Exemplar One-to-Many

Friction Ridge Image and Features Technology Evaluation

Test Plan and Application Programming Interface

Last Updated: 22 August 2024

Contents

Not Human Subjects Research

The National Institute of Standards and Technology Research Protections Office reviewed the protocol for this project and determined it is "not human subjects research" as defined in 15 CFR 27, the Common Rule for the Protection of Human Subjects.

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 National Institute of Standards and Technology (NIST) is conducting an ongoing, large-scale one-to-many (1:N) evaluation of exemplar fingerprint technologies entitled *Friction Ridge Image and Features Technology Evaluation Exemplar One-to-Many (FRIF TE E1N)*. In an effort to assess the state of the art in automated exemplar friction ridge feature extraction and identification, participants are asked to submit software libraries with algorithms capable of:

- extracting features from all types of fingerprint impression images,
- segmenting multi-finger impression images,
- building databases of fingerprint templates, and
- searching templates against databases to produce lists of similar candidates.

1.1 Background

In 2003, NIST conducted their first large-scale 1:N exemplar fingerprint Technology Evaluation (TE) under the name *Fingerprint Vendor Technology Evaluation (FpVTE)*. This first iteration required participants to submit both hardware and software to NIST for evaluation. After nearly a decade, FpVTE 2012 was announced, expanding on the 2003 iteration, but entirely in software libraries running on NIST-provided commodity server hardware. Several other friction ridge, face, and iris recognition evaluations have been run in an "ongoing" mode since FpVTE 2012. As of 2024, NIST will re-launch FpVTE as *FRIF TE E1N* (*FRIF E1N* or *E1N*, for short), completing the transition of all NIST friction ridge TEs to an on-demand service for the biometric community.

1.2 What is FRIF?

NIST conducts several friction ridge TEs, each with their own set of unique application programming interfaces (APIs) and associated requirements. FRIF is an attempt to unify **all** of the APIs and requirements from all of the NIST friction ridge TEs. NIST realizes that there is often outside pressure to participate in their TEs, and having several disparate TEs interfaces to learn consumes value time that could otherwise be spent on other core work tasks. The hope is that unification makes it easier to participate in one or more NIST friction ridge TEs without needing to reimplement shared products and tasks.

In the friction ridge modality, in addition to E1N, NIST also conducts TEs of one-to-one (1:1) verification, mark 1:N identification, and multi-finger segmentation.

1.3 What is FRIF TE E1N?

FRIF TE E1N studies the computational performance and accuracy of automated open set exemplar identification algorithms and their associated feature extraction algorithms. These algorithms are typically components of an Automated Biometric Identification System (ABIS). In FRIF TE E1N, software libraries under test assemble a reference database of templates derived from friction ridge images and/or features and search that database with one or more exemplar friction ridge image and/or feature probes. NIST reports on the computational performance and accuracy of these algorithms in public analysis reports.

2 Evaluation Imagery

2.1 Source

Imagery used in FRIF TE E1N comes 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 Region

The primary friction ridge region evaluated in FRIF TE E1N is the distal phalanx. Images often include other regions of the hand in addition to the distal phalanges, up to and including the entirety of the hand (i.e., a *full palm* friction ridge generalized position).

In addition to the primary distal phalanx focus, implementations may be asked to perform palm searches using templates derived from images that may not contain distal phalanges (i.e., a *lower palm* friction ridge generalized position).

2.3 Quality

Due to the operational nature of the source of the imagery, the quality of the images varies dramatically between datasets and samples within the datasets. NIST may choose to disclose NIST Fingerprint Image Quality 2 (NFIQ 2) quality scores in the future for types of images supported by the NFIQ 2 algorithm.

Participants are encouraged to use the metadata provided with the imagery (Section 2.4) to assess quality in their own way and store this value in their template. This collection of quality values may help advise future directions in friction ridge image quality, especially when it comes to palm and contactless imagery. See Section 4.4 for more details.

2.4 Metadata

Participants may be provided with known metadata about each image during template creation. Depending on the scenario being tested (Section 3), some, all, or none of this information will be provided. Possible metadata is detailed in Section 3.2.2.

2.5 Access

Most FRIF TE E1N evaluation datasets are protected under the Privacy Act (5 U.S.C. §552a) and are treated as controlled unclassified information (CUI) as defined in Executive Order 13556. FRIF TE E1N participants will not have access to such FRIF TE E1N evaluation data, before, during, or after the evaluation. NIST will provide *similar* image data from research datasets that can be used to prepare software libraries for FRIF TE E1N.

Note that participants will additionally not have access to any data generated by their software libraries at NIST, regardless of the source of the imagery used to derive such data. Alterations to this policy are at the discretion of the FRIF liaison.

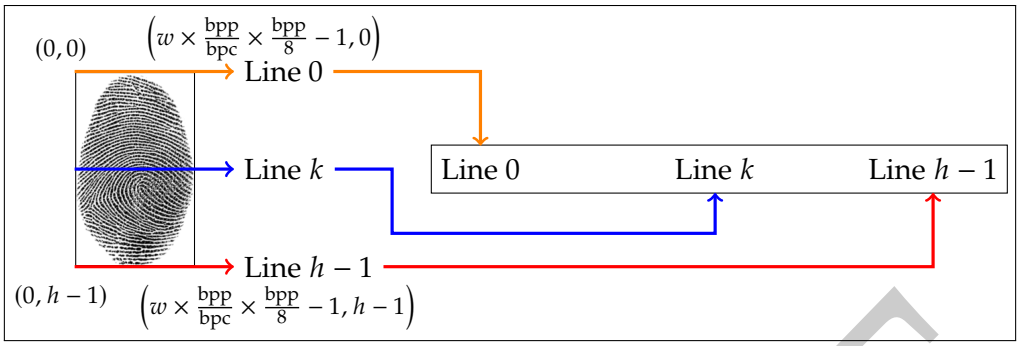


Figure 1: Order of image scanlines in data passed to FRIF TE E1N implementations. In FRIF TE E1N, bpc = bpp (i.e., always a single color component—grayscale). Fingerprint image sourced from NIST Special Database 302 [1, 2].

2.6 Format

The software library under test must be capable of processing friction ridge images sent as buffers of uncompressed raw pixels. Images are all encoded with a single color component (i.e., grayscale). This component may be comprised of either 8 bits or 16 bits per pixel. Sample data in both bit depths is provided during software validation.

Images shall follow the scan sequence described in ISO/IEC 39794-4:2019, §6.2, 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.

Bit depths shall follow the description in ISO/IEC 39794-4:2019, §6.3.4. The minimum value that will be assigned to a "black" pixel (e.g., ■) is zero (0) in all color components. The maximum value that will be assigned to a "white" pixel (e.g., □) is $\left(2^{\text{bpc}} - 1\right)$ in all color components, where *bpc* is bits per component (either 8 or 16).

Image width and height are measured in pixels and will be supplied to the software library under test as supplemental information. Pixels are stored left to right, top to bottom. The number of bytes in an image is equal to $(\text{width} \times \text{height} \times \frac{\text{bpp}}{\text{bpc}} \times \frac{\text{bpp}}{8})$, where *bpp* is bits per pixel. Again, in FRIF TE E1N, images are always in grayscale (i.e., a single color component, or bpc = bpp).

3 Scenarios and Variables

Analysis in FRIF TE E1N will be the result of evaluating search scenarios (described in Section 3.1) with combinations of search variables (described in Section 3.2). Together, these differentiations provide operationally-relevant situations for examination.

3.1 Probe Template Scenarios

The API for FRIF TE E1N is flexible enough to support multiple probe template generation scenarios. Each scenario represents a possible realistic law enforcement search scenario.

3.1.1 Single Region

A single image and/or feature set describing a single friction ridge region is provided to the template creation method. A probe template is produced from this method and is then provided to the search method.

3.1.2 Multiple Single Regions

Like Section 3.1.1, but more than one individual image and/or feature set is provided. The samples may be of the same or different regions.

3.1.3 Simultaneous Region

Like Section 3.1.2, but to obtain multiple regions, software libraries under test need to first segment from a single multi-region sample.

3.1.4 Multiple Simultaneous Regions

Like Section 3.1.3, but there may be more than one sample to be segmented, such as the common "identification flat (4-4-2)" capture, consisting of a right slap, left slap, and thumb slap.

3.1.5 Multiple Simultaneous and/or Single Regions

A combination of Section 3.1.2 and Section 3.1.4.

3.1.6 Palm

Searching one or more palm impressions, with (i.e., *upper palm*, *full palm*) or without (i.e., *lower palm*) distal phalanges included in the sample.

3.2 Variables

Searches can range in difficulty by changing several variables for any of the search scenarios described in Section 3.1.

3.2.1 Probe Friction Ridge Generalized Position

The name of the friction ridge generalized position (i.e., right index) is typically provided when the probe template is created, since this information is typically known in operational scenarios. However, software libraries under test shall be capable of searching based on the friction ridge generalized position being identified as `UnknownFinger` or `UnknownPalm`. Search results will be reported separately when searched with an unknown friction ridge generalized position.

3.2.1.1 UnknownFrictionRidge

Software libraries under test need not be capable of searching probes with the friction ridge position `UnknownFrictionRidge`, since this is more relevant for fingermark searches (e.g., such as those evaluated in ELFT). However, for robustness and potential research purposes, it is appreciated if this scenario produced as meaningful as possible results using capabilities already present in the software library under test, such as treating the probe as both `UnknownFinger` and `UnknownPalm`. It is not expected for software libraries under test to include a separate fingermark algorithm.

3.2.2 Extended Feature Set

ANSI/NIST ITL 1-2011 added support for Extended Feature Set (EFS), a data block which, "defines the content, format, and units of measurement for the definition and/or exchange of friction ridge feature information" [3]. In the FRIF TE E1N API, a subset of EFS information is available to software libraries under test to assist in feature extraction and searching.

Some, all, or none of this data will be provided to the software library under test during feature extraction. The source (e.g., Certified Latent Print Examiner (CLPE), third-party algorithm) of the specified features will not be provided.

Please note that all EFS information is converted from and provided via the API in pixels at the provided resolution with the same image origin as a convenience, *not* units of 10 µm, as defined by the standard. As such, some rounding error may occur.

3.2.3 Features Only

Some or all of the data described in Section 3.2.2 will be provided to the software library under test during feature extraction. **No image will be provided.** The software library under test should encode theses features into their template format for later searching.

3.2.4 Impression Types

The impression type of both probes and references may vary. Examples include, but are not limited to, searching plain impressions against rolled impressions, and searching contactless impressions against plain and rolled impressions.

3.2.5 Finger Positions

Combinations of different finger positions mimicking different operational scenarios may be employed. Examples include, but are not limited to, searching only fingers from the left hand, and searching only index fingers.

3.2.6 Database Size

The number of references in a database has an affect on open set identification. NIST will evaluate performance with reference databases containing various numbers of nonmated references.

3.2.7 Database Quality

The quality of references in the database plays a large part in the quality of candidates returned in a candidate list. Many aspects affect quality, such as the sensor type and the skill of the sensor operator. Where possible, NIST will vary the quality of the mated reference in the database.

4 Application Programming Interface Highlights

The FRIF TE E1N API is only discussed briefly in this test plan. Thorough documentation is available directly in the C++ header file, and is additionally formatted both for the web and for print.

The FRIF TE E1N API is written in C++ and makes use of C++20 features. The core library does not need to adopt this standard, but will need to be compiled with the appropriate flags to support linkage with the FRIF TE E1N test application. All code should be built with the same compiler to ensure compatibility with the FRIF TE E1N test application and to prevent difficult to debug link and runtime errors.

A stub implementation of the API that is provided.

4.1 FRIF Namespace

All API code for FRIF TE E1N exists within the `FRIF` namespace. The `FRIF` namespace itself contains several namespaces containing code shared by all FRIF evaluations. Classes and methods core to Exemplar One-to-Many are found in the namespace `FRIF::Evaluations::Exemplar1N`. To participate in FRIF TE E1N, implementations of the abstract classes `Exemplar1N::ExtractionInterface` and `Exemplar1N::SearchInterface` must be present, as well as implementations of all defined `static` methods within `Exemplar1N`.

4.1.1 ReturnStatus

The `ReturnStatus` struct is used in most non-trivial API methods to return information from the software library under test about the status of performing an operation. If an operation is successful, the default-constructed `ReturnStatus` is sufficient to indicate success (e.g., `return {};` or `return (ReturnStatus());`. In failure conditions, software libraries under test shall set the `result` parameter of `ReturnStatus` to `Result::Failure`. For debugging purposes, it is helpful to include text matching the regular expression `[[:graph:] ]*` regarding why the failure occurred in `ReturnStatus`'s `message` parameter. If it adds *meaningful* information, `message` can also be populated when successful, but is otherwise discouraged.

The API attempts to differentiate between the result of operations in terms of technical functionality and usability. For instance, an implementation may functionally succeed to call `createTemplate()`, but the data provided is of such low quality, that the software library under test chooses not to produce a template. This would result in a `ReturnStatus` with `Result::Success`, but an omitted `CreateTemplateResult`. Similarly, an implementation may successfully search their database, but not find any candidate worth returning, which would be communicated in a similar manner.

While this doesn't change the resulting analysis (i.e., a miss is still miss), it does help differentiate between the root cause of errors. For example, in `searchSubject()`, an empty `CandidateList` could be produced by searching either an invalid template or a template from a low-quality sample. The former would elicit a response of `Result::Failure`, since the operation could not be completed, and the latter, a response of `Result::Success`, since the operation completed, but no useful candidate could be found due to the input.

4.1.2 Features

The `EFS::Features` struct encapsulates metadata about an image of one or more hand regions. Depending on the scenario (Section 3), this data may be provided only in part or not at all. In other cases, some or all of the data simply might not be known. Software libraries under test are expected to process all `Image`s provided, regardless of what metadata is provided.

This struct is also used to expose data encoded in templates, as described in Section 4.4.

4.2 Extracting Features

The FRIF TE E1N API defines the abstract class `ExtractionInterface`, with several pure virtual functions to support extracting features and creating templates from a wide variety of friction ridge images. Participants shall publicly inherit `ExtractionInterface` to implement all feature extraction methods, including `static` methods.

4.2.1 Compatibility

`ExtractionInterface::getCompatibility()` encodes knowledge about what versions of previously-generated artifacts can be reused with this version. It is important to be accurate, as NIST, in an effort to efficiently run evaluations, will almost always opt to reuse artifacts where possible.

This method also lets the FRIF TE E1N test application know at runtime if there is a meaningful implementation of the template introspection methods.

4.2.2 Identification

`ExtractionInterface::getProductIdentifier()` provides a means for organizations to provide marketing information to readers of FRIF TE E1N analysis reports. If desired, populate the `Product-Identifier` struct with marketing and Common Biometric Exchange Formats Framework (CBEFF) information about the feature extraction algorithms embedded within the software library under test. This information will be printed verbatim in FRIF TE E1N analysis reports.

4.2.3 Create Template

`ExtractionInterface::createTemplate()` is the workhorse of the `ExtractionInterface`. Depending on the scenario (Section 3), the software library under test will be provided **zero** or more friction ridge images and/or feature sets and be expected to produce a single buffer of data in the form of a template usable by the software library under test.

Notes

- This method supports differentiation between `Probe` and `Reference` templates.
- The software library under test may be provided more than one image per friction ridge position, especially when creating `Reference` templates.
- Internally, the software library under test may store more than one template, but data must be returned as a single buffer.
- If a single `Image` contains more than one friction ridge position (e.g., an upper palm capture), it is the responsibility of the software library under test to segment into multiple regions

for feature extraction, if desired. Participants should consider participating in NIST's Slap Fingerprint Segmentation III evaluation for segmentation practice and detailed analysis.

- Depending on the scenario, some, all, or none of the `EFS` metadata will be provided. The software library under test is still expected to process the image regardless of what, if any, `EFS` data is received.
- Feature-only searches (Section 3.2.3) are facilitated by not providing an `Image`. If no `Image` is provided, there is guaranteed to be some `EFS` data provided. Implementations should encode this information into a template usable by the `SearchInterface`.
  - For feature-only searches, `Feature` will only ever refer to a single finger or palm region. That is, if the features were based on a multi-finger image, only `Feature` for a single finger of that image would be provided per instance.
- The source of `Feature` data is not specified. It may be from your software library under test, a different software library under test, or a CLPE.

Implementations should target consuming no more than ≈3 GB per process when extracting features. Implementations using significant amounts of RAM may be disqualified at NIST's discretion.

4.2.4 Create Reference Database

Once all `Reference` templates have been created, the FRIF TE E1N test application will provide them en masse to `ExtractionInterface::createReferenceDatabase()`. This method should ingest these templates, do any necessary processing, and write some sort of structure to disk at the location provided by the `databaseDirectory` parameter. This location will be provided to the `SearchInterface` later for searching. When creating the reference database, the number of templates provided is exactly the number of identities represented (i.e., consolidated) and no de-duplication efforts are necessary.

While the structure of the database is not defined by NIST, NIST will enforce certain file count and size limitations to ensure efficient data center operations.

4.2.4.1 Database Size

The path pointed to by `databaseDirectory` will reside on a local disk. Depending on the number of bytes written, it is likely that the amount of RAM available during `searchSubject()` could be significantly less than the size of the reference database. Exact values will be provided at runtime. Software libraries will have an opportunity before searching (via `load()`) to load data into RAM. To that end, NIST encourages software libraries to store data in a structure that facilitates being *partially* cached in RAM (NIST anticipates the amount of RAM available for `load()` to be ≈300 GB).

4.3 Searching the Database

The FRIF TE E1N API defines the abstract class `SearchInterface`, with several pure virtual functions to support searching and modifying the reference database created with `ExtractionInterface`. Participants shall publicly inherit `SearchInterface` to implement all feature extraction methods.

4.3.1 Identification

As in template creation (Section 4.2.2), the FRIF TE E1N API provides a means for providing marketing information about the search algorithm to readers of NIST reports. If desired, return this information in `SearchInterface::getProductIdentifier()`. This information will be printed verbatim in FRIF TE E1N analysis reports.

4.3.2 Compatibility

Like in `ExtractionInterface` (Section 4.2.1), `SearchInterface::getCompatibility()` encodes knowledge of compatibility with previously-generated artifacts and runtime support for optional features.

4.3.3 Load

After construction and prior to the first search, software libraries under test have the opportunity to `load()` up to `maxSize` bytes (provided at runtime) into RAM. NIST anticipates the amount of RAM available for `load()` to be ≈300 GB. This data will be shared by multiple search process through the FRIF TE E1N test application's use of `fork()`. Information in excess of `maxBytes` shall be read from disk.

Database information on disk and loaded into RAM shall remain read-only during the entire `SearchInterface` lifecycle.

4.3.4 Search

`Probe` templates created in the `ExtractionInterface` are searched through the reference database in `SearchInterface::searchSubject()` and `SearchInterface::searchSubjectPosition()`. Software libraries under test will be provided a single template created by their template generation implementation and are expected to return a list of potential candidates with accompanying similarity scores.

The details requested about a candidate in a candidate list are what differentiates the search methods `SearchInterface::searchSubject()` and `SearchInterface::searchSubjectPosition()`. For `SearchInterface::searchSubject()`, only the subject identifier should be returned. This search interface will be called in scenarios where there are several friction ridge positions within the probe template, and the operational scenario only dictates that you find if the represented subject is in the enrollment database. `SearchInterface::searchSubjectPosition()` will be used when it's important to return both the subject identifier *and* the most similar friction ridge generalized position. This is more likely to be called when searching with a friction ridge generalized position of `UnknownFinger` or `UnknownPalm`. Depending on the scenario, omitting, misidentifying, or overgeneralizing a friction ridge position may result in a miss.

In addition to a candidate list, these methods also ask for a `bool` as to whether or not the software library under test believes the candidate list contains the true mate. This may be based on an internal perceived similarity score threshold or any other heuristic.

Implementations should target consuming no more than ≈1 GB per process (in excess of the shared database, described in Section 4.3.3) when searching. Implementations using significant amounts of RAM may be disqualified at NIST's discretion.

4.4 Research Data

The FRIF TE E1N API provides two facilities to gain insight into the decisions made by the software library under test. Currently, providing these insights is **optional**, but may help in debugging errors during the FRIF TE E1N evaluation, provide insights into miss analysis, and aid future NIST research.

4.4.1 Templates

There is no template format requirement for the data returned from `ExtractionInterface::createTemplate()`. `ExtractionInterface::extractTemplateData()` allows for insight into what kind of data is included in the otherwise opaque template. This method should return one `TemplateData` per friction ridge position per image. If the `TemplateData` is derived from a single finger in a multi-region image, the `Coordinates` of a convex polygon enclosing the region of interest of the finger in question should be recorded in the `roi` parameter. An example of returning `TemplateData` from an identification flat image is shown in Figure 2.

4.4.2 Correspondence

Many search algorithms operate by mimicking the actions of CLPEs—corresponding groupings of minutia found in a probe image with the same groupings found in an exemplar image. This information can be exposed via `SearchInterface::extractCorrespondence()`. This method returns a list of corresponding minutia for each candidate in a candidate list for a given search. An example of returning `Correspondence` is shown in Figure 3.

Each Candidate is included in a separat

Important

`Minutia` returned in `Correspondence` shall come from the set returned in `ExtractionInterface::extractTemplateData()`.

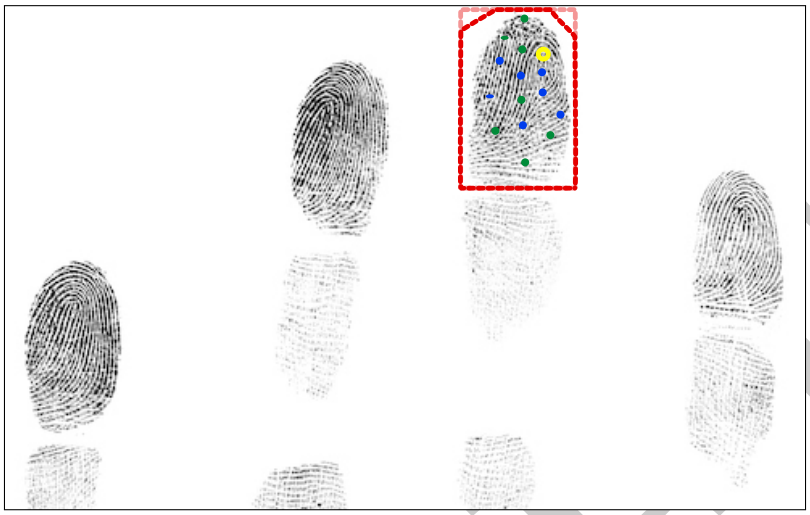
4.5 Fundamentals

4.5.1 Object Construction

Each class defines a static "factory" method named `getImplementation()`. These methods provide the software library under test necessary filesystem paths needed to load provided configurations and their reference database, as applicable. The software library under test is responsible for implementing these methods that return an instance of the child class that implements the appropriate FRIF TE E1N interface. The FRIF TE E1N test application will exclusively use the returned object for calling API methods. When the FRIF TE E1N test application `forks`, calls to `getImplementation()` will occur *before* the `fork`, such that large read-only memory buffers are shared between processes, relying on Linux copy-on-write pages.

4.5.2 Errors

Each non-trivial API method provides a way to return a `ReturnStatus` where information about failures and errors can be expressed to the FRIF TE E1N test application (Section 4.1.1). It's not always possible to safely jump out of code in certain error conditions (e.g., a memory allocation



```
std::vector<TemplateData> tds{};
tds.reserve(4);

TemplateData tdLM{};
tdLM.imageIdentifier = 6; // Copy, provided in createTemplate()

// Parse your template to get this info. It is hardcoded here for clarity.
tdLM.features.frgp = FrictionRidgeGeneralizedPosition::LeftMiddle;
tdLM.features.roi = std::vector<Coordinate>{{285, 16}, {306, 3}, {341, 3},{356, 20}, {356, 114}, {285, 114}};
tdLM.features.orientation = 8;
tdLM.features.valueAssessment = ValueAssessment::Value;
tdLM.features.pct = PatternClassification::LeftLoop;
tdLM.features.plr = false;
tdLM.features.trv = false;

// Locations are relative to the rectangle bounding td.features.roi (lighter red above).
tdLM.features.cores = std::vector<Coordinate>{{48, 24}};
tdLM.features.minutia = std::vector<Minutia>{};
tdLM.features.minutia->emplace_back({{12, 26}, -14.5, MinutiaType::RidgeEnding});
tdLM.features.minutia->emplace_back({{{57, 34}, 15.0, MinutiaType::Bifurcation});
tdLM.features.minutia->emplace_back({{{47, 12}, 48.2, MinutiaType::Bifurcation});
// ...

tdLM.imageQuality = 83;
tds.push_back(tdLM);

// ... Repeat for other fingers. All have same imageIdentifier ...

return (tds);
```

Figure 2: Example of returning `TemplateData` from `ExtractionInterface::extractTemplateData()`. In this example, a left identification flat image (`frgp = 14`) was provided to `ExtractionInterface::createTemplate()`. The graphic visualizes what might have been recorded for the left middle finger. The red dashed line shows a six-sided polygon region of interest, with the lighter red rectangle representing the bounding rectangle formed by the region of interest. All `Coordinate` are relative to the bounding rectangle. Green and blue dots indicate minutia and the yellow circle indicates a core. All annotations are simulated. The code example shows *only* what would be needed to return information for the left middle finger. Fingerprint image sourced from NIST Special Database 302 [1, 2].

```
CandidateListCorrespondence allCorrespondence{};

for ([[maybe_unused]] const auto &[candidate, similarity] : searchResult.candidateList) {
        // Retrieve reference template for this Candidate. For simplicity, we'll assume this private method returns a
        // FRIF::IO::TemplateData.
        const TemplateData referenceTemplate = this->retrieve(candidate);

        // Determine which Minutia align between the probe and this reference
        const std::vector<std::pair<Minutia, Minutia>> minutiae = this->determineCorrespondingPairs(
            probeTemplate, referenceTemplate);

        // Save away all those Minutia
        Correspondence c{};
        c.correspondence.reserve(minutiae.size());
        for (const std::pair<Minutia, Minutia> &m : minutiae) {
                CorrespondenceRelationship cr{};
                cr.probeIdentifier = probeIdentifier // Copy, provided in createTemplate()
                cr.probeImageIdentifier = 6; // Copy, provided in createTemplate()
                cr.referenceIdentifier = candidate.candidateIdentifier
                cr.referenceImageIdentifier = candidate.inputIdentifier;

                // Corresponding Minutia MUST align from TemplateData. The commented values here are for clarity and are from
                // the previous extractTemplateData() examples.
                cr.probeMinutia = m.first;
                cr.referenceMinutia = m.second; // Iteration 1: {12, 26}; Iteration 2: {57, 34}; Iteration 3: {47, 12}; . . .

                c.push_back(cr);
        }
        c.complex = false;

        allCorrespondence[candidate.candidateIdentifier] = c;
}

return (allCorrespondence);
```

Figure 3: Example of returning `Correspondence` within a `CandidateListCorrespondence` from `SearchInterface::extractCorrespondence()`. For each candidate in `searchResult.candidateList`, a `Correspondence` is created and its collection of `CorrespondenceRelationship` is reserved. Then, corresponding `Minutia` for each pair of `probeTemplate` and reference template (retrieved from the loaded reference database) are determined and iteratively set in `CorrespondenceRelationship` objects. The `Minutia` set here *must* match the `Minutia` returned in `TemplateData` from `ExtractionInterface::extractTemplateData()`. A complexity decision is made for each candidate and then the entire collection is returned.

failure). In this case, it may be appropriate to throw an exception. The FRIF TE E1N test application will catch `std::exception` from all API methods. To assist in debugging these failure scenarios, please be sure to throw exceptions inherited from `std::exception` and populate the `what_arg` parameter with a description of the problem. If your existing custom exception type does not inherit `std::exception`, consider catching it and re-throwing a `std::runtime_error` with appropriate information. Like the `message` member of `ReturnStatus`, all exception `what_arg` shall match the regular expression `[[:graph:] ]*`.

4.5.3 Multiprocessing

All API methods shall be single-threaded. The reason is that the NIST test driver operates as a Message Passing Interface (MPI) job to multiple nodes, forking on each node to run many tasks in parallel. See Section 5.1.1 for more details.

`ExtractionInterface::createReferenceDatabase()` and `SearchInterface::load()` are exceptions to this rule. NIST will run these methods on a single node in a single process. For these methods, NIST *expects* the software library under test to make use of threading (if necessary) to complete the creation and loading of the reference database as fast as possible, within the required time limits. Software libraries under test should query the system for a hint of the number of supported concurrent threads (e.g., `std::thread::hardware_concurrency()`) and make reasonable use of them.

4.5.4 Speed

All API methods have speed thresholds that must be achieved before NIST will accept a software library for evaluation. Technical details are described in Section 5.4. Required speeds are as listed in Table 1. Note that multi-finger images are considered multiple samples for the purposes of timing. For example, a four-finger slap image would be considered four samples and a full palm image would be considered sixteen samples.

For template creation and searching, these speeds will be enforced as the mean observed duration on a fixed subset of data from the evaluation datasets (Section 2). For all other methods, these values will be hard maximums. If these times are reached before the method has returned, the NIST test driver will forcibly terminate and NIST will request a faster version from the participant.

	API Method	Metric	Requirement
	`getLibraryIdentifer()`	Max	250 ms
Extraction	`getImplementation()`	Max	5 s
	`getCompatibility()`	Max	250 ms
	`getProductIdentifier()`	Max	250 ms
	`createTemplate()`: Exemplar	Sample Mean	500 ms × M per sample
	`createTemplate()`: Features (no image)	Sample Mean	250 ms per feature set
	`extractTemplateData()`	Max	500 ms
	`createReferenceDatabase()`	Max	5 ms per identifier
Search	`getImplementation()`	Max	5 s
	`getCompatibility()`	Max	250 ms
	`getProductIdentifier()`	Max	250 ms
	`load()`	Max	1 ms per identifier
	`search()`	Sample Mean	40 µs per identifier
	`extractCorrespondence()`	Max	500 ms

Table 1: API runtime requirements. *API Method* indicates the FRIF TE E1N API method for the current timing requirement. *Metric* indicates how the duration will be enforced—either a hard maximum or as a mean value measured over a fixed sample. *Requirement* indicates the value that must be achieved. For some methods, the value is variable based on the number of identifiers in the reference database. In others, a multiplier M is in effect. See Table 2 for a list of these values.

Image Contents	M Value
Single Finger	1
Two-Finger Simultaneous Capture	2
Four-Finger Simultaneous Capture	4
Upper, Lower, or Writer's Palm	8
Joint Regions	8
Full Palm	16
All Other Regions	8

Table 2: M value requirements for feature extraction. *Image Contents* is what friction ridge structure is depicted within an image. *M Value* is what is used as a multipler in the calculation of time permitted for feature extraction. See Table 1 for other parts of the time allocation equation.

5 Software and Documentation

5.1 Software Libraries and Platform Requirements

The methods specified in Section 4 shall be implemented exactly as defined in a software library. The header file used by the FRIF TE E1N test application is provided on the FRIF TE E1N website in the FRIF TE E1N validation package (Section 5.3).

5.1.1 Restrictions

5.1.1.1 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 5.1.6. 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 5.3). Any required libraries that are not standard to Ubuntu Server 24.04 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 (and *only* this directory) to the runtime library search path list (`RUNPATH`, see Section 5.1.2).

5.1.1.2 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 FRIF TE E1N application indicates that NIST will only accept one submission every 120 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 5.3). Filenames and checksums of all provided files will be reported in FRIF TE E1N analysis reports. The path to such files will be provided as a parameter to `getImplementation()` (Section 4.5.1). No participant-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 5.1.6) to correct the defect.

5.1.1.3 Multiprocessing

With two exceptions (Section 4.5.3), the software library shall not make use of threading, forking, OpenMP, `std::execution::par`, or any other multiprocessing techniques. The FRIF TE E1N test

application operates as an 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 FRIF TE E1N participants.

The software library under test shall not acknowledge the existence of other processes running on the test hardware, such as through `semaphore`s or `pipe`s, nor attempt to communicate with any other process.

5.1.1.4 Deterministic Operation

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

5.1.1.5 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.

The only exception to this rule is when interacting with the reference database. In this case, the software library under test shall only read and write to areas at or below the filesystem path provided in `getImplementation()`.

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

5.1.2 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 FRIF TE E1N test application will explicitly link is the core library. The FRIF TE E1N 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 FRIF TE E1N analysis reports.

5.1.3 `libfrif.so`

Core libraries will need to depend on the NIST-provided `libfrif.so` for implementations of FRIF-wide API methods. Participants shall not alter the provided header file for `libfrif.so`. NIST will build and supply `libfrif.so`, and so this library shall **not** be included in validation submissions (Section 5.3).

5.1.4 `libfrif_e1n.so`

A second NIST-provided library, `libfrif_e1n.so`, provides implementations for E1N-specific methods.

Again, core libraries will need to depend on this NIST-provided library. Participants shall not alter the provided header file for `libfrif_e1n.so`. NIST will build and supply `libfrif_e1n.so`, and so this library shall **not** be included in validation submissions (Section 5.3).

5.1.5 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 5.3) on Intel CPUs, including, but not limited to, **Intel Xeon E5-2680**, **Intel Xeon E5-4650**, **Intel Xeon Gold 6140**, and **Intel Xeon Gold 6254**. 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 5.4.

5.1.6 Naming

The core software library submitted for FRIF TE E1N shall be named in a predefined format. The first part of the software library's name shall be `libfrif_e1n_`. 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, 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 FRIF TE E1N test driver is the core library, as demonstrated in Sections 5.1.2 and 5.1.7.

The version number shall match the uppercase hexadecimal version number with leading `0`s, as returned by `Evaluations::Exemplar1N::getLibraryIdentifier()`. 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 4.2.2 for this purpose.

Example
Initech submits a software library named `libfrif_e1n_initech_101C.so` with build 4 124 of their algorithm. This library returns `{0x101C, "initech"}` from `getLibraryIdentifier()`. NIST determines that Initech's `searchSubject()` method is too slow and rejects the library. Initech submits build 4 125 to correct the defect in 4 124. Initech updates `getLibraryIdentifier()` in their implementation to return `{0x101D, "initech"}` and renames their library to `libfrif_e1n_initech_101D.so`. In FRIF TE E1N analysis reports, NIST refers to Initech's library as `initech+101D`.

```
mpicxx -o frif_e1n frif_e1n.cpp -Llib -Wl,--enable-new-dtags -Wl,-rpath,lib \
       -lfrif -lfrif_e1n -lfrif_e1n_initech_101D -std=c++20
```

Figure 4: Example compilation and link command for the FRIF TE E1N test application.

5.1.7 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++20 language test driver application using the compiler `g++` (version `Ubuntu 13.2.0-23ubuntu4`, via `mpicxx`) under **Ubuntu Server 24.04 LTS**, as seen in Figure 4.

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 24.04 LTS. Participants are **strongly encouraged** to verify library-level compatibility with `g++` on Ubuntu Server 24.04 LTS **prior to** submitting their software to NIST to avoid unexpected problems.

5.2 Usage

5.2.1 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 FRIF TE E1N testing status revoked.

5.3 Validation and Submitting

NIST shall provide a *validation package* that will link the participant core software library to a *sample* FRIF TE E1N 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 FRIF TE E1N 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 24.04 LTS will be rejected. Any submissions not signed and encrypted

with the private key whose public key fingerprint is recorded on the participant's FRIF TE E1N 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.

5.3.1 Agreement

Before releasing FRIF TE E1N analysis reports, NIST must receive a signed FRIF TE E1N agreement. Even if the information has not changed, a new agreement must be submitted for each FRIF TE E1N analysis report NIST posts. This agreement may be e-mailed.

5.3.2 Communication

All communication to NIST shall be addressed to the FRIF TE E1N e-mail alias `frif@nist.gov` and not a specific member of the FRIF team. This will help ensure your message is replied to in an efficient manner.

5.4 Speed

Timing tests will be run and reported. Speed requirements are listed in Table 1. For those methods where *Metric* is *Sample Mean*, the test will be performed using a fixed sample of the FRIF TE E1N dataset (Section 2) on an **Intel Xeon Gold 6254** CPU prior to completing the entire test. Submissions that do not meet the timing requirements listed for each method in Table 1 will be rejected. A table of timing requirements can be seen in Section 4.5.4.

Speed tests of `ExtractionInterface::createReferenceDatabase()` will also be performed on an **Intel Xeon Gold 6254** CPU. For expediency, NIST may choose to allow some discretion in the runtime of this method. For example, if the maximum runtime is 10 h but the actual runtime was 10.1 h and there was demand for compute resources in our data center, it *may* be prudent to continue the evaluation, since a resubmission in this example would also require the regeneration of millions of templates.

Due to the nature of the FRIF TE E1N API, timing failures may not be seen by NIST until several days after submission. Participants may resubmit a new validation package immediately upon being notified of a timing 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.

References

[1] Fiumara G, et al. (2018) National Institute of Standards and Technology Special Database 302: Nail to Nail Fingerprint Challenge. National Institute of Standards and Technology, Technical Note 2007. https://doi.org/10.6028/NIST.TN.2007

[2] Fiumara G, et al. NIST Special Database 302 Nail to Nail (N2N) Fingerprint Challenge, National Institute of Standards and Technology, Zip Archive. https://doi.org/10.18434/M31943

[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

22 August 2024

- Updated search API in Section 4.3.4 to separate `search()` into `searchSubject()` and `searchSubjectPosition()`.
- Added clarification to search scenarios, explicitly adding Section 3.1.5.
- Clarified friction ridge generalized position in probes by adding Section 3.2.1.

21 June 2024

- Initial draft for public comment.