Fingerprint Abstraction Layer for Linux

Daniel Drake
dsd@cs.manchester.ac.uk

Supervisor: Toby Howard

May 6, 2008

School of Computer Science
The University of Manchester

Abstract

Fingerprint Abstraction Layer for Linux
Daniel Drake
Supervisor: Toby Howard
May 6, 2008

Fingerprint scanners are popular consumer computing products, but they are generally unusable under Linux-based operating environments. My project implements Linux support for a range of USB fingerprint scanners, allowing for fingerprint-based authentication capabilities. The source code for the software is released under open source licensing terms with the intention of building a community of users and developers.

The software is implemented as an abstraction layer with the aim of integrating into existing Linux desktop software environments. A focus on clean system design lead into areas of asynchronous system architecture and a side project implementing asynchronous USB I/O.

The project has successfully achieved wide hardware support and offers fingerprint-based authentication. The abstraction layer is suitable for integration into a range of applications. Basic levels of application integration were achieved, but further work is needed in order to provide a richer user experience.

Acknowledgements

This project was born from earlier open source projects and I'd like to thank everyone who has contributed to those efforts. I would like to specifically thank the following long list of people and companies who have made direct contributions to my work: Apricorn, Cyrille Bagard, Anthony Bretaudeau, Jan-Michael Brummer, Gustavo Chain, Covadis, Joaquin Custodio, Johannes Erdfelt, Markus Haindl, Timo Hoenig, James Klaas, Vasily Khoruzhick, Seemant Kulleen, Stanislav Lechev, Pavel Machek, Gerrie Mansur from Security Database BV, Bastien Nocera, Torkild Retvedt, Rossano Savino, Wolfram Schlich, Matthias Schmidt, Gavin Smalley, System76, TimeTrex, James Vasile of the Software Freedom Law Center, Tony Vroon, Craig Watson and others from the NIST Image Group.

I would especially like to thank Toby Howard, who kindly agreed to be the project supervisor and has provided unsurpassable support for my work.

Contents

Chapter 1

Introduction

1.1 Fingerprint scanning applications

The processing of human fingerprints is not a new area of research. Fingerprinting has long been used in specialist applications such as crime control.

The recent advancement of technology has lead to new applications for fingerprint scanning. Fingerprint scanning was once seen as a possibility for convenient *and* ultra-secure computerised authentication, but nowadays enough security concerns have been identified to scare people away from the high security promise.

Despite the shortcomings, companies found uses for fingerprint-based authentication based on the convenience aspect; it is trivially easy to scan your finger. Developers have realised its potential in situations where security is of noncritical importance, or where fingerprint scanning can augment an existing authentication system as an additional level of security.

Fingerprint scanning in the home may seem like a novel idea, but the development I find most interesting is that hardware manufacturers have found ways to produce the technology at such low prices that we see fingerprint scanners available on the market as consumer products. Major companies such as Dell, IBM and Microsoft integrate fingerprint scanners into laptops and input devices, and the success of such products has lead to millions of personal computer users having access to fingerprint scanners.

1.2 Software support

New computer hardware products must obviously be accompanied with software to be useful. As is common when new types of hardware enter the market, manufacturers tend to work independently and competing products have almost no degree of standardisation between them. These manufacturers tend to only offer support for the largest player in the operating systems market: Microsoft Windows.

My project targets the Linux operating environment. Although Linux's market share in the desktop space is only small, the industry is so huge that even the small players have many millions of users. Given the success of these hardware products and the rapid growth of the Linux desktop, there are a large number of Linux users who would like to be able to use their fingerprint scanners.

1.3 Fingerprint scanning on Linux

Having a personal interest in taking unsupported devices and making them usable in Linux-based operating environments, my involvement in fingerprint scanning was sparked in 2005 when Tony Vroon donated me an unsupported fingerprint scanner so that I could try to get it going on Linux. My spare time efforts progressed slowly but surely, and reached the point where I realised I was in range of my goal of making that device usable and useful on Linux.

During these developments, I became aware of the other widespread fingerprint scanning devices and other developers working towards making such devices usable. With the exception of one particular type of device, the short story is that there were no common fingerprint scanning devices usable in common Linux environments.

1.4 Project objectives

In addition to learning about the sorry state of fingerprint scanner hardware support on Linux, I realised that us software developers were making the same mistake as the hardware manufacturers: we were not working together. We shared common goals and even shared some common problems, but nobody was looking at the big picture of getting *all* fingerprint readers working on Linux through a standardised interface.

This leads to the primary objective of my project: to get fingerprint scanning hardware (in general) usable and useful on Linux, in a well-designed and well-integrated fashion.

The objective can be broken down into two parts:

1. Integrate fingerprint scanning into common Linux desktop applications.
2. Support a wide range of fingerprint scanning hardware.

In order to achieve application integration, I aim to make it easy for application developers to integrate fingerprint scanning into their work. Such developers should be shielded from the ugly implementation details of supporting the particular hardware that the user has plugged in today. Instead, they should just be presented with a generic high-level API for fingerprint scanning.

The goal of supporting a wide range of hardware calls for a driver-like model where the different devices are supported by individual drivers. Each driver is then brought in-line by presenting an identical interface to the upper layers.

The driver model combined with the aim of providing a generic high-level API leads to my design plan of developing an abstraction layer: each device that I choose to support will be backed by a driver, the driver interfaces are unified to a generic fingerprint scanning interface, and that interface is exposed to applications through a high-level public API.

1.5 Open source software

I proposed to release my project under open source software licenses and to build a community around the software. This makes sense considering that the project involves adapting pre-existing open source code, and also given that the target platform is entirely open source. In this report, I shall explicitly identify any code in my project that is based upon the work of others.

Successful open source projects tend to benefit from the “release early, release often” mantra, which I considered adopting for my project. For purposes of academic assessment, however, I decided to design and implement the core architecture myself before releasing anything, thus demonstrating my technical ability on my own project. I also intend to continue development beyond the academic project.

1.6 Report overview

- In Chapter 2, I discuss the targeted hardware devices and the state of previous open source fingerprint scanning software efforts.
- In Chapter 3, I detail the design of my abstraction layer and the accompanying components.
- In Chapter 4, I document the implementation of the first prototype of the software components.
- In Chapter 5, I discuss the implementation of the second prototype and an extended architecture.
- In Chapter 6, I present results obtained from testing my software components.
- In Chapter 7, I evaluate the results and the success of the project.
- Finally, Chapter 8 discusses the project achievements and future directions for the software.

Chapter 2

Research

2.1 Device support

Through prior involvement with fingerprint scanning efforts on Linux, I picked 4 different devices to support within my software. In this section, I shall identify the key characteristics of each device and document the state of the pre-existing open source efforts to get these devices operational.

2.1.1 AuthenTec AES2501

The AES2501[2] is a swipe-type fingerprint sensor. It is an imaging device which presents a series of 192x16 image slices to the host system. The host must then combine the slices, eliminating overlap and determining scan direction. The host software is also responsible for deciding when a finger has been pressed onto or removed from the sensor.

The AES2501 can be found embedded into many laptops manufactured by Fujitsu-Siemens and Hewlett Packard. It can also be found in some standalone devices and tablets. Even when embedded into laptops, this is a USB device accessible over the USB bus.

Previous open source efforts originated from Wittawat Yamwong, who determined certain aspects of the device protocol and released a project[35] capable of retrieving images from the device. Cyrille Bagard built upon Yamwong's efforts and released a Linux kernel driver[17] which was later rewritten[19] with the assistance of Vasily Khoruzhick. In this form, the driver is able to capture and assemble images but cannot be used for any form of fingerprint authentication. I build upon the work of Bagard, Khoruzhick and Yamwong in my project.

2.1.2 AuthenTec AES4000

The AES4000 is a press-type USB fingerprint sensor. It is an imaging device which presents a square fingerprint image to the host system. It has basic finger presence detection capabilities in hardware.

The AES4000 is available in the Targus PA460U DEFCON Authenticator[13], a USB peripheral device designed for fingerprint authentication.

I am unaware of prior open source efforts to get this device operational on Linux. In order to operate this device from my project, I make use of the AES4000 technical specifications that were once available from AuthenTec's website.

2.1.3 Digital Persona U.are.U 4000B

The U.are.U 4000B is a press-type USB fingerprint sensor which uses optics to capture a high-resolution impression of the fingerprint. It offers hardware-based finger presence detection and 2 image capture modes.

The U.are.U 4000B is available as a standalone product from Digital Persona and is also embedded in several Microsoft input products. Digital Persona also sell this sensor in module-only form which other companies have integrated into their own products.

This is the device that got me involved in Linux fingerprinting. I started the dpfp project[3] in 2005, and it slowly progressed to the point where the software could retrieve images from the device. The dpfp project did not offer any useful functionality such as fingerprint-based authentication. I build upon this code in my project.

2.1.4 UPEK TouchStrip

The UPEK TouchStrip is a swipe-type USB fingerprint scanner. The device includes a *biometric coprocessor*[31] which performs image processing in hardware; the USB-level command protocol is simple as a result. The device does not present images to the computer, rather it only presents a boolean result: access granted or access denied.

The TouchStrip is found in many laptops manufactured by Dell, IBM, Lenovo, and Toshiba. It can also be purchased standalone as the UPEK Eikon reader.

UPEK released software to make this device usable on Linux, but it is entirely proprietary. The installation process is complex and it does not integrate into the system well.

Pavel Machek examined the traffic generated by UPEK's driver and produced an open-source implementation[25] of the basic functionality. Timo Hoenig later adapted Machek's work into a more complete project, ThinkFinger[14]. ThinkFinger progressed far enough to be able to offer fingerprint-based system login. I build upon the work of Hoenig and Machek in my project.

2.2 Image processing

3 out of the 4 devices introduced in the previous section all are simple imaging devices where the hardware presents images to the host computer without making any interpretation of them. Software on the host system must then examine the images to decide whether to grant access to the user.

The image processing aspect of a fingerprint-based authentication system is a complex problem that alone could easily consume a whole 3rd year project. This problem is one of the biggest reasons why Linux support for fingerprint readers has been poor until now; most fingerprint scanners are just imaging devices, but we were lacking good software to actually process and interpret the images.

Given that the focus of my project is on designing and integrating an abstraction layer, I looked to use existing image processing solutions to overcome this problem.

2.2.1 NIST Biometric Image Software

NBIS[10] is a collection of software utilities for fingerprint image processing. It is developed and published by the image group at the US National Institute of Standards and Technology[8].

According to the NBIS user guide[16], the software was developed for the Federal Bureau of Investigation (FBI) and Department of Homeland Security (DHS). Chapter 1 of the user guide suggests that this software is used within the FBI for crime control, and within the DHS for border control. You are required to scan your fingerprint at the immigration desk when entering the United States, and I suspect that NBIS is being used there.

As NBIS was developed[9] by employees of the United States Federal Government in the course of their official duties, the software is “not subject to copyright protection and is in the public domain.” It is therefore suitable for inclusion in an open source project.

I chose to use NBIS to solve the image processing problem. In order to integrate NBIS with my software, I needed to understand the operation of the individual components.

MINDTCT

MINDTCT is the first NBIS component I use in my software. According to the NBIS website[10], MINDTCT is a minutiae detector which automatically locates and records ridge ending and bifurcations in a fingerprint image.

A *ridge* is a solid curvy line that appears on your fingerprint. Your fingerprint is essentially a collection of ridges with different sizes, shapes, and directions. Figure 2.1 shows how ridges tend to end and split (bifurcate) in various places.

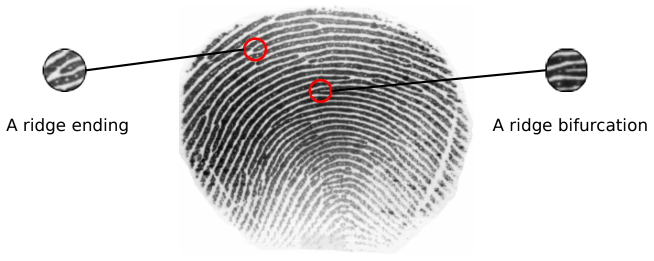


Figure 2.1: Ridge features

In fingerprint terms, a minutia is a point of interest. Ridge endings and bifurcations are regarded as minutiae.

To locate the minutiae, MINDTCT first takes the raw fingerprint image and applies a series of algorithms to it to process and enhance the image. The result is a *binarized* image; a 2-colour image consisting only of black ridge pixels against a white background.

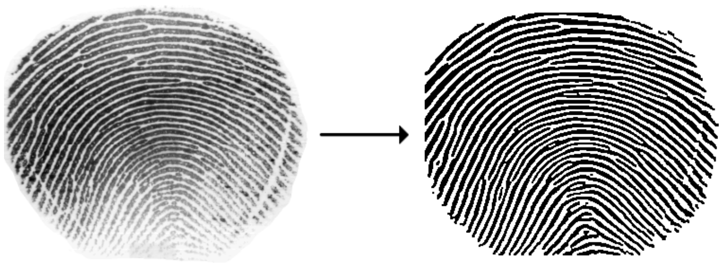


Figure 2.2: MINDTCT binarization results

MINDTCT then performs several simple scans of the image, searching for occurrences of several small predefined pixel patterns. Upon successful match against a particular pattern, MINDTCT records a minutia point in that location.

One disadvantage of the simple approach taken to locate minutiae on the binarized image is that many false minutiae are detected. To counteract these undesirable results, MINDTCT then applies a series of algorithms to attempt to identify and remove the falsely-detected minutiae such as minutiae detected in regions of poor image quality.

The final result is a collection of minutiae points that can be plotted on the fingerprint:

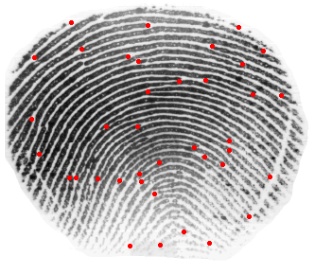


Figure 2.3: Detected minutiae plotted on fingerprint

The locations of the minutiae can then be stored for later use. In addition to the co-ordinate locations for each minutia, MINDTCT additionally records extra information such as the ridge direction, whether the point marks an ending or a bifurcation, and the algorithmic confidence of the reliability of that minutia.

BOZORTH3

BOZORTH3 is a fingerprint matching system which takes minutiae sets generated by MINDTCT and compares them for similarity. The combination of MINDTCT with BOZORTH3 essentially allows you to determine whether two fingerprint images are of the same finger or not, an obvious requirement for any kind of fingerprint-based authentication system.

According to the user guide[15], BOZORTH3 was originally developed by Allan S. Bozorth at the FBI with the aim of investigating the notion of a translation and rotation invariant algorithm for matching two fingerprints to each other. NIST then improved the matcher and integrated it into their NBIS distribution.

The BOZORTH3 matcher compares two minutiae sets, evaluates their similarity, and outputs a *match score*. A suitable threshold is then applied to the match score to decide whether the minutiae sets represent the same finger or not.

2.2.2 FVS/eFinger

It is worth mentioning the only other open source fingerprint image processing library I could find. FVS[28] is a collection of software utilities for fingerprint verification.

I spent time investigating FVS in my work that predated this project, and although the results looked promising at a glance, I found it to be unreliable: too many ridge structures were being destroyed during enhancement, and too many false minutiae were being detected. Additionally, the performance of the library was lacking; the processing and enhancement operations take a number of seconds of CPU time even on modern computers.

FVS lacks minutiae set comparison functionality, but another project, eFinger[29] was built from FVS and does include such code. Perhaps because of inaccuracies of the earlier FVS processing, I was unable to get any consistent results out of eFinger's matcher.

2.3 Summary

I have chosen 4 different hardware devices to support in my software. They are all quite different from each other. Some are press sensors, others are swipe sensors. Most are imaging devices, but one does image processing in hardware. The degree of hardware-based finger presence detection varies between all the devices. Hiding these vast differences within an abstraction layer is one of the most interesting technical challenges encompassed by my project.

3 out of the 4 devices rely on software for the heavy lifting of processing the data from the sensor. As my project is not an image processing project, external software is needed. I have selected NBIS to handle these tasks.

Given my accumulated existing knowledge about fingerprint scanning, I decided that I was ready to begin the design and implementation of my software after only this brief amount of research.

Chapter 3

Design

3.1 Public API design

My project aims to provide easy mechanisms for integrating fingerprint scanning into applications. A well-designed generic fingerprint scanning API ensures that developers can integrate my software without requiring an in-depth understanding of fingerprint scanning. I have designed the API with a view for simplicity, offering key functionality described in this section.

3.1.1 Enrollment

Enrollment is the training process of teaching the system what your finger looks like. During enrollment, the system asks the user to scan a specific finger one or more times, and the system trusts that the finger scanned was the one asked for.

After processing, the result is some *enrollment data* for each enrolled finger. This data encodes specific information about the fingerprint (e.g. co-ordinate locations of the minutiae) and is stored to disk for later use.

3.1.2 Verification

Verification is what most people think of when they think of fingerprint scanning. The user scans a previously-enrolled finger, and the system then performs a one-to-one comparison of the enrollment data against the freshly scanned fingerprint. Access is granted if the system decides the freshly-scanned finger matches the enrollment data, and is denied otherwise.

Verification is suitable for implementing a fingerprint-based logon system where the user enters their username, and then the system prompts for a previously enrolled finger (rather than a password).

3.1.3 Identification

Identification is similar to verification, but is where the freshly scanned fingerprint is compared to an entire database of enrollment data for a number of fingerprints. Identification determines which enrollment data entry matches the freshly scanned fingerprint. Identification can also reveal that the freshly scanned fingerprint is absent from the database.

Compared to verification, identification allows for more flexible authentication scenarios. For a single-user logon, identification allows you to implement a logon system where the user scans *any* of their enrolled fingers. Identification can also be used where the system has no knowledge of which operator is using the system; in authentication terms, you could think of it being able to replace both the username and the password.

3.2 Data storage

As identified in the previous section, storage and retrieval of enrollment data is required functionality for a fingerprint authentication system to implement enrollment, verification and identification. My design allows for two data storage/retrieval mechanisms.

The first interface aims at simplicity. To save a print, the application calls a function with a single parameter identifying which finger the print corresponds to (e.g. left ring finger). Loading previously saved data is an equally simple procedure. Internally, the storage mechanism is fixed: enrollment data is stored in a hidden data directory under the current user's home directory.

I also recognised that some applications will demand more flexible storage capabilities. They may wish to store enrollment data in a database, apply encryption, and so forth. My design provides additional functionality to access a binary blob representing an in-memory enrolled print, and to construct an enrolled print from a binary blob. This interface allows the developer to take the raw print data and store it using whatever mechanism suits them; the data can then be fed back into the software at a later time, and used for verification or identification.

3.3 Data compatibility

An issue arises from the design of supporting a range of devices within the software. Under the storage interfaces described in the previous section, it would be possible for an application to store enrollment data for a finger scanned using one type of fingerprint scanner, and then to attempt to use that enrollment data during verification against a scan from a completely different type of fingerprint scanner.

Without wanting to spend too much time examining compatibility of data from the various hardware devices, I decided to take the simplistic approach and make enrollment data from one type of device completely incompatible with all other device types (i.e. rejected by the software if attempted to be used in this way). After all, the imaging devices have varying sensor sizes and use different technologies for data acquisition; there are significant differences between the appearance of the prints that come back. We must also consider the enrollment data that comes back from the UPEK TouchStrip non-imaging device, the format of which is unknown.

I decided that the binary enrollment data format should include a header identifying which device type it came from. If that data is later loaded for use with a different device, the software is able to reject the request for incompatibility reasons. Similar metadata can also be represented in the file paths used by the simplistic home directory storage interface.

It should be noted that this design does not involve any loss of functionality for the end user. It is still possible to use all 4 device types for verification of a particular finger, provided the finger has been separately enrolled on all 4 devices.

3.4 Driver abstraction

My software design separates code specific to each supported device type into *driver* modules. It makes sense to have the drivers be as small and simplistic as possible, with all common code only present in the upper layers. In other words, drivers in an ideally designed system should offer access to the primitive operations of the corresponding devices, but not be required to do too much processing of input or results.

The primitive operations of the 3 imaging devices are:

1. Detect finger on sensor
2. Capture image
3. Detect finger removed from sensor

Considering the one remaining device (the UPEK TouchStrip), the primitive operations are:

1. Enroll finger
2. Verify finger
3. Identify finger

The TouchStrip device operations are identical to the 3 core operations described in Section 3.1. The TouchStrip driver will therefore be able to directly feed results into the device abstraction layer.

On the other hand, the imaging device operations are distanced from the core operations and require an image processing layer with intermediate logic to bridge the gap. This leads to a multi-layered design where the standard interface is of a driver offering high-level functionality for enrolling and verifying, and if a device driver is unsuitable for that model then additional layers are added to bring it into unison with the others. The TouchStrip driver will plug in as a ‘primitive’ driver, but the imaging drivers will instead plug into an imaging layer that will convert them to fit the primitive driver interface. This architecture is shown in Figure 3.1.

3.5 Device discovery

The 4 targeted devices are all USB devices and I do not know of any fingerprint scanners that are connected through other means. This is quite convenient as it means no further abstraction is needed for accessing devices over different buses.

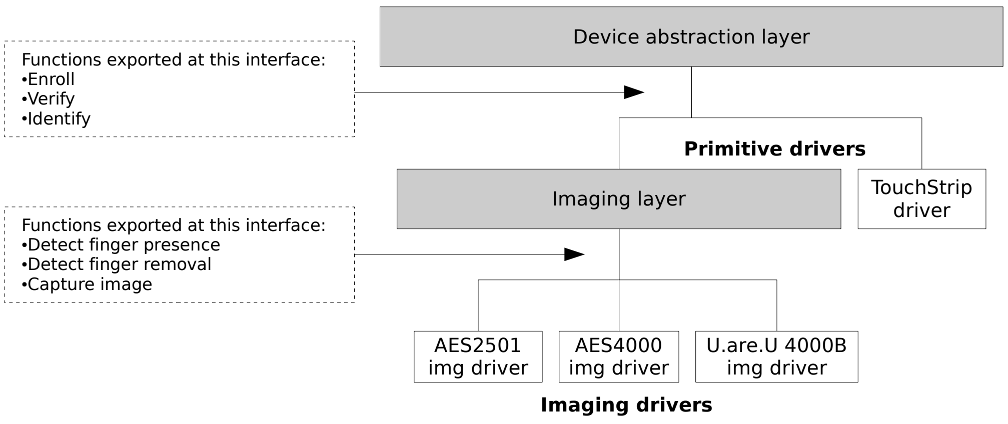


Figure 3.1: Driver abstraction model

All USB devices identify themselves and their capabilities by presenting a device descriptor to the host system. The device descriptor includes a vendor ID and a product ID which are fairly self-explanatory: each vendor has its own vendor ID, and each vendor assigns a product ID (unique to that vendor) for each different product.

Under my design, each driver exports a list of supported device `[vendor ID, product ID]` pairs. The upper layers can then scan the host's USB buses for devices, handing off supported devices to the appropriate drivers.

Chapter 4

First prototype implementation

4.1 Objectives

After completing my research and thinking through the design issues, I was left with a concern. My design involves combining various incomplete and incompatible device drivers, some image processing code, and my self-designed abstraction layers; will these components produce satisfactory results when combined into a library?

I was eager to find out, so I decided to take the prototyping approach and produce an initial implementation with the aim of demonstrating feasibility of my ideas, leaving other issues to be ironed out later.

4.2 Implementation choices

4.2.1 Language

I chose to implement my abstraction layer in C for the following reasons:

- Although it may not be classed as a low-level language, the fundamentals of C are quite primitive, and as such, it is a good choice of language for writing device drivers.
- Application integration is one of my goals. Developing the abstraction layer in C helps here as C is implicitly compatible with most higher level languages; bindings can be created on top of a C library for C++, Java, Python, etc. If I instead implemented the abstraction layer as Java classes, for example, it would be tricky to access my abstraction layer from a Python application.
- Although it falls outside of the primary goals, fingerprint scanning on embedded systems is a likely focus for the future. Software systems written in C are ideal for embedded environments because C code is compiled to native binaries which do not have special run time requirements.
- It is my personal language of choice for building low-level components and systems with considerable architecture. I am comfortable with the language and the surrounding tools; I have written and debugged a lot of C code.

I chose to implement my abstraction layer as a library, compiled into a Dynamic Shared Object (DSO). Applications wishing to use my abstraction layer will link against the library at compile time, and the run-time linker will complete the dynamic linking once the program is executed.

To fit naming conventions, I named my library 'libfprint.' Any references to libfprint in the remainder of this report refer to the abstraction layer implemented as a shared library.

4.2.2 Dependencies

libfprint depends upon two additional libraries at both compile time and run time:

1. **libusb**, a shared library which allows you to perform USB device I/O from userspace. This avoids having to write kernel-mode drivers for the fingerprint readers, and additionally paves the road for cross-platform compatibility; libusb is portable to FreeBSD, Mac OS X, Windows, and more.

2. **glib**, a shared library providing miscellaneous utility functions. You might view glib as a C equivalent of C++'s standard template library (STL). glib was chosen for the convenience of having quick access to linked lists, specialised memory allocators, string handling functions, and so on.

4.3 Abstraction implementation

4.3.1 Device handling

libfprint is implemented as a device-centric model. Applications locate a device they wish to operate, open it, and then use it to start operations such as enrollment.

libfprint offers a `fp_discover_devices()` function call which scans the system for supported fingerprint scanners and returns them in a list of `fp_dscv_dev` structures. libfprint is able to produce a list of supported devices by checking USB device ID's as described in Section 3.5.

To start using a specific device, the application must open it by passing the selected `fp_dscv_dev` structure to the `fp_dev_open()` function. This function returns a device handle in the form of a `fp_dev` structure. Devices are closed after use with the `fp_dev_close()` function.

4.3.2 Core operations

Armed with a `fp_dev` structure, applications can then call functions to perform the core operations introduced in Section 3.1:

1. **Enrollment.** The `fp_enroll_finger()` function takes a device, performs enrollment, and upon success, returns some enrollment data in the form of a `fp_print_data` structure.

2. **Verification.** The `fp_verify_finger()` function takes a device and some enrollment data, performs a fingerprint scan, and returns a status code indicating whether the scanned finger matches the enrollment data.

3. **Identification.** The `fp_identify_finger()` function takes a device and a gallery (an array) of enrolled prints, performs a fingerprint scan, and returns the offset into the gallery where a matching print was found.

Variants of all 3 functions exist which additionally return images of the scanned finger, if available. Imaging is not exposed at the primitive driver level, but I decided to bubble up images through that layer anyway. Many applications will be interested in displaying any available images, and they should not have to care whether the device in question fits into the primitive layer or the imaging layer. The less desirable alternative would be to have another 3 functions which only work for imaging devices, forcing such applications to implement logic such as:

```
if (is_imaging_device(dev))
    fp_enroll_finger_img(&data, &img);
else
    fp_enroll_finger(&data);
```

4.3.3 Data storage

Section 3.2 detailed my design ideas for storing of enrollment data. I implemented a function, `fp_print_data_get_data()`, which takes some enrollment data returned from the enrollment function and returns a binary representation of that data. Applications are not intended to make any interpretations of this data blob. The format is as follows:

- A `FP1` header indicating version 1 of fprint enrollment data
- A driver ID describing which driver produced the enrollment data
- A device type describing which type of device produced the enrollment data
- The enrollment data itself (e.g. NBIS minutiae for imaging devices)

I then implemented a `fp_print_data_from_data()` function which takes binary data previously returned by `fp_print_data_get_data()` and returns a corresponding `fp_print_data` structure. To implement the data compatibility ideas discussed in Section 3.3, the verification and identification functions check that the driver ID and device type identifiers match the device that is performing the scan.

The more simplistic data storage interface is implemented around these functions. The `fp_print_data_save()` function saves some enrollment data to the user's home directory, and the `fp_print_data_load()` function retrieves some previously saved data. Both of these functions take an additional parameter indicating which finger the print corresponds to (e.g. right little finger). The data is stored in the binary format described above at location `.fprint/prints/<driver ID>/<device type>/<finger number>` within the user's home directory.

Enrollment data saved with the simple interface can be located with the `fp_discover_prints()` function, and then opened with `fp_print_data_from_dscv_print()`.

4.3.4 Primitive driver model

Section 3.4 introduced the idea of a top-level driver abstraction layer, where each driver presents the core fingerprint scanning operations. Drivers that implement the primitive model provide a `fp_driver` structure including the following information:

- A statically assigned unique driver ID
- The name of the driver
- A list of supported USB device IDs
- A function to initialise a device
- A function to deinitialise a device
- A function to run an enrollment session
- A function to run a verification session
- A function to run an identification session

Internally, libfprint keeps track of which driver is backing each device. Implementation of the public API is therefore quite simple; for example, the implementation of `fp_verify_finger()` just looks up the driver for the specified device and invokes the verification function supplied in the corresponding `fp_driver` structure.

4.3.5 Imaging driver model

Section 3.4 also discussed the need for a separate driver abstraction layer for imaging devices. Drivers that implement the imaging model provide a `fp_img_driver` structure including the following information:

- All of the information from the `fp_driver` structure except the enroll/verify/identify functions
- The width and height of images presented by this driver
- A function to await a finger-on-sensor event
- A function to await a finger-removed-from-sensor event (optional)
- A function to capture an image

Observe that the imaging driver model is distanced from the concept of fingerprinting (with the exception of finger presence detection): there is no notion of verification or identification or how else the images might be used. Being common to all drivers, all image processing code is present in the upper layers and is not duplicated.

The imaging layer converts each `fp_img_driver` to a `fp_driver` structure and presents it to libfprint. The imaging layer provides generic implementations for the enroll/verify/identify operations which are expressed by a series of calls to driver-supplied `fp_img_driver` functions. For example, enrollment is implemented as follows:

1. Call the driver's await-finger-presence function
2. Call the driver's image capture function and retrieve an image
3. Call the driver's await-finger-removal function
4. Process the image with MINDTCT
5. Return a minutia set as enrollment data

4.4 Driver implementations

Except for where significant problems were identified and solved, discussion of driver performance will be left until Chapter 6.

4.4.1 UPEK TouchStrip

Section 2.1.4 identified ThinkFinger[14] as a suitable basis for a UPEK TouchStrip driver. When I came to implement a libfprint driver based on this code, I realised that the bus protocol was only loosely understood. The ThinkFinger code is difficult to follow because of the way it jumps around a lot.

I spent some time analysing the bus traffic in more detail, and identified various patterns in the messages. This allowed me to implement a more readable driver with a straightforward flow. Attention to detail revealed some questionable behaviour inside ThinkFinger; for example, each incoming message ends with a checksum, and in one case, ThinkFinger looks at the checksum value to determine the message meaning. Although this doesn't cause problems, it is clear that ThinkFinger should instead be looking at a status byte that appears within the message content.

The bus protocol for enrollment involves the driver sending a specific command sequence to begin enrollment, then repeatedly polling for result codes. Result codes indicate good quality scans, bad quality scans that need retrying, enrollment completion, etc. When enrollment completes, the driver receives a blob of approximately 200 bytes of enrollment data from the device. The driver presents this to the upper layers in the form of a `fp_print_data` structure.

The bus protocol for verification involves uploading the enrollment data recorded earlier, and then polling for status codes while the user scans their finger to determine the verification result.

Identification, although believed to be possible[20], is unimplemented as that part of the protocol remains unknown.

4.4.2 Digital Persona U.are.U 4000B

Being the author of the project where the device protocol was determined, I implemented the U.are.U 4000B imaging driver with ease.

The U.are.U 4000B is a mode-driven device. There are modes for waiting for finger presence, waiting for finger removal, and image capture. The implementations of the `fp_img_driver` finger presence detection and image capture functions simply set the appropriate mode on the device and wait for any appropriate interrupts or data stages.

The images returned from the device are standard 8-bit greyscale images and do not require special assembly or processing.

4.4.3 AuthenTec AES4000

The AES4000 driver was implemented based on device specifications. Initially, I had problems with poor image quality (too many ridges being melted together) but I managed to tweak the sensitivity registers to improve this. Ideally, the driver would evaluate the the image data and calibrate accordingly, but I fixed some reasonable values in order to save time.

Implementing finger-on-sensor detection was easy because this is directly supported by the hardware. Image capture required careful interpretation of the data format; the image pixels are packed into 3-bit nibbles, two per byte, and arranged in a confusing order. The device specifications provide enough information to be able to decode the data into standard greyscale images which can be presented to libfprint's imaging layer.

Problems were encountered during the processing of images from the AES4000. The device sensor is only small and MINDTCT was chewing up the small images, as shown in Figure 4.1.

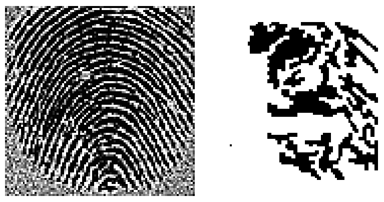


Figure 4.1: AES4000 fingerprint image and binarized version

Page 48 of the NBIS user guide[16] states that the MINDTCT algorithms and parameters have been designed to optimally process images scanned at 500 pixels per inch, whereas this sensor has resolution of less than half that, at 213 PPI. I decided to try artificially enlarging the image prior to processing. Figure 4.2 shows the vast improvement in binarization results.

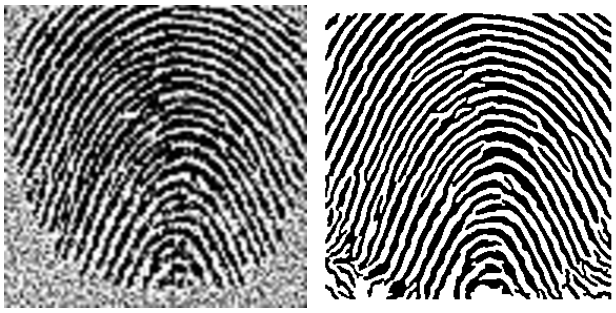


Figure 4.2: Enlarged AES4000 fingerprint image and binarized version

4.4.4 AuthenTec AES2501

I based my AES2501 driver on previous work researched in Section 2.1.1. After studying the code from previous projects, implementing my own driver was relatively painless.

Despite a lack of hardware-based finger detection capabilities, the device does offer an alternative capture mode which simplifies this task. In this mode, the device returns a histogram of the image captured at that moment, without transferring image data. The driver can apply some simple calculations to the histogram data to calculate its area. A suitable threshold is then applied to implement detection of finger presence and removal.

Image capture is more complicated than other devices. The device returns a series of 192x16 strip images, encoded in the same unusual format as the AES4000 images. Figure 4.3 shows the strips from a sample scan decoded and appended to each other on a grid.

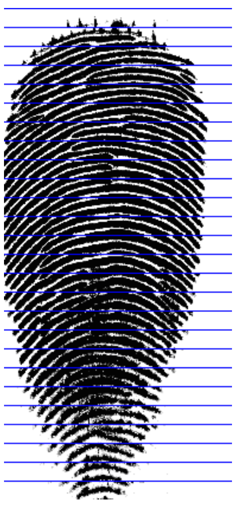


Figure 4.3: AES2501 scanned strips shown on a grid

The driver then eliminates overlap between the strips using a simple algorithm. The resultant merged image is shown in Figure 4.4.

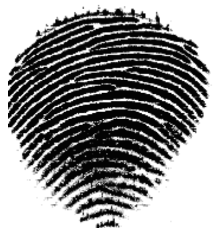


Figure 4.4: AES2501 scanned image after eliminating strip overlap

4.5 Additional components

4.5.1 pam_fprint: Pluggable authentication module

Most Linux desktop systems use a modular authentication system called Pluggable Authentication Modules (PAM). The most common setup consists of one user-visible module that asks for a password, but PAM can be configured to create more elaborate setups involving smart cards, network authentication, and more.

One advantage of PAM is that it is a centralised architecture with a documented API for authentication. Many applications use PAM for authentication, including the standard Unix login program (shadow) and graphical login managers such as the GNOME Desktop Manager (gdm).

As my first application integration step, I developed pam_fprint, a PAM module to provide fingerprint authentication through libfprint's functionality. The implementation can be described as follows:

1. PAM asks pam_fprint to authenticate a user.
2. pam_fprint uses libfprint to scan for available fingerprint scanners and a fingerprint enrolled with one of those scanners.
3. After locating suitable enrollment data, pam_fprint asks PAM to send a message to the user, asking that they scan a specific finger.
4. pam_fprint calls a function within libfprint to start verification.
5. When verification results arrive, pam_fprint reports authentication results to PAM.

4.5.2 fprint_demo: Graphical demonstration application

I developed a GUI application, fprint_demo, with the intention of using this to demonstrate the functionality of my abstraction layer at the assessed demonstration of results.

fprint_demo offers an interface for enrolling fingers and deleting enrollment data for previously enrolled fingers, an interface for verifying a specific finger, and an interface for identifying a finger from a selection of previously enrolled fingerprint data. For imaging devices, it shows the scanned fingerprints on-screen and allows you to view the binarized version with minutiae plotted.

fprint_demo proved useful during development as a testing tool, particularly for evaluating imaging performance and other scenarios where command line test utilities were inconvenient.

4.6 Further developments

After releasing my software to the open source community, further developments ensued.

- Khoruzhick contributed support for scan direction detection to the AES2501 driver. This allowed him to use the sensor embedded in his laptop, which appears to be mounted 180° rotated compared to other systems.
- Jan-Michael Brummer contributed a device driver for another UPEK device. The TouchChip can be found embedded in Samsung laptops.
- Anthony Bretaudeau contributed a device driver for the AuthenTec AES1610, a smaller sensor found in some laptops and tablets.
- Gustavo Chain contributed a device driver for the SecuGen FDU2000 scanner.
- I committed various bug fixes, feature enhancements required for fprint_demo development, image quality improvements for the AES2501 driver, and some code cleanups.

4.7 Deficiencies

The software components forming my prototype implementation successfully dispelled my concerns about feasibility; they were operating nicely. Nevertheless, development drew my attention to 3 main deficiencies which I shall explain in this section.

4.7.1 Synchronous I/O

The most significant design flaw was the lack of an asynchronous interface. The fingerprint scanning functions are all entirely synchronous. To illustrate this further, consider this example:

```
r = fp_verify_finger(dev, enroll_data);
```

The above function call causes the fingerprint reader to be powered up. libfprint then waits until the user has scanned their finger, processes the results, compares the scanned finger to the enrollment data, and returns a verification result.

The important point here is that execution remains within this function for the entire length of time that it takes the user to scan their finger. The user might be away from the computer, meaning that it may potentially take hours for this function to return a value. This is said to be a *blocking* interface.

When calling into these blocking functions, the parent application effectively loses control of that thread for the duration of the function call. Undesirable behaviour will occur if this is a single threaded application that also handles other events. For example, the visual interface in a GUI application would be unresponsive when the main thread is tied up within a libfprint function call.

This issue could be circumvented within the application. The example GUI could create a dedicated thread for fingerprint scanning, leaving the main thread available to respond to GUI events while the fingerprinting thread is tied up within libfprint. Although this may avoid the problem, I feel that it is unacceptable to impose the requirement that any application with other event sources must create a dedicated thread for fingerprinting.

Another deficiency with the blocking interface is the inability to cancel ongoing operations. The user may wish to cancel the verification operation when the application is stuck within `fp_verify_finger()`, but there is no standard way to cancel an ongoing function call. A separate cancellation function would have to be added, invoked from a separate thread, and the original thread would have to check for cancellation. Again, this may be a workable approach, but I feel that it would lead to implementation headaches, and what's more, I feel that we can do better.

The deficiencies associated with the blocking interface can be easily seen within fprint_demo. When you request a fingerprint scan through performing enrollment/verification/identification, there is no button that allows you to abort the operation (and no sensible way of implementing it). If you move the window when the application is waiting for you to scan your finger, certain parts of the screen will go blank and stop redrawing.

4.7.2 Device access contention

My first prototype implementation bears no consideration for multiple applications wanting to use the same fingerprint scanner at the same time. If one application has opened a specific fingerprint reader for potential later use, any other applications will be unable to open the fingerprint reader and it will be unclear as to why that is the case. There is no central resource management within my software; this impacts the user experience when multiple fingerprint-enabled applications are running.

4.7.3 Integration limitations

I developed pam_fprint with the aim of integrating with all existing applications that rely on PAM for authentication. In reality, many such applications operate incorrectly with interactive PAM modules that ask for something other than a password.

Other limitations were encountered due to the PAM architecture. It would be beneficial to be able to provide some kind of interactive fingerprint scanning feedback during authentication, but there is no clear way to pass such data through the PAM architecture.

Additionally, PAM is incapable of creating more flexible authentication scenarios where you are simultaneously presented the choice of scanning your finger *or* entering your password. The pre-existing UPEK TouchStrip project, ThinkFinger, does actually achieve this but the implementation is inelegant: it relies on threads, and relies upon the kernel being able to emulate key presses when a fingerprint is scanned. This leads to unwanted side effects[37] in some applications.

Chapter 5

Second prototype implementation

5.1 Objectives

After evaluating the problems identified in the first prototype, I began working on a new version to overcome these deficiencies. This was planned to be an incremental prototype on top of the earlier code, but it turned out that a fair amount of rewriting was necessary.

Development of this prototype focused upon overcoming the problems identified earlier:

- To overcome problems caused by a purely synchronous library interface, I aimed to add an asynchronous interface.
- To overcome device access contention problems, I decided to implement a systemwide fingerprint device access service on top of libfprint.
- To overcome integration limitations, I decided to circumvent PAM and integrate applications directly with the fingerprint device access service.

5.2 Adding an asynchronous interface

Section 4.7.1 discussed problems with the prototype 1 interface. These problems can be overcome with the addition of an asynchronous interface. Instead of saying "verify a fingerprint and block this thread until you have results," applications can say "please start a verification session now, and when you have some results, please call the following function in my application to let me know." The function call to start the verification session is *non-blocking* and immediately returns control to the application. Similar asynchronous interfaces can also be provided for enrollment and identification.

You might view libfprint as being a layer on top of the USB access layers. Figure 5.1 shows the system architecture from this perspective.

The design of any asynchronous system mandates that any layer below the asynchronous interface must also be asynchronous. If libfprint is to offer an asynchronous interface, all components beneath it must therefore also support asynchronous I/O. There is no problem with usbfs — the low-level kernel interface allowing you to

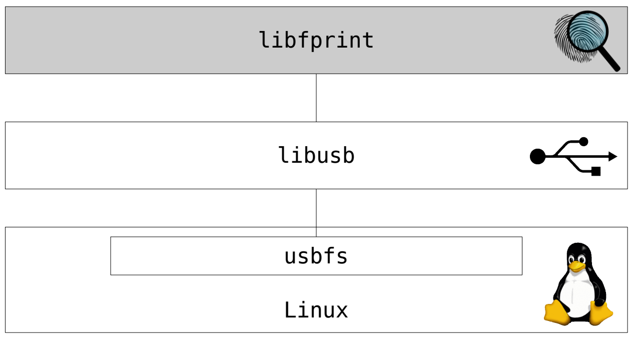


Figure 5.1: USB access layers

perform USB I/O from userspace — which does support asynchronous USB transfers. Issues only arise when we reach libusb (a C interface to usbfs), which only supports synchronous I/O. libusb's lack of asynchronous I/O functionality is the real reason why the asynchronous interface was not implemented from the start (the initial prototype focused on feasibility).

We can view all USB transfers as divided into 2 logical messages. For outgoing data:

1. Host sends data to device.
2. Device sends acknowledgement back to host when data has been processed.

There may be significant delay between these two stages. The device may take a while to process the data. A similar story exists for incoming data transfers, which again happen as two messages:

1. Host requests a certain amount of data from device.
2. Device sends data to host.

Again, long delays may exist between each message. The host may request data long before the device is actually ready to transfer any.

As a software interface, libusb combines both messages into a single function call, and hence only presents a blocking interface. For some of the devices, a function call such as the following would be used to retrieve fingerprint scan data from the device:

```
usb_bulk_read(device, 0x81, buffer, buffer_size, 0);
```

libusb then requests `buffer_size` bytes of data from the device and blocks until some data has been received. When interfacing with the aforementioned fingerprint scanners, this function will block until the user has scanned their finger; we are then left with the same problem as described in Section 4.7.1. It is clearly impractical to implement a true asynchronous libfprint interface when the underlying libusb function calls are blocking.

5.2.1 USB access libraries

OpenUSB

Even though libusb was the root of my inability to provide an asynchronous interface within libfprint, I did not see this as a big problem. libusb development appeared to be stalled, but I was already aware of an alternative USB library under development. OpenUSB[11], a project sponsored by Sun Microsystems, has similar goals and aims to provide missing features such as asynchronous I/O. I anticipated a relatively simple migration of my drivers from libusb to OpenUSB.

Having thought through this problem for some time, I had various expectations for OpenUSB's design with regard to asynchronous I/O. I envisioned a lightweight design where parent applications could monitor the asynchronous event sources. I envisioned an interface where it would be easy for applications to process asynchronous transfer results. After actually looking at OpenUSB, I saw none of this and was disappointed.

I began examining the OpenUSB implementation in more detail, aiming to rework things to better fit my ideas. I encountered various implementation details that I found offputting; notably the creation of several internal library threads and complicated synchronisation between them. It became apparent that it would be difficult for me to apply my own ideas to this design, and this was acknowledged[24] by an OpenUSB developer. I decided that it would be less effort to create my own USB library.

This is not meant as a criticism of OpenUSB; it is a promising project with a number of capable developers behind it. I simply required a different solution in order to cleanly present the libfprint application integration opportunities that I was hoping for.

Rewriting libusb

I moved on to developing my own USB library with the following goals:

- To offer both synchronous and asynchronous I/O.
- Lightweight, with no internal library threads (a "zero threads" model).

Like my other software components, I created this as an open source project with the aim of it being continued beyond the academic project. I borrowed some code from libusb, but the project turned out to be an almost complete rewrite.

After publishing my new library in a development repository, I got in touch with Johannes Erdfelt, the original libusb author. Erdfelt expressed[21] his liking of my design ideas, admitted that he was lacking time to continue the project, and handed the libusb project over to me! This is rather significant given the widespread adoption of existing libusb releases.

My new USB library is destined to become the v1.0 release of libusb. I shall refer to it as libusb-1.0 for the remainder of this report. Further mentions of libusb-0.1 refer to Erdfelt's earlier library which did not offer asynchronous I/O functionality.

This may sound like a lot of work, however, I had given this design enough consideration that it took me less than one week to reach a working implementation suitable for my needs. Other libusb users have different requirements, so I will need to spend time fleshing out the library before it can become a full libusb-0.1 replacement.

Zero threads model

Although discussion of my libusb-1.0 implementation is out of the scope of this report, I do need to explain the asynchronous transfer model to aid understanding of the following sections.

As previously mentioned, all USB transfers can be viewed as logically split into two messages, and an asynchronous interface shares that split view of each transfer. The application calls a function to asynchronously submit a transfer request, and the application receives a callback from libusb when results have become available.

After a transfer has been submitted, libusb must keep track of ongoing operations, monitor them for completion, and invoke the application-supplied callback functions. This presents an interesting question under my threadless design: the library can only be executing code when the parent application is calling a libusb function, so how and when can it perform these event handling duties?

My design dictates that any parent application that performs asynchronous I/O must repeatedly call a function, `libusb_handle_events()`, so that libusb can handle any pending events. This raises the next question; how does the application know when it should call this function?

Most applications of any form of complexity will be built around the concept of a *main event loop.* This loop monitors various event sources and invokes appropriate functions when events are detected. Event sources such as network sockets, timers, and other kernel interfaces are usually represented as *file descriptors* (recall one of the core UNIX design philosophies: everything is a file). An application typically groups file descriptors for all event sources and passes them to the `poll()` or `select()` system call. Those system calls put the application to sleep and awaken it when events are ready to be processed. The application then handles the events and proceeds to the next iteration of the main loop.

My libusb-1.0 design fits this model perfectly. libusb-1.0 exposes a set of file descriptors which the application can monitor. When events are detected on such file descriptors, the application is trusted to call `libusb_handle_events()` as soon as possible.

5.2.2 Rethinking libfprint abstraction layers

With the availability of an asynchronous USB layer, I next returned to my goal of adding an asynchronous interface to libfprint.

In the previous prototype, the drivers implemented blocking functions for operations such as “wait for finger-on-sensor.” There can be no blocking functions in an asynchronous system, so the abstraction was changed to a model where the driver provides a non-blocking function to enable the device, and when enabled, the driver asynchronously informs the library of events (such as finger presence) as they happen.

5.2.3 Rewriting libfprint drivers

Next, I rewrote all the drivers to fit the new abstraction layer interface, and converted them to use solely asynchronous USB I/O through libusb-1.0.

A piece of example code is shown below which I shall refer to throughout this section. The code is simplified and shortened to remove irrelevant details while still highlighting certain asynchronous design considerations.

```
int do_init(struct fp_dev *dev) {
    send_cmdresponse(dev, init_resp03);
    read_msg(dev, ...);
    send_cmd28(dev, 0x06, 0x04, 1);
    read_msg28(dev, 0x06, ...);
}
```

The code sample above is a simplified version of the UPEK TouchStrip initialisation code from the previous prototype. Each function call in this example corresponds to an individual USB transfer. These transfers must be executed in sequence; each transfer must be delayed until the one before it has completed. `do_init` is a synchronous and blocking function which we must convert to an asynchronous model to meet the new design goals.

The execution flow of the above function is clear, but that is easily lost when moving to an asynchronous model. Under an asynchronous model, each function call to fire off a transfer returns immediately, but the subsequent transfer must be deferred until after the current one has completed. A basic implementation could achieve this by firing off the subsequent transfer from the callback function, e.g.:

```
int do_init(struct fp_dev *dev) {
    send_cmdresponse(dev, init_resp03, send_resp03_cb);
}

/* called when the transfer initiated by do_init completes */
void send_resp03_cb(struct fp_dev *dev) {
    read_msg(dev, ..., read_msg03_cb);
}

/* called when the transfer initiated by send_resp03_cb completes */
void read_msg03_cb(struct fp_dev *dev) {
    send_cmd28(dev, 0x06, 0x04, 1, send_cmd28_06_cb);
}

/* and so on. the remaining functions are omitted to save space */
```

When considering extra logic that must be included such as error checking, error handling, and more advanced situations where a transfer is repeated in a loop, the model of each transfer becoming its own function had the effect of reducing readability, losing linear execution flow, and increasing complexity over the synchronous equivalent. I was not happy with such sacrifices.

I considered implementing state machines for these situations, where the asynchronous callback functions could simply advance the machine to the next state and everything else could be done in sequence from a state handling function. Figure 5.2 shows the example function as a state machine.

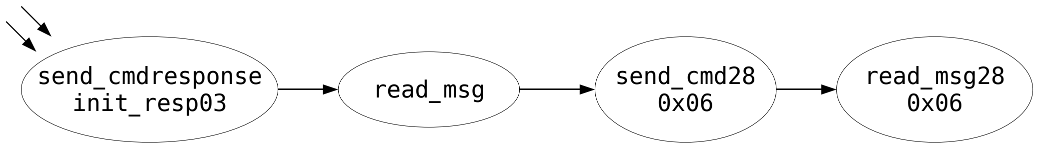


Figure 5.2: Example initialisation as a state machine

When expressed in code, this state machine can be represented in two functions and the execution flow is once again more obvious:

```
void callback_function(state_machine *sm)
{
    iterate_to_next_state(sm);
}

void state_handler(state_machine *sm, struct fp_dev *dev, int state) {
    switch (state) {
    case 0:
        send_cmdresponse(dev, init_resp03, callback_function);
        break;
    case 1:
        read_msg(dev, ..., callback_function);
        break;
    case 2:
        send_cmd28(dev, 0x06, 0x04, 1, callback_function);
        break;
    case 3:
        read_msg28(dev, 0x06, ..., callback_function);
        break;
    }
}
```

It turns out that most of the state machines required by my drivers are similar to the above: the execution flow is rarely anything other than linear. libfprint implements a simplistic state machine model tuned for my requirements:

- Each state machine provides a list of states that will normally be traversed linearly from start to finish.
- An implicit final state is created, with an error parameter. An error code of 0 indicates successful completion, anything else indicates error.
- The final state will normally be reached with error code 0 when the machine is incremented beyond the last non-implicit state.

- The final state can also be reached from any earlier state, with an appropriate error code. This also allows for state machines to terminate with error at any point, and also allows them to complete early (error code 0).
- Although it will normally not happen, each state can jump to any other state through a simple function call.

This design can neatly model the simplistic sequential TouchStrip initialisation procedure, as well as more complex initialisation routines for devices such as the U.are.U 4000B shown in Figure 5.3.

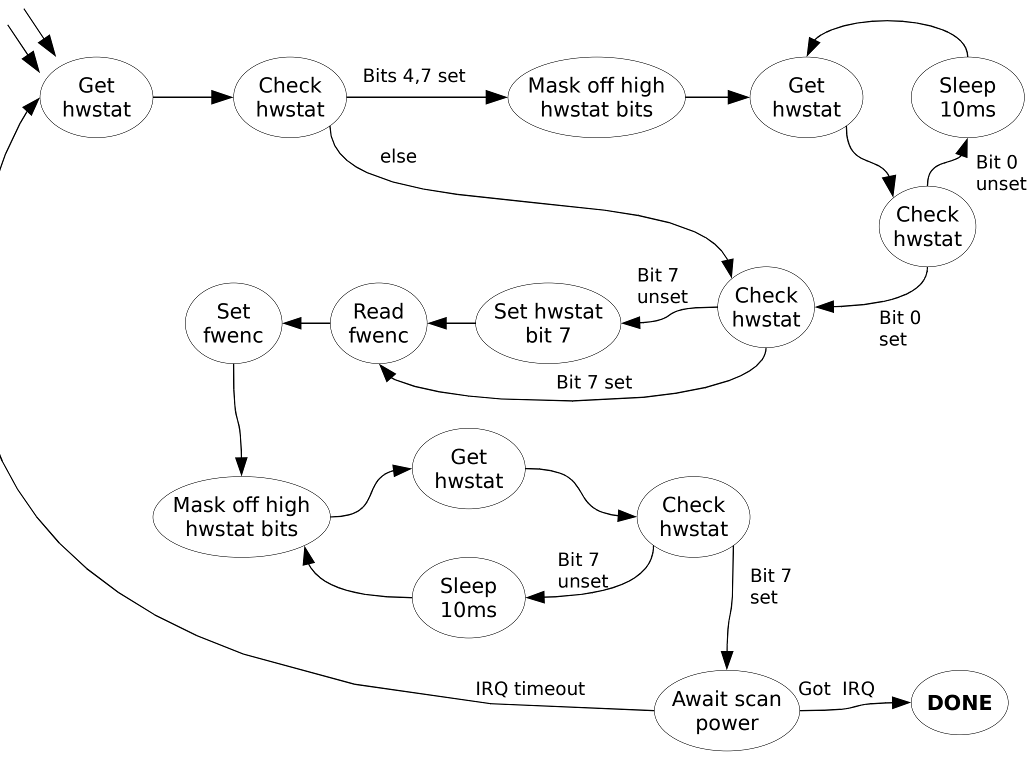


Figure 5.3: U.are.U 4000B initialisation state machine

5.2.4 Exposing an asynchronous libfprint interface

After moving the abstraction layers to an asynchronous model and reworking drivers appropriately, the next step was to add a public interface to the new abstraction layer structure. This was a fairly simple process of adding some publicly exported functions which wrap the new functions in my abstraction layers.

Recall Section 5.2.1 which discussed the *zero threads* model employed in libusb-1.0. Like libusb, libfprint is threadless internally. As libfprint can be viewed as an application built on top of libusb, libfprint must honour libusb's requirement of calling `libusb_handle_events()` when libusb has work to do. libfprint, however, is a zero-threaded library and only executes when a parent application is calling into it.

To solve this problem, libfprint exposes a `fp_handle_events()` function, which effectively boils down to a call to `libusb_handle_events()`. libfprint also provides functions to provide access to libusb's set of file descriptors. Applications built on top of libfprint's asynchronous interface are then required to monitor those file descriptors in their main loop and invoke `fp_handle_events()` when activity is detected.

In order to test the improvements in this prototype, I ported fprint_demo to use the new asynchronous interface. This involved adding the file descriptors exposed by libfprint to GTK+'s main loop, a simple modification. Porting to the asynchronous interface also solved the two problems mentioned in Section 4.7.1; the GUI remains responsive while waiting for a fingerprint scan, and a cancel button has been implemented cleanly.

5.2.5 Reimplementing a synchronous interface

The previously implemented libfprint interface had broken after the internal abstraction layers had been moved to an asynchronous model. Despite the deficiencies of the previous interface, a synchronous interface does have the advantage of simplicity so I decided that I would provide it in addition to the asynchronous interface.

I was able to reimplement the synchronous interface with simple functions that call into the asynchronous layer and wait for results before returning.

5.3 Implementing a systemwide daemon

To solve problems relating to fingerprint device access contention, I decided to implement a systemwide daemon for management of hardware. The daemon would expose libfprint functionality over the D-Bus[6] system message bus. This design choice was independently backed by several community members.

I am no fan of resource wastage, and the idea of a dedicated daemon for fingerprint scanning may seem somewhat sub-optimal. D-Bus addresses this concern through its activation mechanism; D-Bus will only launch the daemon when it is needed, and the daemon is free to shut down when nobody is using it.

Access contention problems are solved by the systemwide nature of this daemon. Even if multiple users/applications are using fingerprint readers concurrently, only one daemon will be running. The daemon will be aware of all users and their applications, and can share resources appropriately.

During development, I was able to implement a prototype daemon, fprintd. fprintd offers enrollment and verification through exposing fingerprint scanner objects on the bus. Unfortunately, time constraints resulted in development halting before I made further progress. My ideas for the next steps in development are presented below.

- Use ConsoleKit[26] to determine the current system user.
- Use PolicyKit[36] to allow configuration of access permissions. PolicyKit would allow for fine-grained configuration policies such as "only these certain users can enroll fingerprints," and more obviously, "only root can delete other users' enrolled fingerprint data."
- Develop a pluggable storage backend to allow for flexible storage scenarios such as storing enrollment data on a remote database server.

- Implement logical ownership tracking. For example, track that user X and application Y are using a specific fingerprint reader at that specific moment. Other applications could then be properly informed why a specific fingerprint reader is unavailable.

- Flesh out the D-Bus interface to expose all libfprint features. For example, the prototype implementation only offers a simple verification interface, but it would be useful to optionally present image data over D-Bus so that applications can present visual feedback of the scan.

Chapter 6

Results

6.1 libfprint performance

6.1.1 Library core

The abstraction layers were implemented as planned. I tested functionality after development of each component and fixed problems as I found them. The final implementation feels to have achieved stability.

6.1.2 UPEK TouchStrip driver performance

Given that the TouchStrip does not present images, the performance of this driver is mostly down to the quality of image processing that happens in hardware. Fortunately, my testing indicates that the image processing results are accurate; I have never encountered a false acceptance, and I only saw a couple of false rejections before I had learnt the scanning technique.

The driver's communication with the device is mostly stable, but occasionally I discover a new status code from the device which the driver does not know how to handle (status code indicate conditions such as “finger scanned too fast” and “finger scanned too slow”). Fixing this should be easy; I have recently been referred to some UPEK documentation including a status code listing.

6.1.3 Digital Persona U.are.U 4000B driver performance

After solving some problems within the device initialisation sequence, this driver has achieved stability. The driver occasionally receives an interrupt of which the meaning is unknown; I have named it *the interrupt of death*. Arrival of this interrupt seems to indicate that the next requested scan session will fail (returning no image data). This only happens sporadically and for unknown reasons, and will need further experimentation to know how to handle.

The imaging performance of this device is excellent. The returned images are large and MINDTCT finds a large number of minutiae, leading to suitably distinct BOZORTH3 scores for matching/non-matching fingers. Figure 6.1 shows an image returned by this driver and its binarized form with minutiae plotted.

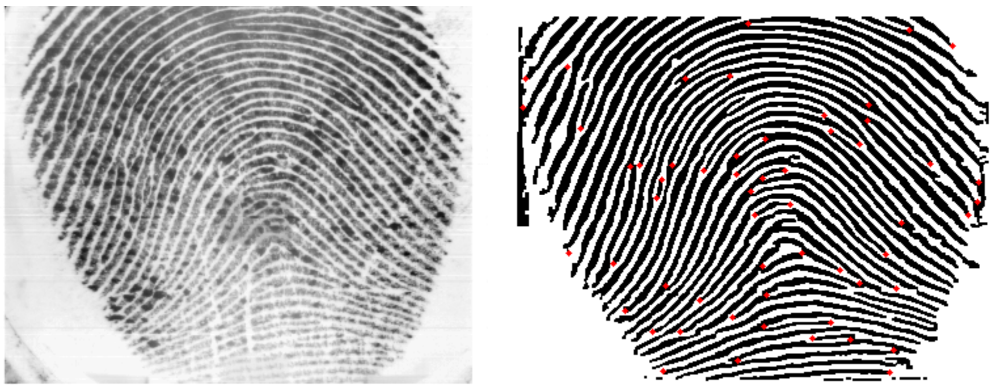


Figure 6.1: U.are.U 4000B fingerprint image and processing results

6.1.4 AuthenTec AES4000 performance

This device presented some challenges due to the low quality of images that are detected by the sensor. After implementing the image enlargement workaround described in Section 4.4.3, image performance of this driver has reached satisfactory levels but is not perfect; I have experienced some false rejections and have seen one false acceptance. I believe the main limitations are the small number of minutiae and the noise around the fingerprint. See Figure 6.2 for example images.

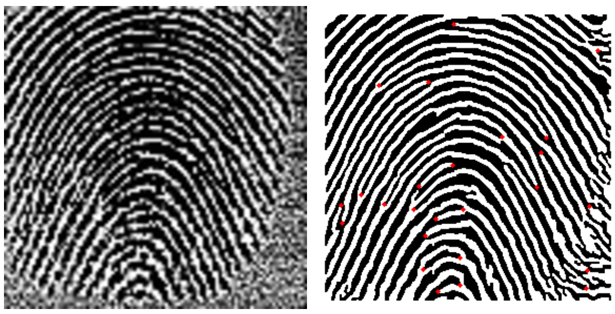


Figure 6.2: AES4000 fingerprint image and processing results

This driver is otherwise working perfectly.

6.1.5 AuthenTec AES2501 performance

This driver is working well. On occasion, the driver incorrectly reports that the finger has been removed sooner than it has, but this should be fixable by making the finger removal detection more dynamic. When this does happen, it is easy enough to retry the scan.

The swipe scanning technique produces images covering a large finger area where many minutiae can be detected. Imaging performance is excellent as a result. Figure 6.3 shows an image returned by this driver and its binarized form with minutiae plotted.

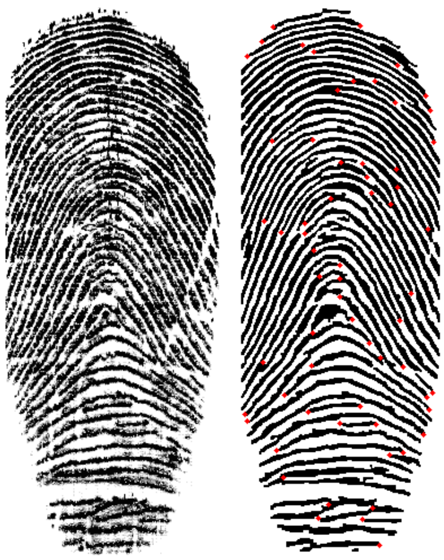


Figure 6.3: AES2501 fingerprint image and processing results

6.2 fprint_demo results

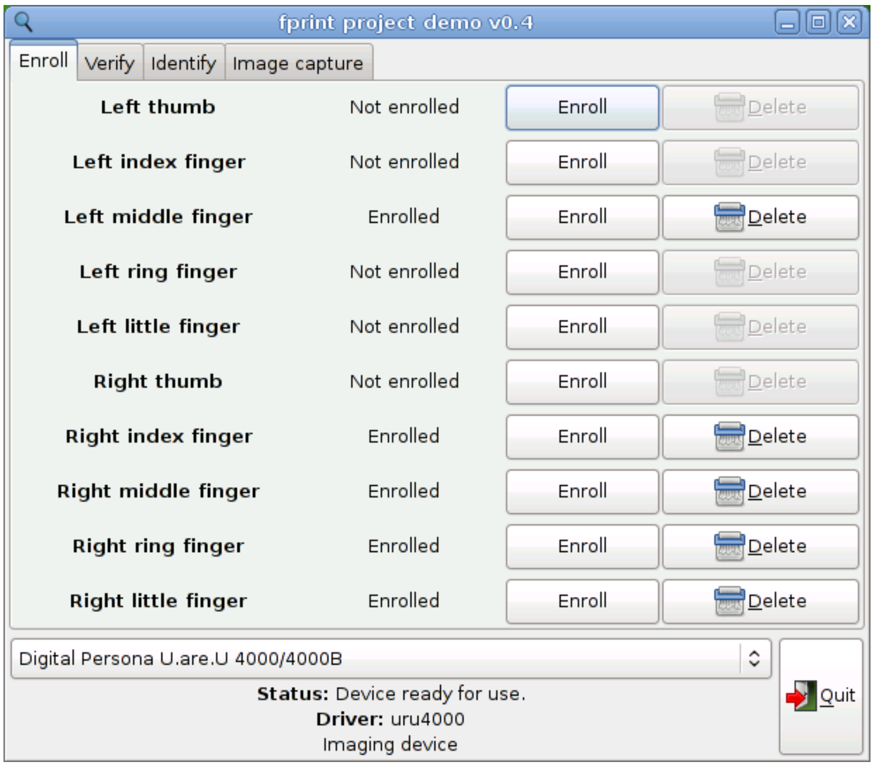
fprint_demo was implemented as planned and works well. I do not know of any problems. Screenshots of the functionality are shown in Figures 6.4, 6.5, 6.6, and 6.7.

6.3 pam_fprint results

pam_fprint implementation went smoothly and worked reliably during my testing. Figure 6.8 shows a shadow login session configured to use pam_fprint rather than asking for a password.

6.4 fprintd results

fprintd development did not progress far enough to present the fully-fledged application integration opportunities that I planned for. Nevertheless, basic functionality has been implemented and tested using simple command line example programs. Figure 6.9 shows the fprintd object space from the view of a D-Bus debugging utility.

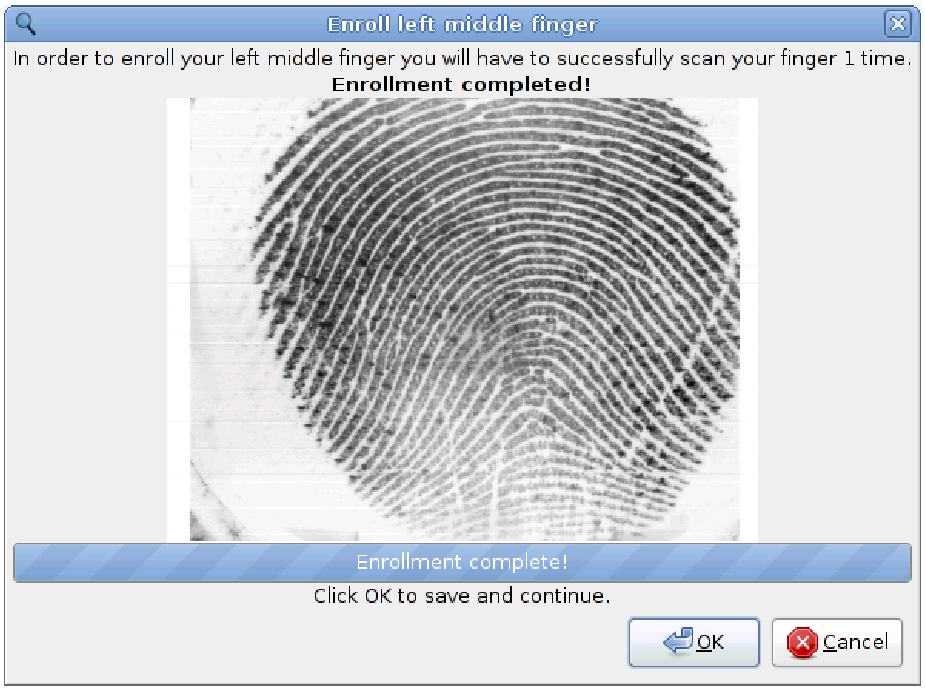


Left thumb	Not enrolled	Enroll	Delete
Left index finger	Not enrolled	Enroll	Delete
Left middle finger	Enrolled	Enroll	Delete
Left ring finger	Not enrolled	Enroll	Delete
Left little finger	Not enrolled	Enroll	Delete
Right thumb	Not enrolled	Enroll	Delete
Right index finger	Enrolled	Enroll	Delete
Right middle finger	Enrolled	Enroll	Delete
Right ring finger	Enrolled	Enroll	Delete
Right little finger	Enrolled	Enroll	Delete

Digital Persona U.are.U 4000/4000B

Status: Device ready for use.
Driver: uru4000
Imaging device

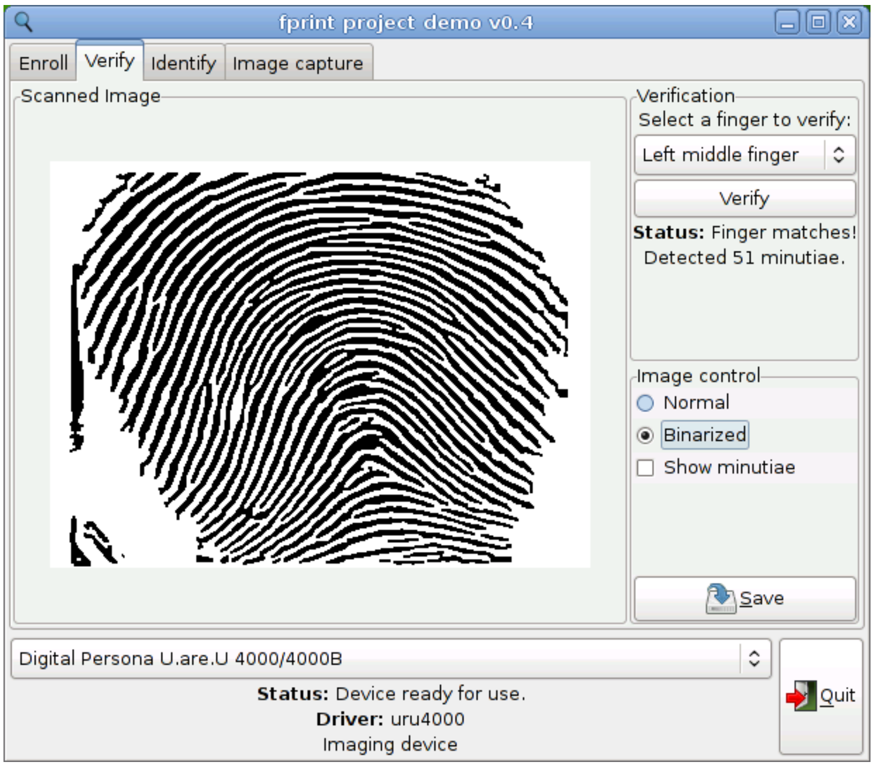
Figure 6.4: fprint_demo enrollment data management



In order to enroll your left middle finger you will have to successfully scan your finger 1 time.
Enrollment completed!

Enrollment complete!
Click OK to save and continue.

Figure 6.5: fprint_demo enrollment interface



Select a finger to verify: Left middle finger

Verify

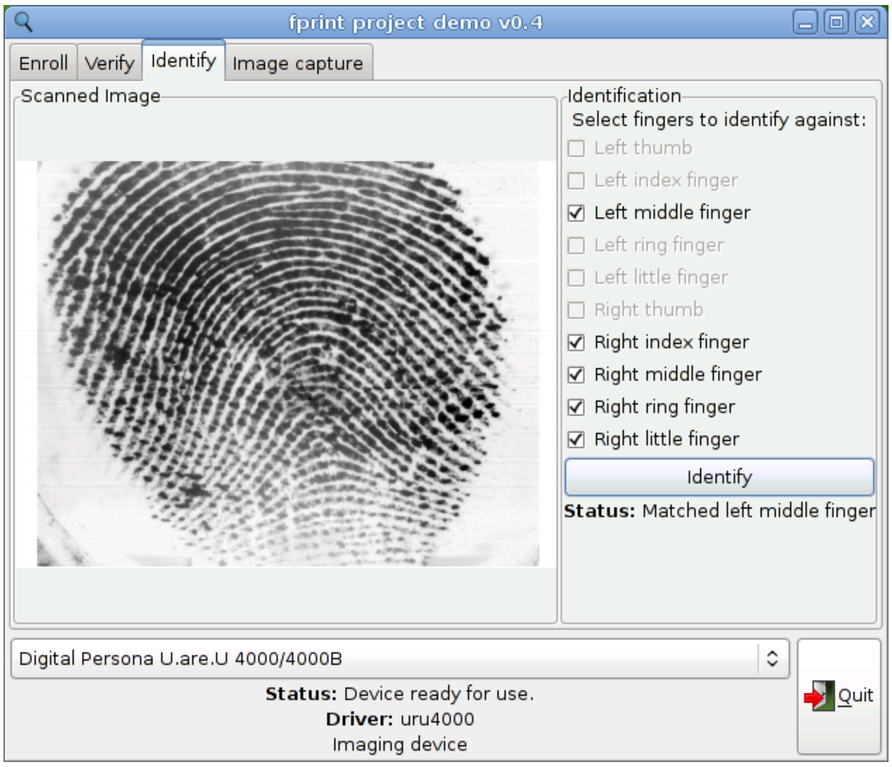
Status: Finger matches!
Detected 51 minutiae.

( ) Normal
(●) Binarized
☐ Show minutiae

Digital Persona U.are.U 4000/4000B

Status: Device ready for use.
Driver: uru4000
Imaging device

Figure 6.6: fprint_demo verification interface



Select fingers to identify against:
☐ Left thumb
☐ Left index finger
☑ Left middle finger
☐ Left ring finger
☐ Left little finger
☐ Right thumb
☑ Right index finger
☑ Right middle finger
☑ Right ring finger
☑ Right little finger

Identify

Status: Matched left middle finger

Digital Persona U.are.U 4000/4000B

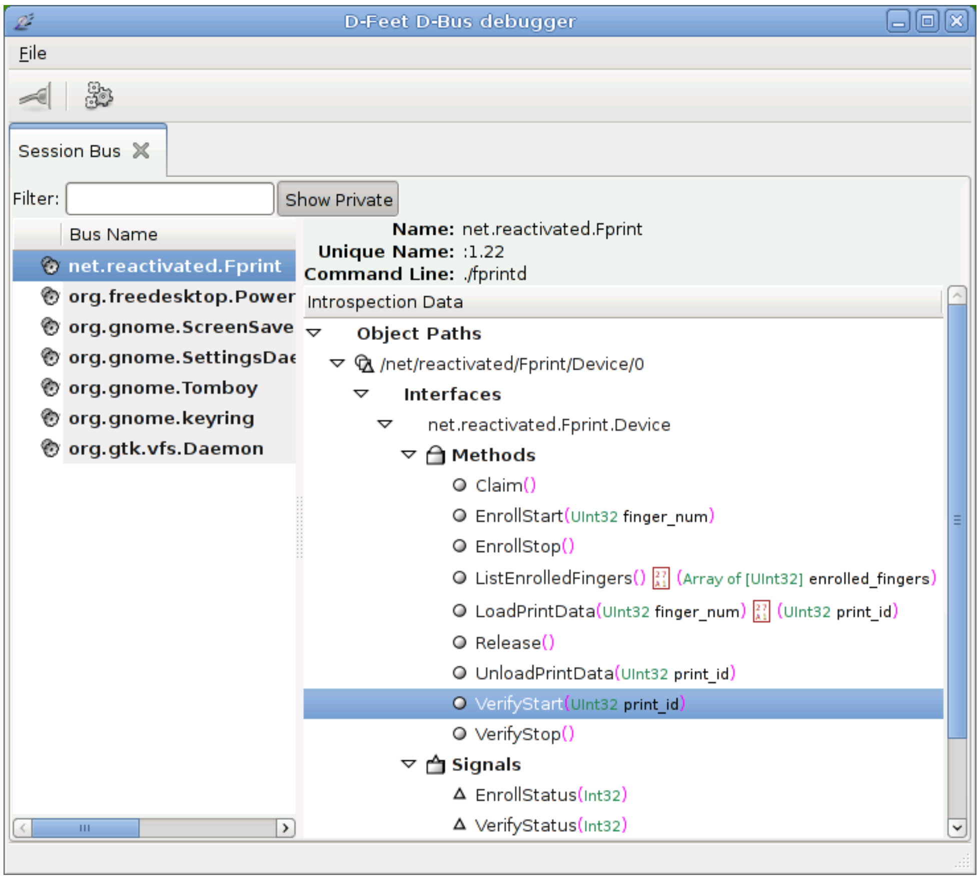
Status: Device ready for use.
Driver: uru4000
Imaging device

Figure 6.7: fprint_demo identification interface

```
This is airbag (Linux i686 2.6.25-rc8) 11:45:47

airbag login: dsd
Please scan right index finger on Digital Persona U.are.U 4000B
Last login:  Tue Apr 8 11:32:31 +0100 2008 on tty1.
dsd@airbag ~$ _
```

Figure 6.8: pam_fprint controlling a shadow login session



```
Name: net.reactivated.Fprint
Unique Name: :1.22
Command Line: ./fprintd
Introspection Data
Object Paths
  /net/reactivated/Fprint/Device/0
    Interfaces
      net.reactivated.Fprint.Device
        Methods
          Claim()
          EnrollStart(UInt32 finger_num)
          EnrollStop()
          ListEnrolledFingers() (Array of [UInt32] enrolled_fingers)
          LoadPrintData(UInt32 finger_num) (UInt32 print_id)
          Release()
          UnloadPrintData(UInt32 print_id)
          VerifyStart(UInt32 print_id)
          VerifyStop()
        Signals
          EnrollStatus(Int32)
          VerifyStatus(Int32)
```

Figure 6.9: D-Bus debugger showing fprintd object space

Chapter 7

Evaluation

7.1 Analysis of results

7.1.1 Hardware support

I have been successful in supporting the 4 ranges of devices I targeted in my design. All of them work well enough for usable fingerprint-based authentication to be offered by my software.

Reliability of the drivers is not 100%, but I am confident that I can solve the known problems given some more time working on them. The outstanding problems only appear occasionally and do not impair usability as the user can just retry the scan.

7.1.2 Abstraction

Providing a high-level generic fingerprint scanning API was one of my core goals. I achieved this through implementing layers of abstraction and providing a public API to access the uppermost layer.

The public API successfully meets this goal through exposing only the core considerations needed for fingerprint scanning. API documentation has been produced, which I believe has been useful to those attempting to use libfprint from other projects.

Development of pam_fprint is worthy of a mention here. I developed and tested the initial prototype for the login module in under 20 minutes, the implementation weighing in at under 250 lines of code. This shows the success of my abstract design simplifying the process of integrating fingerprint scanning into applications.

7.1.3 Asynchronous design

The conversion to an asynchronous model took longer than expected, especially considering the platform improvements (rewriting libusb) that were necessary in order to realize my design ideas. Despite delaying the remainder of the project implementation efforts, I remained confident at every stage that this was the right direction to be going in.

Ultimately, libfprint previously only offered a synchronous interface, but moving in the asynchronous direction allows the library to offer both a synchronous interface *and* an asynchronous interface. No functionality has been lost and the library is now suitable for integration into a wider range of applications. I demonstrated that the asynchronous model solved some important problems through taking fprint_demo to the stages of the 2nd prototype, and I suspect that we will reap further benefits of the asynchronous model as development continues.

7.1.4 Application integration

I had originally aimed to present my software smoothly integrated with existing software such as login managers. Although I managed to reach prototype stages through the development of pam_fprint, I did not have enough time to bring application integration to the stages where I would like.

The targeted authentication system, PAM, presented itself insufficient to meet the unique requirements of fingerprint authentication systems. It seems that the best approach in the short term is to circumvent PAM in applications when fingerprint readers are present. A possible future long-term project is to design a new generic authentication system better meeting the needs of fingerprint scanning and other biometric methods.

Smooth application integration requires the systemwide daemon implementation which only reached prototype stages during my project. Daemon development required the asynchronous interface, the development of which ended up consuming more time than expected.

7.2 Open source project

Being an open source project that made previously unsupported devices usable and useful, my project attracted quite a lot of attention. A community has formed and is quite active with discussion and user support. A few key members also contribute to development and they have been mentioned throughout this report.

I host a mailing list[5] which is the central communication channel for user support, discussion and development. The list is relatively busy, with a total of 520 messages to date from 161 subscribers.

The project website[7] has had 28,312 hits to the front page, and a total of 128,149 hits over the entire website.

I produced the official releases in the form of source code tarballs which were published on SourceForge. SourceForge provides some statistics[12]: there have been 8101 recorded downloads of source components, an average of 56 downloads per day since initial release, or a total of 1.9GB download bandwidth. Figure 7.1 shows downloads by month.

It is unfortunately not possible to count the number of users of my software as the majority of users install it from distribution-supplied binary packages. Nevertheless, I feel that the source tarball download statistics are enough to indicate the successful launch of the open source project.

All statistics above are correct as of April 7th, 2008.

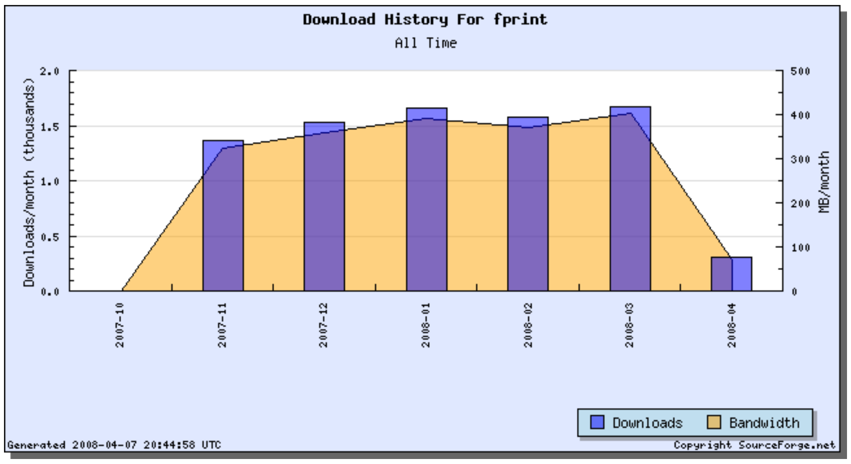


Figure 7.1: SourceForge download statistics by month

7.2.1 Joining of communities

Section 1.4 mentioned how different projects were focusing on their own devices and not really working together. I was pleased to see other developers recognising the advantages of the unified approach I was taking in my project. Hoenig quickly stated his interested in obsoleting the ThinkFinger project in favour of mine. Gerard Klaver, who had previously spent some time tinkering with AES2501 devices and had informally become the definitive source of information for all kinds of AuthenTec scanners, closed his project[23] with the note "Development stopped, use fprint."

7.2.2 Distribution interest

Various Linux distributions expressed interest in packaging up my software to allow their users to install it with ease. Given that I am not ready to commit to stable interfaces, I requested that distributions exclude my software from their stable/official trees. Nevertheless, the following distributions ship packages of my software in their unstable/experimental trees: Arch, Debian, Fedora, Gentoo, Mandriva, SuSE, and Ubuntu.

Some distributors expressed interest in including my software in their base install, with the intention of fingerprint scanning being available out-of-the-box on such systems. My software is being considered for inclusion in Mandriva 2008.1[1] and Fedora 10[4].

7.2.3 Identified problems

Releasing my software to the community allowed for testing and development in environments other than my own. I shall briefly summarise some of the key problems that have been identified by others.

- **NBIS performs badly for small images.** I was already aware of this from my experiences with the AES4000, although I managed to tweak that driver to perform acceptably. Bretaudeau's contributed driver for the AES1610 device encounters similar issues; the AES1610 sensor is small and the resultant images typically have under 10 minutiae[18].

- **NBIS performs badly for older users.** Mark Vytlacil indicated that he was not getting the kind of results that I was presenting with the U.are.U 4000B and suggested[32] that it was because the skin on his fingers had lost elasticity with age.

- **AES2501 auto-calibration is needed for some hardware.** AES2501 sensors can be found in a lot of different laptops, and they appear to exist in different configurations. Our fixed sensitivity values perform badly[22] in certain systems, to the point where images are so bad that they are not processable.

Chapter 8

Conclusion

8.1 Achievements

8.1.1 Targets met

The central project component, libfprint, reaches the following project goals:

- Cleanly layered design and architecture.
- Satisfactory hardware support for all devices discussed in Section 2.1.
- Generic public API for fingerprint scanning.
- Suitability for integration into a wide range of applications.

The authentication module, pam_fprint, allows for integration with many existing applications and serves as a simple example of how to integrate libfprint into an application.

The demonstration GUI, fprint_demo, allows for demonstration and testing. It also serves as an example of how to integrate libfprint into more complex applications with multiple event sources.

All components were released as open source, leading to the formation of an active community.

8.1.2 Targets not met

Although my software has been integrated into other software via the use of pam_fprint, application integration has not reached the stages I would have hoped for. There is no visual scan feedback, communication with the user is poor, and pam_fprint as a proof-of-concept has its own implementation deficiencies.

I established that exposing fingerprint scanning functionality through a D-Bus daemon was a sensible solution here, but my fprintd implementation was limited by time constraints.

I aimed to form a community around my open source project with a "release early, release often" mantra. I produced regular releases for the first prototype, however, no official second prototype releases have been made for reasons to be explained in the following section. Nevertheless, the code has always been available from public development repositories and I am aware of several users who have tried it.

8.2 Future plans

8.2.1 Upcoming development goals

The software components only reached prototype stages under the time constraints of the academic project. As mentioned in the introduction, I intend to continue development, and there is plenty to be done!

I have been unable to create official releases of my recent work because of dependencies between components that do not yet offer interface stability. In order to be able to achieve my application integration goals, I shall be stabilising components from the ground up through the following tasks:

1. **Complete, test and stabilise libusb-1.0.** libusb is the lowest level component in the project. Existing versions are widespread, so I must work to satisfy existing users through a well-designed and documented API.

2. **Tweak libfprint asynchronous API and release libfprint-0.1.0.** Having just designed an asynchronous fingerprint scanning interface for the first time, there are a few things I would like to change before releasing it as a finalised API. The asynchronous libfprint release also depends upon libusb-1.0 reaching stability.

3. **Complete, test and stabilise fprintd.** The short term fprintd plans were discussed in Section 5.3. As fprintd uses libfprint's asynchronous API, it cannot be released until libfprint-0.1.0 has been completed.

4. **Continue application integration tasks.** Existing applications will be modified to optionally use fprintd for authentication. These tasks obviously depend on fprintd being released.

8.2.2 Long term goals

As is natural for an open source project, some community members have raised possible future directions for the project.

- **Fingerprint scanning on embedded devices.** Community members including Jeff White[33] and Jono Woodhouse[34] expressed interest in running my software on small systems with scarce resources. This is impractical at the moment because libfprint depends on some sizable libraries, but work has been started to remove these dependencies. I plan to revisit this after reaching my application integration goals.

- **Cross-platform compatibility.** libfprint is written with architecture portability in mind, and all of its dependencies are cross-platform. I have been focusing development efforts on Linux, but community members have been able to run my software on Mac OS X[27] and OpenBSD[30]. I believe it is also possible to run the first prototype on Windows given that ports of glib and libusb-0.1 are available. Some portability issues remain to be fixed and extra effort will be required to integrate with standard authentication systems on these alternative platforms.

- **More hardware support.** There are a lot of fingerprint readers out there. Some users have offered their assistance in future reverse engineering efforts to allow drivers to be written.

- **Language bindings.** I hope to spark efforts to make libfprint and fprintd easily accessible from development environments based on languages such as Java and Python.

References

[1] *2008.1 Detailed Specifications - Mandriva Community Wiki*, `http://wiki.mandriva.com/en/2008.1_Detailed_Specifications`, Retrieved on April 7th, 2008.

[2] *Authentec AES2501B Sensor*, `http://www.authentec.com/products-pcsandperipherals-aes2501b.html`, Retreived on March 19th, 2008.

[3] *dpfp: DigitalPersona/Microsoft Fingerprint Reader Driver for Linux*, `http://dpfp.berlios.de`, Retrieved on March 19th, 2008.

[4] *Features/Fingerprint - Fedora Community Wiki*, `http://fedoraproject.org/wiki/Features/Fingerprint`, Retrieved on April 7th, 2008.

[5] *fprint Info Page*, `http://lists.reactivated.net/mailman/listinfo/fprint`, Retrieved on April 7th, 2008.

[6] *freedesktop.org - Software/dbus*, `http://dbus.freedesktop.org/`, Retrieved on April 6th, 2008.

[7] *Main Page - fprint project*, `http://www.reactivated.net/fprint/`, Retrieved on April 7th, 2008.

[8] *National Institute of Standards and Technology*, `http://www.nist.gov`, Retrieved on March 20th, 2008.

[9] *NIGOS License*, `http://www.itl.nist.gov/iad/894.03/nigos/NIGOS_licdis_061906.pdf`, Retrieved on March 20th, 2008.

[10] *NIST Biometric Image Software*, `http://fingerprint.nist.gov/NBIS/index.html`, Retrieved on March 20th, 2008.

[11] *OpenUSB*, `http://openusb.sourceforge.net/wiki`, Retrieved on April 6th, 2008.

[12] *SourceForge.net: Project Statistics for fprint*, `http://sourceforge.net/project/stats/detail.php?group_id=208521&ugn=fprint&type=prdownload&mode=alltime`, Retrieved on April 7th, 2008.

[13] *Targus PA460U*, `http://www.targus.com/us/drivers_manuals_archive.asp?SKU=PA460U`, Retreived on March 19th, 2008.

[14] *ThinkFinger*, http://thinkfinger.sourceforge.net/, Retrieved on March 19th, 2008.

[15] *User's Guide to Export Controlled Distribution of NIST Biometric Image Software*, Retrieved from NBIS CDROM Distribution, release 1.1.0.

[16] *User's Guide to NIST Biometric Image Software*, http://fingerprint.nist.gov/NBIS/nbis_non_export_control.pdf, Retrieved on March 20th, 2008.

[17] Cyrille Bagard, *aes2501 kernel driver*, http://home.gna.org/aes2501/index_en.html, Retrieved on March 19th, 2008.

[18] Anthony Bretaudeau, *AES 1610 support*, http://lists.reactivated.net/pipermail/fprint/2007-November/000074.html, November 2007, Email to fprint mailing list.

[19] Daniel Drake, *Interviews with Cyrille Bagard*, November 2007.

[20] Daniel Drake, *Username-less authentication?* http://lists.reactivated.net/pipermail/fprint/2007-December/000305.html, December 2007, In response to email from Eddie Hung.

[21] Johannes Erdfelt, *taking over the libusb project?*, November 2007, Email discussion.

[22] Michael Ivanov, *Poor fingerscan quality on nw9440 aes2501*, http://projects.reactivated.net/fprint/bugs/index.php?do=details&task_id=2, November 2007, libfprint bug report.

[23] Gerard Klaver, *authentec fingerprint information*, http://gkall.hobby.nl/authentec.html, Retrieved on April 25th, 2008.

[24] Sophia Li, *Re: Some untested openusb patches*, http://sourceforge.net/mailarchive/message.php?msg_name=474FE30B.6050602%40sun.com, November 2007, In response to email from Daniel Drake.

[25] Pavel Machek, *driver for thinkpad fingerprint sensor*, http://lkml.org/lkml/2006/8/2/237, August 2006, Email to Linux kernel mailing list.

[26] William Jon McCann, *Announcing ConsoleKit*, http://lists.freedesktop.org/archives/hal/2007-January/006996.html, January 2007.

[27] Geppy Parziale, *Instruction to compile on OS X*, http://lists.reactivated.net/pipermail/fprint/2008-January/000353.html, January 2008, Email to fprint mailing list.

[28] Shivang Patel, *Fingerprint Verification System*, http://fvs.sourceforge.net, Retrieved on March 20th, 2008.

[29] Sanyam Sharma and Sunil Mohan Ranta, *eFinger FingerPrint Matching Tool,* http://efinger.sourceforge.net, June 2004, Retrieved on March 20th, 2008.

[30] Joshua Stein, *libfprint portability fixes,* http://lists.reactivated.net/pipermail/fprint/2007-December/000296.html, December 2007, Email to fprint mailing list.

[31] UPEK Technical Support, *Information from UPEK on Fingerprint reader 147e:2016,* http://lists.reactivated.net/pipermail/fprint/2008-March/000435.html, February 2008, Message from UPEK forwarded by Robert Hoffler.

[32] Mark Vytlacil, *age and fingerprints,* http://lists.reactivated.net/pipermail/fprint/2007-December/000284.html, December 2007, Email to fprint mailing list.

[33] Jeff White, *fprint for embedded,* http://lists.reactivated.net/pipermail/fprint/2008-January/000319.html, January 2008, Email to fprint mailing list.

[34] Jono Woodhouse, *Dependencies on glib/imageMagik,* http://lists.reactivated.net/pipermail/fprint/2007-December/000227.html, December 2007, Email to fprint mailing list.

[35] Wittawat Yamwong, *aes2501-wy source distribution,* http://gkall.hobby.nl/aes2501-wy.tar.bz2, Retreived on March 19th, 2008.

[36] David Zeuthen, *libhal-policy - PolicyKit,* http://lists.freedesktop.org/archives/hal/2006-March/004770.html, March 2006.

[37] David Zeuthen, *Thinkfinger's PAM module emits annoying RETURN key when using password,* https://bugzilla.redhat.com/show_bug.cgi?id=356921, October 2007.