CHEX-IA

Child Exploitation Image Analytics

FACE RECOGNITION EVALUATION

An Evaluation Activity sponsored by the DHS Science & Technology Directorate

Homeland Security

Science and Technology

Concept, Evaluation Plan and API

Version 0.6, January 7, 2016

Patrick Grother and Mei Ngan

Contact via chexia-face@nist.gov

National Institute of Standards and Technology
U.S. Department of Commerce

Provisional Timeline of the CHEXIA-FACE Evaluation

Phase 0 API Development	2015-10-26	Draft evaluation plan
	2015-11-16	Final evaluation plan
Phase 1	2015-12-15	Participation starts: Algorithms may be sent to NIST
	2016-01-20	**Last day for submission of algorithms to Phase 1**
	2016-02-22	Interim results released to Phase 1 participants
Phase 2	**2016-04-20**	**Last day for submission of algorithms to Phase 2**
	2016-05-20	Interim results released to Phase 2 participants
Phase 3	**2016-07-27**	**Last day for submission of algorithms to Phase 3**
	2016-Q4	Release of final public report

Notable differences from FRVT 2012

Anonymous participation is allowed – but see section 1.8.

Please note that this document is derived from the FRVT 2012 API document for continuity and to aid implementers of the CHEXIA-FACE API.

— This evaluation is dedicated solely to imagery relevant to child exploitation. NIST seeks to assist developers in any way possible to improve algorithm accuracy on this task, and is open to creative ideas on how to do so.

— We anticipate running the algorithms only on child exploitation imagery. We may also run algorithms on other images if that will isolate relevant factors that will influence accuracy. We do not intend to run the algorithms on cooperative images used in recent FRVT tests.

— This evaluation drops the following:

  – Facial age, gender, pose conformance, and expression estimation for still images (see section 1.9)

  – The class F evaluation of frontal pose rendering algorithms

— This evaluation:

  – Merges the idea of "still" and "video". This abstraction supports verification and identification functions where either "sample" may be a still or video. See section 2.4.2.

  – Adds a face detection task in which the algorithm reports locations of faces detected in images. See Section 3.3.

  – Adds a clustering task in which the algorithm finds and groups images of an unknown number of identities. See section 3.4.

  – Employs GPUs on some NIST machines.

— The header/source files for the API will be made available to implementers at http://nigos.nist.gov:8080/chexia-face.

Table of Contents

1. CHEXIA-FACE 5
1.1. Scope 5
1.2. Audience 5
1.3. Market drivers 5
1.4. Test datasets 6
1.5. Offline testing 6
1.6. Phased testing 6
1.7. Interim reports 7
1.8. Final reports 7
1.9. Application scenarios 7
1.10. Options for participation 8
1.11. Number and schedule of submissions 8
1.12. Core accuracy metrics 9
1.13. Reporting template size 9
1.14. Reporting computational efficiency 9
1.15. Exploring the accuracy-speed trade-space 9
1.16. Hardware specification 9
1.17. Operating system, compilation, and linking environment 10
1.18. Software and Documentation 10
1.19. Runtime behavior 11
1.20. Threaded computations 12
1.21. Time limits 12
1.22. Ground truth integrity 13
2. Data structures supporting the API 13
2.1. Namespace 13
2.2. Overview 13
2.3. Requirement 13
2.4. File formats and data structures 13
2.5. File structures for enrolled template collection 18
3. API Specification 19
3.1. 1:1 Verification 19
3.2. 1:N Identification 23
3.3. Face Detection 29
3.4. Clustering 31
4. References 33
Annex A Submission of Implementations to the CHEXIA-FACE 34
A.1 Submission of implementations to NIST 34
A.2 How to participate 34
A.3 Implementation validation 35
Annex B Effect of Age on Face Identification Accuracy 36

List of Tables

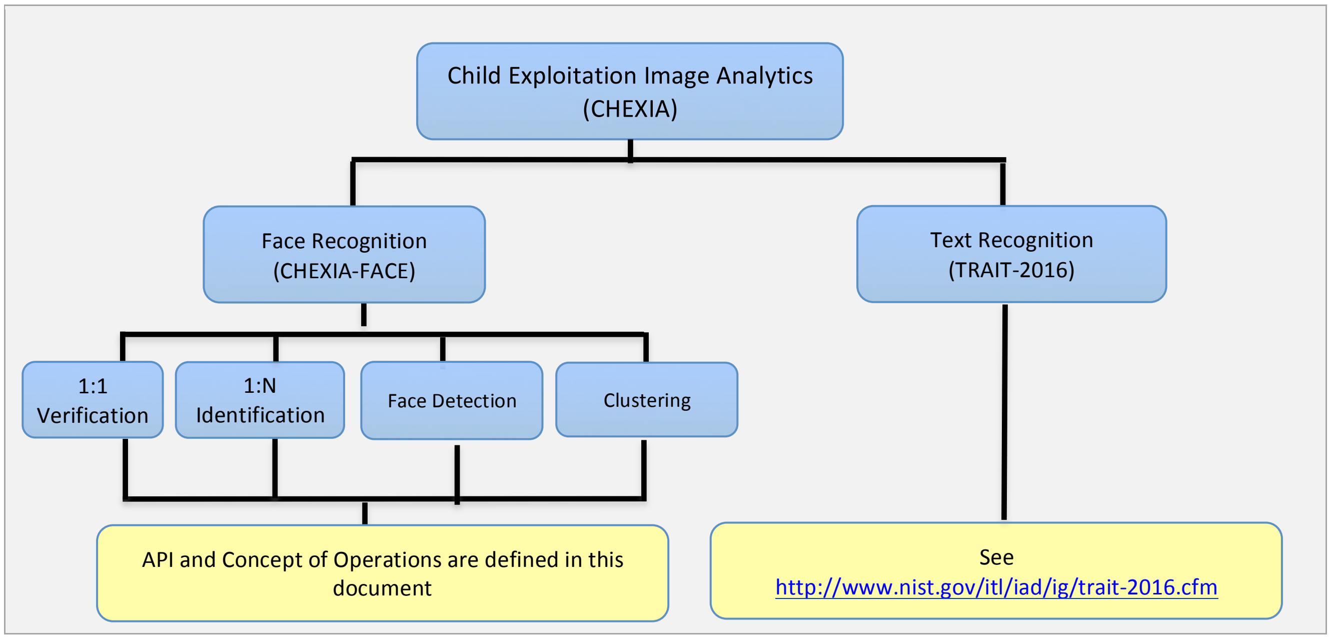
Table 1 – Main image corpora (others may be used) 6
Table 2 – Subtests supported under the CHEXIA-FACE activity 7
Table 3 – CHEXIA-FACE classes of participation 8
Table 4 – Cumulative total number of algorithms, by class 8
Table 5 – Implementation library filename convention 11
Table 6 – Number of threads allowed for each application 12
Table 7 – Processing time limits in milliseconds, per 640 x 480 image 12
Table 8 – Structure for a single image or video frame 14
Table 9 – Structure for a set of images or video frames 14

Table 10 – Labels describing categories of Multifaces....14
Table 11 – Structure for a pair of eye coordinates....15
Table 12 – PersonTrajectory typedef....15
Table 13 - Class for representing a person....16
Table 14 – Structure for bounding box around a detected face....16
Table 15 – Structure for a single hypothesized cluster membership....17
Table 16 – Structure for hypothesized cluster membership for face(s) in an image....17
Table 17 – Structure for a candidate....17
Table 18 – Enumeration of return codes....17
Table 19 – ReturnStatus structure....18
Table 20 – Enrollment dataset template manifest....18
Table 21 – Functional summary of the 1:1 application....19
Table 22 – Initialization....20
Table 23 – GPU index specification....21
Table 24 – Template generation....21
Table 25 – Template matching....21
Table 26 – Procedural overview of the identification test....23
Table 27 – Enrollment initialization....25
Table 28 – GPU index specification....25
Table 29 – Enrollment feature extraction....26
Table 30 – Enrollment finalization....26
Table 31 – Identification feature extraction initialization....27
Table 32 – Identification feature extraction....27
Table 33 – Identification initialization....28
Table 34 – Identification search....28
Table 35 – SDK initialization....29
Table 36 – GPU index specification....29
Table 37 – Face detection....30
Table 38 – SDK initialization....31
Table 39 – GPU index specification....32
Table 39 – Clustering....32

1. CHEXIA-FACE

1.1. Scope

This document establishes a concept of operations and an application programming interface (API) for evaluation of face recognition (FR) implementations submitted to NIST's Child Exploitation Image Analytics Face Recognition Evaluation (CHEXIA-FACE).


Child Exploitation Image Analytics (CHEXIA)
Face Recognition (CHEXIA-FACE)
Text Recognition (TRAIT-2016)
1:1 Verification
1:N Identification
Face Detection
Clustering
API and Concept of Operations are defined in this document
See http://www.nist.gov/itl/iad/ig/trait-2016.cfm

1.2. Audience

Universities and commercial entities with capabilities in any of the following areas are invited to participate in the CHEXIA-FACE test.

— Identity verification with face recognition algorithms.

— Large scale identification implementations.

— Face detection algorithms.

— Implementations with an ability to cluster (find and group) images of an unknown number of identities.

Organizations will need to implement the API defined in this document. Participation is open worldwide. There is no charge for participation. While NIST intends to evaluate technologies that could be readily made operational, the test is also open to experimental, prototype and other technologies.

1.3. Market drivers

There is a growing market around digital forensics – the ability to extract semantic information from imagery that is useful to an investigation. This test specifically is intended to assess the efficacy of face recognition algorithms on child exploitation imagery. These images are of interest to NIST's partner law enforcement agencies that seek to employ face recognition in investigating this area of serious crime. The primary applications are identification of previously known victims and suspects, detection of new victims and suspects. Given a collection of images, produce a cluster of identities, from which (law enforcement) investigations can proceed.

A parallel effort, TRAIT 2016, seeks to improve the capability of algorithms to recognize text in unconstrained images. Text appears frequently in child exploitation imagery. See http://www.nist.gov/itl/iad/ig/trait-2016.cfm .

1.4. Test datasets

NIST anticipates running the algorithms only on child exploitation imagery. NIST may also run algorithms on other images if that will isolate factors that will influence accuracy. NIST does not intend to run the algorithms on cooperative images used in recent FRVT tests. The data has, in some cases, been estimated from initial small partitions. The completion of this section depends on further work. The information is subject to change.

Table 1 – Main image corpora (others may be used)

	Child exploitation	TBD
Collection, environment	Mostly inside a home, sometimes outdoors	
Live photo, Paper scan	Live	
Documentation	See NOTE below	
Compression from [MBE 2010][1]	Variable	
Maximum image size	Some from contemporary SLR camera, some 3000x4000 and higher.	
Minimum image size	240 x 240	
Eye to eye distance pixels	20 to 1000 approximately	
Pose	The images have compound roll, pitch and yaw rotations.	
Full frontal geometry	Rarely	
Intended use	All CHEXIA-FACE tasks	
Age	Many below 10, some to age 18. Rarely an adult.	

NOTE on Child exploitation images: These images are illicit pornographic images and video. The images are present on digital media seized in criminal investigations. The files include children who range in age from infant through adolescent. In addition a few adult faces sometimes occur also. Some of the images are innocuous "family photographs". The majority, however, feature coercion, abuse, and sexual activity.

From a face recognition viewpoint, the images will be difficult for the following reasons (in order): highly variable pose (including adverse compound roll, pitch and yaw); occlusion (by hair, other persons, body parts and objects); variable and directional lighting; evidence that face recognition in children is difficult even with cooperative photographs – see Annex B below which excerpts [NIST8009].

1.5. Offline testing

While CHEXIA-FACE is intended as much as possible to mimic operational reality, this remains an offline test executed on databases of images. The intent is to assess the core algorithmic capability of face detection, recognition and clustering algorithms. This test does not include a live human-presents-to-camera component. Offline testing is attractive because it allows uniform, fair, repeatable, and efficient evaluation of the underlying technologies. Testing of implementations under a fixed API allows for a detailed set of performance related parameters to be measured. The algorithms will be run only on NIST machines by NIST employees.

1.6. Phased testing

To support development, CHEXIA-FACE will run in three phases. In each phase, NIST will evaluate implementations on a first-come-first-served basis and will return results to providers as expeditiously as possible. The final phase will result in the release of public reports. Providers should not submit revised algorithms to NIST until NIST provides results for the prior phase.

For the schedule and number of algorithms of each class that may be submitted, see sections 1.10 and 1.11.

[1] Compression effects were studied under MBE 2010 in NIST Interagency Report 7830, linked from http://face.nist.gov/mbe

1.7. Interim reports

The performance of each implementation will be reported in a "score-card". This will be provided to the participant. It is intended to facilitate research and development, not for marketing. Score cards will: be machine generated (i.e. scripted); be provided to participants with identification of their implementation, include timing, accuracy and other performance results, include results from other implementations, but will not identify the other providers; be expanded and modified as revised implementations are tested, and as analyses are implemented; be produced independently of the status of other providers' implementations; be regenerated on-the-fly, usually whenever any implementation completes testing, or when new analysis is added.

NIST does not intend to release these test reports publicly. NIST may release such information to the U.S. Government test sponsors; NIST will request that agencies not release this content.

1.8. Final reports

NIST will publish one or more final public reports. NIST may also publish: additional supplementary reports (typically as numbered NIST Interagency Reports); in other academic journals; in conferences and workshops (typically PowerPoint).

Our intention is that the final test reports will publish results for the best-performing implementation from each participant. Because "best" is under-defined (accuracy vs. time vs. template size, for example), the published reports may include results for other implementations. The intention is to report results for the most capable implementations (see section 1.12, on metrics). Other results may be included (e.g. in appendices) to show, for example, examples of progress or tradeoffs.

IMPORTANT: All Phase 3 results will be attributed to the providers.

IMPORTANT: Phase 1 and Phase 2 results will be attributed to the providers UNLESS, ahead of the Phase 3 submission deadline, the participant emails NIST to request their organization name should NOT appear in the CHEXIA public reports and presentations. In that case the quantitative results will still appear in the published report but without any appearance of the participant's name. This provision is being included in this evaluation because NIST understands that this is a new and difficult application of face recognition technology.

1.9. Application scenarios

The test will include one-to-one verification and one-to-many identification tests [MBE 2010, NIST8009] for still images and video clips. As described in Table 2, the test is intended to represent:

- Close-to-operational use of face recognition technologies in identification applications in which the enrolled dataset could contain images in the hundreds of thousands.
- Verification scenarios in which samples are compared.
- Face detection in stills and videos with one or more persons in the sample.
- Grouping (clustering) identities in mixed media.

Table 2 – Subtests supported under the CHEXIA-FACE activity

#		A	C	D	G
1.	Aspect	1:1 verification	1:N identification	Detection	Clustering
2.	Enrollment dataset	None (applies to single samples; there is no concept of gallery or enrollment database)	N enrolled subjects	None, application to single images	The concepts of enrollment and search sets do not exist
3.	Prior NIST test references	Equivalent to 1 to 1 matching in [MBE 2010]	Equivalent to 1 to N matching in [NIST 8009]		
4.	Example application	Verification of e-Passport facial image against a live border-crossing image.	Open-set identification of an image against a central database, e.g. a search of a mugshot against a database of known criminals.	Often used in conjunction with face recognition; also used in video surveillance, human computer interaction, and image	Assign images to groups if they contain the same individual. Given many images or videos containing many individuals, produce as many clusters as there are unique individuals, and associate which

				database management.	images they appear in.
5.	Score or feature space normalization support	If any, normalization techniques are only possible over datasets internal to the implementation.	Any score or feature based statistical normalization techniques are applied against enrollment database		Any score or feature based statistical normalization techniques-are applied internally
6.	Intended number of subjects	Up to O(10^4)	Up to O(10^5) but dependence on N will be computed. From O(10^2) upwards.	Expected O(10^4)	Expected O(10^3)
7.	Number of images per individual	Variable: one or more still images, or a video clip	Variable: one or more still images, or a video clip	Variable	Variable
8	Number of persons in one sample	1	1 or more persons	1 or more persons	1 or more persons

NOTE 1: The vast majority of images are color. The API supports both color and greyscale images.

NOTE 2: For the operational datasets, it is not known what processing was applied to the images before they were archived. So, for example, we do not know whether gamma correction was applied. NIST considers that best practice, standards and operational activity in the area of image preparation remains weak.

1.10. Options for participation

The following rules apply:

- A participant must properly follow, complete and submit the Annex A Participation Agreement. This must be done once, not before January 1, 2016. It is not necessary to do this for each submitted implementation.
- All participants shall submit at least one class D algorithm.
- Class A (1:1) algorithms may be submitted only if at least 1 class D (detection) algorithm is also submitted.
- Class C (1:N) algorithms may be submitted only if at least 1 class A (1:1) algorithm is also submitted.
- Class G (clustering) algorithms may be submitted only if at least 1 class C (1:N) algorithm is also submitted.
- Class D (detection) algorithms may be submitted alone, without submission to any other classes of participation.
- All submissions shall implement exactly one of the functionalities defined in Table 3. A library shall not implement the API of more than one class.

Table 3 – CHEXIA-FACE classes of participation

Function	1:1 verification	1:N identification	Detection	Clustering
Class label	A	C	D	G
Co-requisite class	D	D + A	None	D + A + C
API requirements	3.1	3.2	3.3	3.4

1.11. Number and schedule of submissions

The test is conducted in three phases, as scheduled on page 2. The maximum total (i.e. cumulative) number of submissions is regulated in Table 4.

Table 4 – Cumulative total number of algorithms, by class

#	Phase 1	Total over Phases 1 + 2	Total over Phases 1 + 2 + 3
Class A : Verification	2	4	6 if at least 1 was successfully executed by end Phase 1 2 otherwise
Class C : Identification	2	4	6 if at least 1 was successfully executed by end Phase 1

			2 otherwise
Class D : Detection	2	2	3 if at least 1 was successfully executed by end Phase 1 1 otherwise
Class G : Clustering	2	2	4 if at least 1 was successfully executed by end Phase 1 2 otherwise

The numbers above may be increased as resources allow.

NIST cannot conduct surveys over runtime parameters – essentially to limit the extent to which participants are able to train on the test data.

1.12. Core accuracy metrics

Notionally the error rates for verification applications will be false match and false non-match error rates, FMR and FNMR. These will be modified to include the effects of failure to make a template.

For identification testing, the test will target open-universe applications such as watch-lists. It will not address the closed-set task because it is operationally uncommon. Metrics include false positive and negative identification rate (FPIR and FNIR) that depend on threshold and rank.

Rank-based metrics are appropriate for one-to-many applications that employ human examiners to adjudicate candidate lists. Score based metrics are appropriate for cases where transaction volumes are too high for human adjudication or when false alarm rates must otherwise be low. See [NIST8009].

1.13. Reporting template size

Because template size is influential on storage requirements and computational efficiency, this API supports measurement of template size. NIST will report statistics on the actual sizes of templates produced by face recognition implementations submitted to CHEXIA-FACE. NIST may report statistics on runtime memory usage. Template sizes were reported in the FRVT 2012 test[2], IREX III test[3] and the MBE-STILL 2010 test[4].

1.14. Reporting computational efficiency

As with other tests, NIST will compute and report recognition accuracy. In addition, NIST will also report timing statistics for all core functions of the submitted implementations. This includes feature extraction, 1:1 and 1:N recognition, detection, and clustering. For an example of how efficiency can be reported, see the final report of the FRVT 2012 test [NIST8009][2], and the MBE-STILL 2010 test[4].

Note that face recognition applications optimized for pipelined 1:N searches may not demonstrate their efficiency in pure 1:1 comparison applications.

1.15. Exploring the accuracy-speed trade-space

NIST will explore the accuracy vs. speed tradeoff for face recognition algorithms running on a fixed platform. NIST will report both accuracy and speed of the implementations tested. While NIST cannot force submission of "fast vs. slow" variants, participants may choose to submit variants on some other axis (e.g. "experimental vs. mature") implementations. NIST encourages “fast-less-accurate vs. slow-more-accurate” with a factor of three between the speed of the fast and slow versions.

1.16. Hardware specification

NIST intends to support high performance by specifying the runtime hardware beforehand. There are several types of computer blades that may be used in the testing. The following list gives some details about the hardware of each blade type:

- Dell M610 - Dual Intel Xeon X5680 3.3 GHz CPUs (6 cores each)
- Dell M910 - Dual Intel Xeon X7560 2.3 GHz CPUs (8 cores each)

[2] See the FRVT 2012 test report: NIST Interagency Report 8009, linked from http://www.nist.gov/itl/iad/ig/frvt-2013.cfm

[4] See the MBE-STILL 2010 test report, NIST Interagency Report 7709, linked from http://face.nist.gov/mbe

- Dual Intel Xeon E5-2695 3.3 GHz CPUs (14 cores each; 56 logical CPUs total) with Dual NVIDIA Tesla K40 GPUs

NOTE: Implementations must be functional on machines with and without GPU capability.

Each CPU has 512K cache. The bus runs at 667 Mhz. The main memory is 192 GB Memory as 24 8GB modules. We anticipate that 16 processes can be run without time slicing.

NIST is requiring use of 64 bit implementations throughout. This will support large memory allocation to support 1:N identification task. For video, given the data expectations and the occurrence of faces in the imagery, we anticipate the developers will have sufficient memory for video templates. Note that while the API allows read access of the disk during the 1:N search, the disk is, of course, relatively slow.

Some of the section 3 API calls allow the implementation to write persistent data to hard disk. The amount of data shall not exceed 200 kilobytes per enrolled image. NIST will respond to prospective participants' questions on the hardware, by amending this section.

1.17. Operating system, compilation, and linking environment

The operating system that the submitted implementations shall run on will be released as a downloadable file accessible from http://nigos.nist.gov:8080/evaluations/ which is the 64-bit version of CentOS 7 running Linux kernel 3.10.0.

For this test, Windows machines will not be used. Windows-compiled libraries are not permitted. All software must run under Linux.

NIST will link the provided library file(s) to our C++ language test drivers. Participants are required to provide their library in a format that is linkable using the C++11 compiler, g++ version 4.8.3.

A typical link line might be

```
g++ -std=C++11 -I. -Wall -m64 -o chexiaface chexiaface.cpp  -L.  –lchexiaface_Enron_A_07
```

The Standard C++ library should be used for development. The prototypes from this document will be written to a file "chexiaface.h" which will be included via

```
#include <chexiaface.h>
```

The header files will be made available to implementers at http://nigos.nist.gov:8080/chexia-face/

NIST will handle all input of images via the JPEG and PNG libraries, sourced, respectively from http://www.ijg.org/ and see http://libpng.org.

All compilation and testing will be performed on x86 platforms. Thus, participants are strongly advised to verify library-level compatibility with g++ (on an equivalent platform) prior to submitting their software to NIST to avoid linkage problems later on (e.g. symbol name and calling convention mismatches, incorrect binary file formats, etc.).

Dependencies on external dynamic/shared libraries such as compiler-specific development environment libraries are discouraged. If absolutely necessary, external libraries must be provided to NIST upon prior approval by the Test Liaison.

1.18. Software and Documentation

1.18.1. Library and Platform Requirements

Participants shall provide NIST with binary code only (i.e. no source code). Header files (".h") are allowed, but these shall not contain intellectual property of the company nor any material that is otherwise proprietary. The SDK should be submitted in the form of a dynamically linked library file.

The core library shall be named according to Table 5. Additional shared object library files may be submitted that support this "core" library file (i.e. the "core" library file may have dependencies implemented in these other libraries).

Intel Integrated Performance Primitives (IPP) libraries are permitted if they are delivered as a part of the developer-supplied library package. It is the provider's responsibility to establish proper licensing of all libraries. The use of IPP libraries shall not prevent run on CPUs that do not support IPP. Please take note that some IPP functions are multithreaded and threaded implementations may complicate comparative timing.

Table 5 – Implementation library filename convention

Form	libCHEXIAFACE_provider_class_sequence.ending				
Underscore delimited parts of the filename	libCHEXIAFACE	provider	class	sequence	ending
Description	First part of the name, required to be this.	Single word name of the main provider EXAMPLE: Acme	Function classes supported in Table 3. EXAMPLE: C	A two digit decimal identifier to start at 00 and increment by 1 every time a library is sent to NIST. EXAMPLE: 07	.so
Example	libCHEXIAFACE_Acme_C_07.so				

NIST will report the size of the supplied libraries.

1.18.2. Configuration and developer-defined data

The implementation under test may be supplied with configuration files and supporting data files. NIST will report the size of the supplied configuration files.

1.18.3. Submission folder hierarchy

Participant submissions should contain the following folders at the top level

- lib/ - contains all participant-supplied software libraries
- config/ - contains all configuration and developer-defined data
- doc/ - contains any participant-provided documentation regarding the submission

1.18.4. Installation and Usage

The implementation must install easily (i.e. one installation step with no participant interaction required) to be tested, and shall be executable on any number of machines without requiring additional machine-specific license control procedures or activation.

The implementation shall be installable using simple file copy methods. It shall not require the use of a separate installation program.

The implementation shall not use nor enforce any usage controls or limits based on licenses, number of executions, presence of temporary files, etc. It shall remain operable with no expiration date.

Hardware (e.g. USB) activation dongles are not acceptable.

1.18.5. Documentation

Participants shall provide documentation of additional functionality or behavior beyond that specified here. The documentation must define all (non-zero) developer-defined error or warning return codes.

1.18.6. Modes of operation

Implementations shall not require NIST to switch “modes” of operation or algorithm parameters. For example, the use of two different feature extractors must either operate automatically or be split across two separate library submissions.

1.19. Runtime behavior

1.19.1. Interactive behavior, stdout, logging

The implementation will be tested in non-interactive “batch” mode (i.e. without terminal support). Thus, the submitted library shall:

– Not use any interactive functions such as graphical user interface (GUI) calls, or any other calls which require terminal interaction e.g. reads from “standard input”.

- Run quietly, i.e. it should not write messages to "standard error" and shall not write to "standard output".
- If requested by NIST for debugging, include a logging facility in which debugging messages are written to a log file whose name includes the provider and library identifiers and the process PID.

1.19.2. Exception Handling

The application should include error/exception handling so that in the case of a fatal error, the return code is still provided to the calling application.

1.19.3. External communication

Processes running on NIST hosts shall not side-effect the runtime environment in any manner, except for memory allocation and release. Implementations shall not write any data to external resource (e.g. server, file, connection, or other process), nor read from such. If detected, NIST will take appropriate steps, including but not limited to, cessation of evaluation of all implementations from the supplier, notification to the provider, and documentation of the activity in published reports.

1.19.4. Stateless behavior

All components in this test shall be stateless, except as noted. Thus, all functions should give identical output, for a given input, independent of the runtime history. NIST will institute appropriate tests to detect stateful behavior. If detected, NIST will take appropriate steps, including but not limited to, cessation of evaluation of all implementations from the supplier, notification to the provider, and documentation of the activity in published reports.

1.20. Threaded computations

Table 6 shows the limits on the numbers of threads an implementation may use for each of the classes of participation. In many cases multithreading is not permitted (i.e. T=1) because NIST will parallelize the test by dividing the workload across many cores and many machines.

Table 6 – Number of threads allowed for each application

	A	C	D	G
Function	1:1 verification	1:N identification	Detection	Clustering
Feature extraction	1	1	1	1 ≤ T ≤ 16
Verification	1	NA		
Finalize enrollment (before 1:N)	NA	1 ≤ T ≤ 16		
Identification	NA	1		

For comparative timing, the IREX III[3] test report estimated a factor by which the speed of threaded algorithms would be adjusted. Non-threaded implementations will eliminate the need for NIST to apply such techniques [IREX III].

NIST will not run implementations from participants X and Y on the same machine at the same time.

To expedite testing, for single-threaded libraries, NIST will run $P > 2$ processes concurrently. NIST's calling applications are single-threaded.

1.21. Time limits

The elemental functions of the implementations shall execute under the time constraints of Table 7. These time limits apply to the function call invocations defined in section 3. Assuming the times are random variables, NIST cannot regulate the maximum value, so the time limits are 90-th percentiles. This means that 90% of all operations should take less than the identified duration.

The time limits apply per image or video frame. When K images of a person are present, the time limits shall be increased by a factor K.

Table 7 – Processing time limits in milliseconds, per 640 x 480 image

	A	C	D	G

Function	1:1 verification	1:N identification	Detection	Clustering
Feature extraction enrollment	1000 (1 core) 600x480 pixels	1000 (1 core) 600x480 pixels	K*500 (1 core), where K=number of persons in the image	K*1000 (1 core) where K=number of persons in the image. See NOTE.
Feature extraction for verification or identification	1000 (1 core) 600x480 pixels	1000 (1 core) 600x480 pixels		
Verification	5 (1 core)	NA		
Identification of one search image against 1,000,000 single-image Multiface records.	NA	10000 (16 cores) or 160000 (1 core)		
Enrollment finalization of 1,000,000 single-image Multiface records (including disk IO time)	NA	7,200,000 (up to 16 cores)	NA	NA

NOTE: NIST anticipates that the duration of clustering calls will have linear and quadratic components, for template generation and matching respectively. We will assess compliance with our time limit requirements based on small clustering tasks where template generation duration dominates the total.

1.22. Ground truth integrity

Some of the test data is derived from operational systems and may contain ground truth errors in which

— a single person is present under two different identifiers, or

— two persons are present under one identifier, or

— in which a face is not present in the image.

If these errors are detected, they will be removed. NIST will use aberrant scores (high impostor scores, low genuine scores) to detect such errors. This process will be imperfect, and residual errors are likely. For comparative testing, identical datasets will be used and the presence of errors should give an additive increment to all error rates. For very accurate implementations this will dominate the error rate. NIST intends to attach appropriate caveats to the accuracy results. For prediction of operational performance, the presence of errors gives incorrect estimates of performance.

2. Data structures supporting the API

2.1. Namespace

All data structures and API interfaces/function calls will be declared in the `CHEXIAFACE` namespace.

2.2. Overview

This section describes separate APIs for the core face recognition applications described in section 1.9. All submissions to CHEXIA-FACE shall implement the functions required by the rules for participation listed before Table 3.

2.3. Requirement

CHEXIA-FACE participants shall implement the relevant C++ prototyped interfaces of section 3. C++ was chosen in order to make use of some object-oriented features.

2.4. File formats and data structures

2.4.1. Overview

In this face recognition test, an individual is represented by K ≥ 1 two-dimensional facial images (which may be video frames), and by subject and image-specific metadata.

2.4.2. Data structures for encapsulating multiple images or video frames

Some of the proposed datasets includes K > 2 images per person for some persons. This affords the possibility to model a recognition scenario in which a new image of a person is compared against all prior images[5]. Use of multiple images per person has been shown to elevate accuracy over a single image [MBE 2010].

For still-face recognition in this test, NIST will enroll K ≥ 1 images under each identity. Both enrolled gallery and probe samples may consist of multiple images such that a template is the result of applying feature extraction to a set of K ≥ 1 images and then integrating information from them. An algorithm might fuse K feature sets into a single model or might simply maintain them separately - In any case the resulting proprietary template is contained in a contiguous block of data. All verification and identification functions operate on such multi-image templates.

The number of images per person will vary, and images may not be acquired uniformly over time. NIST currently estimates that the number of images K will never exceed 1000. For the CHEXIA-FACE API, K images of an individual are contained in data structure of Table 13. Each file contains a standardized image format, e.g. PNG (lossless) or JPEG (lossy).

NOTE: For the 1:1 verification task, all images will contain one and only one face. For all other CHEXIA-FACE tasks, images in the test may contain one or more faces in an image.

Table 8 – Structure for a single image or video frame

	C++ code fragment	Remarks
1.	`struct Image`	
2.	`{`	
3.	`    uint16_t width;`	Number of pixels horizontally
4.	`    uint16_t height;`	Number of pixels vertically
5.	`    uint16_t depth;`	Number of bits per pixel. Legal values are 8 and 24.
6.	`    uint8_t format;`	Flag indicating native format of the image as supplied to NIST 0x01 = JPEG (i.e. compressed data) 0x02 = PNG (i.e. never compressed data)
7.	`    uint8_t *data;`	Pointer to raster scanned data. Either RGB color or intensity. If image_depth == 24 this points to 3WH bytes RGBRGBRGB... If image_depth == 8 this points to WH bytes `IIIIIII`
8.	`};`	

Table 9 – Structure for a set of images or video frames

	C++ code fragment	Remarks

1.	`struct Multiface`
	`{`	

2.	`    typedef std::vector<Image> images;`	Vector containing F pre-allocated face images. The number of items is accessible via the `vector::size()` function.
3.	`    MultifaceLabel description;`	Single description of the `Multiface`. The allowed values for this field are specified in the enumeration in Table 10.
4.	`    uint16_t framesPerSec;`	The frame rate of the video sequence in frames-per-second. Only defined if `description==Video`; otherwise set to -1.
5.	`};`	

A **Multiface** will be accompanied by one of the labels given below. Face recognition implementations submitted to CHEXIA-FACE should tolerate **Multifaces** of any category.

Table 10 – Labels describing categories of Multifaces

	Label as C++ enumeration	Meaning

[5] For example, if a child is subject to a new exploitation event then imagery from that event can be searched against a database of all prior instances of exploitation, including from that child.

1.	enum class MultifaceLabel {	
2.	Unknown=0,	Either the label is unknown or unassigned.
3.	Fixed=1,	Images are still photos from a non-moving camera.
4.	Event=2,	Images are still photos from the same event, possibly from several cameras or with camera movement.
5.	Group=3,	Still photos from one person on arbitrary occasions.
6.	Video=4	Images are a sequence of video frames.
7.	};	

2.4.3. Data structure for eye coordinates

Implementations should return eye coordinates of each facial image. This function, while not necessary for a recognition test, will assist NIST in assuring the correctness of the test database. The primary mode of use will be for NIST to inspect images for which eye coordinates are not returned, or differ between implementations.

The eye coordinates shall follow the placement semantics of the ISO/IEC 19794-5:2005 standard - the geometric midpoints of the endocanthion and exocanthion (see clause 5.6.4 of the ISO standard).

Sense: The label "left" refers to subject's left eye (and similarly for the right eye), such that xright < xleft.

Table 11 – Structure for a pair of eye coordinates

	C++ code fragment	Remarks
1.	`struct EyePair`	
2.	`{`	
3.	`bool isLeftAssigned;`	If the subject's left eye coordinates have been computed and assigned successfully, this value should be set to true, otherwise false.
4.	`bool isRightAssigned;`	If the subject's right eye coordinates have been computed and assigned successfully, this value should be set to true, otherwise false.
5.	`int16_t xleft;`	X and Y coordinate of the center of the subject's left eye. Out-of-range values (e.g. x < 0 or x >= width) indicate the implementation believes the eye center is outside the image.
6.	`int16_t yleft;`	
7.	`int16_t xright;`	X and Y coordinate of the center of the subject's right eye. Out-of-range values (e.g. x < 0 or x >= width) indicate the implementation believes the eye center is outside the image.
8.	`int16_t yright;`	
9.	`uint16_t frameNum`	For `Multiface`s where `description==Video`, this would be the frame number that corresponds to the video frame from which the eye coordinates were generated. (i.e., the i-th frame from the video sequence). This field should not be set for eye coordinates for a single still image.
10.	`};`	

2.4.4. Data type for representing eye coordinates from a Multiface

Table 12 – PersonTrajectory typedef

	C++ code fragment	Remarks
1.	`using PersonTrajectory = std::vector<EyePair>;`	Vector of `EyePair` objects for a `Multiface` where eyes were detected. This data structure should store eye coordinates for each video frame or image where eyes were detected for a particular person. For `Multiface`s where the person's eyes were not detected, the SDK shall not add an `EyePair` to this data structure. If a face can be detected, but not the eyes, the implementation should nevertheless fill this data structure with $(x,y)_{LEFT} == (x,y)_{RIGHT}$ representing some point on the center of the face.

2.4.5. Class for representing a person detected in a Multiface

Table 13 - Class for representing a person

	C++ code fragment	Remarks
1.	`class PersonRep`	

2.	`{`
	`private:`	

3.	`PersonTrajectory eyeCoordinates;`	Data structure for capturing eye coordinates
4.	`PersonTemplate proprietaryTemplate;`	`PersonTemplate` is a wrapper to a uint8_t* for capturing proprietary template data representing a person from a `Multiface`.
5.	`uint64_t templateSize;`	Size of PersonTemplate
6.	`public:`	
	`PersonRep();`	Default constructor
7.	`void pushBackEyeCoord(const EyePair &eyes);`	This function should be used to add `EyePair`s for the video frames or images where eye coordinates were detected.
~~8.~~	~~`void setTemplate(PersonTemplate templ, uint64_t size);`~~	~~This function should be used to set the template data. After the implementation calls setTemplate(), they should not attempt to modify or delete the template data that the managed pointer refers to.~~
8.	`std::shared_ptr<uint8_t> resizeTemplate(uint64_t size);`	This function takes a size parameter and allocates memory of *size* and returns a managed pointer to the newly allocated memory for SDK manipulation. Please note that this class will take care of all memory allocation and de-allocation of its own memory. The SDK shall not de-allocate memory created by this class.
9.	`std::shared_ptr<uint8_t> getPersonTemplatePtr();`	This function returns a managed pointer to uint8_t to the template data.
10.	`uint64_t getPersonTemplateSize() const;`	This function returns the size of the template data.

11.	`//… getter methods, copy constructor,`
	`//… assignment operator`	

12.	`};`	

2.4.6. Data Structure for detected face

For face detection, implementations shall return bounding box coordinates of each detected face in an image. See section 3.3 for API details.

Table 14 – Structure for bounding box around a detected face

	C++ code fragment	Remarks

1.	`struct BoundingBox`
	`{`	

2.	`uint16_t x;`	x-coordinate of **top-left** corner of bounding box around face
3.	`uint16_t y;`	y-coordinate of top-left corner of bounding box around face
4.	`uint16_t width;`	width, in pixels, of bounding box around face
5.	`uint16_t height;`	height, in pixels, of bounding box around face
6.	`double confidence;`	Higher value indicates more certainty that this region contains a face
7.	`};`	

2.4.7. Data Structure for hypothesized cluster membership

For clustering, implementations shall assign image samples to clusters using the structure of Table 15. See section 3.4 for API details.

Table 15 – Structure for a single hypothesized cluster membership

	C++ code fragment	Remarks
1.	`struct ClusterMember`	
2.	`{`	
3.	`    uint32_t clusterId;`	Non-negative integer assigned by the implementation indicating a cluster label. All images of a person should share this same integer.
4.	`    double similarityScore;`	Measure of similarity between the input sample and other images of the same cluster (presumably of the same person). The score should be a non-negative value on a continuous range. The score will be used to perform threshold-based analysis of algorithm performance. For example, NIST may perform analyses that assigns samples to clusters only when their corresponding similarly score is at or above a threshold.
5.	`    BoundingBox face;`	Bounding box coordinates corresponding to the face/identity belonging to `clusterId` from the image.
6.	`};`	

Table 16 – Structure for hypothesized cluster membership for face(s) in an image

	C++ code fragment	Remarks
1.	`struct ClusterMembersInImage`	
2.	`{`	
3.	`    ReturnStatus status;`	`status.code` should be set as follows: `ReturnCode::Success` if the input sample was processed successfully `ReturnCode::NoFaceError` if no faces could be detected in the input `ReturnCode::ExtractError` if the software failed to process the image `ReturnCode::RefuseInput` if the software detects malformed input
4.	`    std::vector<ClusterMember> members;`	For each person found in an image, the list of hypothesized cluster membership(s).
5.	`};`	

2.4.8. Data structure for result of an identification search

All identification searches shall return a candidate list of a NIST-specified length. The list shall be sorted with the most similar matching entries list first with lowest rank. The data structure shall be that of Table 17. See section 3.2 for API details.

Table 17 – Structure for a candidate

	C++ code fragment	Remarks
1.	`struct Candidate`	
2.	`{`	
3.	`    bool isAssigned;`	If the candidate is valid, this should be set to true. If the candidate computation failed, this should be set to false.
4.	`    std::string templateId;`	The Template ID from the enrollment database manifest defined in section 2.5.
5.	`    double similarityScore;`	Measure of similarity between the identification template and the enrolled candidate. Higher scores mean more likelihood that the samples are of the same person. An algorithm is free to assign any value to a candidate. The distribution of values will have an impact on the appearance of a plot of false-negative and false-positive identification rates.
6.	`};`	

2.4.9. Data structure return value of API function calls

Table 18 – Enumeration of return codes

	Return code as C++ enumeration	Meaning
	`enum class ReturnCode {`	
1.	`    Success=0,`	Success

`ConfigError=1,`	Error reading configuration files
`RefuseInput=2,`	Elective refusal to process the input
`ExtractError=3,`	Involuntary failure to process the image
`ParseError=4,`	Cannot parse the input data
`TemplateCreationError=5,`	Elective refusal to produce a template (e.g. insufficient pixels between the eyes)
`VerifTemplateError=6,`	For matching, either or both of the input templates were result of failed feature extraction
`EnrollDirError=7,`	An operation on the enrollment directory failed (e.g. permission, space)
`NumDataError=8,`	The SDK cannot support the number of persons or images
`TemplateFormatError=9,`	One or more template files are in an incorrect format or defective
`InputLocationError=10,`	Cannot locate the input data - the input files or names seem incorrect
`NoFaceError=11,`	Cannot detect face in image
`VendorError=12`	Vendor-defined failure. Failure codes must be documented and communicated to NIST with the submission of the implementation under test.
`};`	

Table 19 – ReturnStatus structure

C++ code fragment	Meaning
`struct ReturnStatus {`	
`    CHEXIAFACE::ReturnCode code;`	Return Code
`    std::string info;`	Optional information string
`    // constructors`	
`};`	

2.4.10. Data type for similarity scores

Identification and verification functions shall return a measure of the similarity between the face data contained in the two templates. The datatype shall be an eight byte double precision real. The legal range is [0, DBL_MAX], where the DBL_MAX constant is larger than practically needed and defined in the <limits.h> include file. Larger values indicate more likelihood that the two samples are from the same person.

Providers are cautioned that algorithms that natively produce few unique values (e.g. integers on [0,127]) will be disadvantaged by the inability to set a threshold precisely, as might be required to attain a false match rate of exactly 0.0001, for example.

2.5. File structures for enrolled template collection

An SDK converts a `Multiface` into a template, using, for example the `convertMultifaceToEnrollmentTemplate` function of section 3.2.2.3. To support the class C identification functions of Table 3, NIST will concatenate enrollment templates into a single large file, the EDB (for enrollment database). The EDB is a simple binary concatenation of proprietary templates. There is no header. There are no delimiters. The EDB may be hundreds of gigabytes in length.

This file will be accompanied by a manifest; this is an ASCII text file documenting the contents of the EDB. The manifest has the format shown as an example in Table 20. If the EDB contains N templates, the manifest will contain N lines. The fields are space (ASCII decimal 32) delimited. There are three fields. Strictly speaking, the third column is redundant.

Important: If a call to the template generation function fails, or does not return a template, NIST will include the Template ID in the manifest with size 0. Implementations must handle this appropriately.

Table 20 – Enrollment dataset template manifest

Field name	Template ID	Template Length	Position of first byte in EDB
Datatype required	std::string	Unsigned decimal integer	Unsigned decimal integer
Datatype length required		4 bytes	8 bytes

Example lines of a manifest file appear to the right. Lines 1, 2, 3 and N appear.	90201744	1024	0
	person01	1536	1024
	7456433	512	2560
	...		
	subject12	1024	307200000

The EDB scheme avoids the file system overhead associated with storing millions of individual files.

3. API Specification

3.1. 1:1 Verification

3.1.1. Overview

The 1:1 testing will proceed in three phases: preparation of enrollment templates; preparation of verification templates; and matching. These are detailed in Table 21.

Table 21 – Functional summary of the 1:1 application

Phase	Description	Performance Metrics to be reported by NIST
Initialization	Function to read configuration data, if any.	None
Enrollment	Given K ≥ 1 input images of an individual, the implementation will create a proprietary enrollment template. NIST will manage storage of these templates.	Statistics of the time needed to produce a template. Statistics of template size. Rate of failure to produce a template
Verification	Given K ≥ 1 input images of an individual, the implementation will create a proprietary verification template. NIST will manage storage of these templates.	Statistics of the time needed to produce a template. Statistics of template size. Rate of failure to produce a template.
Matching (i.e. comparison)	Given a proprietary enrollment and a proprietary verification template, compare them to produce a similarity score.	Statistics of the time taken to compare two templates. Accuracy measures, primarily reported as DETs.

NIST requires that these operations may be executed in a loop in a single process invocation, or as a sequence of independent process invocations, or a mixture of both.

3.1.2. API

3.1.2.1. Interface

The Class A 1:1 verification software under test must implement the interface `VerifInterface` by subclassing this class and implementing each method specified therein. See

	C++ code fragment	Remarks

1.	`class VerifInterface;`
	
	`typedef std::shared_ptr<VerifInterface> ClassAImplPtr;`
	
	`class VerifInterface`	

2.	`{`
	`public:`	

3.	`    virtual ReturnStatus initializeVerification(`
	`        const std::string &configurationLocation ) = 0;`	

4.	`    virtual ReturnStatus convertMultifaceToEnrollmentTemplate(`
	`         const Multiface &inputFaces,`
	`         PersonRep &templ) = 0;`	

	`    virtual ReturnStatus convertMultifaceToVerificationTemplate(`
	`         const Multiface &inputFaces,`
	`         PersonRep &templ) = 0;`	

5.	`    virtual ReturnStatus matchTemplates(`
	`        const uint8_t *verificationTemplate,`
	`        const uint32_t verificationTemplateSize,`
	`        const uint8_t *enrollmentTemplate,`
	`        const uint32_t enrollmentTemplateSize,`
	`        double &similarity) = 0;`	

6.	`    virtual void setGPU(uint8_t gpuNum) = 0;`	
7.	`    static ClassAImplPtr getImplementation();`	Factory method to return a managed pointer to the `VerifInterface` object. This function is implemented by the submitted library and must return a managed pointer to the `VerifInterface` object.
8.	`};`	

There is one class (static) method declared in `VerifInterface`, `getImplementation()` which must also be implemented by the SDK. This method returns a shared pointer to the object of the interface type, an instantiation of the implementation class. A typical implementation of this method is also shown below as an example.

	C++ code fragment	Remarks

	`#include "chexiafaceNullImplClassA.h"`
	
	`using namespace CHEXIAFACE;`
	
	`NullImplClassA::NullImplClassA() { }`
	
	`NullImplClassA::~NullImplClassA() { }`
	
	`ClassAImplPtr`
	`VerifInterface::getImplementation()`
	`{`
	`        NullImplClassA *p = new NullImplClassA();`
	`        ClassAImplPtr ip(p);`
	`        return (ip);`
	`}`
	
	`// Other implemented functions`	

3.1.2.2. Initialization

The NIST test harness will call the initialization function in Table 22 before calling template generation or matching.

Table 22 – Initialization

Prototype	ReturnStatus initializeVerification(const std::string &configurationLocation);	 Input
Description	This function initializes the SDK under test. It will be called by the NIST application before any call to the Table 24 functions `convertMultifaceToEnrollmentTemplate` or `convertMultifaceToVerificationTemplate`. The implementation under test should set all parameters.	
Input Parameters	configurationLocation	A read-only directory containing any developer-supplied configuration parameters or run-time data files. The name of this directory is assigned by NIST, not hardwired by the provider. The names of the files in this directory are hardwired in the implementation and are unrestricted.
Output Parameters	none	
Return Value	See Table 18 for all valid return code values.	

3.1.2.3. GPU Index Specification

For implementations using GPUs, the function of Table 23 specifies a sequential index for which GPU device to execute on. This enables the test software to orchestrate load balancing across multiple GPUs.

Table 23 – GPU index specification

Prototypes	void setGPU (	
	uint8_t gpuNum);	Input
Description	This function sets the GPU device number to be used by all subsequent implementation function calls. `gpuNum` is a zero-based sequence value of which GPU device to use. 0 would mean the first detected GPU, 1 would be the second GPU, etc. If the implementation does not use GPUs, then this function call should simply do nothing.	
Input Parameters	gpuNum	Index number representing which GPU to use.

3.1.2.4. Template generation

The functions of Table 24 support role-specific generation of template data. Template format is entirely proprietary.

Table 24 – Template generation

Prototypes	ReturnStatus convertMultifaceToEnrollmentTemplate(	
	const Multiface &inputFaces,	Input
	PersonRep &templ);	Output
	int32_t convertMultifaceToVerificationTemplate(	
	const Multiface &inputFaces,	Input
	PersonRep &templ);	Output
Description	Takes a `Multiface` and populates a `PersonRep` object. In all cases, even when unable to extract features, the template generated for the `PersonRep` should be a template that may be passed to the `matchTemplates` function without error. That is, this routine must internally encode "template creation failed" and the matcher must transparently handle this.	
Input Parameters	inputFaces	Implementations must alter their behavior according to the number of images contained in the structure, and the types per Table 10.
Output Parameters	templ	A `PersonRep` object that represents a single template generated from the `Multiface`.
Return Value	See Table 18 for all valid return code values.	

3.1.2.5. Matching

Matching of one enrollment against one verification template shall be implemented by the function of Table 25.

Table 25 – Template matching

Prototype	ReturnStatus matchTemplates(	
	const uint8_t *verificationTemplate,	Input
	const uint32_t verificationTemplateSize,	Input
	const uint8_t *enrollmentTemplate,	Input
	const uint32_t enrollmentTemplateSize,	Input
	double &similarity);	Output
Description	Compare two proprietary templates and output a similarity score, which need not satisfy the metric properties. When either or both of the input templates are the result of a failed template generation (see Table 24), the similarity score shall be -1 and the function return value shall be `VerifTemplateError`.	
Input Parameters	verificationTemplate	A template generated from a call to `convertMultifaceToVerificationTemplate()`.
	verificationTemplateSize	The size, in bytes, of the input verification template $0 \leq N \leq 2^{32} - 1$
	enrollmentTemplate	A template generated from a call to `convertMultifaceToEnrollmentTemplate()`.
	enrollmentTemplateSize	The size, in bytes, of the input enrollment template $0 \leq N \leq 2^{32} - 1$
Output Parameters	similarity	A similarity score resulting from comparison of the templates, on the range [0,DBL_MAX]. See section 0.
Return Value	See Table 18 for all valid return code values.	

3.2. 1:N Identification

3.2.1. Overview

The 1:N application proceeds in two phases, enrollment and identification. The identification phase includes separate pre-search feature extraction stage, and a search stage.

The design reflects the following *testing* objectives for 1:N implementations.

- support distributed enrollment on multiple machines, with multiple processes running in parallel
- allow recovery after a fatal exception, and measure the number of occurrences
- allow NIST to copy enrollment data onto many machines to support parallel testing
- respect the black-box nature of biometric templates
- extend complete freedom to the provider to use arbitrary algorithms
- support measurement of duration of core function calls
- support measurement of template size

Table 26 – Procedural overview of the identification test

Phase	#	Name	Description	Performance Metrics to be reported by NIST
Enrollment	E1	Initialization	Give the implementation advance notice of the number of individuals and images that will be enrolled. Give the implementation the name of a directory where any provider-supplied configuration data will have been placed by NIST. This location will otherwise be empty. The implementation is permitted **read-write-delete access** to the enrollment directory during this phase. The implementation is permitted read-only access to the configuration directory. After enrollment, NIST may rename and relocate the enrollment directory - the implementation should not depend on the name of the enrollment directory.	
	E2	Parallel Enrollment	For each of N individuals, pass multiple images of the individual to the implementation for conversion to a combined template. The implementation will return a template to the calling application. The implementation is permitted **read-only access** to the enrollment directory during this phase. NIST's calling application will be responsible for storing all templates as binary files. These will not be available to the implementation during this enrollment phase. Multiple instances of the calling application may run simultaneously or sequentially. These may be executing on different computers. The same person will not be enrolled twice.	Statistics of the times needed to enroll an individual. Statistics of the sizes of created templates. The incidence of failed template creations.
	E3	Finalization	Permanently finalize the enrollment directory. This supports, for example, adaptation of the image-processing functions, adaptation of the representation, writing of a manifest, indexing, and computation of statistical information over the enrollment dataset. The implementation is permitted **read-write-delete access** to the enrollment directory during this phase.	Size of the enrollment database as a function of population size N and the number of images. Duration of this operation. The time needed to execute this function shall be reported with the preceding enrollment times.

Pre-search	S1	Initialization	Tell the implementation the location of an enrollment directory. The implementation could look at the enrollment data. The implementation is permitted **read-only access** to the enrollment directory during this phase. Statistics of the time needed for this operation.	Statistics of the time needed for this operation.
	S2	Template preparation	For each probe, create a template from a set of input images. This operation will generally be conducted in a separate process invocation to step S3. The implementation is **permitted no access** to the enrollment directory during this phase. The result of this step is a search template.	Statistics of the time needed for this operation. Statistics of the size of the search template.
Search	S3	Initialization	Tell the implementation the location of an enrollment directory. The implementation should read all or some of the enrolled data into main memory, so that searches can commence. The implementation is permitted **read-only access** to the enrollment directory during this phase.	Statistics of the time needed for this operation.
	S4	Search	A template is searched against the enrollment database. The implementation is permitted **read-only access** to the enrollment directory during this phase.	Statistics of the time needed for this operation. Accuracy metrics - Type I + II error rates. Failure rates.

3.2.2. API

3.2.2.1. Interface

The Class C 1:N identification software under test must implement the interface `IdentInterface` by subclassing this class and implementing each method specified therein. See

	C++ code fragment	Remarks
1.	`class IdentInterface;` `typedef std::shared_ptr<IdentInterface> ClassCImplPtr;` `class IdentInterface`	
2.	`{` `public:`	
3.	`virtual ReturnStatus initializeEnrollmentSession( const std::string &configurationLocation, const std::string &enrollmentDirectory, const uint32_t numPersons, const uint32_t numImages) = 0;`	
4.	`virtual ReturnStatus convertMultifaceToEnrollmentTemplates( const Multiface &inputFaces, std::vector<PersonRep> &templates) = 0;`	
5.	`virtual ReturnStatus finalizeEnrollment ( const std::string &enrollmentDirectory, const std::string &edbName, const std::string &edbManifestName) = 0;`	
6.	`virtual ReturnStatus initializeFeatureExtractionSession( const std::string &configurationLocation, const std::string &enrollmentDirectory) = 0;`	
7.	`virtual ReturnStatus convertMultifaceToIdentificationTemplates( const Multiface &inputFaces, std::vector<PersonRep> &templates) = 0;`	
8.	`virtual ReturnStatus initializeIdentificationSession( const std::string &configurationLocation, const std::string &enrollmentDirectory);`	

9.	`virtual ReturnStatus identifyTemplate( const PersonRep &idTemplate, const uint32_t candidateListLength, std::vector<Candidate> &candidateList);`	
10.	`virtual void setGPU(uint8_t gpuNum) = 0;`	
11.	`static ClassCImplPtr getImplementation();`	Factory method to return a managed pointer to the `IdentInterface` object. This function is implemented by the submitted library and must return a managed pointer to the `IdentInterface` object. See section 3.1.2.1 for an example of a typical implementation of this method.
12.	`};`	

3.2.2.2. Initialization of the enrollment session

Before any enrollment feature extraction calls are made, the NIST test harness will call the initialization function of Table 27.

Table 27 – Enrollment initialization

Prototype	ReturnStatus initializeEnrollmentSession(	
	const std::string &configurationLocation,	Input
	const std::string &enrollmentDirectory,	Input
	const uint32_t numPersons,	Input
	const uint32_t numImages);	Input
Description	This function initializes the SDK under test and sets all needed parameters. This function will be called N=1 times by the NIST application immediately before any M ≥ 1 calls to `convertMultifaceToEnrollmentTemplate`. Caution: The implementation should tolerate execution of P > 1 processes on the one or more machines each of which may be reading and writing to this same enrollment directory in parallel. File locking or process-specific temporary filenames would be needed to safely write content in the `enrollmentDirectory`.	
Input Parameters	configurationLocation	A read-only directory containing any developer-supplied configuration parameters or run-time data files.
	enrollmentDirectory	The directory will be initially empty, but may have been initialized and populated by separate invocations of the enrollment process. When this function is called, the SDK may populate this folder in any manner it sees fit. Permissions will be read-write-delete.
	numPersons	The number of persons who will be enrolled $0 \leq N \leq 2^{32} - 1$ (e.g. 1 million)
	numImages	The total number of images that will be enrolled, summed over all identities $0 \leq M \leq 2^{32} - 1$ (e.g. 1.8 million)
Output Parameters	none	
Return Value	See Table 18 for all valid return code values.	

3.2.2.3. GPU Index Specification

For implementations using GPUs, the function of Table 28 specifies a sequential index identifying which GPU device to use. This enables the test software to orchestrate load balancing across multiple GPUs.

Table 28 – GPU index specification

Prototypes	void setGPU (	
	uint8_t gpuNum);	Input
Description	This function sets the GPU device number to be used by all subsequent implementation function calls. `gpuNum` is a zero-based sequence value of which GPU device to use. 0 would mean the first detected GPU, 1 would be the second GPU, etc. If the implementation does not use GPUs, then this function call should simply do nothing.	
Input Parameters	gpuNum	Index number representing which GPU to use.

3.2.2.4. Enrollment

A Multiface is converted to one or more enrollment templates (based on the number of persons found in the Multiface) using the function of Table 29.

Table 29 – Enrollment feature extraction

Prototypes	ReturnStatus convertMultifaceToEnrollmentTemplates(		
	const Multiface &inputFaces,		Input
	std::vector<PersonRep> &templates);		Output
Description	This function takes a Multiface and outputs a vector of PersonRep objects. If the function executes correctly (i.e. returns a successful exit status), the NIST calling application will store the template. The NIST application will concatenate the templates and pass the result to the enrollment finalization function. For a Multiface in which no persons appear, a valid output is an empty vector (i.e. size() == 0). If the function gives a non-zero exit status: – the test driver will ignore the output template (the template may have any size including zero) – the event will be counted as a failure to enroll. IMPORTANT: NIST's application writes the template to disk. The implementation must not attempt writes to the enrollment directory (nor to other resources). Any data needed during subsequent searches should be included in the template, or created from the templates during the enrollment finalization function of section 3.2.2.5.		
Input Parameters	inputFaces	An instance of a Table 9 structure. Implementations must alter their behavior according to the number of images contained in the structure.	
Output Parameters	templates	For each person detected in the Multiface, the function shall identify the person's estimated eye centers for each images/video frame where the person's eye coordinates can be calculated. The eye coordinates shall be captured in the PersonRep.eyeCoordinates variable, which is a vector of EyePair objects. For videos, the frame number from the video of where the eye coordinates were detected shall be captured in the EyePair.frameNum variable for each pair of eye coordinates. In the event the eye centers cannot be calculated (ie. the person becomes out of sight for a few frames in the video), the SDK shall not store an EyePair for those frames.	
Return Value	See Table 18 for all valid return code values.		

3.2.2.5. Finalize enrollment

After all templates have been created, the function of Table 30 will be called. This freezes the enrollment data. After this call the enrollment dataset will be forever read-only.

The function allows the implementation to conduct, for example, statistical processing of the feature data, indexing and data re-organization. The function may alter the file structure. It may increase or decrease the size of the stored data. No output is expected from this function, except a return code.

Implementations shall not *move* the input data. Implementations shall not point to the input data. Implementations should not assume the input data would be readable after the call. Implementations must, at a minimum, copy the input data or otherwise extract what is needed for search.

Table 30 – Enrollment finalization

Prototypes	ReturnStatus finalizeEnrollment (	
	const std::string &enrollmentDirectory,	Input
	const std::string &edbName,	Input
	const std::string &edbManifestName);	Input
Description	This function takes the name of the top-level directory where enrollment database (EDB) and its manifest have been stored. These are described in section 2.5. The enrollment directory permissions will be read + write. The function supports post-enrollment, developer-optional, bookkeeping operations, including indexing, tree-building, statistical processing and data re-ordering for fast in-memory searching. The function will generally be called in a	

	separate process after all the enrollment processes are complete. This function should be tolerant of being called two or more times. Second and third invocations should probably do nothing.	
Input Parameters	enrollmentDirectory	The top-level directory in which enrollment data was placed. This variable allows an implementation to locate any private initialization data it elected to place in the directory.
	edbName	The name of a single file containing concatenated templates, i.e. the EDB of section 2.5. While the file will have read-write-delete permission, the SDK should only alter the file if it preserves the necessary content, in other files for example. The file may be opened directly. It is not necessary to prepend a directory name. This is a NIST-provided input – implementers shall not internally hard-code or assume any values.
	edbManifestName	The name of a single file containing the EDB manifest of section 2.5. The file may be opened directly. It is not necessary to prepend a directory name. This is a NIST-provided input – implementers shall not internally hard-code or assume any values.
Output Parameters	None	
Return Value	See Table 18 for all valid return code values.	

3.2.2.6. Pre-search feature extraction

3.2.2.7. Initialization

Before `Multiface`s are sent to the identification feature extraction function, the test harness will call the initialization function in Table 31.

Table 31 – Identification feature extraction initialization

Prototype	ReturnStatus initializeFeatureExtractionSession(	
	const std::string &configurationLocation,	Input
	const std::string &enrollmentDirectory);	Input
Description	This function initializes the SDK under test and sets all needed parameters. This function will be called once by the NIST application immediately before any M ≥ 1 calls to `convertMultifaceToIdentificationTemplate`. Caution: The implementation should tolerate execution of P > 1 processes on the one or more machines each of which may be reading from this same enrollment directory in parallel. The implementation has read-only access to its prior enrollment data.	
Input Parameters	configuration_location	A read-only directory containing any developer-supplied configuration parameters or run-time data files.
	enrollment_directory	The read-only top-level directory in which enrollment data was placed and then finalized by the implementation. The implementation can parameterize subsequent template production on the basis of the enrolled dataset.
Output Parameters	none	
Return Value	See Table 18 for all valid return code values.	

3.2.2.8. Feature extraction

A `Multiface` is converted to an atomic identification template using the function of Table 32. The result may be stored by NIST, or used immediately. The SDK shall not attempt to store any data.

Table 32 – Identification feature extraction

Prototypes	ReturnStatus convertMultifaceToIdentificationTemplates(	
	const Multiface &inputFaces,	Input
	std::vector<PersonRep> &templates);	Output
Description	This function takes a `Multiface` as input and populates a vector of `PersonRep` with the number of persons detected from the `Multiface`. The implementation could call `vector::push_back` to insert into the vector.	

	If the function executes correctly, it returns a zero exit status. The NIST calling application may commit the template to permanent storage, or may keep it only in memory (the implementation does not need to know). If the function returns a non-zero exit status, the output template will be not be used in subsequent search operations. The function shall not have access to the enrollment data, nor shall it attempt access.	
Input Parameters	inputFaces	One or more faces, or a video clip.
Output Parameters	templates	For each person detected in the video, the function shall create a `PersonRep` object, populate it with a template and eye coordinates for each image or video frame where eyes were detected, and add it to the vector.
Return Value	See Table 18 for all valid return code values.	

3.2.2.9. Initialization

The function of Table 33 will be called once prior to one or more calls of the searching function of Table 34. The function might set static internal variables so that the enrollment database is available to the subsequent identification searches.

Table 33 – Identification initialization

Prototype	ReturnStatus initializeIdentificationSession(	
	const std::string &configurationLocation,	Input
	const std::string &enrollmentDirectory);	Input
Description	This function reads whatever content is present in the `enrollmentDirectory`, for example a manifest placed there by the `finalizeEnrollment` function.	
Input Parameters	configurationLocation	A read-only directory containing any developer-supplied configuration parameters or run-time data files.
	enrollmentDirectory	The read-only top-level directory in which enrollment data was placed.
Return Value	See Table 18 for all valid return code values.	

3.2.2.10. Search

The function of Table 34 compares a proprietary identification template against the enrollment data and returns a candidate list.

Table 34 – Identification search

Prototype	ReturnStatus identifyTemplate(	
	const PersonRep &idTemplate,	Input
	const uint32_t candidateListLength,	Input
	std::vector<Candidate> &candidateList);	Output
Description	This function searches an identification template against the enrollment set, and outputs a vector containing `candidateListLength` Candidates. Each candidate shall be populated by the implementation and added to `candidateList`. Note that candidateList will be an empty vector when passed into this function. The candidates shall appear in descending order of similarity score - i.e. most similar entries appear first.	
Input Parameters	idTemplate	A template from `convertMultifaceToIdentificationTemplates()` - If the value returned by that function was non-zero the contents of `idTemplate` will not be used and this function (i.e. `identifyTemplate`) will not be called.
	candidateListLength	The number of candidates the search should return.
Output Parameters	candidateList	A vector containing `candidateListLength` objects of candidates. The datatype is defined in Table 17 . Each candidate shall be populated by the implementation. The candidates shall appear in descending order of similarity score - i.e. most similar entries appear first.
Return Value	See Table 18 for all valid return code values.	

NOTE: Ordinarily the calling application will set the input candidate list length to operationally typical values, say $0 \le L \le 200$, and $L << N$. However, there is interest in the presence of mates much further down the candidate list. We may therefore extend the candidate list length such that L approaches N. We may measure the dependence of search duration on L.

3.3. Face Detection

3.3.1. API

3.3.1.1. Interface

The Class D detection software under test must implement the interface `DetectInterface` by subclassing this class and implementing each method specified therein.

	C++ code fragment	Remarks

1.	`class DetectInterface;`
	
	`typedef std::shared_ptr<DetectInterface> ClassDImplPtr;`
	
	`class DetectInterface`	

2.	`{`
	`public:`	

3.	`virtual ReturnStatus initializeDetection(`
	`const std::string &configurationLocation) = 0;`	

4.	`virtual ReturnStatus detectFaces(`
	`const Image &inputImage,`
	`std::vector<BoundingBox> &boundingBoxes) = 0;`	

5.	`virtual void setGPU(uint8_t gpuNum) = 0;`	
6.	`static ClassDImplPtr getImplementation();`	Factory method to return a managed pointer to the `DetectInterface` object. This function is implemented by the submitted library and must return a managed pointer to the `DetectInterface` object. See section 3.1.2.1 for an example of a typical implementation of this method.
7.	`};`	

3.3.1.2. Initialization

Before any calls to `detectFaces` are made, the NIST test harness will make a call to the initialization of the function in Table 35.

Table 35 – SDK initialization

Prototype	ReturnStatus initializeDetection(const std::string &configurationLocation);	 Input
Description	This function initializes the SDK under test. It will be called by the NIST application before any call to the function `detectFaces`. The SDK under test should set all parameters.	
Input Parameters	configurationLocation	A read-only directory containing any developer-supplied configuration parameters or run-time data. The name of this directory is assigned by NIST, not hardwired by the provider. The names of the files in this directory are hardwired in the SDK and are unrestricted.
Output Parameters	none	
Return Value	See Table 18 for all valid return code values.	

3.3.1.3. GPU Index Specification

For implementations using GPUs, the function of Table 36 specifies a sequential index for which GPU device to execute on. This enables the test software to orchestrate load balancing across multiple GPUs.

Table 36 – GPU index specification

Prototypes	void setGPU (uint8_t gpuNum);	 Input
Description	This function sets the GPU device number to be used by all subsequent implementation function calls. `gpuNum` is a zero-based sequence value of which GPU device to use. 0 would mean the first detected GPU, 1 would be the second GPU, etc. If the implementation does not use GPUs, then this function call should simply do nothing.	

Input Parameters	gpuNum	Index number representing which GPU to use.

3.3.1.4. Face detection

The function of Table 37 supports the detection of faces in an image. An image may contain one or more faces.

Table 37 – Face detection

Prototypes	ReturnStatus detectFaces(		
	const Image &inputImage,		Input
	std::vector<BoundingBox> &boundingBoxes);		Output
Description	This function takes an Image as input, and populates a vector of `BoundingBox` with the number of faces detected from the input image. The implementation could call `vector::push_back` to insert into the vector.		
Input Parameters	inputImage	An instance of a struct representing a single image from Table 8.	
Output Parameters	boundingBoxes	For each face detected in the image, the function shall create a `BoundingBox` (see Table 14), populate it with a confidence score, the x, y, width, height of the bounding box, and add it to the vector.	
Return Value	See Table 18 for all valid return code values.		

3.4. Clustering

3.4.1. Definitions

Clustering is the act of grouping imagery of the same individuals. If a large image collection has N images in which P ≥ 0 subjects appear, an implementation should return N lists. The n-th list contains zero or more hypotheses about who appears in the n-th input image. Each hypothesis is comprised of: a bounding box; an integer subject identifier; and a similarity score. A similarity is a measure of confidence that a hypothesized identity truly shares the same face as others in that cluster. Subject identifiers are labels assigned by the algorithm.

Clustering will, in general, produce detection errors (where a person is not found at all), and both false positive and negative associations where, respectively, multiple persons appear in one cluster, one person exists in several clusters. A single image can contain one or more faces in it.

3.4.2. API

3.4.2.1. Interface

The Class G clustering software under test must implement the interface `ClusterInterface` by subclassing this class and implementing each method specified therein. See

	C++ code fragment	Remarks

1.	`class ClusterInterface;`
	
	`typedef std::shared_ptr<ClusterInterface> ClassGImplPtr;`
	
	`class ClusterInterface`	

2.	`{`
	`public:`	

3.	`virtual ReturnStatus initializeClustering(`
	`const std::string &configurationLocation) = 0;`	

4.	`virtual ReturnStatus clusterIdentities(`
	`const std::vector<Image> &inputFaces,`
	`std::vector<ClusterMembersInImage> &assignments) = 0;`	

5.	`virtual void setGPU(uint8_t gpuNum) = 0;`	
6.	`static ClassGImplPtr getImplementation();`	Factory method to return a managed pointer to the `ClusterInterface` object. This function is implemented by the submitted library and must return a managed pointer to the `ClusterInterface` object. See section 3.1.2.1 for an example of a typical implementation of this method.
7.	`};`	

3.4.2.2. Initialization

Before any calls to `clusterIdentities` are made, the NIST test harness will make a call to the initialization of the function in Table 38.

Table 38 – SDK initialization

Prototype	ReturnStatus initializeClustering(	
	const std::string &configurationLocation);	Input
Description	This function initializes the SDK under test. It will be called by the NIST application before any call to the function `clusterIdentities`. The SDK under test should set all parameters.	
Input Parameters	configurationLocation	A read-only directory containing any developer-supplied configuration parameters or run-time data files. The name of this directory is assigned by NIST. It is not hardwired by the provider. The names of the files in this directory are hardwired in the SDK and are unrestricted.

Output Parameters	none	
Return Value	See Table 18 for all valid return code values.	

3.4.2.3. GPU Index Specification

For implementations using GPUs, the function of Table 39 specifies a sequential index for which GPU device to execute on. This enables the test software to orchestrate load balancing across multiple GPUs.

Table 39 – GPU index specification

Prototypes	void setGPU (	
	uint8_t gpuNum);	Input
Description	This function sets the GPU device number to be used by all subsequent implementation function calls. `gpuNum` is a zero-based sequence value of which GPU device to use. 0 would mean the first detected GPU, 1 would be the second GPU, etc. If the implementation does not use GPUs, then this function call should simply do nothing.	
Input Parameters	gpuNum	Index number representing which GPU to use.

3.4.2.4. Cluster Identities

The implementation shall implement the function given in Table 40.

Table 40 – Clustering

Prototype	ReturnStatus clusterIdentities(	
	const std::vector<Image> &inputFaces,	Input
	const int32_t numClusters,	Input
	std::vector<ClusterMembersInImage> &assignments	Output
Description	This function takes a collection of images and outputs cluster hypotheses. This function is not mediated by a separate template generation step: All detection, template generation and matching occurs internal to this function. NIST will pre-allocate the `assignments` vector to have size equal to `input_faces.size()`. It is up to the implementations to populate the `assignments` vector based on the number of faces found in the images. For each input image, `inputFaces[i]`, the implementation should assign hypothesized cluster assignment(s) in `assignments[i]`. There may be zero or more persons in each image. If this image contained three faces, then `assignments[i].members.size()` should be 3. When greater than 0, the value `numClusters` represents an upper bound on the number of individuals that might be present in the entire collection. This value assumes that an investigator with domain-specific knowledge would be able to estimate the number of subjects present, at least to an order-of-magnitude. NIST will set `numClusters` to 10, 100, 1000, 10000, 100000. The actual number of individuals will be less than this. NIST will also set `numClusters` to -1 which represents the case where the number of subjects is truly unknown.	
Input Parameters	inputFaces	N Images from P ≥ 0 subjects. There are n_i ≥ 1 images from individual i.
	numClusters	Upper bound on the number of clusters. -1 represents an unset value.
Output Parameters	assignments	N lists of cluster assignments.
Return Value	See Table 18 for all valid return code values.	

617 ## 4. References

FRVT 2002	Face Recognition Vendor Test 2002: Evaluation Report, NIST Interagency Report 6965, P. Jonathon Phillips, Patrick Grother, Ross J. Micheals, Duane M. Blackburn, Elham Tabassi, Mike Bone
FRVT 2002b	Face Recognition Vendor Test 2002: Supplemental Report, NIST Interagency Report 7083, Patrick Grother
FRVT 2012	Patrick Grother and Mei Ngan, Face Recognition Vendor Test (FRVT) Performance of Face Identification Algorithms, NIST Interagency Report 8009, May 26, 2014.
AN27	NIST Special Publication 500-271: American National Standard for Information Systems — Data Format for the Interchange of Fingerprint, Facial, & Other Biometric Information – Part 1. (ANSI/NIST ITL 1-2007). Approved April 20, 2007.
IJBA	See for example the images here: http://www.nist.gov/itl/iad/ig/facechallenges.cfm As documented here: Klare et al. Pushing the Frontiers of Unconstrained Face Detection and Recognition: IARPA Janus Benchmark A, CVPR, June 2015.
MEDS	NIST Special Database 32, Volume 1 and Volume 2 are available at: http://www.nist.gov/itl/iad/ig/sd32.cfm. MEDS-II is an update to MEDS-I and was published in February 2011. Note that NIST does not provide "training" data per se - this differs from the paradigm often used in academic research where a model is trained, tested and validated. Instead CHEXIA-FACE follows operational reality: software is typically shipped "as is" with a fixed internal representation that is designed to be usable "off the shelf" without training and with only minimal configuration.
MBE	P. Grother, G .W. Quinn, and P. J. Phillips, Multiple-Biometric Evaluation (MBE) 2010, Report on the Evaluation of 2D Still Image Face Recognition Algorithms, NIST Interagency Report 7709, Released June 22, 2010. Revised August 23, 2010. http://face.nist.gov/mbe

618

Annex A
Submission of Implementations to the CHEXIA-FACE

A.1 Submission of implementations to NIST

NIST requires that all software, data and configuration files submitted by the participants be signed and encrypted. Signing is done with the participant's private key, and encryption is done with the NIST public key. The detailed commands for signing and encrypting are given here: http://www.nist.gov/itl/iad/ig/encrypt.cfm

NIST will validate all submitted materials using the participant's public key, and the authenticity of that key will be verified using the key fingerprint. This fingerprint must be submitted to NIST by writing it on the signed participation agreement.

By encrypting the submissions, we ensure privacy; by signing the submission, we ensure authenticity (the software actually belongs to the submitter). NIST will reject any submission that is not signed and encrypted. NIST accepts no responsibility for anything that is transmitted to NIST that is not signed and encrypted with the NIST public key.

A.2 How to participate

Those wishing to participate in CHEXIA-FACE testing must do all of the following, on the schedule listed on Page 2.

— IMPORTANT: Follow the instructions for cryptographic protection of your SDK and data here. http://www.nist.gov/itl/iad/ig/encrypt.cfm

— Send a signed and fully completed copy of the *Application to Participate in the Child Exploitation Image Analytics – Face Recognition Evaluation (CHEXIA-FACE)*. This is available at http://www.nist.gov/itl/iad/ig/chexia-face.cfm. This must identify, and include signatures from, the Responsible Parties as defined in the application. The properly signed CHEXIA-FACE Application to Participate shall be sent to NIST as a PDF.

— Provide an SDK (Software Development Kit) library which complies with the API (Application Programmer Interface) specified in this document.

- Encrypted data and SDKs below 20MB can be emailed to NIST at chexia.face@nist.gov.
- Encrypted data and SDKS above 20MB shall be

  EITHER

  - Split into sections AFTER the encryption step. Use the unix "split" commands to make 9MB chunks, and then rename to include the filename extension need for passage through the NIST firewall.
  - `you%   split -a 3 -d -b 9000000  libCHEXIAFACE_enron_A_02.tgz.gpg`
  - `you%   ls -1 x??? | xargs -iQ  mv Q libCHEXIAFACE6_enron_A_02_Q.tgz.gpg`
  - Email each part in a separate email. Upon receipt NIST will
  - `nist%  cat chexiaface_enron_A02_*.tgz.gpg > libCHEXIAFACE_enron_A_02.tgz.gpg`

  OR

  - Made available as a file.zip.gpg or file.zip.asc download from a generic http webserver[6],

  OR

  - Mailed as a file.zip.gpg or file.zip.asc on CD / DVD to NIST at this address:

CHEXIA-FACE Test Liaison (A203) 100 Bureau Drive A203/Tech225/Stop 8940 NIST Gaithersburg, MD 20899-8940 USA	In cases where a courier needs a phone number, please use NIST shipping and handling on: 301 -- 975 -- 6296.

[6] NIST will not register, or establish any kind of membership, on the provided website.

A.3 Implementation validation

Registered Participants will be provided with a small validation dataset and test program available on the website http://www.nist.gov/itl/iad/ig/chexia-face.cfm shortly after the final evaluation plan is released.

The validation test programs shall be compiled by the provider. The output of these programs shall be submitted to NIST.

Prior to submission of the SDK and validation data, the Participant must verify that their software executes on the validation images, and produces correct similarity scores and templates.

Software submitted shall implement the CHEXIA-FACE API Specification as detailed in the body of this document.

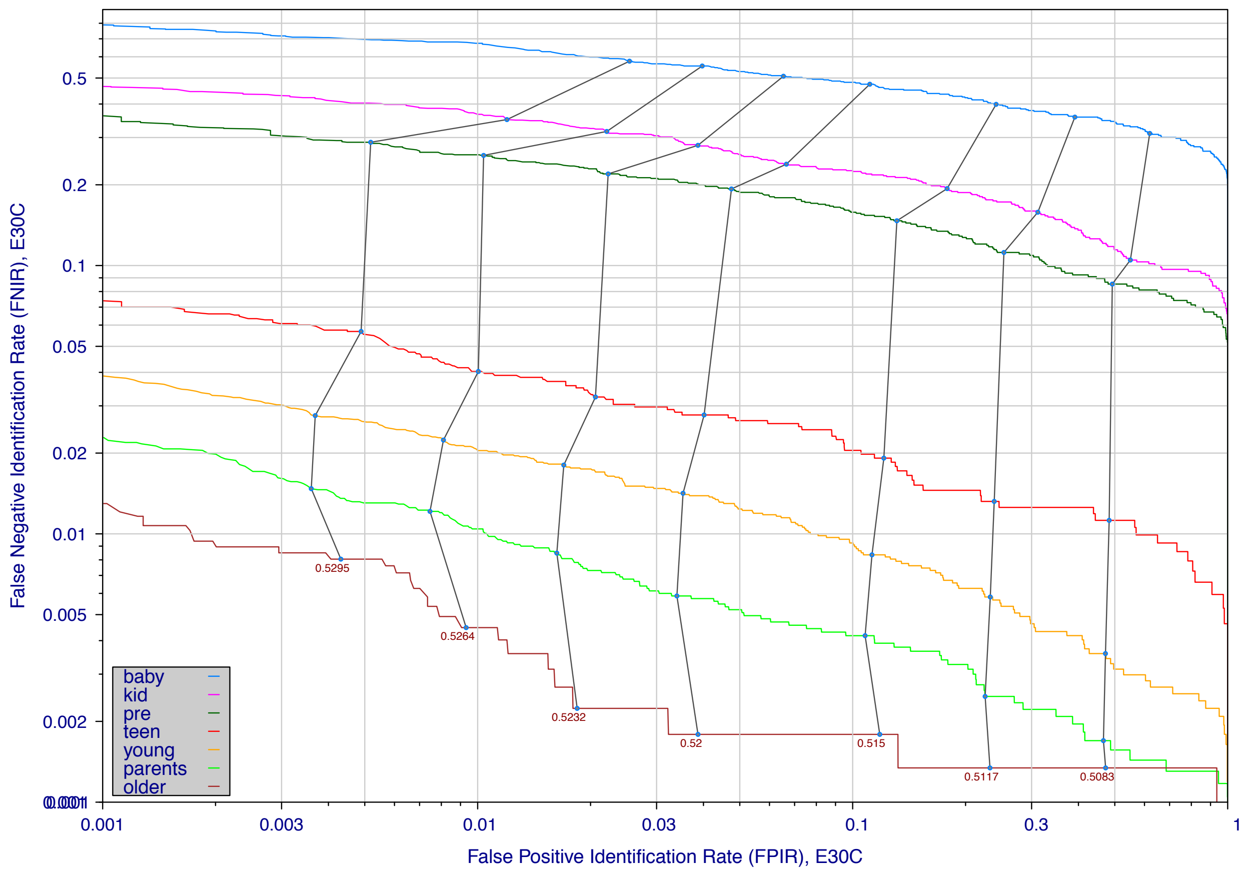
Upon receipt of the SDK and validation output, NIST will attempt to reproduce the same output by executing the SDK on the validation imagery, using a NIST computer. In the event of disagreement in the output, or other difficulties, the Participant will be notified.

Annex B
Effect of Age on Face Identification Accuracy

For the most accurate algorithm provided to NIST's FRVT evaluation in late 2013 the Figure below shows the one-to-many identification accuracy for subjects from particular age groups. The images are visa images. We enrolled a first image from each of N = 19972 individuals. Thereafter, we executed one mated search from those individuals to allow estimation of False Negative Identification Rate (FNIR, aka "miss rate"). We also executed 203,082 non-mated searches to allow computation of the False Positive Identification Rate (FPIR, aka "false alarm rate").

Results for 40 algorithms appear in Annex A of NIST Interagency Report 8009[7]. The discussion from that report is:

- **Recognition is progressively easier with advancing age**: All algorithms exhibit a strong dependence of FNIR on age. This effect is very large, spanning a factor of ten from infant to senior, and a factor of around five from teen to senior. Miss rates for *older* persons are very low: at a fixed FPIR of 0.005, the most accurate algorithm, E30C, gives FNIR of 0.008 for persons over age 55, 0.027 for *young* 20-somethings, and 0.057 for teenagers. For younger persons, the miss rates climb rapidly to 0.29 for *pre*-teens, 0.4 for *kids*, to 0.7 for *babies*. This progression is common to all algorithms.
- **Young children are more difficult to recognize**: Identification miss rates (FNIR) ascend rapidly for *pre*-teens, *kids* and the youngest individuals. For the *baby* group, 0 to about 3 years old, identification fails more often that it succeeds, i.e. FNIR is above 50%. While the sample size is small (57 subjects), error rates are so high that the result remains significant. This result applies for image pairs collected on average 1.6 years apart (Table 13) and will be in considerable part due to the craniofacial shape change associated with rapid growth. The extent to which smooth "feature-less" skin texture affects FNIR is unknown. Likewise the pose variations inherent in photographing children have not been quantified.
- **Young children are more difficult to discriminate**: All of the algorithms exhibit higher false positive identification rates for younger subjects. The grey lines in Figure 11, which link points of equal threshold, slope upwards to the right, indicating simultaneously that younger subjects are less easy to recognize as themselves but also less easy to tell apart. This indicates that younger individuals are more difficult to discriminate from other individuals.



[7] http://biometrics.nist.gov/cs_links/face/frvt/frvt2013/NIST_8009.pdf