Slap Fingerprint Segmentation Evaluation III

Test Plan and Application Programming Interface

Last Updated: 27 January 2022

Contents

Disclaimer

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

1 Introduction

In 2004, the National Institute of Standards and Technology (NIST) conducted *Slap Fingerprint Segmentation Evaluation 2004 (SlapSeg04)* [1] to assess the current state of slap fingerprint segmentation technologies at the time. As compute and capture technology has advanced, it's necessary to examine the latest generation of algorithms. *Slap Fingerprint Segmentation Evaluation III (SlapSeg III)* provides an opportunity for providers of slap fingerprint segmentation algorithms to submit segmentation solutions on an ongoing basis and to compare the results from their latest improvements on a fixed dataset.

1.1 Background

Fingerprint data is collected and maintained in the form of tenprint cards or Identification Flats (ID Flats). Traditional tenprint cards are comprised of the rolled impressions of each of an individual's ten fingers, as well as four *slap* impressions: the left slap (index, middle, ring, and little fingers of the left hand), the right slap (index, middle, ring, and little fingers of the right hand), the left thumb, and the right thumb. Slaps are taken by pressing the associated fingers of one hand onto a scanner or fingerprint card simultaneously. Tenprint card slaps, whether scanned inked cards or live-scan captures, are also called *Two Inch (2 in)* captures, referring to the height of the typical capture area. ID Flats are tenprint records that are constructed by capturing three discrete impressions: left slap, right slap, and thumb slap (left and right thumbs simultaneously). ID Flats are images that were captured on newer live-scan devices that use a larger, 3 in tall platen, giving the resulting images the name *Three Inch (3 in)* captures.

Several Federal agencies rely on slap segmentation algorithms, including to determine if a slap image should be recaptured. The Federal Bureau of Investigation (FBI) receives the majority of fingerprint submissions electronically from live-scan devices, but maintains hundreds of thousands of digitally-converted tenprint ink cards. The Department of State (DOS) and Department of Homeland Security (DHS)'s United States Visitor and Immigrant Status Indicator Technology (US-VISIT) program now capture ID Flats, having previously migrated from two-finger capture.

The FBI also maintains a database of palm images. Palm images are typically captured as half palm on a 5.5 in platen, or the full palm on an 8 in platen. As these devices often produce higher resolution images than typical tenprint or ID Flat images, it's useful to be able to segment fingerprints from palm captures as well. SlapSeg III introduces palm capture images as *Five and a half Inch (5.5 in)* or *upper palm* captures and *Eight Inch (8 in)* or *full palm* captures for evaluation.

1.2 What's New Since Slap Fingerprint Segmentation Evaluation II

The submission process for SlapSeg III has changed significantly since previous tests [2], bringing it in line with other NIST Image Group biometric technology evaluations.

- Participants will submit 64-bit **software libraries** that implement an application programming interface (API) under **Ubuntu 20.04.03 LTS**.
- Addition of two size classes of **palm data**.
- **Reporting** of capture errors and hand orientation.

2 Evaluation Data

The segmentation process varies for the various sizes of slap images due to the groundtruth available, as well as the sizes, rotation, and number of components within the image. Because of the differences in the segmentation process, SlapSeg III evaluates segmentation of the different data types as separate tests. Participants will be given the option of selecting which data types they support. Each test will be run using data from thousands of individuals.

The two inch data consists of images where the fingerprints appear rotated uniformly. Groundtruth data for two inch data maintains a uniform angle of rotation for all four fingerprint segments within an image. Fingers in three inch data are assumed to be in an upright position. The slope of the segments in the bounding rectangle generated by an implementation for three inch data should be 0 (horizontal, y = y intercept) or undefined (vertical line, x = x intercept). Fingers in upper palm and full palm images may have varying degrees of rotation for *each* finger, which is reflected in the groundtruth.

2.1 Dataset Groundtruth

The groundtruth data for two inch and three inch data is based on the NIST fingerprint segmentation algorithm `nfseg` [3]. Humans examined every slap image starting with the `nfseg`-generated segmentation boxes and then hand-corrected all errors to produce the groundtruth segmentation. The three main errors examiners looked for were excess white space between a segmentation box edge and the fingerprint, a box side touching fingerprint ridges, and the bottom side correctly placed at the distal interphalangeal joint. Figure 1 shows an example of groundtruth segmentation boxes. Upper palm and full palm data were groundtruthed completely by hand.

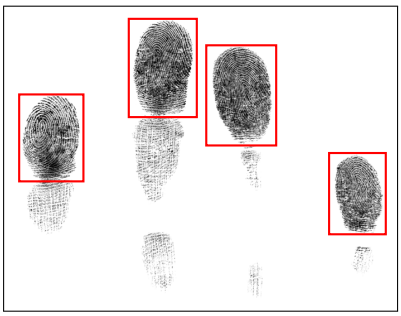

Figure 1: Sample groundtruth boxes for a three inch slap.

The groundtruth boxes were placed to capture only the distal phalanx (i.e., the finger tip or finger joint). The left, right, and top sides of the segmentation boxes were placed so that a small amount of white space existed between the segmentation box and those edges of the fingerprint. If two fingers were touching, the box sides were placed along the point of contact. Sample groundtruth information is provided to SlapSeg III participants as part of software validation, to allow participants to view examples of ideal slap segmentation.

The bottom side of the segmentation box was placed in the middle of the distal interphalangeal joint of the finger. If there was not a well-defined white space at the joint, the box was still placed in the middle of the joint cutting through any ridge information that existed. If there was a slight slant in the fingerprint, the bottom side was placed to include the lowest part of the joint inside the segmentation box. Groundtruth segmentation boxes do not extend past the edges of the slap image for three inch slap images, but corners may be outside the edge of the image (i.e., $x < 0, y < 0, x \geq w, y \geq h$) depending on rotation angle in other size classes. Although not strictly required by implementations, the groundtruth angle of rotation for all fingers in a slap are identical for 2 in. The groundtruth angle of individual fingers in 5.5 in and 8 in slap images may differ. The current guidance from the FBI indicates that thumbs should not be present in captures of the upper palm. As such, no segmentation positions have been recorded for upper palm data that may contain the thumb. Thumbs should not be segmented in full palm captures.

2.1.1 Success and Tolerances

Successful segmentation is determined by comparing the segmentation positions from an implementation to the hand-marked groundtruth segmentation positions for the same data. Tolerances between the two boxes are allowed. These tolerances are based on work completed in Slap Fingerprint Segmentation Evaluation II (SlapSeg II) [2] to show that the tolerances would not impede a fingerprint matching algorithm's ability to correctly match a segmented fingerprint. These calculations are provided in [2] and in supplemental information on the SlapSeg III website.

2.2 Orientation Determination

New in SlapSeg III is a method to predict the orientation of a slap image. Given an image, implementations are asked to return whether they think the image is of a left slap, a right slap, or a thumb slap. Although implementing a routine to predict hand orientation is optional, it is highly encouraged. An accurate routine may be a good candidate for use in a self-service kiosk deployed within the European Union (EU)'s Entry-Exit-System (EES).

2.3 Access to Evaluation Data

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

2.4 Format

The software library must be capable of processing fingerprint images in uncompressed raw 8 bit (one byte per pixel) grayscale format. Images shall follow the scan sequence as defined by ISO/IEC 19794-4:2005, §6.2, paraphrased here, and visualized in Figure 2. The origin is the upper-left corner of the image. The X-coordinate (horizontal) position shall increase positively from the origin to the right side of the image. The Y-coordinate (vertical) position shall increase positively from the origin to the bottom of the image.

Raw 8 bit grayscale images are canonically encoded. The minimum value that will be assigned to a "black" pixel is zero. The maximum value that will be assigned to a "white" pixel is 255. Intermediate gray levels will have assigned values of 1 to 254. The pixels are stored left to right, top to bottom, with one byte per pixel. The number of bytes in an image is equal to its height multiplied by its width as measured in pixels. The image width and height in pixels will be supplied to the software library as supplemental information.

Images for this test will employ varying resolutions, but primarily 500 pixels per inch (PPI) and 1 000 PPI. Horizontal and vertical directions will be equivalent.

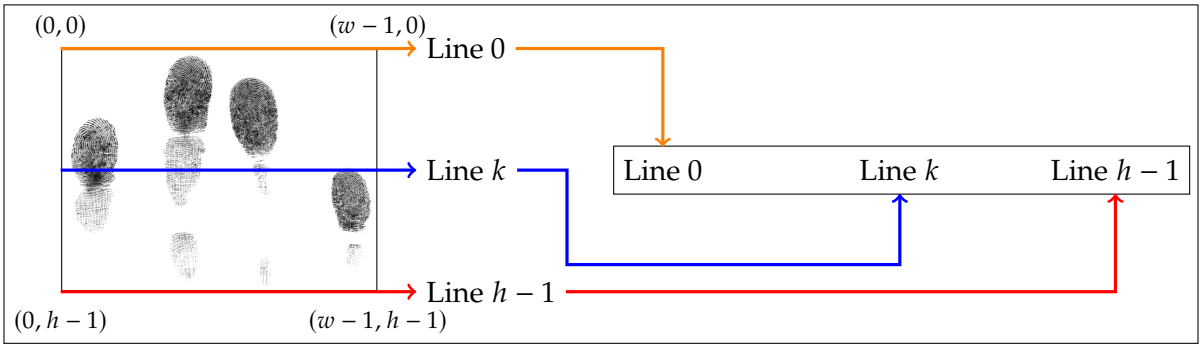

Figure 2: Order of image scanlines in data passed to SlapSeg III implementations.

3 Application Programming Interface

3.1 Enumerations

Enumeration	Explanation
`Unknown`	Unknown
`RightThumb`	Right thumb
`RightIndex`	Right index
`RightMiddle`	Right middle
`RightRing`	Right ring
`RightLittle`	Right little
`LeftThumb`	Left thumb
`LeftIndex`	Left index
`LeftMiddle`	Left middle
`LeftRing`	Left ring
`LeftLittle`	Left little

Table 1: `SlapSegIII::FrictionRidgeGeneralizedPosition`.

Enumeration	Explanation
`Unknown`	Unknown
`ScannedInkOnPaper`	Scanned ink on paper
`OpticalTIRBright`	Optical sensor, black ridges on white background

Table 2: `SlapSegIII::SlapImage::CaptureTechnology`.

Enumeration	Explanation
`Right`	Right Hand
`Left`	Left Hand
`Thumbs`	Both Thumbs

Table 3: `SlapSegIII::SlapImage::Orientation`.

Enumeration	Explanation
`TwoInch`	Tenprint card slaps (2 in)
`ThreeInch`	ID Flats (3 in)
`UpperPalm`	Upper Palm Data (5.5 in)
`FullPalm`	Full Palm Data (8 in)

Table 4: `SlapSegIII::SlapImage::Kind`.

Enumeration	Explanation
`Artifacts`	Moisture, *ghost* impressions, etc.
`ImageQuality`	Low contrast, over-inked, etc.
`HandGeometry`	Incorrect hand placement
`Incomplete`	Structure does not resemble slap

Table 5: `SlapSegIII::SlapImage::Deficiency`.

Enumeration	Explanation
`Success`	Success
`InvalidImageData`	Image data was not parsable
`RequestRecapture`	Image has deficiencies preventing reliable segmentation
`RequestRecaptureWithAttempt`	Image has deficiencies, but segmentation was attempted
`UnsupportedResolution`	Image resolution is not supported
`UnsupportedSlapType`	Slap type is not supported
`NotImplemented`	Method called is not implemented
`VendorDefined`	Vendor-defined error (described in `message`)

Table 6: `SlapSegIII::ReturnStatus::Code`.

Enumeration	Explanation
`Success`	Success
`FingerNotFound`	Finger not found
`FailedToSegment`	Finger found, but could not be segmented
`VendorDefined`	Vendor-defined failure (described in `message`)

Table 7: `SlapSegIII::SegmentationPosition::Result::Code`.

3.2 Classes and Structures

3.2.1 Coordinate

```
const int32_t x;
const int32_t y;
```

Figure 3: `SlapSegIII::Coordinate` members.

```
Coordinate(
    const int32_t x = 0,
    const int32_t y = 0)
    noexcept;
```

Figure 4: `SlapSegIII::Coordinate` constructor.

3.2.1.1 Members

- `x`: X-coordinate.
- `y`: Y-coordinate.

3.2.1.2 Description

Storage of an (x, y) location, assuming an origin at the top left. The default constructor creates the location $(0, 0)$.

3.2.2 SlapImage

```
const uint16_t width;
const uint16_t height;
const uint16_t ppi;
const Kind kind;
const CaptureTechnology captureTechnology;
const Orientation orientation;
const std::vector<std::byte> data;
```

Figure 5: `SlapSegIII::SlapImage` members.

3.2.2.1 Members

- `width`: Width of the image, in pixels.
- `height`: Height of the image, in pixels.
- `ppi`: Resolution of the image, in PPI.
- `kind`: `SlapImage::Kind` of capture depicted in the image.
- `captureTechnology`: `SlapImage::CaptureTechnology` used to create the image.
- `orientation`: `SlapImage::Orientation` of the hand depicted in the image.
- `data`: Image data (Section 2.4).

3.2.2.2 Description

A container for data and properties of a slap image. Participants will never need to construct an instance of `SlapImage`, but will need to access its members. `SlapImage.data` is stored as a `std::vector` of bytes, as described in Section 2.4.

Note
To pass `SlapImage.data` as an C-style array without duplicating memory, invoke the `data()` method, as shown in Figure 6. You may need to `reinterpret_cast` as well, depending on your function signature.

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

/* ... pass image data from segment() like this : */
struct proprietary out;
find_segmentation_positions(data.data(), &out);
```

Figure 6: Converting image `data` to a C-style array in constant time without additional memory allocations.

3.2.3 SegmentationPosition

```
using FRGP =
    FrictionRidgeGeneralizedPosition;
FRGP frgp;
Coordinate &tl;
Coordinate &tr;
Coordinate &bl;
Coordinate &br;
Result result;
```

Figure 7: `SlapSegIII::SegmentationPosition` members.

```
using FRGP =
    FrictionRidgeGeneralizedPosition;
SegmentationPosition(
    const FRGP frgp,
    const Coordinate &tl,
    const Coordinate &tr,
    const Coordinate &bl,
    const Coordinate &br,
    const Result result = {});
```

Figure 8: `SlapSegIII::SegmentationPosition` constructor.

3.2.3.1 Parameters

- `frgp`: Segmented `FrictionRidgeGeneralizedPosition`.
- `tl`: Top-left `Coordinate` of segment, where *top* refers to the top of the fingerprint.
- `tr`: Top-right `Coordinate` of segment, where *top* refers to the top of the fingerprint.
- `bl`: Bottom-left `Coordinate` of segment, where *bottom* refers to the distal interphalangeal joint of the fingerprint.
- `br`: Bottom-right `Coordinate` of segment, where *bottom* refers to the distal interphalangeal joint of the fingerprint.
- `result`: A `Result` summarizing the result of the segmentation operation for only this segment.

3.2.3.2 Description

A `SlapSegIII::SegmentationPosition` is returned to represent the segmentation positions within an image. In failure conditions, it may be useful for participants to provide debugging information in `message`.

- `frgp` shall not be `Unknown`. A failure to identify the friction ridge generalized position is a failure to segment.
- The coordinates shall create a rectangle, where $|\overline{t_l t_r}| = |\overline{b_l b_r}|$ and $|\overline{t_l b_l}| = |\overline{t_r b_r}|$.

3.2.4 SegmentationPosition::Result

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

Figure 9: `SlapSegIII::SegmentationPosition::Result` members.

```
Result(
    const Code code = Code::Success,
    const std::string &message = "");
```

Figure 10: `SlapSegIII::SegmentationPosition::Result` constructor.

3.2.4.1 Members

- `code`: A `SegmentationPosition::Result::Code` summarizing the success or failure of discovering a segmentation position for an individual finger.
- `message`: A message providing more information about why a particular `Code` was chosen (optional).

3.2.4.2 Description

`SlapSegIII::SegmentationPosition::Result` is a structure that allows participants to return information about the status of discovering an individual segmentation position. It consists of a combination of a `Result::Code` and an optional `std::string`. Under normal operating conditions, participants need only to indicate `Result::Code::Success`. Under failure conditions, it may be useful for participants to provide debugging information in the `message` parameter of a `Result` to help document the issue. Implementations shall **always give their best-effort segmentation positions if possible, even under failure**. This helps determine if an implementation can correctly flag and ultimately work around capture errors. Examples of using `Result` are shown in Figure 11.

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

```
/* Use the default arguments to indicate success */
const SlapSegIII::SegmentationPosition::Result middle{};

/* Explicitly indicate the status code */
const SlapSegIII::SegmentationPosition::Result ring{
    SlapSegIII::SegmentationPosition::Result::Code::Success};

/* Provide a debugging message */
const SlapSegIII::SegmentationPosition::Result index{
    SlapSegIII::SegmentationPosition::Result::Code::FingerNotFound,
    "Finger appears to be amputated"};
```

Figure 11: Ways to return a `SlapSegIII::SegmentationPosition::Result`.

3.2.5 SubmissionIdentification

```
uint16_t version;
std::string libraryIdentifier;
std::string marketingIdentifier;
```

Figure 12: `SlapSegIII::SubmissionIdentification` members.

```
SubmissionIdentification(
    const std::string &libraryIdentifier,
    const uint16_t version,
    const std::string &marketingIdentifier = "");
```

Figure 13: `SlapSegIII::SubmissionIdentification` constructor.

3.2.5.1 Members

- `version`: Version number for this submission. The version number must be unique for every submission, including bug fix submissions submitted prior to results being published. Must be the same as the final underscore-delimited token of the filename of the core library. Refer to Section 4.1.5 for complete details.
- `libraryIdentifier`: Identifier for this submission. Should be the same for all submissions from the same organization. Must match the regular expression `[:alnum:]+`. Must be the same as the second underscore-delimited token of the filename of the core library. Refer to Section 4.1.5 for complete details.
- `marketingIdentifier`: Marketing name for this submission. Optional. Must match the regular expression `[[:graph:] ]*`.

3.2.5.2 Description

Storage for identification information about this submission. Values from this structure are used to refer to the algorithm in publications.

3.2.6 ReturnStatus

```
ReturnStatus::Code code;
std::set<SlapImage::Deficiency>
    imageDeficiencies;
std::string message;
```

Figure 14: `SlapSegIII::ReturnStatus` members.

```
ReturnStatus(
    const Code code = Code::Success,
    std::set<SlapImage::Deficiency>
        imageDeficiencies = {},
    const std::string &message = "");
```

Figure 15: `SlapSegIII::ReturnStatus` constructor.

3.2.6.1 Members

- `code`: A `ReturnStatus::Code` summarizing the success or failure of an operation.
- `imageDeficiencies`: A set of `Deficiency` indicating why an image is not suitable for segmentation (required when `code` is `RequestRecapture` or `RequestRecaptureWithAttempt`).
- `message`: A message providing more information about why a particular `Code` was chosen (optional).

3.2.6.2 Description

`SlapSegIII::ReturnStatus` is a structure that allows participants to return information about the status of calling SlapSeg III API methods. Under normal operating conditions, participants need only to indicate `ReturnStatus::Code::Success`. Under failure conditions, it may be useful for participants to provide debugging information in the `message` parameter of a `ReturnStatus::Code` to help resolve the issue. When `code` is `RequestRecapture` or `RequestRecaptureWithAttempt`, one or more `Deficiency` shall be specified. Examples of using `ReturnStatus` are shown in Figure 16.

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

```
/* Use the default arguments to indicate success */
const SlapSegIII::ReturnStatus rs1{};

/* Provide a debugging message */
const SlapSegIII::ReturnStatus rs2{ReturnStatus::Code::UnsupportedResolution,
    {}, "1000 ppi not supported"};

/* Refuse to process an image */
const SlapSegIII::ReturnStatus rs3{ReturnStatus::Code::RequestRecaptureWithAttempt,
     {SlapImage::Deficiency::Artifacts, SlapImage::Deficiency::HandGeometry}};
```

Figure 16: Ways to return a `SlapSegIII::ReturnStatus`.

3.3 Interface

Participants in SlapSeg III must submit a software library that fully implements the pure abstract C++ class `SlapSegIII::Interface`. Since the SlapSeg III test driver will not know the name of the `SlapSegIII::Interface` class at compile time, the software library must also implement a factory method to return an instance of their implementation. NIST's declaration of the factory method is shown in Figure 17.

3.3.1 Obtain SlapSeg III Implementation

```
std::shared_ptr<SlapSegIII::Interface>
getImplementation(
    const std::filesystem::Path &configurationDirectory);
```

Figure 17: Declaration of a function to obtain an instance of the participant's `SlapSegIII::Interface` implementation.

3.3.1.1 Description

Obtain a managed pointer to an object implementing `SlapSegIII::Interface`.

3.3.1.2 Parameters

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

3.3.1.3 Return

A managed pointer to the participant's implementation of `SlapSegIII::Interface`. A sufficient implementation of this method for an implementation whose constructor has no arguments could be the return statement shown in Figure 18.

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

Figure 18: A sufficient implementation of `SlapSegIII::Interface::getImplementation()`.

3.3.1.4 Speed

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

3.3.2 Identification

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

Figure 19: Declaration of a method to obtain identification information for this submission.

3.3.2.1 Description

This function allows for the retrieval of identification and version information of the library at runtime.

- The returned value's `uint16_t` member shall be identical to the four hexadecimal characters prior to the extension of the submitted software library's name (Section 4.1.5).
- The `std::string` member of the returned value shall be identical to the string surrounded by underscores, just prior to the four hexadecimal digit version, in the submitted software library's name (Section 4.1.5).

3.3.2.2 Return

This method shall immediately return a `SlapSegIII::SubmissionIdentification` of the identifier and version number for this software library. A marketing name to be printed in publications may optionally be provided. A sufficient implementation for the library `libslapsegiii_initech_101D.so` is shown in Figure 20.

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

/* A more verbose implementation of getIdentification */
SlapSegIII::SubmissionIdentification id;
id.libraryIdentifier = "initech";
id.version = 0x101D;
id.marketingIdentifier = "Initech Fingerprint Segmenter (version 2.0)";
return (id);
```

Figure 20: Two sufficient implementations of `SlapSegIII::Interface::getIdentification()` for the library `libslapsegiii_initech_101D.so`.

3.3.2.3 Speed

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

3.3.3 Declare Supported Imagery

```
std::tuple<std::set<SlapImage::Kind>, bool>
getSupported()
    const;
```

Figure 21: Declaration of a method to obtain information about the `SlapImage::Kind` supported by this implementation.

3.3.3.1 Description

Determine the kinds of slap imagery supported by this software library at runtime, and whether or not the software library contains an orientation determination routine. Participants will not be evaluated on features not returned by this method. While support of all image types are encouraged, at least one `SlapImage` type is required.

3.3.3.2 Return

This method shall immediately return a tuple whose first member is the set of `SlapImage::Kind` that are supported and whose second member is a boolean indicating whether or not is `determineOrientation()` is implemented. An example of how to return this information is shown in Figure 22.

```
/* Support all types of images and implement an orientation determination routine. */
const std::set<SlapImage::Kind> kinds{
    SlapImage::Kind::TwoInch, SlapImage::Kind::ThreeInch,
    SlapImage::Kind::UpperPalm, SlapImage::Kind::FullPalm};
return (std::make_tuple(kinds, true));

/* Support only those types present in SlapSegII and do not implement determineOrientation */
const std::set<SlapImage::Kind> kinds{
    SlapImage::Kind::TwoInch, SlapImage::Kind::ThreeInch};
return (std::make_tuple(kinds, false));

/* Support only those types present in SlapSeg04 and do not implement determineOrientation */
const std::set<SlapImage::Kind> kinds{SlapImage::Kind::TwoInch};
return (std::make_tuple(kinds, false));
```

Figure 22: Example implementations of `SlapSegIII::Interface::getSupported()`.

3.3.3.3 Speed

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

3.3.4 Segmentation

```
std::tuple<ReturnStatus, std::vector<SegmentationPosition>>
segment(
    const SlapImage &image);
```

Figure 23: Declaration of a method that performs slap fingerprint segmentation.

3.3.4.1 Parameters

- `image`: `SlapImage` data and metadata.

3.3.4.2 Description

This method takes raw image data and metadata as input and returns a collection of `SegmentationPositions`, each identifying the segmentation position of a finger within an image.

- If successful, the collection of `SegmentationPositions` shall contain four entries for `Orientation::Left` and `Orientation::Right`, and two entries for `Orientation::Thumbs`.
- `SegmentationPositions` for `SlapImage::Kind::ThreeInch` data shall not be rotated.
- Groundtruth `SegmentationPositions` for `SlapImage::Kind::TwoInch` data have identical rotation angles for all fingers. Other `SlapImage::Kinds` may have different groundtruth rotation angles.
- Corners of rotated rectangles may be outside of the image.
- Fail without any segmentation positions ***only* as a last resort**. Make use of `Code::RequestRecaptureWithAttempt` to perform best-effort segmentation while demonstrating that you understand the image should be recaptured in a live capture scenario.

3.3.4.3 Return

The `ReturnStatus` member in the return of this method shall be set to `Code::Success` when successful, or another approved `Code` with an optional `message` on failure. If returning `Code::Success` or `Code::RequestRecaptureWithAttempt`, the `std::vector` member shall contain the appropriate number of entries for the `SlapImage::Kind` of image provided, even if there were failures to segment individual fingers. Other `ReturnStatus::Codes` will ignore `SegmentationPositions` and be treated as failures to segment.

3.3.4.4 Speed

The runtime maximums for this method differ based on the kind of image data provided, as shown in Table 8. Values are averages computed over a fixed subset of the dataset.

`SlapImage::Kind`	Mean
`TwoInch`	≤1.5 s
`ThreeInch`	≤1.5 s
`UpperPalm`	≤1.5 s
`FullPalm`	≤1.5 s

Table 8: Maximum mean times to return from `SlapSegIII::Interface::segment()`.

3.3.5 Orientation Determination

```
std::tuple<ReturnStatus, SlapImage::Orientation>
determineOrientation(
    const SlapImage &image);
```

Figure 24: Declaration of a method that detects the orientation of a slap fingerprint image.

3.3.5.1 Parameters

- `image`: `SlapImage` data and metadata.

3.3.5.2 Description

This method takes raw image data and metadata as input and returns a hypothesized hand orientation (`Left`, `Right`, or `Thumbs`).

- All libraries must implement this method, but may return `Code::NotImplemented` if no routine is available (as seen in Figure 25). Libraries must also indicate failure to supply an orientation determination routine in `getSupported()`.
- The field for `orientation` in `image` will be default initialized and implementations should not refer to this field.

```
/* Do not support determining slap fingerprint image orientations */
return (std::make_tuple(ReturnStatus::Code::NotImplemented,
    SlapImage::Orientation{}));
```

Figure 25: Example implementation of `SlapSegIII::Interface::determineOrientation()` when no orientation determination algorithm is present.

3.3.5.3 Return

The `ReturnStatus` member in the return of this method shall be set to `Code::Success` when successful, or another approved `Code` with an optional `message` on failure. `Orientation`s are considered only when `Code::Success` is set. Other `ReturnStatus::Code`s will result in the `Orientation` being marked incorrect, so long as a valid slap image was passed to the method.

3.3.5.4 Speed

The runtime maximums for this method differ based on the kind of image data provided, as shown in Table 9. Values are averages computed over a fixed subset of the dataset.

`SlapImage::Kind`	Mean
`TwoInch`	≤1 s
`ThreeInch`	≤1 s
`UpperPalm`	≤1 s
`FullPalm`	≤1 s

Table 9: Maximum mean times to return from `SlapSegIII::Interface::segment()`.

4 Software and Documentation

4.1 Software Libraries and Platform Requirements

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

4.1.1 Restrictions

4.1.1.1 Dynamic Library

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

4.1.1.2 Single Configuration

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

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

Example
A participant submits a software library that internally has a customizable downsampling resolution. The software library decides which resolution 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 resolution *A*. After NIST discovers a defect, the participant realizes the defect is not present when the resolution is set to *B*, and could be corrected by a small change to `config.txt`. Even though the change is minor, the participant must submit a new `config.txt` *and* a new software library with incremented version number (Section 4.1.5) to correct the defect.

4.1.1.3 Multiprocessing

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

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

4.1.1.4 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.

4.1.1.5 Filesystem

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

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

4.1.2 External Dependencies

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

4.1.3 `libslapsegiii.so`

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

4.1.4 Hardware Dependencies

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

4.1.5 Naming

The core software library submitted for SlapSeg III shall be named in a predefined format. The first part of the software library's name shall be `libslapsegiii_`. The second piece of the software

```
mpicxx -o ss3 ss3.cpp -Llib -Wl,--enable-new-dtags -Wl,-rpath,lib -lslapsegiii \
       -lslapsegiii_initech_101D -std=c++17
```

Figure 26: Example compilation and link command for the SlapSeg III test application.

library's name shall be a non-infringing and case-sensitive unique identifier that matches the regular expression `[:alnum:]+` (likely your organization's name), followed by an underscore. The final part of the software library's name shall be a four hexadecimal digit version number, followed by a file extension. **Be cognizant of the name** you provide, 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** library that will be explicitly linked to the SlapSeg III test driver is the core library, as demonstrated in Section 4.1.6.

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

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

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

Example
Initech submits a SlapSeg III shared library named `libslapsegiii_initech_101C.so` with build 4 124 of their algorithm. This library returns `{"initech", 0x101C}` from `getIdentification()`. NIST determines that Initech's `segment()` method is too slow and rejects the library. Initech submits build 4 125 to correct the defects in 4 124. Initech updates `getIdentification()` in their implementation to return `{"initech", 0x101D}` and renames their library to `libslapsegiii_initech_101D.so`. In reports, NIST refers to Initech's library as `initech+101D`.

4.1.6 Operating Environment

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

NIST will link the provided library files to a C++17 language test driver application using the compiler `g++` (version `Ubuntu 9.3.0-17ubuntu1~20.04`, via `mpicxx`) under **Ubuntu 20.04.03 LTS**, as seen in Figure 26.

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 20.04.03 LTS. Participants are **strongly encouraged** to verify library-level compatibility with `g++` on Ubuntu 20.04.03 LTS **prior to** submitting their software to NIST to avoid unexpected problems.

4.2 Usage

4.2.1 Software Libraries

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

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

4.3 Validation and Submitting

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

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

4.3.1 Agreement

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

4.3.2 Communication

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

4.4 Speed

Timing tests will be run and reported on a fixed sample of the SlapSeg III dataset using 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 Section 3.3 will be rejected. Participants may resubmit a faster submission immediately with a new version number. To avoid the appearance of SlapSeg III as an algorithm speed-checking service, NIST may require that participants with excessive repeated failures exceedingly distant from published timing requirements wait several months before their next submission.

References

[1] Ulery B, Hicklin RA, Watson CI, Indovina MD, Kwong KK (2008) Slap Fingerprint Segmentation Evaluation 2004 Analysis Report. *NIST Interagency Report 7209* https://doi.org/10.6028/NIST.IR.7209

[2] Watson C, Flanagan P (2010) SlapSegII Analysis: Matching Segmented Fingerprint Images. *NIST Interagency Report 7747* https://doi.org/10.6028/NIST.IR.7747

[3] Watson C, et al. (2007) User's Guide to Export Controlled Distribution of NIST Biometric Image Software (NBIS-EC). *NIST Interagency Report 7391* https://doi.org/10.6028/NIST.IR.7391

Revision History

27 January 2022 Add `configurationDirectory` parameter to `getImplementation()` (Section 3.3.1). Change operating system to Ubuntu 20.04.03 LTS from CentOS 7.6 (Section 4.1.6). Synchronize many clauses in Section 4 with other NIST evaluations.

13 February 2019 Final revision posted.