Summer 5-7-2010

Security analysis of a fingerprint-secured USB drive

Benjamin David Rodes

James Madison University

Follow this and additional works at: https://commons.lib.jmu.edu/master201019

Part of the Computer Sciences Commons

Recommended Citation

https://commons.libjmu.edu/master201019/387

This Thesis is brought to you for free and open access by the The Graduate School at JMU Scholarly Commons. It has been accepted for inclusion in Masters Theses by an authorized administrator of JMU Scholarly Commons. For more information, please contact dc_admin@jmu.edu.
Security Analysis of a Fingerprint-Secured USB Drive

Benjamin David Rodes

A thesis submitted to the Graduate Faculty of

JAMES MADISON UNIVERSITY

In

Partial Fulfillment of the Requirements

for the degree of

Master of Science

Department of Computer Science

May 2010
Dedication

This thesis is dedicated to Professor Malcolm Lane, for his help and friendship. Without your support my education in computer science would not exist.
Acknowledgments

First, I would like to thank my advisor, Dr. Xunhua Wang, for going along on this adventure, for his help and hard work, and for his commitment to a high teaching standard. I have learned much. I would also like to thank Dr. Ralph Grove and Dr. Brett Tjaden for agreeing to be on my thesis committee and for their valuable time.

Last but not least, I would like to thank my brother, Mike, and his family, Wendy, Travis, and Morgan, and to some extent their dog, Snickers, for putting up with me this academic year.
# Table of Contents

**Dedication** i  
**Acknowledgments** ii  
**List of Figures** vi  
**Abstract** vii  

## 1 Introduction  
1. Overview ........................................ 1  
2. Problem Statement ................................ 2  
3. Contributions ..................................... 2  
4. Organization ...................................... 2  

## 2 Background Information and Related Work 3  
1. Background Information ............................. 3  
2. Software Reverse Engineering ........................ 3  
3. Fingerprint Biometrics ............................... 5  
4. Fingerprint Matching ................................ 7  
5. Fingerprint Authentication Security ................. 11  
6. Fuzzy Extractors .................................... 13  
7. Related Work ....................................... 15  
8. Fuzzy Vault ........................................ 15  

## 3 Security Analysis 18  
1. Device Description ................................... 19  
2. Reverse Engineering Tools ........................... 21  
3. Initial Security Tests ................................ 21  
4. Software Structure Analysis ......................... 22  
5. Fingerprint Verification Analysis .................... 29  
6. Fingerprint Enrollment Analysis ...................... 35
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Fingerprint Ridges and Valleys</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Fingerprint Singularities</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Minutiae Classifications</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>AliceDevice System Structure</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>IDA Pro Produced Assembly Code for bAPI4_HMFVVerify</td>
<td>30</td>
</tr>
<tr>
<td>3.3</td>
<td>Array Address of ppEnrolledFeatures</td>
<td>33</td>
</tr>
<tr>
<td>3.4</td>
<td>pFingerImage ASCII Image</td>
<td>39</td>
</tr>
<tr>
<td>3.5</td>
<td>pFingerImage JPEG Image</td>
<td>39</td>
</tr>
<tr>
<td>3.6</td>
<td>Dynamic Enrollment Process</td>
<td>41</td>
</tr>
<tr>
<td>3.7</td>
<td>Abstract Reference Template Structure</td>
<td>48</td>
</tr>
<tr>
<td>3.8</td>
<td>Left: the original image. Right: the rotated and flipped image</td>
<td>49</td>
</tr>
<tr>
<td>3.9</td>
<td>Location, Type, and Orientation Mapping</td>
<td>50</td>
</tr>
</tbody>
</table>
Abstract

In response to user demands for mobile data security and maximum ease of use, fingerprint-secured mobile storage devices have been increasingly available for purchase. A fingerprint-secured Universal Serial Bus (USB) drive looks like a regular USB drive, except that it has an integrated optical scanner. When a fingerprint-secured USB drive is plugged into a computer running Windows, a program on this drive will run automatically to ask for fingerprint authentication. (When the program runs the very first time, it will ask for fingerprint enrollment). After a successful fingerprint authentication, a new private drive (for example, drive G:) will appear and data stored on the private drive can be accessed. This private drive will not appear if the fingerprint authentication fails.

This thesis studies the security of a representative fingerprint-secured USB drive referred to by the pseudonym AliceDrive. Our results are two fold. First, through black-box reverse engineering and manipulation of binary code in a DLL, we bypassed AliceDrive’s fingerprint authentication and accessed the private drive without actually presenting a valid fingerprint. Our attack is a class attack in that the modified DLL can be distributed to any naive user to bypass AliceDevice’s fingerprint authentication.

Second, in our security analysis of AliceDrive, we recovered fingerprint reference templates from memory, which may make AliceDrive worse than a regular USB drive: when Alice loses her fingerprint-secured USB drive, she does not only lose her data, she also loses her fingerprints, which are difficult to recover as Alice’s fingerprints do not change much over a long period of time.

In this thesis, we also explore details in integrating fuzzy vault schemes to enhance the security of AliceDrive.
Chapter 1

Introduction

Overview

There have been several high-profile security breaches of data on mobile storage devices, including the theft of an external hard drive owned by an US Department of Veterans Affairs employee, which had personal data of about 26.5 millions of people [8]. Mobile storage devices, such as a portable hard drive and a Universal Serial Bus (USB) drive, support data mobility and thus provide much flexibility. On the other hand, mobile storage devices also pose serious security challenges.

Data stored on a non-portable device can be protected by a surrounding operating system through access control mechanisms. Any access to such non-portable data requires entity authentication first and data access can be made fine-grained through access control. In contrast, data stored on mobile storage devices have no such protective mechanisms to depend on: once stolen, a mobile storage device is exposed without any obstruction to the adversary. Thus, for data on mobile storage devices, some alternative security mechanisms are needed.

In recent years, fingerprint-secured USB drives have appeared more and more frequently on the commercial market. A fingerprint-secured USB drive looks like a regular USB drive, except it has an integrated optical scanner. When a fingerprint-secured USB drive is plugged into a computer running Windows, a program on this drive will run automatically to ask for fingerprint authentication. (When the program runs the very first time, it will ask for fingerprint enrollment). After a successful fingerprint authentication, a new private drive (for example, drive G:) will appear and data stored on the private drive can be accessed. This private drive will not appear if the fingerprint authentication fails.

Historically, biometric authentications such as fingerprint authentication are used by highly secure systems, such as military information systems and nuclear plants. They tend to give us a perception of high-level security.
Problem Statement

In this thesis, we try to answer the following questions: Does a fingerprint-secured USB drive really provide high security? How hard is it to break the security of a fingerprint-secured USB drive? If it is insecure, how to enhance its security?

To answer these questions, we study the security of a representative fingerprint-secured USB drive, called AliceDrive, which is chosen randomly from the commercial market.

Contributions

In this thesis, we demonstrate that AliceDrive is highly insecure. Contrary to our initial thought, it is pretty straightforward to bypass AliceDrive’s fingerprint authentication through binary code manipulation. This authentication bypass is a class attack, as our modified DLL can be downloaded and used by any naive AliceDrive users.

An even more serious vulnerability is that in our study, we can retrieve AliceDrive’s fingerprint templates from memory. A human being’s fingerprints are relatively stable and remain unchanged for a long period of time, making fingerprint revocation very hard. As a result, losing fingerprint templates stored on AliceDrive has serious consequence in both security and privacy. In other words, our study shows that in some sense, a fingerprint-secured USB is worse than a regular USB drive: if stolen, the owner does not only lose his or her data, but also their fingerprints.

This thesis also studies the details of how to integrate fuzzy extractor schemes, specifically fuzzy vault, to improve the security (the confidentiality of both the private data and the fingerprint templates) of AliceDrive.

Organization

The remainder of this thesis is organized as follows. In Chapter 2, we give background information and related work. In Chapter 3, a chosen commercial representative fingerprint-secured USB drive is analyzed, and vulnerabilities are found and exploited. Chapter 4 provides an academic and unimplemented security improvement for the device. Finally, concluding remarks are given in Chapter 5.
Chapter 2

Background Information and Related Work

This chapter provides background information into the fields of research required for this thesis as well as related works. Background information consists of topics necessary for understanding the remainder of this thesis, while related works are the basis for our contributions.

Background Information

Because some research into software reverse engineering tools and techniques was required for the analysis of our fingerprint-secured USB drive, it is worth starting with a brief discussion of this subject before describing in more detail fingerprint biometrics, fingerprint matching, fingerprint authentication security, and finally fuzzy extractors.

Software Reverse Engineering

Reverse engineering is practically identical to the natural sciences, e.g., biology, chemistry, physics, etc. Both reverse engineering and these sciences are concerned with deriving information based on the evidence acquired through observation and testing. The distinction between the two is reverse engineering is the process of extracting information or design from anything man-made, while the sciences extract information or design from anything natural [7]. Reverse engineering is often used as a last resort when information is, for whatever reason, unavailable. In many situations, the desired information may be unattainable because the original information has been lost, or the owner of the information is unknown. If the information is valuable, the owner may be unwilling to release the information as it may encourage competition or conceal a company secret.

In terms of software, reverse engineering is the process of trying to understand the structure and functionality of a program or its components which usually means trying to recover the original source code from a program binary. In most situations original source code recovery is an overly ambitious endeavor as it is usually impossible. This is because there are many ways to write a program that result in similar machine code and many high-
level programming language elements are often omitted during compilation [7]. In some situations it is possible to completely recover source code as languages that compile to an intermediate language, such as Java and any of Microsoft’s .NET languages, can be easily reverse engineered into the original source code, if not something very similar. Intermediate languages provide much more information about the original code than machine language binaries, and therefore allow for near perfect re-creation of the original code. Because of this, designers that use these languages sometimes use obfuscation techniques to greatly obscure the code in such a way that any code acquired during reverse engineering is very difficult to interpret let alone reconstruct into its original form. Because of the difficulty, and more often impossible nature in recovering the original source code, reverse engineering usually requires detective like problem solving skills to infer what a program does and how it works based on clues.

To recover any information from a binary file requires reverse engineering tools. The basic categories of reverse engineering tools are:

- System monitoring tools: these tools leverage the fact that programs must communicate with other files, programs, and devices, and this communication can be observed. This information is helpful in determining a program’s structure and can give clues as to the effect of a program by disclosing what system calls are made, and what files and devices are accessed.

- Disassemblers: these tools take program binary files and convert them to assembly language. Any operation performed by the CPU can be translated into assembly rather easily as there are far fewer interpretations of CPU instructions to assembly code than CPU instructions to higher-level languages. Because of how easy it is to reverse engineer program binaries into assembly code, disassemblers are a primary tool for software reverse engineering, despite the fact assembly is harder to read and understand as compared with high-level languages.

- Debuggers: a debugger provides a dynamic view of the code by allowing one to observe the program as it is running which is often preferable to static code analysis. Other than providing ease in code traceability, dynamic analysis can reveal information that can not be determined from mere static analysis, such as stack and register values.

- Decompilers: these tools translate a binary program file into a high-level programming
language, which does not necessarily have to be the original language the program was written in. Often the code produced from a decompiler is more readable than assembly language produced by a disassembler, and therefore may be preferable in some situations.

The motivation to reverse engineer software is really no different from any reverse engineering endeavor, such as reverse engineering in an attempt to create similar or better products, or malicious motivations as with attackers trying to discover weaknesses. Sometimes reverse engineering is required to understand or fix legacy systems that are no longer supported or well known. Specifically for software, reverse engineering can also be used to analyze malicious software to determine its affects on a system and to derive a defense against it. For the purposes of this thesis, reverse engineering was used to determine system weaknesses for academic purposes and to propose possible solutions to these weaknesses.

Fingerprint Biometrics

Biometric recognition, also called biometrics, is the automatic recognition of individuals based on their physical and sometimes behavioral characteristics [1], which can be used for authentication, identification or data retrieval. Typically, authentication is used as a mechanism to enforce access control. Once a user is authenticated (i.e., their identity is verified as being from the list of valid users) the individual is granted access to something. What they are granted access to could be anything, such as a building, room, bank account, computer, hard drive, or an individual electronic document. Traditional authentication techniques use “something you know”, such as passwords and pin numbers, or “something you have” such as a key card, to verify a person’s identity, but biometric-based authentication uses “something you are”, which provides a degree of extra difficulty for an imposter to forge [10].

Naturally, not all physical or behavioral characteristics are well suited for biometrics. A biometric should adhere to certain characteristics for adequate biometric recognition as follows [1]:

- Universality: every person (or the vast majority of people) should have this characteristic.
- Distinctiveness: any two individuals should be distinguishable based off the charac-
teristic.

- Permanence: the characteristic should not change, or should change very little over time.

- Collectability: the characteristic should be measurable.

Along with these requirements, it is often important in practice for a biometric characteristic to take into account the following issues:

- Performance: the speed, accuracy, throughput and any resource or environmental factors which affect speed, accuracy and throughput of biometric recognition.

- Acceptability: the extent to which people are willing to use a biometric identifier or have a device scan this biometric identifier in everyday life.

- Circumvention: how easily it would be to forge the biometric characteristic of another person.

Fingerprints are an ideal biometric as they are well balanced among these properties [14, 15] and fingerprint scanners are relatively cheap, making them even more ideal for commercial biometric systems.

A fingerprint is composed of ridges and valleys. Most often ridges are indicated by black lines and valleys by white space in between the ridges (see Figure 2.1). The patterns and characteristics formed by ridges and valleys can be classified into three levels of abstraction [14, 15].

- Level 1: the first and highest level of abstraction, called the global level, is concerned with regions of a fingerprint where the ridges and valleys form distinctive shapes, as determined by high curvature, frequent ridge terminations, etc. These regions are called singularities (depicted in Figure 2.2) are subcategorized as loops, deltas, and whorls. These features can be useful in orienting prints for comparison, and also for general categorization to simplify search and retrieval, but do not provide enough distinctiveness for print matching.

- Level 2: called the local level, is concerned with more distinctive fingerprint characteristics called minutiae (meaning small details). With respect to fingerprints, this refers to how a ridge line may terminate (ridge ending) or divide (bifurcation).
- Level 3: the final and most detail-oriented level, called the very-fine level, is concerned with intra-ridge details including sweat pores, skin creases, ridge width and shape, and incipient ridges. This level provides the most distinctive fingerprint information, however requires high quality fingerprint images and therefore high resolution fingerprint scanners (1000 dpi or higher) to capture these features.

Figure 2.1: Fingerprint Ridges and Valleys

Figure 2.2: Fingerprint Singularities

Fingerprint Matching

Fingerprint authentication is traditionally achieved through fingerprint matching. An automatic fingerprint matching system has four main design components [10]:

- Acquisition: the process of converting the original fingerprint into a digital image.
- Representation: it is not often the case that fingerprint images are stored, as they vary
so much from scan to scan. Instead, more invariant representations of key features of the print are stored, which are called reference templates or simply templates.

- Feature Extraction: this is the process of extracting distinctive features from a biometric for comparison against a reference template or for enrollment of a template.

- Matching: this is the process of taking an input fingerprint and comparing it against a stored reference template.

Authentication of a fingerprint is an imperfect procedure, so when prints are compared, variability of fingerprint scans must be taken into account. This means during matching, two fingerprint reference templates are compared and a similarity score is returned representing the likelihood the two prints are from the same finger. Two fingerprints are considered matched if the similarity score is above some threshold.

The imperfect and difficult nature of fingerprint matching stems from the fact that features at each of the three levels of fingerprint characteristic abstractions, more so at the local and very-fine level, can vary substantially from scan to scan due to high variability called intra-class variation. The different classifications of these are described as follows [14, 15, 1]:

- Displacement: a finger may be placed at different locations on a sensor on different scan attempts.

- Rotation: a finger may be positioned at different angles with respect to the sensor surface each time a fingerprint is scanned.

- Partial overlap: displacement and rotation often cause part of the fingerprint area to fall outside the sensor’s scanning range.

- Non-linear distortion: the compression and stretching of a fingerprint due to the plasticity of skin. Capturing a fingerprint is the process of transferring a three dimensional physical fingerprint into a two dimensional image representation. Because skin is flexible, the force applied to the finger on the scanning device and/or friction with the finger and the scanning surface can distort the physical fingerprint which is captured as a non-linear distortion on a two dimensional image.
- Pressure and skin condition: finger pressure, skin conditions such as dryness, disease, sweat, grease, dirt, etc, as well as environmental factors such moisture in the air can result in nonuniform contact with the scanner surface. Nonuniform contact can lead to missing portions of a print or poor image quality regions.

- Noise: introduced by the scanning system, such as dirt or grease on the scanning surface.

- Feature extraction errors: these errors occur as a result of the imperfect nature of feature extraction algorithms. For example, a low-quality fingerprint image may result in the extraction of features that are not actually present in the fingerprint.

Of these classifications of intra-class variation, displacement and rotation are most correctable by software. Correcting displacement and rotation to match the reference template is called alignment, and is a crucial and necessary step for fingerprint matching. The method for aligning fingerprints varies depending on the method of fingerprint matching. There are three basic methodologies for fingerprint matching:

- Correlation-based matching: two fingerprint images, the reference image and an input image, are superimposed to find the alignment with the maximum correlation between pixels.

- Minutiae-based matching: minutiae points from two images (the reference image, and an input fingerprint) are compared to find the alignment with the maximum number of minutiae pairings between the two fingerprints.

- Non-minutiae feature-based matching: uses any other non-minutiae feature of a print for matching, including global and local texture information, level 3 features, shape and size of the finger, etc. This method is a catch all category used to classify any matching algorithm that does not implement the above two schemes. This method can be used for low quality fingerprint matching, as minutiae can be very difficult to extract from a poor quality image. Non-minutiae-based schemes can also be used in conjunction with minutiae-based matching to provide higher accuracy in print matching. Non-minutiae-based matching is also useful when the area of a print to be matched is so small that it only contains a few minutiae. Because this method of matching
can be used in numerous ways with any number of non-minutiae characteristics, the technique for matching and alignment is not standard.

Matching based on local-level fingerprint characteristics, minutiae, is the most popular method of fingerprint matching as well as the most analogous to how prints are matched by experts manually [14, 15]. Minutiae-based matching is ideal for many reasons. Minutiae provide ample distinctiveness, more so than level 1 (global) characteristics, and yet do not require the high quality print samples, as is the case with level 3 (very-fine) characteristics. Also, extracting minutiae from a fingerprint is fairly simple for medium to high quality fingerprints. A full fingerprint can have over 100 minutiae, yet only 12-15 minutiae are required for a high confidence matching [14, 15]. This means only relatively few high quality minutiae (as compared with how many are available on a fingerprint) are required for high confidence matching, further reinforcing the use of minutiae-based matching schemes.

Minutiae-based schemes extract minutiae points from a reference fingerprint image using some metric and store them in a reference template. During matching, minutiae are extracted from another fingerprint and compared against the reference template. An alignment is found that maximizes matches between the reference template and the minutiae extracted from the new fingerprint. Usually a similarity score is returned indicating how much the reference template and the input fingerprint match. If the similarity is above a predefined threshold, the validation attempt is successful.

The most common metric used for minutiae representation uses minutiae type, location, and orientation [17, 18, 3] or a subset of these, which are described below.

- Type: minutiae can be classified into numerous types, such as spurs, islands, points, lakes, crossovers, etc., as well as simple ridge endings and bifurcations (refer to Figure 2.3). Automatic minutiae extraction usually uses a more basic classification scheme consisting of bifurcations and ridge endings and sometimes a third category for unusual points of interest. For example, the American National Standards Institute standard ANSI/NIST-ITL 1-2007 [17] considers only four broad types of minutiae: bifurcations, ridge endings, compound minutiae (such as trifurcations and crossovers), and type undetermined. Similarly, CDEFFS (2008) [18] and M1/02-0142 [3] define three minutiae types: ridge endings, ridge bifurcation and another or unknown type for those minutiae which cannot be defined.
• Location: usually represented as a Cartesian coordinate pair, \((x, y)\). The problem with location of minutiae is the positioning of the origin of the Cartesian plane. The origin could be placed anywhere, and is dependent on the standard used.

• Orientation: specifies an angle, \([0-360)\), of a minutia. The basis from which the angle is derived is dependent on the standard. ANSI/NIST-ITL 1-2007 [17] defines the angle as being between the horizontal axis of the coordinate system and the direction that a ridge ending (or valley ending, for a bifurcation) points.

![Figure 2.3: Minutiae Classifications](image)

While type, location, and orientation are the most common metrics used to store minutiae, there is no uniform standard, and any number and combination of minutiae characteristics, including location, type, orientation (angle), and quality could be used. The metric choice is proprietary and as such, the algorithms used to extract and align minutiae are also proprietary.

**Fingerprint Authentication Security**

Security with respect to fingerprint based authentication, as previously mentioned, usually implies access control. Fingerprint-based access control systems are meant to provide high usability as no password must be memorized, nor a token carried. Fingerprints are innate, and the user only has to have the ability to press their finger to the sensor to authenticate. As great as fingerprint-based access control sounds, fingerprint matching techniques can have serious negative security implications depending on the nature of the system being secured.

If the system is meant to restrict access to a physical location, such as a building, room, amusement park, testing facility, etc., the security of the reference template is the only concern. The communication between the fingerprint scanner and the machine which
performs the print matching is also a concern but this issue is outside the scope of the security provided by fingerprint authentication.

To explain why the security of the reference template is important, consider the analogy of password authentication systems. In these systems the user claims an identity and provides a password as proof of this identity which is compared against a stored password. It is rarely the case, if ever, that the passwords to compare against are stored in clear text, because if the file which stores the passwords is compromised by an intruder or misbehaving system administrator, all the accounts of those whose passwords are listed will also be compromised. Instead, a cryptographic hash of the password is stored, called password verification data or PVD. During validation the attempted password is hashed and compared against the stored PVD. A cryptographic hash is considered a one-way function which is characterized by the ability to calculate a password hash easily; however it is difficult to recover from the hash the original password [22]. An attacker could still mount a dictionary attack on any compromised PVDs, however this attack is against the strength of the password and not the cryptographic hash.

For biometric based authentication systems this approach cannot be taken due to high variability from scan to scan (i.e., intra-class variation). For a cryptographic hash function \( h \) and two close fingerprint minutiae samples taken from the same finger, \( A \) and \( B \), \( h(A) \) and \( h(B) \) will be unpredictably different. For this reason, a fingerprint reference template cannot be hashed and must be stored in a clear text format, which carries with it the same security issues as storing passwords in clear text. The primary difference is, unlike with a password which can vary from one account to another and over time for the same person, a fingerprint does not and cannot be changed in response to an attack, making it very difficult to recover from such a compromise.

Exposure of the reference templates is a concern among all fingerprint-based authentication systems, but for systems that secure data using fingerprints, exposure of any private information is also a concern. If an attacker can gain physical access to the data store of the protected data, it may be possible to bypass fingerprint validation altogether, either programmatically or analyzing the physical device, and look directly at the raw bytes in storage, essentially giving complete access to all supposedly protected data as well as any stored reference templates.

With any system such as this, an attacker could always delete, modify, or physically
destroy the stored data, so security (access control) with respect to these systems refers specifically to confidentiality, as it is the only security characteristic which can actually be protected. Fingerprint matching alone only provides authentication, not confidentiality. In many situations, this is an acceptable risk as it is considered too difficult or too conspicuous for an attacker to access the physical data storage device. For example, most operating systems rely on this principle for user accounts. Removing the hard drive and analyzing the file allocation table could allow an attacker to read all stored data, and no authentication of any kind is necessary. While most of us accept the assumption this is unlikely for systems located in physically secure locations, we are less likely to accept it for portable data storage. Smaller data storage devices, such as laptops, external hard drives, USB drives, etc. are frequently taken outside protected locations, and because of their sizes they are less likely to be noticed missing, and are prone to theft and loss. If an attacker intercepts such a device, they can privately perform whatever procedures are necessary to compromise the data on the drive for as long as it takes. Therefore, it is no longer an acceptable risk to leave data unprotected in clear text.

To truly protect the security (confidentiality) of these systems, encryption must be used on all sensitive data. The question naturally arises, how does one create a cryptographic key from a fingerprint? Fingerprints vary substantially from scan to scan due to intra-class variation, so direct mapping to a cryptographic key from a fingerprint is not possible. To correct for intra-class variation, storage of some sort of reference data is mandatory but clear text storage of reference data may compromise the fingerprints the data represents. The issues of reference template and data security can both be solved using a cryptographic mechanism called fuzzy extractors.

**Fuzzy Extractors**

The concept of a fuzzy extractor was first introduced in [4] and is based off the fuzzy commitment scheme introduced in [13]. A fuzzy extractor reliably extracts a uniformly random secret (or nearly uniformly random secret), S, in an error tolerant way from noisy input (noisy referring to input that may have erroneous data intermingled with valid data). With respect to biometric data, a fuzzy extractor works by taking a biometric reference template, T, and computing the value P (a reference value used for error correction) and a secret S. P reveals little information about T or S hence it can be made public [22]. During
authentication, a fresh biometric with a reference template $T'$ is used with $P$ to recover
$S'$. If $T'$ is sufficiently close to $T$, then $S'$ should be the same as $S$. In order to validate a
successful authentication, $S'$ must be compared with $S$. To do this, a cryptographic hash
of $S$ must be stored, and the hash of $S'$ is compared against the hash of $S$. Thus for some
cryptographic hash function $h$, the set of required information for authentication using a
fuzzy extractor is \{h(S), P\}. Once authenticated, a user can simply be granted access to
the object or location in question or, with the case of information access control, $S$ can then
used as a cryptographic key.

A system using a fuzzy extractor for cryptographic key generation and recovery may
forgo the explicit authentication step of comparing cryptographic hashes and validate im-
plicitly by attempting to decrypt data using $S'$. If the user can read the unencrypted data,
then it can be assumed they are a valid user. This is computationally wasteful if $S'$ is
invalid, especially if the data to decrypt is rather large; therefore, the matching algorithm
as described above should be used to first determine if $S'$ is valid.

Since both elements of \{h(S), P\} reveal little about $S$ or $T$, if the set is compromised by
an attacker, the original reference template and the secret (the cryptographic key) are still
secure, seemingly solving all the shortcomings of biometric authentication. The problem
is the fuzzy extractor concept is rather abstract and can be applied to any biometric,
that is, concrete implementations may deviate somewhat from the above description and
the details of concrete implementation are omitted here. An implementation of a fuzzy
extractor depends on characteristics of the biometric data. Since essentially fuzzy extractors
measure closeness of an original biometric input to a subsequent query biometric, specific
implementations differ on the metric used for closeness. Popular proposed implementations
use such metrics as hamming distance, edit distance, set difference, and set intersection.

Hamming distance [9] implementations are better suited for biometric data that can
be easily represented as a string of bits. An implementation has been developed in [13].
Set difference and set intersection metrics work well with biometric data represented as a
set of elements. Implementations of set difference fuzzy extractors have been developed in
[4, 5, 11, 12], and a set intersection based fuzzy extractors have been developed in [20, 16].

A fingerprint reference template is most easily thought of as a set of elements, where
each element contains information on a specific minutiae point (such as type, location,
orientation, quality, etc.). As such, closeness metrics that use sets are more practical for
fingerprint biometric fuzzy extractors. Because of intra-class variation, the set of minutiae captured from the reference image and the set of minutiae captured from a subsequent print may be drastically different, not only in the ordering and number of elements (minutiae) captured, but also in the details of each element. Therefore, a set metric based fuzzy extractor for fingerprints must be error tolerant to handle these variations.

Related Work

In chapter 3 the security of a fingerprint-secured USB drive is analyzed, which is primarily accomplished by reverse engineering. Despite there being many manuals, books, and tutorials on the general concept of reverse engineering, there is no known related reverse engineering work for our specific analysis for this thesis. In chapter 4, we provide a hypothetical security improvement for the device, which involves fuzzy extractor mechanisms, specifically fuzzy vault. As such, the only real related work for this thesis involves the concept of fuzzy vault and fingerprint specific implementations.

Fuzzy Vault

The fuzzy vault scheme proposed in [11, 12] is a more specific implementation of a fuzzy extractor for set data. While somewhat more specific than the concept of a fuzzy extractor, fuzzy vault is still rather abstract, however it shows potential for implementation for fingerprint-based biometrics.

A fuzzy vault is a cryptographic construction where a user can lock a secret in a fuzzy vault using a set of data (the locking set). Another user can unlock the secret from the vault with a sufficiently close set to the locking set, called the unlocking set. The vault reveals little information about the secret or the locking set and therefore may be made public. The secret secured by the vault must be numeric, specifically an integer, and the size of the integer is limited, therefore the vault cannot secure complex data, however it can be used to secure something like a phone number, social security number, numeric user ID, or more importantly for our purposes, a cryptographic key, which can then be used to secure more complex data.

The vault is essentially a set of Cartesian coordinates (x, y). Some of these points lie on a polynomial P, while the vast majority of these points, called chaff points, do not. The
security of the vault is dependent on the number of chaff points. The secret secured by the vault is stored in one or more coefficients of P. To unlock the secret, a user must be able to select a set of non-chaff points (genuine points) out of the vault greater than some threshold value to form the unlocking set. The threshold is based on the degree of the polynomial, and is always one more than the supposed degree of P. Any coefficients not a part of the secret are chosen randomly from the universe of possible values (thus the “supposed” polynomial degree, as it is possible for the highest degree to have a coefficient of 0, consequently diminishing the polynomial degree, however depending on the universe of possible of values, this may be considered highly unlikely). Reconstruction of the original polynomial is achieved through some set error correction code, either based on set difference or set intersection. No matter what error correction scheme is used for any specific implementation, the underlying principal that allows for the reconstruction of P is polynomial interpolation.

Fuzzy vault is meant to allow for some discrepancies in the unlocking set, meaning some of the elements of the unlocking set may not lie on P; however, such an unlocking set may still be able to reconstruct P. This is the reason for error correcting code of some sort. The most obvious error correcting approach is to use a set intersection metric in which every subset of T elements is taken in an attempt to derive the original polynomial using Lagrange’s polynomial interpolation algorithm. For each attempt, the derived secret is hashed and compared against a hash of the genuine secret. Set intersection in this manner can become computationally infeasible with larger unlocking sets.

To simplify things, [11, 12] choose a set difference error correction for their analysis, specifically Reed-Solomon error correction code. The difference between set intersection and set difference is that with set intersection the unlocking set can be of any size so long as there are T genuine points in the set. With set difference, an unlocking set must have a certain percentage of valid elements in order recover P. The exact number of required valid elements for Reed-Solomon is (T+U)/2 where U is the number of elements in the unlocking set.

The generalized fuzzy vault scheme described above, when applied to fingerprints does not account for inherent issues of fingerprints. The primary issues involve encoding of fingerprint characteristics (specifically minutiae) into the vault during vault creation, aligning fingerprints due to intra-class variation prior to the unlocking set selection and the manner in which the unlocking set is chosen. In [16], the authors attempt to resolve these missing
details, and produce their own fingerprint-based fuzzy vault implementation to secure a cryptographic key. This implementation uses fingerprint minutiae for vault encoding, using minutiae location and orientation. These values are decomposed into binary representations that are then concatenated to form vault x values to be plugged into P to produce vault y values (P(x)). Unlike the originally proposed fuzzy vault scheme, this scheme uses set intersection to reconstruct P. In addition to storing the vault publicly, helper data used in aligning fingerprints is also stored. More specific details of this scheme will be discussed in chapter 4.
Chapter 3

Security Analysis

In this chapter, we analyze the security of our commercial representative fingerprint secured USB drive. Our goal is to evaluate the security the device provides and identify any vulnerabilities as a basis for security improvements. Security with respect to devices such as these refers to the protection of not only the private data but also the fingerprint reference templates used for authentication. Determining if the secret data can be recovered alone, while seemingly the most important goal in this analysis, is in many ways not as powerful as recovering the reference templates and deciphering their format. Conceivably, if the format could be deciphered and enough information is present, any recovered templates could be used to produce an artificial fingerprint (also known as a gummy fingerprint) which could then be used for any fingerprint biometric system, past, present or possibly future, where the recovered fingerprint is enrolled, including simple identity verification systems, physical access control systems, or more pertinent for this research, data access control systems.

It is worth noting that it is neither our intention to discriminate against the company that produces the USB device studied nor to discourage consumers from using this specific product. To avoid identifying and discrediting this fingerprint-protected USB drive, throughout this thesis the selected USB drive will be referred to by a pseudonym, AliceDrive. This pseudonym will also be substituted in diagrams or code references that originally used the actual device name. (Often, in cryptography literature secure communication is discussed in the setting of two users, Alice and Bob. For portable storage devices, such as the one used in this analysis, the communication isn’t between two individuals, rather one user wishes to store data securely for later use. Essentially, Alice is communicating with herself in the future, thus the name AliceDrive.)

Our analysis is focused toward software and we consider the hardware a black box. Software weaknesses with respect to security are more powerful as they do not require physically breaking or opening the device, and consequently do not pose the risk of permanently damaging and corrupting the hardware and any stored data. Also, once a software weakness has been identified, a software exploit can be created and disseminated allowing anyone to
compromise the security of the device with little to no skill.

The information known prior to analyzing the AliceDrive was purely from the user’s manual accompanied with the AliceDrive when it was purchased. This manual explains, from a user’s perspective, how the AliceDrive should be used, including how to connect/disconnect the AliceDrive to a computer, enroll a fingerprint, authenticate with a fingerprint, and back up user credentials. This user manual does not provide any insight to the internal structure of the device or the type of security provided.

Although little was known prior to our analysis with respect to the AliceDrive security, we did make some assumptions. First, we assumed the device was indeed flawed, in that it does not use any fuzzy extractor mechanisms to secure the data. This means that the data is either stored in clear text, or is encrypted but with a key that is stored in clear text somewhere on the device. Similarly, if no fuzzy extractor mechanisms are used, then the reference templates are also stored in clear text (i.e., the reference templates map out the location, orientation, and type of minutiae), however their specific format would still have to be deciphered. Secondly, we assumed that the device did measure the fingerprint with respect to its minutiae. This is the most common fingerprint biometric data used (see chapter 2), and therefore we felt that the AliceDrive would likely not deviate from the norm. These two assumptions were fundamental to our analysis in this chapter.

In the remainder of this chapter, we will first describe the AliceDrive from a user’s perspective and provide an overview of important reverse engineering tools used in this study. Following this, we provide our security analysis which consists of several steps, including initial security tests, analysis of the software structure, and the analysis of the fingerprint verification and enrollment processes. Our security analysis culminates at an authentication bypass and obtaining a significant portion of the AliceDrive fingerprint reference template format.

**Device Description**

The AliceDrive is an eight gigabyte fingerprint-based USB storage drive designed specifically for use with Windows operating systems (XP and newer). The device has three partitions: public, private and public read-only. When the device is inserted into a USB port, the public and public read-only partitions are mounted separately as two independent drives (for example F: and G:). These drives are made available immediately once the de-
vice is plugged in without authentication. Anyone may read from and write to the public drive as with an ordinary USB drive. The public read-only drive contains user manuals in many different languages and an Autorun.exe. As the name suggests, creating new files or modifying preexisting files on this drive is prohibited.

To access the private partition, fingerprint-based authentication is required. On the side of the device is a rectangular shaped fingerprint scanner. Authentication software runs automatically once the drive is inserted into a USB port. If for some reason it does not, the Autorun.exe in the public read-only drive can be executed to bring up the authentication interface. This interface asks the user to swipe his or her fingerprint on the scanner to access the private drive. At this time the user must swipe their fingerprint from the base of the fingerprint to the tip of the finger over the scanner. Through testing it was determined that the device is able to sense the direction the finger is swiped through the friction of the finger against the sensor. This mechanism prevents the positioning of the scanner relative to the fingerprint from affecting the ability of the device to interpret the fingerprint as being right side up so long as the finger is pulled across the scanner in the manner described.

The interface will continue to prompt for a valid fingerprint until a scanned fingerprint is recognized (i.e., the scanned fingerprint matches a previously enrolled fingerprint). Once authenticated, the public drive is unmounted and the private partition is mounted in its place, effectively replacing the public drive with the private drive, hence it is impossible to access the public and private drive simultaneously. After the private drive has mounted, the user can read and write to it as with any other mounted drive through the mechanisms provided by the operating system. To unmount the drive the user must right click the AliceDrive icon from the Windows system tray and select quit. This will cause the private drive to be unmounted and the public drive to be mounted in its place.

Along with simple data storage, the AliceDrive allows users to save usernames and passwords for website authentication and perform managerial tasks, such as backing up, restoring, resetting, removing and adding user profiles. The device allows for the storage of at most ten user profiles. Since each profile is allowed to save one fingerprint, the device can only store ten fingerprints. Through testing, it was determined that a previously enrolled fingerprint may not be enrolled twice, so the device may only store up to ten unique fingerprints.

The AliceDrive provides four authentication preferences: fingerprint only, password only,
or both. Any authentication involving passwords was ignored as the focus of this analysis was strictly on fingerprint-based authentication.

**Reverse Engineering Tools**

Reverse engineering tools were required for the security analysis of the AliceDrive. The primary tool of choice was IDA Pro Advanced version 5.2.0. An unofficial guide for IDA Pro [6] was used as a reference. IDA Pro is an all encompassing reverse engineering tool, and while it is very powerful, containing many capabilities, for our purposes it was only used for its basic decompiler, disassembler, and debugger.

Version 5.2.0 does not have all the capabilities of newer versions. One capability desired which was not available was the ability to alter a binary and save the changes back to the file. To accomplish this, the tool OllyDbg version 1.10 was used. OllyDbg is a free assembler-level debugger that allows a user to disassemble a binary, alter or add to the assembly code, and commit the changes back to the file. This functionality is necessary for manipulating any AliceDrive libraries to create an authentication bypass.

**Initial Security Tests**

Before attempting to reverse engineer the AliceDrive, some basic tests were performed to validate that the device’s security was sufficient enough to require reverse engineering tools and techniques, i.e., the security cannot be bypassed simply using the operating system or using some automated tool.

The first test was to determine if the addition of a file to the private drive had a visible effect on either of the two public drives. A document was added to the private drive, and then the remaining available space of both the public and public read-only drives were compared to their original sizes before the file was added. No changes were observed indicating they indeed were separate drives.

The next test was to determine if the reference templates were stored hidden on one of the three drives. To test this, a new fingerprint was enrolled and the remaining available space of each of the three drives was checked for differences. No changes were observed indicating that there is another storage partition on the drive never made visible.

The device was also plugged into a Linux machine and the disk dump command (dd)
was used. If the drive somehow does not protect against a disk dump, it is possible to view the raw bytes of the entire drive and recover all information stored on the device, including the private partition and the reference templates. Disk dump could only dump ten megabytes of the drive’s eight gigabytes, and could not determine any viable partitions on the drive. The ten megabytes dumped did not contain data on the private partition or any user profile information, such as user names or fingerprint templates.

Not surprisingly these results indicated that the device’s security is not completely transparent, and cannot be simply bypassed without reverse engineering. The next step was to understand the device’s software (the software that executes when Autorun.exe is called), which we will refer to as the AliceDrive controller. Since the controller manages, among other things, authentication and making available the private drive, understanding how it works and where key functionality is performed provides locations of high interest for further detailed reverse engineering and analysis.

**Software Structure Analysis**

To begin analysis of the controller program, the executable was first located; however the controller was not explicitly installed nor were there any apparent directories containing any AliceDrive executables or library files in the Windows programs directory.

To determine the location of the controller software, the authentication interface was started and the Windows processor manager was opened. In the list of running processes was a process called AliceDrive.exe. Searching the C drive for the executable name showed it was located in the directory C:\Documents and Settings\All Users\Application Data\AliceDrive (this directory listing is for a Windows XP operating system and may differ on later Windows OS releases). The directory “Application Data” is a hidden directory apparently used by many programs to discreetly store data and binaries.

In addition to AliceDrive.exe, the AliceDrive directory contains the following dynamic link library (DLL) files:

- LTTS1NDUT176.dll
- LTTUSB.dll
- PasswordBank.dll
- PTFVLib.dll

- PTSDK4_SS500A_PTFV.dll

All files here, including AliceDrive.exe, are assumed parts, or modules that work together. Therefore, each file of the AliceDrive directory will be referred to as a module.

Since the controller software can run only on Windows operating systems, it was our hope that each module was developed in a .NET language, which is compiled to an intermediate language (Microsoft intermediate language). Many tools, including IDA Pro, can decompile intermediate languages into a nearly perfect copy of the original source code. Unfortunately, IDA Pro determined that all the AliceDrive modules were in machine code, therefore each file was disassembled (converted to assembly language) as opposed to decompiled into a high level language for initial analysis (decompilation is possible with machine code binaries, but the resulting code is usually as complicated if not more so than the assembly alone).

Code obfuscation is a technique used when the source code is easily recoverable, such as if the code is compiled to an intermediate language, or for security critical systems that depend on the secrecy of their algorithms (security through obscurity). Our fear was that because of the current state of secure biometrics, and their lack of commercial implementation, the AliceDrive does not provide any real computational security, and will therefore rely on the obscurity of its execution. If such obfuscation techniques were used with the AliceDrive, it would make static analysis of the assembly very difficult as any meaningful function names would be replaced with nonsense. After disassembling each module, it was apparent no obfuscation techniques were used as functions had descriptive names that clearly indicated their functionality.

The modules of the AliceDrive are designed for Windows, and as such these files follow the executable file format specific to Windows called the portable executable (PE) format. One important characteristic of PE files useful for analyzing the structure and functionality of modules are imports and exports. In each PE file there is a table of imported and exported functions. An import table contains a list of functions used by the current module but located in some other module. Each function in the import table is identified by its function name and the module it is contained in. When a PE file is loaded for execution, each function in the import table is found in the appropriate module's export table. An export table provides a list of public functions and an address for each function called a
relative virtual address or RVA, used to determine the actual address of the function during execution.

Import tables can reveal a lot about the relationships among modules, and export tables reveal the provided functionality of each individual module (provided the names of the functions are not obfuscated). Ideally an import table would be a complete list of all other functions used by a module but this is not always the case. Linking DLL files can be done in one of two ways, statically or at run time. If a DLL is linked statically, a reference to any linked functions are listed in a PE file’s import table. If a DLL is linked at runtime, the address of the required DLL function is acquired while the program is running. In these situations an import table entry is not necessary since the function address is resolved dynamically. This can make deciphering a program’s structure and relationships not as transparent since the import tables cannot be trusted to contain all imported functions. Instead of only checking the import table, locating all function calls to a Windows API called GetProcAddress can be used. This function determines the address of a function at runtime, effectively performing runtime linking of a library. Using import tables and a search for GetProcAddress revealed the AliceDrive software structure (as seen in Figure 3.1) and analysis of the function names present in each module’s export table in relation to this structure hinted at their responsibilities.

![Figure 3.1: AliceDevice System Structure](image)

Based on the observed AliceDrive structure, AliceDrive.exe communicates with only two other modules, PasswordBank.dll and PTSDK4_SS500A_PTFV.dll. It was apparent from the export table that PasswordBank.dll exported functions related to web browsing, as seen
Most functions either have “IE” (presumably Internet Explorer) or “Url” in the function name. As mentioned in the AliceDrive description, part of the device’s functionality is to save usernames and passwords for website authentication so a module devoted to this kind of functionality was expected. Since this module apparently is not involved in verification or any security related functionality (access to the private directory or the reference templates), then if its not provided by AliceDrive.exe it must be handled through PTSDK4_SS500A_PTFV.dll either directly or indirectly through the remaining three modules. In fact, it can be easily verified by viewing the export table PTSDK4_SS500A_PTFV.-dll (seen below) that this module is probably responsible for, among other things, mounting the private drive as well as enrolling new fingerprints.
• bAPI4_GetImage
• bAPI4_GetLogicalDrive
• bAPI4_HMFVCloseLib
• bAPI4_HMFVEnroll
• bAPI4_HMFVGetParas
• bAPI4_HMFVOpenLib
• bAPI4_HMFVSetParas
• bAPI4_HMFVStartEnroll
• bAPI4_HMFVVerify
• bAPI4_Inquiry
• bAPI4_MediaChange2Cdrom
• bAPI4_OpenDevice
• bAPI4_OpenSensor
• bAPI4_ReadSecureArea
• bAPI4_StopImage
• bAPI4_WriteSecureArea
• bMediaChange2Flash

Several functions jump out as potential functions of interest, such as bAPI4_ReadSecureArea, bAPI4_WriteSecureArea, bAPI4_HMFVVerify, and bAPI4_HMFVEnroll.

Many functions in PTFLib.dll, LTTS1NDU176.dll, and LTTUSB.dll have very similar names to those in PTSDK4_SS500A_PTFV.dll implying that PTSDK4_SS500A_PTFV.dll may serve as some kind of controller or application programming interface (API) for security related functionality, providing a common interface but delegating most work to subsidiary programs, as the structure in Figure 3.1 implies. Most of the functions contained in PTSDK4_SS500A_PTFV.dll begin with the prefix “bAPI4”, probably representing the
type and version of whatever API the module provides. Even the name of the module itself implies this since it contains “SDK” most likely standing for software development kit. Often software development kits are simply APIs.

Since SDKs and APIs are usually standardized and widely used, our hope was documentation for PTSDK4_SS500A_PTFV.dll would be publicly available online. A Google search was used for “PTSDK4_SS500A_PTFV.dll” to find any documentation for this module. This search yielded no results, so each module name of the AliceDrive system structure was also searched in hopes of finding any related material whatsoever. The only module that had any meaningful Google results was PTFVLib.dll. The result was a document “A Programmer’s Guide for Fingerprint’s SDK” [23]. This document contains descriptions and parameter lists for what the authors of this document call “SDK functions” as well as the names of related fingerprint SDK modules for their product. The document is rather vague and does not give any indication as to what the described system or product actually is and which one of the SDK modules mentioned actually contains the functions listed, however reviewing the list of fingerprint SDK modules reveals some information. The names and descriptions of each SDK module are as follows:

- WISCMOS2.sys: a driver program.
- PTSDK4_WISCMOS2_PTFV.dll: a controller for fingerprint verification and a sensor.
- WisCMOS2.dll and DEC207.dll: controls the sensor and USB communication.
- PTFVLib.dll: a fingerprint verification kernel.

Apparently there is some sort of sensor and USB drive involved, which sounds very similar to the AliceDrive. Comparing what can be observed with the AliceDrive with each of these modules, it can be seen that whatever device this document was intended for is similar, if not identical in system structure to the AliceDrive. WISCMOS2.sys is described as a driver which is analogous to AliceDrive.exe which we previously described as a controller providing all functionality of the device. The naming convention for PTSDK4_WISCMOS2_PTFV.dll is very similar to PTSDK4_SS500A_PTFV.dll, and its description matches the assumption made before that it is probably a controller or API. Looking at the export tables of both LTTUSB.dll and LTTS1NDU176.dll indicates these functions have something to do with the sensor and USB communications, even the name
LTTUSB.dll implies as such, which is consistent with the descriptions of WisCMOS2.dll and DEC207.dll. Finally, PTFVLib.dll, which curiously is found both in this product and the AliceDrive, is described as responsible for fingerprint verification. Observing the export tables of the AliceDrive PTFVLib.dll reveals function names containing “enroll” and “verify”.

More evidence became apparent that this SDK document was indeed relevant to the AliceDrive system when reviewing the functions it described. Seventeen functions are described in this document, and of these eleven have identical names to those found in PTSDK4_SS500A_PTFV.dll. The function of the most interest for proving the validity of this document was bAPI4_HMFVCLoseLib. Presumably this stands for close library. What is interesting is the unusual spelling of close with a capital L. This could have been intentional, that is the function does not actually stand for close library, however it seems like a typographical error in the API. Since the same spelling is found in the AliceDrive SDK module, this seemed to indicate that while not all of the functions in the AliceDrive SDK module are represented in this document, perhaps both the AliceDrive and this product are pooling from the same standard API source.

Finally, before more detailed analysis could be performed on the disassembled code, it was important to determine the calling convention used by each AliceDrive module. A calling convention defines how functions are called and returned in assembly language, i.e., how parameters are received by functions, the order of parameters, and where return values are placed. IDA Pro was able to determine the calling convention used by each module is cdecl. This is the standard C and C++ calling convention. In this convention parameters are passed to a function using the stack. The parameters are pushed on the stack from right to left, meaning the last parameter is pushed first and the first parameter is pushed last. It is the responsibility of the caller to restore the stack pointer back to the original position after the function returns. Since the stack grows from higher addresses to lower addresses, the caller can reset the stack pointer by incrementing the stack pointer register, ESP, by the number of bytes pushed as parameters. At the end of every cdecl function call there is a RET command, which pops the instruction pointer pushed when the function was called. Execution then resumes at this instruction pointer’s address. If a function returns a value, it is stored in the EAX register sometime before the function returns.
Fingerprint Verification Analysis

Fingerprint verification is a logical place to start detailed analysis since a verification function may be key to not only bypassing authentication (since after verification the private drive is mounted) but also identifying the location of the AliceDrive reference templates (since they must be required for comparison to authenticate). According to the SDK document the function responsible for fingerprint verification is bAPI4_HMFVVerify. This function is one of the eleven also contained in the AliceDrive SDK module. The next step was then to analyze this function both statically and dynamically to determine if this function is indeed responsible for fingerprint verification, and if the parameters and return value of this function match those described in the SDK document.

The disassembly of bAPI4_HMFVVerify, as seen in Figure 3.2, appears to affirm our prior assumption of that the PTSDK4_SS500A_PTFV.dll is a controller API since the assembly code essentially does nothing more than pass its parameters on to the function referenced in the DS register, apparently delegating responsibility to subsidiary modules. IDA Pro is able to decipher the value in DS register as pointing to a function called bPTFVVerify which is located in the PTFVLib.dll module. According to the SDK document bAPI4_HMFVVerify returns 1 (true) if verification is “OK”, and 0 (false) if verification is “NG”, and takes the following parameters in this order:

- int iResolution: the image resolution.
- int iWidth: the image width.
- int iHeight: the image height.
- BYTE *pFingerImage: pointer to an image buffer ready for verification.
- BYTE **ppEnrolledFeatures: pointer to the enrolled features buffer addresses.
- int iEnrolledNum: the number of enrolled features.
- int *piMatchedID: returned pointer for the matched ID.
- int *piStatus: returned pointer for the verification status.

Static analysis of the function did not show any discrepancies with the SDK document, however, the values of these parameters and where exactly this function is called during execution cannot be verified statically.
IDA Pro provides debugging capability. A debugger allows the code to be analyzed while it executes (dynamic analysis). Breakpoints can be set at any location in the assembly code to pause execution and view the values of the stack and various registers and flags. It is also possible to step line by line through the assembly code to view the execution path taken.

Using IDA Pro’s debugger for dynamic analysis, it was determined that bAPI4_HMFVVerify was called every time a fingerprint was scanned for authentication. Forgoing a detailed analysis of each parameter passed to this function at this point, the assumption was made that this function was correctly represented in the SDK document. Under this assumption, the two values of interest are piStatus and the return value since these two are described as indicating something about the verification status.

Since bAPI4_HMFVVerify does nothing more than pass its parameters on to bPTFVVerify, analysis of bAPI4_HMFVVerify is essentially an analysis of bPTFVVerify. As mentioned before, the return value in the cdecl function calling convention is stored in the EAX register. bAPI4_HMFVVerify does not store any value into this register after the function call to bPTFVVerify, therefore if bPTFVVerify sets the EAX register, this value will be considered the return value. Indeed bPTFVVerify does set EAX before returning. Observing the assembly statically revealed that only the values 0 and 1 are stored in this register before

Figure 3.2: IDA Pro Produced Assembly Code for bAPI4_HMFVVerify
returning. During dynamic analysis, there was no situation discovered where the value 0 was assigned into the EAX register, no matter if verification failed or was successful. Perhaps if verification is “OK” (a return value of 1), no erroneous situations occurred, not necessarily that verification passed, and if verification is NG, verification is not good, or no go, because of some unexpected situation resulting in the inability of the algorithm to continue its verification attempt.

If the return value is used for what is essentially the algorithm status, then perhaps piStatus is actually the result of the verification attempt, i.e., verification success or failure. Static analysis of bPTFVVerify showed that the hexadecimal values FFFFFFFF, FFFFFFFF, 1 and 2 are stored in this variable (that is the pointer stored in this location points to these values). It was expected that only 0 and 1 would be stored in this parameter, indicating failure and success respectively, however observing a larger universe of values for piStatus indicated verification is not that simple. Reviewing the SDK document, there are actually four possible predefined values for piStatus:

- HMFV_STS_VF_SUCCESS = 2
- HMFV_STS_VF_FAIL = 1
- HMFV_STS_VF_POORIMG = -1
- HMFV_STS_VF_ERROR = -2

The IDA Pro debugger was used to verify that these values are stored in piStatus in the appropriate situations. It was observed that upon successful verification, the value 2 was stored in piStatus, but if verification failed, no value was set. When piStatus is originally passed as a parameter, the value pointed to by piStatus was 4 and after invalid verification, piStatus still remained 4. If the image was poor, piStatus pointed to the value of FFFFFFFF hex, which is two’s complement for -1. Much like for piStatus of 1, a situation where -2 (FFFFFFFE hex) is stored in piStatus was never observed however, static analysis does verify that both 1 and -2 are possible values of piStatus. Knowing these values definitively indicated we could conceivably bypass authentication (to be discussed in the section Break #1).

Changing our focus from how verification is achieved to the acquisition and deciphering of reference templates, we continued to focus on bAPI4_HMFVVerify function. To verify
a fingerprint as is done in bAPI4_HMFVVerify, there has to be a comparison between the stored reference templates and a fingerprint query. The fingerprint query may be in the form of a data structure similar in content and structure to the reference template, in which case preprocessing of the fingerprint image is required, or the actual fingerprint image may be passed to this function, and processing of the fingerprint image into a data structure for comparison is performed within the verification function. Either way, the previously stored reference templates must be supplied for comparison. bAPI4_HMFVVerify has a parameter ppEnrolledFeatures, which is a two dimensional array and according to the SDK document, each array of ppEnrolledFeatures contains “enrolled features”. A reference template essentially is a list of enrolled features, minutiae, so the assumption was that this parameter contains the reference templates for every enrolled fingerprint on the AliceDrive.

A total of five fingerprints had been enrolled on the device at this point in analysis (more than five had been enrolled prior to analysis, however these were deleted previously), so five array addresses were expected after dereferencing the ppEnrolledFeatures pointer address. The parameter iEnrolledNum of bAPI4_HMFVVerify which is described in the SDK document as the number of enrolled features also contained the value five, further validating that ppEnrolledFeatures contains the reference templates for the AliceDrive. Extracting the addresses ppEnrolledFeatures points to yielded five evident addresses (as seen Figure 3.3). Since the machine used for this analysis uses an Intel architecture, these addresses are represented in little-endian notation. Converting them into a readable format yielded the following hexadecimal addresses:

- 00483840
- 00483A34
- 00484010
- 00484204
- 004843F8

Having the starting location of each reference template does not reveal the size of each template. Based on these array starting addresses, the possible sizes in bytes for each array, except for the fifth array, can be determined and are as follows (array one through four respectively):
Figure 3.3: Array Address of ppEnrolledFeatures

- 500
- 1500
- 500
- 500

According to the SDK document the maximum template size is 500 (presumably bytes), which seemingly holds true for three arrays; however the second array starting at address 00483A34 is a maximum of 1500 bytes long. It was first assumed that the arrays of a two dimensional array are contiguous, but judging from these results it is either the case that that the array sizes are variable beyond 500 bytes, or arrays do not have to be in contiguous memory space. Since additional fingerprints had been enrolled prior to analysis but were deleted, it is possible that these reference templates are never physically deleted, just not explicitly referenced (this was validated later after our security analysis). Since the SDK document has held true for the AliceDrive SDK, the assumption was made that each reference template does not exceed 500 bytes.
Based off of this assumption, to determine the sizes of the AliceDrive reference templates, another assumption was made that during a verification attempt of an invalid fingerprint, every reference template must be compared against the invalid fingerprint in its entirety. Using a special debugger breakpoint called a hardware breakpoint enables the IDA Pro debugger to pause execution if a specific memory address is read from or written to. During verification, it was assumed that the entire reference template structure of each reference template stored must be read in for comparison purposes if a query fingerprint is invalid (if a print were valid, it would not necessarily the case that each reference template must be compared). Based off our assumption of each reference template size and our observations of the reference template addresses, hardware read/write breakpoints were set at the following hexadecimal memory addresses:

- 00483840 (array 1 starting address)
- 00483A33 (array 1 ending address)
- 00483A34 (array 2 starting address)
- 00483C27 (assumed array 2 ending address)
- 00483C28 (assumed irrelevant address immediately after array 2)
- 0048400F (assumed irrelevant address immediately before array 3)
- 00484010 (array 3 starting address)
- 00484203 (array 3 ending address)
- 00484204 (array 4 starting address)
- 004843F7 (array 4 ending address)
- 004843F8 (array 5 starting address)
- 004845EB (array 5 ending address)
- 004845EC (assumed irrelevant address immediately after array 5)

If our assumption were true, then every breakpoint except for the breakpoints at 00483C-28, 0048400F, and 004845EC should have been activated during an invalid fingerprint verification attempt; however the assumption was not upheld by the results. The results
were surprisingly inconsistent and not easily repeatable. For the first attempt only two breakpoints were activated during an invalid verification attempt at location 00483840 and 00483A34 (the starting addresses of the first and second arrays). Further testing had different results, in that with different fingerprints different arrays were read from. The results did have some consistencies in that no breakpoint located outside our assumed array locations were ever activated, and neither were any breakpoints located at the end of our assumed templates. These results may indicate that perhaps global fingerprint characteristics such as fingerprint type, or areas of high curvature, are used to obviate the need to compare against every stored reference template. Also, the fact that each array was not read in its entirety (500 bytes), indicates all 500 bytes of the template are not used.

Without the size of the reference templates it is difficult to definitively obtain a reference template for analysis. Although the results of this experiment did indicate an upper bound for reference template sizes, the exact sizes still must be determined. Since the entire enrolled feature data structure could not be extracting in the running program, new reference templates would have to be extracted through function calls to the available enrollment functions of the AliceDrive SDK.

Fingerprint Enrollment Analysis

PTSDK4_SS500A_PTFV.dll exports two functions of interest for fingerprint enrollment, bAPI4_HMFVStartEnroll and bAPI4_HMFVEnroll. The SDK document indicates both of these functions are indeed involved in enrolling new fingerprints, and dynamic analysis confirms this. According to the SDK document, bAPI4_HMFVStartEnroll should be called to start fingerprint enrollment and takes only one parameter, int iDefaultEnrolledTimes which is the number of times to enroll the fingerprint (recommended 10).

Using the IDA Pro debugger, we observed that bAPI4_HMFVStartEnroll is only called once and the parameter used is 10, after which bAPI4_HMFVEnroll is called several times. The SDK document describes bAPI4_HMFVEnroll as actually creating the enrolled features. The parameters for bAPI4_HMFVEnroll are as follows:

- int iResolution: the image resolution.
- int iWidth: the image width.
- int iHeight: the image height.
• BYTE *pFingerImage: pointer to an image buffer ready for verification.

• BYTE *pEnrolledFeatures: pointer to a buffer for returned features.

• DWORD *pwEnRetSize: returned pointer to the enrolled features size.

• int *piStatus: returned pointer to the enrollment status.

The function return values for both bAPI_HMFVStartEnroll and bAPI4_HMFVEnroll, as with all functions in the SDK document are described the same as the return value for verification. Therefore the assumption was made that the return value has the same meaning for all SDK functions, indicating non-erroneous algorithm operation, and as such we ignored the return value.

The parameters of bAPI4_HMFVEnroll are very similar to the parameters of bAPI4_HMFVVerify with only a few differences. As expected, both of these functions take in a parameter described as a fingerprint image, pFingerImage. This image may be an actual image, however it could be some other data structure which represents a fingerprint image (e.g., a reference template). In order to create a program to call the AliceDrive enrollment functions, the structure of this parameter must be determined. Since this parameter is described the same in both the verification and enrollment functions, discovering the structure of this parameter may also shed some light on how the verification process behaves.

Both verification and enrollment are passed additional parameters specifying resolution, height, and width of the image, already indicating pFingerImage is indeed an image and not some other data structure. The fact that this information is present at all indicates that the image is most likely not stored in some predefined image format, such as a bit map or JPEG, since height, width and resolution information are stored within the data structures for these formats. Just to make sure, various sizes of the pFingerImage were extracted and stored. The linux “file” command was used with these stored binary streams to see if a recognizable image header could be found. As long as a recognizable header can be found in the data file, a complete file is not necessary for file type identification. All binary streams used with this command were unrecognized.

Since no known image format was present in pFingerImage and the resolution, width, and height are required parameters for both verification and enrollment, it was assumed that if pFingerImage is indeed an actual image, then it must be in a raw byte format. In
order to attempt to extract an image from raw bytes, all information about the image must be gathered. According to the SDK document, the resolution of the image is a defined constant at 500, however no definitions are set for the width and height. Through dynamic analysis of both enrollment and verification, 500 was observed as the value for iResolution, and the width and height were always observed to be 280 and 320 respectively. Knowing definitively the width and height of pFingerImage does not directly indicate the image’s byte length as there are many schemes for byte representation of an image. Before exhaustively checking each scheme, the assumption was made that a color image is unlikely since color adds an extra degree of complexity for minutiae extraction, and it requires higher priced scanners which are unlikely found in the relatively cheap AliceDrive.

Consulting the SDK document revealed there is a function also present in the AliceDrive SDK called bAPI4_GetImage. Through dynamic analysis, it was determined this function was called every time a fingerprint was scanned, either for authentication, or enrollment. This function supposedly returns a gray fingerprint image in one of its output parameters, called picture, which is a pointer to a byte array. Because this parameter is not called pFingerImage, there was some doubt as to whether the two parameters are related, however, since it is the precursor to any function that requires an image, starting with the assumption pFingerImage is a gray scale image was logical.

A black and white image could be represented with one bit per pixel, but a gray scale image is typically represented as one byte per pixel. If this is the case, it is known the image has a width of 280 and a height of 320. Assuming these values correspond to pixels means the image has 280 * 320, or 89,600 pixels and consequently 89,600 bytes.

Extracting 89,600 bytes from IDA Pro dynamically can be a bit messy, however analyzing the pFingerImage parameter at runtime showed the pointer returned by bAPI4_GetImage for the picture parameter was the same pointer for pFingerImage, whether used for verification or enrollment, therefore the returned picture of bAPI4_GetImage is the same as pFingerImage. A program was then developed to call bAPI4_GetImage and recover an image for analysis. This required a more detailed analysis of what other library function calls are necessary in order to call this function. The functions called before a fingerprint scan from the AliceDrive SDK were observed to be bAPI4_OpenDevice, bAPI4_OpenSensor, and bAPI4_CheckSensorStatus. Reviewing the descriptions in the SDK document indicated that bAPI4_CheckSensorStatus was not crucial to capture a fingerprint image, but instead
checked the current status of the sensor. The sensor status can either be ready or not found, and since the scanner should always be found, this function was omitted for our program. The remaining functions were then used in the bAPI4_GetImage test program.

bAPI4_OpenDevice and bAPI4_OpenSensor do not require any parameters, so the only function call of any complexity was to bAPI4_GetImage. The parameters for bAPI4_GetImage are described in the SDK document as follows:

- BYTE *picture: returned pointer to an image buffer.
- int timeout: timeout period for getting the image (in milliseconds).
- int iResolution: image resolution.
- int *piHeight: returned pointer to the height of the image.
- int *piWidth: returned pointer to the width of the image.

Based on the description for the timeout parameter, the value of 1000 (1 second) was arbitrarily chosen. The value for the parameter iResolution was set to 500 per the specified value in the SDK document and the observed value through dynamic analysis. The picture buffer was initialized to a size of 89,600 bytes. After the function call was made, the picture array was stored in a file. After running our test program, but before attempting to translate the raw binary obtained into an image, the file was opened up in a text editor. Luckily the characters represented by the binary were all printable, and observing them showed some kind of pattern. Since a gray scale image usually has a high contrast, it should be possible to see the image by simply looking at its ASCII representation. As expected, the returned piHeight and piWidth values were 320 and 280 respectively. Editing the binary in a text editor, a new line character was placed at the end of every 280 character string. Once each new line character was in place it was quite evident that this binary was indeed an image. Reducing the font size in the text editor practically produced an image, as seen in Figure 3.4. Having definitively ascertained that each byte represents a pixel, and only gray scale images can be represented with one byte per pixel, it was then easy to convert this raw binary image into an actual image (Figure 3.5).

Now having determined the format of pFingerImage, it was not only possible to create a program to call bAPI4_HMFVEnroll and obtain a reference template, but to pass it any desired image for experimentation. Deciphering pFingerImage also sheds some light into
As with creating a program to call bAPI4_GetImage, a dynamic analysis of the enrollment process was required to determine what other function calls were necessary for the enrollment process to complete an enrollment program. Systematic dynamic analysis of the enrollment process yielded the AliceDrive SDK function call structure found in Figure 3.6. The functions required for setting up the enrollment process are bAPI4_HMFVOpenLib, bAPI4_HMFVStartEnroll, bAPI4_HMFVSetParas, and bAPI4_OpenSensor. Of these only bAPI4_HMFVSetParas and bAPI4_HMFVStartEnroll require parameters. As mentioned before bAPI4_HMFVStartEnroll has one parameter, which was discovered to be 10. The parameters of bAPI4_HMFVSetParas are int iParaID and int iParaValue. The values of these parameters were observed through dynamic analysis to be 100 and 0 respectively. After enrollment setup has been accomplished bAPI4_GetImage is called. The image returned by this function is fed into bAPI4_HMFVVerify for verification. If the image is successfully verified (verification parameter piStatus = 2), the system will not attempt to enroll the fingerprint, and will prompt for another scan repeatedly apparently with no limit. This effectively prevents a user from enrolling the same fingerprint multiple times.

If the fingerprint has never been previously enrolled, bAPI4_HMFVEnroll is called. The system will continually cycle through getting a fingerprint image, verifying the image, and enrolling if the fingerprint has never been enrolled while piStatus of bAPI4_HMFVEnroll returns a CONTINUE flag. The SDK document defines the following flags for piStatus for bAPI4_HMFVEnroll:
• HMFV_STS_EN_CONTINUE 1
• HMFV_STS_EN_SUCCESS 0
• HMFV_STS_EN_FAIL -1
• HMFV_STS_EN_NOINIT -2
• HMFV_STS_EN_TOOMANY_POORIMG -3
• HMFV_STS_EN_TOOMAY_TRIALS -4

Enrollment will terminate if any value other than SUCCESS or CONTINUE is found in piStatus. If SUCCESS is observed, the value stored in the output parameter pEnrolled-Features has been properly filled. At any other time, this output parameter contains only a seven byte header followed by all zeros. The seven bytes observed in an unsuccessful pEnrolledFeatures is always the same, and is as follows for each byte:

• 0x78
• 0x01
• 0x33
• 0x00
• 0x78
• 0x01
• 0x33

After enrollment has returned a SUCCESS flag, one final call to bAPI4_GetImage then bAPI4_HMFVVerify is made. According to the GUI at this time, this last swipe is to confirm enrollment was successful. Unlike normal verification, which has been observed to use a two dimensional array of enrolled features, this time ppEnrolledFeatures only contains one array, and consequently, iEnrolledNum contains the value 1. Obviously this last verification is to verify the produced reference template is viable, and can authenticate the user. If unsuccessful, this process is repeated a few more times before enrollment is terminated. If successful, a call to bAPI4_OpenDevice (which calls bAPI4_OpenSensor) is
made after which bAPI4_WriteSecureArea is called, presumably permanently saving the reference template to some hidden partition in the AliceDrive.

Knowing the structure of the enrollment process an enrollment program was created to read fingerprints from the AliceDrive sensor and save pEnrolledFeatures to a file. Since pEnrolledFeatures is a passed in array to the enrollment function, a size had to be chosen. The SDK document states the maximum size of enrolled features is 500, so the array was initialized to 500 bytes. Another output parameter for enrollment, pwEnRetSize is described in the SDK document as the returned enrolled features size. It was our hope that this size
would indicate how many bytes are in pEnrolledFeatures that are actually a part of the reference template and this value would not exceed 500 bytes. The returned pEnrolledFeatures obtained through various enrollments contained a stream of bytes up until a variable position. The value of pwEnRetSize always precisely indicated how long in bytes each template was, and the normal size observed was between 150 and 190 bytes. Every observed template’s first byte was always 9, and its seventh byte was always 0. Since seven bytes were observed in the unsuccessful pEnrolledFeatures (returned if piStatus != SUCCESS), our assumption was these first seven bytes are some sort of header. Observing the stored AliceDrive reference templates passed in to the verification function (ppEnrolledFeatures), showed the starting location of each array begins with 9 and the seventh byte is always 0, validated our assumption that these enrolled features are indeed the same features used for verification (i.e., the reference templates).

Having now obtained the ability to produce reference templates manually, we may now remove the call to bAPI4_GetImage and supply our own images for systematic experimentation with the goal of determining the structure and contents of the AliceDrive reference template data structure.

*Break #1: Bypassing Fingerprint Authentication*

With the verification function’s piStatus parameter having been identified and confirmed as the verification status variable, the problem in deriving a bypass for authentication now was in exploiting this information. IDA Pro makes manipulation during execution of the values in the stack or the register values difficult, so for the initial authentication bypass attempt, a different approach was taken.

The IDA Pro debugger was used to identify the conditional after which piStatus is set to the value 2. A traditional C style conditional (i.e. an if statement) doesn’t exist in assembly language. In assembly, a conditional is often achieved by performing some sort of operation and setting special flag registers. A jump instruction is then used to jump to different locations in the assembly based on these flag registers. For example, a comparison can be achieved by subtracting two values. If the two values are identical, the subtraction will set the zero flag register. A conditional jump based on the zero flag will jump to another place in the assembly code if the zero flag is set, or continue execution after the jump instruction if the flag is not set.
The conditional observed in the AliceDrive is a JZ instruction. A JZ is a jump instruction that is performed if the zero flag is set, much like in the above example. IDA Pro allows for easy manipulation of flag registers during execution, so for the initial bypass attempt, after scanning an invalid fingerprint, execution was paused immediately before this JZ instruction. The zero flag was manually set to 1, and execution was resumed. As a result, the fingerprint was verified and the private drive was mounted. Therefore the device does not provide adequate data security in any way, and even though breaking into the AliceDrive required research and use of reverse engineering tools and techniques, the device was relatively easy to crack. Essentially the AliceDrive provides absolutely no security other than security through the obscurity of deciphering the AliceDrive binaries.

Manually setting the zero flag is not a powerful and permanent solution for an authentication bypass as it requires the user to have some knowledge of assembly language and access to tools that allow the user to alter the execution of the assembly while the program is running. The ultimate solution was to use OllyDbg to permanently change the assembly code for bPTFVVerify. A permanent solution could have been accomplished in any number of ways, such as manually setting piStatus to 2 immediately before bPTFVVerify returns, or even manipulating bAPI4_HMFVVerify in the same manner. Our approach was to simply manipulate the conditional jump to be unconditional, allowing all attempts to verify a fingerprint to succeed. JMP is the assembly language unconditional jump instruction. Using OllyDbg, JZ was changed to JMP, a change of only two characters, thereby completely undermining the security provided through fingerprint authentication. Once this change was saved to PTFVLib.dll, any acceptable quality fingerprint from anyone, valid or invalid, passes authentication. The fingerprint must be an acceptable quality because the manipulated jump instruction occurs after a sanity check has been performed on the fingerprint scan image. If the image fails the sanity check, the jump is apparently never executed, and consequently piStatus is never set to 2.

Analysis of the code revealed that this authentication bypass would not work in all circumstances. If only one reference template is stored on the AliceDrive, a different branch of execution is taken inside bPTFVVerify. As with the above situation, a conditional jump instruction controls whether the piStatus is set to indicate a verified or invalid print. The difference in this situation is that if this jump is performed, piStatus is set to indicate failure. Using OllyDbg, this jump instruction was removed and replaced with two bytes of NOP
instructions. While this forces the execution of the code for valid fingerprint verification attempts, this modification was not successful in bypassing authentication.

In reviewing the differences between the values set for invalid and valid fingerprints, it was discovered that the value for the output parameter piMatchedID was set to 0 for valid fingerprints, and for invalid fingerprints the twos complement for -1 was observed (FFFFFFF). Using the IDA Pro debugger, execution was paused during assignment to this parameter for an invalid fingerprint verification attempt. Manually placing the value 0 into this parameter (by manually setting the register used for this assignment, EAX), resulted in a successful authentication. To provide a permanent bypass, the test instruction used to set the appropriate flags prior to the previously deleted jump instruction uses the EAX register (TEST EAX, EAX). Changing the test instruction which is only used to set CPU flags to a sub (subtraction) instruction ensures the EAX register will contain the value 0 (SUB EAX, EAX subtracts the two operands and stores the result in the first operand), and since this registers contents are used in the assignment of piMatchedID, this parameter will also be 0. By making this final change, it was no longer necessary to replace the jump instruction with NOP instructions. This is because the conditional jump in this case is a JNZ instruction, which jumps only if the zero flag is not set. By subtracting a register with itself just prior to the jump instruction ensures the zero flag is always set. We replaced the previously deleted jump instruction, and simply changed TEST to SUB. Saving this change to the augmented library file allows us to bypass authentication in either circumstance.

Running the AliceDrive controller software by using Autorun.exe on the public read-only drive will always overwrite all modules in the AliceDrive hidden directory. For this augmented library to work, the controller software must be executed by using AliceDrive.exe located in the AliceDrive hidden directory.

Break #2: Deciphering Reference Templates

At this point, we knew that reference templates were used for both verification and enrollment, and that they can be manually pulled out of the verification function dynamically. We had also determined abstractly how reference templates are created and how to create our own, but the details of the reference template format were still unknown. Using the built-in mechanisms to make reference templates that we had discovered, we then systematically created templates to find patterns that will give away the internal structure of the
AliceDrive reference templates.

Keeping with our initial assumption that the AliceDrive uses minutiae for its reference templates, a simple experiment was created. A fingerprint image was used as a reference image. This image, which was in the format of a bit map, was converted into a raw binary stream of the same format observed for the enrollment and verification processes. The image binary stream was then used for enrollment to produce a reference template. The image was then manipulated to produce a multitude of variation images. In each variation image, one feature was changed, which, for most of these images, meant one minutia was removed or one minutia was added; however, an additional variation was created that was the negative of the original. Each variation image was converted to a binary stream and used with the enrollment function to produce a reference template for comparison against the original unaltered image’s reference template. The results of these comparisons provided some evidence of the AliceDrive reference template structure.

The major hurdle in analyzing the reference template comparisons was that the effect of adding or removing minutiae did not produce a uniformly similar result in all reference templates, i.e., removing one minutia or adding a minutia did not result in a consistent decrease or increase in the reference template size for each variation reference template. Also, while it was the case that some reference templates were hardly affected by the variation, some were drastically different. The assumed explanation for these inconsistent variations was that manipulation of the original fingerprint image can result in an adverse affect to the positioning algorithm used by the AliceDrive to standardize positioning for all stored reference templates. This can happen because global fingerprint characteristics (see chapter 2) are often used for fingerprint positioning. Removing or adding features from a fingerprint may change these global features and therefore change the positioning of a fingerprint for reference template creation. If the position changes drastically then characteristics of each minutia, specifically location and orientation, will also change drastically, and it could even result in some minutiae points being lost if positioning leaves some minutiae outside the minutiae extraction area.

The first experiments performed involved removing entire ridges near the periphery of the fingerprint image. Peripheral ridge manipulation should have minimal if not no affect on global characteristics of the fingerprint, but a ridge alone is not a minutia point, however at this point we were uncertain as to whether or not the AliceDrive discriminates between
ridge ending minutiae and ridges ending because the remaining portion of the ridge falls outside the sensor range. The reference templates for these variations were no different from the original image reference template, so it was apparent the AliceDrive does make this discrimination.

Subsequent experiments involved removing ridge endings, not only simple ridge endings, but also independent ridges and spurs (see Figure 2.3 in chapter 2). Comparing between these reference templates and the original reference template produced differences. As mentioned in the analysis of the enrollment process, a seven byte header was observed, where the first byte is always 9 and the seventh byte is always 0. If this area is a header, certain information is expected, namely the number of features (minutiae) stored. Analysis of the remaining bytes of the header for each variation reference template against the original reference template showed that the numerical difference between the sixth byte of the ridge ending removal variations and the original was consistently very small, a difference of two or three at most. For a difference of one, the size of the reference template data structure was off by six bytes, if the value was off by two, the size of the reference template was off by twelve bytes, and for a difference of three the size was off by eighteen bytes. The assumption was made that the sixth byte indeed represented the number of minutiae present in the data structure, and that each minutiae required a total of six bytes to represent. Observing the templates with the smallest difference in the sixth byte, where the sixth byte only differed by one, showed that when compared against the original reference template a contiguous block of five bytes somewhere in the data structure was completely missing, and a separate byte near the end of the data structure was also missing.

The next experiments involved adding bifurcations by extending ridge endings to intersect with an adjacent ridge (using the GNU image manipulation program). Since this involved changing only a preexisting minutia into another type, the sixth byte was expected to remain consistent to the original reference template. This was observed, further validating our assumption of the sixth byte. Like the ridge removal experiments, bifurcation creation produced varying reference template differences, however the simplest difference observed was a block of five contiguous bytes and a byte near the end of the data structure, all other bytes were consistent with the original template.

The experimental results indicated that the abstract reference template structure contained a header, a body, and a trailing footer section. The body contains X number of
blocks of five bytes where X is the number observed in the sixth byte of the header. If our assumption about the header is correct, the bytes of the body should begin at byte eight. Since it was observed that each five block byte section has a separate but related byte near the end of the data structure, following the body section should be X number of bytes, one for each minutiae (see Figure 3.7). Analyzing the data structure in this manner showed that this kind of division fits with our observed reference templates, and additional testing revealed that each byte in the footer section is ordered in accordance to how the five byte blocks in the body section are ordered. For example, if the sixth five byte block in the body section is removed, then the sixth byte starting from our defined footer section would also be missing.

Researching popular reference template data structures did turn up some structures that have defined header, body and footer sections, such as BIO API [2], however the size and values for the header were not consistent with our findings and even if they were, the exact formatting of each minutiae in the body section is still proprietary. There are many proposed data structures for minutiae in which they are represented with six bytes but these differ not only with the characteristics stored for minutiae but also the number of bits used for each characteristic. No such schemes were apparently consistent with the observed AliceDrive reference template structure.

In an attempt to derive whether the minutiae type is stored, an additional experiment was performed using the reference template of the negative image of the original. A fingerprint negative will contain the same number of minutiae in roughly the same locations, but the types will be inverted, i.e., bifurcations become ridge endings and ridge endings become bifurcations. Comparing the negative reference template with the original did not yield any findings and did not definitively point to any part of the data structure that would store the minutiae type. If the type of each minutia is stored, it could still be verified by using the verification function with the original image, but using the negative reference template for comparison. If verification fails, it is likely minutiae type is stored in the reference template. Performing this verification attempt was successful, possibly indicating minutiae types are not stored in the template, however it could be the nature of the matching algorithm. It seems unlikely that type would not be stored, since it is easy to store (within one or two bits) and is rather important for distinguishing fingerprints, that is, if this information is not present, the distinguishing characteristics stored by the reference template are dimin-
ished. Therefore, it could be within the error tolerance of the verification algorithm to allow validation of a negative fingerprint with a non-negative template, or vice versa, since all other data is almost identical.

Through static analysis of the reference templates created for this experiment, essentially only the generalized structure depicted in Figure 3.7 was now known. To gain any more details of the reference template structure, a more detailed analysis of how reference templates are created was necessary.

![Figure 3.7: Abstract Reference Template Structure](image)

The parameter for enrollment `pEnrolledFeatures` will contain a reference template once enrollment succeeds. Following this parameter during enrollment should reveal how data is stored in it, and consequently, the breakdown of minutiae characteristics in the template. In one unnamed subroutine called during enrollment execution and traced through static analysis, `pEnrolledFeatures` is passed in as a parameter, and values are stored in it. After the header sections, blocks of data of the following sizes are placed into the `pEnrolledFeatures` data structure:

- 11 bits
- 11 bits
- 2 bits
- 8 bits
At a later point in the code, a final byte is placed in the structure as well (presumably the sixth byte in the assumed footer). Minutiae are typically represented by location (a Cartesian coordinate pair), orientation (an angle), and a type (bifurcation, ridge ending, or other). Location is typically of a high importance, so we made the assumption that the first 22 bits stored represent 11 bits for an X value and 11 bits for a Y value. Additionally, since minutiae type can be represented in one or two bits, the smallest observed block of 2 bits was assumed to represent type. Orientation was assumed to be represented by the following 8 bits after the supposed type bits. This leaves the final byte of the contiguous five byte block and the trailing sixth byte as unknown.

From each block of five bytes located in a selected reference template, the first 22 bits were captured, the first 11 bits were considered X coordinates, and the next 11 bits were considered Y coordinates. Initial mapping of these locations on the image that produced the reference template did not line up, no matter how the mapping or the original image was scaled, rotated, or positioned. Reviewing the mapping more carefully, and trying to find a orientation that worked, the mapping does line up if the image is rotated 180 degrees and flipped. As seen in Figure 3.8 this mapping falls on almost all minutiae in the image therefore we considered our location values verified and proceeded to prove our assumption of the remaining bytes in the reference template data structure.

Before attempting to map the minutiae type and orientation onto the image, the orientation value had to be deciphered. Orientation is an angle, and an angle is represented
usually as an integer [0-360). One byte can only store up to 256 integers (0-255). An angle can be made to fit into a single byte by dividing the angle by a transform value, and the byte representation can be converted back by multiplying by transform value. The transform value in this case would be 360/255 or approximately 1.4118.

Also before mapping, two angle characteristics must be determined, the starting angle (angle 0) and the direction in which angles increase (clockwise or counterclockwise). An angle is analogous to a hand on a clock. The hands move in a particular direction, and point to a particular location. It is not sufficient to define an angle simply by an axis, but rather the direction the angle points must also be specified. Before mapping of the reference template angles, it was assumed the starting position is the X axis pointing to the right (due east), and angles increase counterclockwise (this is in accordance with angle representation in the standard [17]).

For type, only two values were observed in the supposed type value for any reference template, the values 1 and 2. A circle was mapped to the value 1 and a diamond to the value 2. The complete mapping of location, type and orientation to the rotated and flipped image is seen in Figure 3.9.

![Figure 3.9: Location, Type, and Orientation Mapping](image)

Further mappings with other reference templates to their respective images had similar results. The types and location were obviously correct. Orientation was not exactly perfect, but storing an angle in a single byte does result in a loss of precision. With respect to ridge endings, angles always point back toward the terminating ridge where as with bifurcations angles point toward the valley between the bifurcation. In the standard [17], angles are
always determined for ridge endings, therefore for bifurcations, the image negative is taken (thereby causing bifurcations to become ridge endings), and the angle is derived from the resulting ridge ending. The AliceDrive apparently adopts this strategy as well.

Our mapping experiments had previously only been attempted on one fingerprint image. To further validate our format findings, we used an additional fingerprint. The resulting reference template mapping produced for this image and its reference template was consistent with the above findings.

Although for each minutiae stored in the AliceDrive reference templates there are two additional bytes that are still unknown, having deciphered this much of the reference template we have shown that not only is the reference template data structure insecure but we believe that based on [19] this is enough information to conceivably reconstruct the original fingerprints for any acquired reference template.

Summary

Through our experimentation and observation we have shown that the AliceDrive fails to protect the two critical security areas of fingerprint-secured data, the private data itself and the fingerprint reference templates. Not only are the private data and the reference templates stored in the clear, but no obfuscation techniques are used whatsoever to deter reverse engineering. Through analysis of the verification function we were able to bypass authentication to gain access to the private drive, and to manually pull out all reference templates contained in the AliceDrive used for verification.

Furthermore, and probably more importantly, we were also able to derive a partial format of the reference template data structure. The only minutiae characteristics we were looking for were location, orientation, and type, and yet while we did decipher these characteristics from the reference template, the template still stores an additional two bytes per each minutia. Based on the work in [19] we considered location, orientation, and type enough information to theoretically derive an original fingerprint.

Further Research

Although we believe we have deciphered enough information from the AliceDrive reference templates to conceivably reconstruct a fingerprint, there are two addition bytes
unaccounted for in the reference template data structure. We have made some assumptions that perhaps one of the remaining bytes represents quality such that only the highest quality minutiae are used for authentication and the lowest quality minutiae would only be used if there were no matches among the higher quality minutiae. We are unsure as to whether this assumption is valid or not, and even if it is, this still leaves one byte completely unaccounted for, for which we do not have any assumptions at all. Further research is needed to ascertain the purpose of these two bytes.

Additionally, we have not provided any mechanism for obtaining reference templates automatically. Automatic extraction of reference templates, as with our automatic authentication bypass is more powerful since it does not require any knowledge or special tools. During our research we attempted to find a way to extract them without having to manually copy and paste the binary from IDA Pro, but none of our efforts proved fruitful. The function bAPI4_ReadSecureArea in the AliceDrive SDK module plays a part in retrieving these templates, but does not retrieve them directly. Somehow the data retrieved by this function is converted by AliceDrive.exe into the reference templates. The function that is responsible for this conversion is not publicly accessible, and since it is apparent this function uses many global variables, simply extracting the assembly to create our own function was too time consuming for this research.

As stated in the introduction, the hardware was considered a black box, but the internal physical structure of the device is of some interest. The exact location of where all the system data (the user profiles, libraries, etc.) is stored is still unknown. Fully understanding the hardware may make it possible to somehow mount this drive as well as the private drive without any use of the AliceDrive software.

Finally, our authentication bypass provided in this chapter is not necessary the best solution, as it still requires a fingerprint, and the fingerprint has to pass an image sanity check. Through our observations, the function bMediaChange2Flash in the AliceDrive SDK, once called, mounts the private drive. Some attempt was made, after our provided bypass had been created, to call this function directly in order to mount the private drive obviating the need to provide any fingerprint for authentication, but all attempts failed. Since we had a bypass at this time, not much effort was placed in pursuing this route; however, this bypass could prove much more powerful, as it would work even if the scanner no longer functions.
In the previous chapter, the AliceDrive was scrutinized from a security perspective. The results of this analysis indicate that the two facets of fingerprint biometric data security (the private data and the reference templates) are unprotected and in clear text on the device. In addition, the reference template format of the AliceDrive was partially derived revealing significant information about the fingerprints they represent. In order for the AliceDrive to provide adequate fingerprint-based security, the private data must be encrypted using current cryptographic mechanisms (e.g., AES) in which the key is derived from the user’s fingerprint, and any required reference templates cannot be recoverable by unauthorized individuals. A function to produce an AES key from a fingerprint is trivial, the problem is producing consistent keys. Due to intra-class variation some sort of reference data must be made public for consistent key generation using the same fingerprint. As discussed in the previous chapter, storing any fingerprint reference data in clear text undermines the security of the scheme and may allow for the reconstruction of the original fingerprint. This would seem to be an irreconcilable paradox, as the reference data must be stored in the clear but should not be accessible by unauthorized individuals. As discussed in chapter 2, fuzzy extractor schemes have been suggested for various biometrics, including fingerprints, which theoretically solve this issue. In this chapter we adopt a fuzzy extractor mechanism, specifically fuzzy vault (see chapter 2), for a proposed academic and unimplemented AliceDrive fuzzy vault.

The provided fuzzy vault adaptation is based on the fingerprint-based fuzzy vault of [16], to be further referred to as FFV. The purpose of the proposed implementation of this chapter is to illustrate the issues and possible parameters of a fingerprint-based fuzzy vault for the AliceDrive. Although we do make suggestions for improvement of FFV, our primary goal is to demonstrate that a practical improvement to the AliceDrive may be possible, and provide a starting place for future research. This proposal is limited in that no implementation is made, nor is the scheme heavily scrutinized.

Before citing the difference of our proposed adaptation of FFV, a general overview of
the original FFV scheme is described. Following this, issues and concerns for the proposed AliceDrive fuzzy vault are discussed.

**Original Scheme Overview**

As described in chapter 2, the scheme proposed in FFV is itself based on the fuzzy vault scheme of [11, 12]. FFV is fingerprint-based fuzzy vault in which minutiae characteristics of location and orientation are used for vault encoding. Minutiae location and orientation are converted to binary string representations which are then concatenated to form a string of 16 bits, X, for each minutiae. Each X is then encoded into a vault using a polynomial, P. The result, P(X), is a value in a Galois finite field of $2^{16}$ (denoted GF($2^{16}$)). The coordinate (X, P(X)) for each minutiae is stored in the vault along with many chaff points that do not lie on P. The encoding polynomial secures a secret, K, in its coefficients. The entire secret is encoded over n 16 bit coefficients where n is the degree of the encoding polynomial. FFV differs from the first proposed fuzzy vault of [11, 12] in that set intersection error correction is used to account for intra-class variation as opposed to Reed-Solomon (set difference) error correction. Set intersection works by attempting to derive the original polynomial for every permutation of T elements in the unlocking set with Lagrange’s polynomial interpolation, where T represents the Lagrange polynomial decoding threshold (n+1). As such, in order to determine a matching polynomial, FFV appends to K a 16 bit cyclic redundancy check (CRC) code generated for the secret. This CRC code is appended in the remaining coefficient of the encoding polynomial. For every polynomial generated during decoding, the CRC code is extracted and used to validate the remaining coefficients. If the CRC finds no error, there is a high probability the original polynomial has been decoded.

The nature of fuzzy vault does account for some intra-class variation through selection/filtering of vault elements, but only minor differences are corrected. Alignment is still required to correct the vast majority of differences between the encoding print and subsequent prints. During vault encoding, alignment helper data is extracted based on global fingerprint characteristics, specifically areas of high curvature. The helper data must be stored publicly, and consequently cannot reveal any information about the original fingerprint or K. According to FFV, areas of high curvature do not reveal sufficient information to reconstruct the fingerprint or the fingerprint orientation field, and therefore may be made public.
FFV also stipulates that only minutiae that are above a specified quality threshold may be used for vault encoding or unlocking. Furthermore, only minutiae that are considered well separated from other minutiae may be used for encoding or unlocking. The former ensures that a key is not generated from specious minutiae and the latter ensures each minutiae point is assigned a unique value when encoded into the finite field.

An abstract overview of both the encoding and decoding procedures for FFV is described below.

FFV Vault Encoding Procedure:

1. Minutiae are extracted from a template image. A quality index is applied to each extracted minutia. In addition, alignment helper data is also extracted.

2. Well separated and high quality minutiae are selected for encoding, called the locking set.

3. Chaff minutiae are created by generating random location and orientation values. A chaff point is selected for encoding if the value generated is unique, and well separated from all other previously selected chaff and genuine minutiae.

4. The locking set and chaff minutiae selected for encoding are represented as bit strings for location (a Cartesian coordinate pair) and orientation (an angle). The strings of location and orientation are then concatenated to form 16 bit values, X, for vault encoding.

5. The degree of the polynomial is selected, n, and a secret, K, of 16n bits is generated. Appended to this secret is a 16 bit CRC code for K. This creates a new secret, K’ of 16(n+1) bits.

6. K’ is encoded into a polynomial, P, of degree n by dividing K’ into 16 bit sections, and considering them coefficients of P.

7. Each X derived from the locking set is then evaluated using P to form vault X and Y coordinates (X, P(X)). The remaining chaff X values are given random Y values such that Y does not lie on P.

8. The vault is randomly reordered. The vault and the helper data are stored publicly.
FFV Vault Decoding Procedure:

1. Minutiae are extracted from a query image. A quality index is applied to each extracted minutia and alignment helper data is also extracted.

2. An alignment algorithm is used to align the query minutiae set. Alignment is accomplished using the query alignment helper data and the stored alignment helper data for the template image.

3. Well separated and high quality minutiae from the query image are selected.

4. The vault abscissa values are broken into binary strings of location and orientation.

5. The location and orientation values in the vault are used with the query minutiae location and orientation values to first coarsely filter chaff points, then a minutia matching algorithm is used to select vault values from the remaining unfiltered vault elements. The coarse filter marks vault values as chaff if the minimum distance between the vault minutiae and the query minutiae is greater than some threshold.

6. For each matching vault abscissa with the query minutiae, the vault coordinate is added to the unlocking set.

7. Every subset of $n+1$ from the unlocking set is used with the Lagrange interpolation formula to recover a polynomial, $P'$.

8. For every recovered $P'$, the CRC code is extracted to validate remaining coefficients. If an error is detected, the secret has not been recovered. If no error is detected, there is a high probability $P' = P$, and consequently the original secret has been recovered.

The experiments performed on the implemented FFV scheme are primarily concerned with the rates at which valid fingerprints unlock the vault (genuine accept rates, GAR), invalid fingerprints unlock the vault (false accept rates, FAR), and failure of fingerprints to meet the minimum number minutiae requirements for either unlocking or locking the vault (failure-to-capture rates, FTCR). The minimum number of minutiae required for both unlocking and locking, $r$, is a requirement on the number of genuine points in the vault. Setting $r$ across all fingerprints results in higher FTCR. To account for this, FFV allows $r$ to be set individually for each fingerprint however the value must be within a set range.
Experiments were performed on two different fingerprint databases using variable polynomial degrees, genuine points in the vault and number of chaff points. The total number of points in the vault as well as minimum distance parameters remained consistent among experiments using the same fingerprint database. Although FFV claims to provide improvements to GAR, FAR, and FTCR when compared to their earlier attempt for this scheme [21] the security and efficiency details are missing for each parameter set used. They claim the decoding mean time was 8 seconds, and that the mean number of candidate secrets is 33, but the details of how they arrived at these figures are absent. Similarly, they claim that for a polynomial of degree 8, and a vault containing 24 genuine and 200 chaff points, that the expected number of combinations that need to be evaluated to brute force the vault is 2.5x10^9, corresponding to a 13 year computation time. Again, the details as to how this number of combinations computes to 13 years is not present.

While security is not scrutinized in FFV, they do address a few security concerns. Besides the relatively weak security of their claimed 13 years brute force computation time, they also address issues involving the alignment helper data. While the alignment helper data itself does not allow for reconstruction of the fingerprint minutiae or an orientation field, it may allow a clever attacker to filter chaff points from the vault based on minutiae orientation. The other concern addressed is their assumption of uniform spatial distribution of minutiae, as related to their stipulation of a minimum spatial minutiae threshold. This assumption is not correct, as usually minutiae are clustered. They claim that this assumption may allow an attacker to use statistical minutiae distribution models to filter chaff points.

*Proposed Fuzzy Vault Implementation*

A pure adaptation of the FFV scheme for the AliceDrive would require simply increasing the bit string sizes for each minutiae used for encoding from 16 bits, to match the size of the AliceDrive location and orientation data, 30 bits (although since this does not lie on a byte boundary 32 bits is more tractable). We, however, consider a pure adaptation to be undesirable for many reasons.

One undesirable characteristic noted of FFV is the possible filtering of chaff points from the vault using the alignment helper data. FFV suggests mitigating this problem by adding a sufficient number of chaff points whose orientation is derived from alignment
helper data. Since this effectively standardizes the orientation of all minutiae in the vault, the orientation data may not provide that much distinctiveness. We propose removing orientation information from each minutia and to mitigate the loss of distinctiveness for each minutia, we also propose adding minutiae type. The total number of bits required for this information for the AliceDrive is 24 bits (22 for location and 2 for type).

Another security concern noted in FFV is the assumption that the minutiae spatial distribution is naturally uniform. This may allow an attacker to filter the vault of chaff using statistical minutiae distribution models. Their reasoning for requiring well separated minutiae is to assure that unique values are assigned when they are encoded into the finite field, but we do not see the necessity for this stipulation. Increasing the Galois field from GF(2^{16}) to GF(2^{24}) assures that every X value encoded into the vault can have a unique Y value. We therefore propose that this spatial separation stipulation be removed for both encoding and decoding minutiae and that the finite field be increased to 24 bits (GF(2^{24})).

The degree of the polynomial determines the size of the secret secured by the vault. With a finite field GF(2^{24}) each coefficient may contain 24 bits. An AES 128 key cannot be evenly distributed into 24 bit coefficients. A scheme could be devised to spread this key over these coefficients; however we recommend the use of an AES 192 key which is evenly divisible by 24, obviating the need for any special operations to store the key. A polynomial of degree 8 could conceal an AES 192 key as well as a 24 bit CRC code. The threshold to decode would then be 9 genuine vault points. As mentioned in chapter 2, 12-15 minutiae are generally accepted for high probability fingerprint matching. As such, we propose that the polynomial degree be set at a minimum of 11 (requiring 12 genuine vault points to unlock the vault), leaving more coefficients than is necessary for encoding the AES 192 key and the 24 bit CRC code. We propose that 8 24 bit coefficients be reserved for AES key, and one 24 bit coefficient be used as a CRC code. The remaining coefficients should be chosen randomly from the finite field universe. Since the CRC is only computed on the AES key, the entire polynomial does not have to be recovered during vault decoding.

Concerning the size of the vault, specifically the number of chaff points to generate, we take a paranoid approach. Given the current worlds fastest computer, with a computational capacity at 1.759 PFLOPS we assume the worst case scenario, that each CPU cycle equals one vault decoding attempt. The number of vault decoding attempts possible in a century is then approximately 5.55x10^{24}. We propose that the total number of invalid permutations
of the vault unlocking threshold in the vault be at least $5.55 \times 10^{24}$. For example, given a decoding threshold of 12 and 24 genuine points in the vault, if we choose 2078 chaff points, the total number of possible combinations is $\binom{2102}{12} \approx 1.5 \times 10^{31}$ and of these combinations $\binom{24}{12} \approx 2.7 \times 10^6$ will unlock the vault and recover the secret. The probability that a combination of points unlocks the secret is $(2.7 \times 10^6)/(1.5 \times 10^{31}) \approx 1.8 \times 10^{-25}$ and the expected number of combinations needed to be evaluated is $(1.5 \times 10^{31})/(2.7 \times 10^6) \approx 5.56 \times 10^{24}$.

Finally, we propose that the vault need not be randomly organized, but instead ordered according to the vault abscissa values. We do not see the security concerns requiring random organization, and it may in fact be a hindrance on efficiency during vault decoding. Also, we propose that along with the vault and the alignment helper data, the number of genuine points in the vault, $r$, be made public such that users know the maximum number of points that need to be selected for vault decoding. Without this information, a user could select any number of vault points, which could greatly impact the efficiency of the scheme. All other details of the scheme are described in FFV, and those that are not, we leave up to the implementer.

**Issues and Concerns**

Increasing the number of chaff points in the vault also increases the vault’s security, but coupled with the lack of a spatial minimum distance threshold for genuine points and chaff, may have an adverse affect on its efficiency, making the scheme practically useless. As the number of chaff increases, it is more likely chaff minutiae will be placed spatially close to a genuine minutiae and have the same type. If this occurs it is conceivable that filtering these chaff points may be very difficult, resulting in their frequent selection for the unlocking set. If this occurs for many genuine points, it may result in extreme difficulty in decoding the vault, rendering it ineffective. The exact nature of what constitutes spatially close minutiae such that it results in the above scenario is dependent on the alignment and filtering algorithms of FFV, which we consider a black box. More research is necessary to first see if this problem poses a serious concern, and if it does, determine a possible solution.

Of course the primary issue with the proposed scheme is the fact that it is unimplemented and untested. Questions still remain as to how the parameters of the scheme will affect FAR, GAR, and FTCR rates, as well as the overall efficiency and any possible loopholes to
undermine the computational security the scheme provides.
Chapter 5

Conclusion

Fingerprint-secured USB drives are increasingly more available for purchase recently. This thesis studies the security of a representative fingerprint-secured USB drive called AliceDrive.

In chapter 3, through systematic analysis and testing of the AliceDrive, a verification bypass was created to access the private drive without a valid fingerprint. Furthermore, we were able to recover stored fingerprint reference templates on the device, and through some experimentation, most of the format of these templates was deciphered, conceivably allowing an attacker to recover the original fingerprint.

In addition, we have provided an academic and unimplemented security improvement based on the fuzzy vault scheme in chapter 4. While this scheme is limited in its impact since it is not implemented, this suggestion does provide an idea of the kind of issues and steps required for realistic fingerprint-based fuzzy vault implementations and a basis for future research.

To our knowledge, this is the only formal security analysis or formal documentation of any kind of a device such as the AliceDrive. Our contribution is officially bringing to the attention of the public the possible vulnerabilities of any fingerprint-secured USB drive and the current state of commercial fingerprint-based data security in relation to current fingerprint-based biometric security mechanisms.

Areas of Further Research

In chapter 3, future work is suggested for alternative and more powerful vulnerability exploits as well as further analysis of the AliceDrive, including more in-depth investigation into both the AliceDrive hardware and reference template data structure. Fully implementing and testing a fuzzy vault implementation or any fuzzy extractor mechanism, whether that suggested in chapter 4 or some other realization, specifically for a commercial drive such as the AliceDrive is of a high interest for future work as well. Such an implementation would bridge the gap between the theoretical and commercial worlds.


