Face Recognition Vendor Test
MORPH

Performance of Automated Facial Morph Detection and Morph Resistant Face Recognition Algorithms

Concept, Evaluation Plan and API

VERSION 1.1

Updates since the last version of this document are highlighted in cyan.

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

September 6, 2018

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

Table of Contents

List of Tables

1. MORPH

1.1. Scope

Facial morphing (and the ability to detect it) is an area of high interest to a number of photo-credential issuance agencies and those employing face recognition for identity verification. The FRVT MORPH test will provide ongoing independent testing of prototype facial morph detection technologies. The evaluation is designed to obtain an assessment on morph detection capability to inform developers and current and prospective end-users. This document establishes a concept of operations and an application programming interface (API) for evaluation of two separate tasks:

1. Algorithmic capability to detect facial morphing (morphed/blended faces) in still photographs
    a. Single-image morph detection of non-scanned photos, printed-and-scanned photos, and images of unknown photo format/origin
    b. Two-image differential morph detection of non-scanned photos, printed-and-scanned photos, and images of unknown photo format/origin
2. Face recognition algorithm resistance against morphing

1.2. Audience

Participation is open to any organization worldwide involved in development of morph detection 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 C++ API defined in this document. There is no charge for participation.

1.3. Reporting

For all algorithms that complete the evaluation, NIST will provide performance results back to the participating organizations. NIST may additionally report and share results with partner government agencies and interested parties, and in workshops, conferences, conference papers, presentations and technical reports.

Important: This is a 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 provided alongside results from other implementations. Results will be expanded and modified as additional implementations are tested, and as analyses are implemented. Results may be regenerated on-the-fly, usually whenever additional implementations complete testing, or when new analyses are added.

1.4. Accuracy metrics

This test will evaluate algorithmic ability to detect whether an image is a morphed/blended image of two or more faces and/or to correctly reject 1:1 comparisons of morphed images against other images of the subjects used to create the morph (but similarly, correctly authenticate legitimate non-morphed, mated pairs and correctly reject non-morphed, non-mated pairs). Per established metrics[1,2] for assessment of morphing attacks, NIST will compute and report:

[1] International Organization for Standardization: Information Technology – Biometric presentation attack detection – Part 3: Testing and reporting. ISO/IEC FDIS 30107-3:2017, JTC 1/SC 37, Geneva, Switzerland, 2017

[2] U. Scherhag, A. Nautsch, C. Rathgeb, M. Gomez-Barrero, R. Veldhuis, L. Spreeuwers, M. Schils, D. Maltoni, P. Grother, S. Marcel, R. Breithaupt, R. Raghavendra, C. Busch: "Biometric Systems under Morphing Attacks: Assessment of Morphing Techniques and Vulnerability Reporting", in Proceedings of the IEEE 16th International Conference of the Biometrics Special Interest Group (BIOSIG), Darmstadt, September 20-22, (2017)

- Attack Presentation Classification Error Rate (APCER) – the proportion of morph attack samples incorrectly classified as bona fide presentation
- Bona Fide Presentation Classification Error Rate (BPCER) – the proportion of bona fide samples incorrectly classified as morphed samples
- Mated Morph Presentation Match Rate (MMPMR) - the proportion of comparisons where the morphed image successfully authenticates against all constituents
- True Acceptance Rate (TAR) – the proportion of non-morphed, mated comparisons that correctly authenticate
- False Match Rate (FMR) – the proportion of non-morphed, non-mated comparisons that incorrectly authenticate

We will report the above quantities as a function of alpha (the fraction of each subject that contributed to the morph), image compression ratio, image resolution, image size, and others.

We will also report error tradeoff plots (BPCER vs. APCER, MMPMR vs. FMR, parametric on threshold).

2. Rules for participation

2.1. Implementation Requirements

Developers are not required to implement all functions specified in this API. Developers may choose to implement one or more functions of this API – please refer to Section 4.2.1 for detailed information regarding implementation requirements.

2.2. Participation agreement

A participant must properly follow, complete, and submit the FRVT MORPH Participation Agreement. 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.

2.3. Number and Schedule of Submissions

Currently, the number and schedule of submissions is not regulated, so participants can send submissions at any time. NIST reserves the right to amend this section with submission volume and frequency limits. NIST will evaluate implementations on a first-come-first-served basis and provide results back to the participants as soon as possible.

2.4. Validation

All participants must run their software through the provided FRVT MORPH validation package prior to submission. The validation package will be made available at https://github.com/usnistgov/frvt. The purpose of validation is to ensure consistent algorithm output between the participant's execution and NIST's execution. Our validation set is not intended to provide training or test data.

2.5. 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 machine has at least 192 GB of memory. We anticipate that 16 processes can be run without time slicing, though NIST will handle all multiprocessing work via `fork()`[3]. Participant-initiated multiprocessing is not permitted.

All implementations shall use 64-bit addressing.

NIST intends to support highly optimized algorithms by specifying the runtime hardware. There are several types of

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

computers that may be used in the testing.

2.5.1. Central Processing Unit (CPU)-only platforms

The following list gives some details about the hardware of each CPU-only blade type:

- Dual Intel® Xeon® CPU E5-2630 v4 @ 2.2GHz (10 cores each)[4]
- Dual Intel® Xeon® CPU E5-2680 v4 @ 2.4GHz (14 cores each)[4]

This test will not support the use of Graphics Processing Units (GPUs). NIST intends on running algorithms over a very large number of CPU cores to support large-scale, timely test execution.

2.6. 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, MacOS and 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 frvt_morph frvt_morph.cpp -L. -lfrvtmorph_acme_000.so
```

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

The header files will be made available to implementers at https://github.com/usnistgov/frvt. All algorithm submissions will be built against the officially published header files – developers should not alter the header files when compiling and building their libraries.

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.).

2.7. Software and documentation

2.7.1. 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 1. 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 FRVT MORPH test driver is the "core" library.

---

[4] cat /proc/cpuinfo returns fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch ida arat epb pln pts dtherm tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm rdseed adx smap xsaveopt cqm_llc cqm_occup_llc

Developers may obviously use common deep learning frameworks (e.g. Caffe, TensorFlow, etc.) and should submit those dependencies as supplemental libraries. NIST has successfully received and run implementations leveraging such deep learning frameworks in other evaluations with no issues.

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 1 – Implementation library filename convention

Form	libfrvtmorph_provider_sequence.ending			
Underscore delimited parts of the filename	libfrvtmorph	provider	sequence	ending
Description	First part of the name, required to be this.	Single word, non-infringing name of the main provider EXAMPLE: Acme	A three digit decimal identifier to start at 000 and incremented by 1 every time a library is sent to NIST. EXAMPLE: 007	.so
Example	libfrvtmorph_acme_007.so			

Important: Results will be attributed with the provider name and the 3-digit sequence number in the submitted library name.

2.7.2. Configuration and developer-defined data

The implementation under test may be supplied with configuration files and supporting data files. These might include, for example, model, calibration or background feature data. NIST will report the size of the supplied configuration files.

2.7.3. 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

2.7.4. 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 twelve months from the submission date.

2.8. Runtime behavior

2.8.1. 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.

2.8.2. 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.

2.8.3. 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.

2.8.4. 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.

2.8.5. 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.

2.8.6. 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.

3. Data structures supporting the API

3.1. Requirement

FRVT MORPH participants should implement the relevant C++ prototyped interfaces of section 4. C++ was chosen in order to make use of some object-oriented features. Any functions that are not implemented should return `ReturnCode::NotImplemented`.

3.2. File formats and data structures

3.2.1. Overview

In this test, an individual is represented by a K = 1 two-dimensional facial image. All images will contain exactly one face.

Table 2 – 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;	

3.2.2. ImageLabel describing the format of an image

Table 3 – Enumeration of image label

Return code as C++ enumeration	Meaning
enum class ImageLabel {	
Unknown=0,	Image origin is unknown or unassigned
NonScanned=1	Non-scanned photo
Scanned=2,	Printed-and-scanned photo
};	

3.2.3. Data type for similarity scores

1:1 comparison/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.

3.2.4. Data structure for return value of API function calls

Table 4 – Enumeration of return codes

Return code as C++ enumeration	Meaning
enum class ReturnCode {	
Success=0,	Success
ConfigError,	Error reading configuration files
RefuseInput,	Elective refusal to process the input, e.g. because cannot handle greyscale
ExtractError,	Involuntary failure to process the image, e.g. after catching exception
ParseError,	Cannot parse the input data
MatchError,	Error occurred during the 1:1 comparison operation
FaceDetectionError,	Unable to detect a face in the image
NotImplemented,	Function is not implemented

`VendorError`	Vendor-defined failure. Vendor errors shall return this error code and document the specific failure in the ReturnStatus.info string from Table 5.
`};`	

Table 5 – ReturnStatus structure

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

4. API specification

Please note that included with the FRVT MORPH 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.

4.1. Namespace

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

4.2. API

4.2.1. Implementation Requirements

Developers are not required to implement all functions specified in this API. Developers may choose to implement one or more functions of Table 6, but at a minimum, developers must submit a library that implements

1. `MorphInterface` of Section 4.2.2,
2. `initialize()` of Section 4.2.3, and
3. AT LEAST one of the functions from Table 6. For any other function that is not implemented, the function shall return `ReturnCode::NotImplemented`.

Table 6 – API Functions

Function	Section
detectMorph() – single image morph detection of • Non-scanned photo • Printed-and-scanned photo • Image of unknown format	4.2.4
detectMorphDifferentially() – two image differential morph detection of • Non-scanned photo • Printed-and-scanned photo • Image of unknown format	4.2.5
compareImages() – 1:1 comparison	4.2.6
trainMorphDetector() – training for morph detection	4.2.7

4.2.2. Interface

The software under test **must** implement the interface `MorphInterface` by subclassing this class and implementing AT LEAST ONE of the methods specified therein.

C++ code fragment	Remarks
`Class MorphInterface`	
`{` `public:`	
`static std::shared_ptr<MorphInterface> getImplementation();`	Factory method to return a managed pointer to the `MorphInterface` object. This function is implemented by the submitted library and must return a managed pointer to the `MorphInterface` object.
`// Other functions to implement`	
`};`	

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

4.2.3. Initialization

Before any morph detection or matching calls are made, the NIST test harness will call the initialization function of Table 7. This function will be called BEFORE any calls to fork() are made. This function <u>must</u> be implemented.

Table 7 – Initialization

Prototype	ReturnStatus initialize(	
	const std::string &configDir,	Input
	const std::string& configValue);	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 any morph detection or matching functions 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.
	configValue	An optional string value encoding algorithm-specific configuration parameters. Developers may provide documentation for such configuration parameter(s) in their submission to NIST. Otherwise, the default value for this parameter will be an emptry string.
Output Parameters	None	
Return Value	See Table 4 for all valid return code values. This function <u>must</u> be implemented.	

4.2.4. Single-image Morph Detection

The function of Table 8 evaluates morph detection on non-scanned photos, scanned photos, and photos of unknown formats. A single image along with an associated image label describing the image format/origin is provided to the function for detection of morphing. Both morphed images and non-morphed images will be used, which will support measurement of a morph attack presentation classification error rate (APCER) with a bona fide presentation classification error rate (BPCER).

Non-scanned photos

Non-scanned photos are digital images known to not have been printed and scanned back in. There are a number of operational use-cases for morph detection on such digital images.

Scanned photos

While there are existing techniques to detect manipulation of a digital image, once the image has been printed and scanned back in, it leaves virtually no traces of the original image ever being manipulated. So the ability to detect whether a printed-and-scanned image contains a morph warrants investigation.

Photos of unknown format

In some cases, the format and/or origin of the image in question is not known, so images with "unknown" labels will also be tested.

Multiple instances of the calling application may run simultaneously or sequentially. These may be executing on different computers.

Table 8 – Single-image Morph Detection

Prototypes	ReturnStatus detectMorph(	
	const Image &suspectedMorph,	Input
	const ImageLabel &label,	Input
	bool &isMorph,	Output
	double &score);	Output
Description	This function takes an input image and associated image label describing the image format/origin, and outputs a binary decision on whether the image is a morph and a "morphiness" score on [0, 1] indicating how confident the algorithm thinks the image is a morph, with 0 meaning confidence that the image is not a morph and 1 representing absolute confidence that it is a morph.	
Input Parameters	suspectedMorph	Input Image
	label	ImageLabel (Section 3.2.2) describing the format of the input image • NonScanned = non-scanned digital photo • Scanned = a photo that is printed, then scanned • Unknown = unknown photo format/origin
Output Parameters	isMorph	True if image contains a morph; False otherwise
	score	A score on [0, 1] representing how confident the algorithm is that the image contains a morph. 0 means certainty that image does not contain a morph and 1 represents certainty that image contains a morph.
Return Value	See Table 4 for all valid return code values. If this function is not implemented, the return code should be set to `ReturnCode::NotImplemented`. If this function is not implemented for a certain type of image, for example, the function supports non-scanned photos but not scanned photos, then the function should return `ReturnCode::NotImplemented` when the function is called with the particular unsupported image type.	

4.2.5.

4.2.5. Two-image Differential Morph Detection

Two face samples are provided to the function of Table 9 as input, the first being a suspected morphed facial image and the second image representing a known, non-morphed face image of one of the subjects contributing to the morph (e.g., live capture image from an eGate). This procedure supports measurement of whether algorithms can detect morphed images when additional information (provided as the second supporting known subject image) is provided.

Similar to single-image morph detection, the function of Table 9 will support non-scanned, scanned, and photos of unknown format/origin. The input image type will be specified by the associated ImageLabel input parameter.

Multiple instances of the calling application may run simultaneously or sequentially. These may be executing on different computers.

Table 9 – Two-image Differential Morph Detection

Prototypes	ReturnStatus detectMorphDifferentially(	
	const Image &suspectedMorph,	Input
	const ImageLabel &label,	Input
	const Image &probeFace,	Input
	bool &isMorph,	Output
	double &score);	Output
Description	This function takes two input images - a known unaltered/not morphed image of the subject (`probeFace`) and an image of the same subject that's in question (may or may not be a morph) (`suspectedMorph`) with an associated image label describing the image format/origin. This function outputs a binary decision on whether `suspectedMorph` is a morph (given `probeFace` as a prior) and a "morphiness" score on [0, 1] indicating how confident the algorithm thinks the `suspectedMorph` is a morph, with 0 meaning confidence that the `suspectedMorph` is not a morph and 1 representing absolute confidence that it is a morph.	
Input Parameters	suspectedMorph	Input Image
	label	ImageLabel (Section 3.2.2) describing the format of the suspected morph image • NonScanned = non-scanned digital photo • Scanned = a photo that is printed, then scanned • Unknown = unknown photo format/origin
	probeFace	An image of the subject known not to be a morph (e.g., live capture image)
Output Parameters	isMorph	True if image contains a morph; False otherwise
	score	A score on [0, 1] representing how confident the algorithm is that the image contains a morph. 0 means certainty that image does not contain a morph and 1 represents certainty that image contains a morph.
Return Value	See Table 4 for all valid return code values. If this function is not implemented, the return code should be set to `ReturnCode::NotImplemented`. If this function is not implemented for a certain type of image, for example, the function supports non-scanned photos but not scanned photos, then the function should return `ReturnCode::NotImplemented` when the function is called with the particular unsupported image type.	

4.2.6. 1:1 Comparison

Two face samples are provided to the function of Table 10 for one-to-one comparison of whether the two images are of the same subject. The expected behavior from the algorithm is to be able to correctly reject comparisons of morphed images against constituents that contributed to the morph. The goal is to show algorithm robustness against morphing alterations when morphed images are compared against other images of the subjects used for morphing. Comparisons of morphed images against constituents should return a low similarity score, indicating rejection of match. Comparisons of unaltered/non-morphed images of the same subject should return a high similarity score, indicating acceptance of match.

Multiple instances of the calling application may run simultaneously or sequentially. These may be executing on different computers.

Table 10 – 1:1 Comparison

Prototypes	ReturnStatus compareImages(	
	const Image &enrollImage,	Input
	const Image &verifImage,	Input
	double &similarity);	Output
Description	This function compares two images and outputs a similarity score. In the event the algorithm cannot perform the comparison operation, the similarity score shall be set to -1.0 and the function return code value shall be set appropriately.	
Input Parameters	enrollImage	The enrollment image
	verifImage	The verification image
Output Parameters	similarity	A similarity score resulting from comparison of the two images, on the range [0,DBL_MAX].
Return Value	See Table 4 for all valid return code values. If this function is not implemented, the return code should be set to `ReturnCode::NotImplemented`.	

4.2.7. Training for Morph Detection

For developers who implement the training function, NIST will run tests with and without training to assess the performance impacts of turn-key training. The training function of Table 11 will be invoked as a separate process outside of the morph detection and/or comparison process. So, given 1) K ≥ 1 images with associated labels on whether the photo is a morph or not and 2) the implementation's configuration directory, the implementation may use the provided training data to populate a new "trained" configuration directory. This directory will be used to initialize the algorithm during subsequent morph detection and/or comparison processes.

Please note that this function may or may not be called prior to morph detection or matching. The implementation's ability to detect a morph or match images should not be dependent on prior execution of this function.

This function will be called from a single process/thread.

Table 11 – Training

Prototype	ReturnStatus trainMorphDetector(	
	const std::string &configDir,	Input
	const std::string &trainedConfigDir,	Input
	const std::vector<Image> &faces,	Input
	const std::vector<bool> &isMorph);	Input
Description	This function provides the implementation a list of face images and whether they are morphs. This function may or may not be called prior to the various morph detection and/or matching functions. The implementation's ability to detect morphs should not be dependent on this 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. 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.

	trainedConfigDir	A directory with read-write permissions where the implementation can store any training output. 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. Important: This directory is what will subsequently be provided to the implementation's `initialize()` function as the input configuration directory if this training function is invoked. If this function is not implemented, the function shall do nothing, and the return code should be set to `ReturnCode::NotImplemented`.
	faces	A vector of face images provided to the implementation for training purposes
	isMorph	A vector of boolean values indicating whether the corresponding face image is a morph or not. The value in isMorph[i] corresponds to the face image in faces[i].
Output Parameters	none	
Return Value	See Table 4 for all valid return code values. If this function is not implemented, the return code should be set to `ReturnCode::NotImplemented`.	