Face Recognition Prize Challenge (FRPC)

Still Face
Concept, Evaluation Plan and API

Version 3.0

All updates to this version of the document are highlighted in yellow.

Patrick Grother and Mei Ngan

Contact via frpc@nist.gov

Image Group
Information Access Division
Information Technology Laboratory

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

May 22, 2017

Table of Contents

List of Tables

1. Face Recognition Prize Challenge

1.1. Roles of IARPA and NIST

IARPA directs the FRPC and awards the prizes. NIST is the test laboratory implementing the FRPC for IARPA. Prospective participants in the FRPC should consult the following IARPA documents before reading this document.

— IARPA's FRPC challenge.gov Homepage

— IARPA's FRPC Homepage

— IARPA's FRPC Rules

1.2. Scope

This document establishes a concept of operations and an application programming interface (API) for evaluation of face recognition (FR) implementations submitted to the Face Recognition Prize Challenge (FRPC). There are two challenges within FRPC, named "Challenge IDENT" and "Challenge VERIF". Respectively, these are intended to attract the most accurate one-to-many identification and one-to-one verification face recognition algorithms.

1.3. Audience

Participation in FRPC is open to any organization worldwide, subject to a few restrictions (see [IARPA-FRPC]. There is no charge for participation. The target audience is researchers and developers of FR algorithms. While NIST intends to evaluate stable technologies that could be readily made operational, the test is also open to experimental, prototype and other technologies. All algorithms **must** be submitted as implementations of the APIs defined in this document.

1.4. Important Dates

Algorithms must be submitted to NIST by the date given on the IARPA challenge.gov website.

1.5. Rules for participation

1.5.1. Participation agreement

A participant must properly follow, complete, and submit the FRPC Participation Agreement (available from the FRPC website). This must be done once, either prior or in conjunction with the very first algorithm submission. It is not necessary to do this for each submitted implementation thereafter.

1.5.2. Options for participation

All submissions shall implement exactly one of the functionalities defined in Table 1. A library shall not implement the API of more than one challenge class.

Table 1 – FRPC Challenge Participation

Function	Challenge IDENT	Challenge VERIF
API requirements	3.2	3.3

1.5.3. Number of submissions

Participants may submit zero, one, or two (0 - 2) algorithms to Challenge IDENT. Participants may enter zero or one (0 - 1) algorithms to Challenge VERIF.

1.5.4. Validation

All participants must run their software through the provided FRPC validation package prior to submission. The validation package will be made available at https://github.com/usnistgov/frpc. The purpose of validation is to ensure consistent algorithm output between the participant's execution and NIST's execution.

1.6. Reporting

IARPA will announce the winners of the Prize Challenge. NIST may additionally report results in workshops, conferences, conference papers and presentations, journal articles and technical reports.

> **Important:** This is an open test in which NIST will identify the algorithm and the developing organization. Algorithm results will be attributed to the developer. Results will be machine generated (i.e. scripted) and will include timing, accuracy and other performance results. These will be posted alongside results from other implementations.

1.7. 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. 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, though NIST will handle all multiprocessing work via `fork()`[1]. Participant-initiated multiprocessing is not permitted.

NIST is requiring use of 64 bit implementations throughout.

1.7.1. Central Processing Unit (CPU)-only platforms

Algorithms running only on CPUs will be executed on machines equipped with Intel Xeon X5690 3.47 GHz CPUs.

1.7.2. Duel Intel Xeon E5-2630 v4 2.2 GHz - Graphics Processing Units (GPU)-enabled platforms

Algorithms running on GPUs will be executed on machines equipped with

- Intel Xeon E5-2695 v3 3.3 GHz CPUs and
- Dual NVIDIA Tesla K40 GPUs with 12GB of memory per GPU.

All GPU-enabled machines will be running CUDA version 7.5. cuDNN v5 for CUDA 7.5 will also be installed on these machines. Implementations that use GPUs will only be run on GPU-enabled machines.

1.8. 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/CentOS-7-x86_64-Everything-1511.iso, which is the 64-bit version of CentOS 7.2 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 CentOS 7.2.

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 dynamically-linkable using the C++11 compiler, g++ version 4.8.5.

A typical link line might be

```
g++ -std=c++11 -I. -Wall -m64 -o frpc frpc.cpp -L. –lfrpc_1N_acme_0_cpu
```

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

```
#include <frpc.h>
```

The header files will be made available to implementers at https://github.com/usnistgov/frpc.

All compilation and testing will be performed on x86_64 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.).

[1] http://man7.org/linux/man-pages/man2/fork.2.html

1.9. Software and documentation

1.9.1. Library and platform requirements

Participants shall provide NIST with binary code only (i.e. no source code). The implementation should be submitted in the form of a dynamically-linked library file.

The core library shall be named according to Table 2. Additional supplemental libraries may be submitted that support this "core" library file (i.e. the "core" library file may have dependencies implemented in these other libraries). Supplemental libraries may have any name, but the "core" library must be dependent on supplemental libraries in order to be linked correctly. The **only** library that will be explicitly linked to the FRPC test driver is the "core" library.

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 running on CPUs that do not support IPP. Please take note that some IPP functions are multithreaded and threaded implementations are prohibited.

NIST will report the size of the supplied libraries.

Table 2 – Implementation library filename convention

Form	libfrpc_challenge_provider_sequence_processor.ending					
Underscore delimited parts of the filename	libfrpc	challenge	provider	sequence	processor	ending
Description	First part of the name, required to be this.	"1N" for IDENT implementation "11" for VERIF implementation	Single word, non-infringing name of the main provider EXAMPLE: Acme	A one digit decimal identifier to start at 0 and incremented by 1 for each library sent to NIST.	"gpu" if implementation uses GPUs; "cpu" otherwise	.so
Example	libfrpc_1N_acme_0_cpu.so					

1.9.2. 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.

1.9.3. Submission folder hierarchy

Participant submissions shall 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
- validation/ - contains validation output

1.9.4. Installation and usage

The implementation shall be installable using simple file copy methods. It shall not require the use of a separate installation program and shall be executable on any number of machines without requiring additional machine-specific license control procedures or activation. The implementation shall not use nor enforce any usage controls or limits based on licenses, number of executions, presence of temporary files, etc. The implementation shall remain operable for at least six months from the submission date.

1.9.5. 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.

1.9.6. 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.

1.10. Runtime behavior

1.10.1. 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”.
- Only 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.

1.10.2. 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.

1.10.3. 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, nor otherwise manipulate it. 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.

1.10.4. Stateless behavior

All components in this test shall be stateless, except as noted. This applies to face detection, feature extraction and matching. 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.

1.11. Single-thread requirement and parallelization

Implementations must run in single-threaded mode, because NIST will parallelize the test by dividing the workload across many cores and many machines. Implementations must ensure that there are no issues with their software being parallelized via the `fork()` function – this applies to both GPU and CPU implementations submitted to FRPC.

For implementations using the GPU: For any given GPU, NIST will run a single implementation process (i.e., fork() once per GPU), with 12GB of main memory available for use by the algorithm. NIST machines are equipped with dual GPUs, and the NIST test harness will load balance by telling the implementation which GPU to use via the section 3.2.2.3 setGPU() function call. All calls to setGPU() will be performed after a call to fork(). Implementations using the GPU are encouraged to perform initialization within the setGPU() function where 1. which GPU to use is provided to the implementation and 2. to support known limitations of commonly used deep learning frameworks such as Caffe, where initialization must take place in the worker process.

1.12. Time limits

The elemental functions of the implementations shall execute under the time constraints of Table 3. 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. Timing will be estimated from at least 1000 separate invocations of each elemental function.

The time limits apply per image.

Table 3 – Processing time limits in milliseconds, per 640 x 480 color image, on a single CPU or GPU

Function	Challenge IDENT (1:N)	Challenge VERIF (1:1)
Template Generation	2000	2000
1:N finalization (on gallery of 100K enrolled templates)	3600000	NA
1:N Search (on 100K enrolled templates)	25000	NA
1:1 Comparison	NA	1

2. Data structures supporting the API

2.1. Requirement

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

2.2. File formats and data structures

2.2.1. Overview

In this face recognition test, an individual is represented by a K = 1 two-dimensional facial image. Most images will contain exactly face. In a small fraction of the images, other, smaller, faces will appear in the background. Algorithms should detect one foreground face in each image and produce one template.

Table 4 – Structure for a single image

C++ code fragment	Remarks
`typedef struct Image`	
`{`	
`uint16_t width;`	Number of pixels horizontally
`uint16_t height;`	Number of pixels vertically
`uint16_t depth;`	Number of bits per pixel. Legal values are 8 and 24.
`std::shared_ptr<uint8_t> data;`	Managed 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`
`} Image;`	

2.2.2. Data structure for eye coordinates

Implementations have the option to 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 returning of eye coordinates is optional for implementations. For those who choose not to implement this, both isLeftAssigned and isRightAssigned should be set to false.

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 5 – Structure for a pair of eye coordinates

C++ code fragment	Remarks
`typedef struct EyePair`	

`{`	
`bool isLeftAssigned;`	If the subject's left eye coordinates have been computed and assigned successfully, this value should be set to true, otherwise false.
`bool isRightAssigned;`	If the subject's right eye coordinates have been computed and assigned successfully, this value should be set to true, otherwise false.
`uint16_t xleft;` `uint16_t yleft;`	X and Y coordinate of the center of the subject's left eye. If the eye coordinate is out of range (e.g. x < 0 or x >= width), `isLeftAssigned` should be set to false.
`uint16_t xright;` `uint16_t yright;`	X and Y coordinate of the center of the subject's right eye. If the eye coordinate is out of range (e.g. x < 0 or x >= width), `isRightAssigned` should be set to false.
`} EyePair;`	

2.2.3. Template role

Labels describing the type/role of the template to be generated will be provided as input to template generation. This supports asymmetric algorithms where the enrollment and recognition templates may differ in content and size.

Table 6 – Labels describing template role

Label as C++ enumeration	Meaning
`enum class TemplateRole {`	
`Enrollment_1N,`	Enrollment template for 1:N identification
`Search_1N,`	Search template for 1:N identification
`Enrollment_11,`	Enrollment template for 1:1 comparison
`Verification_11`	Verification template for 1:1 comparison
`};`	

2.2.4. 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 <climits> 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.

2.2.5. File structure for enrolled template collection

To support the Challenge IDENT (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 7. 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 7 – 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
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.

2.2.6. 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 8.

Table 8 – 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.
4.	`std::string templateId;`	The Template ID from the enrollment database manifest defined in clause 2.2.5.
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.	`} Candidate;`	

2.2.7. Data structure for return value of API function calls

Table 9 – Enumeration of return codes

Return code as C++ enumeration	Meaning
`enum class ReturnCode {`	
`Success=0,`	Success
`ConfigError=1,`	Error reading configuration files
`RefuseInput=2,`	Elective refusal to process the input, e.g. because cannot handle greyscale
`ExtractError=3,`	Involuntary failure to process the image, e.g. after catching exception
`ParseError=4,`	Cannot parse the input data
`TemplateCreationError=5,`	Elective refusal to produce a “non-blank” 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
`NumDataError=7,`	The implementation cannot support the number of images
`TemplateFormatError=8,`	Template file is in an incorrect format or defective
`EnrollDirError=9,`	An operation on the enrollment directory failed (e.g. permission, space)
`InputLocationError=10`	Cannot locate the input data – the input files or names seem incorrect
`GPUError=11,`	There was a problem setting or accessing the GPU
`VendorError=12`	Vendor-defined failure. Failure codes must be documented and communicated to NIST with the submission of the implementation under test.
`};`	

Table 10 – ReturnStatus structure

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

3. API specification

Please note that included with the FRPC validation package (available at https://github.com/usnistgov/frpc) 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.

3.1. Namespace

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

3.2. Challenge IDENT (1:N)

3.2.1. Overview

The 1:N identification application proceeds in two phases, enrollment and identification. The identification phase includes separate probe 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 11 – Procedural overview of the Challenge IDENT (1:N) test

Phase	#	Name	Description	Performance Metrics to be reported by NIST
Enrollment	E1	Initialization	**initializeEnrollmentSession()** 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	**createTemplate(TemplateRole=Enrollment_1N)** For each of N individuals, pass K = 1 image 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. 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	**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.	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 implementation is permitted **read-write-delete access** to the enrollment directory during this phase.	the preceding enrollment times.
Probe Template Creation	S1	Initialization	**initializeProbeTemplateSession()** 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	**createTemplate(TemplateRole=Search_1N)** For each probe, create a template from K = 1 image. This operation will generally be conducted in a separate process invocation to step S3. 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	**initializeIdentificationSession()** 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	**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 `createTemplate()` function.	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 software under test must implement the interface `IdentInterface` by subclassing this class and implementing each method specified therein.

	C++ code fragment	Remarks
1.	`Class IdentInterface`	
2.	`{` `public:`	
3.	`virtual ReturnStatus initializeEnrollmentSession(` `const std::string &configDir) = 0;`	

4.	`virtual ReturnStatus createTemplate(`
	`const Image &face,`
	`TemplateRole role,`
	`std::vector<uint8_t> &templ,`
	`EyePair &eyeCoordinates) = 0;`	

5.	`virtual ReturnStatus finalizeEnrollment(`
	`const std::string &enrollmentDir,`
	`const std::string &edbName,`
	`const std::string &edbManifestName) = 0;`	

6.	`virtual ReturnStatus initializeProbeTemplateSession(`
	`const std::string &configDir,`
	`const std::string &enrollmentDir) = 0;`	

7.	`virtual ReturnStatus initializeIdentificationSession(`
	`const std::string &configDir,`
	`const std::string &enrollmentDir) = 0;`	

8.	`virtual ReturnStatus identifyTemplate(`
	`const TattooRep &idTemplate,`
	`const uint32_t candidateListLength,`
	`std::vector<Candidate> &candidateList,`
	`bool &decision) = 0;`	

9.	`virtual ReturnStatus setGPU(uint8_t gpuNum) = 0;`	
10.	`static std::shared_ptr<IdentInterface> 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.
11.	`};`	

There is one class (static) method declared in `IdentInterface.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 "frpc.h"`
	
	`using namespace FRPC;`
	
	`NullImpl:: NullImpl () { }`
	
	`NullImpl::~ NullImpl () { }`
	
	`std::shared_ptr<IdentInterface>`
	`IdentInterface::getImplementation()`
	`{`
	`    return std::make_shared<NullImpl>();`
	`}`
	
	`// Other implemented functions`	

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 12. This function will be called BEFORE any calls to fork() are made.

Table 12 – Enrollment initialization

Prototype	ReturnStatus initializeEnrollmentSession(const std::string &configDir);		 Input
Description	This function initializes the implementation under test and sets all needed parameters. This function will be called N=1 times by the NIST application, prior to parallelizing M >= 1 calls to `createTemplate(TemplateRole=Enrollment_1N)` via `fork()`.		
Input Parameters	configDir	A read-only directory containing any developer-supplied configuration parameters or run-time data files.	
Output Parameters	None		
Return Value	See Table 9 for all valid return code values.		

3.2.2.3. GPU Index Specification

For implementations using GPUs, the function of Table 13 specifies a sequential index for which GPU device to execute on. This enables the test software to orchestrate load balancing across multiple GPUs. This function will be called AFTER a call to fork() is made.

Table 13 – GPU index specification

Prototypes	ReturnStatus 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.
Return Value	See Table 9 for all valid return code values.	

3.2.2.4. Enrollment

An Image is converted to a single enrollment template using the function of Table 14. For the more information regarding the types of imagery that will be used, please refer to the FRPC Rules Document at https://www.challenge.gov/wp-content/uploads/2017/04/IARPA_NIST_FRPC_Rules.pdf.

Table 14 – Enrollment feature extraction

Prototypes	ReturnStatus createTemplate(	
	const Image &face,	Input
	TemplateRole role,	Input
	std::vector<uint8_t> &templ,	Output
	EyePair &eyeCoordinates);	Output
Description	Takes an Image and outputs a proprietary template and, optionally, associated eye 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 NIST calling application will store the template. The NIST application will concatenate the templates and pass the result to the enrollment finalization function (see section 13). 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 13 *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	face	Input face image
	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	(Optional) The function may choose to return the estimated eye centers for the input face image.
Return Value	See Table 9 for all valid return code values.	

3.2.2.5. Finalize enrollment

After all templates have been created, the function of Table 15 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 15 – Enrollment finalization

Prototypes	ReturnStatus finalizeEnrollment(	
	const std::string &enrollmentDir,	Input
	const std::string &edbName,	Input
	const std::string &edbManifestName);	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 2.2.5. 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.	
Input Parameters	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 2.2.5. 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 2.2.5. 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 9 for all valid return code values.	

3.2.2.6. Probe Template Feature Extraction Initialization

Before Images are sent to the identification feature extraction function, the test harness will call the initialization function in Table 16. This function will be called BEFORE any calls to fork() are made.

Table 16 – Probe template feature extraction initialization

Prototype	ReturnStatus initializeProbeTemplateSession(	
	const std::string &configDir,	Input
	const std::string &enrollmentDir);	Input
Description	This function initializes the implementation under test and sets all needed parameters. This function will be called once by the NIST application immediately before any M ≥ 1 calls to createTemplates(TemplateRole=Search_1N). The implementation has read-only access to its prior enrollment data.	
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 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 9 for all valid return code values.	

3.2.2.7. Search Initialization

The function of Table 17 will be called once prior to one or more calls of the searching function of Table 18. 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 17 – Identification initialization

Prototype	ReturnStatus initializeIdentificationSession(	
	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.	
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.
Return Value	See Table 9 for all valid return code values.	

3.2.2.8. Search

The function of Table 18 compares a proprietary identification template against the enrollment data and returns a candidate list.

Table 18 – Identification search

Prototype	ReturnStatus identifyTemplate (	
	const std::vector<uint8_t> &idTemplate,	Input
	const uint32_t candidateListLength,	Input
	std::vector<Candidate> &candidateList,	Output
	bool &decision);	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 from createTemplate(TemplateRole=Search_1N) - 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 2.2.6. 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.
	decision	A best guess at whether there is a mate within the enrollment database. If there was a mate found, this value should be set to true, Otherwise, false. Many such decisions allow a single point to be plotted alongside a DET.
Return Value	See Table 9 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.

3.3. Challenge VERIF (1:1)

3.3.1. Overview

The 1:1 testing will proceed in the following phases: optional offline training; preparation of enrollment templates; preparation of verification templates; and matching. Note that training, template creation, and matching may all be performed as separate processes. These are detailed in Table 19.

Table 19 – Functional summary of the Challenge VERIF (1:1) test

Phase	Description	Performance Metrics to be reported by NIST
Initialization	**initialize()** Function to read configuration data, if any.	None

Enrollment	**createTemplate(TemplateRole=Enrollment_11)** 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	**createTemplate(TemplateRole=Verification_11)** 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)	**matchTemplates()** 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, including for partitions of the input datasets.

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.

3.3.2. API

3.3.2.1. Interface

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

	C++ code fragment	Remarks
1.	`class VerifInterface`	
2.	`{` `public:`	
3.	`virtual ReturnStatus initialize(` `const std::string &configDir) = 0;`	

4.	`virtual ReturnStatus createTemplate(`
	`const Image &face,`
	`TemplateRole role,`
	`std::vector<uint8_t> &templ,`
	`EyePair &eyeCoordinates) = 0;`	

5.	`virtual ReturnStatus matchTemplates(`
	`const std::vector<uint8_t> &verifTemplate,`
	`const std::vector<uint8_t> &enrollTemplate,`
	`double &similarity) = 0;`	

6.	`virtual ReturnStatus setGPU(uint8_t gpuNum) = 0;`	
7.	`static std::shared_ptr<VerifInterface> 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 implementation. 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 "frpc.h"

using namespace FRPC;

NullImpl:: NullImpl () { }

NullImpl::~ NullImpl () { }

std::shared_ptr<VerifInterface>
VerifInterface::getImplementation()
{
    return std::make_shared<NullImpl>();
}

// Other implemented functions
```

3.3.2.2. Initialization

The NIST test harness will call the initialization function in Table 20 before calling template generation or matching. This function will be called BEFORE any calls to fork() are made.

Table 20 – Initialization

Prototype	ReturnStatus initialize(const std::string &configDir);	 Input
Description	This function initializes the implementation under test. It will be called by the NIST application before any call to `createTemplate()` or `matchTemplates()`. The implementation under test should set all parameters. This function will be called N=1 times by the NIST application, prior to parallelizing M >= 1 calls to `createTemplate()` via `fork()`.	
Input Parameters	configDir	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 9 for all valid return code values.	

3.3.2.3. GPU Index Specification

For implementations using GPUs, the function of Table 21 specifies a sequential index for which GPU device to execute on. This enables the test software to orchestrate load balancing across multiple GPUs. This function will be called AFTER a call to fork() is made.

Table 21 – GPU index specification

Prototypes	ReturnStatus 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.
Return Value	See Table 9 for all valid return code values.	

3.3.2.4. Template generation

The function of Table 22 supports role-specific generation of a template data. Template format is entirely proprietary. For the more information regarding the types of imagery that will be used, please refer to the FRPC Rules Document at https://www.challenge.gov/wp-content/uploads/2017/04/IARPA_NIST_FRPC_Rules.pdf.

Table 22 – Template generation

Prototypes	ReturnStatus createTemplate(	
	const Image &face,	Input
	TemplateRole role,	Input
	std::vector<uint8_t> &templ,	Output
	EyePair &eyeCoordinates);	Output
Description	Takes an Image and outputs a proprietary template and optionally, associated eye 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. In all cases, even when unable to extract features, the output shall 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	face	Input face image
	role	Label describing the type/role of the template to be generated. In this case, it will either be Enrollment_11 or Verification_11.
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	(Optional) The function may choose to return the estimated eye centers for the input face image.
Return Value	See Table 9 for all valid return code values.	

3.3.2.5. Matching

Matching of one enrollment against one verification template shall be implemented by the function of Table 23.

Table 23 – Template matching

Prototype	ReturnStatus matchTemplates(	
	const std::vector<uint8_t> &verifTemplate,	Input
	const std::vector<uint8_t> &enrollTemplate,	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 22), the similarity score shall be -1 and the function return value shall be `VerifTemplateError`.	
Input Parameters	verifTemplate	A verification template from createTemplate(role=Verification_11). The underlying data can be accessed via verifTemplate.data(). The size, in bytes, of the template could be retrieved as verifTemplate.size().
	enrollTemplate	An enrollment template from createTemplate(role=Enrollment_11). The underlying data can be accessed via enrollTemplate.data(). The size, in bytes, of the template could be retrieved as enrollTemplate.size().
Output Parameters	similarity	A similarity score resulting from comparison of the templates, on the range [0,DBL_MAX]. See section 2.2.4.
Return Value	See Table 9 for all valid return code values.	