FpVTE 2012

FINGERPRINT VENDOR TECHNOLOGY EVALUATION
PARTICIPANT'S GUIDE
VERSION 0.1

CRAIG WATSON
WAYNE SALAMON
GREGORY FIUMARA

IMAGE GROUP
INFORMATION ACCESS DIVISION
INFORMATION TECHNOLOGY LABORATORY

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

MARCH 22, 2012

Contents

Chapter 1

Introduction

1.1 Background

NIST has conducted several fingerprint related evaluations in the last decade. The first fingerprint evaluation was Proprietary Fingerprint Templates 2003 (PFT2003) which was also called SDK and more recently has changed to PFTII. PFTII is a one-to-one matching evaluation that looks at the core match capabilities of the matching software. It does not evaluate one-to-many capabilities.

Minutiae exchange (MINEX) was started in 2004, also a one-to-one matching evaluation, but MINEX evaluates the use of interoperable fingerprint templates. MINEX was started to support test fingerprint matching using INCITS 378 standard interoperable templates. About a year later Ongoing MINEX was establish to support Personal Identity Verification (PIV).

The first one-to-many fingerprint evaluation conducted at NIST was FpVTE2003. FpVTE2003 had participants bring their own hardware and software to NIST for the evaluation and NIST supplied the data and retained all the matching results for final analysis. [1]

1.2 Purpose

This document establishes the testing method for the one-to-many Fingerprint Vendor Technology Evaluation 2012 (FpVTE2012). In addition to describing the planned evaluation, this document includes the application program interface (API), software submission procedures, and participation application. See [2] for updates and additional documentation related to FpVTE2012.

FpVTE2003 only used enrollment sizes around 10,000 subjects. NIST has already conducted one-to-many biometric (face MBE2010 [3] and iris IREXIII [4]) evaluations using multi-million enrollment sizes in which all templates fit into the RAM of a single compute blade. FpVTE2012 will be the first biometric evaluation at NIST that partitions the enrollment set across multiple compute blades.

This technology evaluation is being done to meet several goals. FpVTE2012 will

- assess the current performance accuracy of one-to-many fingerprint matching software using operational fingerprint data,
- use enrolled sample sizes extending into the multiple millions,
- provide a testing framework and API for enrollment sizes that must be spread across the memory of multiple computer blades,
- support US Government and other sponsors in setting operational thresholds,
- evaluate on operational datasets containing newer datasets from live-scan ten-print "Identification Flats" capture systems, other live-scan capture devices (single finger and multi-finger), and historically significant scanned inked fingerprints. If available, datasets may include mobile device fingerprints.

[1] http://www.nist.gov/itl/iad/ig/fpvte2003.cfm
[2] http://www.nist.gov/itl/iad/ig/fpvte12.cfm
[3] http://www.nist.gov/itl/iad/ig/mbe.cfm
[4] http://www.nist.gov/itl/iad/ig/irex.cfm

1.3 Audience

Commercial, non-profit and research organizations with an ability to implement a large scale one-to-many fingerprint identification algorithm are invited to participate in FpVTE2012. Organizations only interested in one-to-one verification should participate in the Proprietary Fingerprint Testing II (PFTII). [5]

Participants will need to implement the API defined in this document. Participation is world wide and there is no charge. NIST expects to evaluate technologies that could be prototype or experimental.

1.4 Application Scenarios

All matching algorithms being tested in FpVTE2012 must first complete PFTII participation and have a FNMR less than 5% at FAR of 0.0001 on all four PFTII datasets. Based on current PFTII results 38 of the 46 SDKs tested would meet this threshold. PFTII is a one-to-one matching evaluation. The reason for this first step is that FpVTE2012 will require a significant investment in compute cycles and the minimum performance requirements for one-to-one matching should allow for trade offs in speed versus accuracy while minimizing the time spent testing algorithms that may not be ready for one-to-many matching. This will allow NIST to make the best use of its compute resources for the evaluation.

FpVTE2012 will evaluate one-to-many identification implementations. The plan is for the submitted software to perform segmentation of all plain impression images or some alternative matching without segmentation. Planned testing scenarios for FpVTE2012 include the follow:

- 1 and 2 plain fingers (right/left index) against enrolled database of 2 plain images. The plain images are from single finger captures.
- 10 rolled fingers against an enrolled database of 10 print rolled and plain images. Plain impression images will be 4-4-1-1, right/left four finger plain and right/left single thumb plain impression.
- 10 plain fingers against an enrolled database of 10 print rolled and plain images. Plain impression images will be 4-4-1-1, right/left four finger plain and right/left single thumb plain impression.
- 1, 2, 4, 8, 10 print Identification Flats (4-4-2, right/left four finger plain and two thumbs plain) against enrolled database of 10 print Identification Flats and 10 print rolled and plain (4-4-1-1) images.

Enrollment Sample Sizes

Search Data Type	Finger(s)	Enrollment Data Type	Number Enrolled Subjects
Single Plain Capture	2f Right and Left Index 1f Right or Left Index	2f Plain Capture	10,000,000
Ten Print Capture	10f Rolled 10f Plain (4-4-1-1)	10f Rolled 10f Plain	5,000,000 5,000,000
Identification Flats (4-4-2)	1f Right or Left Index 2f Right and Left Index 4f Right or Left Plain 8f Right and Left Plain 10f	10f Identification Flats 10f Rolled 10f Plain	3,000,000 3,500,000 3,500,000

1.5 Timeline

The following timeline is planned for FpVTE2012.

- March - Release draft API for public comments.

[5] http://www.nist.gov/itl/iad/ig/pftii.cfm

- **April 12 - Comments on Draft API due to NIST. Send to craig.watson@nist.gov.**
- April 27 - Release final API.
- End of May - Release driver software and validation data.
- End of May - Deadline to submit application to participate.
- June - Begin accepting SDKs for testing in phase I.
- October 31 - End of Phase I testing.
- December 7 - Deadline to submit final SDK for testing in phase II.
- March-April 2013 - Publication of results.

1.6 Offline Testing

While this test is intended to mimic operational reality, it remains an offline test performed on fixed sets of operational images maintained at NIST. The intent is to assess the core capability of the algorithms. The offline testing does allow for uniform, fair, repeatable, and efficient evaluation of underlying technologies. Testing under a fixed API allows for a detailed set of performance related parameters to be measured.

1.7 Phased Testing

To support research and development and to get robust and complete assessment of the algorithmic capabilities, FpVTE will be conducted over several rounds. These test rounds are intended to support development of improved recognition performance. Once the test commences, NIST will accept implementations on a first-come-first-served basis and will return results to participants as expeditiously as possible. Participants may submit revised implementations to NIST only after NIST provides results for the prior submission. The frequency with which a participant may submit implementations to NIST will depend on the times needed for participant preparation, validation, execution, scoring, and participant review and decision processes.

1.7.1 Interim Reports

Each submission will result in a "score-card" provided to the participant. While the score-card may be used by the participant for arbitrary purposes, they are intended to promote development. The score-cards will be:

- machine generated (i.e. scripted),
- provided to participant with identification of their algorithm,
- include template size, timing, and performance accuracy results,
- include anonymous results from other participant implementations,
- be regenerated as new implementations become available and when new analysis is added.

NIST does not intend to release these test reports but may share such information with U.S. Government test sponsors. While these reports are not intended to be made public, NIST can only request that sponsoring agencies not release the content.

1.7.2 Publication of results

Once NIST completes the testing rounds, one or more final reports will be released.

- A NIST Interagency Report (NISTIR) will be published.
- NIST may publish/present results in other academic literature, conferences, or workshops.

The final report will publish results for all submissions, with the intent being to show progress and performance trade-offs made by the implementations. Ultimately, the reported results will highlight the most capable implementations.
IMPORTANT: Results will be attributed to the providers in the final reporting.

1.8 Reporting of Failure to Enroll or Acquire (FTE, FTA)

FTE and FTA have different meanings in production systems where humans interact with biometric readers. For offline testing, soft failures, where the algorithm elects not to produce a template (e.g. image quality reasons) shall be treated identical to failures, where the software crashes or hangs. For this test, any failure to convert images into templates shall be counted as a FTE and reported as such. Further instructions on handling FTE cases are provided in the next section.

1.9 Matching of Empty, Broken, and Missing Templates

For a FTE, the template generator must return an empty (0 byte) template.

After software failure, the absence of an output template will cause NIST to produce an empty (0 byte) template.

For enrollment templates the finalization step must handle these empty templates.

NIST will not call the one-to-many search function if the preceding search template is empty (i.e. FTE). The search template will be stored with a candidate list of all -1 match scores.

1.10 Reporting of Template Size

Because template size is influential to storage requirements, computational efficiency, and wireless/mobile applications, this API supports measurement of template size. NIST will report statistics on the actual size of the templates produced by implementations submitted to this test.

1.11 Runtime Memory Usage

NIST will attempt to allocate enough blades, such that the enrollment data can be distributed across them without enrollment data storage exceeding 75% of the blade's memory. The other 25% will be reserved for OS and implementation usage. The blades being used for the evaluation have 192GB of memory. Based on this distribution there would be 144GB for enrollment data and 48GB for OS and implementation usage. NIST may report the amount of memory used during one-to-many searches. **Please comment on the amount of memory available for the implementation. Too much? Too Little?**

1.12 Reporting of Computational Efficiency

As with other tests, NIST will compute and report performance accuracy. Additionally, NIST will report timing statistics for core functions of the submitted implementation. This includes feature extraction and identification functions.

1.13 Exploring the Accuracy-Speed Trade-Off

Each participant will be allowed to submit two implementations for evaluation. The intent will be that the second is a slower, perhaps by a factor of three. The intent here is do demonstrate the trade off of accuracy versus speed. NIST cannot require that both submissions are actually “fast” and “slow” variants. Participants can always choose to submit two different implementations based on some other criteria (e.g. “mature” and “experimental”). NIST will test them regardless. as long as they meet the minimum PFTII requirement.

1.14 Hardware Specification

NIST intends to support high performance by specifying the runtime hardware beforehand. There are several types of computer blades that may be used in the testing. The blades are labeled as Dell M905, M910, M605, and M610. The following list gives some details about the hardware of each blade type:

- Dell M605 - Dual Intel Xeon E5405 2 GHz CPUs (4 cores each)
- Dell M610 - Dual Intel Xeon X5680 3.3 GHz CPUs (6 cores each)
- Dell M905 - Quad AMD Opteron 8376HE 2 GHz CPUs (4 cores each)
- Dell M910 - Dual Intel Xeon X7560 2.3 GHz CPUs (8 cores each)

NIST is requiring the use of 64-bit implementations. This will support large memory allocation to support 1:N identification with image counts in the multi-millions. Note that while the API allows read access of the disk during execution, the disk access, relatively speaking is slow.

Some of the API calls allow the implementation to write persistent data to hard disk. Note that the file system used doesn’t efficiently support creation of millions of small files. Participants are strongly encouraged to use some database structure to store the finalized enrollment templates.

1.15 Operating system and Compilation Environment

All submitted implementations will be tested on 64-bit CentOS 6.2 running Linux kernel 2.6.32-220. NIST will link the provided libraries to our C++ language test driver. Participants are required to provide their library in a format that is linkable using g++ version 4.4.6. The standard libraries are:

- /usr/lib64/libstdc++.so.6.0.13
- lib64/libc.so.6 -> libc-2.12.so
- lib64/libm.so.6 -> libm-2.12.so

and a typical link command would be:

```
g++ -I. -Wall -m64 -o fpvte12test fpvte12test.cpp -L. -lfpvte12_Participant_A
```

1.16 Threaded Computations

Threaded computations will be allowed for finalizing the enrollment set only. All other functions are not to perform multi-threaded computations for this test. For these other functions, the NIST driver will be running multiple independent instances.

1.17 Time Limits

The reported template extraction timing statistics will be based on a subset on the data run on a dedicated compute blade. These time limits are per image. If all ten fingers of a subject are being extracted, the time limit is increased by a factor of 10.

- Feature extraction - should be accomplished in 3 seconds or less for each input image.
- Finalization - should be accomplished in 2-3 hours on a single compute blade.
- Identification - one search against 1,000,000 enrollment templates (10 print) in ?? seconds.

NIST is looking for feedback on timing expectations. This will help set realistic timing limits and help insure the evaluation can be completed in a timely manner. How long will one search against an enrollment of 1,000,000 take? 5,000,000? 10,000,000? Using 1f, 2f, 4f, 8f, 10f? You can email comments to craig.watson@nist.gov or call Craig at 301-975-4402.

1.18 Ground Truth Integrity

The test datasets are derived from operational systems. They may contain ground truth errors in which:

- a single subject is present under multiple different identifiers,
- two subjects are present under one identifier, or
- finger positions are mislabeled.

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

Chapter 2

Datasets

This section describes the datasets that will be used for the evaluation.

2.1 Image Types

The evaluation dataset used in FpVTE2012 will be from operational datasets made available to NIST for fingerprint evaluations. The datasets are government use only and will not be released to the public. The datasets will, to the extent permitted by law, be protected under the Freedom of Information Act (5 U.S.C. 552) and the Privacy Act (5 U.S.C. 552a) as applicable.

The datasets will be comprised of several fingerprint impression types, including rolled, multi-finger plains, and single finger plains. Rolled are all individual captures that attempt to capture the full width of the fingerprint by rolling from side to side during capture. Multi-finger plains capture the four left/right fingers all at the same time and for Identification Flats (IDFlats) the two thumbs are also captured at the same time. Single finger plains are individual captures of the left and right index fingers on a single finger capture device. If available, the single finger captures may include fingerprints captured on mobile devices. FpVTE2012 will look at performance of plain-to-rolled, plain-to-plain, and if time and resources permit rolled-to-rolled.

Many of the datasets are larger samples of data used in previous NIST evaluations such as PFT, MINEX, NFIQ, and FpVTE2003. The single finger capture fingerprint images will be data from DHS. The ten print rolled and slap (4-4-1-1) fingerprint images will be data from FBI, DHS, LACNTY, AZDPS, and TXDPS. The Identification Flat (4-4-2) fingerprint images will be from DHS.

All images in the datasets are 8-bit grayscale. Images have been compressed using Wavelet Scalar Quantization (WSQ) compression, but they will be passed to the submitted implementation as uncompressed raw pixel images. All images were scanned at 500 pixels per inch and will have varying dimensions that will be passed in a data structure to the implementation.

Multiple finger plain captures will not be segmented into individual fingerprints. The implementation is required to perform any needed segmentation of the fingerprints.

Chapter 3

SDK Application Programming Interface

This section defines the interface for software submitted to the Fingerprint Vendor Technology Evaluation 2012. FpVTE2012 is intended to be a large scale one-to-many identification evaluation of proprietary fingerprint templates.

FpVTE2012 will only use Linux (CentOS 6.2), no other operating systems are allowed. All software must run under Linux

The intent of this API is to support:

- distribution of the enrolled templates across the memory of multiple blades,
- distributed enrollment and identification with multiple instance of each process running independently and in parallel,
- recovery after a fatal exception and record the number of failures,
- the "black-box" nature of biometric templates and matching,
- the freedom of the provider to use arbitrary algorithms,
- measure duration of core function calls,
- measure template sizes,
- measure matching performance.

The following sections give general descriptions of the processes that make up the API and their role in the evaluation. Details of the subroutine interface are given in section 3.5 and Appendix D.

3.1 1:N Identification with Large Enrollment Set

The 1:N identification has two main phases, enrollment and identification. Enrollment has two steps. First all the templates used in the enrolled set are extracted from the fingerprint images. Second, a process is run that finalizes the enrollment templates used during identification. Identification is completed in three steps. First, the templates are extracted for the search image(s). Second, the extracted search templates are passed to stage one of identification, which searches the entire set of enrolled templates, across several blades, for potential matches. Finally, stage two identification is invoked and uses the results from stage one identification to produce the final candidate list of potential matches to the search subject.

The expectation is that the total memory needed to store the entire set of enrolled templates in RAM will exceed the memory capacity of a single blade available for the evaluation. For this reason, stage one identification is accomplished by dividing the enrolled templates into pieces, where each piece can be loaded into RAM on separate blades. Stage one identification searches each piece of the divided enrollment set separately and stores the needed information for that search, so stage two of identification can produce a candidate list.

While the expectation is dividing the enrollment set into pieces across blades, it is conceivable for a given test within the evaluation that the enrolled templates could fit into the memory of a single blade. So implementations should be able to handle any number of blades being used for stage one identification ($1 <= B <= B_{total}$). Additionally, it is expected that if a test were run with $B = 1$ and the same test were run with $B = 2$ (or more) the resulting candidate list would be the same.

Each function in the API enrollment process, search template extraction, stage one identification, and stage two identification call initialization routines that allow the implementation to pre-configure itself as needed. No additional initialization functions are allowed and all configuration information will be located in the configuration directory which has read only access.

3.1.1 Threading and Multi-Processing

Enrollment finalization is the only function allowed to use multi-threading. It is a single process that must run to completion before proceeding to the identification stages.

For all other functions (template extraction and identification) the NIST driver will handle invoking multiple processes to fully utilize the available resources. Implementations submitted should anticipate that each stage, enrollment/search template extraction and stage one/two identification will have multiple instances invoked by the NIST driver. These instances could be across multiple CPUs/cores on a single blade or across multiple blades. It is important that the implementation account for this during process initialization so that each instance runs completely independent of the others.

3.1.2 Implementation Identifiers

Each implementation must return a NIST assigned ID and a contact email address.

3.2 Input Fingerprint Image Structure

Structure details are in Appendix D (D.2).

FpVTE2012 will enroll multiple fingerprints from a subject in a single function call. The input data structure `FingerImage` will be used for passing images for a single subject into the template extraction process. The details related to this structure are described in the reference documentation `FingerImage` and `FingerImageSet`. `FingerImage` defines the structure for a single fingerprint image and `FingerImageSet` is a vector of `FingerImage` for each finger of that subject. If the test is using 10 rolled fingerprints for each subject the `FingerImageSet` for each subject will contain `FingerImages`. If the test is 10 plain fingerprint images per subject the input FingerImageSet will have FingerImage(s) for the left/right four finger plains and left/right thumb plains. Any segmentation into individual fingerprints is left to the implementation.

The number of images per subject will vary depending on the specific test being performed, but will typically be either 1, 2, 4, or 10 fingerprint images. The implementation will need to support this accordingly in enrollment and matching.

3.3 Output Fingerprint Template Structure

Structure details are in Appendix D (D.3).

The template extraction process will return both the resulting fingerprint templates and a data structure `FingerOutput` containing information about the templates (i.e. actual template size) and the input fingerprints for a given subject. The format of the returned templates are uncontrolled by FpVTE2012. Details related to the structure can be found in the reference documentation `Finger Output` and `FingerOutput`.

3.3.1 Enrollment Template Extraction and Finalization

The enrollment template extraction process is accomplished in two steps. First, templates for each subject in the enrolled set are extracted independently of each other. This allows for multiple processes to run simultaneously across many cores/blades. After all

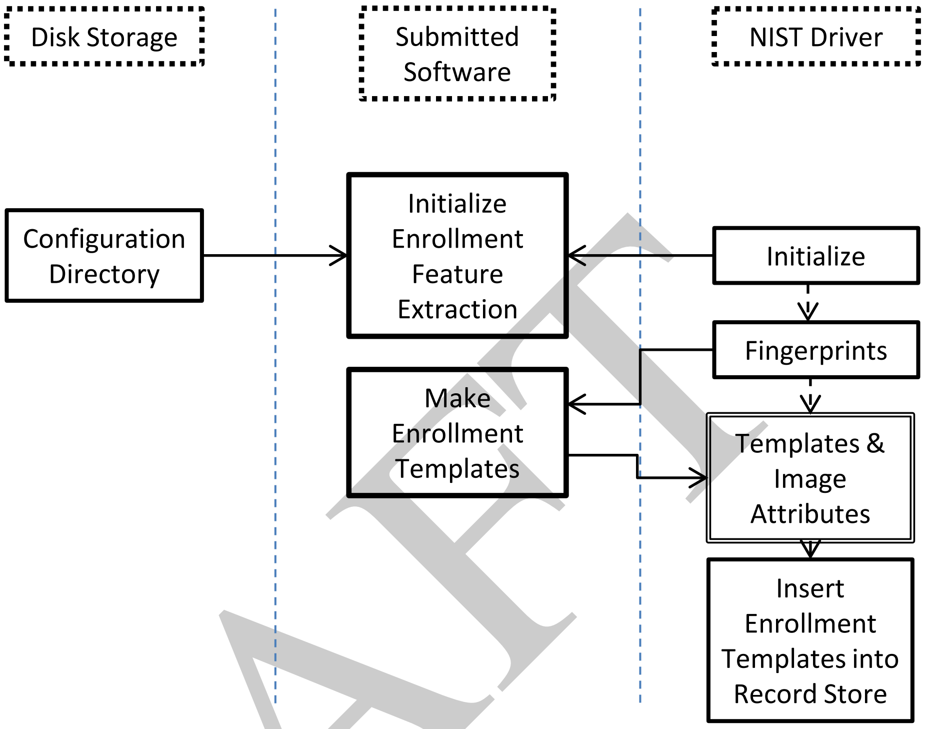
Disk Storage
Submitted Software
NIST Driver
Configuration Directory
Initialize Enrollment Feature Extraction
Initialize
Fingerprints
Make Enrollment Templates
Templates & Image Attributes
Insert Enrollment Templates into Record Store

Figure 3.1: Flowchart for enrollment feature template extraction process

templates are extracted, the second step is the enrollment finalization described in more detail in section 3.3.1.2.

3.3.1.1 Enrollment Template Extraction

Function call parameters and details are in Appendix D (D.5.1.2 and D.5.1.3).

The enrollment template extraction process is shown in figures 3.1 and 3.2. For each instance of the enrollment template extraction process, the NIST driver will call the enrollment initialization function once before multiple calls to enrollment template extraction. The resulting templates are then stored in a template RecordStore. The RecordStore is described in section 3.5.3.1. The enrollment template extraction process will be accomplished by running multiple template extraction processes in parallel across multiple cores and blades. NIST will merge the multiple template RecordStores created into a single enrollment template RecordStore that gets passed to the enrollment finalization process.

3.3.1.2 Enrollment Finalization

Function call parameters and details are in Appendix D (D.5.1.4).

After extracting all enrollment templates, the enrollment finalization process shown in figures 3.3 and 3.4 is called. The input to this process is the merged single enrollment RecordStore and the output is a directory that the application will populate as needed. The implementation must divided the finalized enrollment set into B partitions, one for each of the blades in the search process. The NIST test driver will tell the finalization process how many blades (B) can be used for identification. After finalization, the enrollment set is "frozen" and this directory will be made read only during template searching.

Finalization allows the implementation to conduct, for example, statistical processing of the feature data, indexing, and data reorganization. The function may alter the file structure. It may increase or decrease the size of the stored data. No output to the test driver is expected from this function, except a successful `StatusCode`.

The format of the enrollment directory is completely controlled by the participant. A major note here is that the file system used in the

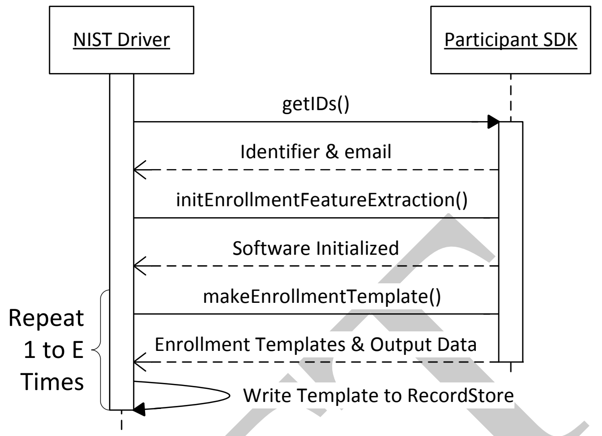

Figure 3.2: Calling sequence for enrollment feature template extraction process

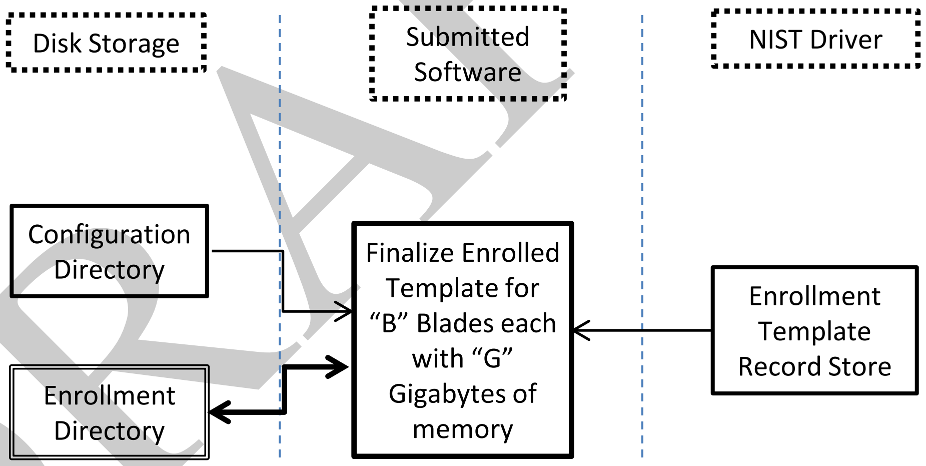
Disk Storage
Submitted Software
NIST Driver
Configuration Directory
Enrollment Directory
Finalize Enrolled Template for "B" Blades each with "G" Gigabytes of memory
Enrollment Template Record Store

Figure 3.3: Flowchart of enrollment template finalization process

NIST test environment is not optimal for large numbers of small files. If an implementation is creating a large number of individual files, it will impact performance and most likely be excluded from the evaluation. Implementations must not create individual files for each subject in the enrollment set.

3.3.2 Search Template Extraction

Function call parameters and details are in Appendix D (D.5.1.5 and D.5.1.6).

This section of the API describes the search template extraction process as shown in figures 3.5 and 3.6. The enrollment templates will not be used as search templates. This allows for a different function to be used for search template creation. Like enrollment template creation, the NIST driver will call initialization once for each instance of search template creation before multiple calls to search template creation. The resulting templates will be stored for later testing or passed directly to stage one identification.

3.3.3 Identification Stage One

Function call parameters and details are in Appendix D (D.5.1.7 and D.5.1.8).

This section describes the first stage of identification as shown in figures 3.7 and 3.8. For this stage of identification, the enrollment

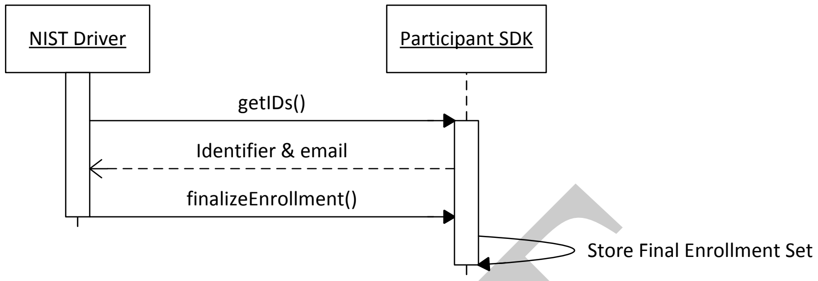

Figure 3.4: Calling sequence for enrollment template finalization process

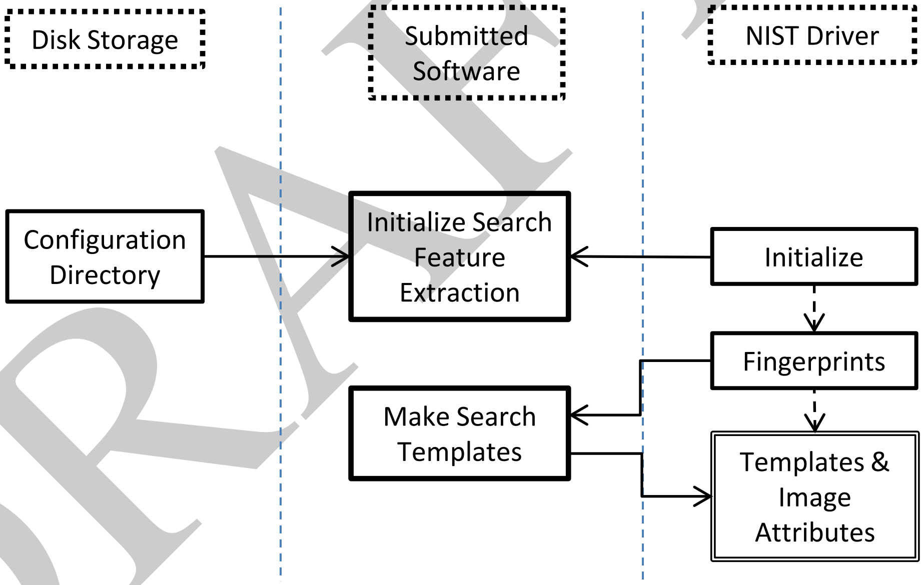

Figure 3.5: Flowchart of search feature template extraction process

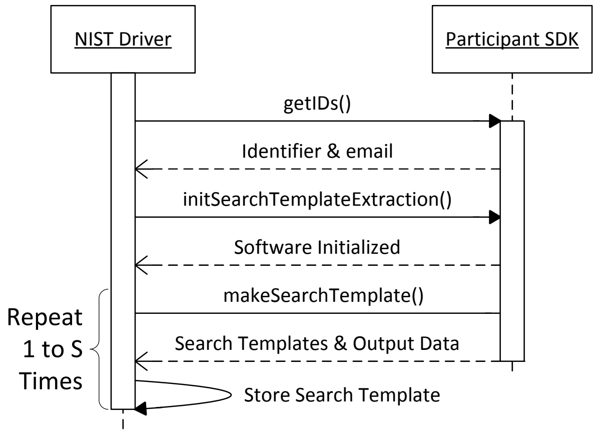

Figure 3.6: Calling sequence for search feature template extraction process

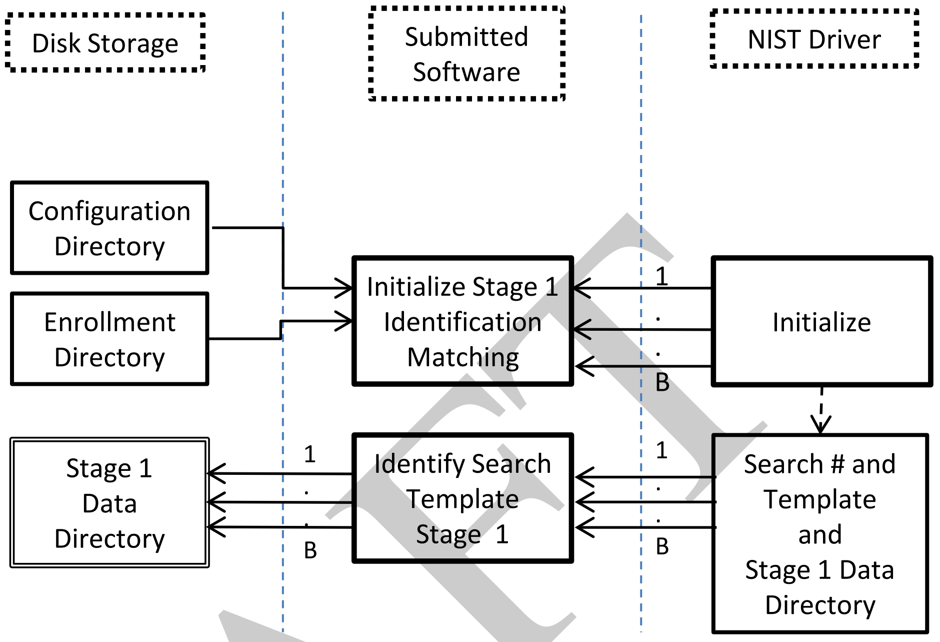
Disk Storage
Submitted Software
NIST Driver
Configuration Directory
Enrollment Directory
Initialize Stage 1 Identification Matching
Initialize
1
B
Stage 1 Data Directory
Identify Search Template Stage 1
Search # and Template and Stage 1 Data Directory

Figure 3.7: Flowchart of first stage of identification process

set will be stored in pieces across multiple blades. The number of blades (B) will be the same as was input for enrollment finalization. Each separate piece of the enrollment set is loaded into memory by the implementation during the initialization step. The NIST driver will run multiple search subjects against this piece of the enrollment set simultaneously on that blade.

There will be a 4 GB storage area in which results from this first stage of matching can be stored for each search subject. Note that the storage area is shared by all identification processes running on that blade. In an effort to remove disk I/O from match timing functions, the moving of stage one identification results data from the dedicated storage area to a shared SAN storage device is done by the NIST driver after the call to stage one identification is completed for each search subject.

Once results are moved to the SAN storage, they are removed from the dedicated storage freeing up space for other stage one results. It is important to note that the output directory for stage one identification results is shared by all B blades running stage one identification processes. This directory is where the output results from all stage one identification processes are being grouped back together for processing in stage two identification. The multiple identification processes in stage one across the blades are asynchronous, but stage two identification can not be run until all blades have finished stage one for a specific search subject.

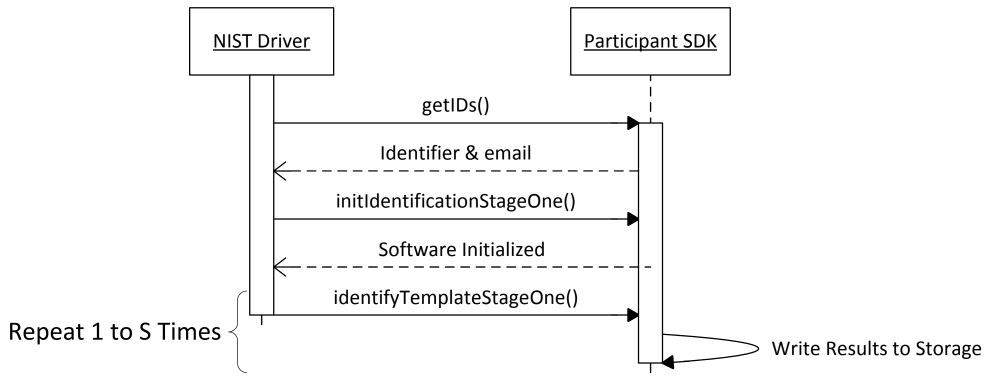

Figure 3.8: Calling sequence for first stage of identification process

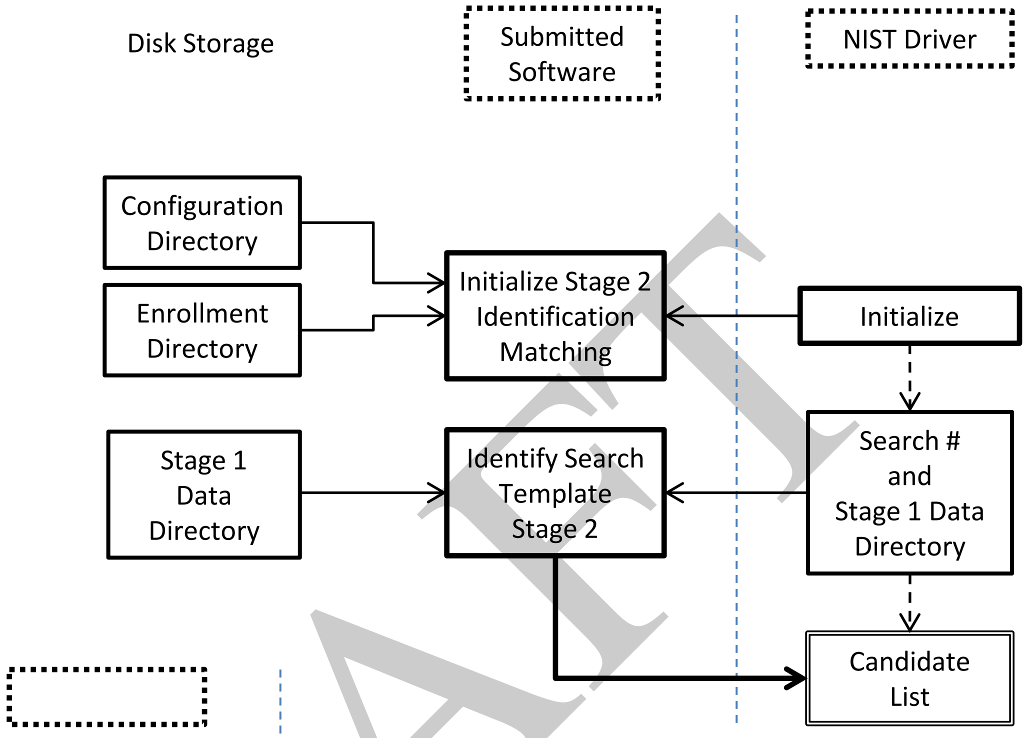

Figure 3.9: Flowchart of second stage of identification process

3.4 Output Candidate Lists Structure

Structure details are in Appendix D (D.1).

The second identification stage will produce a candidate list for results analysis. The list will be returned in a data structure defined in the reference documentation `Candidate` and `CandidateSet`.

3.4.1 Identification Stage Two

Function call parameters and details are in Appendix D (D.5.1.9 and D.5.1.10).

This section describes the second stage of identification as shown in figures 3.9 and 3.10. After initializing this stage of identification, multiple search subjects will be passed to the identification process, one at a time. The search subjects will be the same ones from stage one identification. The second stage of identification will have access to both the original enrollment directory and the data directory containing the output of stage one identification. Stage two identification will output the candidate list and required scoring data for the search subject.

There is no intended control on how stage two reaches the final candidate list but there will be time constraints. Stage two could be as simple as sorting/organizing stage one results to get the candidate list or it could involve a more complex algorithm that involves some level of additional matching of feature templates.

3.5 SDK C++ Interface

FpVTE has defined an abstract C++ [1] class that must be implemented by the software under test. This class, `SDKInterface`, contains the methods that will be called by the NIST SDK driver software. SDK vendors must deliver a library that implements the abstract interface defined by that class. C++ was chosen in order to make use of some object-oriented features and for interoperability

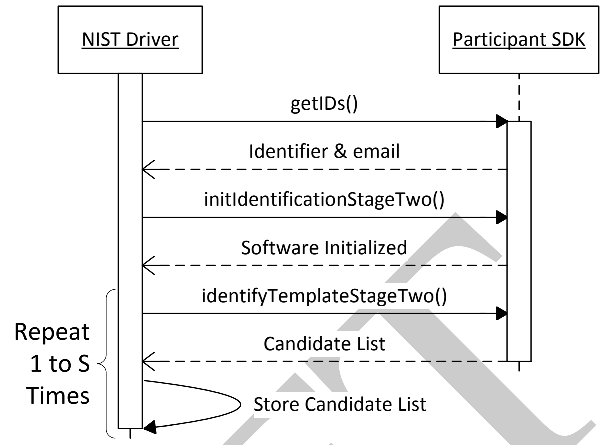

Figure 3.10: Calling sequence for second stage of identification process

with existing C code.

3.5.1 The SDK Interface

The abstract class `SDKInterface` must be implemented by the SDK vendor. The processing that takes place during each phase of the test is done via calls to the methods declared in the interface as pure virtual, and therefore are to be implemented by the SDK. The test driver will call these methods, handling all return values and exceptions. Example code will be distributed by NIST to show how an SDK can implement the methods.

An example of an implementation's header file might be:

Listing 3.1: SDKInterface Subclass Declaration

```
#include <fpvte2012.h>

namespace FPVTE2012 {
class NullSDK : public FPVTE2012::SDKInterface {
public:

        NullSDK();
        ~NullSDK();

        void getIDs(
            string &nist_assigned_identifier,
            string &email_address);
}
```

Listing 3.1 shows how to subclass the abstract interface class and declare the implementation of the `getIDs()` method. The definition of the method is then:

Listing 3.2: SDKInterface Subclass Definition

```
#include <fpvteNullSDK.h>

using namespace BiometricEvaluation;
using namespace FPVTE2012;

NullSDK::NullSDK () { }

```

```
NullSDK::~NullSDK() { }

void
NullSDK::getIDs(
    string &nist_assigned_identifier,
    string &email_address)
{
        nist_assigned_identifier = "0x12345678";
        email_address = "foo@example.com";
}

SDKIptr
SDKInterface::getSDK()
{
        NullSDK *p = new NullSDK();
        SDKIptr ap(p);
        return (ap);
}
```

The other methods of the SDKInterface class are implemented in a similar manner.

There is one class (static) method declared in `SDKInterface`, `getSDK()` which must also be implemented by the SDK. This method returns a shared pointer to the object of the interface type, an instantiation of the SDK implementation class. A typical implementation of this method is also shown in Listing 3.2.

3.5.2 Exceptions

Many of the SDK API methods are declared to throw an exception in the case of error. SDKs should only throw an exception when the method call cannot be completed. For example, in the matching call, the SDK should *not* throw an exception if there is a failure to match, or the image quality is too low. An example of the appropriate use of the exception is when a memory allocation failure, or an input template is corrupted. The exception string should then be filled out with as much detail as desired to enable debugging.

3.5.3 Biometric Evaluation Classes

The API makes use of several classes that are part of the NIST Image Group Evaluation framework. The pertinent components of the framework will be provided to SDK vendors as source code, including a build system for the Linux operating system. Descriptions of the framework classes are provided in this section.

3.5.3.1 IO::RecordStore

A `RecordStore` is an abstraction that associates keys with an opaque object. These key-value pairs are unique, based on the key which is a string. Applications, in addition to retrieving the data associated with a key, can insert, remove, or replace the data. Functionality is also provided to sequence through a RecordStore; however, there is no guarantee that the key-value pairs will be retrieved in the order they were inserted.

The API defined for FpVTE provides an opened RecordStore to the SDK as read-only in some cases. The SDK must not attempt to insert or remove a key for those RecordStores else an exception will be thrown.

For FpVTE2012 the RecordStore is being used to store the large set of enrollment templates in a single file that can then be passed to the finalization process.

3.5.3.2 Memory::AutoArray

The `AutoArray` is used to manage an array of objects in a safe manner with the efficiency associated with standard C++ arrays. The size of the array can be changed dynamically, usually done when a larger array is required.

Many of the SDK interface functions pass a buffer containing an image or a biometric template. This buffer is defined as a `uint8Array`, which is an `AutoArray` of unsigned 8-bit integers.

3.5.3.3 Error::Exception

All exceptions thrown by the SDK are of the `Exception` class. An object of this class can be constructed with a string giving more details on the nature of the exception. The functions implemented by the SDK should set the exception string to contain information that will be useful for debugging.

Some of the methods of the framework objects throw exceptions which must be handled by the SDK. All of these exception objects are subclasses of the `Exception` class. Therefore, the SDK can simply catch Exception, and then may simply rethrow it, or throw a new Exception object with more detailed information in the information string of the Exception object. In all cases the SDK methods must throw an `Exception` object.

Chapter 4

Software and Documentation

This section describes the specific software and documentation requirements for implementations submitted by participants.

4.1 Library and Platform Requirements

Participants shall provide NIST with binary code only (no source code). Header files ("h" shouldn't be necessary but are allowed. If used, they shall not contain intellectual property of the participant nor material that is otherwise proprietary. It is preferred the implementation be submitted in the form of a single static library. However, dynamic and shared library files are permitted.

The core library shall be named as follows:

libFpVTE12_*NIST_Assigned_ID*_[1 *or* 2]_*sequence*.[so *or* a]

NIST_Assigned_ID The implementation identifier assigned by the NIST test liaison.

1 *or* **2** "Fast" (1) or "slow" (2) submission for this implementation.

sequence Two digit sequence number, incremented by 1 when a new implementation is sent to NIST (i.e. 01, 02, . . .).

If necessary, additional dynamic or shared library files may be submitted that support this "core" library file. The core library file may have dependencies implemented in these other files.

Access to any GPUs is not permitted.

4.2 Configuration and Vendor-Defined Data

The implementation under test may be supplied with configuration and supporting data file in the configuration directory.

NIST will report the size of the supplied configuration files.

4.3 Linking

NIST will link the provided library file(s) to a C++ language test driver application developed at NIST. The runtime environment will be CentOS 5.5 based on later Linux 2.6 kernels, with the Gnu C++ compiler, g++ 4.1.2-14 (**PREFERRED**).

The NIST test driver is compiled using g++ and participants are required to provide their library in a format that is linkable to the NIST test driver using g++. All compilations and testing will be performed on X86 platforms. Thus, participants are strongly advised to verify library-level compatibility with g++ (on an equivalent platform) prior to submitting their software to NIST. This will help avoid linkage problems once the software is sent to NIST.

FpVTE2012 will only use Linux (CentOS 6.2), no other operating system are allowed. All software must run under Linux.

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

4.4 Installation and Usage

The implementation must install easily (no participant interaction required) to be tested, and shall be executable on any number of machines without additional machine-specific license control procedures or activation. The implementation shall be installable using simple file copy methods and shall not require the use of a separate installation program.

The implementation shall neither implement nor enforce any usage controls or limits based on licenses, number of executions, presence of temporary files, etc. The implementation shall remain operable until December 31, 2013.

Hardware activation dongles are not acceptable.

4.5 Modes of Operation

Individual implementations provided shall not include multiple "modes" of operation, or algorithm variations. No switches or options are allowed within a library. For example, the use of two different "coders" by a feature extractor must be split across two separate implementation library submissions.

4.6 Runtime Behavior

4.6.1 Interactive Behavior

The implementation will be run in a non-interactive batch mode, without terminal support. Therefore, the submitted library shall not use any interactive functions such as graphical user interface (GUI) calls or any other calls which require any type of terminal interaction.

4.6.2 Status Messages

Since the implementation will be run in a non-interactive batch mode, the submitted library must run quietly meaning no messages should be written to standard output or error. An implementation may write debugging messages to a log file, which must be declared in the documentation.

4.6.3 Error Codes and Exception Handling

The implementation should report errors back the the caller by setting the appropriate status code, and optionally providing a information string. Section 3.5 describes the handling of exceptions encountered by the SDK. In the case of an exception being thrown by the SDK, the returned status object will not be available to the caller. Therefore, use of the information string inside the exception object is encouraged.

4.6.4 External Communication

Processes running on NIST hosts shall not create side effects in the runtime environment in any manner, except for memory allocation and release. Implementations shall not write any data to an external resource (e.g. server, network, or other process).

Implementations shall not attempt to read any resource other than those explicitly allowed in this document. If detected, NIST reserves the right to cease evaluation of all implementations from the participant, notify the participant, and document the activity in the published reports.

4.6.5 Stateful Behavior

All components in this test shall be stateless and deterministic, except as noted. This applies to image segmentation, template creation and matching. Thus, all functions should give identical output, for a given input, independent of the runtime history. This holds for enrollment set partitioning. For example, results should be the same if the enrollment set is spread across four blades or eight blades. NIST will institute appropriate tests to detect stateful behavior. If detected, NIST reserves the right to cease evaluation of all implementations from the participant, notify the participant, and document the activity in the final reports.

Chapter 5

How to Submit Implementations to FpVTE2012

5.1 Confidentiality and Integrity Protection

NIST requires that all software, data, and configuration files submitted by the participants be signed and encrypted. Signing is done with the participants private key, and encryption is done with the NIST public key. The detailed commands for signing and encrypting are given here:

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

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

5.2 How to Participate

Those wishing to participate in FpVTE2012 must meet all the deadlines of the FpVTE 2012 calendar.

Software must be submitted as follows:

- IMPORTANT: Follow the instructions for encrypting submissions in section 6.
- Complete the "Application to Participate in FpVTE2012".
- Provide an implementation library (encrypted) that complies with the API specified in this document.
  - Encrypted data and implementations below 20MB can be emailed to NIST at fpvte@nist.gov
  - Encrypted data and implementations over 20MB can be
    - Made available for download from generic websites where NIST is not required to register or establish any membership, or
    - FEDEX/UPS or similar shipping method (so package can be tracked) to NIST on CD/DVD at this address (tracking number must be emailed to NIST):

      NIST
      c/o FpVTE2012 Test Liaison
      100 Bureau Drive MS 8940
      Gaithersburg, MD 20899-8940
      USA

5.3 Implementation Validation

Participants will be provided validation datasets containing imagery from public available datasets in the same format as the evaluation test datasets. These validation datasets will be used to test implementation functionality before submitting to NIST. The output from these validation datasets will be submit with the implementation. NIST will repeat the processing of the validation datasets on our hardware to confirm the implementation is functioning correctly.

The validation datasets will most likely include Special Database 14, Special Database 29, Images from Special Database 24, and possibly another source of live-scan slap capture data.

Chapter 6

Encrypting Software for Transmission to NIST

6.1 Scope

NIST requires that all software submitted by the participants be signed and encrypted. Signing is done with the participant's private key, and encrypting is done with the NIST public key, which is published on the project's Web site. NIST will validate all submitted materials using the participant's public key, and the authenticity of that key will be verified using the key fingerprint. This fingerprint must be submitted to NIST as part of the signed agreement.

By encrypting the submissions, we ensure privacy; by signing the submission, we ensure authenticity (the software actually belongs to the submitter). NIST will not take ownership of any submissions that are not signed and encrypted.

All cryptographic operations (signing and encrypting) shall be performed with software that implements the OpenPGP standard, as described in Internet RFC 4880. The freely available Gnu Privacy Guard (GPG) software, available at www.gnupg.org, is one such implementation.

6.2 Submission of Software to NIST

NIST requires that all software submitted by the participants be signed and encrypted. Two keys pairs are needed:

- Signing is done with the software provider's private key, and
- Encryption is done with the NIST public key, which is published on the evaluation website.

6.2.1 Creating a cryptographic key pair

The steps below show how to create a public/private key pair and fingerprint using the GPG software.

Generate the key pair:

Command	Description
`gpg --gen-key`	Press Enter for the default key type, DSA and Elgamal Choose a key size of 2048 Choose a non-expiring key Press "y" Enter Real Name Enter an email address; this is the key identity Enter an optional comment Press "O" to continue Enter a passphrase for the secret (private) key

Once the key pair is generated, the key must be exported in the proper format to be sent to NIST:

Export the public key:

Command	Description
`gpg --armor --output forfpvte.gpg --export` *email*	Where *email* is the address used in the key generation step above. This address is the key identity. The public key will be saved into the file named `forfpvte.gpg`. The file containing the public key must be sent to the NIST Test Liaison.

Generate the key fingerprint:

Command	Description
`gpg --fingerprint` *email*	The key fingerprint will be shown in the output as a set of hex digits. The fingerprint must be copied onto the participant agreement sent to NIST.

6.2.2 Obtaining the NIST key pair

The next series of step show how the participant will import the NIST public key, and authenticate that key using the key fingerprint. The NIST public key will be sent to each participant after receiving the signed agreement. The following example assumes the NIST key is saved into a file named `nistfpvte.gpg`.

Import the NIST public key:

Command	Description
`gpg --import nistfpvte.gpg`	The output should be similar to: `key 856B9B28: public key "NIST Test Liaison (NIST Test Liaison Key) <fpvte@nist.gov>" imported`

Authenticate the NIST key:

Command	Description
`gpg --fingerprint fpvte@nist.gov`	The key fingerprint will be shown in the output as a set of hex digits. These digits must be the same as the NIST public key fingerprint which is printed on the participant agreement.

Optionally, the participant may want to assign a level of trust to the NIST public key:

Command	Description
`gpg --edit-key fpvte@nist.gov`	Enter "trust" at the Command prompt Choose a trust level; 3 is a good choice Enter "y" to approve the trust selection, if asked Enter "q" to quit

6.2.3 Encryption and signing

By following the instructions in Sections 6.2.1 and 6.2.2, the keys have been generated and exchanged between NIST and the participant. From this point forward, all software submissions must be signed and encrypted. In addition, general email communication can be encrypted and signed, if desired. This section shows how to encrypt and sign a file to be sent to NIST.

Encrypt and sign a file:

Command	Description
`gpg --default-key` *email* `--output` *filename*`.gpg --encrypt --recipient fpvte@nist.gov --sign` *filename*	*email* is the key identity chosen when the key pair was created *filename* is the file to be submitted to NIST Enter the passphrase chosen for the private key

Bibliography

[1] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, special edition, 2000. 14

Appendix A

Namespace Index

A.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Appendix B

Class Index

B.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Appendix C

Namespace Documentation

C.1 FPVTE2012 Namespace Reference

Contains all the structures and functions used by the API.

Classes

- struct FingerImage

  Object containing the fingerprint image and corresponding attributes.
- struct FingerOutput

  Object for output data related to an extracted template.
- struct Candidate

  Object defining format of candidate reporting.
- class StatusCode
- struct ReturnStatus

  A structure to contain information about a failure by the software under test.
- class SDKInterface

Typedefs

- typedef struct FingerImage **FingerImage**
- typedef Memory::AutoArray < FingerImage > FingerImageSet
- typedef struct FingerOutput **FingerOutput**
- typedef Memory::AutoArray < FingerOutput > FingerOutputSet
- typedef struct Candidate **Candidate**
- typedef Memory::AutoArray < Candidate > CandidateSet
- typedef struct ReturnStatus **ReturnStatus**
- typedef tr1::shared_ptr < SDKInterface > **SDKIptr**

Functions

- std::ostream & operator<< (std::ostream &s, const FPVTE2012::StatusCode::Kind &sc)
- std::ostream & operator<< (std::ostream &s, const FPVTE2012::ReturnStatus &rs)

C.1.1 Detailed Description

Contains all the structures and functions used by the API.

C.1.2 Typedef Documentation

C.1.2.1 typedef Memory::AutoArray<FingerImage> FPVTE2012::FingerImageSet

Object representing a set of Finger records. Using this object anywhere from one to ten fingerprints will be input for a single test subject.

C.1.2.2 typedef Memory::AutoArray<FingerOutput> FPVTE2012::FingerOutputSet

Object representing a set of Finger output records. The number of records output should be equal to the number of records input to the process.

C.1.2.3 typedef Memory::AutoArray<Candidate> FPVTE2012::CandidateSet

Object representing a list of Candidates. This object allows the search function to return a candidate list of 1:L as defined by the evaluation protocol.

C.1.3 Function Documentation

C.1.3.1 std::ostream& FPVTE2012::operator<< (std::ostream & *s*, const FPVTE2012::StatusCode::Kind & *sc*)

Output stream operator for a StatusCode object.

C.1.3.2 std::ostream& FPVTE2012::operator<< (std::ostream & *s*, const FPVTE2012::ReturnStatus & *rs*)

Output stream operator for a ReturnStatus object.

Appendix D

Class Documentation

D.1 FPVTE2012::Candidate Struct Reference

Object defining format of candidate reporting.

```
#include <fpvte2012.h>
```

Public Attributes

- uint32_t **templateID**
- double **similarity**

D.1.1 Detailed Description

Object defining format of candidate reporting.

Each candidate returned for a search subject will report the information defined in this structure. Candidates should be sorted into ascending order of match probability.

Parameters

templateID	The Template ID of the candidate. The ID returned for the candidate must be the same ID that was passed into enrollment finalization.
similarity	The similarity score that the candidate and search templates are from the same subject.

The documentation for this struct was generated from the following file:

- fpvte2012.h

D.2 FPVTE2012::FingerImage Struct Reference

Object containing the fingerprint image and corresponding attributes.

```
#include <fpvte2012.h>
```

Public Attributes

- Finger::Position::Kind **fgp**

- Finger::Impression::Kind **imp**
- Image::Raw **raw_image**
- uint8_t **nfiq**

D.2.1 Detailed Description

Object containing the fingerprint image and corresponding attributes.

The fingerprint image object used to pass the image and attributes to the template extraction process.

Parameters

fgp	Fingerprint position.
imp	Impression type for the fingerprint.
raw_image	Input image data.
nfiq	Image nfiq value with range from 1 (best) ... 5 (worst).

The documentation for this struct was generated from the following file:

- fpvte2012.h

D.3 FPVTE2012::FingerOutput Struct Reference

Object for output data related to an extracted template.

```
#include <fpvte2012.h>
```

Public Attributes

- uint16_t **template_size**
- uint8_t **image_quality**
- bool **core_location_present**
- Image::Coordinate **core_location**
- bool **delta_location_present**
- Image::Coordinate **delta_location**

D.3.1 Detailed Description

Object for output data related to an extracted template.

Information need by the NIST driver to store the extracted template into a RecordStore and potentially for use in results analysis.

Parameters

template_size	The actual size of the extracted template.
image_quality	Image quality computed by the template extraction application. Valid range is 0-100 with 0 being lowest quality.
core_location_present	Whether or not core_location has been set.
core_location	If available, the core location of the fingerprint. This is an x,y pixel coordinate location in the image.
delta_location_present	Whether or not delta_location has been set.
delta_location	If available, the delta location of the fingerprint. This is an x,y pixel coordinate location in the image.

The documentation for this struct was generated from the following file:

- fpvte2012.h

D.4 FPVTE2012::ReturnStatus Struct Reference

A structure to contain information about a failure by the software under test.

```
#include <fpvte2012.h>
```

Public Member Functions

- ReturnStatus (const StatusCode::Kind code, const string info="")

  Create a ReturnStatus object.

Public Attributes

- StatusCode::Kind **code**
- string **info**

D.4.1 Detailed Description

A structure to contain information about a failure by the software under test.

An object of this class allows the software to return some information from a function call. The string within this object can be optionally set to provide more information for debugging etc. The status code will be set by the function to Success on success, or one of the other codes on failure. In those cases, processing will proceed with further calls to the function.

Note

If the SDK encounters a non-recoverable error, an exception should be thrown and processing will stop.

D.4.2 Constructor & Destructor Documentation

D.4.2.1 FPVTE2012::ReturnStatus::ReturnStatus (const StatusCode::Kind *code*, const string *info* = "")

Create a ReturnStatus object.

Parameters

in	*code*	The return status code; required.
in	*info*	The optional information string.

The documentation for this struct was generated from the following file:

- fpvte2012.h

D.5 FPVTE2012::SDKInterface Class Reference

Public Member Functions

- virtual void getIDs (string &nist_assigned_identifier, string &email_address)=0

  Return the identifier and contact email address for the software under test.
- virtual ReturnStatus initEnrollmentTemplateExtraction (const string &configuration_location, const uint32_t num_subjects)=0 throw (Error::Exception)

This function initializes the makeEnrollmentTemplate process and sets all needed parameters.

- virtual ReturnStatus makeEnrollmentTemplate (const FingerImageSet &fingers, Memory::uint8Array &enrollment_template, FingerOutputSet &output_features)=0 throw (Error::Exception)

  This function takes a FingerImageSet and outputs a enrollment template representing the one or more fingers that were in the input FingerImageSet.
- virtual ReturnStatus finalizeEnrollment (const string &configuration_directory, const string &enrollment_directory, const uint8-_t blade_count, const uint16_t blade_memory, IO::RecordStore &input_templates)=0 throw (Error::Exception)

  This function does final processing of the complete set of enrollment templates that will then be used in matching search templates.
- virtual ReturnStatus initSearchTemplateExtraction (const string &configuration_directory)=0 throw (Error::Exception)

  This function initializes the makeSearchTemplate process and sets all needed parameters.
- virtual ReturnStatus makeSearchTemplate (const FingerImageSet &fingers, Memory::uint8Array &search_template, FingerOutputSet &output_features)=0 throw (Error::Exception)

  This function takes a FingerImageSet and outputs a search template representing one or more fingers. The search template can be produced with a different algorithm than makeEnrollmentTemplate.
- virtual ReturnStatus initIdentificationStageOne (const string &configuration_directory, const string &enrollment_directory, const uint8_t blade_number)=0 throw (Error::Exception)

  This function initializes identifyTemplateStageOne.
- virtual ReturnStatus identifyTemplateStageOne (const uint32_t search_ID, const Memory::uint8Array &search_template, const string &stage1_data_directory)=0 throw (Error::Exception)

  This function searches a template against the partial enrollment set selected by the blade_number in initIdentificationstageOne.
- virtual ReturnStatus initIdentificationStageTwo (const string &configuration_directory, const string &enrollment_directory)=0 throw (Error::Exception)

  This function initializes identifyTemplateStageTwo.
- virtual ReturnStatus identifyTemplateStageTwo (const uint32_t search_ID, const string &stage1_data_directory, CandidateSet &candidates)=0 throw (Error::Exception)

  This function takes the results from identifyTemplateStageOne and produces a candidate list for the search subject.

Static Public Member Functions

- static SDKIptr getSDK ()

  Factory function to return a managed pointer to the SDK object.

D.5.1 Member Function Documentation

D.5.1.1 virtual void FPVTE2012::SDKInterface::getIDs (string & *nist_assigned_identifier*, string & *email_address*) `[pure virtual]`

Return the identifier and contact email address for the software under test.

This function retrieves an identifier that the provider must request from NIST and hard wired into the source code. NIST will assign the identifier that will uniquely identify the supplier and the SDK version number.

Parameters

out	*nist_assigned_identifier*	An ID which identifies the SDK under test. Cannot be the empty string.
out	*email_address*	Point of contact email address. Cannot be the empty string.

D.5.1.2 virtual ReturnStatus FPVTE2012::SDKInterface::initEnrollmentTemplateExtraction (const string & *configuration_location,* const uint32_t *num_subjects*) throw (Error::Exception) `[pure virtual]`

This function initializes the makeEnrollmentTemplate process and sets all needed parameters.

The function is called once by the NIST application before n >= 1 calls to makeEnrollmentTemplate. The implementation must tolerate execution of multiple instances of the makeEnrollmentTemplate process (each initialized separately) running simultaneously and independently on the same machine and/or across multiple machines.

Parameters

`in`	*configuration_location*	A read-only directory containing vendor-supplied configuration parameters or run-time data files.
`in`	*num_subjects*	The total number of subjects that will be used in the enrollment set.

Returns
The object containing a required status code and optional information string.

Exceptions

Error::Exception	There was an error processing this request, and the exception string may contain additional information.

D.5.1.3 virtual ReturnStatus FPVTE2012::SDKInterface::makeEnrollmentTemplate (const FingerImageSet & *fingers,* Memory::uint8Array & *enrollment_template,* FingerOutputSet & *output_features*) throw (Error::Exception) `[pure virtual]`

This function takes a FingerImageSet and outputs a enrollment template representing the one or more fingers that were in the input FingerImageSet.

The memory for the template is managed in the uint8Array (an AutoArray) object. This routine should not perform memory allocation but can resize the AutoArray if more space is needed.

Parameters

`in`	*fingers*	A set of finger images. The implementation will have to adjust to the number of available fingerprints based on the dataset being used. For example 10 rolled or 2 index plain impressions.
`out`	*enrollment_template*	The template to be added to the enrollment set, extracted from the finger images. The format is not regulated.
`out`	*output_features*	Additional information for the images, produced by the SDK. This information will be used in final results analysis.

Returns
The object containing a required status code and optional information string.

Exceptions

Error::Exception	There was an error processing this request, and the exception string may contain additional information.

D.5.1.4 virtual ReturnStatus FPVTE2012::SDKInterface::finalizeEnrollment (const string & *configuration_directory,* const string & *enrollment_directory,* const uint8_t *blade_count,* const uint16_t *blade_memory,* IO::RecordStore & *input_templates*) throw (Error::Exception) `[pure virtual]`

This function does final processing of the complete set of enrollment templates that will then be used in matching search templates.

The finalization step must prepare the enrolled templates to be distributed across a number of blades. The enrollment directory will then have read-only access for stages 1 and 2 of the identification process.

Parameters

in	*configuration_location*	A read-only directory containing vendor-supplied configuration
in	*enrollment_directory*	The top-level directory in which all enrollment data will reside. After this finalization step the directory and its contents with become read-only. The directory is initially empty. Access permission will be read-write and the application can populate this folder as needed. Additionally, the file system does not perform well with the creation of millions of small files, so the application should consolidate templates into some sort of database file.
in	*blade_count*	The number of blades the enrollment set will be spread across. It is up to the implementation to determine how best to spread the enrolled templates across the blades in order to get best performance.
in	*blade_memory*	Amount of memory available on each blade in gigabytes.
in	*input_templates*	The input RecordStore containing the enrolled templates, opened IO::READONLY.

Returns
The object containing a required status code and optional information string.

Exceptions

Error::Exception	There was an error processing this request, and the exception string may contain additional information.

D.5.1.5 virtual ReturnStatus FPVTE2012::SDKInterface::initSearchTemplateExtraction (const string & *configuration_directory*) throw (Error::Exception) [pure virtual]

This function initializes the makeSearchTemplate process and sets all needed parameters.

The function is called once by the NIST application before n >= 1 calls to makeSearchTemplate. The implementation must tolerate execution of multiple instances of the makeSearchTemplate process running simultaneously and independently on the same machine and/or across multiple machines.

Parameters

in	*configuration_location*	A read-only directory containing vendor-supplied configuration

Returns
The object containing a required status code and optional information string.

Exceptions

Error::Exception	There was an error processing this request, and the exception string may contain additional information.

D.5.1.6 virtual ReturnStatus FPVTE2012::SDKInterface::makeSearchTemplate (const FingerImageSet & *fingers*, Memory::uint8Array & *search_template*, FingerOutputSet & *output_features*) throw (Error::Exception) [pure virtual]

This function takes a FingerImageSet and outputs a search template representing one or more fingers. The search template can be produced with a different algorithm than makeEnrollmentTemplate.

The memory for the template is managed in the uint8Array (an AutoArray) object. This routine should not perform memory allocation but can resize the AutoArray if more space is needed.

Parameters

in	*fingers*	A set of finger images. The implementation will have to adjust to the number of available fingerprints based on the dataset being used. For example 10 rolled or 2 index plain impressions.
in	*fingers*	A set of finger images. The implementation will have to adjust to the number of available fingerprints based on the dataset being used. For example 10 rolled or 2 index plain impressions.
out	*search_template*	The search template, extracted from the finger images. The format is not regulated. The memory is allocated in the AutoArray object.
out	*output_features*	Additional information for the images, produced by the SDK. This information will be used in final results analysis.

Returns
The object containing a required status code and optional information string.

Exceptions

Error::Exception	There was an error processing this request, and the exception string may contain additional information.

D.5.1.7 virtual ReturnStatus FPVTE2012::SDKInterface::initIdentificationStageOne (const string & *configuration_directory,* const string & *enrollment_directory,* const uint8_t *blade_number*) throw (Error::Exception) `[pure virtual]`

This function initializes identifyTemplateStageOne.

The function will be called to initialize each blade that will contain a portion of the enrolled templates. The number of blades will be the same as used in finalizeEnrollment.

Parameters

in	*configuration_location*	A read-only directory containing vendor-supplied configuration
in	*enrollment_directory*	The top-level directory in which all finalized enrollment data resides. The directory will have read-only access.
in	*blade_number*	Blade number from blades in finalizeEnrollment that is being initialized. This parameter lets the process know which piece of the enrolled templates to load into memory. Blades are numbered 0 to B - 1.

Returns
The object containing a required status code and optional information string.

Exceptions

Error::Exception	There was an error processing this request, and the exception string may contain additional information.

D.5.1.8 virtual ReturnStatus FPVTE2012::SDKInterface::identifyTemplateStageOne (const uint32_t *search_ID,* const Memory::uint8Array & *search_template,* const string & *stage1_data_directory*) throw (Error::Exception) `[pure virtual]`

This function searches a template against the partial enrollment set selected by the blade_number in initIdentificationstageOne.

Parameters

in	*search_ID*	The ID of the search subject. This ID does not identify the subject it is merely a sequence number used to distinguish different searches performed by the system and it will be used as the input to identifyTemplateStageTwo.
in	*search_template*	A template from makeSearchTemplate. The size is given by the AutoArray object.
in	*stage1_data_directory*	This directory will have read-write access. The output information from identifyTemplateStageOne that is needed in indentifyTemplateStageTwo is written in this directory. This directory will be unique for each search performed.

Returns
The object containing a required status code and optional information string.

Exceptions

Error::Exception	There was an error processing this request, and the exception string may contain additional information.

D.5.1.9 virtual ReturnStatus FPVTE2012::SDKInterface::initIdentificationStageTwo (const string & *configuration_directory*, const string & *enrollment_directory*) throw (Error::Exception) [pure virtual]

This function initializes identifyTemplateStageTwo.

This second stage of identification uses the output results from identifyTemplateStageOne to produce a candidate list for the search subject.

Parameters

in	*configuration_location*	A read-only directory containing vendor-supplied configuration
in	*enrollment_directory*	The top-level directory in which all finalized enrolled data resides. The directory will have read-only access.

Returns
The object containing a required status code and optional information string.

Exceptions

Error::Exception	There was an error processing this request, and the exception string may contain additional information.

D.5.1.10 virtual ReturnStatus FPVTE2012::SDKInterface::identifyTemplateStageTwo (const uint32_t *search_ID*, const string & *stage1_data_directory*, CandidateSet & *candidates*) throw (Error::Exception) [pure virtual]

This function takes the results from identifyTemplateStageOne and produces a candidate list for the search subject.

Parameters

in	*search_ID*	The ID of the search subject. This ID does not identify the subject it is merely a sequence number used to distinguish different searches performed by the system.
in	*stage1_data_directory*	This directory will have read-only access and contains all the stored output data from the identify-TemplateStageOne processes for the search subject.
out	*candidates*	The candidate list returned from identifyTemplateStageTwo.

Returns
The object containing a required status code and optional information string.

Exceptions

Error::Exception	There was an error processing this request, and the exception string may contain additional information.

D.5.1.11 static SDKIptr FPVTE2012::SDKInterface::getSDK ( ) `[static]`

Factory function to return a managed pointer to the SDK object.

This function is implemented by the SDK library and must return a managed pointer to the SDK object.

The documentation for this class was generated from the following file:

- fpvte2012.h

D.6 FPVTE2012::StatusCode Class Reference

Public Types

- enum Kind { Success = 0, ImageSizeNotSupported = 1, TemplateTypeNotSupported = 2, FailedToExtract = 3, FailedToMatch = 4, **FailedToParseInput** = 5, Vendor = 6 }

    A class that contains an enumeration which defines the set of status codes.

D.6.1 Member Enumeration Documentation

D.6.1.1 enum FPVTE2012::StatusCode::Kind

A class that contains an enumeration which defines the set of status codes.

The status codes that are returned from a function call:

Enumerator:

Success	Successful completion
ImageSizeNotSupported	Image size too small or large
TemplateTypeNotSupported	Unsupported template type
FailedToExtract	Could not extract template from image
FailedToMatch	Could not match samples
Vendor	Vendor-defined error

The documentation for this class was generated from the following file:

- fpvte2012.h