Face Recognition Vendor Test
Ongoing

Still Face and Iris 1:N Identification
Application Programming Interface

VERSION 3.0

Patrick Grother
Mei Ngan
Kayee Hanaoka
Information Access Division
Information Technology Laboratory

April 6, 2023

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

Revision History

Date	Version	Description
FRVT 2018		Prior evaluation documented in NIST IR 8238
April 1, 2019	1.0	Initial document
September 9, 2020	1.0.1	- Update link to General Evaluation Specifications document - Adjust the legal similarity score range
August 16, 2021	1.0.2	Removed FRVT 1:1 pre-requisite. Developers may now participate in FRVT 1:N without having to participate in FRVT 1:1
November 3, 2021	1.0.3	- Added clarification that multi-threading is allowed in the finalizeEnrollment() function - Removed holdover text from 2018 - Added clarification on function time limits to be based on a single core
January 7, 2022	2.0	Add second version of createTemplate() function from Section 8.4.4 that supports the existence of multiple people in an image
April 6, 2023	3.0	1. Add support for iris images, allowing 1:N evaluation of iris recognition algorithms – this replaces the IREX 10 evaluation. 2. Allow evaluation of multimodal (face + iris) algorithms. 3. Specify new time limits and faster CPU processor for measurement of processing duration. 4. Add support for non-visible illumination wavelengths for iris and face

Table of Contents

List of Tables

1. FRVT 1:N

1.1. Scope

This document establishes a concept of operations and an application programming interface (API) for evaluation of one-to-many face recognition algorithms, one-to-many iris recognition algorithms, and algorithms that can extract information from face and iris images of the same person to implement multimodal one-to-many recognition.

Developers may submit a one-to-many search algorithm that operates on

- Face images only, **or**
- Iris images only, **or**
- Multimodal samples comprised of both face and iris images. The implementation must handle some unimodal samples – for example, a gallery for which 80% of enrolled samples are face and iris, but 10% of samples are face-only, and 10% are iris-only.

2. General Evaluation Specifications

General and common information shared between all Ongoing FRVT tracks are documented in the FRVT General Evaluation Specifications document - https://pages.nist.gov/frvt/api/FRVT_common.pdf. This includes rules for participation, hardware and operating system environment, software requirements, reporting, and common data structures that support the APIs.

3. Core accuracy metrics

This test will execute open-universe searches. That is, some proportion of searches will not have an enrolled mate. From the candidate lists returned by algorithms, NIST will compute and report accuracy metrics, primarily:

- False negative identification rate (FNIR) – the proportion of mated searches which do not yield a mate within the top R ranks and at or above threshold, T.
- False positive identification rate (FPIR) – the proportion of non-mated searches returning any (1 or more) candidates at or above a threshold, T.
- Selectivity – the number of non-mated candidates returned at or above a threshold, T. This quantity has a value running from 0 to L, the number of candidates requested. It may be fractional, as it is estimated as a count divided by the number of non-mate searches.

These quantities are estimated from candidate lists produced by requesting the top L most similar candidates to the search. We do not intend to execute searches requesting only those candidates above a specified input threshold.

We will report FNIR, FPIR and selectivity by sweeping the threshold over the interval [0, infinity). Error tradeoff plots (FNIR vs. FPIR, parametric on threshold) will be the primary reporting mechanism.

We will also report FNIR by sweeping a rank R over the interval [1, L] to produce (the complement of) the cumulative match characteristic (CMC).

We will report proportions of template generations that fail to produce a viable template – i.e. failure to enroll rate (FTE).

4. Application relevance

NIST anticipates reporting FNIR in two FPIR regimes:

- Investigation mode: Given candidate lists and a threshold of zero, the CMC metric is relevant to investigational applications where human examiners will adjudicate candidates in decreasing order of similarity. This is common in law enforcement “lead generation”.

- Identification mode: We will apply (high) thresholds to candidate lists and report FNIR values relevant to identification applications where human labor is matched to the tolerable number of false positives per unit time. This is used in duplicate-ID detection searches for credential issuance and, more so, in surveillance applications.

Developers are encouraged to submit variants tailored to minimize FNIR in the two FPIR regimes, and to explore the speed-accuracy trade space.

5. Limits

5.1. Time limits

The elemental functions of the implementations shall execute under the time constraints of Table 1. These time limits apply to the function call invocations defined in section 8. Assuming the times are random variables, NIST cannot regulate the maximum value, so the time limits are median values. This means that the median of all operations should take less than the identified duration. Timing will be estimated from at least 1000 separate invocations of each elemental function.

Timing will be measured on a fixed Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz computer. Durations are measured by wrapping API function in calls to the std::chrono() high-resolution timer.

Table 1 – Processing time limits in seconds, per 640 x 480 image

Function	1:N Face	1:N Iris	1:N Face + Iris
Template Generation: Conversion of one 640x480 image to one template	1.5 sec (1 core)	1.5 sec (1 core)	3.0 seconds (one face + one eye)
1:N finalization (on gallery of 1 million enrolled templates) e.g. for building of a fast search data structure	40000 sec	40000 sec	80000
1:N template search for: – N = 1 million enrolled templates – L = 50 returned candidates	10 sec (1 core)	25 sec (1 core)	25 sec (1 core)

5.2. Template size limits

There are no template size limits. However, NIST anticipates evaluating performance with N in excess of 10^7. For implementations that represent a gallery in memory with a linear data structure, the memory of our machines implies a limit on template sizes. For example, given machines equipped with 768GB of memory, and N = 25 million, templates cannot exceed 32KB.

The API, however, supports multi-stage searches and read access of the disk during the 1:N search. Disk access would likely be very slow. In all cases, algorithms shall meet the duration limits given in Table 1, with linear gallery size scaling.

6. Implementation Library Filename

- The core library shall be named as libfrvt_1N_<***provider***>_<***sequence***>.so, with
- provider: non-infringing name of the main provider. Do not use names of product lines, and do not include organizational legal organizational abbreviations like LLC, Corp, Gmbh, Ltd. Example: acme.
- sequence: a three digit decimal identifier to start at 000 and incremented by 1 every time a library is sent to NIST. Example: 007

Example core library names: *libfrvt_1N_acme_000.so, libfrvt_1N_myface_000.so, etc.*
Important: Public results will be attributed with the provider name and the 3-digit sequence number in the submitted library name.

7. Data structures supporting the API

The general data structures supporting this API are documented in the FRVT - General Evaluation Specifications document available at https://pages.nist.gov/frvt/api/FRVT_common.pdf. The data structures specific to this particular test are described within this document. The header files are published at https://github.com/usnistgov/frvt.

7.1. File structure for enrolled template collection

To support this 1:N test, 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 many 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 2. 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 2 – Enrollment dataset template manifest

Field name	Template ID	Template Length	Position of first byte in EDB
Datatype required	std::string	uint64_t	uint64_t
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 small individual files.

7.1.1. Gallery Type

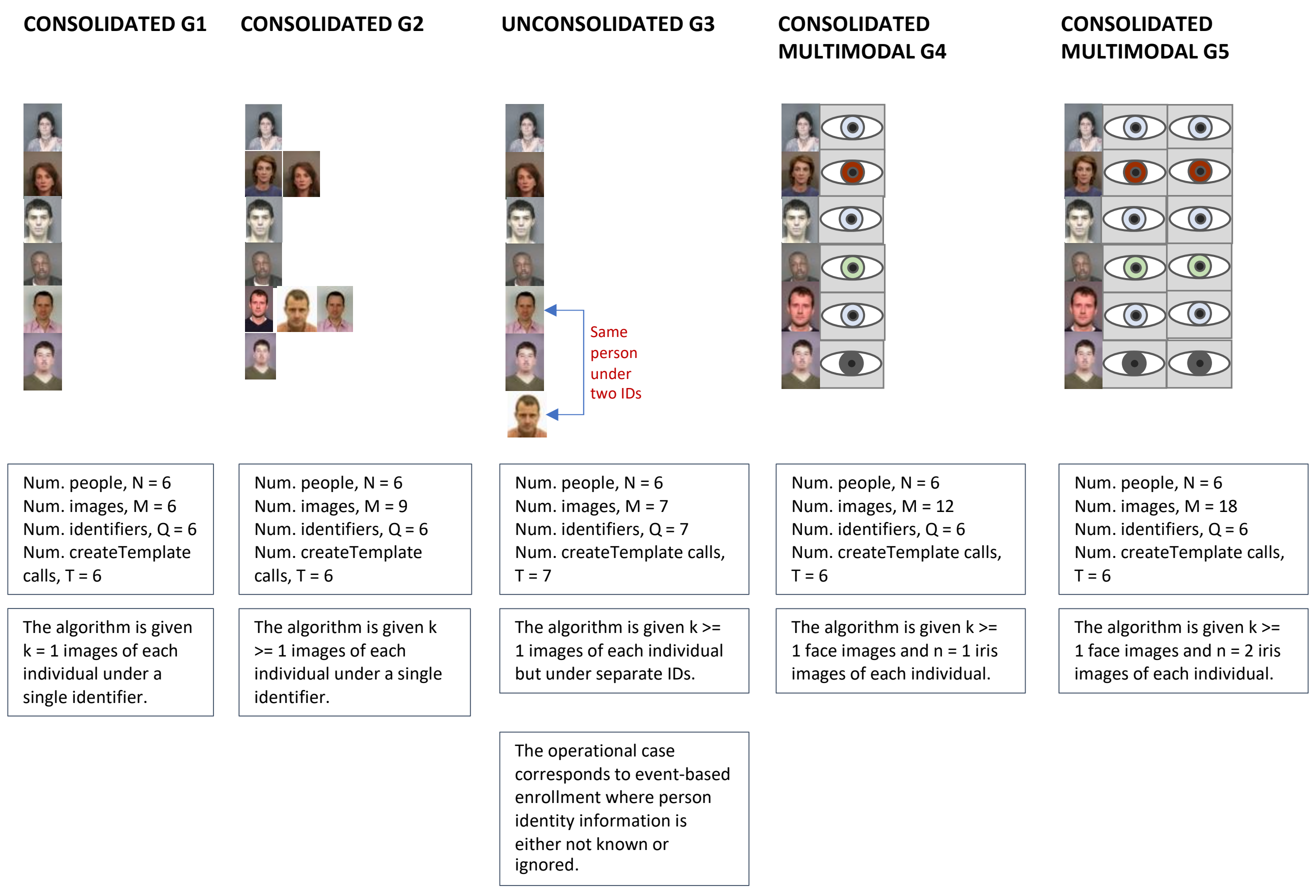

Figure 1 – Illustration of consolidated versus unconsolidated enrollment database[3]

Figure 1 illustrates four examples of two types of galleries:

- **Consolidated:** The database is formed by enrolling all images of a subject under a common identity label. The result is a gallery with N identities and N templates. This type of gallery presents us with the cleanest experimental design, "one needle in a haystack" scenario. It allows algorithms to perform image and feature level fusion. Operationally it requires high integrity biographical information to maintain.
- **Unconsolidated:** The database is formed by enrolling photographs without regard to whether the subject already has already been enrolled or not. Under this scheme, different images of the same person can exist in the gallery under different subject identifiers, that is, there are N identities, and M > N database entries.

During gallery finalization, algorithms will be provided with an enumerated label from Table 3 which specifies the type of gallery being processed.

Table 3 – Labels describing gallery composition

Label as C++ enumeration	Meaning
`enum class GalleryType {`	
`    Consolidated,`	Consolidated, subject-based enrollment
`    Unconsolidated`	Unconsolidated, event-based or photo-based enrollment
`};`	

7.1.2. 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 4.

Table 4 – Structure for a candidate

	C++ code fragment	Remarks
1.	`typedef struct Candidate`	
2.	`{`	
3.	`bool isAssigned;`	If the candidate computation succeeded, this value is set to true. False otherwise. If value is set to false, score and templateId will be ignored entirely.
4.	`std::string templateId;`	The Template ID from the enrollment database manifest defined in clause 7.1.
5.	`double score;`	Measure of similarity or dissimilarity between the identification template and the enrolled candidate. – For face recognition, a similarity score - higher is more similar – For iris recognition, a non-negative measure of dissimilarity (maybe a distance) - lower is more similar – For multimodal face and iris, a similarity score - higher is more similar An algorithm is free to assign any value to a candidate. The distribution of values will have an impact on the false-negative and false-positive identification rates. The score values should be reported on the range that is used in the developer's software products. We require scores to be non-negative. Developers often use [0,1], for example. Our test reports include various plots with threshold values e.g. FMR(T), to allow end-users to set thresholds in operations. These plots may become difficult to interpret if scores span many orders of magnitude.
6.	`} Candidate;`	

[3] The face images contained in this figure are from the publicly available Special Database 32 - Multiple Encounter Dataset (MEDS). https://www.nist.gov/itl/iad/image-group/special-database-32-multiple-encounter-dataset-meds

8. API specification

FRVT 1:N participants shall implement the relevant C++ prototyped interfaces of section 8. C++ was chosen in order to make use of some object-oriented features.

Please note that included with the FRVT 1:N validation package (available at https://github.com/usnistgov/frvt) is a "null" implementation of this API. The null implementation has no real functionality but demonstrates mechanically how one could go about implementing this API.

8.1. Header File

The prototypes from this document will be written to a file named **frvt1N.h** and will be available to implementers at https://github.com/usnistgov/frvt.

8.2. Namespace

All supporting data structures will be declared in the `FRVT` namespace. All API interfaces/function calls for this track will be declared in the `FRVT_1N` namespace.

8.3. Overview

The 1:N identification application proceeds in three phases: enrollment, finalization and identification. The identification phase includes separate probe feature extraction and search stages.

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
- support measurement of template insertion and removal times into an enrollment database

Table 5 – Procedural overview of the 1:N test

Phase	#	Name	Description	Performance Metrics to be reported by NIST
Enrollment	E1	Initialization	**initializeTemplateCreation(TemplateRole=Enrollment_1N)** 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-only** access to the configuration directory.	
	E2	Parallel Enrollment	**create{Face,Iris,FaceAndIris}Template(TemplateRole=Enrollment_1N)** For each of N individuals, pass K >= 1 images of the individual to the implementation for conversion to a template. The implementation will return a template to the calling application. 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.	Statistics of the times needed to enroll an individual. Statistics of the sizes of created templates. The incidence of failed template creations.

Gallery Finalization	F1	Finalization	**finalizeEnrollment()** 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 and **read-only access** to the configuration directory during this phase. **Note:** finalizeEnrollment() will be called in a separate process than the enrollment functions.	Size of the enrollment database as a function of population size N. Duration of this operation. The time needed to execute this function shall be reported with the preceding enrollment times.
Probe Template Creation	S1	Initialization	**initializeTemplateCreation(TemplateRole=Search_1N)** 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-only** access to the configuration directory.	Statistics of the time needed for this operation.
	S2	Template preparation	**create{Face,Iris,FaceAndIris}Template(TemplateRole=Search_1N)** For each probe, create a template from K >= 1 images. The result of this step is a search template. Multiple instances of the calling application may run simultaneously or sequentially. These may be executing on different computers.	Statistics of the time needed for this operation. Statistics of the size of the search template.
Search	S3	Initialization	**initializeIdentification()** Tell the implementation the location of an enrollment directory that contains the gallery files produced from the **finalize()** function. The enrollment directory will always contain a successfully finalized gallery (i.e. will never be empty). 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. **Note:** The search functions (initializeIdentification(), identifyTemplate()) will be called in a separate process from the enrollment functions, therefore, you <u>cannot</u> assume that initializeTemplateCreation() is called by the test harness prior to the search functions.	Statistics of the time needed for this operation.
	S4	Search	**identifyTemplate()** A template is searched against the enrollment database. Developers shall not attempt to improve the duration of the identifyTemplate() function by offloading any of its processing into the template creation function.	Statistics of the time needed for this operation. Accuracy metrics - Type I + II error rates. Failure rates.

8.4. API

8.4.1. Interface

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

	C++ code fragment	Remarks
1.	`Class Interface`	
2.	`{` `public:`	

3.	static std::shared_ptr<Interface> getImplementation();	Factory method to return a managed pointer to the `Interface` object. This function is implemented by the submitted library and must return a managed pointer to the `Interface` object.
4.	// Other functions to implement	
5.	};	

There is one class (static) method declared in `Interface.getImplementation()` which must also be implemented. 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 “frvt1N.h”
	
	using namespace FRVT_1N;
	
	NullImpl:: NullImpl () { }
	
	NullImpl::~ NullImpl () { }
	
	std::shared_ptr<Interface>
	Interface::getImplementation()
	{
	    return std::make_shared<NullImpl>();
	}
	// Other implemented functions	

8.4.2. Initialization of template creation

Before any feature extraction/template creation calls are made, the NIST test harness will call the initialization function of Table 6. This function will be called BEFORE any calls to fork() are made.

Table 6 – Template creation initialization

Prototype	ReturnStatus initializeTemplateCreation(	
	const std::string &configDir,	Input
	TemplateRole role);	Input
Description	This function initializes the implementation under test and sets all needed parameters in preparation for template creation. This function will be called N=1 times by the NIST application, prior to parallelizing M >= 1 calls to the template creation function via `fork()`. This function will be called from a single process/thread.	
Input Parameters	configDir	A read-only directory containing any developer-supplied configuration parameters or run-time data files.
	role	A value from the TemplateRole enumeration that indicates the intended usage of the template to be generated. In this case, either Enrollment_1N or Search_1N.
Output Parameters	None	
Return Value	See General Evaluation Specifications document for all valid return code values.	

8.4.3. Template Creation from one or more images of exactly one person

The functions of Table 7 supports role-specific generation of template data from one or more images of exactly one person. A vector of face or iris or face+iris images is converted to a single template using this function.

NOTE: For any given submission, developers may only implement ONE of the functions in Table 7. That is, a single submission may only support face recognition or iris recognition or multimodal recognition. For the functions that are not implemented, the function shall return ReturnCode::NotImplemented.

Some of the proposed datasets include 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. Use of multiple images per person has been shown to elevate accuracy over a single image.

For this test, NIST will enroll K >= 1 images under each identity. Normally the probe will consist of a single image, but NIST may examine the case that it could consist of multiple images. Ordinarily, the probe images will be captured after the enrolled images of a person. The method by which the face recognition implementation exploits multiple images is not regulated. The test seeks to evaluate developer provided technology for multi-presentation fusion.

This document defines a template to be the result of applying feature extraction to a set of K >= 1 images. An algorithm might internally fuse K feature sets into a single model or maintain them separately - in any case the resulting proprietary template is contained in a contiguous block of data. All identification functions operate on such multi-image templates.

Table 7 – Template Creation/Feature Extraction from one or more images of exactly one person

Prototype for face recognition	ReturnStatus createFaceTemplate(	
	const std::vector<Image> &faces,	Input
	TemplateRole role,	Input
	std::vector<uint8_t> &templ,	Output
	std::vector<EyePair> &eyeCoordinates);	Output
Prototype for iris recognition	ReturnStatus createIrisTemplate(	
	const std::vector<Image> &irises,	Input
	TemplateRole role,	Input
	std::vector<uint8_t> &templ,	Output
	std::vector<IrisAnnulus> &irisLocations);	Output
Prototype for multimodal face + iris recognition	ReturnStatus createFaceAndIrisTemplate(	
	const std::vector<Image> &facesIrises,	Input
	TemplateRole role,	Input
	std::vector<uint8_t> &templ);	Output
Description	Takes a vector of image(s) and outputs a proprietary template and associated coordinates. The vector to store the template will be initially empty, and it is up to the implementation to populate it with the appropriate data. *For enrollment templates (TemplateRole=Enrollment_1N)*: If the function executes correctly (i.e., returns a successful return code), the template will be enrolled into a gallery. The NIST calling application may store the resulting template, concatenate many templates, and pass the result to the enrollment finalization function (see section 8.4.5). The resulting template may also be inserted immediately into previously finalized gallery. When the implementation fails to produce a template (i.e., returns a non-successful return code), it shall still return a blank template (which can be zero bytes in length). The template will be included in the enrollment database/manifest like all other enrollment templates but is not expected to contain any feature information. IMPORTANT: NIST's application writes the template to disk. Any data needed during subsequent searches should be included in the template or created from the templates during the enrollment finalization function of section 8.4.5. *For identification/probe templates (TemplateRole=Search_1N)*: The NIST calling application may commit the template to permanent storage or may keep it only in memory (the developer implementation does not need to know). If the function returns a non-successful return status, the output template will not be used in subsequent search operations.	
Input Parameters	faces, irises, or faceIrises	Input face, iris, or face+iris images **Note:** For multimodal (face+iris), the implementation must handle some unimodal samples - for example, a gallery for which 80% of enrolled samples are face and iris, but 10% of samples are face-only, and 10% are iris-only.
	role	Label describing the type/role of the template to be generated. In this case, it will either be Enrollment_1N or Search_1N.
Output Parameters	templ	The output template. The format is entirely unregulated. This will be an empty vector when passed into the function, and the implementation can resize and populate it with the appropriate data.
	eyeCoordinates or irisLocations	The function shall return – For face images, eye coordinates – the estimated eye centers for left and right eyes

		– For iris images – iris locations - estimates of the limbus center and pupil and limbus radii
Return Value	See General Evaluation Specifications document for all valid return code values.	

8.4.4. Template Creation of one or more people detected from a face image

This function supports role-specific generation of one or more templates that correspond to one or more people's faces are detected in an image. Some of the proposed test images include K > 1 persons for some images and situations where the subject of interest may or may not be the foreground face (largest face in the image). This function allows the implementation to return a template for each person detected in the image. For testing, NIST will

1. Enroll one more templates from a single call to this function or the function of Table 7
2. Generate one or more search templates from a single call to this function or the function of Table 7
3. Search all templates generated from 2) against the enrollment database
4. Use the **maximum** similarity score or best rank across all searches from 3) in our calculation of FNIR and FPIR (this applies to both genuine and imposter searches)

NOTE 1: The implementation must be able to match any combination of enrollment and search templates generated from this function and the function of Table 7. In other words, the output template format should be consistent between this function and the function of Table 7.

NOTE 2: This function will not be called with iris images.

Table 8 – Template Creation/Feature Extraction of one or more people detected from an image

Prototypes	ReturnStatus createFaceTemplate(	
	const Image &image,	Input
	TemplateRole role,	Input
	std::vector<std::vector<uint8_t>> &templs,	Output
	std::vector<EyePair> &eyeCoordinates);	Output
Description	This function supports template generation of one or more people detected from a single image. It takes a single input image and outputs one or more proprietary templates and associated eye coordinates based on the number of people detected. The vectors to store the template(s) and eye coordinates will be initially empty, and it is up to the implementation to populate them with the appropriate data. *For enrollment templates (TemplateRole=Enrollment_1N)*: If the function executes correctly (i.e. returns a successful return code), the template(s) will be enrolled into a gallery. The NIST calling application may store the resulting template(s), concatenate many templates, and pass the result to the enrollment finalization function (see section 8.4.5). The resulting template(s) may also be inserted immediately into previously finalized gallery. When the implementation fails to produce a template (i.e. returns a non-successful return code), it shall still return a blank template (which can be zero bytes in length). The template will be included in the enrollment database/manifest like all other enrollment templates, but is not expected to contain any feature information. IMPORTANT: NIST's application writes the template to disk. Any data needed during subsequent searches should be included in the template, or created from the templates during the enrollment finalization function of section 8.4.5. *For identification/probe templates (TemplateRole=Search_1N)*: The NIST calling application may commit the template(s) to permanent storage, or may keep it only in memory (the developer implementation does not need to know). If the function returns a non-successful return status, the output template(s) will not be used in subsequent search operations.	
Input Parameters	image	A single image that contains one or more people in the photo
	role	Label describing the type/role of the template to be generated. In this case, it will either be Enrollment_1N or Search_1N.
Output Parameters	templs	A vector of output template(s). The format of the template(s) is entirely unregulated. This will be an empty vector when passed into the function, and the implementation can resize and populate it with the appropriate data.

	eyeCoordinates	For each person detected in the image, the function shall return the estimated eye centers. This will be an empty vector when passed into the function, and the implementation shall populate it with the appropriate number of entries. Values in eyeCoordinates[i] shall correspond to templs[i].
Return Value	See General Evaluation Specifications document for all valid return code values.	

8.4.5. Finalization

After all templates have been created, the function of Table 9 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 will be readable after the call. Implementations must, at a minimum, copy the input data or otherwise extract what is needed for search.

Table 9 – Enrollment finalization

Prototypes	ReturnStatus finalizeEnrollment(	
	const std::string &configDir,	Input
	const std::string &enrollmentDir,	Input
	const std::string &edbName,	Input
	const std::string &edbManifestName,	Input
	GalleryType galleryType);	Input
Description	This function takes the name of the top-level directory where the enrollment database (EDB) and its manifest have been stored. These are described in section 7.1. The enrollment directory permissions will be read + write. The function supports post-enrollment, developer-optional, book-keeping operations, 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. This function will be called from a single process/thread. Implementation of this function does not need to be single-threaded (i.e., developers may use multiple threads within this function).	
Input Parameters	configDir	A read-only directory containing any developer-supplied configuration parameters or run-time data files.
	enrollmentDir	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 7.1. While the file will have read-write-delete permission, the implementation 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 7.1. 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.
	galleryType	A label from Table 3 specifying the composition of the gallery.
Output Parameters	None	
Return Value	See General Evaluation Specifications document for all valid return code values.	

8.4.6. Search Initialization

The function of Table 10 will be called once prior to one or more calls of the searching function of Table 11 and the gallery insert and delete functions of Section 0. The function might set static internal variables so that the enrollment database is available to the subsequent identification searches. This function will be called BEFORE any calls to fork() are made.

Table 10 – Identification initialization

Prototype	ReturnStatus initializeIdentification(	
	const string &configDir,	Input
	const string &enrollmentDir);	Input
Description	This function reads whatever content is present in the enrollmentDir, for example a manifest placed there by the finalizeEnrollment() function. This function will be called from a single process/thread.	
Input Parameters	configDir	A read-only directory containing any developer-supplied configuration parameters or run-time data files.
	enrollmentDir	The read-only top-level directory in which enrollment data was placed. This directory will contain the gallery files produced from the **finalize()** function. The enrollment directory will always contain a successfully finalized gallery (i.e. will never be empty).
Return Value	See General Evaluation Specifications document for all valid return code values.	

8.4.7. Search

The function of Table 11 compares a proprietary identification template against the enrollment data and returns a candidate list.

Table 11 – Identification search

Prototype	ReturnStatus identifyTemplate (	
	const std::vector<uint8_t> &idTemplate,	Input
	const uint32_t candidateListLength,	Input
	std::vector<Candidate> &candidateList);	Output
Description	This function searches a template against the enrollment set, and outputs a list of candidates. The candidateList vector will initially be empty, and the implementation shall populate the vector with candidateListLength entries.	
Input Parameters	idTemplate	A template generated from the template creation function - 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 section 7.1.2. Each candidate shall be populated by the implementation. The candidates shall appear in descending order of similarity - i.e. most similar entries appear first.
Return Value	See General Evaluation Specifications document for all valid return code values.	

NOTE: Ordinarily the calling application will set the input candidate list length to operationally typical values, say $0 \leq L \leq 200$, and $L << N$. We will measure the dependence of search duration on L.