Evaluation of Latent Friction Ridge Technology

Test Plan and Application Programming Interface

Last Updated: 07 May 2020

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 reviving Evaluation of Latent Fingerprint Technology as *Evaluation of Latent Friction Ridge Technology (ELFT)*, a biometric technology evaluation that aims to study the state of the art in automated latent friction ridge identification algorithms. NIST's interest in latent friction ridge identification goes back to 2006, when they convened their first latent fingerprint testing workshop to learn about the needs and wants of automated latent fingerprint system stakeholders [1]. A series of tests were conducted in the subsequent decade, but has languished since 2012. In 2020, ELFT joins the ranks of *ongoing* NIST biometric technology evaluations.

1.1 Background

Latent friction ridge identification is a largely human-intensive process. Crime Scene Investigators (CSIs) and Certified Latent Print Examiners (CLPEs) take care to capture or develop friction ridge marks deposited on a surface by chance due to contaminants, skin secretions, or other matrix. CLPEs carefully annotate these marks for distinctive features, compare observed groupings of features with an exemplar capture, and conclude whether or not there is sufficient information to assert that the two marks originated from the same source. The exemplar chosen for comparison could come from an individual on trial for committing a crime or may be a top-ranked candidate returned from an Automated Biometric Identification System (ABIS). ELFT focuses on the evaluation of algorithms that create such ABIS candidate lists.

1.2 What is ELFT?

ELFT studies the computational performance and accuracy of automated open set latent identification algorithms and their associated feature extraction algorithms. These algorithms are typically components of an ABIS. In ELFT, software libraries under test assemble a reference database of images and search that database with one or more latent image probes. NIST reports on the computational performance and accuracy of these algorithms in public analysis reports.

1.3 What's New Since ELFT-EFS

The concept of operations for Evaluation of Latent Fingerprint Technology—Extended Feature Set (ELFT-EFS) #2 [2] was last updated in 2010. In short, **much** has changed in the intervening decade. Developers of submissions for ELFT should read this document in its entirety before beginning work on their software libraries. NIST encourage developers and other stakeholders to send feedback, especially in ways that the ELFT application programming interface (API) dramatically differs from the current operations of their commercially-deployed algorithms.

2 Evaluation Imagery

2.1 Source

Imagery used in ELFT 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

Images used in ELFT encompass all regions of the hand. This includes images of all parts of the palm, all phalanges, and all interphalangeal joints. Although it will be the most represented, neither probe nor reference imagery used in ELFT is limited to the distal phalanx.

2.3 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 latent or exemplar friction ridge image quality for all combinations of resolution, bit depth, and sensor type represented in ELFT, 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 2.4) 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 latent 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 3.2.3.

2.5 Access

Most ELFT 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. ELFT participants will not have access to such ELFT evaluation data, before, during, or after the evaluation. NIST will provide *similar* image data from publicly-available research datasets that can be used to prepare software libraries for ELFT.

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.

2.6 Format

The software library under test must be capable of processing friction ridge images sent as buffers of uncompressed raw pixels. Images may be in red-green-blue color (RGB) triplets or single-component grayscale. A single color component may be comprised of either 8 bits or 16 bits.

Images shall follow the scan sequence as defined by ISO/IEC 19794-4:2005, §6.2, visualized in Figure 1. The origin is the upper-left corner of the image. The X-coordinate (horizontal) position

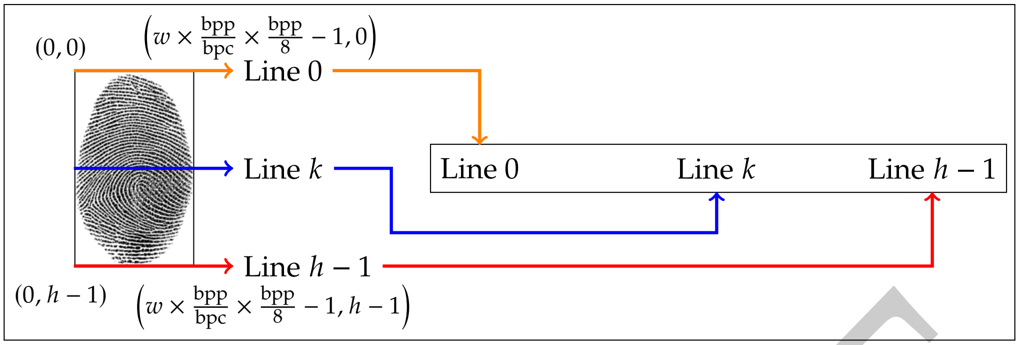

$(0, 0)$
$\left(w \times \frac{\text{bpp}}{\text{bpc}} \times \frac{\text{bpp}}{8} - 1, 0\right)$
Line 0
Line k
Line $h - 1$
$(0, h - 1)$
$\left(w \times \frac{\text{bpp}}{\text{bpc}} \times \frac{\text{bpp}}{8} - 1, h - 1\right)$
Line 0
Line k
Line $h - 1$

Figure 1: Order of image scanlines in data passed to ELFT implementations. Fingerprint image sourced from NIST Special Database 302 [3, 4].

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. Images are canonically encoded. 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.

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 (width × height × $\frac{\text{bpp}}{\text{bpc}} \times \frac{\text{bpp}}{8}$), where *bpp* is bits per pixel.

3 Scenarios and Variables

Analysis in ELFT 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 ELFT is flexible enough to support multiple probe template generation scenarios. Each scenario represents a possible realistic law enforcement search scenario.

3.1.1 Single Latent

This search scenario is the most basic scenario in ELFT. A single latent image or feature set 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 Latents, Same Region

The probe template in a search will always be derived from *at least* one latent friction ridge image or set of features. The API can test searching multiple latent impressions or feature sets simultaneously. In this case, more than one latent image or feature set would be provided to the template creation method to return a single template. All images and features incorporated into a probe template in this search scenario would come from the same hand region, as specified in the set of features provided.

3.1.3 Multiple Latents, Different Regions, Same Subject

A probe template might also be created per-subject instead of per-region. In this case, multiple latent images or feature sets from the same subject but potentially *different* regions of the hand would be passed to the template creation method to return a single template.

When searching with this type of probe template, `Candidate` friction ridge position will be ignored.

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 Image Only

Image only refers to a probe template made from an image without any human-supplied annotation. This is equivalent to an image of a developed latent print being sent directly to an ABIS for fast search. No metadata about the image (e.g., region of interest, orientation, substrate, etc.) is provided.

It's possible some *limited* human intervention may have been applied to the image in this scenario. For example, developing prints with a chemical like 1,2-indanedione requires photographing through a color filter. A CLPE may have converted the image to grayscale manually, using their best judgment to optimize image brightness and contrast. This is known as *image enhancement*. Other development techniques, such as using black powder and tape, would likely be photographed or scanned without any manual intervention, instead relying on device defaults for grayscale capture or conversion.

Image only searching is a step ahead of “lights-out” Tier 1 [5], since image enhancement and proper orientation are not guaranteed.

3.2.2 Enhanced Image Only

This is the same as Image Only (Section 3.2.1), but the image is *known* to have been enhanced such that it is suitable for a human CLPE to annotate. In this scenario, NIST may be able to compare searches of enhanced and non-enhanced images. Software libraries under test will not know if an image has been enhanced.

3.2.3 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” [6]. In the ELFT API, a subset of EFS information is available to software libraries under test to assist in feature extraction and searching. Available features include:

- Position
- Capture technology
- Impression type
- Orientation
- Processing method
- Value assessment
- Pattern classification
- Known/possible lateral reversal
- Known/possible tonal reversal (partial or overall)
- Core, delta, and/or other minutia locations
- Region of interest polygon

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

3.2.4 Features Only

Some or all of the data described in Section 3.2.3 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.

3.2.5 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.

3.2.6 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.

3.2.7 Database Type

Traditionally, only exemplar-quality images are present in a reference database. However, there is emerging interest in latent-to-latent searches. This scenario may help enable linkage of unsolved cases where only latent prints are available. Multiple impressions of some latents may enable enhanced searching through image composition.

4 Application Programming Interface Highlights

The ELFT 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 ELFT API is written in C++ and makes use of C++17 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 ELFT test application. All code should be built with the same compiler to ensure compatibility with the ELFT test application and to prevent difficult to debug link and runtime errors.

4.1 ELFT Namespace

All API code for ELFT exists within the `ELFT` namespace. The namespace contains declarations for several enumerations and structs that are used throughout ELFT API methods.

4.1.1 Metadata Enumerations

Depending on the scenario (Section 3), the software library may be provided with *some* metadata about the image under test. These enumerations map directly to codes described in detail in ANSI/NIST-ITL 1-2011 Update 2015 [6].

- `Impression`
- `FrictionRidgeCaptureTechnology`
- `FrictionRidgeGeneralizedPosition`
- `ProcessingMethod`
- `PatternClassification`
- `ValueAssessment`
- `Substrate`

4.1.2 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.

4.1.3 EFS

The `EFS` 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. Metadata provided in `EFS` can be assumed to come from CLPEs.

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

4.2 Extracting Features

The ELFT 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.

4.2.1 Identification

`ExtractionInterface::getIdentification()` serves two purposes. The primary purpose is a run-time means for the ELFT test application to obtain the name and version number of a submission. This helps NIST enable an automated way of providing the correct inputs to the software library under test and structurally storing its output.

This method also provides a means for organizations to provide marketing information to readers of ELFT analysis reports. If desired, populate the `SubmissionIdentification` struct with marketing and Common Biometric Exchange Formats Framework (CBEFF) information about the exemplar and latent feature extraction algorithms embedded within the software library under test. This information will be printed verbatim in ELFT analysis reports.

4.2.2 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 expected to produce a single buffer of data in the form of a template usable by the software library under test.

Notes

- Both exemplar and latent images are processed in this method. The `EFS imp` member *may* distinguish between the two, but is not required to do so.
- 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 an `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 3.2.4) are facilitated by not providing an `Image`. If no `Image` is provided, there is guaranteed to be some `EFS` data provided. Simply encode this information into a template usable by the `SearchInterface`.
- The source of `EFS` data is not specified. It may be from your software library under test, a different software library under test, or a CLPE.

4.2.3 Create Reference Database

Once all `Reference` templates have been created, the ELFT 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.

4.2.3.1 Database Size

Do not write over `maxSize` bytes when creating the reference database. The data written will be returned to `SearchInterface` in RAM, and if more than `maxSize` bytes are written, this will not be possible. If the software library under test requires more space than `maxSize`, return `ReturnStatus::Result::Failure` and indicate in the `message` parameter approximately how much space is necessary given the `Reference` templates provided.

If possible, fail as soon as possible if `maxSize` is not suitable for the software library under test. This will allow NIST to find more suitable hardware or communicate with the participant about reducing the size of the enrollment database without wasting evaluation resources for other participants. Participants may count on `maxSize` being ≥96 GB. The size of the reference database will be printed in ELFT analysis reports.

4.3 Searching the Database

The ELFT 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 4.2.1), the ELFT API provides a means for providing marketing information about the search algorithm to readers of NIST reports. If desired, return this information in `SearchInterface::getIdentification()`. This information will be printed verbatim in ELFT analysis reports.

4.3.2 Database Manipulation

Methods designed to modify the database are provided as part of the ELFT `SearchInterface` class. Software libraries under test should do whatever processing necessary to perform these operations, but keep in mind the relatively short amount of time allowed before they must return control to the ELFT test application. These are designed to be *rapid* modifications and not a substitute for the database creation operations described in Section 4.2.3. Any modifications made as a result executing of these methods shall persist.

- `exists()` returns `true` or `false` based on if the identifier in question is represented in the reference database.
- `insert()` is used to insert new `Reference` templates with previously unseen identifiers into the database. If the `identifier` already exists in the reference database, data in the newly-provided template should be *merged* with existing data, if such a manipulation is relevant for the software library under test's implementation.
- `remove()` shall remove all information about an identifier from the reference database. The `identifier` shall no longer appear in any candidate lists after this method returns successfully.

4.3.3 Search

`Probe` templates created in the `ExtractionInterface` are searched through the reference database in `SearchInterface::search()`. 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 `Candidate`s with accompanying similarity scores. Since multiple friction ridge positions may be included in a single reference template, `Candidate`s shall include the most localized friction ridge position that is most similar. If friction ridge position cannot be determined but the search is sure of the `Candidate` identifier, a friction ridge position of `UnknownFinger`, `UnknownPalm`, or `UnknownFrictionRidge` can be returned. Depending on the scenario, omitting, misidentifying, or over-generalizing a friction ridge position may result in a miss.

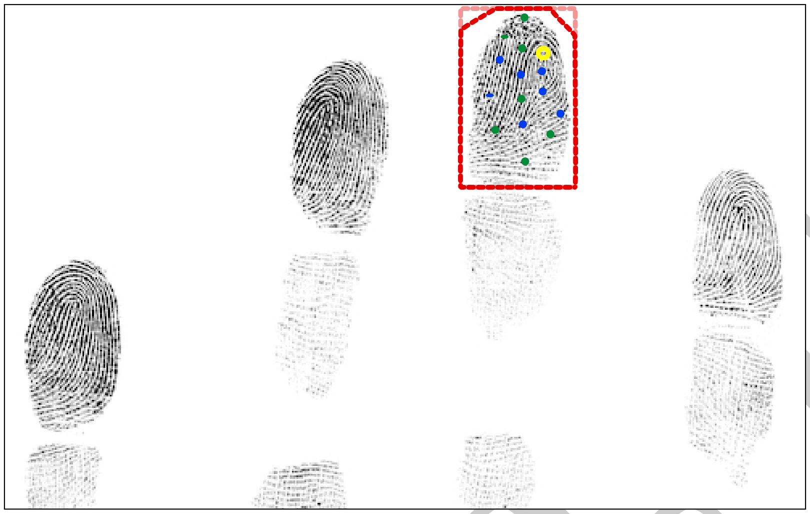
In addition to a list of `Candidates`, this method also asks for a `bool` as to whether or not the software library under test believes the candidate list contains the true mate. This may be based on an internal perceived similarity score threshold or any other heuristic.

4.4 Research Data

The ELFT 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 ELFT evaluation, provide insights into miss analysis, and aid future NIST research.

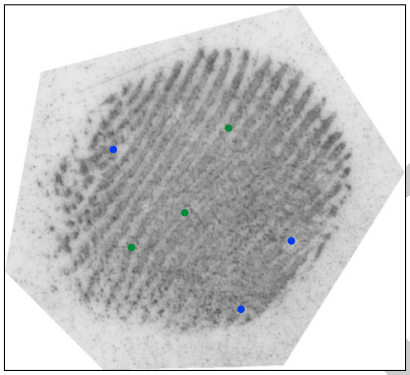
4.4.1 Templates

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 as well as from a latent image in Figure 3.



```
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.efs.frgp = FrictionRidgeGeneralizedPosition::LeftMiddle;
tdLM.efs.roi = std::vector<Coordinate>{{285, 16}, {306, 3}, {341, 3},{356, 20}, {356, 114}, {285, 114}};
tdLM.efs.orientation = 8;
tdLM.efs.valueAssessment = ValueAssessment::Value;
tdLM.efs.pct = PatternClassification::LeftLoop;
tdLM.efs.plr = false;
tdLM.efs.trv = false;

// Locations are relative to the rectangle bounding td.efs.roi (lighter red above).
tdLM.efs.cores = std::vector<Coordinate>{{48, 24}};
tdLM.efs.minutia = std::vector<Minutia>{};
tdLM.efs.minutia->emplace_back({{12, 26}, -14.5, MinutiaType::RidgeEnding});
tdLM.efs.minutia->emplace_back({{{57, 34}, 15.0, MinutiaType::Bifurcation});
tdLM.efs.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 [3, 4].



```
TemplateData td{};
td.imageIdentifier = 13; // Copy, provided in createTemplate()

// Parse your template to get this info. It is hardcoded here for clarity.
td.efs.orientation = 47;
td.efs.valueAssessment = ValueAssessment::Value;
td.efs.pct = PatternClassification::Unknown;
td.efs.plr = false;
td.efs.trv = true;

// No region of interest specified. Locations are relative to the image.
td.efs.minutia = {
    {{59, 35}, -12.5, MinutiaType::RidgeEnding},
    {{90, 107}, 13.3, MinutiaType::Bifurcation},
    {{121, 73}, 47.0, MinutiaType::Bifurcation},
    // . . .
};

td.imageQuality = 71;

return ({td});
```

Figure 3: Example of returning `TemplateData` from `ExtractionInterface::extractTemplateData()`. In this example, a single latent lift (`fct = 22`) was provided to `ExtractionInterface::createTemplate()`. The graphic above visualizes what data might have been recorded in a template. Green and blue dots indicate minutia. Because no region of interest is specified, all `Coordinate` are relative to the bounds of the image. All annotations are simulated. Fingerprint image sourced from NIST Special Database 302 [3, 4].

```
std::vector<std::vector<Correspondence>> allCandidateCorrespondence{};
allCandidateCorrespondence.reserve(searchResult.candidateList.size());

for (const Candidate &candidate : searchResult.candidateList) {
        // Retrieve reference template for this Candidate. For simplicity, we'll assume this private method returns an
        // ELFT::TemplateData.
        const TemplateData referenceTemplate = this->retrieve(candidate);

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

        // Save away all those Minutia
        std::vector<Correspondence> oneCandidateCorrespondence{};
        oneCandidateCorrespondence.reserve(minutia.size());
        for (const Minutia &m : minutia) {
                Correspondence c{};
                c.probeImageIdentifier = 13; // Copy, provided in createTemplate()
                c.referenceImageIdentifier = 6; // Copy, provided in createTemplate()

                // Corresponding Minutia MUST align from TemplateData. The commented values here are for clarity and are from
                // the previous extractTemplateData() examples.
                c.probeMinutia = m.first; // Iteration 1: {12, 26}; Iteration 2: {57, 34}; Iteration 3: {47, 12}; . . .
                c.referenceMinutia = m.second; // Iteration 1: {59, 35}; Iteration 2: {90, 107}; Iteration 3: {121, 73}; . . .

                oneCandidateCorrespondence.push_back(c);
        }

        allCandidateCorrespondence.push_back(oneCandidateCorrespondence);
}

return (allCandidateCorrespondence);
```

Figure 4: Example of returning `Correspondence` from `SearchInterface::extractCorrespondence()`. For continuity, the same images parsed in Figures 2 and 3 are used here. For each `Candidate` in `searchResult.candidateList`, a placeholder `std::vector` of `Correspondence` 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 `Correspondence` objects. The `Minutia` set here *must* match the `Minutia` returned in `TemplateData` from `ExtractionInterface::extractTemplateData()`.

4.4.2 Correspondence

Many latent search algorithms operate by mimicking the actions of CLPEs—corresponding groupings of minutia found in latent images 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 4.

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 ELFT interface. The ELFT test application will exclusively use the returned object for calling API methods. When the ELFT test application `fork`s, 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 ELFT test application (Section 4.1.2). It's not always possible to safely jump out of code in certain error conditions (e.g., a memory allocation failure). In this case, it may be appropriate to throw an exception. The ELFT 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, `fork`ing on each node to run many tasks in parallel. See Section 5.1.1 for more details.

`ELFT::ExtractionInterface::createReferenceDatabase()` is an exception to this rule. NIST will run this method on a single node in a single process. For this method, NIST *expects* the software library under test to make use of threading (if necessary) to complete the creation 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.

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
Extraction	`getImplementation()`	Max	5 s
	`getIdentification()`	Max	250 ms
	`createTemplate()`	Sample Mean	500 µs per sample
	`extractTemplateData()`	Sample Mean	500 µs
	`createReferenceDatabase()`	Max	10 ms per identifier
Search	`getImplementation()`	Max	5 s
	`getIdentification()`	Max	250 ms
	`exists()`	Max	5 s
	`insert()`	Max	5 s
	`remove()`	Max	5 s
	`search()`	Sample Mean	300 µs per identifier
	`extractCorrespondence()`	Max	5 s

Table 1: API runtime requirements. *API Method* indicates the ELFT 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.

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 ELFT test application is provided on the ELFT website in the ELFT 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 5.1.5. 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 5.3). Any required libraries that are not standard to CentOS 8.1.1911 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 (`RPATH`).

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 ELFT 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 5.3). Filenames and checksums of all provided files will be reported in ELFT analysis reports. The path to such files will be provided as a parameter to `getImplementation()` (Section 4.5.1). 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 5.1.5) to correct the defect.

5.1.1.3 Multiprocessing

With one exception, the software library shall not make use of threading, forking, OpenMP, or any other multiprocessing techniques. The ELFT 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 ELFT 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.

The single exception to this rule is the API method `ELFT::ExtractionInterface::createReferenceDatabase()`, which can and should use multiple threads, standard template library parallel execution policies, OpenMP, or any similar multi-threading technique in order to create reference databases as fast as possible. The ELFT test application will call this method from a single process on an otherwise idle machine. The reference database will be read-only after creation.

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 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 `RPATH` of these dependent libraries be set to `$ORIGIN`, since the only participant library that the ELFT test application will explicitly link is the core library. The ELFT test application's `RPATH` will include the directory containing the participant's core library. Filenames and checksums of all library files will be reported in ELFT analysis reports.

5.1.3 `libelft.so`

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

5.1.4 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 5.3) on the **Intel Xeon E5-2680, Intel**

Xeon E5-4650, and **Intel Xeon Gold 6140** CPUs. Speed tests that run on a fixed sample dataset will be run as described in Section 5.4.

5.1.5 Naming

The core software library submitted for ELFT shall be named in a predefined format. The first part of the software library's name shall be `libelft_`. 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 ELFT test driver is the core library, as demonstrated in Sections 5.1.2 and 5.1.6.

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 4.2.1 for this purpose.

Example
Initech submits a software library named `libelft_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 `libelft_initech_101D.so`. In ELFT analysis reports, NIST refers to Initech's library as `initech+101D`.

5.1.6 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 `RedHat 8.3.1-4`, via `mpicxx`) under **CentOS 8.1.1911**, as seen in Figure 5.

```
mpicxx -o elft elft.cpp -Llib -Wl,-rpath,lib -lelft -lelft_initech_101D -lstdc++fs
```

Figure 5: Example compilation and link command for the ELFT test application.

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 CentOS 8.1.1911. NIST is using the base CentOS release install and **not** CentOS Stream, which typically targets sources to be present in the next minor release. Participants are **strongly encouraged** to verify library-level compatibility with `g++` on CentOS 8.1.1911 **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 ELFT 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* ELFT 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 ELFT 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 CentOS 8.1.1911 will be rejected. Any submissions not signed and encrypted with the private key whose public key fingerprint is recorded on the participant's ELFT 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 ELFT analysis reports, NIST must receive a signed ELFT agreement. This agreement must be physically mailed or faxed to NIST. E-mailed agreements cannot be accepted. Even if the information has not changed, a new agreement must be submitted for each ELFT analysis report NIST posts.

5.3.2 Communication

All communication to NIST shall be addressed to the ELFT e-mail alias `elft@nist.gov` and not a specific member of the ELFT 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 ELFT dataset (Section 2) on 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 Table 1 will be rejected. A table of timing requirements can be seen in Section 4.5.4.

Speed tests of `ELFT::ExtractionInterface::createReferenceDatabase()` will also be performed on an **Intel Xeon Gold 6140** 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 ELFT 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] Dvornychenko VN, Garris MD (2006) Summary of NIST Latent Fingerprint Testing Workshop. *NIST Interagency Report 7377* `https://doi.org/10.6028/NIST.IR.7377`

[2] Indovina MD, Dvornychenko VN, Hicklin RA, Kiebuzinskio GI (2012) ELFT-EFS Evaluation of Latent Fingerprint Technologies: Extended Feature Sets [Evaluation #2]. *NIST Interagency Report 7859* `https://doi.org/10.6028/NIST.IR.7859`

[3] 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`

[4] Flanagan PA NIST Special Database 302 Nail to Nail (N2N) Fingerprint Challenge, National Institute of Standards and Technology, Zip Archive. `https://doi.org/10.18434/M31943`

[5] Meagher S, Dvornychenko V (2011) Defining AFIS Latent Print "Lights-Out". *NIST Interagency Report 7811* `https://doi.org/10.6028/NIST.IR.7811`

[6] 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

07 May 2020 Initial draft for public comment.