GhostPeerShare

Jeffrey Murray Jr

A project
submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2024

Reading Committee:
Brent Lagesse, Chair
Geethapriya Thamilarasu
Yang Peng

Program Authorized to Offer Degree:
Cybersecurity Engineering

University of Washington

Abstract

GhostPeerShare

Jeffrey Murray Jr

Chair of the Supervisory Committee:
Dr. Brent Lagesse
Computing & Software Systems

Fully Homomorphic Encryption (FHE) schemes allow computations over encrypted data without access to the decryption key. Microsoft's Simple Encryption Arithmetic Library (SEAL) provides a state-of-the-art C++ library that enables addition, subtraction, and multiplication on encrypted integers or real numbers. SEAL is difficult to implement on Android due to its outdated documentation. Furthermore, it requires expertise in C++, Kotlin, and Gradle. This project presents a natively compiled Dart plugin that abstracts the underlying C/C++ library. The FHE Library plugin enables developers to access SEAL's full functionality within other Dart plugins and Flutter applications. To evaluate the versatility of the plugin, we employ two implementations: 1) the Distance Measure Dart plugin and 2) the GhostPeerShare Flutter application. The Distance Measure plugin implements Kullback-Leibler Divergence, Bhattacharyya Coefficient, and Cramer Distance, along with their FHE counterparts. GhostPeerShare adapts the implementation from Proof of Presence Share (Pop-Share), a mobile application that can detect similar videos using FHE. GhostPeerShare achieved a precision score of 0.9614 and an F1 score of 0.9709 for identifying similar videos. Although video pre-processing was considerably slower, the average FHE computation of each distance measure matched that of Pop-Share. These results demonstrate that GhostPeerShare represents a significant advance in the accessibility of FHE within the mobile community.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

FULLY HOMOMORPHIC ENCRYPTION (FHE): A type of encryption that allow computations over encrypted data without access to the decryption key.

CHEON-KIM-KIM-SONG (CKKS) SCHEME: A fully homomorphic encryption scheme that supports approximate arithmetic on real numbers used for decimal-based computations.

SIMPLE ENCRYPTED ARITHMETIC LIBRARY (SEAL): An open-source library developed by Microsoft for performing fully homomorphic encryption arithmetic on encrypted data.

KULLBACK-LEIBLER DIVERGENCE (KLD): A measure of how one probability distribution diverges from another, ranges from 0 to positive infinity, where 0 represents identical probability distributions.

BHATTACHARYYA COEFFICIENT (BC): A similarity measure that quantifies the amount of overlap between two probability distributions, ranges from 0 to 1, where 1 represents identical probability distributions.

CRAMER'S DISTANCE (CD): A statistical measure used to quantify dissimilarity between two probability distributions, ranges from 0 to 1, where 0 represents identical probability distributions.

FLUTTER: An open-source, cross-platform framework by Google for building natively compiled applications for mobile, web, and desktop from a single codebase.

DART: A programming language optimized for building fast applications on any platform, used in the development of Flutter apps.

FFMPEG: A multimedia framework commonly used for video and audio processing, used here for video preprocessing in mobile applications.

OPENCV: An open-source computer vision and machine learning software library that provides tools for image and video processing.

GITHUB ACTIONS: A CI/CD tool that automates tasks such as building, testing, and deploying software, supporting the automation of plugin testing and deployment in this project.

Chapter 1
INTRODUCTION

This chapter outlines the problem, identifies the beneficiaries, and describes the project's contributions. In Section 1.1, we summarize the problem statement. In Section 1.2, we identify beneficiaries and major stakeholders within the Fully Homomorphic Encryption (FHE) domain. In Section 1.3, we present a modular interface that grants access to FHE functionality on a state-of-the-art software development platform.

1.1 Problem Statement

The Microsoft Simple Encrypted Arithmetic Library (SEAL) [1] requires significant prerequisite knowledge to compile and embed the binary within resource-constrained devices. These prerequisites have limited the accessibility of SEAL, posing a significant obstacle to the adoption within the mobile community.

1.2 Stakeholders

Microsoft pioneered the Fully Homomorphic Encryption (FHE) research and led the development of the Simple Encryption Arithmetic Library (SEAL) in 2018. SEAL established a foundation for efficient homomorphic encryption, which made it accessible to the open-source community of security researchers and developers. SEAL was among the first of many libraries within the homomorphic encryption community. Other dominant libraries included the Homomorphic Encryption Library (HElib) [2] and Privacy-Preserving Approximate Secure Computation for Encrypted Data (PALISADE) [3]. Collectively, the open-source community aimed to combine their efforts into the Open-Source Fully Homomorphic Encryption Library (OpenFHE). A flexible, community-driven project that was initially released in 2022.

Compared to SEAL, OpenFHE offers support for additional encryption schemes for evaluating Boolean circuits and arbitrary functions over larger plaintext spaces using lookup tables. However, OpenFHE is a relatively new framework with a growing community but requires further adoption in the research community to assess its proficiency. Currently, developers and security researchers are limited in creating FHE applications within the programming language constraints of C or C++, as a result, the mobile community must overcome significant hurdles to access this functionality.

Flutter, a state-of-the-art software development kit by Google captured 46% of the cross-platform application market [4]. This framework is built on top of Dart, a programming language recognized for its high performance and robust plugin system. Together, they provide compatibility with all major operating systems, specifically Android, Linux, macOS, iOS, and Windows. As a beneficiary of this work, the growing Flutter community may integrate Fully Homomorphic Encryption (FHE) into existing or new applications. This framework enables security researchers and developers to rapidly develop applications without the required prerequisite knowledge of the underlying C FHE libraries.

1.3 Contributions

The Pyfhel library [5] provides a native Python interface to SEAL cryptosystems, abstracting core SEAL functionalities into a more generic interface. While Pyfhel supports cross-platform compatibility on Windows, Linux, and macOS, it currently lacks support for mobile platforms, e.g. Android and iOS. This project adopts a similar approach, creating a modular interface that supports multiple backend libraries.

This project contributes a modular, Dart plugin that abstracts and implements Microsoft Simple Encrypted Arithmetic Library. The abstraction design enables the integration and versioning of additional backend libraries. With the use of CMake configurations, each C library can be synchronized and recompiled as new versions become available, simplifying the update process. Automating these compilations reduces maintenance overhead, ensuring that the libraries remain up-to-date with minimal manual intervention.

To assess the library's performance and accuracy, we re-implement the Proof of Presence Share [6] methodology. This method calculates similarity scores between videos without exposing their content. We set up cameras from various angles to record simultaneously and compare the videos for similarity, as well as, perform comparisons against other scenes of the same duration. Using a supervised artificial neural network, we classify the distance measures of Kullback-Leibler Divergence [7], Bhattacharyya Coefficient [8], and Cramer Distance [9] for training and testing. The binary classification model is trained to differentiate between pairs of scenes, classifying them as either similar or different based on the unique representations generated by our system. By analyzing the model's accuracy through many predictions, we gain insight into the precision and reliability of our implementation in distinguishing matching scenes without revealing the videos' content.

Chapter 2
BACKGROUND

This chapter outlines the technologies and methods used in this research. In Section 2.1, we start with Flutter, an open-source framework for building cross-platform applications, and highlight its benefits for mobile development. In Section 2.2, we introduce the Simple Encrypted Arithmetic Library and explain the process of performing computations on encrypted data. In Section 2.3, we discuss the distance measures used for privacy-preserving video similarity analysis, focusing on Kullback-Leibler Divergence, Bhattacharyya Coefficient, and Cramer's Distance. We examine the properties and uses of each metric to show their importance in our work.

2.1 Flutter

Flutter is a modern, open-source software development kit designed to build stylish and efficient applications for mobile, web, and desktop within a single code base. Flutter is built on top of Dart; an object-oriented language with C-style syntax. Dart includes advanced features out of the box such as garbage collection for memory management, synchronous and asynchronous handlers, null safety, and compile-time data type checking. We selected Flutter for its low development cost and wide variety of plugins.

The plugin at the root of this project's design, the Foreign Function Interface allows the application to invoke functions from pre-compiled C libraries. Each target platform, specifically Windows, Linux, Android, iOS, and macOS requires different methods to compile, however, the fundamental method of invoking C functions and working with native data structures remains the same across platforms. The plugin is limited to transforming primitive data types, e.g. Booleans, Integers, and Characters, restricting the parameters of native C

functions to primitive data types. However, a notable workaround to this problem is passing complex objects by memory address, known as pass by reference, for the underlying C library to access the object stored in this stack.

To facilitate the networking between peer-to-peer Android devices, QuickShare [10] was developed by Samsung as a file-sharing utility application for nearby wireless devices. It leverages Bluetooth to discover nearby wireless devices and optionally uses WiFi to transfer large files between two nodes. QuickShare is accessible through a Flutter plugin that enables users to share data using their device's native sharing capabilities. This allows users to easily send content to other apps, such as social media platforms, messaging apps, or email.

Alternatives such as React Native (with JavaScript) or native development languages with Kotlin (Android) and Swift (iOS) involve different compilation methods. React Native is a popular framework for cross-platform development, comparable to Flutter in plugins and community activity; however, it lacks the native support of C interoperability and performance. React Native leverages Kotlin and Swift to configure and compile Android and iOS applications, respectively. This approach carries inherent challenges to developing and testing an application, often requiring additional dependencies to handle trivial maintenance. Flutter aims to streamline the process of building cross-platform applications by unifying the codebase, eliminating the need for separate code for each platform.

2.2 Simple Encrypted Arithmetic Library

To perform computations on encrypted data, this project utilizes Simple Encrypted Arithmetic Library (SEAL) [1], an open-source homomorphic encryption library developed by Microsoft. The library supports limited arithmetic including addition, subtraction, and multiplication to be performed directly on encrypted data without needing to decrypt it. We chose SEAL primarily for its maturity, prevalent adoption in research, and use in Proof of Presence Share [6].

SEAL supports three Fully Homomorphic Encryption (FHE) schemes: Brakerski-Fan-Vercauteren (BFV) [11], Brakerski-Gentry-Vaikuntanathan (BGV) [12], and Cheon-Kim-

Kim-Song (CKKS) [13]. BFV and BGV are integer-based FHE schemes that allow for computations on encrypted integers but differ in their ciphertext structure and noise management techniques. BFV encodes the message with the most significant bits of the ciphertext and accumulates more noise growth for each multiplication circuit. When too much noise accumulates, the ciphertext cannot be decrypted. For simpler computations, BFV offers better performance. BGV encodes the message in the least significant bits and manages noise through modulus switching. This difference in noise management makes BGV suitable for deeper computations with many sequential operations.

CKKS is specifically designed for scenarios where approximate arithmetic is acceptable, and this trade-off allows for more efficient computations on encrypted real numbers. This scheme is efficient in performing computations, by representing real numbers as complex numbers. The encoding allows for efficient arithmetic operations, specifically addition, multiplication, and subtraction. When decoding, the result is transformed into a meaningful representation of the true value, with some degree of error.

2.3 Distance Measures

To accurately measure the similarity between the two videos, this project implements the Kullback-Leibler Divergence, Bhattacharyya Coefficient, and Cramer's Distance. Pop-Share [6] established these metrics as an effective indicator for comparing video similarity. Unlike other probability distribution algorithms that involve complex calculations, these three metrics rely primarily on element-wise addition, subtraction, and multiplication of vectors. This simplicity is crucial for their integration with Fully Homomorphic Encryption, as it allows for efficient computation on a list of encrypted floating-point numbers.

Kullback-Leibler Divergence [7], represented as KLD in Equation 2.1, is a state-of-the-art measure of how one probability distribution diverges from a second, expected probability distribution. A limitation of KLD is that it is not symmetric, in that the divergence from distribution P to Q is not necessarily the same as from Q to P, which may complicate its interpretation. The divergence ranges from 0 to positive infinity, where a value closer to 0

indicates high similarity between the distributions.

$$KLD = \sum_i P(i) \log \left(\frac{P(i)}{Q(i)} \right) \tag{2.1}$$

Bhattacharyya Coefficient [8], represented as BC in Equation 2.2, is a measure that quantifies the amount of overlap between two probability distributions. The coefficient ranges from 0 to 1, where a value closer to 1 indicates high similarity between the distributions.

$$BC = \sum_x \sqrt{P(x)Q(x)} \tag{2.2}$$

Cramer's Distance [9], represented as CD in Equation 2.3, is a measure used to quantify the dissimilarity between two probability distributions. It is sensitive to small sample sizes, which can result in inaccuracies. The distance ranges from 0 to 1, where a value closer to 0 indicates high similarity between the distributions.

$$CD = \sqrt{\sum_i (P_i - Q_i)^2} \tag{2.3}$$

Chapter 3

RELATED WORK

This chapter reviews significant advances in Fully Homomorphic Encryption (FHE) and related methodologies relevant to this project. Section 3.1 begins with an overview of existing FHE libraries. In Section 3.2, we examine the Proof of Presence Share methodology, an innovative application of FHE to compute video similarity. In Section 3.3, we discuss alternative video similarity measures.

3.1 Fully Homomorphic Encryption Libraries

Open-Source Fully Homomorphic Encryption Library (OpenFHE) [14], is a state-of-the-art C++ library born from a merger of Privacy-Preserving Approximate Secure Computation for Encrypted Data (PALISADE) [3], Homomorphic Encryption library (HElib) [2], and HEAAN [13]. OpenFHE supports modern, Fully Homomorphic Encryption schemes and can be configured to use hardware acceleration.

From existing systems, OpenFHE adapted the modular design of PALISADE with the built-in support of Brakerski-Fan-Vercauteren (BFV) [11], Brakerski-Gentry-Vaikuntanathan (BGV) [12] and Cheon-Kim-Kim-Song (CKKS) [13]. From HElib, an efficient homomorphic encryption library primarily focused on BGV and CKKS schemes, pioneering research and early development in this domain. HEAAN was born from a new scheme, CKKS, to support arithmetic operations on approximate numbers, in contrast to existing libraries that could only support integer-based arithmetic.

Unlike SEAL, the OpenFHE library has an actively growing community. The documentation and development guides for OpenFHE are frequently updated and comprehensive. SEAL, however, is much more established within the research community. There have not

been any peer-reviewed publications directly comparing the performance of Microsoft SEAL and OpenFHE.

3.2 Proof of Presence Share

Proof of Presence Share (Pop-Share) [6] introduces a novel approach for accurately detecting similar video scenes with strong privacy guarantees. Pop-Share is based on the pre-processing methodology of Similarity of Simultaneous Observation (SSO) [15] to convert videos into byte-count arrays and adapts the distance measure algorithms from SSO to be compatible with Fully Homomorphic Encryption (FHE).

Developed by Wu and Lagesse, SSO was designed to detect streaming Wi-Fi cameras by comparing the probability distribution between network traffic and recording videos on the network. In addition, SSO contributes a computationally efficient way to transform video frames into probability distribution form to be compared by various statistical measures, including Jensen-Shannon Divergence (JSD), Kullback-Leibler Divergence, and Dynamic Time Warping (DTW) [16]. The pre-processing approach of SSO slices the video into one-second segments, computes the total number of the bytes for all frames in each segment, and normalizes the byte count array so that it summates to one. Pop-Share could not adapt JSD and DTW to be compatible with FHE, so Cramer Distance and the Bhattacharyya Coefficient were selected to handle encrypted distance measure computations.

Pop-Share depended on SEAL 3.3 to handle fully homomorphic operations. Once both videos were preprocessed, the transformation was irreversible. In addition, the arrays were encrypted and computed using the plaintext counterpart, resulting in less noise accumulation than when using only ciphertext. In a distributed peer-to-peer application, the initiator shares an encrypted video for the recipient to apply their plaintext array onto the untrustworthy ciphertext. The modified ciphertext was returned to the originator to be decrypted, and the summation of the decrypted array represents the corresponding similarity score. This was performed for each similarity score measure. A notable limitation of this approach is that Pop-Share's implementation was tightly coupled with a static version of SEAL, making

the results difficult to reproduce.

This method demonstrated that SSO generates a unique, one-way signature of videos that can be used to compare for similarity without access to the video's content. For this project, the results from Pop-Share serve as a baseline comparator for the modular Flutter re-implementation of the Pop-Share peer-to-peer application.

3.3 Video Similarity

Using content-based algorithms, Shan and Lee [17] presented a series of optimal mapping algorithms to detect similarity between video frames of unequal sizes. In addition, Wu, Zhuang, and Pan [18] present a similar shot graphing algorithm to identify similar videos within a large database of videos. An enhancement of Bhattacharyya was proposed by Loza, et al. [19] to generate a particle filter that would iterate over each sample to predict and resample based on the similarity. Toward Privacy-Preserving Photo Sharing [20] presents an innovative photo encoding algorithm that enables the comparison of photos without disclosing the content of the frames. The distance measure algorithms would need to be simplified to use basic arithmetic to be compatible with fully homomorphic encryption.

As a part of Similarity of Simultaneous Observation (SSO) [15] and Proof of Presence Share (Pop-Share) [6], a video was represented in probability distribution form, to compare two probability distributions, there are many methods to produce a similarity score. In this section, we cover alternative considerations for computing a similarity score, as well as, acknowledge alternative approaches to comparing videos.

Jensen-Shannon Divergence [21], represented as JSD in Equation 3.1, is a symmetric adaption of Kullback-Leibler Divergence that quantifies the similarity between two probability distributions. It is bounding the upper limit of KLD to 1, where 0 indicates high similarity, and 1 indicates the maximum dissimilarity between the distributions.

$$JSD = \frac{1}{2}D_{KL}(P||M) + \frac{1}{2}D_{KL}(Q||M) \tag{3.1}$$

Pearson Correlation Coefficient [22], represented as PCC in Equation 3.2, measures the

linear correlation between two variables, producing a value between -1 and 1. This coefficient is designed to measure the direction of a straight-line relationship between two probability distributions. However, if the relationship between the variables follows a nonlinear pattern, the coefficient may not accurately capture the relationship.

$$PCC = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}} \tag{3.2}$$

Dynamic Time Warping (DTW) [16], represented as $C(i, j)$ in Equation 3.3, is an algorithm that measures similarity between two-time series by exhaustively finding the optimal alignment between them, even if they have different speeds or lengths. However, DTW has a high computational cost, especially for long sequences, which can limit its use in practice.

$$C(i,j) = D(i,j) + \min \begin{cases} C(i-1,j) \\ C(i,j-1) \\ C(i-1,j-1) \end{cases} \tag{3.3}$$

Chapter 4

DESIGN

The core design principle is modularity, with each component designed to support additional cryptographic libraries and be flexible for future applications. This project is comprised of three components: two Dart plugins and a Flutter application. In Section 4.1, the Fully Homomorphic Encryption (FHE) Library plugin integrates Microsoft SEAL and delivers FHE arithmetic for the Dart and Flutter communities. In Section 4.2, the Distance Measure plugin implements Kullback-Leibler Divergence, Cramer Distance, and the Bhattacharyya Coefficient, along with simplified versions of their original algorithms developed for basic FHE arithmetic operations. In Section 4.3, GhostPeerShare depends on both Dart plugins as a peer-to-peer Flutter application.

4.1 Fully Homomorphic Encryption Library

This implementation adheres to the core design principle of modularity, remaining agnostic to the underlying C++ library. Instead of directly invoking methods within the plugin, the plugin uses the Bridge and Adapter structural design patterns inspired by the work of Alberto Ibarrondo and Alexander Viand [5]. The Bridge pattern decouples the high-level Dart API from the specific C++ implementation by defining an abstract interface. The Adapter pattern connects the Dart interface to the C++ implementation through a C-based intermediary, translating Dart requests into operations understood by the native library. This approach enables compatibility with multiple Fully Homomorphic Encryption (FHE) backends without modifying the Dart API.

The Fully Homomorphic Encryption Library provides a modular API that integrates Microsoft SEAL with the Flutter ecosystem. It offers a straightforward interface for developers,

Table 4.1: Class Structure Overview

File	**Class**	**Description**
seal.dart	Seal	Entry-point for the end-user API in Dart
afhe.dart	Afhe	Dart adapter connecting to the C interface
fhe.cpp	N/A	C interface bridging Dart and C++
afhe.h	Afhe	Pure abstract class defining the FHE contract
aseal.cpp	Aseal	Concrete implementation of the Afhe abstraction

abstracting low-level complexities while delivering access to advanced encryption functionalities required for secure applications. The topology of this Dart plugin ensures a clear separation of responsibilities to promote maintainability and extensibility. Table 4.1 outlines the distinct roles of each component, including the high-level API, the adapter interface, and the concrete implementation.

Each file plays a specific role in this architecture. The *seal.dart* file provides the primary entry point for the end-user API, allowing developers to interact with the encryption library in Dart. The *afhe.dart* file acts as the adapter, invoking lower-level C functions. The Afhe class exposes abstracted data types such as Keys, Ciphertext, and Plaintext objects. The *fhe.cpp* file defines the C interface. It exposes the inherited methods from *afhe.h* and references the memory address of the underlying C++ object. Depending on the reference, the corresponding concrete implementation will be invoked. For example, *aseal.cpp* manages Microsoft SEAL objects.

The Foreign Function Interface (FFI) is this plugin's core dependency; It enables Dart to interact with the pre-compiled C binary. For each target platform, the dynamically linked library must be accessible on the local file system. This plugin exposes methods to cast native C data types into Dart objects and vice versa. Our implementation creates Dart objects that mirror their corresponding underlying C++ class. For example, in Microsoft SEAL, a Ciphertext can be saved or loaded. We expose *save_ciphertext* and *load_ciphertext*

in the C interface. In Dart, we create a Ciphertext class that contains both *save* and *load*, referencing the memory address of the underlying abstract Ciphertext object. This implementation pattern is consistent and transparent for security researchers familiar with the underlying C++ API. For new developers, our implementation mirrors the structure of Microsoft SEAL, providing a clear and familiar interface.

In order to prevent breaking changes, unit tests ensure that the functional behavior of the library remains consistent and reliable throughout the development and release. Unit tests serve as a safety net. They ensure that existing features function as expected when new changes occur. This process reduces the risks of regressions. For SEAL, the authors developed a set of examples that walk the user through their APIs. As a part of the unit tests, we re-implemented these examples with our interface to increase our confidence that our implementation did not introduce any new or unexpected behaviors. For automation, GitHub Actions compiles and executes unit tests using GoogleTest [23] with CMake. In total, There are 78 unit tests: 35 in C and 43 in Dart.

In order to distribute this plugin to Linux and multiple Android flavors, GitHub Actions employs Conan [24]. Conan automates the management of dependencies and packaging, specifically addressing the versioning of Microsoft SEAL and other backend libraries. For Linux, Conan streamlines the building and packaging process by automatically resolving dependencies tailored to various distributions (e.g., Ubuntu, Fedora, and CentOS) while ensuring compatibility with different versions of Microsoft SEAL and system architectures (e.g., x86, ARM). Similarly, Conan simplifies cross-compilation for Android by allowing developers to specify configurations for various architectures and API levels, ensuring that the plugin can be compiled for the supported CPU architectures x86_64, ARMv8, and ARMv7. The compiled binaries are all hosted on all platforms and are available for download for each release of the Dart plugin. When developers add the plugin to their dependencies, the binaries are pulled down from GitHub and stored within their local, versioned dependency cache. This automated approach enhances the consistency and reliability of the plugin across different environments, facilitating smoother deployment and easier updates for users.

4.2 Distance Measure Plugin

This Dart plugin calculates three distance measures: Kullback-Leibler Divergence, Cramer Distance, and Bhattacharyya Coefficient, as well as their Fully Homomorphic Encryption (FHE) counterpart. To manage the complexity of these computations, the plugin only supports operations on lists of ciphertext doubles. In this section, we explain the fully homomorphic computations for each algorithm, along with details on testing and distribution.

Each algorithm's encrypted score remains interpretable because it retains the mathematical relationships needed for distance calculation without exposing the underlying plaintext values. For more information on the equations of the plaintext scores, refer to Section 2.3. A major limitation in FHE schemes is their inability to directly support division due to the mathematical structures in FHE, which operate within polynomial or modular arithmetic. Workarounds, such as using multiplicative inverses, do exist, though they add complexity and are unreliable. To accommodate the limitations of FHE, we cannot directly encrypt the input array of doubles for score computation. Instead, we adjust certain parameters tailored to each algorithm's requirements before encryption.

To compute the encrypted Kullback-Leibler Divergence using FHE, noted as KLD_{enc} in Equation 4.1, the ciphertext modification requires three parameters: X, $logX$, and Y, where X and $logX$ are encrypted arrays, and Y is plaintext. The modified ciphertext is obtained by multiplying each encrypted double in X by the difference between $logX$ and $logY$. This yields an encrypted list of doubles, which, when decrypted and summed, results in the divergence of Y from X, shown in Equation 4.2.

$$KLD_{enc} = (\log(X)^i_{enc} - \log(Y^i_{plain})) \cdot X^i_{enc} \tag{4.1}$$

$$KLD_{plain} = \sum_{i=1}^{n} D_K(KLD_{enc}) \tag{4.2}$$

For the Bhattacharyya Coefficient, indicated as BC_{enc} in Equation 4.3, the square root of X is taken before encryption. This calculation involves two arrays, $sqrtX$ and Y, both

floating-point arrays. We multiply each encrypted double in $sqrtX$ by the square root of Y, yielding an encrypted list. Decrypting and summing this list gives the measure of overlap between the two probability distributions, shown in Equation 4.4.

$$BC_{enc} = (\sqrt{X})^{i}_{enc} \cdot \sqrt{Y^{i}_{plain}} \tag{4.3}$$

$$BC_{plain} = \sum_{i=1}^{n} D_K(CD_{enc}) \tag{4.4}$$

To calculate the Cramer Distance, or CD_{enc} in Equation 4.5, the cumulative distribution of X is computed iteratively so that the last element sums to one. This requires two arrays, X and Y, of floating-point numbers. The computation squares the difference between corresponding elements in X and Y, resulting in an encrypted list of values, which, when decrypted, summed, and square-rooted provides the distance between the distributions, shown in Equation 4.6.

$$CD_{enc} = (X_i - Y_i)^2 \tag{4.5}$$

$$CD_{plain} = \sqrt{\sum_{i=1}^{n} D_K(CD_{enc})} \tag{4.6}$$

The plugin undergoes automated testing via GitHub Actions, which validates scores generated from identical input data to within a 7-decimal-point precision (as shown in Table 5.1, which details the noise introduced by FHE). Testing covers both the symmetric and asymmetric behavior of each algorithm. A core design goal of this library is modularity, allowing for the future integration of additional distance measures using FHE. This work introduces the FHEL Dart plugin, creating a foundational Dart plugin for distance measurements that supports fully homomorphic encryption.

4.3 Video Similarity Application

GhostPeerShare uses the CKKS [13] fully homomorphic scheme for all encryption arithmetic. Specifically, this scheme uses three security parameters: 1) Polynomial Modulus Degree, 2) Encoder Scalar, and 3) Sizes of the Coefficient Modulus Chain (qSizes). The Polynomial Modulus Degree configures the size of the ciphertext and the number of slots available for encoding data. Next, CKKS encodes real numbers into polynomials, and the encoder scalar is used to scale real numbers by a significant factor to preserve precision. Finally, the Sizes (in bits) in the Coefficient Modulus Chain contribute to the noise budget. The modified ciphertext cannot be decrypted if too much noise is introduced. By default, GhostPeerShare selects the Polynomial Modulus Degree of 4096, Encoder Scalar of 2^{40}, and qSizes of [60, 40, 40, 60].

In order to detect similarities between the two videos, we represent each video as a probability distribution of byte size changes between one-second segments, shown in Figure 4.1. To transform raw video into comparable probability distributions, we count the number of bytes in each frame, calculate the sum of frame lengths in each segment, and then normalize the array of segments between zero and one, known as a Probability Distribution Form (PDF), ensuring that the distributions for both videos have the same scale and can be directly compared. This re-implementation of Proof of Presence Share [6] provides strong privacy guarantees, as if the encryption is broken, the raw byte data cannot be reconstructed from the normalized byte array due to the loss of information during the aggregation and normalization process. Alternative approaches include a motion estimation approach with the Lucas-Kanade method [25] supported by OpenCV, which could provide more detailed information about video content. However, for this study, it is crucial to compare our approach to the established baseline metrics, and new methods would hinder this comparison.

To compute multiple similarity scores using homomorphic encryption, separate byte arrays are encrypted for each score. The recipient can perform computations with plaintext decimal values against the corresponding encrypted element in the array. This approach

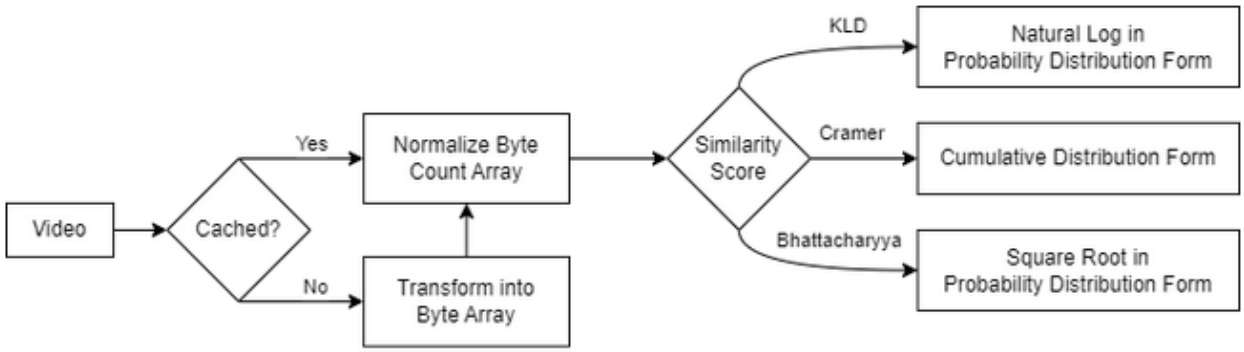

Figure 4.1: Pre-process Data Flow

leverages the unique ability of fully homomorphic encryption and was borrowed from PopShare, as this approach is much faster than comparing ciphertext against ciphertext.

For Kullback-Leibler Divergence (KLD), we share the PDF and its natural log. For each encrypted double in the PDF, we subtract the natural log of the plaintext element and multiply the difference. The product remains encrypted and is returned to the originator, where it can be decrypted. The summation is the similarity score.

For Bhattacharyya Coefficient (BC), we share the square root of the PDF. For each encrypted double, we multiply the ciphertext by the square root of the plaintext element. The product remains encrypted and is returned to the originator, where it can be decrypted. The summation is the similarity score.

For Cramer Distance (CD), we share the Cumulative Distribution Form (CDF) derived from PDF. The CDF is the cumulative sum of all of the elements in the PDF, such that the last value of CDF is equal to 1. For each encrypted double, we square the difference of the plaintext element. The product remains encrypted and returned to the originator, where it can be decrypted, and the square root of the summation is the similarity score.

In order to share the encrypted byte-count arrays, GhostPeerShare packages all ciphertext objects into an archive, the contents illustrated in Figure 4.2. The *meta.json* files contain all computed metadata about the video file, including the timestamp, frames per second, and

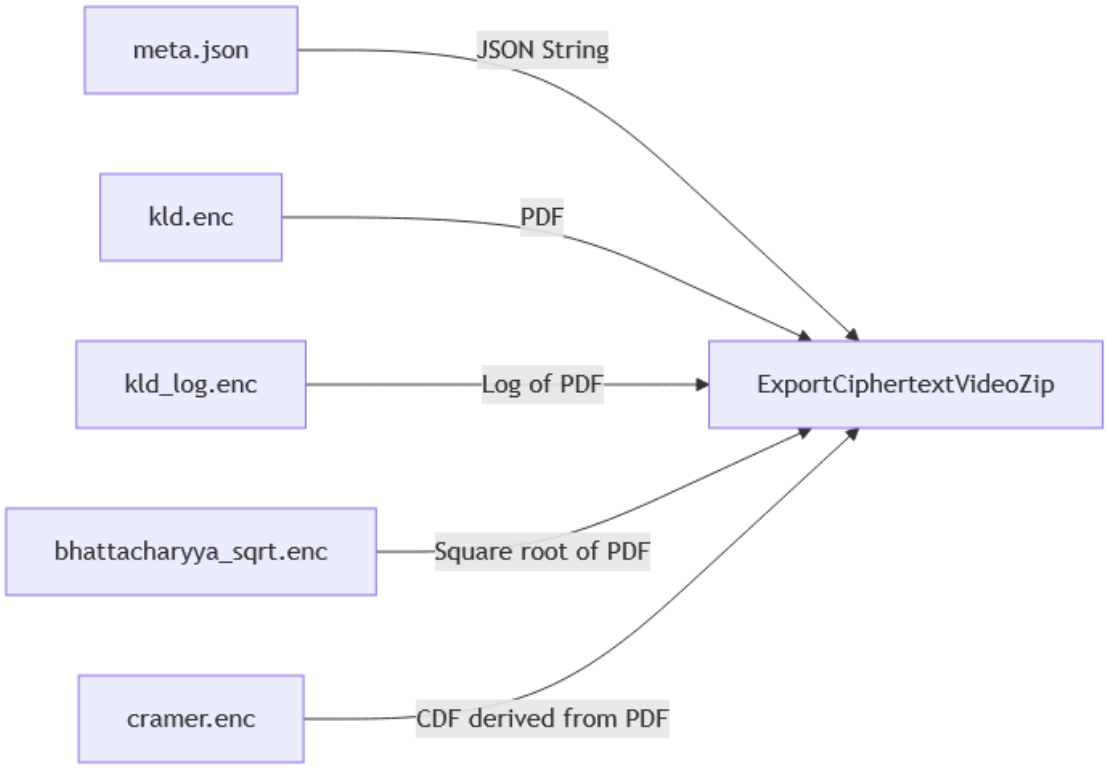

Figure 4.2: Serialized Ciphertext Archive

duration. For a one-minute video, each encrypted archive, files ending with *.enc*, contains 60 binary files, each representing a one-second segment. Each binary file contains an encrypted floating-point number. This structure is useful for cases where the two videos are different lengths, simplifying the trimming process. The *kld.enc* and *kld_log.enc* are the binary archives containing the encrypted parameters to compute the KLD distance measure scores. The *kld.enc* binary archive contains the encrypted PDF. The *kld_log.enc* binary archive contains the encrypted logarithm of PDF. The *bhattacharyya_sqrt.enc* binary archive contains the encrypted square root of PDF. The *cramer.enc* binary archive contains the encrypted CDF. The serialized archive, *ExportCiphertextVideoZip* contains all binary archives and metadata files. In total, the serialized archive averages around 40 MB on Linux and Android for 1

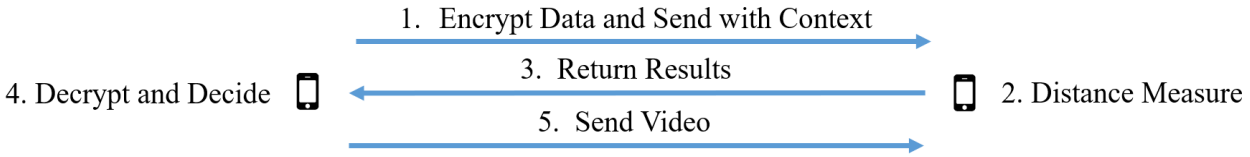

Figure 4.3: Peer-to-peer Data Flow

minute of video.

To facilitate a peer-to-peer exchange of files shown in Figure 4.3, GhostPeerShare supports QuickShare [10] for Android devices. QuickShare [10] was developed by Samsung as a file-sharing utility application for nearby wireless devices. It leverages Bluetooth to discover nearby wireless devices and optionally uses WiFi to transfer large files between two nodes. QuickShare is accessible through a Flutter plugin that enables users to share data using their device's native sharing capabilities. This allows each node to asynchronously send and receive files, leveraging a built-in Android feature rather than embedding similar functionality within the Flutter application.

Chapter 5

RESULTS

In this chapter, we compare our implementation to the foundational contributions of SSO and Pop-Share. In order to evaluate the accuracy of the Fully Homomorphic Encryption Library plugin in Dart, we measure the amount of noise generated during encrypted operations in Section 5.1. In Section 5.2, we compare our implementation of our distance measures using the original pre-processed video frames from SSO. In Section 5.3, we compare the same video from various angles and determine the accuracy of our implementation. In Section 5.4, we compare the distance measures for similar scenes against different scenes. In Section 5.5, we compare the performance across multiple Linux and Android devices.

5.1 Noise Accumulation

When performing fully homomorphic operations, a small amount of noise is introduced that affects the accuracy of the decrypted distance measure. If too much noise is accumulated, the plaintext cannot be recovered. Table 5.1 represents the baseline Mean Error from Pop-Share [6], compared to our implementation. For Kullback-Leibler Divergence (KLD), Cramer Distance (CD), and Bhattacharyya Coefficient (BC), our implementation introduced half the amount of noise compared to the original. This was computed by subtracting the mean error of GhostPeerShare (GPS) from Pop-Share (PS), such that $PS - GPS$ resulted in approximately the same value as Pop-Share. This difference is likely attributed to the changes in the security parameters and advances in Microsoft SEAL between v3.3 and v4.1, respectively.

GhostPeerShare uses the CKKS [13] fully homomorphic scheme for all encryption arithmetic. Specifically, this scheme uses three security parameters: 1) Polynomial Modulus Degree, 2) Encoder Scalar, and 3) Sizes of the Coefficient Modulus Chain (qSizes). The

Table 5.1: Difference of Mean Error for each algorithm

Function	**Pop-Share** (PS)	**GhostPeerShare** (GPS)	**Difference** ($PS - GPS$)
KLD	4×10^{-9}	1.71×10^{-11}	3.98×10^{-9}
Cramer	4×10^{-4}	1.61×10^{-9}	3.99×10^{-4}
BC	8×10^{-4}	3.85×10^{-10}	7.99×10^{-4}

Polynomial Modulus Degree configures the size of the ciphertext and the number of slots available for encoding data. Next, CKKS encodes real numbers into polynomials, and the encoder scalar is used to scale real numbers by a significant factor to preserve precision. Finally, the Sizes (in bits) in the Coefficient Modulus Chain contribute to the noise budget. The modified ciphertext cannot be decrypted if too much noise is introduced. By default, GhostPeerShare selects the Polynomial Modulus Degree of 4096, Encoder Scalar of 2^{40}, and qSizes of [60, 40, 40, 60].

These parameters are a significant factor in managing noise growth. Due to the different major versions of SEAL used in Pop-Share, the only shared parameter value is the Polynomial Modulus Degree of 4096. Otherwise, all other parameters differ. However, Pop-Share compared the performance of 128-bit security to 256-bit security and found that tasks take twice as long using a Polynomial Modulus Degree of 8192. GhostPeerShare, by default, offers a balance in performance and security, achieving an equivalent 128-bit security in SEAL v4.1.

5.2 SSO Distance Measure

One of the core components of this work is the derivation of Distance Measures; however, the baseline for this implementation was established in Python from SciPy [26], sklearn entropy, functions to compute Kullback-Leibler Divergence (KLD) and Cramer's Distance (CD) for their probability distribution arrays. The original data is utilized to ensure a fair comparison between this work and the SSO, comparing video frame byte count arrays against network

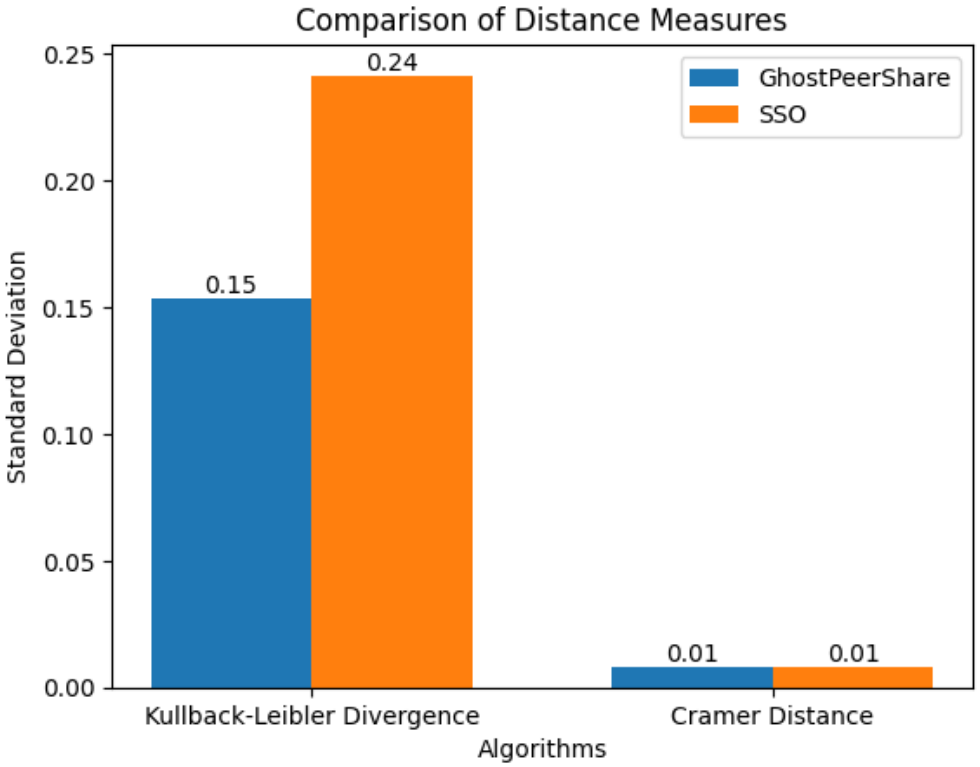

Figure 5.1: Standard Deviation of Distance Measure

traffic data obtained from Pcap packet byte count arrays. Using this data, the standard deviations of our Dart implementations of KLD and CD are derived and compared to those of the Python implementation.

As shown in Figure 5.1, our Dart implementation of Kullback-Leibler Divergence (KLD) demonstrates a standard deviation of 0.15, which is lower than the Python implementation's standard deviation of 0.24. Both implementations incorporate a small epsilon value of $1.0e-12$ to address cases where either p or q are zero. The minimal variance of 0.09 in the Dart implementation suggests that our implementation exhibits less fluctuation compared to the Python implementation, indicating greater stability and consistency in its distance-measure calculations.

For Cramer's Distance (CD), the standard deviation was the same for both implemen-

tations at 0.1. This finding aligns with our hypothesis that we should observe the same or very small amounts of variance between the two implementations, further supporting the reliability of the Dart implementation in maintaining consistency across distance measure calculations.

5.3 Field of View

A significant contribution from Pop-Share [6] was the ability to accurately classify videos from various angles for similarity. Applying the methodology from Similarity of Simultaneous Observation (SSO) [15] to compare videos, Pop-Share re-used the pre-processing procedure to convert videos into a byte-count array. Instead of comparing videos to capture network traffic, the application of SSO was altered to compare two videos to each other for similarity. In addition, Pop-Share expanded upon the distance measure implementation to support Fully Homomorphic Encryption, requiring a new set of algorithms that were simplified into basic arithmetic.

The field-of-view experiment demonstrated that comparing two videos taken at the same time and place of the same subject could be accurately classified using an Artificial Neural Network (ANN). This machine-learning model was trained on the distance measure scores between two videos computing using three probability distribution functions: Kullback-Leibler Divergence (KLD), Bhattacharyya Coefficient (BC), and Cramer Distance (CD). For supervised training, the binary labeling system assigns a one when two scenes are identical, or the first comparison pair is labeled as one to train the model to match similar videos. Otherwise, a zero is assigned for all other comparison pairs, regardless of visual similarity.

Once the videos are aligned and pre-processed, we compute the distance measures of the byte-count arrays representing a one-second segment of each video. In the generated CSV file, each row contains the binary label, followed by all of the scores: KLD, BC, and CD. Since KLD is asymmetric, the scores will differ when comparing A to B versus B to A. We performed an exhaustive comparison of every video, as well as the inverse comparison, to account for asymmetric differences.

Table 5.2: Direct Comparison of SSO-based Systems

System	**F1 (%)**	**Precision (%)**	**Recall (%)**	**Accuracy (%)**	**Error (%)**
GhostPeerShare	97.09	96.14	98.05	98.05	1.95
Handheld[2]	97.97	99.32	96.67	98.00	2.00
SSO[6]	96.13	92.56	100.00	96.30	3.70
PoP-Share[2]	96.63	97.73	95.56	95.16	4.84

The Multi-Layer Perceptron (MLP) classifier is part of the scikit-learn [27] library. The MLP Classifier used the Limited-memory Broyden-Fletcher-Goldfarb-Shanno (lbfgs) solver to apply weights within the neural network. The model was trained with two hidden layers, each with 10 neurons, and used Rectified Linear Unit (relu), a popular activation function in deep learning. This is the same machine learning model used in SSO and Pop-Share. SSO and Pop-Share were trained for 15 hours using a Nexus 6p and a D-Link Wi-Fi camera (DCS-936L) fixed-motion cameras. The video data from the two cameras include video captures at different resolutions and different relative angles, such as 0, 90, and 180 degrees offset from each other. The videos were also taken at varying distances from each other, ranging from 1 to 25 meters away. In addition, the videos were taken from both an indoor and an outdoor environment with varying levels of motion and lighting conditions.

The Handheld model shown in Table 5.2 was tested using 150 minutes of training data recorded with a Google Pixel 2, a Motorola Moto Z, a Lenovo Phab2 Pro, an LG Nexus 5, and a Huawei Nexus 6p. All phones captured video using h.264 with 3840x2160 resolution at 30 FPS with Optical Image Stabilization (OIS) enabled except the Nexus 5 and Phab2 Pro, which only support 1920x1080 resolution, and the Nexus 6p, which only has Electronic Image Stabilization (EIS).

GhostPeerShare was trained and tested on a less diverse dataset of 100 minutes of raw videos, which included three twenty-minute videos of low movement featuring an individual sitting at his desk, denoted as *office*, and two twenty-minute videos of high movement cap-

turing an individual vacuuming his living room, denoted as *vaccum* in Appendix Table A.1. The video data was recorded on fixed-motion phones: Pixel 3XL using h.264 with 1920x1080 resolution at 30 FPS and Samsung S9 using h.264 with 1280x720 resolution at 30 FPS. The phones were placed at different relative angles and varying distances. However, all videos were filmed indoors with relatively similar lighting.

For a practical analysis, the ANN was trained on a high-movement scene and tested on a low-movement scene, achieving an Accuracy and Recall score of 98.05%, a Precision Score of 96.14%, and an F1 Score of 97.09%. The lack of diversity within the training and testing dataset is likely a contributing factor to the lower scores for F1 and Precision. Recording more video data in various locations with a wide variety of devices and environments, the F1 and Precision scores may improve. This model demonstrates that GhostPeerShare consistently generates a unique representation of video data and can accurately predict the binary label given the three distance measure scores for a different dataset.

5.4 Scene Distance Measure

The purpose of this experiment is to investigate the range of distance measure scores from our training data in Section 5.3. Figure 5.2 illustrates the average distance measure score for each algorithm from our dataset, comparing different and same scenes. The scenes were labeled according to our binary labeling system, where one indicates similar scenes and zero otherwise. The graph also includes the standard error above and below the mean distance measure, indicating the range within which the true mean distance measure score is expected. This graph mirrors the same experiment performed by Pop-Share. Pop-Share's training data was much more diverse, with around 16 hours of raw video spanning across various environments and recorded from many devices. GhostPeerShare was training on a total of 100 minutes of indoor video data recorded by two phones.

For the Bhattacharyya Coefficient, a perfect score of 1.00 indicates that the two scenes are identical, while a score of zero denotes no overlap. From Pop-Share, the same pattern was found for the Bhattacharyya Coefficient, containing little variance between different and

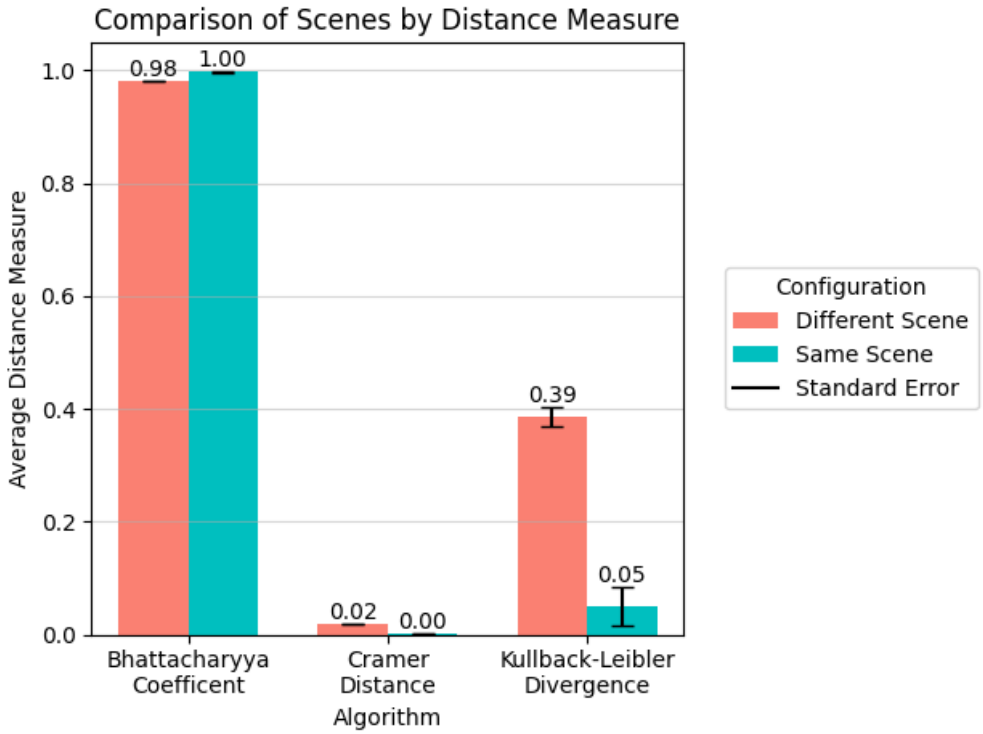

Figure 5.2: Scene Distance Measure

same scenes and significantly less standard error than other metrics. In our experiments, we observed a 0.02 variance between the distance measures for different and same scenes. This minimal variance is likely attributed to the uniformity of the scene types recorded, as both sets were filmed indoors, and the fact that the same devices were used during the experiments, which limited the diversity in the captured footage.

For Cramer Distance, a perfect score of 0.00 indicates that the two scenes are identical, while a score of 1.0 is the maximum distance between two probability distributions. From Pop-Share, Cramer Distance contained a significant absolute difference of around 0.25 between different and same scenes. In our experiments, we observed significantly less variance, 0.02, between the distance measures for different and same scenes. The lack of variance is likely attributed to the limited data used in our experiments compared to Pop-Share.

For the Kullback-Leibler Divergence (KLD), a perfect score of 0.00 indicates that two

scenes are identical, while the upper limit is infinity; the smaller the score, the more similar the two probability distributions are. From Pop-Share, KLD was the leading indicator of dissimilarity, however, their experiments contained significant variance greater than 1 between different and same scenes. In our experiments, we observed a 0.34 variance between the distance measures for different and same scenes. This represents the highest variance observed among all of our distance measures and serves as a significant indicator that two scenes may be different.

The key takeaway from our experiments is that while the individual scores for the distance measures exhibit minimal variance, they still effectively differentiate between similar and different scenes. This consistency suggests that the implemented algorithms can reliably assess similarity, even in scenarios where the overall variation in scores is low. Consequently, the results highlight the robustness of our methodology in accurately identifying scene differences, which is crucial for applications requiring precise content analysis without revealing the underlying video data.

5.5 Performance

To compare this implementation to the mobile Proof of Presence Share (Pop-Share) application, the preprocessing speed for one-minute videos was benchmarked, and the duration of encrypted operations was measured against plaintext operations. This comparison allowed for assessing the implementation's efficiency and quantifying the impact of encryption on performance.

For Pop-Share, pre-processing is not part of the synchronous transaction of comparing videos for similarity. The primary use case of pre-processing is to be performed once, ideally in the background at a later point in time. This intention was brought into GhostPeerShare, where the anticipated use case of pre-processing is to be performed when the user imports the video into the application.

Experiments used one-minute videos with h264 encoding, results shown in Table 5.3. On a Pixel 3 XL mobile phone, preprocessing with OpenCV required 438 seconds (about 7.3 min-

Table 5.3: Pre-processing duration (in seconds) for a one-minute video.

System	**Average (s)**	**Min (s)**	**Max (s)**
PC	111.25	13.00	321.00
Samsung S9	421.62	180.00	755.50
Pixel 3XL	454.50	171.00	875.00

utes) on average. In contrast, Pop-Share on a Pixel 2 took only 17 seconds on average using FFmpeg. This difference stems from the use of different libraries for preprocessing. Limited operating system support for FFmpeg plugins restricted plugin selection to one compatible with both Linux and Android. Regarding Android resource constraints, GhostPeerShare was run on a Pixel 3XL that has 8 threads at about 2.5 GHz with 4 GB of memory. From the Pop-Share benchmark, the Pixel 2 has 8 threads at about 2.35 GHz with 4 GB of memory. Overall, the performance impact is unlikely correlated to the differences in the hardware differences between devices, as they are relatively similar.

On a Linux PC with OpenCV, preprocessing took 111 seconds (about 2 minutes) on average. This result demonstrates that the PC operated approximately four times faster than the average Pixel 3XL. The disparity likely arises from differences in CPU thread availability and speed. The PC has 32 threads at 3.4 GHz with 32 GB of memory, while the Pixel 3XL has 8 threads at 2.5 GHz with 4 GB of memory.

Dart code executes in isolates, which resemble threads but with separate memory allocations. Flutter applications perform all operations on a single isolate. Although the implementation creates an isolate to count bytes in each one-second segment, it does not incorporate parallelism. The frame parsing process remains sequential. This design choice prioritized stability over performance to ensure consistent operation. Table 5.3 presents the execution times on mobile and Linux desktop environments. Pre-processing speed is not a significant concern, as video import is typically performed once in the application's background, preparing the content for sharing at a later point in time.

Table 5.4: Comparison of Cryptography (in Milliseconds) on Android

Algorithm	**Encryption (ms)**	**Decryption (ms)**
KLD	1019.00	229.88
BC	503.00	140.25
CD	507.12	217.50

The encryption and encoding of each distance measure varies, as the parameters to the Fully Homomorphic distance measure implementation are different, described in further detail in Section 4.2. Unfortunately, we cannot compare these metrics to Pop-Share as they were not explicitly mentioned in their evaluation. However, our approach treats the preprocessing of video frames as an independent transaction, separate from the encoding and encryption steps. Shown in Table 5.4, the Kullback-Leibler Divergence (KLD) encryption is about twice that of Bhattacharyya and Cramer due to the requirement of double the amount of data. This pattern was also indicated in Pop-Share.

The Fully Homomorphic Encryption computations, shown in Table 5.5, are within ±20% of Pop-Share, validating our abstraction design and interface implementation. This performance increase is likely due to the advancements in Microsoft SEAL. Pop-Share used version v3.3, while our implementation uses v4.1.

- Kullback-Leibler Divergence: Pop-Share reported an average time of 400 milliseconds, whereas our implementation achieved 346 milliseconds, representing a 13.5% increase in performance.

- Cramer Distance: Pop-Share reported an average time of 386 milliseconds, whereas our implementation achieved 310 milliseconds, representing a 19.4% increase in performance.

- Bhattacharyya Coefficient: Pop-Share reported an average time of 186 milliseconds,

Table 5.5: Comparison of FHE Operations (in Milliseconds) on Android

Algorithm	**FHE Compute (ms)**	**Plaintext Compute (ms)**
KLD	345.38	1.12
BC	202.38	0.41
CD	309.38	0.45

whereas our implementation yielded 203 milliseconds, representing a 9.1% decrease in performance.

The plaintext computations, shown in Table 5.5, are much faster than expected, a significant performance increase in Dart compared to Python implementation. This performance increase may be due to the underlying data structure selection in Dart.

- Kullback-Leibler Divergence: Pop-Share reported an average time of 3.8 milliseconds, whereas our implementation achieved 1.12 milliseconds, representing a 70.5% increase in performance.

- Cramer Distance: Pop-Share reported an average time of 5.2 milliseconds, whereas our implementation achieved 0.45 milliseconds, representing a 91.35% increase in performance.

- Bhattacharyya Coefficient: Pop-Share reported an average time of 3.2 milliseconds, whereas our implementation achieved 0.41 milliseconds, representing an 87.5% increase in performance.

To extract a distance measure from the archive, we calculate the score by decrypting each ciphertext in the modified byte count array and summing all the elements. Cramer's method differs as it involves calculating the square root of the result. Shown in Table 5.4, Pop-Share had an average duration of 250 milliseconds, while our Android implementation took 588

milliseconds. Although the difference is insignificant, it suggests potential differences in our implementation.

In total, on Android, uploading and converting the video frames into byte count arrays took, on average, 438 seconds (7 minutes) for a one-minute video. Encoding, encrypting, and generating the archive took, on average, 2 seconds. When performing Fully Homomorphic Encryption (FHE) operations on all ciphertexts took, on average, 857 milliseconds. Finally, decryption took, on average, 588 milliseconds. In total, it took 3.4 seconds to derive a similarity score using FHE. As the preprocessing is expensive, it only has to be performed once for every video uploaded to the application.

Overall, this study benchmarked the performance of our implementation on Android to the existing Pop-Share application. The results showed significantly slower preprocessing times due to library limitations, but the encryption, computation, and decryption times were competitive with Pop-Share. Our implementation demonstrates the high efficiency of the Fully Homomorphic Encryption operations. Notably, plaintext computations were significantly faster in our implementation.

Chapter 6

DISCUSSION

6.1 Qualitative Analysis

In order to address the usability of our Flutter application and Dart plugins, this section aims to propose methods of gathering qualitative metrics of our implementation that benefit the Flutter community and security researchers. There are two audiences we wish to target in this analysis: 1) Developers and 2) Security Researchers. Developers may be enthusiasts or professionals seeking access to integrate Fully Homomorphic Encryption within an existing or new application. Security Researchers may be trying to apply Fully Homomorphic Encryption to a novel problem or integrate a new C++ library into our FHEL plugin. To reach our target audience, we may employ developer surveys to gather data on consumers of our plugins, monitor the adoption rate using open-source tools, and publish a workshop paper to advertise and peer-review our implementation utilizing reputable journals.

For developer surveys, we aim to gather information about each individual's programming knowledge or familiarity. Using a Likert-scale questionnaire, we may rate the clarity of our API documentation on a scale of 1 to 5. Using open-ended questions to capture insightful feedback from users who may be seeking additional functionality or identify gaps in our implementation. This structured survey format enables maintainers to assess the level of difficulty in integrating the plugins relative to each individual's experience with open-source projects. Gathering data on the consumers of our packages will help us understand their use cases and the greater impact of this project within the Flutter community. These surveys may be available on the distribution platforms the packages are hosted on, specifically GitHub and Pub.Dev.

To capture the adoption rate of our packages, we may leverage the platforms on which

they are hosted. The source code for this project is available on GitHub. Using the star functionality is a popular method of endorsing a project to receive notifications on development updates. The maintainers of this project may visualize the number of stars over time to measure the project's popularity.

Pub.Dev is the official package repository for the Dart and Flutter community. The repository contains a comprehensive set of scores to measure the quality of each package, specifically popularity and pub points. To approximate the popularity of a package within the last 60 days, the repository contains a thumbs-up mechanism similar to the GitHub star button. This button is used as a quantitative metric of the number of developers who like the package. In addition, the number of downloads from distinct callers and the number of dependents contribute to the popularity score. A unique qualitative score, known as Pub Points, evaluates packages across five categories: Dart file conventions, documentation quality, platform support, static analysis, and support for up-to-date dependencies. The maximum achievable score is 160 points.

The *fhel* plugin received a Pub Score of 140 pub points, 1 likes, and a 27% popularity score. On GitHub, the *fhel* repository has received 5 stars from the community. The *fhe_similarity_score* plugin has received a Pub Score of 150 points, 1 likes, and 42% popularity score. On Github, the repository has received 1 star from the community.

6.2 Security Analysis

GhostPeerShare implements the methodology from Pop-Share as a Flutter application. The authors of Pop-Share presented two key insights in their security analysis: 1) Symmetric Key Cryptographic Equivalency and 2) Brute Force.

First, the security of Fully Homomorphic Encryption (FHE) is equivalent to that of other symmetric key cryptosystems, and its strength depends on the chosen parameters. The security equivalency of FHE is a function of the ring dimension and the size of the ciphertext modulus [28]. By default, GhostPeerShare and Pop-Share balance security and performance by selecting a polynomial modulus of 4096, which is equivalent to a symmetric-

key cryptosystem with a 128-bit key, such as the Advanced Encryption Standard (AES-128).

Second, a brute-force attack is unlikely due to the high precision score of the SSO-based system, as demonstrated in Section 5.3. An attacker attempting to input a random video is highly unlikely to produce erroneous results. Since the attacker cannot ascertain how close their video was to being accepted as co-present, the information they can derive regarding the proximity of their video to acceptance is minimal. From a practical perspective, users may choose to discontinue interaction after a certain number of failed attempts.

GhostPeerShare uses archives to package all ciphertext objects and metadata for data sharing between two nodes. However, these archives do not include signature validation and may be vulnerable to tampering. To mitigate this risk, a signature can be derived from the contents of the archive and sent as part of the payload. The recipient can then validate whether the signature matches their computation. From a practical standpoint, this solution requires users to decide how they share the archive with the recipient.

On Android, QuickShare [10] employs end-to-end encryption to secure data in transit. Samsung has also released Private Share for newer Android devices, allowing users to configure a time-to-live (TTL) for shared files. From a practical perspective, users can revoke access to their shared archive after a designated duration with no response.

Chapter 7

CONCLUSION

GhostPeerShare significantly advances the accessibility of Fully Homomorphic Encryption (FHE) within the mobile community. Key contributions demonstrate the efficiency of the Fully Homomorphic Encryption Library (FHEL) and ease of integration into other Dart plugins or Flutter applications. To measure the efficiency of our application, we benchmark GhostPeerShare against Proof of Presence Share (Pop-Share). GhostPeerShare generated the byte-count array 7 minutes slower than Pop-Share when pre-processing videos. However, our implementation may be improved using multi-threading or replaced with the same underlying FFmpeg video processing library as Pop-Share. When trained on a dataset with continuous movement and tested on a dataset with infrequent movement, the application achieved 0.9614 precision and an F1 score of 0.9709. These results indicate that GhostPeerShare consistently generated a distinct representation of each video in the dataset. When executing FHE operations, GhostPeerShare demonstrated similar performance to Pop-Share. The computational times of FHE distance measure under CKKS reveal that our implementation was within 10% of Pop-Share. These results suggest that the FHEL plugin maintains Microsoft SEAL's high performance. Overall, GhostPeerShare re-implements Pop-Share with efficient FHE computation. However, the application requires significant improvement in the pre-processing implementation.

The scope of this project was to develop an Android and Linux application. The FHEL plugin requires additional Conan profiles to support additional operating systems, including Windows, iOS, and macOS. However, this involves prerequisite knowledge in CMake to extend our implementation and depends on the compatibility of the underlying C++ library. The binary distribution system can also be extended with the automated GitHub actions run

on macOS and Windows runners to compile their respective dynamic libraries for release.

A significant limitation of the FHEL plugin is its versioning strategy. Currently, the plugin supports Microsoft SEAL; however, when extended to support additional libraries, changes to any backend library require a minor version change. Rather than deploying a monolith plugin, the version system should deploy multiple artifacts from the same shared codebase, for example, $fhel\_seal$ and $fhel\_openfhe$. The proposed version system allows each FHEL plugin to align its version numbers with the underlying libraries' versions, simplifying version management for package maintainers and consumers.

Ultimately, this project lays the foundation for future applications of FHE in mobile environments, especially in fields like healthcare and secure data transactions, where privacy and accuracy are paramount. By making SEAL accessible within a mobile-compatible, modular framework, we are opening pathways for broader adoption of FHE and setting the stage for innovation in mobile security.

BIBLIOGRAPHY

[1] "Microsoft SEAL (release 4.1)." `https://github.com/Microsoft/SEAL`, Jan. 2023. Microsoft Research, Redmond, WA.

[2] S. Halevi and V. Shoup, "Helib." C++ library, Oct. 2020.

[3] D. Cousins, K. Rohloff, and Y. Polyakov, "Palisade: A library for privacy-preserving cryptography." C++ library, Dec. 2022.

[4] L. S. Vailshery, "Cross-platform mobile frameworks used by global developers 2023," June 2024.

[5] A. Ibarrondo and A. Viand, "Pyfhel," in *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, (New York, NY, USA), ACM, Nov. 2021.

[6] B. Lagesse, G. Nguyen, U. Goswami, and K. Wu, "You had to be there: Private video sharing for mobile phones using fully homomorphic encryption," in *2021 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, IEEE, Mar. 2021.

[7] S. Kullback and R. A. Leibler, "On information and sufficiency," *Ann. Math. Stat*, vol. 22, no. 1, pp. 79–86, 1951.

[8] A. Bhattacharyya, "On a measure of divergence between two multinomial populations," *Sankhyā Indian J. Stat*, vol. 7, no. 4, pp. 401–406, 1933.

[9] H. Cramér, "On the composition of elementary errors," *Scand. Actuar. J.*, vol. 1928, pp. 13–74, Jan. 1928.

[10] Samsung, "Quick share," Feb. 2020.

[11] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," in *Cryptology ePrint Archive*, no. 2012/144.

[12] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pp. 309–325, Association for Computing Machinery.

[13] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology – ASIACRYPT 2017*, Lecture notes in computer science, pp. 409–437, Cham: Springer International Publishing, 2017.

[14] A. Al Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, Z. Liu, D. Micciancio, I. Quah, Y. Polyakov, Saraswathy, K. Rohloff, J. Saylor, D. Suponitsky, M. Triplett, V. Vaikuntanathan, and V. Zucca, "OpenFHE," in *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, (New York, NY, USA), ACM, Nov. 2022.

[15] K. Wu and B. Lagesse, "Do you see what i see? detecting hidden streaming cameras through similarity of simultaneous observation," in *2019 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, IEEE, Mar. 2019.

[16] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," *IEEE Trans. Acoust.*, vol. 26, pp. 43–49, Feb. 1978.

[17] M.-K. Shan and S.-Y. Lee, "Content-based video retrieval based on similarity of frame sequence," in *Proceedings International Workshop on Multi-Media Database Management Systems (Cat. No.98TB100249)*, IEEE Comput. Soc, 2002.

[18] Y. Wu, Y. Zhuang, and Y. Pan, "Content-based video similarity model," in *Proceedings of the eighth ACM international conference on Multimedia*, (New York, NY, USA), ACM, Oct. 2000.

[19] A. Loza, L. Mihaylova, N. Canagarajah, and D. Bull, "Structural similarity-based object tracking in video sequences," in *2006 9th International Conference on Information Fusion*, IEEE, July 2006.

[20] M.-R. Ra, R. Govindan, and A. Ortega, "P3: Toward privacy-preserving photo sharing," Feb. 2013.

[21] J. Lin, "Divergence measures based on the shannon entropy," *IEEE Trans. Inf. Theory*, vol. 37, no. 1, pp. 145–151, 1991.

[22] K. Pearson, "Mathematical contributions to the theory of evolution. III. regression, heredity, and panmixia, " philos," *Philos. Trans. R. Soc. Lond. Ser. Contain. Pap. Math. Phys. Character*, vol. 187, pp. 253–318, 1896.

[23] "googletest: GoogleTest - google testing and mocking framework."

[24] "conan: Conan - the open-source C and c++ package manager."

[25] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *Proceedings of the 7th international joint conference on Artificial intelligence*, vol. 2, pp. 674–679, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc, 1981.

[26] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[28] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan, "Homomorphic encryption standard," in *Protecting Privacy through Homomorphic Encryption*, pp. 31–62, Cham: Springer International Publishing, 2021.

[29] J. Fan, H. Luo, M.-S. Hacid, and E. Bertino, "A novel approach for privacy-preserving video sharing," in *Proceedings of the 14th ACM international conference on Information and knowledge management*, (New York, NY, USA), ACM, Oct. 2005.

Appendix A

SUPERVISED LEARNING

In this section, we present all of the configurations of the Supervised Artificial Neural Network experiments. We trained and tested the model on a small dataset of 100 minutes of raw videos, which included three twenty-minute videos of low movement featuring an individual sitting at a desk, denoted as *office*, and two twenty-minute videos of high movement capturing an individual vacuuming his living room, denoted as *vaccum* in Appendix Table A.1. We label *default* representing the default parameters of the Multi-Layer Perceptron (MLP) Classifier [27]. The MLP Classifier used the Limited-memory Broyden-Fletcher-Goldfarb-Shanno (lbfgs) solver to apply weights within the neural network. The model was trained with two hidden layers, each with 10 neurons, and used Rectified Linear Unit (relu), a popular activation function in deep learning. Labeled as *gridSearch*, a model fitting algorithm. Due to the visual similarity of the videos filmed indoors from the same devices, the grid search performs very well on training data but poorly on additional test data. Due to this overfitting pattern with grid search on our small dataset, we only considered results where we trained on one scene and tested on a different scene to ensure we have proper diversity.

Table A.1: Performance metrics for different configurations.

Configuration	**Accuracy (%)**	**Precision (%)**	**Recall (%)**	**F1-Score (%)**
office_default	97.42	94.90	97.42	96.14
office_gridSearch	99.85	99.85	99.85	99.85
vacuum_default	97.58	95.21	97.58	96.38
vacuum_gridSearch	99.65	99.66	99.65	99.64
office_vs_vacuum_default	99.82	100.00	99.82	99.91
office_vs_vacuum_gridSearch	99.95	100.00	99.95	99.98
train_office_test_vacuum_default	97.23	94.54	97.23	95.86
train_vacuum_test_office_default	98.05	96.14	98.05	97.09

Appendix B

DESIGN ARTIFACTS

In this section, we present the design artifacts used to develop GhostPeerShare. These artifacts serve as a visual aid to represent the author's intent. This is not a comprehensive collection of all classes within GhostPeerShare but only a subset to demonstrate the application's structure and organization.

To optimize the application, Figure B.1, a singleton pattern used to manage imported videos, pre-processed byte-count arrays, and imported archives. This system manages the read-and-write flow of data throughout the application. This system enables preprocessing to only be run once and load previously imported videos.

To implement modular support of various media types, GhostPeerShare defines additional media types. Figure B.2 demonstrates the abstraction pattern to support future media types rather than tightly coupling the application to only support videos. Furthermore, Figure B.3 demonstrates the logic of adding a trimmed video to the manifest, and Figure B.4 is a separate intent to preprocess the video and cache the resulting byte-count array.

Finally, to facilitate the import and export of encrypted data between instances, we used archives to compress and share larger, serialized objects, shown in Figure B.5. The contents of the *CiphertextVideoZip* and *ImportCiphertextVideoZip* are shown in Figure 4.2. To convey the intent, regardless of how data is transmitted between devices, the behavior of handling archives are shown in Figure B.6.

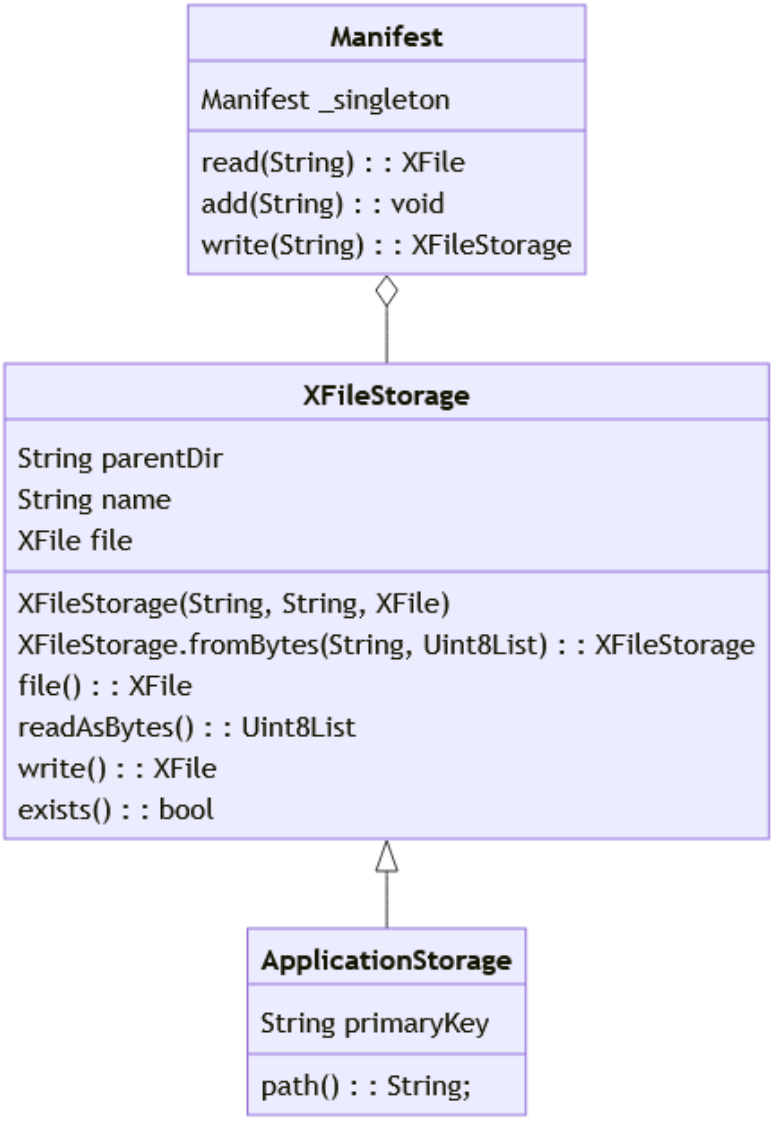

Figure B.1: UML Manifest Class Diagram

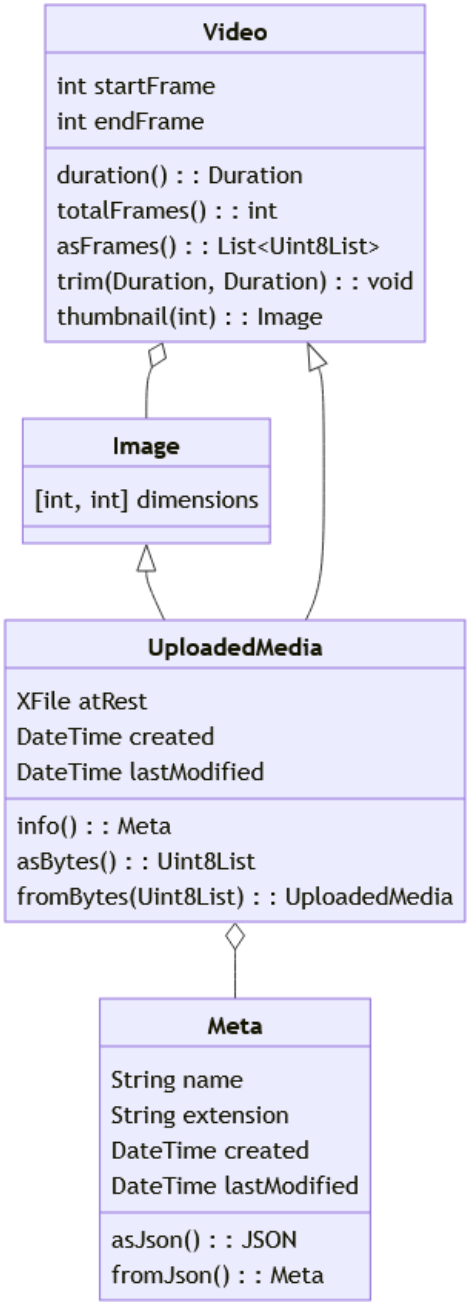

Figure B.2: UML Video Class Diagram

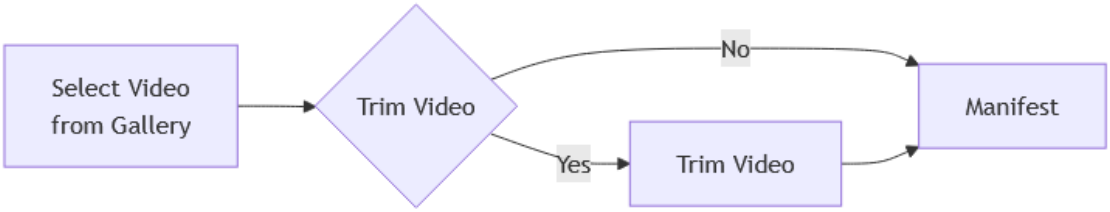

Figure B.3: Video Import Data Flow Diagram

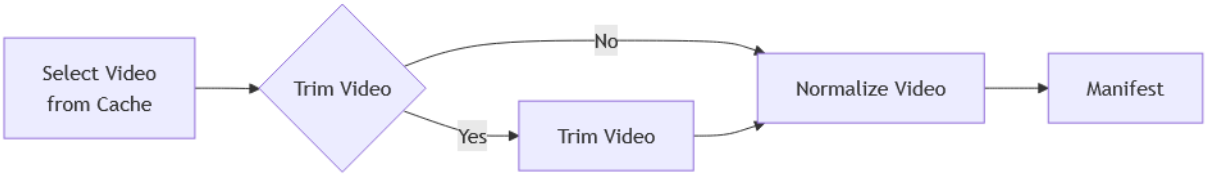

Figure B.4: Video Preprocess Data Flow Diagram

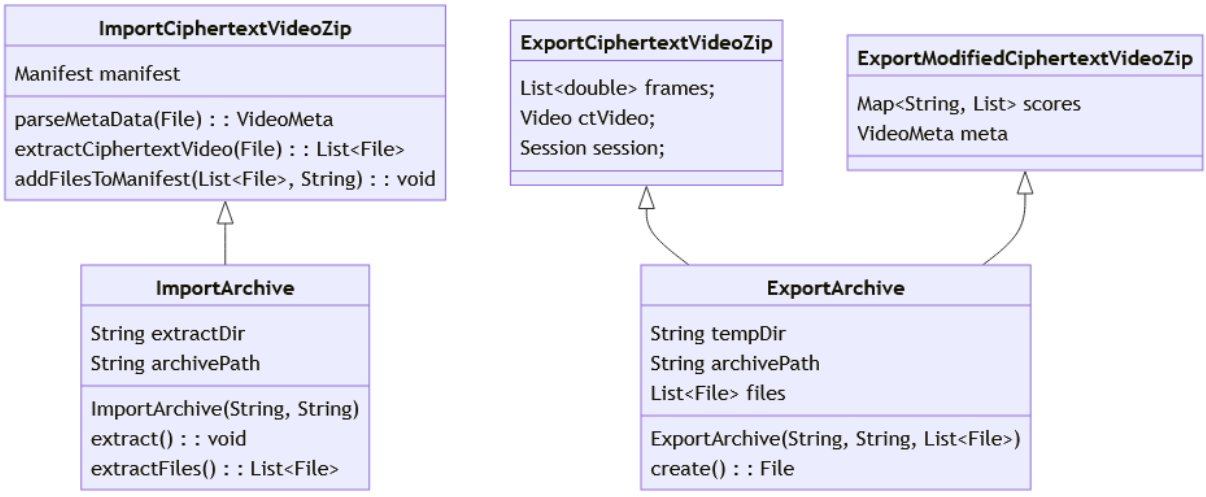

Figure B.5: UML Archive Class Diagram

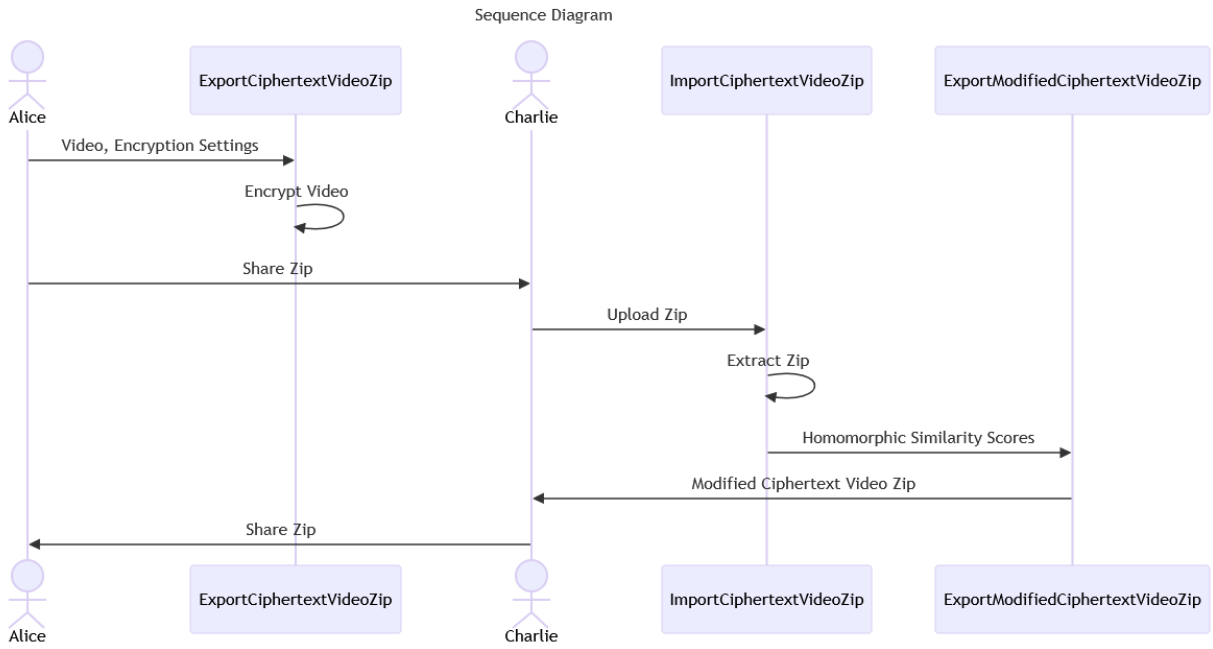

Figure B.6: Archive Data Flow