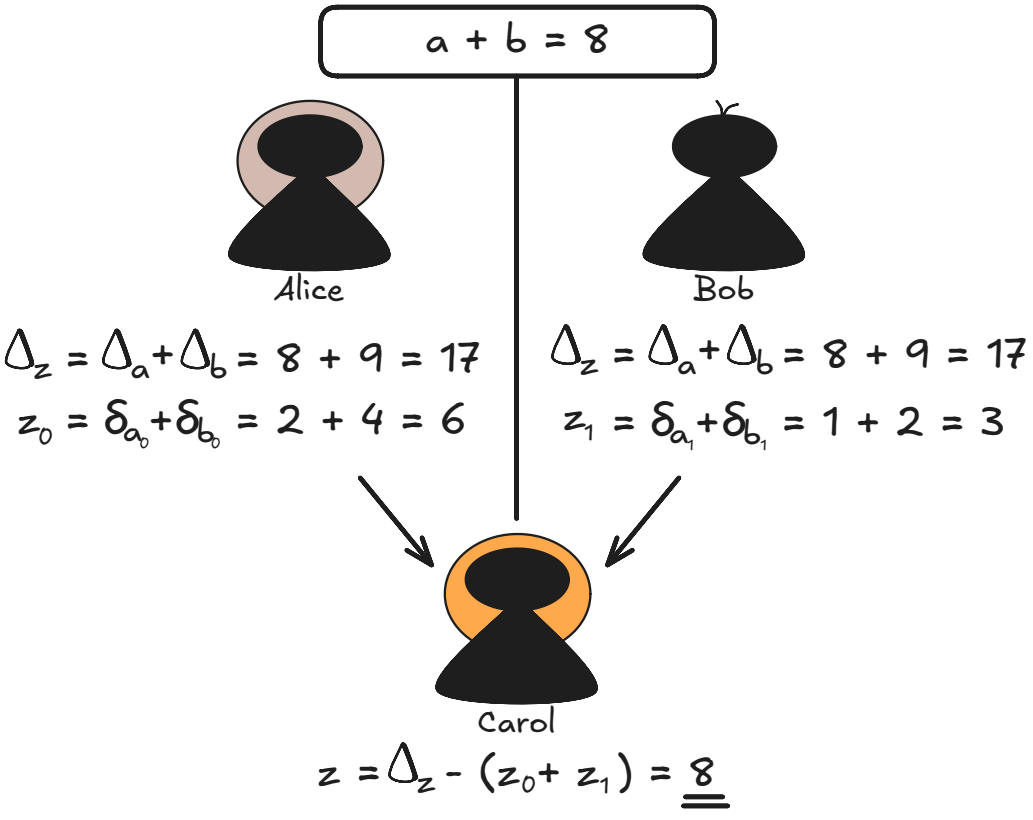


Figure 2.15: Π-Secret Sharing Addition.

When multiplying, the process follows a similar structure (cf. Figure 2.16). Suppose Alice and Bob want to multiply their values, a and b. The multiplication is computed as follows:

$$z = ab = (\Delta a - \delta_a)(\Delta b - \delta_b) = \Delta a \Delta b - \Delta a \delta_b - \Delta b \delta_a + \delta_a \delta_b$$

Since $\Delta v = v + \delta_v$, the input value can be substituted accordingly with $\Delta v - \delta_v$, but each party only knows their own δ_{v_i} additive share of δ_v, not the other part, $\delta_{v_{1-i}}$. After expanding the expression, the result, unlike its previous form, can be broken down further to more separable pieces:

$$\begin{aligned} z &= \Delta a \Delta b - \Delta a \delta_b - \Delta b \delta_a + \delta_a \delta_b \\ &= \Delta a \Delta b - \Delta a(\delta_{b_0} + \delta_{b_1}) - \Delta b(\delta_{a_0} + \delta_{a_1}) + \delta_a \delta_b \\ &= \Delta a \Delta b - \Delta a \delta_{b_0} - \Delta a \delta_{b_1} - \Delta b \delta_{a_0} - \Delta b \delta_{a_1} + \delta_a \delta_b \end{aligned}$$

Both of the parties know both Δ values, and they can locally compute the multiplication with the δ -values. The challenging part arises with $\delta_a \delta_b$ and within this only the two cross-party product of $\delta_{a_0} \delta_{b_1}$ and $\delta_{a_1} \delta_{b_0}$, since they both belong to different parties.

$$\delta_a \delta_b = (\delta_{a_0} + \delta_{a_1})(\delta_{b_0} + \delta_{b_1}) = \delta_{a_0} \delta_{b_0} + \delta_{a_0} \delta_{b_1} + \delta_{a_1} \delta_{b_0} + \delta_{a_1} \delta_{b_1}$$

To calculate the product of the masked inputs, the cross-terms must be handled securely. Assuming secure computation techniques — like Oblivious Transfer [51] or Homomorphic Encryption [22, 54] — are used to compute the additive shares of $\delta_a \delta_b$, the additive share of the final result, z_j, can be calculated by both parties and summed together [48]. It is important to note that based on the full expression, only one party needs to compute the multiplication of the masked input values $(\Delta a \Delta b)$, as performing this step on both sides would yield incorrect results due to $\Delta a \Delta b$ is present once in the equation.

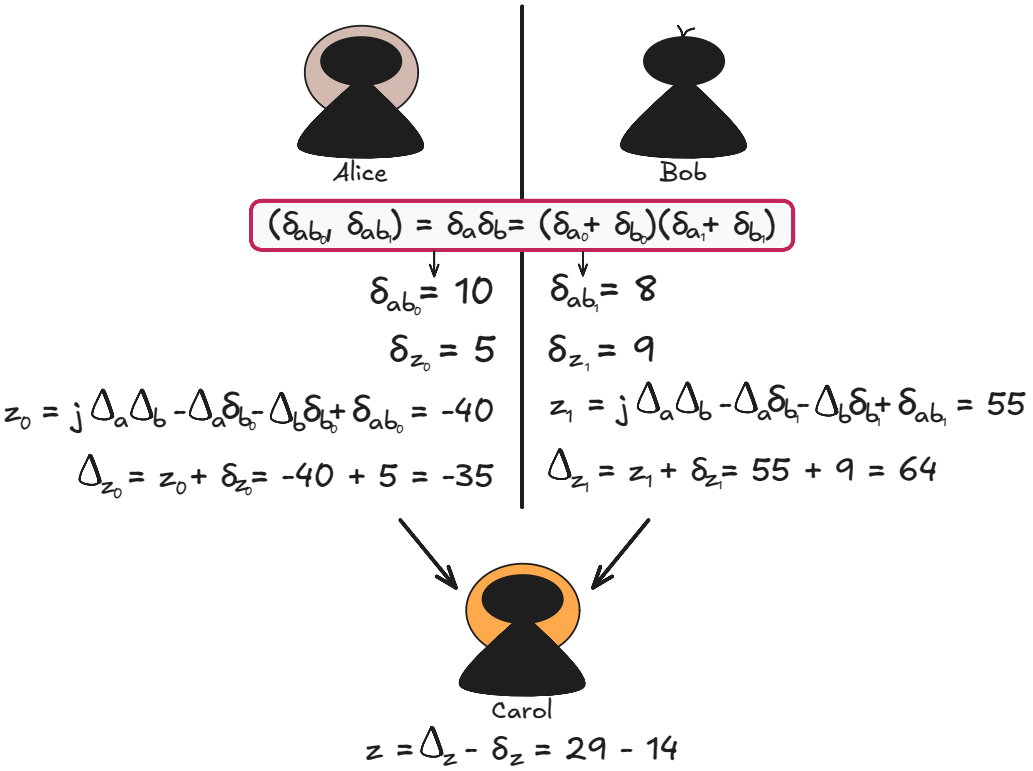


Figure 2.16: Π-Secret Sharing Multiplication.

2.7 Function Secret Sharing (FSS)

Function Secret Sharing [10] is designed to securely split a function into multiple components, or shares, in such a way that no individual share reveals any information about the original function. Only when all shares are combined can the original function be reconstructed. Similarly to other Secret Sharing methods, if enough participants are compromised, then an attacker could potentially recover the original formula. This method is useful in scenarios when multiple parties are needed to collaboratively perform a computation without any of them having access to the entire function (cf. Figure 2.17).

The basic idea behind FSS is to divide a function, $f(x)$, into two or multiple seemingly meaningless shares, each assigned to different participants [64]. These shares, when evaluated on the same input value x, generate outputs that, when combined, yield the result of the original function. The key distinction here is that it is not the input that is kept secret, but the function itself, which remains hidden from all parties except the Dealer. The final result, computed after combining the partial outputs, will match the output of the original function.

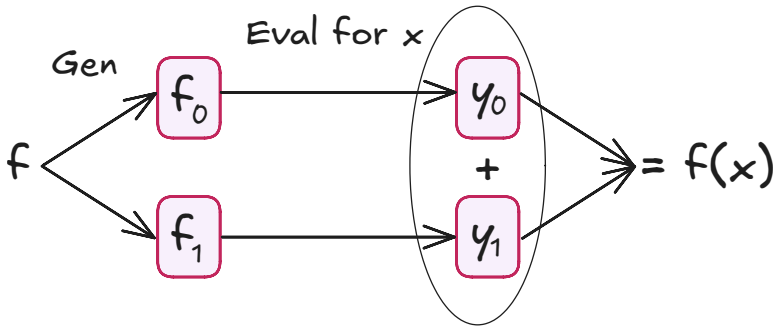


Figure 2.17: Function Secret Sharing. Image is based on [63].

We break down the method in a sense of two parties [9, 29]:

- **Key Generation:** $\mathrm{Gen}(1^\lambda, f) \rightarrow (k_0, k_1)$ is the key generation algorithm. It takes the given security parameter λ and the description of a function $f : \mathbb{G}_{in} \rightarrow \mathbb{G}_{out}$, producing a pair of function keys (k_0, k_1). The security parameter defines how long the bit-string generated by a pseudo-random generator (PRNG) will be, and the key length partially depends on this. The larger the value, the harder for the attacker to break the scheme. $\mathbb{G}_{in}$ and $\mathbb{G}_{out}$ represents the input and outputs domain of the function.
- **Evaluation:** $\mathrm{Eval}(j, k_j, x) \rightarrow z_j$ is the evaluation phase of the process. As inputs, it takes a party identifier $j \in \{0; 1\}$, the party's key k_j, and a public input value $x \in \mathbb{G}_{in}$. The output of the $f_j(x)$ function, $z_j \in \mathbb{G}_{out}$, is an additive share of the final result such that $f(x) = z = z_0 + z_1$.

By splitting the function into shares (for example a polynomial as visualized in Figure 2.18) and using these steps, each party only evaluates their part of the function and holds an additive share of the final result, without learning anything about the original function itself.

We use an example use case to describe the process. Assume that there is a database containing sensitive patient information — such as names, date of birth, phone number, medical record, etc. Alice is working for a company and for research purposes she would like to know how many (registered) persons have a specific illness.

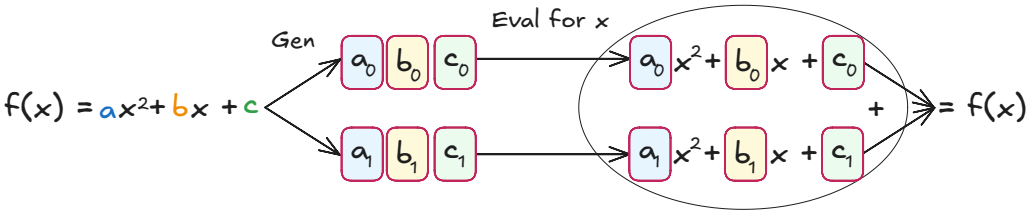


Figure 2.18: A simple example of FSS. Image is based on [63].

One simple solution would be to give Alice direct access to the database and let her calculate whatever she is curious about. However, this would expose sensitive patient data to a third party (Alice and her company), potentially leading to privacy breaches.

Alice could also perform the calculation through a server connected to the database. She gets her results and is happy that she could do her job, but the same is not true for the company. While this keeps the data away from her, it raises another problem: the server operator might learn about the company's research queries, which could lead to competitive disadvantages if leaked to rivals.

It would be advantageous to both participants to use Function Secret Sharing. In this approach, Alice generates function shares and distributes them to multiple non-colluding servers, all connected to the same database as shown in Figure 2.19.

These servers collaboratively handle the evaluation of the function on the patients' data without learning what Alice is querying. Once the evaluation is complete, Alice receives the additive shares of the result, and calculates the sums to get the final output — without any server or party knowing the full query or result [63].

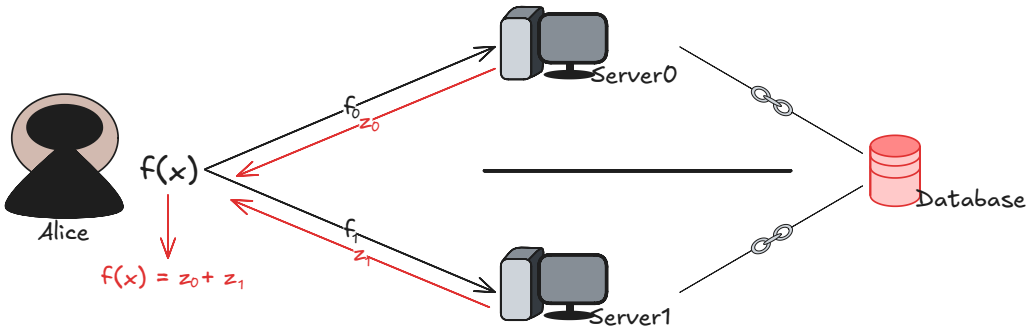


Figure 2.19: Practical example of FSS. The image is partially based on [63].

2.7.1 Distributed Point Function (DPF)

In the previous example, Alice used Function Secret Sharing (FSS) to count the patients with a specific criteria, she performed FSS over the class of point functions. Another important use case is querying a specific item from a database.

A point function is defined as follows:

$$f_\alpha : \{0;1\}^n \to \{0;1\} \quad f_\alpha(x) = \begin{cases} 1 & \text{if } \alpha = x \\ 0 & \text{else} \end{cases}$$

Alternatively, it can be defined to output a value β:

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

This fundamental construction is called the Distributed Point Function (DPF) and forms the basis for a more interesting concept discussed in this thesis, the Distributed Comparison Function (DCF) [64].

The core idea behind DPF is to create binary trees for each participant of the computation [10]. These trees are structured such that there's only one unique path that leads to a non-zero result, and this path corresponds to the bitwise representation of the input. Specifically, at each level of the tree, if the corresponding bit of the input $\alpha[i] = 1$, the path continues to the right; if $\alpha[i] = 0$, it goes to the left.

Consider the case where there are two parties, i.e., $m = 2$. Let z_j^k represent the values at the leaves along all possible paths, where $j \in \{0,1\}$ is the party identifier, and $k \in [0, 2^n)$ is the index of the leaf (cf. Figure 2.20). The correct result is only produced by summing the values at the correct leaf (indexed by $\mathtt{correct}$). The result is given by:

$$z = z_0^{\text{correct}} + z_1^{\text{correct}}$$

For every other path, the sum of the values yields zero. Because of this design, the computing parties only see random values, and only the party that receives the final result from all participants can derive the true outcome [10, 64].

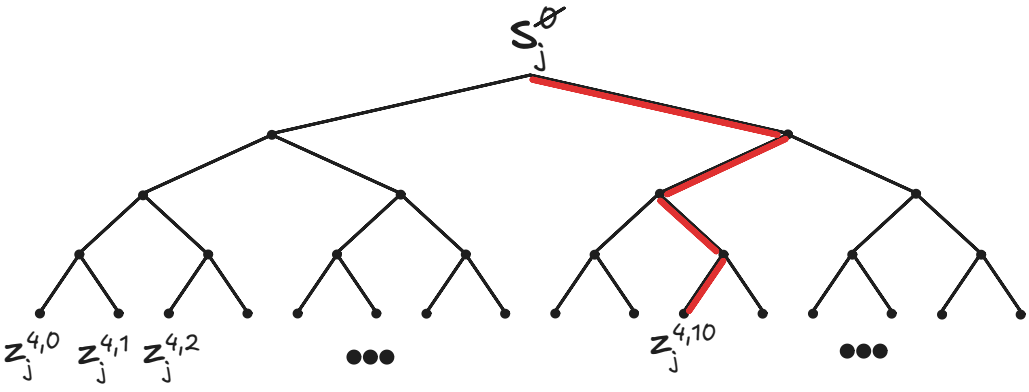


Figure 2.20: In this example, the correct path corresponds to the bit string $\alpha = 1010$, and only $z_0^{10} + z_1^{10}$ can reveal the correct result. Image is based on [64].

The process begins with a random seed, $s_j \in 1^\lambda$, which serves as the basis for generating the trees. At each step i, a pseudo-random generator uses a seed to generate two leaves: $\mathtt{Left}$ and $\mathtt{Right}$, each containing a new seed s_j^i and a boolean

value t_j^i. In the first iteration, the algorithm uses the initial seed, but subsequent steps use the newly generated ones, ensuring that randomness is maintained throughout the process [10, 64].

At the end of each step, so called correction words (CW_i) are created based on the random values and the n long bit-string input $\alpha \in \{0;1\}^n$. The words guide the correct $x \in \{0;1\}^n$ input's path towards the appropriate leaf during the evaluation phase. Correction words adjust the randomness in the process, allowing the correct path to be traversed [10, 64].

However, the evaluation algorithm uses these correction words only conditionally, strongly depending on the x known to both parties — in a similar way to α — and the generated random boolean values. This ensures that neither of the parties can independently figure out the original value of α [64]. They only know a kind of guideline, but not the exact path.

Imagine it as a game where a person is navigating out from a maze and at each crossroad, they calculate their direction based on a random value. If their calculation results in some good values, but their hunch leads them off the correct path, the guide (i.e., the correction word) steers them to the correct way, however, if their intuition says otherwise, the guide won't tell a thing.

Correction words are also essential for maintaining consistency when the input value is incorrect (cf. Figure 2.21). If the next bit of the input does not match the expected bit, the algorithm ensures that the values on both sides of the tree remain equal by calculating the same tuples. Since the same seed is used for the remaining part of the computation, the values cancel out due to the properties of exclusive or (XOR), resulting in a zero-sum [10].

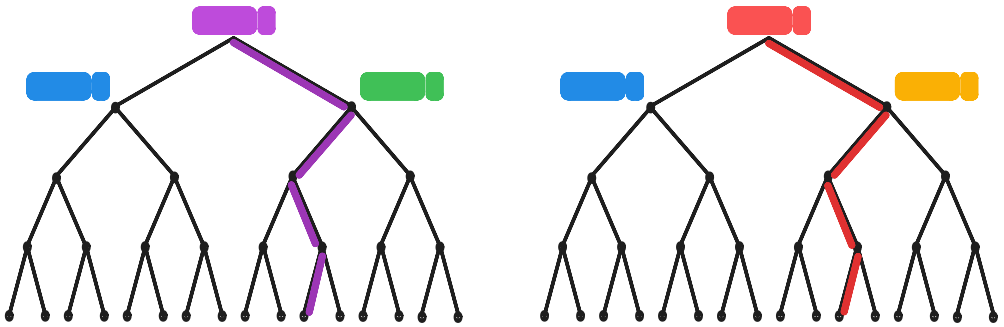


Figure 2.21: Correction words adjust DPF evaluation trees in off- and on-path. Image is based on [64].

In conclusion, correction words are the cornerstone of Distributed Point Functions. They guide the algorithm toward the correct leaf based on the input while ensuring that any incorrect paths cancel out, preventing erroneous outputs (cf. Figure 2.23). DPFs are crucial in enabling secure, privacy-preserving queries, where only the correct result is revealed without leaking additional information.

2.7.2 Distributed Comparison Function (DCF)

As mentioned in the previous chapter, the Distributed Comparison Function (DCF) [9] builds upon the fundamental principles of DPF. DCF enables the comparison of a value x against a threshold α:

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

or alternatively,

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

The idea remains the same, the generation algorithm creates a "binary" tree structure that serves as a guide for the participants. Unlike DPF, however, more leaves in the tree produce correct results. Specifically, all leaves to the left of the leaf representing the input value α correspond to valid paths, returning a non-zero result (cf.Figure 2.22) [64].

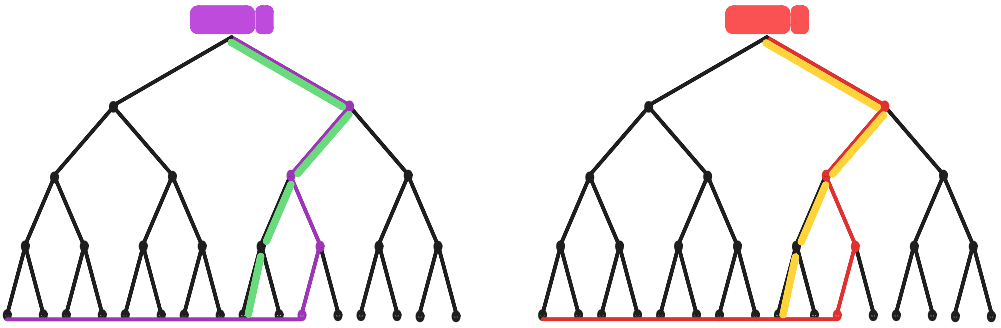


Figure 2.22: DCF Tree. Image is based on [64].

To return an arbitrary β, the correction words are extended with a new n bit long value v. This value is constructed such that if the path goes to the left, the correct term is returned. However, if the path follows the correct input or goes to the right, the final answer remains zero. This is achieved by introducing a new variable v_α during the key generation phase, which is computed using the random ones and in the next iteration, included in the correction words' v values. During the evaluation phase, in one side only, each v will be added to the correction terms. Due to the inclusion of v_α, they are effectively cancel each other out, leaving only a correct, but still seemingly random value behind [9].

To maintain the secrecy of α, both parties play a role in applying the correction terms and fixing the randomness along the right path. This is achieved by assigning responsibilities: Party P_0 handles the left side, and Party P_1 the right side (cf. Figure 2.23). Since the value x is publicly known, it helps determine whose turn is next [9].

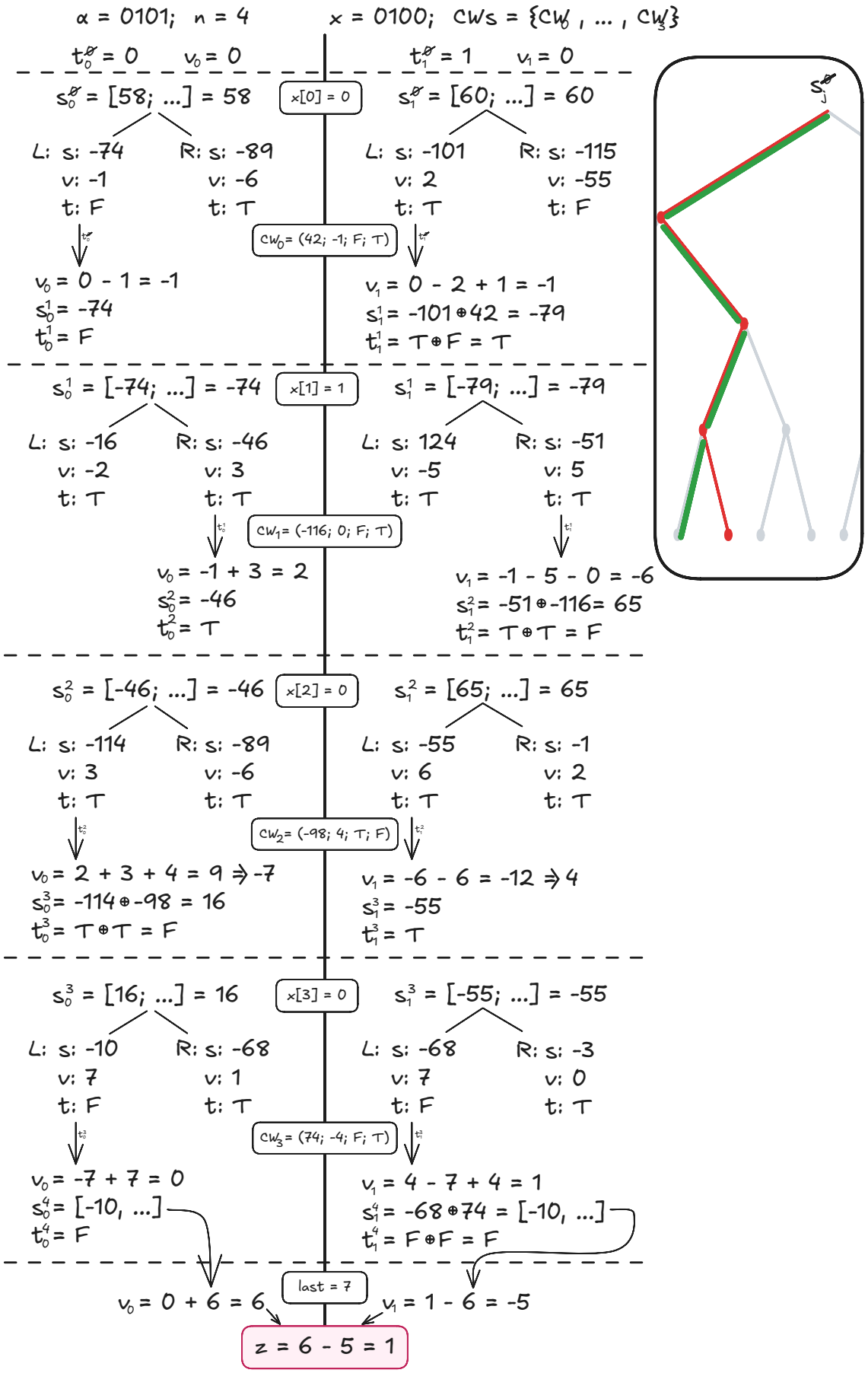


Figure 2.23: DCF Evaluation phase. Drawn and generated based on the algorithm in [9].

2.7.3 Interval Containment Gate (IC Gate)

Interval containment computation is a vital building block in secure multi-party computations, often used in scientific applications like machine learning. Originally, this was achieved by stacking two DCF constructions: the first to check whether $p \leqslant x$, and the second to verify if $x \leqslant q$. However, recent advancements [9] allow this to be done in a single generation phase with two evaluations, enabling parties to determine if a number falls within a specified interval [29]. In this thesis, Funshade uses it to compare the result of the distance metric evaluation to the specified threshold.

$$f : \{0;1\}^n \to \{0;1\} \quad f_{p,q}(x) = 1_{x\in[p,q]} = \begin{cases} 1 & \text{if } p \leqslant x \text{ and } x \leqslant q \\ 0 & \text{else} \end{cases}$$

Prior to evaluation, the key generation algorithm does not know the input value x and neither should the involved parties. To keep x secret, a masking value r is introduced [9]. However, adding a mask to x shifts the interval by r, leading to two potential scenarios (cf. Figure 2.24): the interval either stays within the specified bounds (e.g. in an $i32$ space), or it overflows and wraps around [27].

In the case where the masked interval stays within bounds, the evaluation proceeds as follows. Let $\hat{p} = p + r$ represent the masked lower bound. If the evaluation on $\hat{p}$ return -1 (and 0 elsewhere), that indicates that the input value is below $\hat{p}$. This means that even in the correct interval this evaluation returns with 0. However, if the evaluation on the upper bound $\hat{q} = q + r$ returns 1, the combined result of the two evaluations will yield $0 + 1 = 1$, confirming that x is within the interval. If $\hat{x}$ is below $\hat{p}$, the results cancel out ($-1 + 1 = 0$), while above $\hat{q}$, both evaluations return zero ($0 + 0 = 0$) [27].

The more complex case arises when the masked interval overflows the bounds. In this instance, if the same strategy applies, the final result of 1 will only occur if the masked x value falls within the wrapped-around portion of the interval.

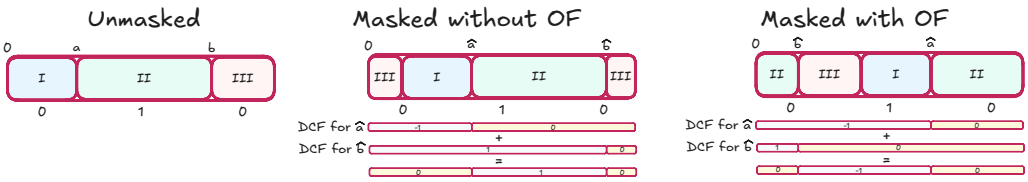


Figure 2.24: Different cases of Overflow and Interval Containment. Image is based on [27].

This approach is clearly not suitable when $\hat{x} > \hat{p}$. To remedy this problem, correction terms c are introduced, which account for different overflow scenarios [9]. The dealer, who knows the complete value of r, generates these correction terms and distributes additive shares of c to the parties, ensuring the correctness of the final evaluation without revealing the masked x value [27].

Chapter 3

Related Work

Before diving into Funshade, it is helpful to look at similar efforts in secure multi-parti computation. Funshade takes inspiration from and build upon the strengths of different projects focused on privacy-preserving biometric data handling [29]. This includes frameworks such as AriaNN, GSHADE, and ABY2.0.

AriaNN [56] is designed for secure neural network computation on sensitive data, such as private neural network training. It focuses on minimizing the communication rounds required to securely compute neural network functions. AriaNN by combining Beaver Triples and Additive Secret Sharing with FSS, achieves two rounds of communication during the online phase. As it is using only a single DCF evaluation, that would make AriaNN more efficient than Funshade. However, the two rounds of communication could create enough overhead, that in practice, the overall performance may be slower [29].

GSHADE [11] enables the computation of various distance metric for biometric identification without exposing any private data from the involved parties. To accomplish this, it breaks down the distance function into two local evaluations and a scalar product. Funshade takes inspiration from GSHADE's handling of distance metrics, though it approaches the scalar product calculation differently. Unlike GSHADE, which uses Oblivious Transfer [51] for the calculation at a cost of two rounds of communication, Funshade optimizes this process with a single-round solution.

ABY2.0 [48] introduces the concept of Π-Secret Sharing that is utilized in their protocol as well as in Funshade. This technique enables parties to generate shares of their private inputs, which are then used to evaluate the distance function. ABY2.0 for scalar product evaluation, specifies a small protocol taking the shares to form a basis. However, while this approach may be computationally efficient, it does not maintain a constant number of communication rounds.

The work by Luo et al. introduces an anonymous biometric access control system that employs homomorphic encryption [39] to perform computations on encrypted data without the need for decryption. This system verifies the membership status of a user while protecting their privacy from all involved parties. The Secure Similarity Search module within this system, based on the Hamming Distance, handles the comparison of encrypted biometric data. Although this system may ensure the privacy of the users, each iteration requires additional rounds of communication between the servers along with the computational overhead caused homomorphic encryption. While they may aim to minimize the cost, homomorphic encryption caused computational overhead can be observed even in other works, such as Ghostshell [12] and THRIVE [31] among others.

SEMBA [3], developed by Barni et al., is a secure multi-biometric authentication protocol that combines multiple biometric inputs to enhance security. It relies on SPDZ[19], a secure multi-party computation framework to combine

multiple templates within a single authentication instance. SPDZ allows several parties to perform a computation of a function without revealing the actual inputs by generating multiplicative triples using somewhat homomorphic encryption to perform secret sharing operations in the online phase. In contrast with Funshade's single-round communication phase, SEMBA's architecture requires two transmission in each of its iteration.

Lee et al. propose a privacy-preserving biometric authentication scheme focusing on computing Hamming distance [35]. To preserve privacy, the protocol relies on homomorphic encryption and their proposed primitive, Function-hiding Inner Product Encryption for Binary strings (FFB-IPE), to compute Hamming distance on encrypted data. FFB-IPE helps to encrypt and hide the base biometric template from the server, while their authentication protocol handles the distance metric evaluation on encrypted data and the verification of the user in the server-side. This construction prevents exposing the sensitive biometric information during the authentication process.

BioPass [69] and Chun et al. [13] offers constructions similar to that Lee et al. While these approaches may protect the biometric templates from exposure, they involve sending encrypted data to a server and use homomorphic encryption causing a computation overhead, which is not applicable for the Digidow use case. In Digidow, if either the PIA is or the sensor becomes malicious, one party could potentially control the result entirely, depending on which task they perform. This event is less likely to happen in scenarios where a company operates the server to access its own resources, as described in both referenced papers.

The signature scheme proposed by Takahashi et al. introduces a concept known as the fuzzy private key [61]. Compared to other, traditional cryptographic keys, this method addresses the challenge of noise and variability in biometric data when one would like to use it as a private key. Fuzzy extractors typically require an additional helper string, which the user needs to carry or store on a server. In this construction, the biometric inputs (the keys) do not need to match exactly; they allow small variances, while still maintaining secure authentication. However, in this scheme, if the user's secret key is compromised, a new key can be generated, but only for a limited number of times. This amount is dependent on the type and the quantities of adopted features.

Some of these protocols contributes to Funshade by showing how to process and secure data. AriaNN minimizes communication rounds to speed up computations, GSHADE allows multiple parties to contribute to a shared distance evaluation result without revealing private data, and ABY2.0 provides a way for two parties to compute jointly while keeping their inputs secret. Funshade combines these into a single round online communication phase and is therefore the best match latency-sensitive use cases such as those envisioned in Digidow.

Chapter 4

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

Now that the necessary context is established, this chapter will describe how Funshade [29] operates as a cryptographic solution that enhances the security of biometric authentication. Funshade "*builds upon recent advances in function secret sharing and makes use of an optimized version of arithmetic secret sharing*". This protocol addresses the need for both privacy and efficiency in (sensitive) biometric data matching applications.

In fields such as biometric authentication or machine learning, a constant challenge revolves around how to handle highly sensitive data. As previously mentioned throughout this thesis, biometric data is inherently personal, and mishandling it could lead to severe privacy violations or legal consequences. Regulations, such as the EU General Data Protection Regulation (GDPR) [46], enforce strict guidelines on how personal data should be handled and shared. However, beyond legal consideration, there are risks that institutions and individuals face when sensitive data is not secured properly. For example, if an employee's biometric data were to be stolen, it could be exploited by malicious actors, either by selling it or using it to breach company secrets. Much like the scenarios depicted in spy movies where fingerprint scanners are bypassed by using a tape. This is an actual threat in real life scenarios and it should be countered with liveness checks, such as examining sweat pores or pulse [26].

To mitigate these risks, Funshade is designed to ensure the protection of sensitive data from being exposed to malicious unauthorized entities during the authentication process. Before taking a deep dive into the inner workings of Funshade, we build an example inspired by the Digidow project [41], where Funshade can be applied to enhance biometric authentication.

Based on the three key participants of Digidow we choose a scenario, where three key components are communicating to authenticate a person's identity before granting access to a secure room:

1. **Personal Identity Agent (PIA)**

   The PIA holds the biometric template of the registered user, denoted as y. This component can be hosted by a trusted provider, such as a cloud-based service, or on a self-operated server controlled by the user.

2. **Sensor: Camera**

   A Camera sensor is operated by a third-party company and captures a person's biometric data whenever they approach the door. The captured biometric image is converted to an embedding, referred to as the live data x.

3. **Verifier: Lock**

   The Lock component controls the physical access to the room. It communicates with both the PIA and Camera to determine whether the person at the door has permission to enter. Based on the biometric comparison, the door either opens to grant access or remains closed.

The goal of this setup is to authenticate the individual at the door without exposing the biometric data of others to the PIA or the authorized persons' to the Camera, while protecting both from the Lock. The key challenge here is securely comparing the live biometric data x with the stored template y without compromising privacy.

Funshade aims to securely perform biometric authentication under a semi-honest threat model. As mentioned in Section 2.4, a semi-honest adversary follows the protocol honestly, but attempts to extract as much information as possible from the data they handle. We also refer to such adversary as *honest-but-curious*. Therefore, the protocol ensures that the biometric templates and thresholds remain hidden from all participants during the authentication process.

Besides ensuring privacy during the process, it is also important to operate efficiently. To meet both requirements, Funshade employs Function Secret Sharing (cf. Figure 4.1) combined with Π-Secret Sharing, enabling secure multiparty computation with only one round of communication during the online phase. This significantly reduces the computational overhead that is typically introduced by numerical tricks, while maintaining security and privacy [49], making Funshade appealing for practical use.

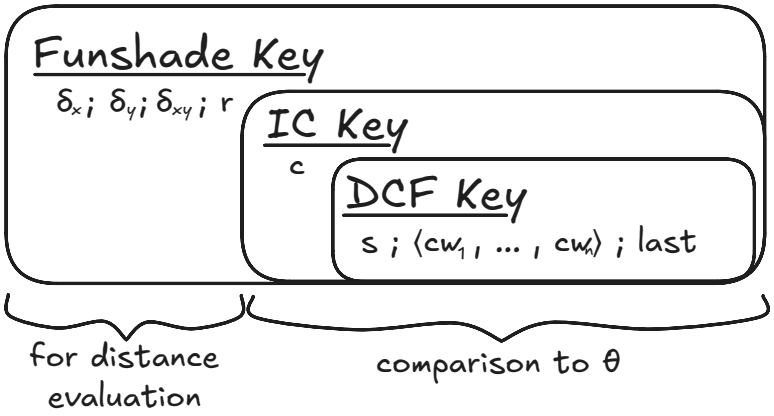


Figure 4.1: FSS keys embedded in Funshade key.

At the core of Funshade's secure computation is a threshold comparison applied to a distance metric evaluation, represented as $d(\mathbf{x}, \mathbf{y})$. This function is crucial for determining whether the live biometric data $\mathbf{x}$ matches the stored template $\mathbf{y}$, subject to a threshold θ. The following is the critical function that Funshade aims to protect using Function Secret Sharing:

$$f : \mathbb{G} \times \mathbb{G} \times \mathbb{G}^l \times \mathbb{G}^l \to \{0;1\} \quad f(d, \theta, \mathbf{x}, \mathbf{y}) = 1_{d(\mathbf{x},\mathbf{y}) \geqslant \theta} = \begin{cases} 1 & \text{if } d(\mathbf{x}, \mathbf{y}) \geqslant \theta \\ 0 & \text{else} \end{cases}$$

This function determines whether the distance between the live biometric data $\mathbf{x}$ and the stored template $\mathbf{y}$ meets a threshold θ. With Function Secret Sharing, the computation remains hidden, ensuring that neither of the involved parties

learn the actual distance or the threshold. In this way, a malicious party can not brute-force different variations of $\mathbf{y}$ until it is close enough to the threshold.

During the online phase, when both of them already knows their inputs, the Interval Containment gate evaluation requires both parties to possess the same input for the comparison, referred to as z. Therefore, the distance function $d(\mathbf{x}, \mathbf{y})$ is reformulated to fit the two-party computation (2PC) model, where each party computes an additive share of the distance evaluation result, denoted by z_j. This share is the only one that requires a round of communication in this phase. After the exchange, both parties compute their share of the comparison result and send it to the designated party.

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

The distance function is separated into two local parts, $d_{local}(\mathbf{x})$ and $d_{local}(\mathbf{y})$, which can be computed independently by each party that holds an input. The term d_{cp} represents the constant cross-product factor present in the metric. This separation allows the secure evaluation of the metric.

To clarify the process, let's see the reformulation step-by-step through an example using the Squared Euclidean Distance:

$$\begin{aligned} d_{ED}(x, y) &= \sum_{i=1}^{n} (x_i - y_i)^2 \\ &= \sum_{i=1}^{n} (x_i^2 - 2x_i y_i + y_i^2) \\ &= \sum_{i=1}^{n} x_i^2 - \sum_{i=1}^{n} 2x_i y_i + \sum_{i=1}^{n} y_i^2 \\ &= \sum_{i=1}^{n} x_i^2 + \sum_{i=1}^{n} y_i^2 + (-2) \cdot \sum_{i=1}^{n} x_i y_i \end{aligned}$$

Let $d_{local}(\mathbf{v}) = \sum_{i=1}^{n} v_i^2$ and $d_{cp} = -2$. Thus, the Squared Euclidean Distance becomes:

$$d_{ED}(x, y) = d_{local}(\mathbf{x}) + d_{local}(\mathbf{y}) + d_{cp} \cdot \sum_{i=1}^{n} x_i y_i$$

To further protect the sensitive input information, the input holders calculate the additive shares of the d_{local} computation results and sent to the parties.

$$d_{local}(\mathbf{v}) = d_{v_0} + d_{v_1}$$

While the distance function is now split into local parts, the final summation still presents a challenge. Specifically, multiplying the elements and securely adding the result requires a solution. From the earlier chapter, it became evident that for secure, privacy-preserving multiplication and addition of values, Π-secret shares are good candidates. By applying Π-shares, the sum expression becomes secure and correct, as the evaluation of Π-shares concludes with a simple addition. For the scalar product, this means summing the results of the shared evaluations ensures that each party's contributions are correctly combined.

Let $p_{j,i}$ represent the result of the secret shared multiplication for party $j \in \{0;1\}$ at elements index i (for the computation of p_j refer to Section 2.6.3):

$$\begin{aligned}\sum_{i=1}^{l} x_i y_i &= \sum_{i=1}^{l} p_{0,i} + \sum_{i=1}^{l} p_{1,i} = p_{0,1} + p_{1,1} + p_{0,2} + p_{1,2} \dots p_{0,l} + p_{1,l} \\ &= \sum_{i=1}^{l} p_{0,i} + p_{1,i}\end{aligned}$$

In the evaluation phase, the protocol introduces an additional randomized value r_{θ_j} to the equation to mask the threshold θ and secure the distance result. The Interval Containment gate of Function Secret Sharing can work with masked inputs, so this additional masking does not interfere with the protocol:

$$z = r + d_{local}(\mathbf{x}) + d_{local}(\mathbf{y}) + d_{cp} \cdot \sum_{i=1}^{l} (x_i y_i)$$

Everything considered, the parties compute the following for the comparison algorithm mixing Π-Secret Sharing and Function Secret Sharing:

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

By securely computing these values, Funshade ensures that biometric authentication is conducted both privately and efficiently.

4.1 Roles

To perform secure biometric authentication, much like other two-party computation techniques, Funshade distributes the computation across several roles, each responsible for a specific part of the process [29] as visualized in Figure 4.2:

- R_{setup}: The party with setup role covers the generation of the necessary values during the offline phase. It distributes the required keys (cf. Figure 4.1) and shares to the other parties involved in the computation. These values used during the evaluation phase to determine the final answer.

  The generation method requires the following parameters: the length of the input vector l, that represents the biometric template's size; a number of bits n, which determines the value space; the security parameter λ and the threshold θ.

  During this phase, Beaver Triples and their additive shares are generated for Π-secret sharing: $\delta_x = \delta_{x_0} + \delta_{x_1}$, $\delta_y = \delta_{y_0} + \delta_{y_1}$ and δ_{xy_j}, all of which are l-length vectors, one for each element of the inputs. Additionally, the shares of the masking value, $r_\theta = r_{\theta_0} + r_{\theta_1}$, and the key for the Interval Containment gate, k_j, are created. The setup party distributes the δ-shares to the input parties and the key, which contains the appropriate Beaver Triples, to the evaluating parties.

- R_{in_x} R_{in_y}: These roles are assigned to the input data holders with access to input x and y.

  Their responsibility is to prepare the Π-secret shares of their input and send them to the evaluating parties. Additionally, since they hold the original data, they compute the local distance function d_{local} and they forward additive shares of the result along with the Π-shares.

  They are preparing the Π-secret shares that they send to the computing parties. Additionally, the parties are calculating the d_{local} function as they are the ones who own the original input data and send additive shares, d_{v_0} and d_{v_1}, to the evaluating parties.

  If the input vectors are available during the offline phase, this computation can take place then. Otherwise, it is performed at the beginning of the online phase when the inputs become available.

- P_x, P_y: These parties are responsible for the evaluation part during the online phase. After receiving the remaining part of Π-shares from the input parties, they use the data contained in the keys to calculate the additive shares of the distance result z_j.

  Once they exchange these shares and compute the masked result of the distance function, they evaluate the Interval Containment gate to determine if the result is below the secret threshold θ. The output, o_j, will be sent to the designated authorized party.

- R_{res}: This party receives the result shares from the secure computation and uses them to reconstruct the result.

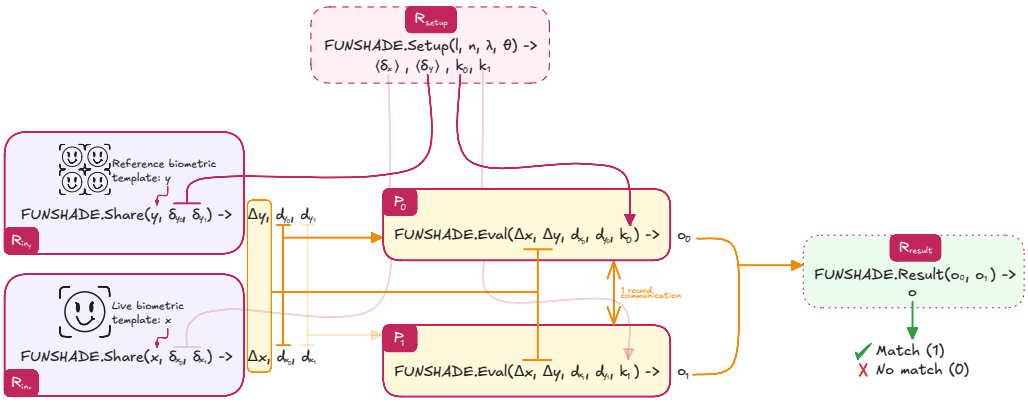


Figure 4.2: Funshade roles and their relations. Image is based on [29].

Multiple roles can be assigned to the same party, depending on the specific use case. For instance, the computing parties might also act as the input holders. Obviously, this does not rule out that every role is performed by different parties.

It is also important to consider the setup phase of Funshade, which can be handled in three different ways (cf. Figure 4.3) [29]:

1. **Trusted hardware:** "*The* R_{setup} *role can be emulated within a trusted execution environment*", providing secure generation and distribution of values.
2. **Semi-honest third-party:** A semi-honest third party takes the R_{setup} role, distributing keys and shares to the other participants, while adhering to the protocol without malicious intent.

3. **Pure 2PC:** Two parties collaboratively handle the setup, each acting as both input holders and evaluators. This method may require additional communication rounds during the offline phase to complete the setup securely.

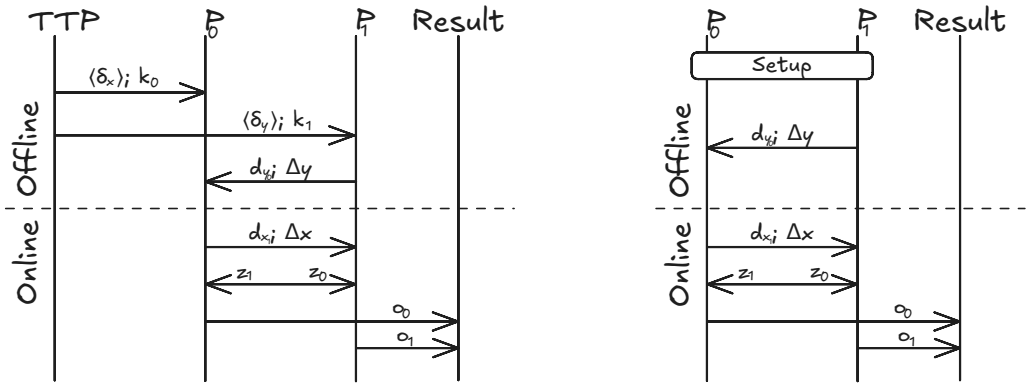


Figure 4.3: Funshade process with third-party setup (left) and 2PC (right).

Funshade can be applied to the Digidow-inspired biometric authentication scenario. Assume that Alice is the only individual authorized to access a secure room guarded by the Lock component, with her PIA storing her biometric template y. Additionally, let the PIA and Camera operate in a Pure 2PC way, meaning that they collaboratively perform the setup, both acting as an input holder and carry out the evaluation. The Lock component will be the designated party receiving the evaluation results. Figure 4.4 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, including the Beaver Triples, derived from them the δ-shares and keys for Funshade protocol.

   After finishing the process, the PIA owns δ_y and the Funshade key for the protocol k_0 containing the Beaver Triples δ_{x_0}, δ_{y_0} and δ_{xy_0}. Similarly, Camera possess δ_x and k_1 with δ_{x_1}, δ_{y_1} and δ_{xy_1}.

2. Since the PIA already possesses Alice's data, it can pre-compute 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. Somebody passes by, therefore the online phase begins. The camera captures a person's live biometric data x and generates the corresponding shares. These shares are sent to the PIA for evaluation.

4. The construction of the additive shares for the distance evaluation begins as both parties possess the secret shares derived from the inputs $\mathbf{x}$ and $\mathbf{y}$. At this point, they require a round of communication to produce the common value representing the masked distance calculation result. To conclude the process, both evaluate the Interval Containment gate and send the values to the Lock component.

5. The Lock component combines the values and either grants or denies access to the person standing at the door.

Through this process, Alice's biometric template is never revealed to the Camera, and the live biometric data remains unknown to the PIA. For instance, if Bob just walked down in the corridor, his data never became exposed to Alice's PIA. Funshade ensures that privacy is preserved while maintaining a secure and efficient authentication process.

By utilizing Funshade, a system can be created where sensitive biometric data remains private during authentication, even when multiple parties are involved.

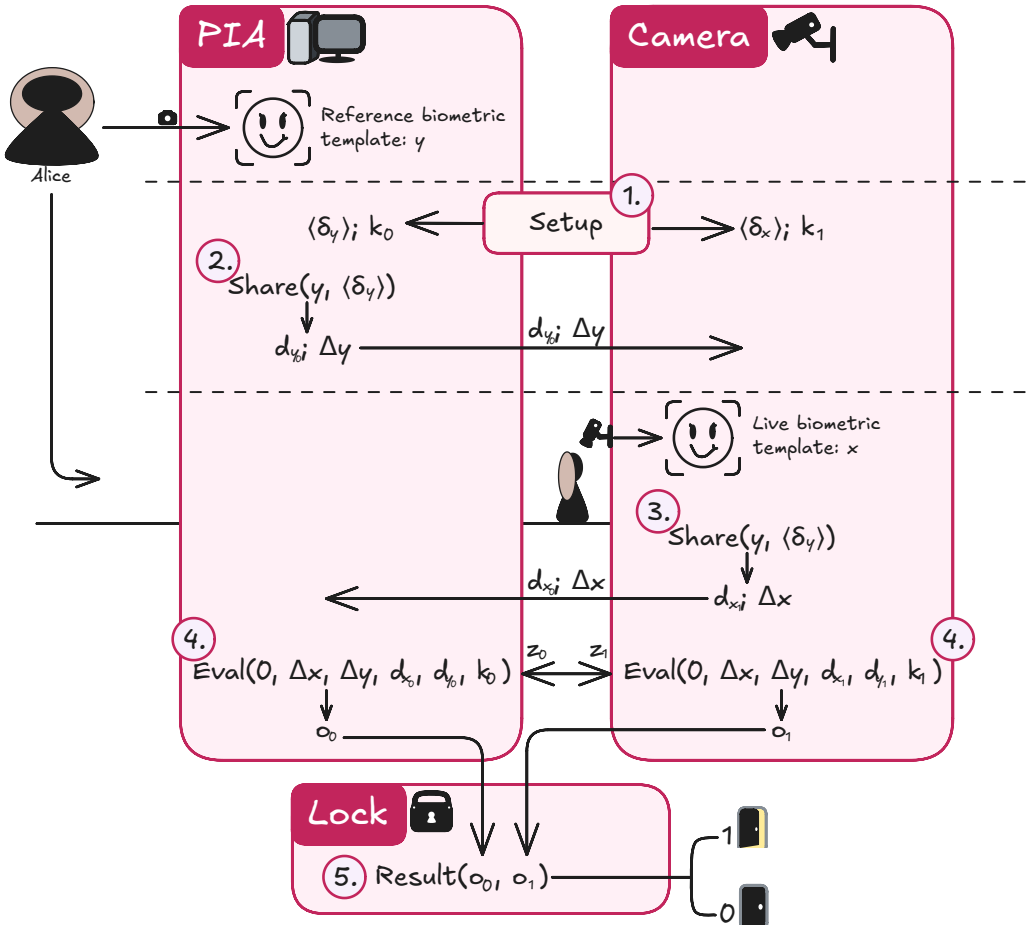


Figure 4.4: Digidow-inspired example of Funshade. Image is partially based on [29].

4.2 Two-Party Scenario

This section outlines the two-party computation (2PC) setup process and explains why it became a topic for future work during this thesis. To summarize, the 2PC setup process involves two participants collaboratively and securely generating the necessary materials for the protocol's execution.

Ibarrondo et al. [29] provides a guideline for constructing the protocol in a 2PC setting. Specifically, for distance function evaluation, ABY2.0 [48] describes the solution for Π-secret shares in 2PC. This can be implemented using Homomorphic Encryption [22, 54], which allows operations to be performed on encrypted data.

Distributed Comparison Function [9] also briefly describes the protocol in a secure two-party environment. In Funshade, this method can be applied for the Function Secret Sharing gate key generation. During this process, a participant holding parameters $\alpha \in \{0;1\}^n$ and β splits them to additive shares, keeping

the values secret from both parties. The algorithm generates a pseudo-random vector with 2^i element in each iteration i and relies on the XOR operation to hide the value. A secure computation method generates correction words that ensure values in incorrect path cancel each other out.

However, performing this calculation with a 32-bit input parameter may not be feasible for every machine, especially if communication is required in every iteration. The storage requirements could become problematic since the generated values are used in subsequent iterations. When $i = 32$ and the security parameter $\lambda = 128$, the algorithm requires approximately $2^i \cdot (|s_j^i| + |v_j^i| + |t_j^{L,i}| + |t_j^{R,i}|) = 2^{32} \cdot (128 + 32 + 8)$ bits for a single key. Even if this amount only needed to finish the generation, this value may be kept in memory for a longer time than intended due to the communication delays.

Reducing the input size to 16-bits could mitigate these issues. However, depending on the distance metric used in Funshade and on how much precision one is willing to sacrifice, this approach might not be ideal for all scenarios. Funshade might produce incorrect results if an overflow happens during the calculation of the masked distance value, z, meaning that the distance function and the number of elements highly influence the permitted values in the biometric template. For instance, with a 32-bit limit, $l = 512$ as the length of the vectors and using Scalar Product as the distance metric, the maximum value m can be expressed as:

$$\begin{aligned} 2 \cdot \log_2 m + \log_2 l &\leqslant 32 \text{ bits} \\ m &\leqslant 2^{\frac{32 - \log_2 512}{2}} \\ m &\leqslant 2^{11.5} \end{aligned}$$

After exchanging messages with one of the authors, it became clear that there is no definitive, written process for performing the 2PC setup yet. Additionally, the current work only assumes semi-honest adversaries, not malicious ones. The protocol is already planned for further research, not only to adapt to 2PC but also to address potential attacks from malicious actors. This area represents a promising direction for future research, and I hope to contribute to it in the near future.

Chapter 5

Rust

Systems programming as an expression might sound unfamiliar, but it is used almost everywhere. Systems programming is resource-oriented, where every byte and CPU cycle counts [6]. It is used in game development, networking, operating systems, cryptography, etc.

Rust [55] is a modern, system-level programming language that emphasizes safety, performance, and concurrency [33]. It was designed to resolve many common problems found in older languages like C and C++, such as dangling pointers and null pointer dereferences [6]. This is achieved by the strict compile-time checks and the unique ownership model. Additionally, the language provides memory safety without a garbage collector.

For this thesis, Rust was chosen due to its memory safety features at compile-time without runtime overhead. Its distinctive ownership system prevents memory-related errors, that are common sources of bugs in languages like C and C++. As data can not be passed around freely, it also ensures memory safety when using parallel threads. Rust is also the language that was used to develop other components for the Digidow project [41], making it easier to integrate with the larger system.

Rust's build system and package manager is called Cargo. It manages Rust projects, downloads external libraries, builds both the dependencies and code [33]. Rust libraries, called crates, are listed in the `Cargo.toml` configuration file, where project details like name, version, and Rust edition are specified (cf. Listing 5.1).

```
[package]
name = "funshade"
version = "0.1.0"
edition = "2021"

[dependencies]
aes-prng = "0.2.1"
```

Listing 5.1: `Cargo.toml` file in thesis project.

Cargo can be used to create a new project, build both in development and release mode, execute, and run tests (cf. Listing 5.2). By applying the `run` command, Cargo will automatically build the project.

```
cargo new funshade_rust
cargo build
cargo build --release
cargo run
cargo test
```

Listing 5.2: Basic Cargo commands to create, build, run, and test a project.

During building the project in `--release` mode, the compiler will make some optimizations, therefore the software will be faster than what the developer works with. One such optimization is allowing overflows during operations like addition `+`, which is not available during development in this form. The developer needs to tell the compiler if the overflow is intentional using the `wrapping_add` method, otherwise they receive an error. In the release build the compile switch the method call to a simple addition.

Cargo automatically calls the Rust compiler, `rustc`. One of `rustc`'s uncommon features is the excellent error reporting. Unlike many compilers that provide hard to understand error messages, `rustc` gives detailed feedback and suggestions for fixing issues [33].

Rust supports modules that enable to organize and separate code for readability and easy reuse [6, 33]. Modules can be defined with the `mod` keyword, and a corresponding `.rs` file or a new folder with a `mod.rs` file. By default, every element in Rust is private, but this structure allows defining flexible protection levels. The two common levels, `pub` (public) and `private`, might be familiar from other programming languages. Aside from these, there is `pub(super)`, which makes the element visible only to the parent module; `pub(crate)`, visible to the whole project, but hidden to the user; and `pub(path:to:module)` visible to a specific module.

The following sections describe the functionalities provided by Rust used in this thesis. The official book [33] was the mainly used learning material to get into Rust and the main inspiration for writing this chapter.

5.1 Basic Concepts

In Rust, variables are designed to help developers write safe and efficient code. By default, the variables are immutable to prevent unintended changes. While mutability is allowed, it needs to be explicitly told to the compiler to denote that this is an intended behaviour.

Variables are declared using the `let` keyword. As they are immutable, their values can not be changed once they are set.

```
1  let x: i32 = 5;
2  x = 6; // compile error
```

Listing 5.3: Declaring a variable and attempting to change its value results in a compile-time error.

To change the value of the variable, it must be declared explicitly as mutable with the `mut` keyword:

```
1  let mut y: i32 = 5;
2  y = 6; // OK
```

Listing 5.4: Declaring a mutable variable and changing its value.

This is one of Rust's safety features, forcing the developer to think carefully about when a value should be changed. Understanding the basics of variables helps write clean, efficient, and safe Rust code.

Rust also supports constants, which are like immutable variables but are evaluated at compile-time, therefore their values can not be changed. It can be declared using the `const` keyword:

```
const SECURITY_PARAM: usize = 128;
```

Listing 5.5: Declaring a constant value.

Rust allows shadowing variables: Upon declaring a new variable with an already existing name, the new shadows the old one. In the background, it reserves space for both of the values. Declaring the variable switches the reference to the second memory address. This also enables to declare the variable with the same name inside the same or in a new scope, and to transform a variable's value without declaring it mutable. In other words, it does not overwrite the old value, but the place where the variable points to is overwritten until the end of the scope:

```
let x: i32 = 5;
{
  let x: i32 = 6;
  // Operations with x = 6
} // End of scope, x = 6 dies
// x is 5 again
```

Listing 5.6: The second `x` declaration shadows the first one. After it dies at the end of the scope, the value of `x` becomes 5 again.

Shadowing not only allows a variable's value to be redefined, but it also enables changing the its type:

```
// The variable spaces declared as a String with 5x" "
let spaces: String = String::from("     ");
// The cariable spaces points to the integer with the value 5.
let spaces: i32 = spaces.len();
```

Listing 5.7: The second `spaces` declaration shadows the first one's value and type.

Functions in Rust are defined using the `fn` keyword followed by the function name, parameter list, and function body enclosed in curly braces. Parameters require both a name and type in `name: type` format. Functions returning values specify the return type after `->`, and the `return` keyword is only required for early exits, not for the final returned value.

```
fn calculate_number_of_bytes(bits: usize) -> usize {
  if bits == 0 {
    return 0;
  }

  (bits - 1) / 8 + 1
}
```

Listing 5.8: Defining a function that returns with an unsigned value.

Rust is a statically-typed language, meaning that the compiler needs to know the type. The most common variables are signed (e.g., `i32`) and unsigned integers (e.g., `u8`), floating-point numbers (e.g., `f64`) and boolean (`bool`). Rust allocates variables on the stack by default, therefore variable sizes must be known at compile-time. Dynamic allocation on the heap is explicitly requested by using `Box`. This also means that an array with arbitrary length can not be instantiated, as a user specified value can not be known at compile time. For instance, the code presented in Listing 5.9 will not compile.

```rust
// Compiler error: Doesn't have a size known at compile-time,
//                 consider borrowing
fn create_array_with_ones(l: usize) -> [u8; l] {
  let result: [u8; l] = [0; l];
  result
}
```

Listing 5.9: Attempt to create a dynamically sized array, resulting in a compile error.

Rust's `if` and `else` syntax resembles that of C#, while `switch` cases are implemented as `match` arms:

```rust
let t: bool = true;

let sign: i32 = if t { -1 } else { 1 };

let sign = match t {
  true => -1,
  false => 1
};
```

Listing 5.10: Conditional assignment with if and match expressions.

Rust supports loops that in the thesis were used in two ways. First, when it iterates in a specified range taking the integer values one by one, e.g., `0..alpha_bits.len()` in C# would be `int i = 0; 0 < alpha_bits.length; i++`. The second method employs iterators, where each iteration retrieves the next array element. For example `alpha_i` at the start of the loop will be the first bit in `alpha_bits`:

```rust
for i in 0..alpha_bits.len() {
  // content
}
// or
for alpha_i in alpha_bits.iter() {
  // content
}
```

Listing 5.11: Range-based and iterator-based for loop examples.

In Rust and other programming languages, tuples allow grouping multiple values into a single compound type. Each element can be of any type, but once they have declared their length can not be changed. These elements can be accessed using a dot notation with the index of the element. Destructuring also allows extracting the values into separate variables. This makes tuples a good choice for returning multiple values from a function.

```
fn setup(settings: &FunshadeSettings, theta: TYPE) ->
(FunshadeKey, FunshadeKey, DeltaShare, DeltaShare) {/*...*/}// ...

//...
let tuple: (FunshadeKey, FunshadeKey, DeltaShare, DeltaShare) = setup(&s, theta)
  ;

// Reach elements
let key0: &FunshadeKey = &tuple.0;
let key1: &FunshadeKey = &tuple.1;
let share0: &DeltaShare = &tuple.2;
let share1: &DeltaShare = &tuple.3;

// Destructure elements
let (key0, key1, share0, share1) = tuple;

```

Listing 5.12: Tuple usage example, with both direct access and destructuring of elements.

5.2 Ownership and Borrowing

Rust's ownership system is one of its most distinctive features. Each value has a single so-called owner, who owns this value. This value lives until its owner goes out of scope, ensuring memory is freed when it is no longer needed. The ownership can be transferred to another variable, this means that if the value's ownership is given to another variable that is outside the scope, then the value will live longer than its original owner.

Ownership transfer occurs by default when assigning values, but this does not apply to all types. Every object whose type implements the `Copy` trait (see Section 5.4) will automatically clone the value and becomes its owner. Every basic type implements this, integers, boolean, floating point numbers etc., however the String type does not.

```
let a: i32 = 5;
let b: i32 = a; // value of a is copied: b = 5
println!("{}", a); // No error

let hello: String = String::from("Hello!");
let hello_again: String = hello; // The ownership of value "Hello" is
  transferred
println!("{}", hello); // Compile error: value moved
```

Listing 5.13: Ownership transfer with `Copy` and `String` types.

An important thing to mention is that this rule is not only constrained to variables, but also to parameters and return values:

```
fn main() {
  let a: String = String::from("Hello!");
  let b: String = takes_ownership(a); // function takes ownership and gives it
  to b
  println!("{}", a); // Compile error: value moved
}
fn takes_ownership(s: String) -> String {
  s
}
```

Listing 5.14: Ownership transfer when calling functions.

Requiring ownership transfer for each access would complicate working with non-`Copy` types, especially for complex computations and software. Therefore, Rust allows references to a value without taking ownership with `&`. However, if the owner goes out of scope and the value will be destroyed, this variable will not point to any. To remedy this problem, Rust's compiler will take care of this by analyzing lifetimes to avoid dangling reference. If the value goes out of scope before the variable that stores its reference, then the program will not compile and write a report about the bug.

Sometimes, it is necessary to modify a value without transferring ownership. In these cases, Rust allows mutable references using `&mut`, making value modification via the reference possible:

```
fn main() {
  let a: String = String::from("Hello!");
  let b: &String = takes_reference(&a); // function takes and gives reference to
   the value "Hello!"
  println!("{}", a); // No error
  takes_mutable_reference(&mut a)
}
fn takes_reference(s: &String) -> &String {
  s
}
fn takes_mutable_reference(s: &mut String) {
  s = String::from("Hello World!");
}
```

Listing 5.15: Example of immutable and mutable references.

If the code is well-written and the developer took great attention to it, then Rust could achieve a clean and readable structure. For example, just by looking at a function, one can tell that the function will modify the value or not. If someone would like to use the `takes_mutable_reference` function, and reads its signature, they would know that this will change the given value, while `takes_reference` will not, as it is just a simple reference.

5.3 Custom Types

Most of the time, structs might be better suited than tuples. Structs have names that describe what they group together. Each piece of data is named after what they represent, in order to know what they should be used for, just like in most programming languages. These need to be instantiated to be used. Even if there are no constructors in the traditional sense, they can be instantiated by specifying the values for each field, or writing a "constructor of our own".

```
pub(super) struct DcfNode {
  s: AesSeed,
  v: i32,
  t: bool
}

// ...
let node = DcfNode {
  s: AesSeed::new_random(),
  v: 234,
  t: false
}
```

Listing 5.16: Basic struct instantiation.

Rust provides structs similar to tuples. They have a name, but no added meaning to their fields. These are useful, when the tuple should have a name to differentiate it from the other tuples. Their creation starts with the keyword `struct`, a name and the types in brackets. They can be used as struct, the only difference is how the fields are accessed.

```
struct AesSeed([u8; SEED_BYTES]);
```

Listing 5.17: Tuple struct example.

Enumerations, or also referred to as enums, give a way to say that a value is one from a possible set of options. They can be created with the `enum` keyword, defining its name and listing the options in curly brackets. They can not only store single options, but in a form of a tuple struct they can store data, each can be defined with different length and types.

```
enum Message {
  Init,
  Setup,
  DeltaShare(DeltaShare),
  OValue(bool, TYPE),
  // ...
}
```

Listing 5.18: Enum definition with variant data types.

Tuple structs and structs can also be instantiated by using methods, which are all defined in the context of a struct, an enum or a trait object. In the scope of structs and enums, they must be placed in an `impl` block. This is Rust's way to draw a clean line between the data and the behaviour. They also possess a `Self` keyword which refers to the struct's type. Aside from this, they can also refer to themselves — like in C# — with `&self`, explicitly mentioned in the method's signature.

```
struct AesSeed([u8; SEED_BYTES]);

impl AesSeed {
  // "Constructor"
  pub fn new_random() -> Self {
    Self(AesRng::generate_random_seed())
  }

  // A method with &self
  pub fn xor(&self, other: &AesSeed) -> AesSeed {
    let mut result: AesSeed = AesSeed::new_empty();

    for i in 0..SEED_BYTES {
      result[i] = self[i] ^ other[i];
    }

    result
  }
}

//...
let seed0: AesSeed = AesSeed::new_random();
let seed1: AesSeed = AesSeed::new_random();
let xored: AesSeed = seed0.xor(seed1);
```

Listing 5.19: Method and "constructor" definitions for a struct.

5.4 Traits

Besides using structs and their implementation block, another way to separate behaviour from data with the advantage of defining common behaviour, is traits. They are similar to interfaces and abstract classes in other languages, like Java or C#, allowing Rust to support a kind of polymorphism.

A trait defines a set of methods that a type can implement, either by providing a pre-defined behaviour or requiring the type to implement it. Compared to abstract classes, the difficulty to write traits stems from the fact that they are completely separated from the data. They should specify the data in the method's signature or let the type that already has access to the data implement it.

Traits can be created with the `trait` keyword, followed by methods created in a block defined by curly brackets. A trait can also build upon other traits, specified by using a `:` after the trait name. This is called trait inheritance and allows defining a trait that requires another trait to be implemented.

To implement a trait for a type, Rust uses `impl` and `for` keywords, as "implement the trait for struct". If any methods are left unimplemented, then the compiler will ask to complete the trait's definition.

```
trait Party : FunshadeSession {
  fn send_message(stream: &TcpStream, message: &Message) {
    // do something that uses FunshadeSession's methods
  }

  fn handle_init_message(&self);
}

// ...
struct SetupParty {/* some data */}
impl SetupParty {...}
impl FunshadeSession {...}
impl Party for SetupParty {
  fn handle_init_message(&self){ /* do something with data */ }
}
```

Listing 5.20: Example of trait definition and implementation.

However, not any trait can be implemented to any type. Due to compatibility reasons, the developer is not allowed to implement a crate's trait for a type from another one. If the developer wished to use a new crate that does the same, then these two implementations at once could cause serious and unexpected bugs. For instance, the `Add` trait from the standard library can not be implemented to `String`.

5.5 Function Pointers, "Delegates"

Rust supports anonymous functions behind the alias of closures, which can be stored in a variable and passed to functions. Closures are allocated on the stack, providing fast access [6]. They are defined as follows:

```
let add = |x| x + 1; // or |x: i32| -> i32 {x + 1};
```

Listing 5.21: Defining a closure.

In addition to closures, Rust supports function pointers, which were already mentioned in Section 5.1 under the type `fn`. The main difference between function pointers and closures is that closures can capture their environment. This means that from the scope they defined, they can use variables and their values when they are called. Function pointers are not capable to use outside values in this form.

```
let hello = String::from("Hello");

let closure = |x| println!("{hello} {x}");

closure("World!");
```

Listing 5.22: Closure usage.

Function pointers and closures can be stored in variables and even struct fields. Since both implements the `Fn` closure trait, functions that expect closures can also accept function pointers. However, because the size of closures and functions is not always known, this will cause a compile-time error. Allocating on the heap by using the `Box` type, will solve this problem.

To specify that a function only accepts function pointers, the `fn` type can be used. This distinction is useful, when handling external code, like C, that only supports function pointers.

```
// Struct only accepts function pointers
struct DistanceMetric {
  pub(crate) f_local: fn(&[TYPE]) -> TYPE,
  pub(crate) f_cp: TYPE
}

// Struct accepts both function pointers and closures.
struct DistanceMetric {
  pub(crate) f_local: Box<dyn Fn(&[TYPE]) -> TYPE>,
  pub(crate) f_cp: TYPE
}
```

Listing 5.23: Struct field types accepting function pointers and closures.

5.6 Error Handling

Errors are inevitable in any software, regardless of the language. Rust categorizes errors into two categories: unrecoverable and recoverable.

For unrecoverable errors, Rust uses `panic!` which immediately stops execution. This equals to leaving out a try-catch in C# or Java that would handle the exception.

For recoverable errors, Rust provides the `Result<T, E>` enum, that contains two variants: `Ok(T)` which holds successful result of type `T` and an `Err(E)`, which holds an error type. The main advantage of this structure is that the function explicitly signals to the user that using it could result in an error that needs to be handled in match arms.

```rust
fn main() {
  let greeting_file_result = File::open("hello.txt");

  let greeting_file = match greeting_file_result {
    Ok(file) => file,
    Err(error) => panic!("Problem opening the file: {error:?}"),
  };
}
```

Listing 5.24: Error handling with the Result type.

5.7 Rust Documentation

Documentation is essential in any software, serving both developers and users by explaining functionality, usage, and maintenance. In Rust, documentation is integrated into code through documentation comments, denoted by `///`. It supports Markdown notation and creates HTML pages upon executing the `cargo doc` command.

Documentation can be added to almost any element, including modules, data types and methods. For each element, the first paragraph should briefly describe its purpose, followed by a more detailed explanation if needed. In case of data types, like structs, description can be provided for the entire struct as well as for individual fields.

For methods and functions, it is recommended to specify conditions under which the function might panic, along with an example demonstrating its usage. These examples are runnable, so even the compiler checks their validity, which is especially important when a change happens in the codebase.

```rust
/// A short description.
///
/// A detailed description.
///
/// # Panics
///
/// Cases when the program panics
///
/// # Examples
///
/// ```rust
/// // Some example code
/// ```
pub(crate) fn bit_representation_with_len(num: usize, number_of_bits: usize,
  order: &BitOrder, signed: &Sign) -> Vec<bool> {...}
```

Listing 5.25: Inline documentation for a struct and its fields. See result in Figure 5.1.

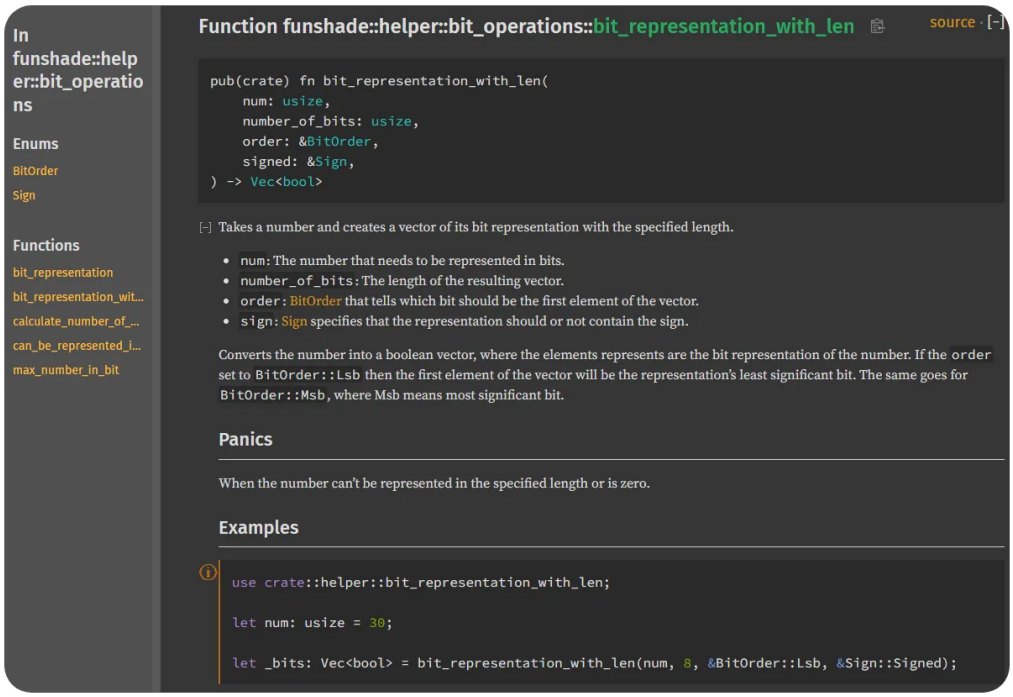


Figure 5.1: HTML page generated from documentation.

Chapter 6

Rust Implementation

This chapter details the implementation of Funshade using the Rust programming language. Building on the theoretical foundations discussed earlier, the focus here is translating those concept into a functional system. The main inspiration was the C implementation created by the original authors [28].

The codebase for this thesis is structured in a way to ensure clarity and maintainability. Each module is designed to handle a specific set of specific functionalities, ensuring separation of concerns. The FSS module encapsulates the capabilities of Function Secret Sharing, while Funshade logic is housed in its own. In addition, helper functions, data structures and role-specific logic for different parties are organized in their own modules.

The following sections explore these components in detail, highlighting key design decisions, potential future improvements and the integration of external libraries. The attached implementation is available in Appendix A.

```
FUNSHADE-RUST
└── src
    ├── fss
    │   ├── dcf
    │   └── ic
    ├── funshade
    ├── helper
    └── sessions
        └── third_party_session
```

Listing 6.1: Module hierarchy in Funshade project generated with the `tree .` command.

6.1 External Crates

The Funshade implementation uses a few external crates to manage random number generation, data serialization and command-line interface parsing. These crates provide well-tested solutions, allowing the focus to remain on developing the Funshade-specific functionality.

1. `aes-prng` [1]: This crate provides a pseudo-random number generator [38] using AES [18, 62] as the underlying mechanism. It depends on the `aes` crate, which implements the AES and `rand` crate for random number utilities. This crate is used during the key generation phase of Function Secret Sharing, applying for the function $G : \{0;1\}^{\lambda} \rightarrow \{0;1\}^{2(2\lambda+1)}$.
2. `rand` [52]: This crate provides utilities for random number generation. It is used in Funshade to generate the random mask, Beaver Triples, test biometric templates and general test cases.

3. `clap` [14]: This crate simplifies the process of parsing a command-line arguments through the terminal. In Funshade, `clap` is used in `main.rs` to enable running and testing the project via the command-line. For example, the following command can be used to simulate the PIA party:

```
    cargo run -- -m 1 --pia
```

Listing 6.2: Execute the software with a PIA role.

The `clap` crate works by defining custom command-line arguments within a Rust data structure. This crate provides an API that supports settings for both short (`-p`) and long (`--pia`) flags, as well as defining default values for the arguments in the case the user does not specify them. Here is an example of how it is integrated:

```
  #[derive(Parser, Debug)]
  #[command(version, about, long_about = None)]
  struct Args {
    // ....
    /// Use PIA
    #[arg(short, long, default_value_t = false)]
    pia: bool,
    // .....
  }
```

Listing 6.3: `Args` data structure for `clap` crate.

In this example, the `Args` struct defines the custom command-line argument for the `--pia` flag, which defaults to `false` if not provided. The `#[command(version, about, long_about = None)]` macro, provided by `clap`, sets metadata for the CLI such as the version and description. The properties of the argument (`#[arg(short, long, default_value_t = false)]`) specifies both short and long flags for it and ensures a default value is assigned.

Although Funshade is primarily designed as a library, incorporating `clap` provides a convenient interface for running the program from the command line. This functionality is not essential for the core library but is essential for the purposes of this thesis, as it allows for the straightforward execution of the program and testing of different scenarios.

4. `serde` [57] and `serde_json` [58]: These crates form a framework for serializing and deserializing Rust data structures. In Funshade, they are used to format messages exchanged between different parties.

These directly used crates pull in a larger dependency tree as depicted in Listing 6.4. The `clap` branch can be avoided when using this implementation only as a library.

```
funshade v0.1.0
├── aes-prng v0.2.1
│   ├── aes v0.8.4
│   │   ├── cfg-if v1.0.0
│   │   ├── cipher v0.4.4
│   │   │   ├── crypto-common v0.1.6
│   │   │   │   ├── generic-array v0.14.7
│   │   │   │   │   └── typenum v1.17.0
│   │   │   │   │   [build-dependencies]
│   │   │   │   │   └── version_check v0.9.4
│   │   │   │   └── typenum v1.17.0
│   │   │   └── inout v0.1.3
```

```
│   │   │   │   └── generic-array v0.14.7 (*)
│   │   │   └── cpufeatures v0.2.12
│   │   ├── byteorder v1.5.0
│   │   └── rand v0.8.5
│   │       ├── rand_chacha v0.3.1
│   │       │   ├── ppv-lite86 v0.2.17
│   │       │   └── rand_core v0.6.4
│   │       │       └── getrandom v0.2.12
│   │       │           └── cfg-if v1.0.0
│   │       └── rand_core v0.6.4 (*)
│   ├── clap v4.5.4
│   │   ├── clap_builder v4.5.2
│   │   │   ├── anstream v0.6.13
│   │   │   │   ├── anstyle v1.0.6
│   │   │   │   ├── anstyle-parse v0.2.3
│   │   │   │   │   └── utf8parse v0.2.1
│   │   │   │   ├── anstyle-query v1.0.2
│   │   │   │   │   └── windows-sys v0.52.0
│   │   │   │   │       └── windows-targets v0.52.5
│   │   │   │   │           └── windows_x86_64_msvc v0.52.5
│   │   │   │   ├── anstyle-wincon v3.0.2
│   │   │   │   │   ├── anstyle v1.0.6
│   │   │   │   │   └── windows-sys v0.52.0 (*)
│   │   │   │   ├── colorchoice v1.0.0
│   │   │   │   └── utf8parse v0.2.1
│   │   │   ├── anstyle v1.0.6
│   │   │   ├── clap_lex v0.7.0
│   │   │   └── strsim v0.11.1
│   │   └── clap_derive v4.5.4 (proc-macro)
│   │       ├── heck v0.5.0
│   │       ├── proc-macro2 v1.0.81
│   │       │   └── unicode-ident v1.0.12
│   │       ├── quote v1.0.36
│   │       │   └── proc-macro2 v1.0.81 (*)
│   │       └── syn v2.0.60
│   │           ├── proc-macro2 v1.0.81 (*)
│   │           ├── quote v1.0.36 (*)
│   │           └── unicode-ident v1.0.12
│   ├── rand v0.8.5 (*)
│   ├── serde v1.0.202
│   │   └── serde_derive v1.0.202 (proc-macro)
│   │       ├── proc-macro2 v1.0.81 (*)
│   │       ├── quote v1.0.36 (*)
│   │       └── syn v2.0.60 (*)
│   └── serde_json v1.0.117
│       ├── itoa v1.0.11
│       ├── ryu v1.0.18
│       └── serde v1.0.202 (*)
```

Listing 6.4: Dependencies used in Funshade project generated with `cargo tree` command.

6.2 Helpers

This section details the helper data structures and methods employed in the Funshade implementation. These components simplify operations such as data representation, conversions, and scaling, which are fundamental for execution. They also provide data structures used consistently throughout the project.

6.2.1 Group Data Structure

The `Group` structure addresses the challenge of managing arbitrary bit widths in computations, particularly in the context of 2PC DCF key generation. This involves hiding input shares and seeds behind 2^i random generated values. Section 4.2 mentioned the possible reduction in input size, however, it also addressed a potential precision loss, which may not be ideal depending on the situation and the chosen distance metric.

For instance, while the computations with 32-bit integers may be overly time-consuming, 16-bit integers may lack sufficient precision. The potential need for intermediate bit sizes between 16 and 32 bits became evident, and as a result, Funshade was modified to be able to handle different bit sizes that entailed a lot of interval calculations.

This gave rise to the `Group` structure, which encapsulates the desired bit width, calculating signed maximum and minimum values upon instantiation, along with a range mask for wraparound operations. The wrap operation ensures that values exceeding the interval, wrap around within the group.

In addition to storing the frequently used values, `Group` provides basic arithmetic operations, like addition, subtraction and multiplication, each of which returns a result that is wrapped around the interval defined by the group.

6.2.2 Funshade Settings Data Structure

The `FunshadeSettings` manages configuration parameters used throughout the project. Currently, it stores settings such as bit-length for secure 2PC computation and the length of accepted input vectors.

The structure was initially introduced to hold frequently used data pairs, like bit length, maximum value and vector length, which appeared repeatedly in method headers. Encapsulating these parameters in a dedicated structure not only made the program more readable, but also ensures that all necessary settings are readily available for the parties as they rely on these parameters.

Future versions of this structure are expected to manage custom settings (e.g. PRNG) and store more pre-computed values for processing. These values might include, but are not limited to, `max_el`, the maximum value that can appear in input vectors and other global configurations like the security parameter λ.

6.2.3 Bit Operations

The bit operations module purely contains methods for bit-level manipulation and calculation. This module contains the functionality for determining how many bits are required to represent a specific number of bytes

(`calculate_number_of_bytes`) and to calculating the maximum value that can be represented in a given number of bits (`max_number_in_bit`).

Perhaps the most frequently used function is `bit_representation_with_len`, which constructs the bit representation of a given number. This is a crucial part for building the DCF tree both in the generation and in the evaluation algorithm. The algorithm supports signed and unsigned representation with the help of a `Sign` enum. This distinction is necessary because the DCF method operates on signed (`i32`) integers, while unsigned (`u8`) representation is needed for byte-to-bit conversions. Although generics could improve this, it is more challenging to implement in Rust compared to languages like C#. As a result, the Rust community generally recommends using the `num_traits` [45] crate for this purpose.

The similar problem for `Sign`, also stands for `BitOrder` enum, which differentiates between least significant (LSB) and most significant (MSB) bit ordering. While LSB ordering is required for the number conversion by definition, the pre-existing implementation uses MSB ordering for input values. An optimization could be to use consistent ordering in both places, that would also open the possibility for an early termination in case of mismatches during evaluation, preventing the unnecessary computation of the entire tree [10, 64].

Additionally, the bit representation algorithm does not convert the numbers blindly, but verifies if the number can be represented with the specified sign within the given bits, using the `can_be_represented_in_bits` method.

6.2.4 Convert Methods

The `convert` helper module includes methods that perform conversion, particularly for bit and byte-level transformation. While several methods have been developed over the time of this thesis for similar purposes, the primary method that is currently utilized by Funshade is `convert_bytes_by_bits`, which converts a byte array into a signed integer number.

The implementation is based on the definition provided along with DCF [9]. The method takes an array of bytes and a specified number of bits to extract. At the start, it calculates the required number of bytes needed to represent the given bits with the help of `calculate_number_of_bytes` method from the previously discussed module. Then, it iterates over the byte array, combining the relevant bytes into a final result using SHIFT and OR operations. In the case, when the number of bits is not divisible by 8, the remaining bits are processed individually in a separate iteration, ensuring the final result is constructed accurately.

6.2.5 Scaling

Initially, these methods were developed within test modules to provide correct test data. However, as explained in Section 4.2, the values in the input embedding vectors have a defined cap. Additionally, the restrictions imposed by the number of bits also influences the possible range of values, making it essential to tailor the input values to align with the protocol requirements.

This module includes methods to scale normalized floating-point inputs, integer inputs, and vectors that already had a cap on them. For integer handling, both methods rely on a general `scale_input` function to perform the actual scaling, with their primary task being the calculation of the maximum value. The scaling process is based on the minimum value, meaning that if any array is constrained, for instance, within the range of $[-8; 8)$ and needs to be scaled

to fit $[-16;16)$, the boundaries can be divided without losing precision. While this division may seem straightforward in simple cases such as doubling the range, the situation becomes more complex when dealing with larger intervals, such as the range of `i32` values. Since 2^{32} can not be stored within 32 bits, even though dividing 2^{32} by 2^{16} produces an integer result, it cannot be accurately represented without extending the bit size. Calculating the ratio by dividing the maximum values as negative numbers, as -2^{32} is representable in 32 bits, the division will keep its accuracy.

6.3 Function Secret Sharing

Function Secret Sharing is a crucial component of Funshade. Having laid out the theoretical foundation in Section 2.7, this part will focus on the concrete implementation of FSS with the Rust programming language.

Starting from smaller units, Boyle et al. describes various tuples in the algorithm. In the implementation to enhance readability, these are sorted into small, dedicated data structures instead of tuples. This approach ensures that each part of the system is decoupled and easy to maintain.

The most straightforward structure is the `DcfKey`, which stores the initial seed required for tree generation, the correction words for each step, and a final correction term. The final correction term is stored separately since it is a single integer. This structure represents the result of the whole key generation process and is sent out to other parties. Notably, the `DcfKey` is a passive data structure: it merely stores values and takes no part in computation, as the actual generation and evaluation functions use it.

The DCF algorithm heavily relies on pseudo-random numbers generated from a new seed in each iteration. Since these seeds are stored as bytes and require dedicated operations (e.g., XOR), they are encapsulated in a separate structure, named `AesSeed`. This structure holds an array of 16 bytes. Even early in the implementation, it became clear that a seed of constant length may not suit every possible pseudo-random number generator (PRNG). However, Rust requires the array size to be known at compile time, making dynamic array sizing impossible during instantiation.

Two potential solutions can be considered. One approach would involve creating a trait that operates on vectors, taking the vector length as a parameter and implementing the methods for different PRNGs. While this provides flexibility, it limits the supported PRNGs. A better solution involves using the `arrayvec` [2] crate, which allows specifying the array length during runtime. Implementing this in the future would greatly enhance flexibility and usability.

As described by Boyle et al. [9], the $s^L||v^L||t^L||s^R||v^R||t^R \leftarrow G(s^{(i-1)})$ random tuples, generated in each iteration, correspond to the `DcfNodes` structure. From each seed, two nodes are generated—one for the left branch and one for the right—each consisting of a seed, an integer value, and a boolean. A private constructor and a method generate these node pairs taking the PRNG instance and the group as parameters. These form the foundation for generating the correction words.

The correction words tuples are frequently used, making it reasonable to create a dedicated structure, `CorrectionWord`. Similar to `DcfKey`, its responsibility is limited to storing pre-computed values, which are exclusively used by the evaluation algorithm.

Converting tuples to structured objects made it easier to follow the provided pseudocode from the paper. However, small changes were unavoidable due to the nature and variability of the features provided by programming languages. For example, loops were adjusted to start at index `0`, and the list of correction words was pre-allocated with a defined capacity. Rust's overflow behaviour, which causes a panic on overflow in development build, had to be accounted for. To avoid panics wrapping methods are used instead of regular operators (e.g., $+, -, \cdot$). In the release build, Rust changes them back for performance reasons, so wrapping methods like `wrapping_add` were used to ensure execution during development, testing and debugging.

```
let a: i32 = 2022;
let b: i32 = 2023;

let unwrapped_result = a + b;
let wrapped_result = a.wrapping_add(b); // In release build this = a + b
```

Listing 6.5: After optimizations done by the compiler, the two operations will be the same.

An interesting challenge arose when managing the randomized initial seeds used at the start of the key generation process. These seeds need to persist until the function ends, but they must also be used in the first iteration of the generation process. In Java, references could be stored in variables, and updating the reference at the end of each iteration would retain the initial seed values. However, Rust's ownership model does not allow this. The simplest solution would be to handle the first iteration separately, but duplicating code is undesirable. An option more alike to Java, is to store a reference and update it in each iteration, though Rust's borrow checker complicates this approach by ending the variable's life at the loop's end. Consequently, the program panics during the next node generation because the seed variable does not have any value.

```
let s: AesSeed = AesSeed::new_random();
let mut s_i = s;
for i in 0..10 {
  // Perform some calculation
  s_i = // some new value
}
let key = DcfKey::new(s, cws, last);
```

Listing 6.6: The last line throws an error because the ownership of the value is transferred in line 2.

To solve this, the initial seeds are cloned in the first iteration to avoid ownership transfer, allowing them to persist throughout the function. Additionally, the variables are made mutable to enable changing the values in each iteration.

Unlike seed management, correction words are not in a disadvantage from Rust's ownership model. The `push` method for lists transfers ownership of the value, ensuring that the values persist beyond the end of each iteration.

A private function, `gen_dcf_with_seed`, was implemented to reuse the same seeds for testing and debugging purposes. This function ensures consistent behaviour across different test cases. The rest of the implementation is pretty similar to what's written in the paper.

The interval containment module builds on the more complex DCF module. The `ICKey` structure is responsible for encapsulating the necessary data, which includes the DCF key and a correction term. One notable deviation from the C implementation is the `compare_as_unsigned` method. While casting signed values

to unsigned is straightforward in C, Rust requires additional handling to avoid panics. As a result, a custom comparison function was implemented to perform this task safely without type casting or using a bigger type.

6.4 Funshade

This section outlines the concrete steps taken to implement Funshade in Rust. The implementation relies on several components, such as the Function Secret Sharing module, helper data structures and functions. While these components have been discussed in prior sections, the focus here is on how they are utilized by the Funshade module.

As discussed in the theoretical part (cf. Chapter 4), the Funshade protocol is divided into distinct phases: setup, secret sharing (hiding the input data as shares), evaluation, and result computation. Each of these roles involves specific data structures and processes.

As with the FSS module, the outcome of the setup phase is a key, represented here as `FunshadeKey`. This key is also a passive data structure that stores precomputed values, which are later used during the evaluation phase. Importantly, this key encapsulates the necessary information generated during the setup without performing any active operations on it.

The `DeltaShare` structure is designed to hold the additive shares to hide the input data from each other. These shares are randomly generated during the setup phase, removing the need for a dedicated traditional "constructor" or the requirement to prevent unauthorized instantiation. One possible optimization would be to send a single array — the sum of the two shares — to the input-holding parties. This approach would reduce the communication cost by half. The existence of this structure would become debatable as it would hold only one variable. By deleting this structure, it would transfer the responsibility of generating the two shares to a different — and a new — structure, such as `BeaverTriples`, since the additive shares are the same elements the triples hold.

The `DistanceMetric` data type contains two values: a function representing the metric that needs to be evaluated locally and the scalar factor for computing the scalar product. This structure allows the user to select and apply the distance metric of their choice. While the type needs to be public for the user to construct an instance, the fields themselves should not be modifiable by external components. Therefore, to prevent the fields to inherit the protection levels, they are adjusted to restrict field access.

The somewhat unconventional type signature for the `f_local` field is a necessity to use and store closures, which implement the `Fn` trait. These closures are wrapped in a `Box` because their size is not known at compile time.

```
1 pub struct DistanceMetric {
2   pub(crate) f_local: Box<dyn Fn(&[TYPE]) -> TYPE>,
3   pub(crate) f_cp: TYPE
4 }
```

Listing 6.7: `DistanceMetric` data structure.

Ibarrondo et al. [29] already provided the reformulation of distance metrics which are pre-implemented in the project. To provide built-in support for these, the project includes an enum named `BuiltInDistanceMetric`. The enum also provides the instantiation of the pre-defined distance metrics in a dedicated method. This uses `match` arms to generate the instances efficiently, minimizing

code duplication and visualizing similarities between metrics (e.g., the Squared Euclidean and Hamming Distance, as shown in [29]).

The Funshade implementation closely follows the protocol definitions presented in [29], but with some necessary adjustments to adapt the methods for Rust and to ensure efficiency.

The `setup` method, used by R_{setup}, is responsible for preparing the pre-computed values for the protocol. The essence of the implementation mirrors the protocol steps: generating random values, performing basic operations, and constructing the keys. However, the method's header deviates slightly from the paper, as it takes a `FunshadeSettings` instance to encapsulate the required parameters except θ.

The input handler invokes the `share` method to evaluate the distance metric locally and hide their input vectors. The method signature has been extended to include both a `FunshadeSetting` and a `DistanceMetric` instance, ensuring that the necessary configuration and function definitions are readily available.

The evaluation algorithm is divided into two methods [28]: `eval_scalar` and `eval_sign`. The former takes the triples from the `FunshadeKey`, data from the input handlers and a `DistanceMetric` instance to compute the additive share of the distance metric result. This value is exchanged between the parties and given to the `eval_sign` algorithm, in which it is used to evaluate the IC gate. Together, these two methods perform the core computations required for secure evaluation.

Finally, the `result` method is invoked by R_{result} to reconstruct the final output based on the results of the previous computations.

6.5 Party Structure

The Funshade implementation includes pre-implemented party structures designed to provide users with ready-to-use interfaces. These structures offer users a reliable foundation, allowing them to build on existing methods and integrate the Funshade protocol with minimal setup effort.

Each party is implemented in the form of a struct rather than as a trait. The structs are essential for storing party-specific data, such as keys, and they implement various traits to support the necessary functionality (cf. Figure 6.1). A significant limitation with traits is that they define behaviour only and cannot store data. Coming from a mostly object-oriented background and the fact that there's no traditional inheritance, this restriction caused a lot of problems not only in this case, but throughout the project.

Additionally, to avoid potential communication issues and enhance thread-safety, each party operates independently. Rather than a single static instance, each party maintains individual state and data for every connected party, preventing conflicts in a multi-party setup like the one illustrated in Figure 6.1.

The parties store the role-specific data in dedicated structures, except for a shared configuration data structure. Early in the development, the configuration information was handled by `ConfigData` struct to ensure that all parties possess the same parameters. However, this structure's role eventually overlapped with the `FunshadeSettings`, making it redundant in the end. The data, it stored, was queried from the instance and passed onto the Funshade methods.

The `SetupData` struct contains necessary setup-phase information, such as the θ value. This structure is exclusive to the third party. The other two participants

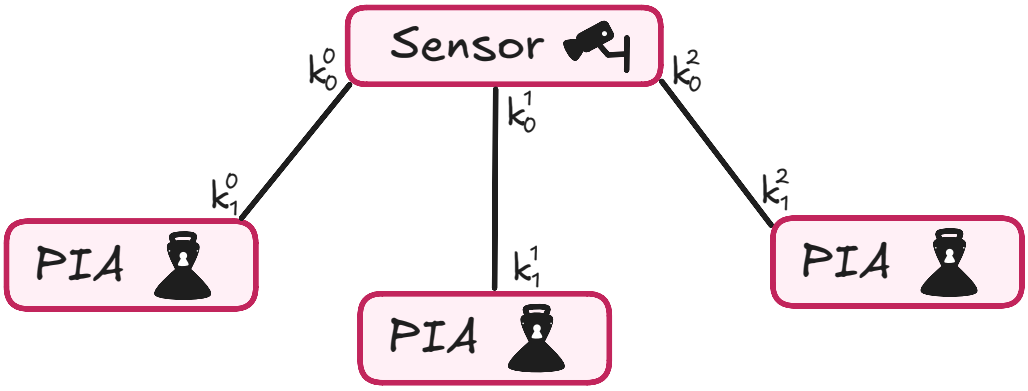


Figure 6.1: A Sensor connected to multiple PIA storing keys for each session.

have a dedicated structure for the input data and δ-shares, while in a different one they store data for the evaluation phase, such as keys, shares and the result from the `scalar_eval` function. The latter is needed to ensure every value is available for the IC gate evaluation, as when the two of them have delays — due to communication or performance — they need to wait for each other's value to progress further.

The parties not only store the necessary data, but also manage communication via references to `TcpStream` instances, which they use to forward messages. The messages are sent in a form of `Message` enums (see Listing 6.8), that contains the appropriate command, some of them even contain values. For example, `Message:Start` will tell the other party that it's time to start the protocol, while `Message:Key(key)` encloses the Funshade key to be sent.

In order to send messages, the `write_all` method can be used provided by the stream. This accepts a byte array, therefore to deliver the messages, they need to be converted to one. The `serde` library already provides a solution by using their two traits, the `Serialize` and `Deserialize` to convert them back and forth. However, because not all incoming data might belong to Funshade, the deserialization process is not initiated by the library, but the user. The project offers a handling function to manage the messages, which accepts the user's already deserialized instances.

```rust
#[derive(Serialize, Deserialize, Debug)]
pub enum Message {
  Init,
  Setup,
  Config(ConfigData),
  Abort,
  DeltaShare(DeltaShare),
  Key(FunshadeKey),
  Start,
  DShare(bool, Vec<i32>, i32),
  Eval,
  ZValue(i32),
  OValue(bool, i32)
}
```

Listing 6.8: `Message` enum.

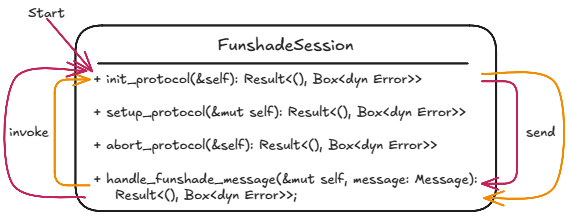
To organize functionality, Funshade defines two main traits: a public `FunshadeSession` and a private one, called `Party`. This distinction needed because Rust does not allow the method's protection level to differ from the

trait's. In this way, the software can still provide an interface to handle the processes and prevents the user to invoke internal methods.

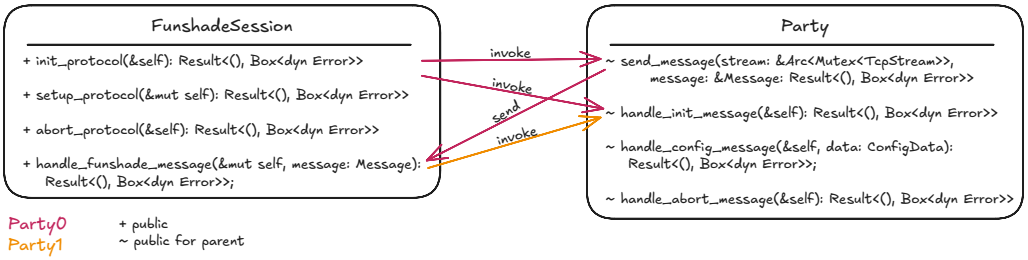
The `FunshadeSession` provides the user with structured methods to manage the protocol. It includes `init_protocol`, which initialize the protocol to agree on parameters, and `setup_protocol`, which begins the setup phase. If an error occurs during any method, `abort_protocol` can be used to terminate the process. Additionally, `handle_funshade_message` is available to interpret and manage incoming Funshade messages effectively.

The `Party` trait includes internal methods for sending a message to a given address, and handle specific requests, such as initialization.

In other terms, the `FunshadeSession` provides the interface for the user to tell the parties what to do and a command to interpret the message, but to actually perform the tasks, it uses private functions shown in Figure 6.2(b). The method invoked by the user does not perform the logic itself, rather it forwards the command to the appropriate element. For instance, if the called interface would also perform the logic itself then the `init_protocol` would send a message to the other party, by interpreting the message it would also invoke `init_protocol`, which would start a loop sending the same message back and forth (cf. Figure 6.2(a)).



(a) Without calling another unit for help the algorithm runs into an infinite loop.



(b) The `init_protocol` method called by the user will send the message and invoke the same method that the receiver to handle the initialization tasks.

Figure 6.2: Initialization request.

Since each party has a distinct role, Funshade also includes specialized traits acting as an interface, such as `SensorInput` and `PIAInput` shown in Listing 6.9. As the protocol can not be started until the sensor receives the data, it possesses a `start_matching` command to start the process, meanwhile the PIA has the opportunity to update its input and calculate shares in advance.

```rust
pub trait SensorInput<'a>{
  fn start_matching(&mut self, input: &'a Vec<TYPE> ) -> Result<(), Box<dyn Error
    >>;
}

pub trait PiaInput<'a> {
  fn change_input(&mut self, input_vec: Vec<&'a Vec<TYPE>>);
  fn calculate_shares(&mut self) -> Result<(), Box<dyn Error>>;
}
```

Listing 6.9: Interface of Sensor and PIA input roles.

To perform role-specific tasks, the parties needs to implement the specific traits, such as the trait `Compute` for the evaluator parties. It contains methods for managing the state of the struct and to perform the appropriate computations, such as `eval`. This role-oriented structuring enables focused, clean code that reflects each party's responsibilities within the protocol.

It is worth noting, that the naming of these parties is somewhat specific to the Digidow project [41], especially PIA, which is not very descriptive in a broader context. The name has been retained here for clarity and consistency with the thesis and for other team members. However, as development progresses, these names will likely change. In future versions, the party structure is expected to be refactored and undergo significant changes, additionally choosing more expressive and appropriate names to better reflect the roles each party represents in the protocol.

The software was developed early in the learning curve, and with hindsight, there are aspects that could be significantly improved. Several elements would benefit from refinement, and specific recommendations for these improvements are detailed in Chapter 8.

6.6 How to Use this Library?

The project is designed as a library to support integration into distributed systems, such as the Digidow project [41], facilitating secure multiparty computation. The goal is to provide a flexible yet efficient implementation that can be adapted to a range of scenarios where secure data processing is required.

Currently, the library provides several pre-implemented parties tailored for the third-party scenario, where a neutral entity helps facilitate the computation. These include `ThirdPartySetup`, responsible for distributing the keys and shares, `ThirdPartyPIA`, the party who knows their input data beforehand, and `ThirdPartySensor`, which activates the protocol when live data becomes available.

As discussed in the previous section, these parties are implemented as structures that hold necessary data, such as communication streams used to message other parties. The user's task is to instantiate these structures in their project (see Figure 6.3). This makes it simple to integrate the Funshade protocol into larger systems without needing to understand its internal complexity.

Each of these objects offers various functionalities (see Figure 6.4), including:

- Protocol initialization (`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 parameters. This is an optional, but not mandatory step.
- Protocol setup (`setup_protocol`): Perform the setup phase of the protocol.

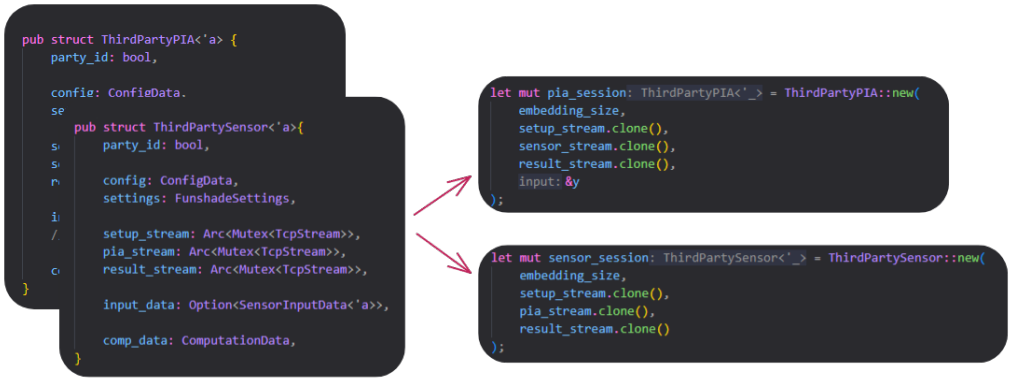


Figure 6.3: `ThirdPartyPIA` and `ThirdPartySensor` structs and instantiation.

- Abort protocol (`abort_protocol`): In cases where the computation cannot continue, or an error is detected, the `abort_protocol` method can be invoked to halt 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's 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 the `change_input` method. Additionally, the PIA can use the `calculate_shares` method to pre-compute its shares before the protocol begins, reducing the overhead during the online phase.

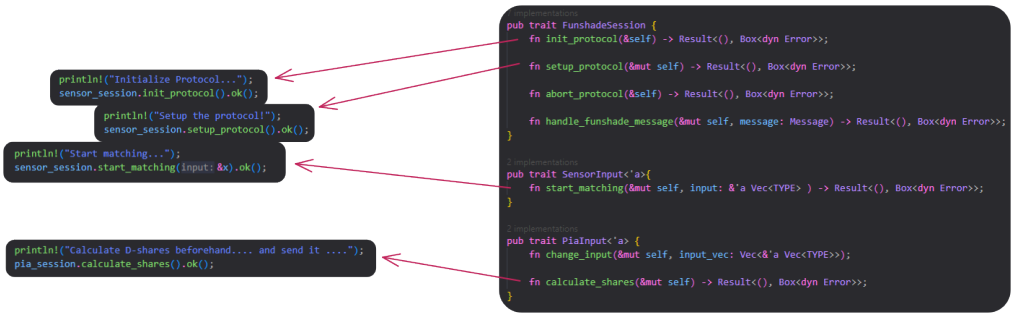


Figure 6.4: Public trait acting as an interface mapped to method calls.

Chapter 7

Evaluation

This chapter covers a successful and a negative match runs with concrete values, run-time measurements over multiple randomized inputs, and a brief summary of the unit tests implemented for this thesis. The test data are generated randomly, but for concrete cases the Labeled Faces in the Wild [20] dataset is used.

The tests and benchmarks are executed on an 11th Gen Intel® Core™ i5-1135G7 × 8 processor with 8GB of memory, using Ubuntu 24.04.1 LTS 64-bit operation system.

7.1 Positive Test Run

In a successful match run, the implementation correctly identifies a user by comparing the provided biometric data (x) with the stored reference (y). Figure 7.1 shows the matching faces that the test embeddings are based on.

Figure 7.1: Biometric data for matching test case.

- Bits: 32 bits
- Distance metric: Scalar Product
- Input Values: x and y are vectors with 512 elements.
- Output: Match successful (1).

In this case, the input and stored reference match, demonstrating that the implementation validates users:

```
running 1 test
test funshade::simple_funshade_test::funshade_positive_case ... ok

successes:

---- funshade::simple_funshade_test::funshade_positive_case stdout ----
Number of bits used: 32 bits
The selected distance metric is Scalar Product
The length of the embeddings is 512 and the maximum value of each element is
  2896
The treshold is 0.81 which converted to the integer range is 6793321
The expected result of the distance metric evaluation is 6866520
Therefore the expected result is z >= theta: true
The keys are generated and correct!
The input shares are generated and correct!
The masked result of the distance metric evaluation is: 621409186
The result is 1671688461+-1671688460=true


successes:
funshade::simple_funshade_test::funshade_positive_case

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 175 filtered out;
  finished in 0.01s
```

Listing 7.1: A positive test run of Funshade.

7.2 Negative Test Run

In an unsuccessful match run, the implementation recognizes that the person is not authorized to access the requested resource, by comparing the provided biometric data `x` with the stored reference `y`. Figure 7.2 shows the unmatching faces that the test embeddings are based on.

Figure 7.2: Biometric data for unmatching test case.

- Bits: 32 bits
- Distance metric: Scalar Product
- Input Values: `x` and `y` are arrays with 512 elements.
- Output: Match successful (`0`).

In this case, the input and stored reference does not match, demonstrating that the implementation determines correctly that the person is not authorized:

```
running 1 test
test funshade::simple_funshade_test::funshade_negative_case ... ok

successes:

---- funshade::simple_funshade_test::funshade_negative_case stdout ----
Number of bits used: 32 bits
The selected distance metric is Scalar Product
The length of the embeddings is 512 and the maximum value of each element is
  2896
The treshold is 0.81 which converted to the integer range is 6793321
The expected result of the distance metric evaluation is -188360
Therefore the expected result is z >= theta: false
The keys are generated and correct!
The input shares are generated and correct!
The masked result of the distance metric evaluation is: 61989206
The result is -287013062+287013062=false


successes:
funshade::simple_funshade_test::funshade_negative_case

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 175 filtered out;
  finished in 0.00s
```

Listing 7.2: A negative test run of Funshade.

7.3 Performance

For performance measurement the criterion [17] crate is used. Criterion is a micro-benchmarking library used for measuring performance and detecting improvements.

At the start of the benchmark, configurations are defined for the measurement phase. The `iter_batched` function of Criterion allows setup operations to be performed before each iteration. In the form of a closure, the generation of the input vectors are occurs without getting them involved in the measurement. The output is passed as a parameter for the next closure that defines the actual commands to be measured. The average run-time of the protocol without taking communication costs into account is around $120\mu s$ as visualized in Figure 7.3.

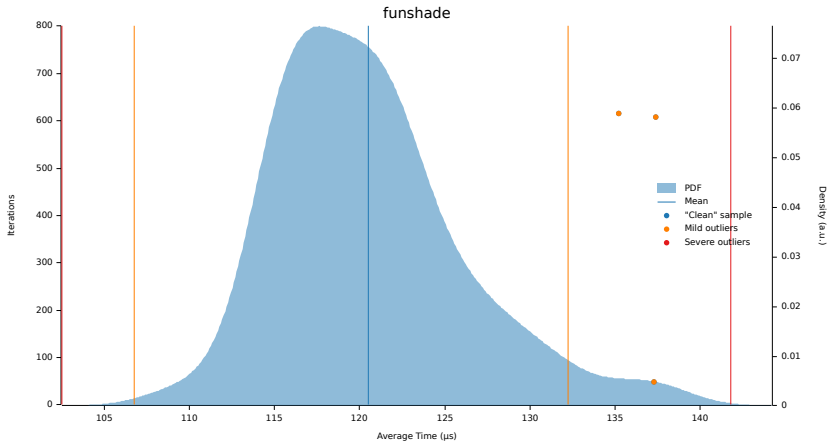


Figure 7.3: The average time per iteration for this benchmark.

7.4 Unit Tests

Along the implementation, various unit tests are also created for each module and method:

```
test fss::ic::ic_tests::ic_random_n ... ok
test funshade::simple_funshade_test::arbitrary_bit_single_funshade ... ok
test funshade::simple_funshade_test::arbitrary_theta_funshade ... ok

test result: ok. 174 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
  finished in 14.53s
```

Listing 7.3: Test report.

Unit tests cover various functions, including bit representation, smaller bit-related calculations, scaling, and convert methods. Similar tests are created for `dcf`, `ic`, and `funshade` modules to validate the protocols. Each test is organized in a separate `functionality_name_test` submodule, to group the related ones together. For instance, Listing 7.4 shows a part of the unit tests implemented to validate bit representation methods:

```
#[cfg(test)]
mod bit_rep_tests {
  ...

  #[test]
  fn five_bit_len_representation_msb() {
    let length= 4;
    let num = 5;

    let actual: Vec<bool> = bit_representation_with_len(num, length, &BitOrder::
    Msb, &Sign::Signed);

    let mut expected: Vec<bool> = Vec::new();
    expected.push(false);
    expected.push(true);
    expected.push(false);
    expected.push(true);

    assert_eq!(actual, expected);
  }
```

```
20
21    #[test]
22    fn five_bit_len_representation_lsb() {
23      let length= 4;
24      let num = 5;
25
26      let actual: Vec<bool> = bit_representation_with_len(num, length, &BitOrder::
        Lsb, &Sign::Signed);
27
28      let mut expected: Vec<bool> = Vec::new();
29      expected.push(false);
30      expected.push(true);
31      expected.push(false);
32      expected.push(true);
33      expected.reverse();
34
35      assert_eq!(actual, expected);
36    }
37
38    #[test]
39    fn big_number_len_representation() { ... }
40
41    #[test]
42    fn negative_number_len_representation_msb() { ... }
43
44    #[test]
45    fn negative_number_len_representation_lsb() { ... }
46
47    #[test]
48    fn large_number_len_representation() { ... }
49
50    ...
51  }
```

Listing 7.4: The bit representation unit tests are organized in a small test module.

One of these tests for `funshade` module, presented in Listing 7.5, happens in the following way (the execution order is the same as for Listing 7.1 and Listing 7.2):

1. Generate the necessary data for the protocol, such as the threshold (θ), the number of bits used (n), the length of the input vectors (l), the distance metric used (`dm`) and the maximum value of the input elements (max_{el}).
2. Begin a loop from 0 until a specified number (e.g. 1000), determining the number of protocol executions.
3. At the start of each iteration, generate two new random input vectors within the valid interval. Following this operation, calculate the expected result based on the distance metric and the threshold.
4. Generate keys and delta-shares using the `setup` method.
5. Calling two `share` method on the input vectors and delta-shares, generates the shares of the masked inputs.
6. Perform the first part of the evaluation method using `eval_scalar`. The results are then used in the `eval_sign` to evaluate the IC gate and acquire the result the two party would have in the end.
7. Summing both values reveals the protocols' outcome which can be compared with the expected value.
8. Jump to the next iteration.

```rust
#[test]
fn funshade_more_embedding() {
  let mut rng: ThreadRng = rand::thread_rng();
  let number_of_test_inputs = 1000;

  let n: usize = TYPE_BITS;
  let l: usize = 512;

  let settings: FunshadeSettings = FunshadeSettings::new(l, n);

  let max_el: f32 = calculate_max_el(&settings);

  let theta_real: f32 = 0.8;
  let theta = theta_real * max_el * max_el;
  let theta = theta.floor() as TYPE;

  let dm_type: BuiltInDistMetrics = BuiltInDistMetrics::Scalar;
  let dm: DistanceMetric = BuiltInDistMetrics::create(&dm_type);

  for _ in 0..number_of_test_inputs {

    let x_float: Vec<f32> = sample_biometric_template(&mut rng,  1, l).remove(0);
    let y_float: Vec<f32> = sample_biometric_template(&mut rng,  1, l).remove(0);

    let x: Vec<TYPE> = scale_normalized_input(&x_float, &settings);
    let y: Vec<TYPE> = scale_normalized_input(&y_float, &settings);

    let mut z: TYPE = 0;
    for i in 0..l {
      z = z.wrapping_add( x[i].wrapping_mul(y[i]) );
    }

    let (k0, k1, delta_x, delta_y): (FunshadeKey, FunshadeKey, DeltaShare,
    DeltaShare)
      = setup(&settings, theta);

    assert_key(&settings, &k0, &delta_x);
    assert_key(&settings, &k1, &delta_y);

    let (in_x0, in_x1) = share(&delta_x, &x, &dm, &settings);
    let (in_y0, in_y1) = share(&delta_y, &y, &dm, &settings);

    assert_share_result(&settings, &in_x0);
    assert_share_result(&settings, &in_x1);
    assert_share_result(&settings, &in_y0);
    assert_share_result(&settings, &in_y1);

    let z0: TYPE = eval_scalar(false, &k0,
      &in_x0, &in_y0, &dm, &settings);
    let z1: TYPE = eval_scalar(true, &k1,
      &in_x1, &in_y1, &dm, &settings);

    let o0: TYPE = eval_sign(false, &k0.key, z0, z1, &settings.g);
    let o1: TYPE = eval_sign(true, &k1.key, z0, z1, &settings.g);

    let expected: bool = z >= theta;
    let actual: bool = res(o0, o1, &settings.g);

    assert_eq!(actual, expected);
  }
}
```

Listing 7.5: Funsahde test generates embeddings with more elements in every iteration and executes the protocol.

7.5 Execution with Party Structures

For the purpose of testing the TCP communication between the parties, the `main` performs a session with random values. The program realizes the scenario using a third-party for setup and an additional one, who receives the results.

The program can be executed from command line with custom parameters for which the `clap` crate [14] was used (cf. Section 6.1). In the parameters it needs to be specified that which party's task should performed: `--sensor`, `pia`, `setup`, and `result`. The scenario can be selected with the `-m` flag, setting `1` for the third-party scenario and `3` for the 2PC.

The process happens as follows:

- Initiate the process with the Result party. This will wait for incoming connections.

```
    cargo run -- --result -m 1

```

Listing 7.6: Start the Result Party.

- Start the third, the setup party. It connects to the Result party and wait for incoming connections.

```
    cargo run -- --setup -m 1

```

Listing 7.7: Start the Setup Party.

- Launch the PIA that connects to the third and the result party and wait for incoming connection from the sensor.

```
    cargo run -- --pia -m 1

```

Listing 7.8: Start the PIA.

- Finally, start the sensor.

```
    cargo run -- --sensor -m 1

```

Listing 7.9: Start the Sensor.

- After all participants are present, the protocol starts automatically. The results are the followings:

```
    Listening on 127.0.0.1:6868!
    Setup Party Connected!
    PIA Connected!
    Sensor Connected!
    Creating Result Party...
    Waiting for o0 and o1...
    One received!
    Other received!
    o0: 1499305535
    o1: -1499305535
    result: 0 - false
    Expected result (z >= theta): false
    Done!
```

Listing 7.10: Print outs from Result Party.

```
Listening on 127.0.0.1:4848!
Connected to Result Party!
PIA connected!
Sensor connected!
Creating Setup Party...
Waiting for init request...
Received
Waiting for config data...
First config data received
Other config data received
Waiting for setup request...
Received
Done!
```

Listing 7.11: Print outs from Setup Party.

```
Listening on 127.0.0.1:3838!
Connected to Setup Party!
Connected to Result Party!
Sensor connected!
Creating PIA...
Waiting for init request...
Received
Waiting for config data....
First config data received
Other config data received
Waiting for key...
Key received
Waiting for Delta-share...
DeltaShare received
Calculate D-shares beforehand.... and send it ....
Waiting for Start message...
Received Start message
Waiting for Dshare
Received Dshare from Sensor
Waiting for sensor's z value...
Sensor's z value received
Done!
```

Listing 7.12: Print outs from PIA.

```
Connected to Setup Party!
Connected to PIA!
Connected to Result Party!
Creating Sensor ...
Initialize Protocol...
Waiting for config data...
Config data received
Other config data received
Setup the protocol!
Waiting for key....
Key received
Waiting for Delta-share
DeltaShare received
Waiting for dshare from PIA...
Dshare received
Start matching...
Waiting for PIA's z value...
Z received
Done!
```

Listing 7.13: Print outs from Sensor.

Chapter 8

Conclusion and Future Work

In conclusion, this thesis presents an in-depth explanation and implementation of Funshade. This work established the theoretical foundation, covering FSS, DCF and interval containment, along with other secret-sharing schemes. By leveraging Rust's memory safety and concurrency capabilities, Funshade was built to securely compare biometric data between parties without revealing sensitive information to any individual party. Several optimizations and design decisions—such as using structs for data storage and pre-configured traits for specific functionality—helped balance usability and flexibility with hiding the private parts from the user.

While this implementation of Funshade achieves its primary goals, numerous areas for future improvement have been identified during the creation of this thesis to enhance security, adaptability, and efficiency further. As Section 4.2 described, a major opportunity lies in refining the setup phase, where the parties collaboratively compute the outputs and to address potential attacks from compromised parties. Hopefully, this development will happen in the form of a collaboration with the original authors of the Funshade paper.

A minor addition would be to allow the user to choose their own PRNG or use a default one. For this purpose developing an AES-based PRNG using `aes` crate would be advantageous, as the current PRNG relies on a third-party crate, without any confirmation about how secure it is.

From the implementation perspective, parties should receive a more refined and flexible structure. This is planned to achieve by overcoming the difficulty of the separated data and behaviour aspects of Rust and pre-implement function in traits that handles data, but behaves the same in every situation. Taking advantage of the fact that Funshade has clearly separable phases (e.g., setup, share, eval), implementing a state machine would be an efficient addition.

Finally, a formal verification of the implementation and refining communication structure, whether through a fully user-controlled or library-managed approach, would increase Funshade's flexibility across various use cases. A custom writer could also make Funshade adaptable to diverse communication protocols, ensuring a broader application range without sacrificing reliability.

Bibliography

[1] 2024. aes-prng - crates.io: Rust Package Registry. en. (February 2024). Retrieved 10/21/2024 from https://crates.io/crates/aes-prng.

[2] 2024. arrayvec - crates.io: Rust Package Registry. en. (August 2024). Retrieved 10/24/2024 from https://crates.io/crates/arrayvec.

[3] Mauro Barni, Giulia Droandi, Riccardo Lazzeretti, and Tommaso Pignata. 2019. SEMBA: secure multi-biometric authentication. en. *IET Biometrics*, 8, 6, 411–421. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1049/iet-bmt.2018.5138. ISSN: 2047-4946. DOI: 10.1049/iet-bmt.2018.5138. Retrieved 11/11/2024 from https://onlinelibrary.wiley.com/doi/abs/10.1049/iet-bmt.2018.5138.

[4] Donald Beaver. 1992. Efficient Multiparty Protocols Using Circuit Randomization. en. In *Advances in Cryptology — CRYPTO '91*. Joan Feigenbaum, (Ed.) Springer, Berlin, Heidelberg, pp. 420–432. ISBN: 978-3-540-46766-3. DOI: 10.1007/3-540-46766-1_34.

[5] S.M. Bellovin and M. Merritt. 1992. Encrypted key exchange: password-based protocols secure against dictionary attacks. In *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*. (May 1992), pp. 72–84. DOI: 10.1109/RISP.1992.213269. Retrieved 10/14/2024 from https://ieeexplore.ieee.org/document/213269.

[6] J. Blandy, J. Orendorff, L. Tindall, and an O'Reilly Media Company Safari. 2021. *Programming Rust, 2nd Edition*. O'Reilly Media, Incorporated. https://books.google.at/books?id=BU1YzQEACAAJ.

[7] Abraham Bookstein, Vladimir A. Kulyukin, and Timo Raita. 2002. Generalized Hamming Distance. en. *Information Retrieval*, 5, 4, (October 2002), 353–375. ISSN: 1573-7659. DOI: 10.1023/A:1020499411651. Retrieved 10/03/2024 from https://doi.org/10.1023/A:1020499411651.

[8] Victor Boyko, Philip MacKenzie, and Sarvar Patel. 2000. Provably Secure Password-Authenticated Key Exchange Using Diffie-Hellman. en. In *Advances in Cryptology — EUROCRYPT 2000*. Bart Preneel, (Ed.) Springer, Berlin, Heidelberg, pp. 156–171. ISBN: 978-3-540-45539-4. DOI: 10.1007/3-540-45539-6_12.

[9] 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. Publication info: Preprint. MINOR revision. (2020). Retrieved 10/03/2024 from https://eprint.iacr.org/2020/1392.

[10] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2018. Function Secret Sharing: Improvements and Extensions. Publication info: Published elsewhere. Major revision. ACM CCS 2016. (2018). Retrieved 10/08/2024 from https://eprint.iacr.org/2018/707.

[11] 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* (*IH&MMSec '14*). Association for Computing Machinery, New York, NY, USA, (June 2014), pp. 187–198. ISBN: 978-1-4503-2647-6. DOI: 10.1145/2600918.2600922. Retrieved 10/30/2024 from https://dl.acm.org/doi/10.1145/2600918.2600922.

[12] Jung Hee Cheon, HeeWon Chung, Myungsun Kim, and Kang-Won Lee. 2016. Ghostshell: Secure Biometric Authentication using Integrity-based Homomorphic Evaluations. Publication info: Preprint. MINOR revision. (2016). Retrieved 11/17/2024 from https://eprint.iacr.org/2016/484.

[13] 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* (*ASIA CCS '14*). Association for Computing Machinery, New York, NY, USA, (June 2014), pp. 401–412. ISBN: 978-1-4503-2800-5. DOI: 10.1145/2590296.2590343. Retrieved 11/13/2024 from https://dl.acm.org/doi/10.1145/2590296.2590343.

[14] 2024. clap - crates.io: Rust Package Registry. en. (October 2024). Retrieved 10/21/2024 from https://crates.io/crates/clap.

[15] Ronald Cramer and Ivan Damgård. 2005. Multiparty Computation, an Introduction. en. In *Contemporary Cryptology*. Dario Catalano, Ronald Cramer, Giovanni Di Crescenzo, Ivan Darmgård, David Pointcheval, and Tsuyoshi Takagi, (Eds.) Birkhäuser, Basel, pp. 41–87. ISBN: 978-3-7643-7394-8. DOI: 10.1007/3-7643-7394-6_2. Retrieved 10/03/2024 from https://doi.org/10.1007/3-7643-7394-6_2.

[16] Ronald Cramer, Ivan Bjerre Damgård, and Jesper Buus Nielsen. 2015. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, Cambridge. ISBN: 978-1-107-04305-3. DOI: 10.1017/CBO9781107337756. Retrieved 10/03/2024 from https://www.cambridge.org/core/books/secure-multiparty-computation-and-secret-sharing/4C2480B202905CE5370B2609F0C2A67A.

[17] 2023. criterion - crates.io: Rust Package Registry. en. (May 2023). Retrieved 11/12/2024 from https://crates.io/crates/criterion#features.

[18] Joan Daemen and Vincent Rijmen. 2002. *The Design of Rijndael*. Ueli Maurer, Ronald L. Rivest, Martin Abadi, Ross Anderson, Mihir Bellare, Oded Goldreich, Tatsuaki Okamoto, Paul Van Oorschot, Birgit Pfitzmann, Aviel D. Rubin, and Jacques Stern, (Eds.) *Information Security and Cryptography*. Springer, Berlin, Heidelberg. ISBN: 978-3-662-04722-4. DOI: 10.1007/978-3-662-04722-4. Retrieved 10/21/2024 from http://link.springer.com/10.1007/978-3-662-04722-4.

[19] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. 2012. Multiparty Computation from Somewhat Homomorphic Encryption. en. In *Advances in Cryptology – CRYPTO 2012*. Reihaneh Safavi-Naini and Ran Canetti, (Eds.) Springer, Berlin, Heidelberg, pp. 643–662. ISBN: 978-3-642-32009-5. DOI: 10.1007/978-3-642-32009-5_38.

[20] Gary B. Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. 2007. Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments. University of Massachusetts, Amherst. http://vis-www.cs.umass.edu/lfw/lfw.tgz, (October 2007). Retrieved 11/14/2024 from https://vis-www.cs.umass.edu/lfw/.

[21] Shweta Gaur, V. A. Shah, and Manish Thakker. 2012. Biometric recognition techniques: a review. *International journal of advanced research in electrical, electronics and instrumentation engineering*, 1, 4, 282–290. Retrieved 10/31/2024 from https://d1wqtxts1xzle7.cloudfront.net/81089243/biometric-recognition-techniques-a-review-libre.pdf?1645370827=&response-content-disposition=inline%3B+filename%3DBiometric_Recognition_Techniques_A_Revie.pdf&Expires=1730384760&Signature=Bza7~S67OpU0qUAmI1CTw5d8YTpHq1s03yVT6zzLhi79wN5X5n3dpKkRFRaIHwciNJWPxq3~sGAZX7ZHfcYrhLvy7a70PPUc7q9v9cp61hd9ElG2n23FDt~tpoV0A6ewlVl4w6VPgzBXyvIEHZrQON~IyU0E-qQlj-dE9eXjXMckb3bXkRfdvuMnN-AmYqhzTa5GSJcwRm4iUxV-sHeVpKpA3XN0tJN-ZeXOrsA4N790dqTMdTDCfi8QTPMKzbvpttpIeNnzKjL6q4OiZ-ybuBvXScBS33Nv2rPQsNfnEX8Knl8~ZP~TpygyT6eUCPIVrT8x9GQQC1yqxFaWblqX-A___&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA.

[22] Craig Gentry. 2009. *A fully homomorphic encryption scheme*. phd. Stanford University, Stanford, CA, USA. AAI3382729 ISBN-13: 9781109444506.

[23] S. Goldwasser, S. Micali, and C. Rackoff. 1989. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18, 1, (February 1989), 186–208. ISSN: 0097-5397. DOI: 10.1137/0218012. Retrieved 11/04/2024 from https://doi.org/10.1137/0218012.

[24] Feng Hao and Paul C. van Oorschot. 2021. SoK: Password-Authenticated Key Exchange – Theory, Practice, Standardization and Real-World Lessons. Publication info: Published elsewhere. AsiaCCS 2022. (2021). Retrieved 10/15/2024 from https://eprint.iacr.org/2021/1492.

[25] Feng Hao and Peter Y. A. Ryan. 2011. Password Authenticated Key Exchange by Juggling. en. In *Security Protocols XVI*. Bruce Christianson, James A. Malcolm, Vashek Matyas, and Michael Roe, (Eds.) Springer, Berlin, Heidelberg, pp. 159–171. ISBN: 978-3-642-22137-8. DOI: 10.1007/978-3-642-22137-8_23.

[26] Philipp Hofer. 2024. *Enhancing Privacy-Preserving Biometric Authentication through Decentralization*. eng. PhD thesis. Linz. Retrieved 10/31/2024 from https://resolver.obvsg.at/urn:nbn:at:at-ubl:1-79283. Book Title: Enhancing Privacy-Preserving Biometric Authentication through Decentralization.

[27] IACR. 2021. Function Secret Sharing for Mixed-Mode and Fixed-Point Secure Computation. (October 2021). Retrieved 10/08/2024 from https://www.youtube.com/watch?v=22BfFkP_Hbk.

[28] Alberto Ibarrondo. 2024. ibarrond/funshade. original-date: 2023-01-19T20:00:44Z. (September 2024). Retrieved 10/22/2024 from https://github.com/ibarrond/funshade.

[29] Alberto Ibarrondo, Hervé Chabanne, and Melek Önen. 2022. Funshade: Function Secret Sharing for Two-Party Secure Thresholded Distance Evaluation. Publication info: Published elsewhere. Minor revision. PETS23. (2022). Retrieved 10/03/2024 from https://eprint.iacr.org/2022/1688.

[30] A.K. Jain, A. Ross, and S. Prabhakar. 2004. An introduction to biometric recognition. *IEEE Transactions on Circuits and Systems for Video Technology*, 14, 1, (January 2004), 4–20. Conference Name: IEEE Transactions on Circuits and Systems for Video Technology. ISSN: 1558-2205. DOI: 10.1109/TCSVT.2003.818349. Retrieved 10/11/2024 from https://ieeexplore.ieee.org/abstract/document/1262027.

[31] Cagatay Karabat, Mehmet Sabir Kiraz, Hakan Erdogan, and Erkay Savas. 2015. THRIVE: threshold homomorphic encryption based secure and privacy preserving biometric verification system. en. *EURASIP Journal on Advances in Signal Processing*, 2015, 1, (August 2015), 71. ISSN: 1687-6180. DOI: 10.1186/s13634-015-0255-5. Retrieved 11/17/2024 from https://doi.org/10.1186/s13634-015-0255-5.

[32] Sándor Király. 2020. Szolgáltatás-orientált programozás. Hungarian. Eger, Eszterházy Károly University, (2020).

[33] S. Klabnik and C. Nichols. 2024. The Rust Programming Language - The Rust Programming Language. (2024). Retrieved 10/28/2024 from https://doc.rust-lang.org/book/title-page.html.

[34] S. Krenn and T. Lorünser. 2023. *An Introduction to Secret Sharing: A Systematic Overview and Guide for Protocol Selection*. *SpringerBriefs in Information Security and Cryptography*. Springer International Publishing. ISBN: 978-3-031-28161-7. https://books.google.at/books?id=RRi2EAAAQBAJ.

[35] Joohee Lee, Dongwoo Kim, Duhyeong Kim, Yongsoo Song, Junbum Shin, and Jung Hee Cheon. 2018. Instant Privacy-Preserving Biometric Authentication for Hamming Distance. Publication info: Preprint. MINOR revision. (2018). Retrieved 11/11/2024 from https://eprint.iacr.org/2018/1214.

[36] Qiongxiu Li, Ignacio Cascudo, and Mads Graesbøll Christensen. 2019. Privacy-Preserving Distributed Average Consensus based on Additive Secret Sharing. In *2019 27th European Signal Processing Conference (EUSIPCO)*. ISSN: 2076-1465. (September 2019), pp. 1–5. DOI: 10.23919/EUSIPCO.2019.8902577. Retrieved 10/03/2024 from https://ieeexplore.ieee.org/abstract/document/8902577.

[37] Kálmán Liptai. 2023. Kriptográfia. Hungarian. Lecture. Eger, Eszterházy Károly University, (2023). Retrieved 10/16/2024 from http://liptai.ektf.hu/uploads/2011/12/Kriptografia.pdf.

[38] Elena Almaraz Luengo. 2022. A brief and understandable guide to pseudo-random number generators and specific models for security. *Statistics Surveys*, 16, none, (January 2022), 137–181. Publisher: Amer. Statist. Assoc., the Bernoulli Soc., the Inst. Math. Statist., and the Statist. Soc. Canada. ISSN: 1935-7516. DOI: 10.1214/22-SS136. Retrieved 10/21/2024 from https://projecteuclid.org/journals/statistics-surveys/volume-16/issue-none/A-brief-and-understandable-guide-to-pseudo-random-number-generators/10.1214/22-SS136.full.

[39] Ying Luo, Sen-ching Cheung, and Shuiming Ye. 2009. Anonymous Biometric Access Control based on homomorphic encryption. In (June 2009), pp. 1046–1049. DOI: 10.1109/ICME.2009.5202677.

[40] T. Soni Madhulatha. 2012. An Overview on Clustering Methods. arXiv:1205.1117 [cs]. (May 2012). DOI: 10.48550/arXiv.1205.1117. Retrieved 10/03/2024 from http://arxiv.org/abs/1205.1117.

[41] René Mayrhofer, Michael Roland, Tobias Höller, and Mario Lins. 2024. An Architecture for Distributed Digital Identities in the Physical World.

[42] René Mayrhofer, Michael Roland, Tobias Höller, and Mario Lins. [n. d.] Digidow. en-us. (). Retrieved 10/03/2024 from https://www.digidow.eu/.

[43] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. 2020. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, (May 2020). ISBN: 978-0-429-46633-5. DOI: 10.1201/9780429466335.

[44] CPP Nelson Joseph. 2013. Chapter 12 - Biometrics Characteristics. In *Effective Physical Security (Fourth Edition)*. Lawrence J. Fennelly, (Ed.) Butterworth-Heinemann, (January 2013), pp. 255–256. ISBN: 978-0-12-415892-4. DOI: 10.1016/B978-0-12-415892-4.00012-2. Retrieved 10/03/2024 from https://www.sciencedirect.com/science/article/pii/B9780124158924000122.

[45] 2024. num-traits - crates.io: Rust Package Registry. en. (May 2024). Retrieved 10/22/2024 from https://crates.io/crates/num-traits.

[46] European Parliament and Council of the European Union. 2016. Regulation - 2016/679 - EN - gdpr - EUR-Lex. en. Doc ID: 32016R0679 Doc Sector: 3 Doc Title: Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) (Text with EEA relevance) Doc Type: R Usr_lan: en. (May 2016). Retrieved 10/15/2024 from https://eur-lex.europa.eu/eli/reg/2016/679/oj.

[47] Christof Paar, Jan Pelzl, and Tim Güneysu. 2024. *Understanding Cryptography: From Established Symmetric and Asymmetric Ciphers to Post-Quantum Algorithms*. en. Springer, Berlin, Heidelberg. ISBN: 978-3-662-69007-9. DOI: 10.1007/978-3-662-69007-9. Retrieved 11/04/2024 from https://link.springer.com/10.1007/978-3-662-69007-9.

[48] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2020. ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. Publication info: Published elsewhere. Major revision. 30th USENIX Security Symposium (USENIX Security '21). (2020). Retrieved 10/03/2024 from https://eprint.iacr.org/2020/1225.

[49] Privacy Enhancing Technologies Symposium. 2023. [5A] Funshade: Function Secret Sharing for Two-Party Secure Thresholded Distance Evaluation. (November 2023). Retrieved 10/11/2024 from https://www.youtube.com/watch?v=eFJPZMxzdpQ.

[50] Pille Pullonen. 2013. Actively Secure Two-Party Computation: Efficient Beaver Triple Generation. In Retrieved 10/10/2024 from https://www.semanticscholar.org/paper/Actively-Secure-Two-Party-Computation%3A-Efficient-Pullonen/4694fe38d28985cd36fab42d5f22a3e6f8e67336.

[51] Michael O. Rabin. 2005. How To Exchange Secrets with Oblivious Transfer. Publication info: Published elsewhere. Harvard University Technical Report 81. (2005). Retrieved 11/01/2024 from https://eprint.iacr.org/2005/187.

[52] 2022. rand - crates.io: Rust Package Registry. en. (February 2022). Retrieved 10/21/2024 from https://crates.io/crates/rand.

[53] R. L. Rivest, A. Shamir, and L. Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21, 2, (February 1978), 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. Retrieved 10/16/2024 from https://dl.acm.org/doi/10.1145/359340.359342.

[54] Ronald L. Rivest and M. Dertouzos. 1978. ON DATA BANKS AND PRIVACY HOMOMORPHISMS. In Retrieved 11/01/2024 from https://www.semanticscholar.org/paper/ON-DATA-BANKS-AND-PRIVACY-HOMOMORPHISMS-Rivest-Dertouzos/c365f01d330b2211e74069120e88cff37eacbcf5.

[55] [n. d.] Rust Programming Language. en-US. (). Retrieved 10/28/2024 from https://www.rust-lang.org/.

[56] 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, (January 2022), 291–316. DOI: 10.2478/popets-2022-0015.

[57] 2024. serde - crates.io: Rust Package Registry. en. (September 2024). Retrieved 10/21/2024 from https://crates.io/crates/serde.

[58] 2024. serde_json - crates.io: Rust Package Registry. en. (October 2024). Retrieved 10/21/2024 from https://crates.io/crates/serde_json.

[59] S. Singh. 2000. *The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography*. Knopf Doubleday Publishing Group. ISBN: 978-0-385-49532-5. https://books.google.at/books?id=skt7TrLK5uYC.

[60] Juan Luis Suárez-Díaz, Salvador García, and Francisco Herrera. 2020. A Tutorial on Distance Metric Learning: Mathematical Foundations, Algorithms, Experimental Analysis, Prospects and Challenges (with Appendices on Mathematical Background and Detailed Algorithms Explanation). arXiv:1812.05944. (August 2020). DOI: 10.48550/arXiv.1812.05944. Retrieved 10/11/2024 from http://arxiv.org/abs/1812.05944.

[61] Kenta Takahashi, Takahiro Matsuda, Takao Murakami, Goichiro Hanaoka, and Masakatsu Nishigaki. 2019. Signature schemes with a fuzzy private key. en. *International Journal of Information Security*, 18, 5, (October 2019), 581–617. ISSN: 1615-5270. DOI: 10.1007/s10207-019-00428-z. Retrieved 11/11/2024 from https://doi.org/10.1007/s10207-019-00428-z.

[62] National Institute of Standards Technology (NIST), Morris J. Dworkin, Elaine Barker, James R. Nechvatal, James Foti, Lawrence E. Bassham, E. Roback, and James F. Dray Jr. 2001. Advanced Encryption Standard (AES). en. *NIST*, (November 2001). Last Modified: 2024-07-25T12:07-04:00 Publisher: National Institute of Standards and Technology (NIST), Morris J. Dworkin, Elaine Barker, James R. Nechvatal, James Foti, Lawrence E. Bassham, E. Roback, James F. Dray Jr. Retrieved 10/21/2024 from https://www.nist.gov/publications/advanced-encryption-standard-aes.

[63] The BIU Research Center on Applied Cryptography and Cyber Security. 2022. FSS Part 1 - Elette Boyle. (January 2022). Retrieved 10/03/2024 from https://www.youtube.com/watch?v=fAXlOOs2t88.

[64] The BIU Research Center on Applied Cryptography and Cyber Security. 2022. FSS Part 2 - Elette Boyle. (January 2022). Retrieved 10/03/2024 from https://www.youtube.com/watch?v=Zm-MUVve2_w.

[65] USENIX. 2021. USENIX Security '21 - ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. (September 2021). Retrieved 10/03/2024 from https://www.youtube.com/watch?v=X7IKSQyNEto.

[66] Maarten van Steen and Andrew S. Tanenbaum. 2016. A brief introduction to distributed systems. en. *Computing*, 98, 10, (October 2016), 967–1009. ISSN: 1436-5057. DOI: 10.1007/s00607-016-0508-7. Retrieved 10/03/2024 from https://doi.org/10.1007/s00607-016-0508-7.

[67] A.C. Weaver. 2006. Biometric authentication. *Computer*, 39, 2, (February 2006), 96–97. Conference Name: Computer. ISSN: 1558-0814. DOI: 10.1109/MC.2006.47. Retrieved 10/03/2024 from https://ieeexplore.ieee.org/abstract/document/1597098.

[68] Andrew C. Yao. 1982. Protocols for secure computations. English. In ISSN: 0272-5428. IEEE Computer Society, (November 1982), pp. 160–164. DOI: 10.1109/SFCS.1982.88. Retrieved 10/03/2024 from https://www.computer.org/csdl/proceedings-article/focs/1982/542800160/12OmNyUnEJP.

[69] Kai Zhou and Jian Ren. 2018. PassBio: Privacy-Preserving User-Centric Biometric Authentication. *IEEE Transactions on Information Forensics and Security*, 13, 12, (December 2018), 3050–3063. Conference Name: IEEE Transactions on Information Forensics and Security. ISSN: 1556-6021. DOI: 10.1109/TIFS.2018.2838540. Retrieved 11/13/2024 from https://ieeexplore.ieee.org/abstract/document/8361432.

Appendix A

Code Reachability

The description aligns with the implementation that is attached to this PDF and can be downloaded by clicking on the following icon:  . Alternatively it can be found in the GitLab repository: https://git.ins.jku.at/proj/digidow/funshade-rust.