Towards Repeatable, Reproducible, and Efficient Biometric Technology Evaluations

Gregory Fiumara
gregory.fiumara@nist.gov

Wayne Salamon
wayne.salamon@nist.gov

Craig Watson
craig.watson@nist.gov

National Institute of Standards and Technology
100 Bureau Drive, Gaithersburg, MD 20899-8940

Abstract

With the proliferation of biometric-based identity management solutions, biometric algorithms need to be tested now more than ever. Independent biometric technology evaluations are needed to perform this testing, but are not trivial to run, as demonstrated by only a handful of organizations attempting to perform such a feat. Worse, many software development packages designed for running biometric technology evaluations available today shy away from techniques that enable automation, a concept that supports reproducible research. The evaluation software used for testing biometric recognition algorithms needs to efficiently scale as the sample datasets employed by researchers grow increasingly large. With better software, additional entities with their own biometric data collection repositories could easily administer a reproducible biometric technology evaluation. Existing evaluation software is available, but these packages do not always follow best practices and they are lacking several important features. This paper identifies the necessary requirements and ideal characteristics of a robust biometric evaluation toolkit and introduces our implementation thereof, which has been used in several large-scale biometric technology evaluations by multiple organizations.

1. Introduction

Independent biometric technology evaluations are an important task, given the widespread applications of biometric technologies. Evaluations of core accuracy and functionality of biometric recognition algorithms help to identify limitations and reveal the current state of the art. True one-to-many evaluations can prove to be especially challenging, as this increases the central processing unit (CPU), storage, and analysis demands on the host, as compared to one-to-one evaluations. More demand means more cost. Many researchers have insisted on *large-scale* biometric technology evaluations, employing datasets with sample counts in the multi-millions, adding to the already steep requirements. Recently, only a handful of organizations have undertaken public biometric technology evaluations, such as the University of Bologna (FVC) [2], the National Institute of Standards and Technology (NIST) (FpVTE, FRVT, IREX) [18, 3, 4], and the Idiap Research Institute (BEAT) [5].

Unfortunately, none of the evaluation software behind these biometric technology evaluations has been made available. In a controlled scientific evaluation, recognition algorithms serve as the *software under test*, or independent variables, while the *evaluation software* serves as the unchanging experiment procedure. Without a reproducible procedure, all the public is presented with are irreproducible and non-comparable results. Even when recreating a published evaluation using identical data (*e.g.*, public datasets), a technology experiment cannot be truly reproduced unless the evaluation software driving the data consumption is also public [6]. Evaluation software errors as obvious as the incorrect coding of a published analysis equation or as subtle as the premature truncation of a biometric template will cause vast differences in the reported accuracy of a recognition algorithm, yet these errors are untraceable without published source code. Much existing free and open source evaluation software lacks functionality to enable repeatability or relies on unscalable and inefficient programming techniques that are better suited for small-scale proof-of-concept experimentation. They have major limitations that prevent use in running a large-scale evaluation.

The purpose of this paper is to identify evaluation software as a significant gap in biometric testing, and propose a free and open source framework for aiding and enabling biometric technology evaluations, regardless of biometric data format, modality, size, or application (*i.e.*, verification or identification). The proposed software, the *NIST Biometric Evaluation framework*, has been integrated with several biometric technology evaluations and other projects at var-

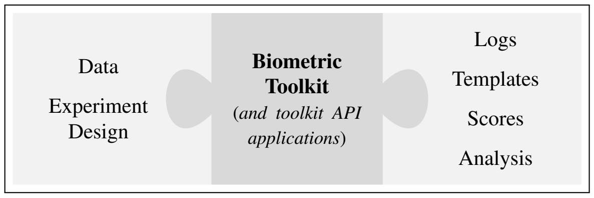

Figure 1. A biometric toolkit is a piece of the biometric technology evaluation puzzle that focuses on driving the experiment and providing reliable, repeatable, and efficient outputs. Many other systems focus on experiment design or data organization.

ious organizations, improving on existing methods outlined in Section 2. It does *not* aim to be a replacement for designing and controlling experiments. There are many existing solutions for these complex problems and tackling them is beyond the scope of this paper. Instead, a framework, or "toolkit," should simply be a collection of building-blocks used by an application to *drive* an experiment, which may in turn import any number of experiment organizational strategies (Figure 1). A toolkit need not require low-level systems administration or advanced development knowledge. Any significant aspects that may be required in writing a biometric evaluation application, such as image decompression and biometric file format parsing, should be included in the toolkit, so that a researcher does not need to maintain a collection of third-party dependencies and learn their associated syntaxes. It should be compatible with the best practices of the industry, simple for new research and development collaborators to comprehend, and difficult to misuse during implementation. Preferably, it should be written in a portable programming language for use on multiple operating systems, lack complex dependency chains, and be compiled to machine code for speed and efficiency. Finally, it should be as fool-proof as possible. Let the toolkit handle the intricacies of maintaining the infrastructure, so that the researcher can get to running an experiment and analyzing its results without spending time debugging software issues. The job of the biometric researcher is to research biometrics, not to write evaluation applications, so biometric toolkits make the evaluation application writing process simple. Not needing to remember specific evaluation steps and techniques leads to complete *automation*, which enables consistency and reproducibility.

The remainder of this paper is organized as follows. Some existing biometric evaluation systems are detailed in Section 2. The importance of creating CPU-bound evaluation applications and ideas on efficiently distributing data are outlined in Section 3. Commonly missing evaluation pieces are listed in Section 4. Application Programming Interface (API) design is addressed in Section 5. Source code consistency and simplicity is discussed in Section 6. Our implementation of a biometric toolkit along with some of our future work is revealed in Section 7. Finally, the features of a biometric toolkit are summarized in Section 8.

2. Existing Work

Several organizations have developed software to help evaluate biometric algorithms. A system employed to power biometric research at a large organization was examined. The organization connects hundreds of campus computers together, forming a compute grid and storage cluster, both of which are, by definition, highly scalable. However, this system has deep dependencies on copying files around a distributed file system, whose limitations in biometric technology evaluations are described in Section 3.2. Other systems provide a highly-generic plugin architecture for adding new algorithms, along with a grammar for stringing those plugins together. This modular system's API is closer to a viable solution, but lacks many of the building-blocks that prove helpful in running a large-scale biometric technology evaluation. Some organizations sponsor web-based biometric technology evaluations. The toolkits powering their evaluations run biometric algorithms in an automated fashion, but the server-side code is not necessarily available to the public, and therefore lacks the ability to create truly reproducible research. Similarly, at least one U.S. Government research organization has created their own evaluation software, but has only delivered subsets of the code to participants of their evaluations for use in implementing an evaluation's API.

3. Always Be Computing

3.1. Keep RAM Reliable

A biometric toolkit should provide some sort of smart memory object to circumvent frustrations of low-level memory allocation. In lower-level languages like C and C++, manually managing memory and passing around pointers and references to memory locations is commonplace. By passing memory locations to other functions instead of copying and duplicating data, applications eliminate unnecessary data swapping and become more efficient. A common complaint is the complexity of code created to enable this efficiency, as seen in Figure 2 [11]. It's quite easy to leak memory over the course of an evaluation or even corrupt memory through invalid pointer math. A smart memory object used throughout a biometric toolkit would help alleviate many of these problems, removing a large area of complication for both researchers and algorithm developers.

3.2. Intelligent I/O

A biometric toolkit should provide an abstraction for all input/output (I/O) operations (*e.g.*, reads/writes to files on disk). Possibly the single greatest bottleneck in a biometric technology evaluation (or any other computing task) is I/O. Even with the proliferation of solid state media, persisting

```
/* Stop doing this ... */
struct stat sb;
if (stat("img00001.png", &sb) != 0)
        return;
uint64_t bufferSize = (uint64_t)sb.st_size;
uint8_t *buffer = (uint8_t *)malloc(sizeof(uint8_t) *
    bufferSize);
if (buffer == NULL)
        return;
FILE *fp = fopen("img00001.png", "rb");
if (fp == NULL) {
        free(buffer);
        return;
}
if (fread(buffer, 1, bufferSize, fp) != bufferSize) {
        free(buffer);
        return;
}
free(buffer);

/* ... and start doing this. */
auto buffer = IO::Utility::read("img00001.png");
```

Figure 2. Efficient but verbose C code, and equally efficient but more concise C++ 2011 code using an object to manage memory. A biometric toolkit provides a memory object that encapsulates all the complexities of memory management and presents an effortless API to the developer.

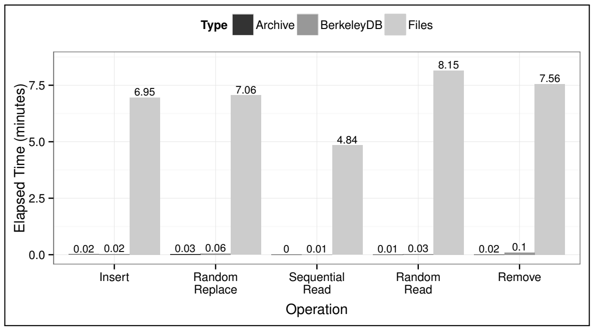

Figure 3. Summary of I/O performance of various file formats on an NFS-mounted EMC Isilon Network Attached Storage device (lower is better). Each operation was performed with 110 503 records of 1 153 bytes each. These prime values were chosen to prevent any potential optimization. While other file systems and devices will certainly produce different results, the values presented here are crucial for understanding decisions made in choosing I/O abstractions for evaluations running on *this* hardware.

data to disk is much slower than to RAM. As such, it behooves one to minimize the amount of interaction with a file system and maximize the amount of data passing in memory (Section 3.1). When files must be utilized, a biometric toolkit should use optimal structures and techniques for permanent data storage. Biometric toolkits must make every effort to keep their applications CPU-bound and not I/O-bound for the sake of the overall evaluation runtime. Figure 3 demonstrates how dramatically the runtime of an operation can differ based solely on the operation's I/O strategy.

Relational databases are a common alternative to individual files for storing biometric data. It is easy to issue queries to select subsets of data, but hosting blobs of data in a relational database can prove inefficient and database maintenance can be complex [16]. Systems have traditionally chosen a hybrid approach, where metadata is stored in a relational database, along with a file path to the actual data. This hybrid compromise is useful, but hosting individual small files on disk is a pain point for many file systems. In addition to the large number of input files, biometric technology evaluations generate millions of small files in the form of processed images and biometric templates. In most file systems, managing an extremely large number of files is inefficient, leads to longer application runtimes, and adds complexity to storage archival [20]. As seen in Figure 3, performance observed while reading and writing individual files is abysmal, even when compared to simple key/data flat file formats like BerkeleyDB.

To counteract these issues, all reads and writes should occur from a convenient high-level API. Whatever the most optimal I/O technique is, the I/O API handles it in the most efficient manner, acting as a self-contained entity, managing data on the application's behalf. If the storage backend ever changes, the evaluation code written by the researcher does not need to change. This abstraction allows an application to use the same code that reads and writes from samples on a researcher's laptop hard drive to run identically on a multi-petabyte fibre-connected storage area network.

When utilizing multiprocessing techniques in a biometric technology evaluation (Section 3.3), a shared file system should be used to avoid excessive copying. Consider Hadoop, a popular distributed computing software package [1]. In the Hadoop Distributed File System (HDFS), input data is broken down into blocks that are stored locally on a number of compute nodes. The `Reduce` function of the Hadoop job is performed, and the derivative data is written to HDFS. In a biometric technology evaluation, this derivative data (*i.e.*, biometric templates, logs, etc.) persists via a large number of small writes, but eventually needs to coalesce for processing into a larger enrollment set or for analysis. While Hadoop is a great solution for many large-scale projects, HDFS may prove inefficient in some environments, as hundreds of gigabytes of data will be constantly swapped around compute nodes, and the many small writes create a huge performance penalty on the file system [12]. In these cases, it is better to set up a shared file system to deal with data persistence.

3.3. Multiprocessing

A biometric toolkit should provide a simple abstraction for parallelizing a biometric operation (*e.g.*, feature extraction and template matching) across one or more heterogeneous compute nodes. In order to complete a large-scale biometric technology evaluation in a reasonable amount of time, it becomes absolutely critical to utilize parallelization techniques. At a basic level, applications must be able to support batch processing, either through an API or by accepting a series of inputs. When evaluating a large number of samples, it is inefficient to spin up an application, process a single operation, and shut it down, as has been done in previous biometric technology evaluations. While this ensures that no learning techniques are used, it extends the runtime, forces crude scripting loops, and prevents any possibility of automated parallelism.

In a large-scale biometric technology evaluation like NIST's FpVTE, up to 5 million subjects needed to have templates generated to form an enrollment set, over which hundreds of thousands of searches were performed. Using a single process on a single machine would have added dozens of *years* to the evaluation's runtime. Instead, NIST used Message Passing Interface (MPI) [17] to control the runtime and manage data flow, enabling parallelization.

Writing MPI code is not an easy task, and biometric researchers would rightly balk at doing so. There are other multiprocessing techniques, but not without their own faults. Hadoop was previously mentioned, but is arguably equally as complex and its file system is largely inefficient for a biometric technology evaluation. POSIX threads are more easily written, but only allow for parallelization on a single compute node and have the additional requirement that the biometric algorithm be thread-safe. Cluster computing software could certainly be used, but only by organizations with large quantities of hardware.

A biometric toolkit can simplify enabling parallelism by abstracting parallelization techniques. Given a list of compute nodes, the toolkit can create a software *distributor* process that figures out how many *worker* processes to spawn, given the available RAM and CPU cores. To prevent idle cycles, the *distributor* would segment the work to be performed into small pieces and let individual *worker* processes ask for more when their data segment has been exhausted. The biometric toolkit's parallelization abstraction must execute one process on a single compute node in the exact manner that it executes p processes on n compute nodes. That is, there should be no change in development from a small experiment on a laptop to an automated biometric identification system deployment in a data center. The evaluation's results must be identical regardless of the environment.

The evaluation administrator needs a way to interact with the large collection of processes that have been created by the biometric toolkit's parallelization facilities. The biometric toolkit should enable establishing a remote connection to the distribution center of a parallel job. The *distributor* would listen for commands, such as, "count remaining samples," "pause for 30 minutes," or "checkpoint and stop."

4. Necessary Components

4.1. Logging

A biometric toolkit should have a reliable method for logging. After running a large-scale biometric technology evaluation, a researcher must analyze the results. While many evaluation systems focus on the reproducibility of the experiment design (*i.e.*, BEE), none place emphasis on the persistence of the outcome. Analysis reproducibility is exceptionally important, but it is far from the only records that must be kept from a biometric technology evaluation. Just like I/O processing (Section 3.2), logging is another toolkit aspect that utilizes an abstract strategy and familiar coding conventions in its API. The same code that prints to the console on a researcher's laptop during development will need to run transactional `INSERT`s to a relational database when running in production.

Logging needs to take place in a consistent manner, which will require limited standardization among biometric technology evaluation APIs (Section 6.1). Method return codes should mean the same thing in every application. Timing information should be recorded in consistent units. Failures to enroll and other biometric errors must be handled in the same way. Consistency in the logs amounts to an easier time performing analysis, additional opportunities for automation, and a more uniform reporting of accuracy across evaluations.

A toolkit must implement its logging methods in such a way that confirms the integrity of the commit. If an entry fails to be written, the application may need to cease operation, as there is no point in wasting compute cycles evaluating an algorithm if the output cannot be recorded.

Log files are not just processed by analysis scripts, they must also be human-readable. The toolkit may prefer to write delimited text logs so that the logs are parsable by the researcher during development and easily sequenced over during analysis. Consider an algorithm that is failing to enroll every image in a 5 million image batch processing job. A human could open this log file, see the errors (thanks to consistent return code values), and stop the job. Better still, if a `syslog` logging strategy were being utilized, a rule could be set to trigger an alert to the researcher to stop the job after a customizable error threshold had been achieved.

4.2. Data Access Abstractions

A biometric toolkit should encapsulate processing of biometric file formats without the need for third-party dependencies. Websites like GitHub have popularized the availability of open source materials for a seemingly endless genre of disciplines, with biometrics being no exception [10]. The increasing number of small projects have taken a toll on the configurations of software build systems and made way for language-specific dependency managers. In a biometric toolkit, the many trivial pieces of code needed to perform basic tasks in a biometric system should be integrated into a shared framework, written in a common and cohesive style.

There are a number of standardized biometric data interchange formats [13, 8], and many options for parsing them, available in both open source and commercial products [19, 14]. These products often do more than just simply parse biometric files, and so an extremely large codebase that may be complex to configure and build now accompanies the evaluation software. The same is true for image formats. Nearly every biometric system needs to decompress both standard and biometric-specific image formats, but no single third-party distribution supports all formats with the speed and ability for fine-grained control that is needed in order to satisfy industry best practices [15].

The only external dependencies for a toolkit should be those provided by the operating system. It is not enough to simply incorporate properly licensed and attributed source code. It's also necessary to provide a cohesive style to the code, promoting software development best practices. The same code that retrieves fingerprint minutia from an ISO 19794-2 template should also retrieve minutia from an ANSI/NIST ITL Type 9 record and an INCITS 378 record. The same code that decompresses a Portable Network Graphics (PNG) file should decompress a Wavelet Scalar Quantization (WSQ) file.

A biometric toolkit should provide a common representation of encapsulated data. This includes enumerations for some data (*e.g.*, finger position, impression type, etc.) that are translated between file formats when necessary. The intricacies of dealing with third-party distributions should be abstracted away behind the dramatically simplified API provided by the biometric toolkit, returning data that interoperates effortlessly with other toolkit packages (Section 6.1).

4.3. Error Handling

Developers make plenty of mistakes, and a biometric toolkit should ship with utilities to detect, catch, and handle errors to prevent them from halting a compute node. In FpVTE, NIST reported on 36 total algorithms from 18 participants. Before the evaluation could be completed, participants were required to submit *validation* packages to ensure that algorithms were free of major errors (*e.g.*, crashes, memory leaks, time constraints, etc.) and that score generation was repeatable on NIST's systems. NIST reported that in order to get these 36 final submissions, they accepted 733 validation submissions. While error avoidance seems obvious, the extremely high error rate in FpVTE tells a different story that must be considered during an evaluation's technical design. A single troublesome image in a dataset of millions is enough to elicit errors from many algorithms.

When an application process executes in a POSIX environment, the operating system can generate signals to the process. In many cases, if the signal is not handled by the process, execution terminates (*i.e.*, "crashes"). Because many biometric technology evaluations are designed to be used with "black-box" software (*e.g.*, software for which no source code is available), the crashing code cannot be examined or changed by evaluation administrators.

A biometric toolkit provides smart memory management (Section 3.1), but cannot enforce its usage in "black-box" software to prevent memory leaks. On a large-memory system, detrimental effects of a slow leak, including crashes, may take days or even weeks to manifest, at which point, an enormous amount of compute cycles have been wasted.

A biometric toolkit can do many things to prevent an evaluation process from terminating due to "black-box" errors. Signal handling is a low-level task that should be abstracted away by a biometric toolkit, which can then be easily installed in an evaluation application. To catch memory leaks, memory usage can be monitored in a separate thread. This information can be used to report memory statistics of an algorithm in analysis, as well as to find a baseline memory consumption value and alert evaluation administrators if that value is greatly exceeded or on an incline.

4.4. Time Tracking

Timing facilities should be provided by a biometric toolkit. In their FpVTE biometric technology evaluation, NIST allowed participants to spend 90 seconds on average performing searches over the enrollment set. Participants likely wanted to get as close to 90 seconds as possible to have the highest accuracy, but NIST needed to limit the runtime of the evaluation. In addition to enforcing time limits, some algorithms can enter deep loops when extracting features that may not return for a very long time, if ever (in error conditions).

In both of these scenarios, a watchdog timer could be used to force an evaluation application to jump out of any block of code and return control to a specified function. A generous grace period would need to be used if time limits imposed are supposed to be averages, as opposed to maximums. The toolkit would need to provide abstractions for timing, as the low-level syntax for precise clock measurement is not standardized between operating systems. Theoretically, signal handling, watchdog timing, as well as function timing could all be performed in a closure (*i.e.*, an anonymous function object), as seen in Figure 4. Writing code like this lets the researcher focus on the batch processing that is taking place, while the biometric toolkit takes care of the messy enforcement of rules.

5. Uncommon Core

A biometric toolkit strives for simplicity—not only for the evaluation administrator, but perhaps more impor-

```
auto images = database->getImages();
auto e = EvaluationAPI::getFeatureExtractor();

/* For each input image ... */
std::for_each(images.begin(), images.end(),
    [&e](const Image &image) {
        /* Call extraction function wrapped in signal handler, */
        /* watchdog timer, function timer ...                  */
        auto result = callAPIMethodSafely(30 /* sec */,
            [&e, &image]() -> Result {
                return (e->getFeatures(image));
        });

        /* Check for errors from the closure. */
        if (result.errors.signalHandled())
                /* ... */
        else if (result.errors.timeExceeded())
                /* ... */
        else
                database->write(result);
    }
);
```

Figure 4. Calls to a biometric API should be wrapped with code that performs error handling and timing. In this example, every image in a database is passed to a feature extraction function wrapped in a closure that times the function call, catches signals, and ensures that the function call returns within a specified amount of time.

tantly for the evaluation participant. The BioAPI standard introduced a high-level generic biometric authentication model [7]. Primitive functions like `Enroll` and `Verify` are expected to fit every modality and type of algorithm. BioAPI and plugin-based systems are worthy pursuits, but not always practical. It would be impossible to satisfy the constraints of every participant's algorithm in an API, but a consensus must be achieved, which likely involves a different evaluation API for each evaluation type and modality.

Consider again NIST's FpVTE, which imposed a two-stage matching protocol. Modifying a plugin API to add two-stage matching would render existing plugins inoperable until they are updated, which might not be trivial. Likewise, changing the API layer of BioAPI would require the consensus of a standards body. In addition to differing protocols, it's useful for an evaluation API to make use of specific datatypes, as opposed to packing and parsing information as raw bytes. For instance, if attempting to pass pose information to a face algorithm, it's easier to pass an object containing yaw and pitch than it is to pack and unpack this information from a system-specific binary format. While some form of commonality is appropriate in evaluation APIs (Sections 4.1 and 6.1), a wholesale generic approach does not prove as useful.

The goal for any successful large-scale biometric technology evaluation is to have many participants. For commercial entities to participate, it must be simple to take an existing product and adapt it to the evaluation protocol. In the generic API case, this could be incredibly difficult, but the more the evaluation API maps into a commercial product's implementation, the easier it is for commercial entities to participate. Individuals, universities, and startups can benefit greatly by participating with a revolutionary algorithm, so evaluation APIs must be flexible enough to allow participation from these resource-limited organizations.

6. Predictability

With the goal of providing the easiest possible way to create an evaluation application with collaboration from others, a biometric toolkit should strive for consistency, even in trivial tasks. Even with many major concepts covered by the biometric toolkit (*e.g.*, error handling, multiprocessing, file formats, etc.), there are still many conveniences the researcher will need during the day-to-day writing of an evaluation application. In a compiled language such as C++, many tasks trivial in other modern languages (*e.g.*, tokenizing a string on a delimiter, reading a file, etc.) can be extremely arduous and consume many lines of code. A biometric toolkit should provide utility methods for many common tasks that need to be performed in an evaluation application.

There are several popular third-party C++ frameworks that provide such utilities. One goal of a biometric toolkit is to make evaluation writing easier for the researcher. While third-party frameworks like Qt and Boost provide some useful utility methods, they also provide a host of other methods that are not useful to the researcher. The researcher must then scour documentation (if available) to find beneficial content. This creates an additional external dependency, but more importantly, distracts outsiders from reading and comprehending the code. Consider Figure 5. Nearly any developer (C++ or Java, novice or expert) will understand a biometric toolkit's method for tokenizing a string, but the methods from Qt and Boost are more obscure and could cause confusion.

6.1. Internal Consistency

A biometric toolkit should be consistent within its own API, in order to make it simple to use and difficult to misuse. In Section 3.1, the concept of a smart memory object was presented. To be consistent, anywhere in the toolkit's API that a memory buffer is needed, the smart memory object proposed in Section 3.1 should be used. When dealing with durations of time, all API methods should use identical units. The result of a timing operation in one method should not need to be converted into different units in order to be correctly used in another method. If an error occurs within an API method, a common set of toolkit errors or exceptions should be used to report it. This all seems like common sense, but remaining consistent can be challenging as a codebase grows and outside collaborators commit changes. The toolkit should always appear as a well-designed API created by a single developer.

Section 5 proposed that a separate API be created specifically for each biometric technology evaluation. This API conforms to the same consistency measures of a biometric toolkit. Doing so allows an evaluation developer to interact with a "black-box" algorithm in the same manner that they would interact with their own toolkit code. This helps eliminate errors due to misuse of the software under test and implicitly enable efficiency gains on the evaluation administrator's hardware.

```
std::string str{"foo,bar,baz"};

/* Obvious to all developers */
auto tokens = Text::Utility::split(str, ',');

/* Qt-specific */
QString qstr(str);
QStringList qtokens = qstr.split(QRegExp("\\,"));
auto tokens = qtokens.toVector().toStdVector();

/* Boost-specific */
std::vector<std::string> tokens;
boost:split(tokens, str, boost::is_any_of(","));
```

Figure 5. Simple utility functions should be encapsulated into a biometric toolkit. This eliminates dependencies and creates a cohesive codebase that is easy for all developers to read, regardless of what (if any) frameworks they are familiar with. In this example, a C++ `std::string` is split on a comma delimiter in an ideal way, as well as with third-party frameworks.

7. The *NIST Biometric Evaluation Framework*

The *NIST Biometric Evaluation framework*, freely available from https://github.com/usnistgov/BiometricEvaluation, is a stable *work-in-progress* implementation of the proposed biometric toolkit. Even though the framework was written to support biometric evaluations, much of it is useful for general applications.

The framework is divided into several discrete packages, each providing a set of related functionality. The CORE package provides memory objects (Section 3.1), timing facilities (Section 4.4), and error handling (Section 4.3) across the entire framework. The IMAGE and VIDEO packages provide an abstract way to read many image and video formats, while the FACE, FINGER, and IRIS packages do the same for biometric data interchange formats (Section 4.2). Multiprocessing complexity is greatly reduced by conforming to the APIs provided by the MPI and PROCESS packages, and simple communication between processes is enabled with the MESSAGE package (Section 3.3). The IO package provides many I/O facilities, most notably for key/data persistence (Section 3.2) and logging (Section 4.1) abstractions. All of these packages adhere to a familiar calling structure and use CORE package objects (Section 6.1) for the ease and speed of implementation into applications.

NIST has used packages of the framework to power many substantial biometric technology evaluations, including FIVE, FpVTE, FRVT 2013, IREX III, MINEX, and PFT-II. Evaluations like FpVTE would not have been feasible without multiprocessing techniques, which were easily enabled with the framework. In other cases, rewrites of evaluation applications to use framework packages resulted in a vast reduction of code and dramatic performance increases, shaving off weeks of processing time per submission in the case of MINEX. Use of the framework is not limited to evaluations. It has been used for internal applications at NIST and other organizations.

7.1. Future Work

Many improvements can be made that enable even better usability in biometric technology evaluations. Section 3.1 describes the need for a smart memory object, which solves many, but not all, memory programming issues. Adding "guard bits" to this object would increase the usability of the toolkit, allowing it to automatically detect possible memory corruption before all work is lost in a crash.

Figure 1 shows that scores are an output of a biometric toolkit application. Since the toolkit is already logging these scores, it makes sense to have the toolkit bootstrap the analysis on these scores as well. A common set of basic analysis tools to generate consistent reporting across all modalities will enable faster publication of research while ensuring repeatable report generation.

ISO/IEC 19795-2 [9] provides standard guidance on biometric technology evaluations. While the toolkit merely powers a biometric technology evaluation application, safeguards and techniques can be added to the toolkit to steer applications toward 19795-2 conformance.

8. Summary

Running a reproducible large-scale biometric technology evaluation is no easy feat, but by using efficient software and techniques, some of the burden can be lifted. Unfortunately, existing public biometric evaluation system software does not completely satisfy the need of the evaluation administrator. RAM should be utilized as much as possible, to help keep evaluations CPU-bound instead of I/O-bound. Parallelization and multiprocessing techniques must be employed in order to churn through large quantities of data. Logging facilities must be consistent and reliable, while the output must be easily parseable for both automated systems and humans. Images, biometric file formats, and other common file structures need to be directly supported by the toolkit API when writing evaluation applications. Errors must be handled as soon as possible in an evaluation's validation submission cycle to prevent wasting resources. Evaluation- and modality-specific APIs should be used to allow for the largest number of interested parties to participate in the shortest amount of time. Code in a biometric toolkit must be as consistent as possible, from the most complex to the most trivial piece of functionality. All of these pieces must scale from a low-powered laptop to a multinode datacenter with ease. With all of these building-blocks in one place, it will become simpler for more individuals and organizations to run reproducible biometric technology evaluations.

Disclaimer: Certain commercial equipment, instruments, or materials are identified in this paper 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.

References

[1] Apache. Hadoop. http://hadoop.apache.org/. [Accessed: Apr 06, 2015]. 3

[2] B. Dorizzi, R. Cappelli, M. Ferrara, D. Maio, D. Maltoni, N. Houmani, S. Garcia-Salicetti, and A. Mayoue. Fingerprint and On-Line Signature Verification Competitions at ICB 2009. In *Proceedings International Conference on Biometrics (ICB), Alghero, Italy*, pages 725–732, June 2009. 1

[3] P. Grother and M. Ngan. Face Recognition Vendor Test (FRVT): Performance of Face Identification Algorithms. *NIST Interagency Report 8009*, 2014. 1

[4] P. Grother, G. W. Quinn, J. R. Matey, M. Ngan, W. Salamon, G. Fiumara, and C. Watson. IREX III: Performance of Iris Identification Algorithms. *NIST Interagency Report 7836*, 2014. 1

[5] Idiap Research Institute. Biometrics Evaluation and Testing (BEAT). https://www.beat-eu.org/. [Accessed: Apr 06, 2015]. 1

[6] D. C. Ince, L. Hatton, and J. Graham-Cumming. The case for open computer programs. *Nature*, 482:485–488, Feb 2012. 1

[7] ISO/IEC JTC 1/SC 37. *ISO/IEC 19784-1:2006: Information technology – Biometric application programming interface – Part 1: BioAPI specification*, 2006. 6

[8] ISO/IEC JTC 1/SC 37. *ISO/IEC 19794: Information technology – Biometric data interchange formats – Parts 1–14*, 2006. 5

[9] ISO/IEC JTC 1/SC 37. *ISO/IEC 19795-2:2007: Information technology – Biometric performance testing and reporting – Part 2: Testing methodologies for technology and scenario evaluation*, 2007. 7

[10] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The Promises and Perils of Mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 92–101, New York, NY, USA, 2014. ACM. 4

[11] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*, chapter 5. Prentice-Hall software series. Prentice Hall, 1988. 2

[12] X. Liu, J. Han, Y. Zhong, C. Han, and X. He. Implementing WebGIS on Hadoop: A case study of improving small file I/O performance on HDFS. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1–8, Aug 2009. 3

[13] R. McCabe and E. M. Newton. *NIST SP 500-271: Data Format for the Interchange of Fingerprint, Facial, & Other Biometric Information (ANSI/NIST-ITL 1-2007)*, 2007. 5

[14] Neurotechnology. Fingerprint BSS (Biometric Standards Support). http://www.neurotechnology.com/fingerprint-components.html#bss. [Accessed: Apr 06, 2015]. 5

[15] S. Orandi, J. Libert, J. Grantham, K. Ko, S. Wood, F. Byers, B. Bandini, S. Harvey, and M. Garris. Compression Guidance for 1000 ppi Friction Ridge Imagery, 2014. 5

[16] R. Sears, C. V. Ingen, and J. Gray. To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem. Technical Report MSR-TR-2006-45, Microsoft Research, April 2006. 3

[17] The Open MPI Project. Open MPI: Open Source High Performance Computing. http://open-mpi.org. [Accessed: Apr 06, 2015]. 4

[18] C. Watson, G. Fiumara, E. Tabassi, S. L. Cheng, P. Flanagan, and W. Salamon. Fingerprint Vendor Technology Evaluation. *NIST Interagency Report 8034*, 2014. 1

[19] C. Watson, M. D. Garris, E. Tabassi, C. L. Wilson, R. M. McCabe, S. Janet, and K. Ko. User's Guide to Export Controlled Distribution of NIST Biometric Image Software (NBIS-EC). *NIST Interagency Report 7391*, 2007. 5

[20] R. Wheeler. One Billion Files: Pushing Scalability Limits of Linux. In *Red Hat Summit and JBoss World*, 2011. 3