Proprietary Fingerprint Template III

Test Plan and Application Programming Interface

Last Updated: 27 January 2022

Contents

Disclaimer

Certain commercial equipment, instruments, or materials are identified in this document in order to specify the experimental procedure adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the materials or equipment identified are necessarily the best available for the purpose.

1 Introduction

The Proprietary Fingerprint Template (PFT) evaluation is one of many tests of fingerprint technology run by the National Institute of Standards and Technology (NIST). PFT specifically examines the accuracy and performance of *proprietary* fingerprint templates—encodings of features of a finger that can only be understood by their creator. This is in contrast to a *standardized* template that can be understood by any standards-compliant implementation. As fingerprint imaging and computing technology improve, so must evaluations of proprietary fingerprint template software. As a response, NIST is conducting PFT III.

1.1 Background

From 2003 to 2010, NIST conducted the first PFT evaluation [1]. This ongoing test allowed participants to submit software for evaluation at any time. Implementations were provided images of fingerprints and encoded them into proprietary templates. Later, the implementation was provided two templates and asked to produce a similarity score (verification or one-to-one matching). This process was repeated for thousands of images across several operationally-collected datasets of ink and live scan rolled and plain fingerprints. At the time, in-house offline testing was a novel idea, but the PFT evaluation proved it was a viable strategy.

In 2010, NIST announced a change to PFT and introduced PFT II [2]. This revision greatly increased the size of the datasets used, enabling measurement of false non-match rates (FNMRs) at lower false match rates (FMRs). Other metrics, such as average speed per match and and average template size, were additionally reported.

As of 2019, the qualification criteria for PFT II were beginning to show their age. NIST sought to build upon PFT II and update the test to better align with the current state of fingerprint template generation and matching technology. New and larger datasets were added, including 1 000 pixels per inch (PPI)/393.70 pixels per centimeter (PPCM) images and images from new types of imaging devices, such as non-contact sensors. Computational resources were additionally adjusted for current hardware.

1.2 What's New Since PFT II

- Variable resolution (including 1 000 PPI) and non-contact sensor imagery.
- Datasets from other NIST evaluations for comparison.
- 64-bit software libraries linking under Ubuntu Server 20.04.03 LTS.
- Faster turnaround of report cards.
- Tighter computational complexity requirements.

2 Evaluation Data

2.1 Source

Data used in PFT III come from a variety of sources. The vast majority of images are operational in nature. This means they were collected by law enforcement, border protection, or other local or federal government employees as a part of their professional duties. Other data may come from subjects recruited as part of institutional review board (IRB)-approved collections.

2.2 Quality

Due to the operational nature of the source of the imagery, the quality of the images vary dramatically between datasets and samples within the datasets. No open-source algorithm currently exists that can quantify fingerprint image quality for *all* combinations of resolution, bit depth, and sensor type represented in PFT III, and therefore, NIST will not be providing quality values along with the imagery during the test. Participants are encouraged to use the metadata provided with the imagery (Section 2.3) to assess quality and store this value in their proprietary template as an aid during template verification.

2.3 Metadata

Participants will be provided with known metadata about each image during template creation. Metadata may include the finger position, impression type, and sensor capture technology. Implementations are expected to successfully process all types of images presented, regardless of metadata.

2.4 Access

The PFT III test datasets are protected under the Privacy Act (5 U.S.C. §552a), and will be treated as controlled unclassified information (CUI) and/or law enforcement sensitive. PFT III participants will not have access to PFT III test data, before, during, or after the test. NIST will provide similar publicly-available data that can be used to prepare submissions for PFT III.

2.5 Format

The software library must be capable of processing fingerprint images in uncompressed raw 8 bit (one byte per pixel) grayscale format. Images shall follow the scan sequence as defined by ISO/IEC 19794-4:2005, §6.2, paraphrased here, and visualized in Figure 1. The origin is the upper-left corner of the image. The X-coordinate (horizontal) position shall increase positively from the origin to the right side of the image. The Y-coordinate (vertical) position shall increase positively from the origin to the bottom of the image.

Raw single-channel images are canonically encoded. The minimum value that will be assigned to a "black" pixel is zero. The maximum value that will be assigned to a "white" pixel is 255. Intermediate gray levels will have assigned values of 1 to 254. The pixels are stored left to right, top to bottom. The number of bytes in an image is equal to its height multiplied by its width as measured in pixels. The image width and height in pixels will be supplied to the software library as supplemental information.

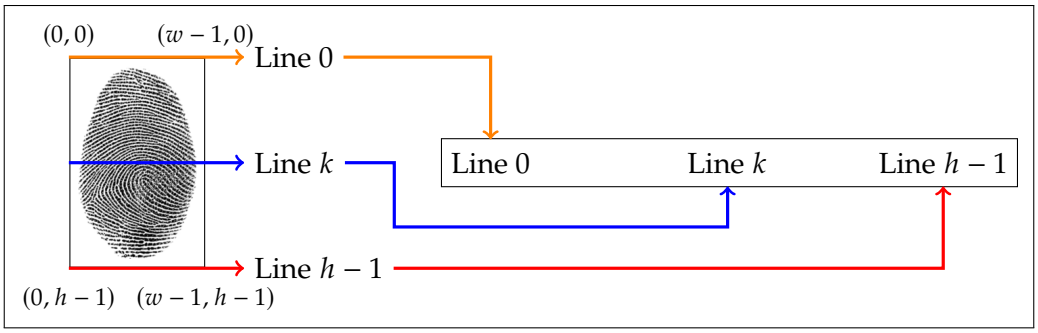

Figure 1: Order of image scanlines in data passed to PFT III implementations for 8 bits per pixel (bpp) images. Fingerprint image sourced from NIST Special Database 302.

2.6 Resolution

Images for this test will employ **varying resolutions**, but primarily 500 PPI (196.85 PPCM) and 1 000 PPI (393.70 PPCM). Horizontal and vertical resolutions will be equivalent. Implementations shall be able to handle images at all resolutions *without* resampling. Implementations will also be provided images at "unusual" resolutions, such as 333 PPI and 600 PPI to identify tradeoffs in resolutions.

3 Application Programming Interface

3.1 Enumerations

Most enumeration values correspond with similarly-named fields from the document American National Standard Institute/NIST-Information Technology Laboratory (ANSI/NIST-ITL) 1-2011: Update 2015 [3].

Enumeration	Explanation
`PlainContact`	`IMP` 0
`RolledContact`	`IMP` 1
`LiveScanSwipe`	`IMP` 8
`PlainContactlessStationary`	`IMP` 24
`RolledContactlessStationary`	`IMP` 25
`Unknown`	`IMP` 29
`RolledContactlessMoving`	`IMP` 41
`PlainContactlessMoving`	`IMP` 42

Table 1: `PFTIII::Impression`.

Enumeration	Explanation
`Unknown`	`FCT` 0
`ScannedInkOnPaper`	`FCT` 2
`OpticalTIRBright`	`FCT` 3
`OpticalDirect`	`FCT` 5
`Capacitive`	`FCT` 9
`Electroluminescent`	`FCT` 11

Table 2: `PFTIII::FrictionRidgeCaptureTechnology`.

Enumeration	Explanation
`Success`	Successfully performed operation
`Failure`	Unable to perform operation

Table 3: `PFTIII::Result`.

Enumeration	Explanation
`Unknown`	`FGP` 0
`RightThumb`	`FGP` 1
`RightIndex`	`FGP` 2
`RightMiddle`	`FGP` 3
`RightRing`	`FGP` 4
`RightLittle`	`FGP` 5
`LeftThumb`	`FGP` 6
`LeftIndex`	`FGP` 7
`LeftMiddle`	`FGP` 8
`LeftRing`	`FGP` 9
`LeftLittle`	`FGP` 10
`RightExtraDigit`	`FGP` 16
`LeftExtraDigit`	`FGP` 17

Table 4: `PFTIII::FrictionRidgeGeneralizedPosition`.

Enumeration	Explanation
`Supported`	Image supported
`InvalidImageData`	Image data is not parsable
`VendorDefined`	Vendor-defined error (described in `message`)

Table 5: `PFTIII::FingerImageStatus::Code`.

3.2 Classes and Structures

3.2.1 FingerImageStatus

```
FingerImageStatus::Code code;
std::string message;
```

Figure 2: `PFTIII::FingerImageStatus` members.

```
FingerImageStatus(
    const Code code = Code::Supported,
    const std::string &message = "");
```

Figure 3: `PFTIII::FingerImageStatus` constructor.

3.2.1.1 Members

- `code`: A `FingerImageStatus::Code` summarizing if this software library can support feature extraction from an image described in a `FingerImage`.
- `message`: A message providing more information about why a particular `Code` was chosen. *Optional*.

3.2.1.2 Description

`PFTIII::FingerImageStatus` is a structure that allows participants to return information whether an image described by a `FingerImage` is supported by their software library during calls to `createProprietaryTemplate()`. Under normal operating conditions, robust implementations need only to indicate `FingerImageStatus::Code::Supported`. Under failure conditions, it may be useful for participants to provide debugging information in the `message` parameter of a `FingerImageStatus` to help resolve the issue.

In `createProprietaryTemplate()`, implementations should consider the `FingerImage` and set this value *before* attempting to perform feature extraction and create a proprietary template. Errors that occur *after* validating the parameters (i.e., during the feature extraction or template generation step) should be expressed via `CreateProprietaryTemplateResult.result` (Section 3.2.2).

Examples of using `FingerImageStatus` are shown in Figure 4.

- NIST will be unable to provide participants with the contents of `message` if it contains information about fingerprint imagery, as the imagery and derivative information used in this test may not be distributed. Information that is not immediately decipherable by humans (e.g., Base64-encoded data) will be assumed sensitive.
- The contents of `message` shall match the regular expression `[[:graph:] ]*`.

```
/* Use the default arguments to indicate success */
const PFTIII::FingerImageStatus status{};

/* Provide a debugging message */
const PFTIII::FingerImageStatus status{
    FingerImageStatus::Code::VendorDefined, "Couldn't initialize extractor"};
```

Figure 4: Ways to return a `PFTIII::FingerImageStatus`.

3.2.2 CreateProprietaryTemplateResult

```
Result result;
std::vector<uint8_t>
    proprietaryTemplate;
std::string message;
```

Figure 5: `PFTIII::CreateProprietaryTemplateResult` members.

```
CreateProprietaryTemplateResult();
```

Figure 6: `PFTIII::CreateProprietaryTemplateResult` constructor.

3.2.2.1 Members

- `result`: A `Result` summarizing the success or failure of extracting features and creating a proprietary template in `createProprietaryTemplate()`.
- `message`: A message providing more information about why a particular `Result` was chosen. *Optional*.

3.2.2.2 Description

`PFTIII::CreateProprietaryTemplateResult` is a structure that allows participants to return information about the status of extracting features from an image and creating a proprietary template. Under normal operating conditions, participants need only to indicate `Result::Success`. Under failure conditions, it may be useful for participants to provide debugging information in the `message` parameter of a `CreateProprietaryTemplateResult` to help resolve the issue. Examples of using `CreateProprietaryTemplateResult` are shown in Figure 7.

A failure due to not understanding or supporting input parameters should be recorded with a `FingerImageStatus` (Section 3.2.1), not a `Result`.

- NIST will be unable to provide participants with the contents of `message` if it contains information about fingerprint imagery, as the imagery and derivative information used in this test may not be distributed. Information that is not immediately decipherable by humans (e.g., Base64-encoded data) will be assumed sensitive.
- The contents of `message` shall match the regular expression `[[:graph:] ]*`.

```
/* Use convenience method to indicate success */
std::vector<uint8_t> proprietaryTemplate = ...;
const auto result1 = PFTIII::CreateProprietaryTemplateResult::success(
    proprietaryTemplate);

/* Use convenience method to provide a debugging message */
const auto result2 = PFTIII::CreateProprietaryTemplateResult::failure(
    "Image quality is too poor");
```

Figure 7: Ways to return a `PFTIII::CreateProprietaryTemplateResult`.

3.2.3 CompareProprietaryTemplatesStatus

```
Result result;
std::string message;
```

Figure 8: `PFTIII::CompareProprietaryTemplatesStatus` members.

```
CompareProprietaryTemplatesStatus(
    const Result result = Result::Success,
    const std::string &message = "");
```

Figure 9: `PFTIII::CompareProprietaryTemplatesStatus` constructor.

3.2.3.1 Members

- `result`: A `Result` summarizing the success or failure of comparing two templates in `compareProprietaryTemplates()`.
- `message`: A message providing more information about why a particular `Result` was chosen. *Optional*.

3.2.3.2 Description

`PFTIII::CompareProprietaryTemplatesStatus` is a structure that allows participants to return information about the status of comparing two proprietary templates. Under normal operating conditions, participants need only to indicate `Result::Success`. Under failure conditions, it may be useful for participants to provide debugging information in the `message` parameter of a `CompareProprietaryTemplatesStatus` to help resolve the issue. Examples of using `CompareProprietaryTemplatesStatus` are shown in Figure 10.

- NIST will be unable to provide participants with the contents of `message` if it contains information about fingerprint imagery, as the imagery and derivative information used in this test may not be distributed. Information that is not immediately decipherable by humans (e.g., Base64-encoded data) will be assumed sensitive.
- The contents of `message` shall match the regular expression `[[:graph:] ]*`.

```
/* Use the default arguments to indicate success */
const PFTIII::CompareProprietaryTemplatesStatus status1{};

/* Use convenience method to provide a debugging message */
const auto status2 = PFTIII::CompareProprietaryTemplatesStatus::failure(
    "Probe template has too few minutia");
```

Figure 10: Ways to return a `PFTIII::CreateProprietaryTemplatesStatus`.

3.2.4 FingerImage

```
uint16_t width;
uint16_t height;
uint16_t ppi;
std::vector<uint8_t> pixels;
Impression imp;
FrictionRidgeCaptureTechnology
    frct;
FrictionRidgeGeneralizedPosition
    frgp;
```

Figure 11: `PFTIII::FingerImage` members.

```
FingerImage(
    const uint16_t width,
    const uint16_t height,
    const uint16_t ppi,
    const std::vector<uint8_t> &pixels,
    const Impression imp =
        Impression::Unknown,
    const FrictionRidgeCaptureTechnology frct =
        FrictionRidgeCaptureTechnology::
        Unknown,
    const FrictionRidgeGeneralizedPosition frgp =
        FrictionRidgeGeneralizedPosition::
        Unknown);
```

Figure 12: `PFTIII::FingerImage` constructor.

3.2.4.1 Members

- `width`: Width of the image.
- `height`: Height of the image.
- `ppi`: Resolution of the image in PPI.
- `pixels`: Raw pixel data of image.
- `imp`: Impression type of the depicted finger.
- `frct`: Capture technology of the sensor that created this image.
- `frgp`: Position of the depicted finger.

3.2.4.2 Description

A container for data and properties of an image of a single finger. Participants will never need to construct an instance of `FingerImage`.

3.2.4.3 Note

To pass `FingerImage.pixels` as an C-style array without duplicating memory, invoke the `data()` method, as shown in Figure 13.

```
/* Given this C function declaration ... */
void find_minutia(void *data, size_t size, struct proprietary *out);

/* ... pass image data from createProprietaryTemplate () like this : */
struct proprietary out;
find_minutia(fingerImage.pixels.data(), fingerImage.pixels.size(), &out);
```

Figure 13: Converting image `data` to a C-style array in constant time without additional memory allocations.

3.2.5 SubmissionIdentification

```
uint16_t versionNumber;
std::string libraryIdentifier;
std::tuple<std::string, bool>
    featureExtractionAlgorithm-
    MarketingIdentifier;
std::tuple<std::string, bool>
    comparisonAlgorithmMarketing-
    Identifier;
std::tuple<uint16_t, bool>
    cbeffFeatureExtractionAlgorithm-
    ProductOwner;
std::tuple<uint16_t, bool>
    cbeffFeatureExtractionAlgorithm-
    Identifier;
std::tuple<uint16_t, bool>
    cbeffComparisonAlgorithmProduct-
    Owner;
std::tuple<uint16_t, bool>
    cbeffComparisonAlgorithmIdentifier;
```

Figure 14: `PFTIII::SubmissionIdentification` members.

```
SubmissionIdentification(
    const uint16_t versionNumber,
    const std::string &libraryIdentifier,
    const std::tuple<std::string, bool>
        &featureExtractionAlgorithm-
        MarketingIdentifier =
        std::make_tuple("", false),
    const std::tuple<std::string, bool>
        &comparisonAlgorithmMarketing-
        Identifier =
        std::make_tuple("", false),
    const std::tuple<uint16_t, bool>
        cbeffFeatureExtractionAlgorithm-
        ProductOwner =
        std::make_tuple(0x0000, false),
    const std::tuple<uint16_t, bool>
        cbeffFeatureExtractionAlgorithm-
        Identifier =
        std::make_tuple(0x0000, false),
    const std::tuple<uint16_t, bool>
        cbeffComparisonAlgorithmProduct-
        Owner =
        std::make_tuple(0x0000, false),
    const std::tuple<uint16_t, bool>
        cbeffComparisonAlgorithmIdentifier =
        std::make_tuple(0x0000, false));
```

Figure 15: `PFTIII::SubmissionIdentification` constructor.

3.2.5.1 Members

- `versionNumber`: Version number of this submission. Required to be unique for each new submission. **Required**.
- `libraryIdentifier`: Non-infringing identifier of this submission. Should be the same for all submissions from an organization. **Required**. Case sensitive. Must match the regular expression `[:alnum:]+`.
- `featureExtractionAlgorithmMarketingIdentifier`: Non-infringing marketing name of the feature extraction algorithm include in this submission. *Optional*. Case sensitive. Must match the regular expression `[[:graph:] ]*`. The first tuple member is the value and second is a boolean indicating the initialization status of the value.
- `comparisonAlgorithmMarketingIdentifier`: Non-infringing marketing name of the comparison algorithm include in this submission. *Optional*. Case sensitive. Must match the regular expression `[[:graph:] ]*`. The first tuple member is the value and the second is a boolean indicating the initialization status of the value.
- `cbeffFeatureExtractionAlgorithmProductOwner`: Common Biometric Exchange Formats Framework (CBEFF) Product Owner of the feature extraction algorithm, if registered. *Op-*

tional, unless `cbeffFeatureExtractionAlgorithmIdentifier` is supplied. The first tuple member is the value and second is a boolean indicating the initialization status of the value.

- `cbeffFeatureExtractionAlgorithmIdentifier`: CBEFF Feature Extraction Algorithm Identifier, if registered. *Optional*. The first tuple member is the value and the second is a boolean indicating the initialization status of the value.
- `cbeffComparisonAlgorithmProductOwner`: CBEFF Product Owner of the template comparison algorithm, if registered. *Optional*, unless `cbeffComparisonAlgorithmIdentifier` is supplied. The first tuple member is the value and second is a boolean indicating the initialization status of the value.
- `cbeffComparisonAlgorithmIdentifier`: CBEFF Comparison Algorithm Identifier, if registered. *Optional*. The first tuple member is the value and the second is a boolean indicating the initialization status of the value.

3.2.5.2 Description

Identifying information about this submission that will be included in reports.

3.3 Interface

Participants in PFT III must submit a software library that fully implements the pure abstract C++ class `PFTIII::Interface`. Since the PFT III test driver will not know the name of the `PFTIII::Interface` class at compile time, the software library must also implement a factory method to return an instance of their implementation. NIST's declaration of the factory method is shown in Figure 16.

3.3.1 Obtain PFT III Implementation

```
std::shared_ptr<PFTIII::Interface>
getImplementation(
    const std::filesystem::path &configurationDirectory);
```

Figure 16: Declaration of a function to obtain an instance of the participant's `PFTIII::Interface` implementation.

3.3.1.1 Description

Obtain a managed pointer to an object implementing `PFTIII::Interface`. To avoid name collisions, the class implementing the interface should be in the `PFTIII` namespace.

3.3.1.2 Parameters

- `configurationDirectory`: Path to a directory on disk where participant-provided configuration files are located.

3.3.1.3 Return

A managed pointer to the participant's implementation of `PFTIII::Interface`. A sufficient implementation of this method for an implementation whose constructor has no arguments could be the return statement shown in Figure 17.

```
return (std::make_shared<PFTIII::Implementation>());
```

Figure 17: A sufficient implementation of `PFTIII::Interface::getImplementation()`.

3.3.1.4 Speed

This method shall return in $\leq$10 s.

3.3.2 Identification

```
PFTIII::SubmissionIdentification
PFTIII::Interface::getIdentification()
    const;
```

Figure 18: Declaration of a method to obtain identification information for this submission.

3.3.2.1 Description

This method allows for the retrieval of identification information of the library at runtime.

- The returned value's `versionNumber` member shall be identical to the four hexadecimal characters prior to the extension of the submitted software library's name (Section 4.1.5).
- The `libraryIdentifier` member of the returned value shall be identical to the string surrounded by underscores, just prior to the four hexadecimal digit version, in the submitted software library's name (Section 4.1.5).

3.3.2.2 Return

This method shall immediately return a `PFTIII::SubmissionIdentification` of the identifier and version number for this software library. Marketing names to be printed in publications may be provided. Similarly, if your organization is registered with International Biometrics + Identity Association (IBIA) and has received CBEFF information, include the information in the respective fields in the `SubmissionIdentification` struct for publication in report cards.

Sufficient implementations for the library `libpftiii_initech_101D.so` are shown in Figure 19.

```
/* A sufficient implementation of getIdentification */
return {"initech", 0x101D};

/* A more verbose implementation of getIdentification */
PFTIII::SubmissionIdentification id{};
id.libraryIdentifier = "initech";
id.version = 0x101D;
id.featureExtractionAlgorithmMarketingIdentifier = std::make_tuple("IniEx (v2.16)", true);
id.comparisonAlgorithmMarketingIdentifier = std::make_tuple("IniCmp (v2.3.3)", true);
id.cbeffProductOwner = std::make_tuple(0x000F, true);
id.cbeffFeatureExtractionAlgorithmIdentifier = std::make_tuple(0x351A, true);
id.cbeffComparisonAlgorithmIdentifier = std::make_tuple(0x42F2, true);
return (id);
```

Figure 19: Implementations of `PFTIII::Interface::getIdentification()` for the library `libpftiii_initech_101D.so`.

3.3.2.3 Speed

This method shall return immediately ($\leq \approx 0.001$ s).

3.3.3 Feature Extraction

```
std::tuple<FingerImageStatus, CreateProprietaryTemplateResult>
PFTIII::Interface::createProprietaryTemplate(
    const FingerImage &fingerImage);
```

Figure 20: Declaration of a method that extracts features from a fingerprint image and encodes them into a proprietary template.

3.3.3.1 Description

This method provides an image of a single fingerprint to an implementation. The implementation should extract, encode, and return those features for later use in verification.

- This method shall be deterministic. Providing the same `FingerImage` repeatedly shall result in the same returned value, regardless of time or previously-encountered images.
- All values, including `Unknown`, for types `FrictionRidgeGeneralizedPosition`, `Impression`, and `FrictionRidgeCaptureTechnology` shall be supported. It is not acceptable to return a failure solely because metadata is not specified.
- Entire classifications of images (e.g., 1 000 PPI, `RolledContactlessMoving`, etc.) shall not fail to have templates created.

3.3.3.2 Return

The `Code` member of the `FingerImageStatus` member in the return of this method shall be set to `Supported` if the `FingerImage` passed to this method should be supported, or another approved `Code` with an optional `message` otherwise. If the `FingerImageStatus::Code` is not `Supported`, the second member of the returned tuple will not be consulted.

The `Result` member of the `CreateProprietaryTemplateResult` member in the return of this method shall be set to `Success` if the implementation was able to successfully extract features and produce a proprietary template in `proprietaryTemplate` for later use in `compareProprietaryTemplates()`. Otherwise, `Result` shall be set to `Failure` and an optional debugging `message` shall be provided.

3.3.3.3 Note

- Be careful not to confuse `FingerImageStatus::Code` and `Result`. `FingerImageStatus::Code` indicates if it should have been technically possible to call and complete the method based on the inputs provided, while `Result` indicates the actual result of extracting features and encoding a template.
  - PFT III does not distinguish between failures to extract features, failures to encode extracted features into a template, not supporting an image type, or *any* other situations resulting in the lack of `proprietaryTemplate` being produced. All such failures will be categorized as *failure to enroll*.
- This method does not differentiate between probe and reference templates.

- `proprietaryTemplate` will be saved *only* when `Result` is `Success` and `FingerImageStatus::Code` is `Supported`. In *all* other cases, an empty (0 byte) entry will be saved by the PFT III test driver instead. Regardless, *all* pre-determined comparisons will be performed.

3.3.3.4 Speed

This method shall return on average in ≤0.5 s, as measured on a dedicated timing sample.

3.3.4 Template Comparison

```
std::tuple<CompareProprietaryTemplatesStatus, double>
PFTIII::Interface::compareProprietaryTemplates(
    const std::vector<uint8_t> &probeTemplate,
    const std::vector<uint8_t> &referenceTemplate);
```

Figure 21: Declaration of a method that compares two proprietary templates.

3.3.4.1 Description

This method provides two proprietary templates (created by `createProprietaryTemplate()`, detailed in Section 3.3.3), and returns a value indicating their similarity in terms of being derived from the same finger.

- This method shall be deterministic. Providing the same two templates repeatedly shall result in the same similarity score returned, regardless of time or previously-encountered comparisons.

3.3.4.2 Return

The `Result` member of the `CompareProprietaryTemplatesStatus` member in the return of this method shall be set to `Success` if the method was able to be successfully executed, or `Failure` with an optional `message` otherwise. If returning `Success`, the `double` member of the returned tuple shall be the similarity of comparing `probeTemplate` to `referenceTemplate`.

3.3.4.3 Note

- This method must tolerate empty (0 byte) templates for both probe and reference templates, in the case of failure when calling `createProprietaryTemplates()`.
- The similarity score will be used in analysis so long as the `Result` when returned from `createProprietaryTemplates()` was `Success`. In other occurrences, the lowest non-mated similarity score returned (as determined after all evaluation comparisons have completed) will be used instead. That is, a comparison score will be assigned for *all* evaluation comparisons, even if the comparison fails or the templates involved in the comparison do not exist.

3.3.4.4 Speed

This method shall return on average in $\leq$0.01 s, as measured on a dedicated timing sample.

4 Software and Documentation

4.1 Software Libraries and Platform Requirements

The functions specified in Section 3 shall be implemented exactly as defined in a software library. The header file used by the PFT III *test driver* executable is provided on the PFT III website in the PFT III validation package (Section 4.3).

4.1.1 Restrictions

4.1.1.1 Dynamic Library

Participants shall provide NIST with binary code in the form of a software library only (i.e., no source code or headers). Software libraries must be submitted in the form of a dynamic/shared library file (i.e., `.so` file). This library shall be know as the *core* library, and shall be named according to the guidelines in Section 4.1.5. Static libraries (i.e., `.a` files) are not allowed. Multiple shared libraries are permitted if technically required and are compatible with the validation package (Section 4.3). Any required libraries that are not standard to Ubuntu Server 20.04.03 LTS must be built and submitted alongside the core library. All submitted software libraries will be placed in a single directory, and NIST will add this directory to the runtime library search path list (`RUNPATH`, see Section 4.1.2).

4.1.1.2 Single Configuration

Individual software libraries provided must not include multiple modes of operation or algorithm variations managed by NIST. No NIST-managed configurations or options will be tolerated within one library. For example, the use of two different minutia sorting techniques would be split across two separate software libraries (though the Proprietary Fingerprint Template III (PFT III) application indicates that NIST will only accept one submission every 90 days).

Supplemental non-library files (e.g., pre-specified configurations and training models) are permitted. If necessary, these files shall be placed in a dedicated directory as specified in the validation submission instructions (Section 4.3). Filenames and checksums of all provided files will be reported in PFT III analysis reports. The path to such files will be provided as a parameter to `getImplementation()` (Section 3.3.1). No vendor-specific environment variables will be set for an implementation to affect operation. NIST will additionally not alter any system-level configuration.

Example
A participant submits a software library that internally has a customizable minutia sorting algorithm. The software library decides which sorting algorithm to used based on the contents of a text file, `config.txt`. The participant submits their software library *and* `config.txt` pre-configured to use a sorting algorithm A. After NIST discovers a defect, the participant realizes the defect is not present when the sorting algorithm is set to B, and could be corrected by a small change to `config.txt`. Even though the change is minor, the participant must submit a new `config.txt` *and* a new software library with incremented version number (Section 4.1.5) to correct the defect.

4.1.1.3 Multiprocessing

The software library shall not make use of threading, forking, OpenMP, or any other multiprocessing techniques. The PFT III test application operates as an Message Passing Interface (MPI) job

over multiple compute nodes, and then `fork`s itself into many processes. In the test environment, **there is no advantage to threading**. It limits the usefulness of NIST's batch processing and makes it impossible to compare timing statistics across PFT III participants.

4.1.1.4 Deterministic Operation

The software library under test shall remain stateless and deterministic. application programming interface (API) calls with the same inputs shall produce the same outputs on all nodes at all times.

4.1.1.5 Filesystem

The software library under test shall not read from or write to any file system or file handle, including standard streams. It shall not attempt any external communication such as network connections via sockets.

Software libraries under test shall be permitted to read configuration files at or below the filesystem path provided in `getImplementation()`. This path and its contents shall be read-only.

4.1.2 External Dependencies

It is preferred that the API specified by this document be implemented in a single core library if possible, to reduce the likelihood of difficult to remotely debug linking errors. Additional libraries may be submitted that support this core library file (i.e., the core library file may have dependencies implemented in other libraries if a single library is not feasible). It is recommended that the `RUNPATH` of these dependent libraries be set to `$ORIGIN`, since the only participant library that the PFT III test application will explicitly link is the core library. The PFT III test application's `RUNPATH` will include the directory containing the participant's core library. Filenames and checksums of all library files will be reported in PFT III analysis reports.

4.1.3 `libpftiii.so`

Core libraries will need to depend on the NIST-provided `libpftiii.so`. Participants shall not alter the provided header file for `libpftiii.so`. NIST will build and supply `libpftiii.so`, and so this library shall **not** be included in validation submissions (Section 4.3).

4.1.4 Hardware Dependencies

Use of intrinsic functions and inline assembly is allowed and encouraged, but software libraries shall be able to run and are required to pass validation (Section 4.3) on Intel CPUs, including, but not limited to, **Intel Xeon Gold 6140**, **Intel Xeon E5-2680**, and **Intel Xeon E5-4650**. Unavailable intrinsics shall be avoided where unsupported and their lack of use shall not change output. Speed tests that run on a fixed sample dataset will be run as described in Section 4.4.

4.1.5 Naming

The core software library submitted for PFT III shall be named in a predefined format. The first part of the software library's name shall be `libpftiii_`. The second piece of the software library's name shall be a non-infringing and case-sensitive unique identifier that matches the regular expression `[:alnum:]+` (likely the participating organization's name), followed by an underscore. The final part of the software library's name shall be a four uppercase hexadecimal digit version number,

```
mpicxx -o pft3 pft3.cpp -Llib -Wl,--enable-new-dtags -Wl,-rpath,lib -lpftiii \
       -lpftiii_initech_101D -std=c++17
```

Figure 22: Example compilation and link command for the PFT III test application.

followed by a file extension. **Be cognizant of the name** provided, as this will be name NIST uses to refer to your submission in reports. Supplemental libraries may have any name, but the core library must be dependent on supplemental libraries in order to be linked correctly. The **only** participant library that will be explicitly linked to the PFT III test driver is the core library, as demonstrated in Sections 4.1.2 and 4.1.6.

The version number shall match the uppercase hexadecimal version number with leading `0`s, as returned by `ELFT::ExtractionInterface::getIdentification()`. With this naming scheme, **every core library received by NIST shall have a unique filename**. Incorrectly named or versioned software libraries will be rejected.

Note
When NIST encounters an error, NIST will expect a different version number on resubmission. Incrementing the version number is *not* a penalty. It is NIST's way of ensuring they're always running with the latest version of a software library and its templates, and that analysis is run against appropriate log files.

NIST discourages trying to align version numbers for marketing use. Instead, make use of the marketing identification features of the API described in Section 3.3.2 for this purpose.

Example
Initech submits a software library named `libpftiii_initech_101C.so` with build 4 124 of their algorithm. This library returns `{"initech", 0x101C}` from `getIdentification()`. NIST determines that Initech's `search()` method is too slow and rejects the library. Initech submits build 4 125 to correct the defect in 4 124. Initech updates `getIdentification()` in their implementation to return `{"initech", 0x101D}` and renames their library to `libpftiii_initech_101D.so`. In PFT III analysis reports, NIST refers to Initech's library as `initech+101D`.

4.1.6 Operating Environment

The software library will be tested in non-interactive "batch" mode (i.e., without terminal support) in an isolated environment (i.e., no Internet connectivity). Thus, the software library under test shall not use any interactive functions, such as graphical user interface calls, or any other calls that require terminal interaction (e.g., writes to `stdout`) or network connectivity. Any messages for debugging failure conditions shall be provided via the `message` parameter of `ReturnStatus` (or via exceptions in extreme cases) and *not* write to files or the console.

NIST will link the provided library files to a C++17 language test driver application using the compiler `g++` (version `Ubuntu 9.3.0-17ubuntu1~20.04`, via `mpicxx`) under **Ubuntu Server 20.04.03 LTS**, as seen in Figure 22.

Participants are required to provide their software libraries in a format that is linkable using `g++` with the NIST test driver. All compilation and testing will be performed on 64-bit hardware running Ubuntu Server 20.04.03 LTS. Participants are **strongly encouraged** to verify library-level compatibility with `g++` on Ubuntu Server 20.04.03 LTS **prior to** submitting their software to NIST to avoid unexpected problems.

4.2 Usage

4.2.1 Software Libraries

The software library shall be executable on any number of machines without requiring additional machine-specific license control procedures, activation, hardware dongles, or any other form of rights management.

The software library under test's usage shall be **unlimited**. No usage controls or limits based on licenses, execution date/time, number of executions, etc., shall be enforced by the software library. Should a limitation be encountered, the software library under test shall have PFT III testing status revoked.

4.3 Validation and Submitting

NIST shall provide a *validation package* that will link the participant core software library to a *sample* PFT III test application. A script included in the validation package runs a series of tests and reporting routines to help ensure correct operation at NIST. Once the validation successfully completes on the participant's system, a file with logs, the participant's software libraries, and any provided configuration files will be created. After being signed and encrypted, **only** this file and a public key shall be submitted to NIST. Any software library submissions not generated by an unmodified copy of the latest version of NIST's PFT III validation package will be rejected. Any software library submissions that generate errors while running the validation package on NIST's hardware will be rejected. Validation packages that have recorded errors while running on the participant's system will be rejected. Any submissions of successful validation runs not created on Ubuntu Server 20.04.03 LTS will be rejected. Any submissions not signed and encrypted with the private key whose public key fingerprint is recorded on the participant's PFT III agreement will be rejected.

Participants may resubmit a new validation package immediately upon being notified of a validation rejection. NIST may impose a "cool down" period of several months for participants with excessive repeated rejections in order to most efficiently make use of test hardware.

4.3.1 Agreement

Before releasing PFT III analysis reports, NIST must receive a signed PFT III agreement. This agreement must be **both** physically mailed or faxed ***and*** e-mailed to NIST. E-mailed agreements *alone* cannot be accepted. Even if the information has not changed, a new agreement must be submitted for each PFT III analysis report NIST posts.

4.3.2 Communication

All communication to NIST shall be addressed to the PFT III e-mail alias `pft@nist.gov` and not a specific member of the PFT III team. This will help ensure your message is replied to in an efficient manner.

4.4 Speed

Timing tests will be run and reported on a fixed sample of the PFT III dataset using an **Intel Xeon Gold 6140** CPU prior to completing the entire test. Submissions that do not meet the timing requirements listed for each method in Section 3.3 will be rejected. Participants may resubmit a

faster submission immediately with a new version number. To avoid the appearance of PFT III as an algorithm speed-checking service, NIST may require that participants with excessive repeated failures exceedingly distant from published timing requirements wait several months before their next submission.

References

[1] Watson C, Wilson C, Marshall K, Indovina M, Snelick R (2005) Studies of One-to-One Fingerprint Matching with Vendor SDK Matchers. *NIST Interagency Report 7221* https://doi.org/10.6028/NIST.IR.7221

[2] Cheng SL, Fiumara G, Watson C (2011) PFTII report: Plain and Rolled Fingerprint Matching with Proprietary Templates. *NIST Interagency Report 7821* https://doi.org/10.6028/NIST.IR.7821

[3] American National Standard for Information Systems (2016) Information Technology: ANSI/NIST-ITL 1-2011 Update 2015 — Data Format for the Interchange of Fingerprint, Facial & Other Biometric Information. *NIST Special Publication 500-290e3* https://doi.org/10.6028/NIST.SP.500-290e3

Revision History

25 February 2021 Require Ubuntu 20.04.03 LTS (Section 4.1.6). Synchronize many clauses in Section 4 with other NIST evaluations.

25 February 2021 Require CentOS 8.2.2004 (Section 4.1.6).

30 October 2019 Initial release for API version 1.0.0.